data_tructures(&algorithms, lecture05)

Doan Trung Tung, PhD – University of Greenwich (Vietnam)

Plan

01

Binary Search Tree

Delete notes

02

AVL Tree

Balance tree, rotation operations

03

Heap Sort

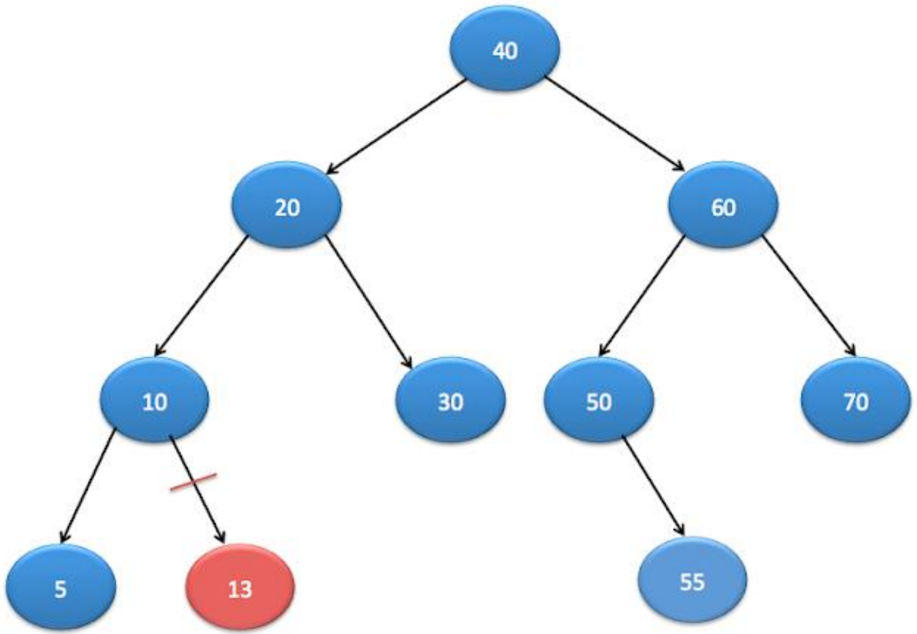
Construct a Heap, sort with Heap structure

Binary Search

Delete a Node

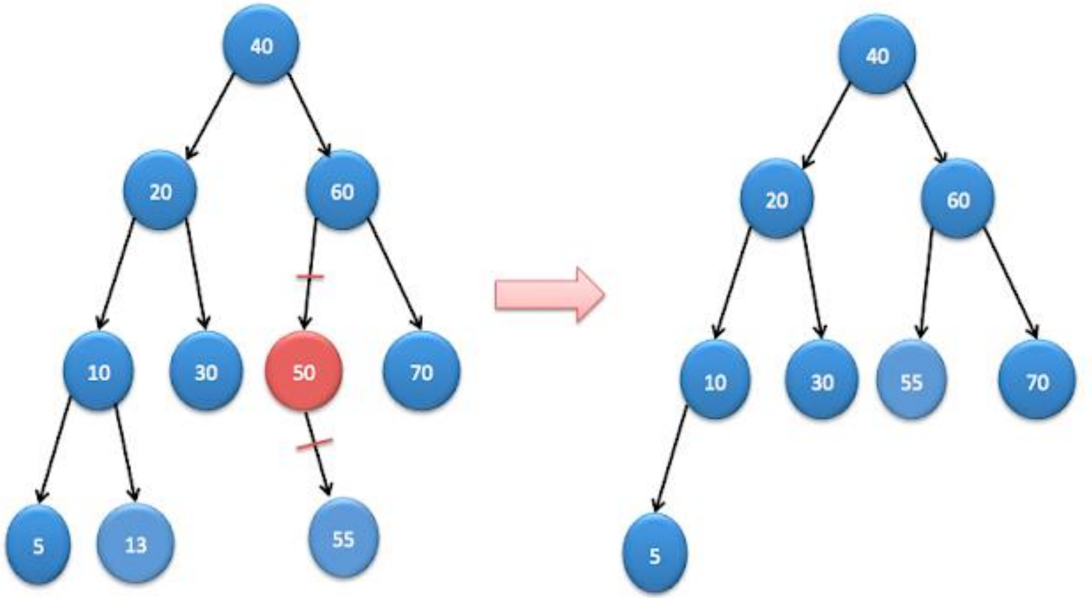
Delete A Node in BST

- If the node is a leaf, it can be deleted immediately.



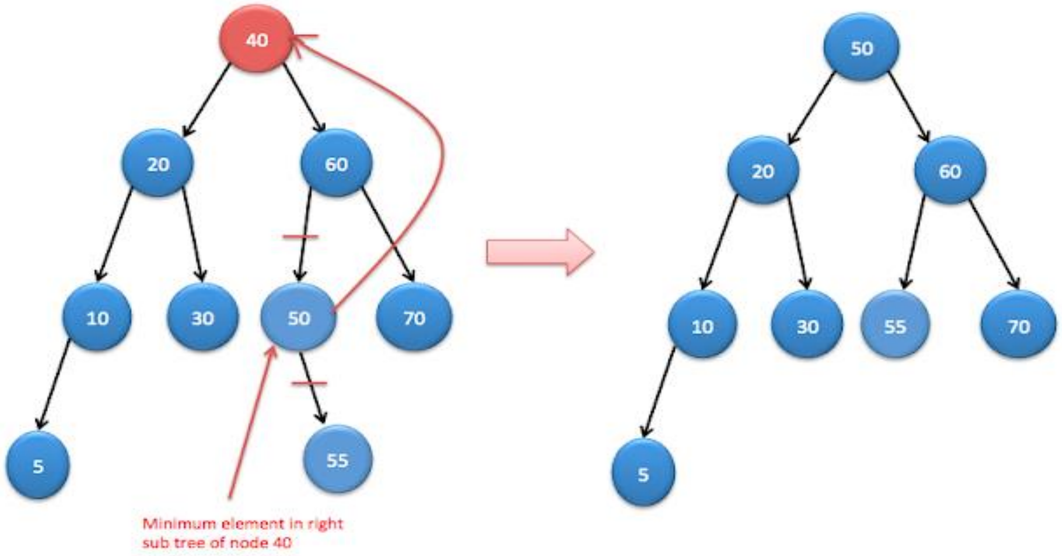
Delete A Node in BST

- If the node has one child, the node can be deleted after its parent adjusts a pointer to bypass the node



Delete A Node in BST

- If the node has two children, the general strategy is to replace the key of this node with the smallest key of the right subtree and then recursively delete it.



Delete A Node in BST


```
node* delete_tree_node(node* root, const int key)
{
    if (root == NULL) return NULL;

    if (root->key < key)
        root->right = delete_tree_node(root->right, key);
    else if (root->key > key)
        root->left = delete_tree_node(root->left, key);
    else
    {
        if (root->left && root->right)
        {
            // find min on the right
            // swap key between root vs min
            // delete the old key on the right
        }
        else if (root->left)  // move root to the left, remove old root
        else if (root->right) // move root to the right, remove old root
        else // remove root;
    }
    return root;
}
```

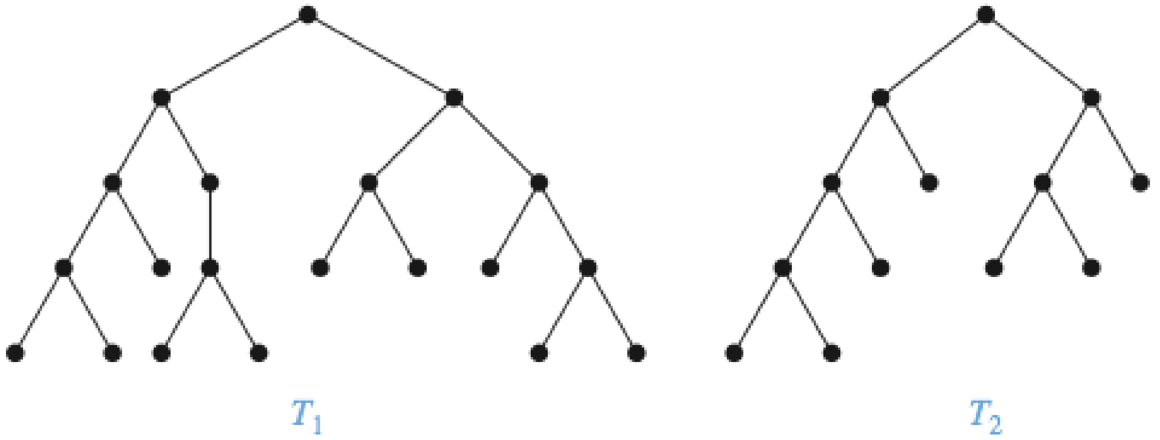
Complexity: O(h)

AVL Tree

Balance tree, rotation operations, complexity

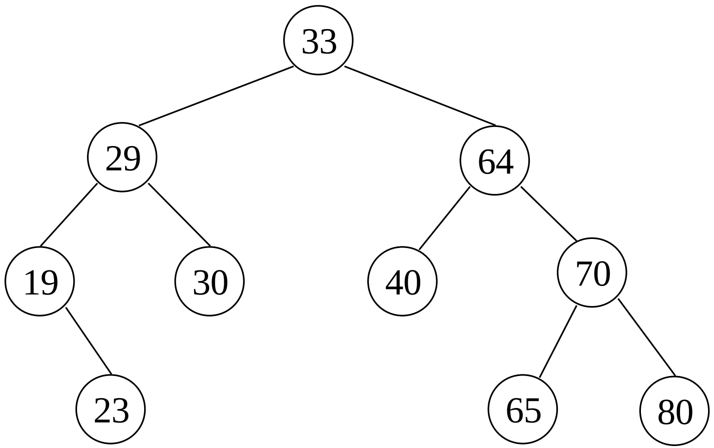
AVL Tree

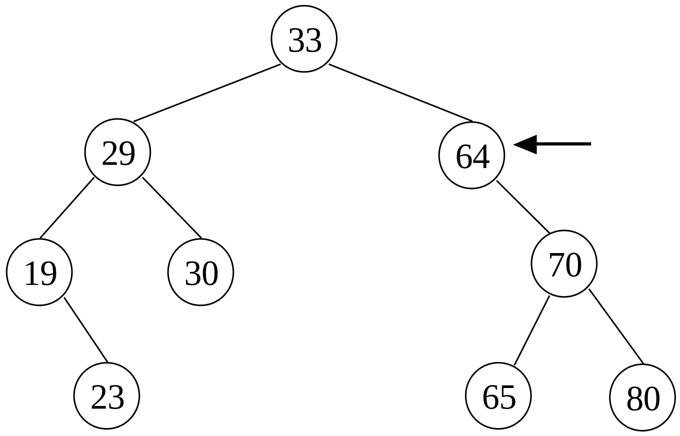
- An AVL (Adelson-Velskii and Landis) tree is a binary search tree with a balance condition.



AVL Tree

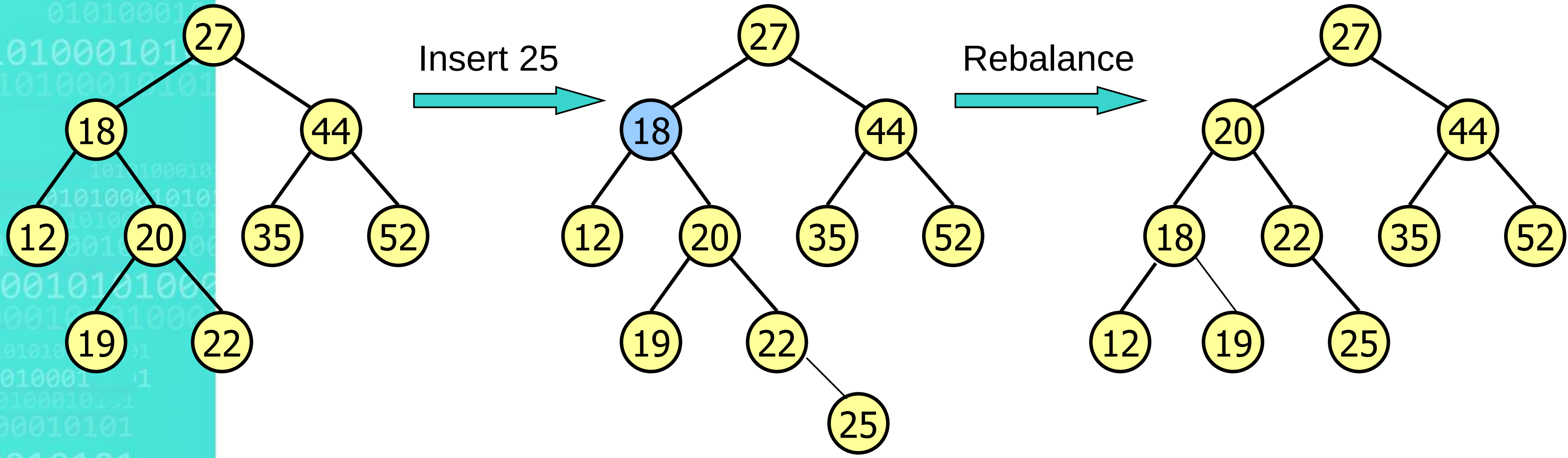
- An AVL tree is identical to a binary search tree, except that for every node in the tree, the height of the left and right subtrees can differ by at most 1
- All the tree operations can be performed in O(log n) time, except possibly insertion





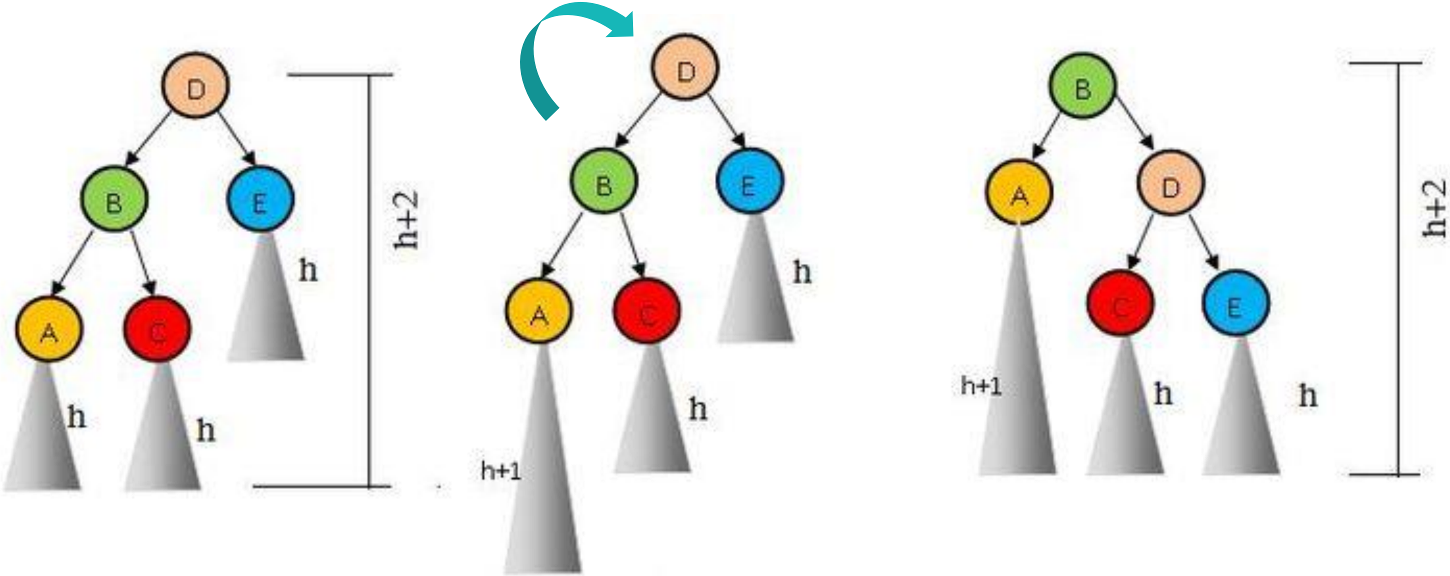
Insert A Node To AVL Tree

- Insert a node to AVL Tree is similar as in BST
- But it can violate the balance of AVL Tree, so it needs more work to re-balance again



Rotation on AVL Tree

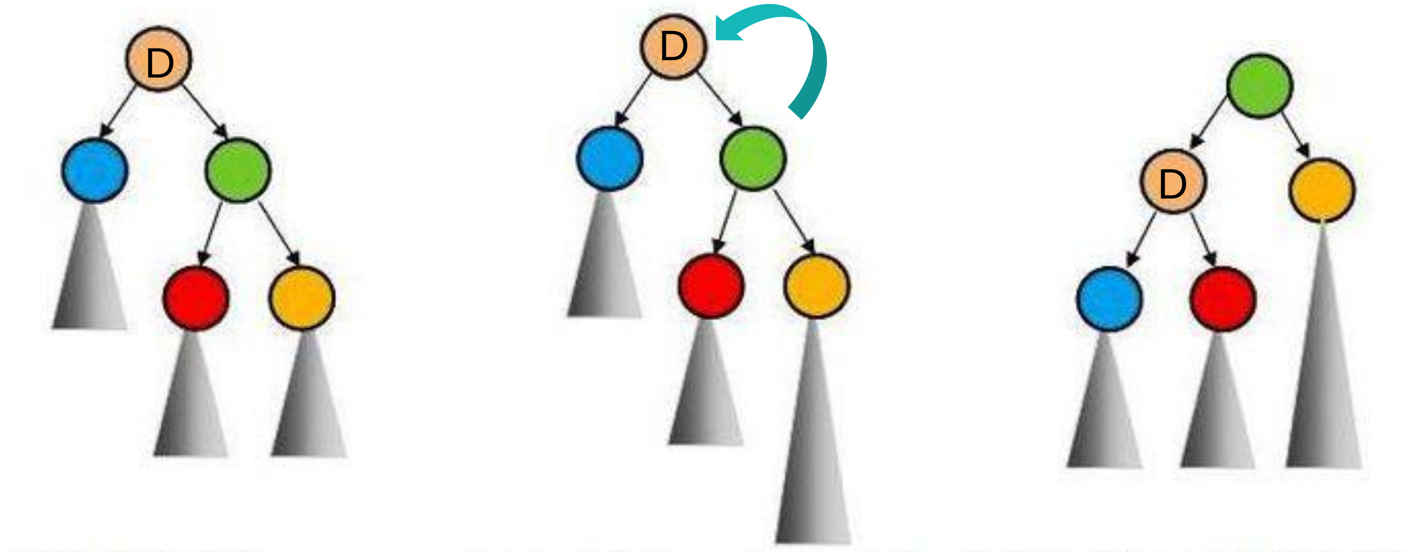
- Re-balance is done by rotation
- Case 1: Right rotation



Insert node on left subtree of D => Lost balance in right subtree => Right rotation at D

Rotation on AVL Tree

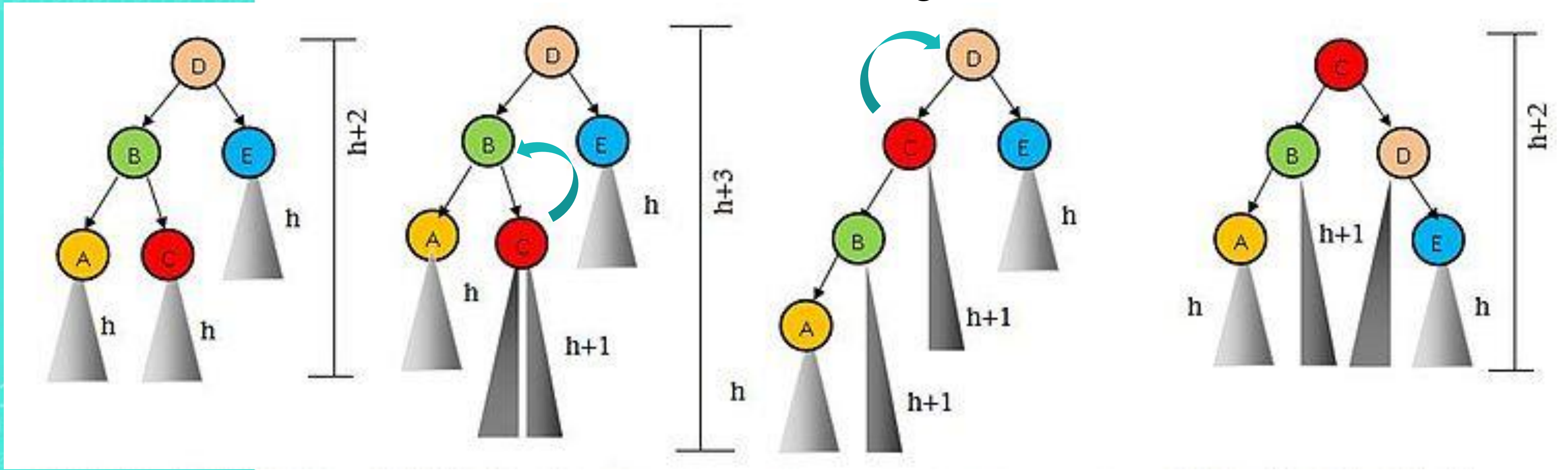
- Re-balance is done by rotation
- Case 2: Left rotation



Insert node on right subtree of D => lost balance in left subtree => Left rotation at D

Rotation on AVL Tree

- Re-balance is done by rotation
- Case 3: Left rotation then right rotation



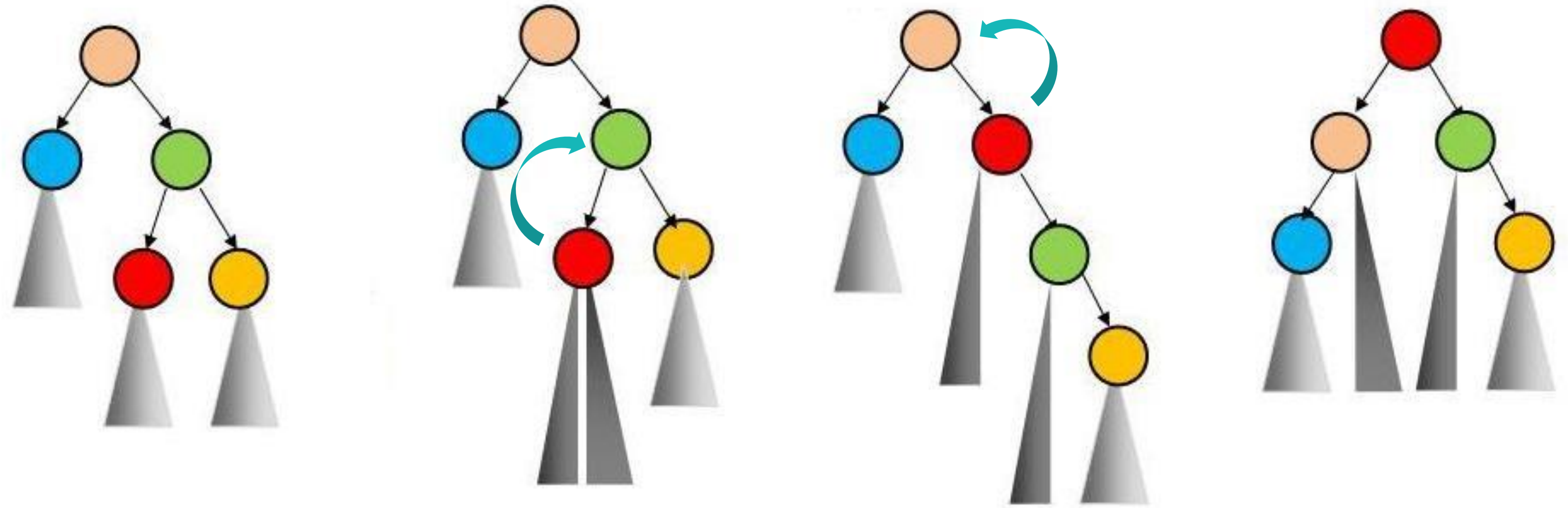
Insert node on left subtree of D
Lost balance in right subtree

Left rotation at left of D

Right rotation at D

Rotation on AVL Tree

- Re-balance is done by rotation
- Case 4: Right rotation then left rotation



Insert node on right subtree of D
Lost balance in left subtree

Right rotation at right of D

Left rotation at D

Helpers For Rotation

- Recalculate height of a subtree

```
int get_height(node *n)
{
    if (n == NULL) return 0;

    int lh = left_height(n);
    int rh = right_height(n);

    return max(lh, rh);
}
```

```
int left_height(node *n)
{
    if (n->left == NULL) return 0;
    else return 1 + n->left->height;
}
int right_height(node *n)
{
    if (n->right == NULL) return 0;
    else return 1 + n->right->height;
}
```

Helpers For Rotation

- Calculate balance factor

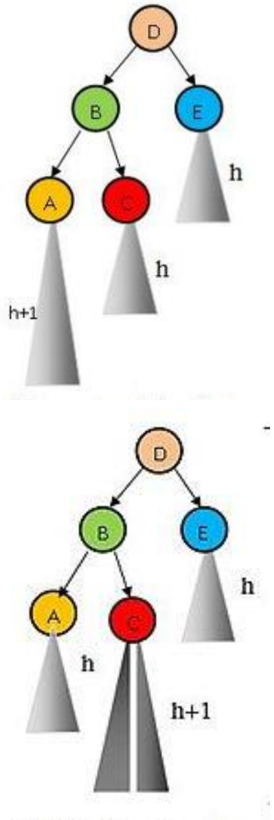
```
int balance_factor(node* n)
{
    if (n == NULL) return 0;

    int lh = left_height(n);
    int rh = right_height(n);

    return lh - rh;
}
```

Insert Node


```
Insert_Node(root, key)
    If empty tree then return a new node

    If key need to insert to right subtree
        Insert (recursively) key to right subtree;
        Rebalance on left subtree
    Else // key need to insert to left subtree
        Insert (recursively) key to right subtree;
        Rebalance on right subtree

    Update height of root

    Return root
```

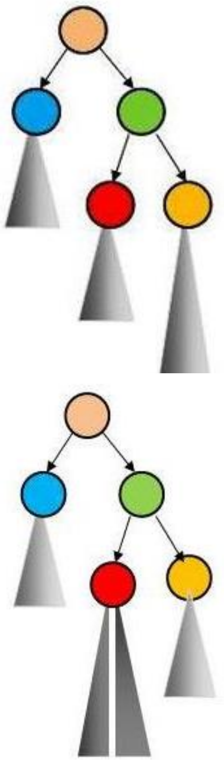
Rebalance On Right Subtree

```
If left of root >> right of root
    n = left of root
    If left of n > right of n
        Right rotation at root
    Else
        Left-right rotation at root
return root;
```



Rebalance On Left Subtree

```
If right of root >> left of root
    n = right of root
    If left of n < right of n
        Left rotation at root
    Else
        Right-left rotation at root
return root;
```

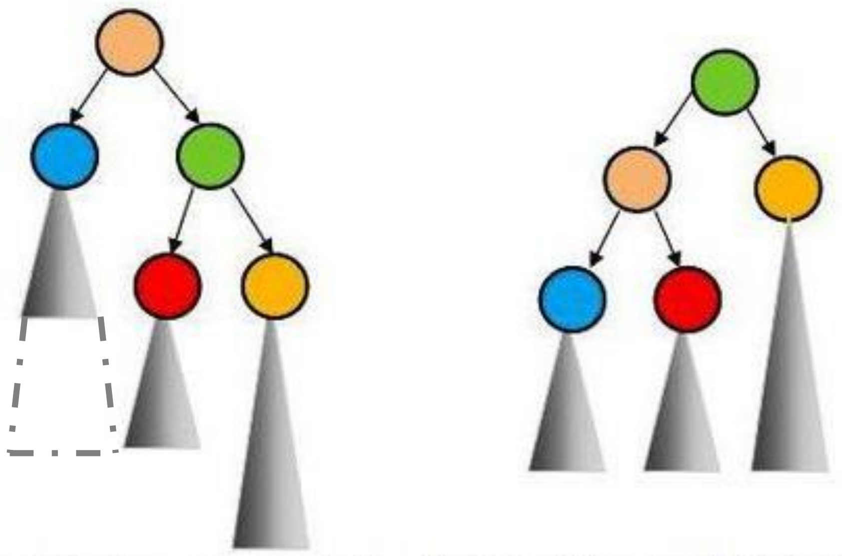


Example

- Create ALV Tree from the following numbers
- 5, 3, 6, 4, 1, 7, 9, 8, 10, 12, -10, -5, -3, 2, 11

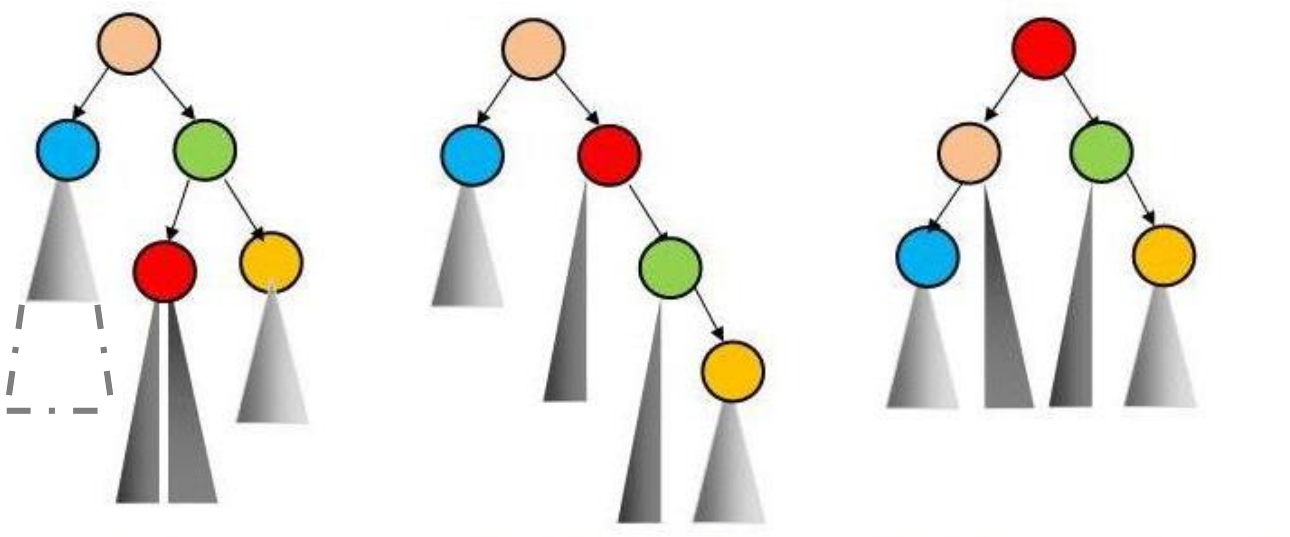
Delete A Node

- Deleted node is on left subtree
- Lost balance on left sub tree: left rotation



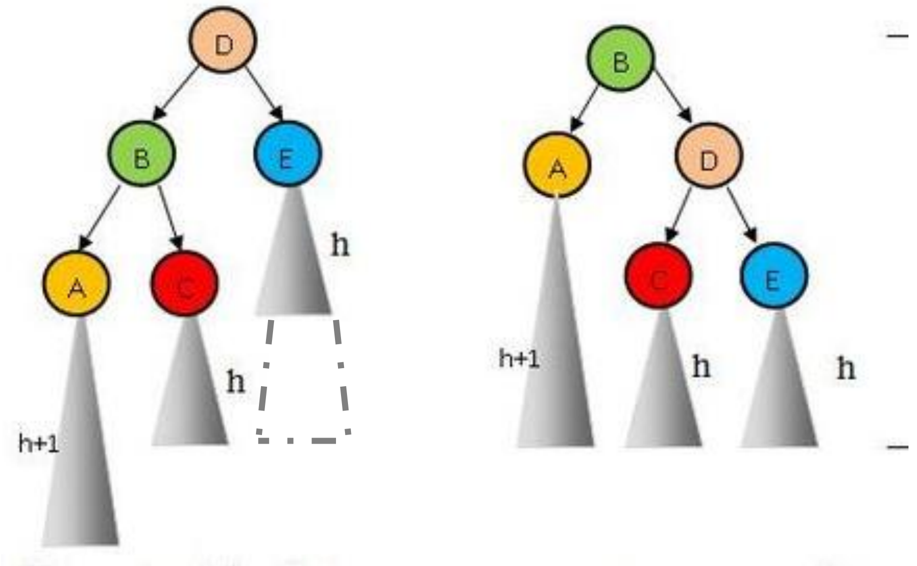
Delete A Node

- Deleted node is on left subtree
- Lost balance on left sub tree: right-left rotation



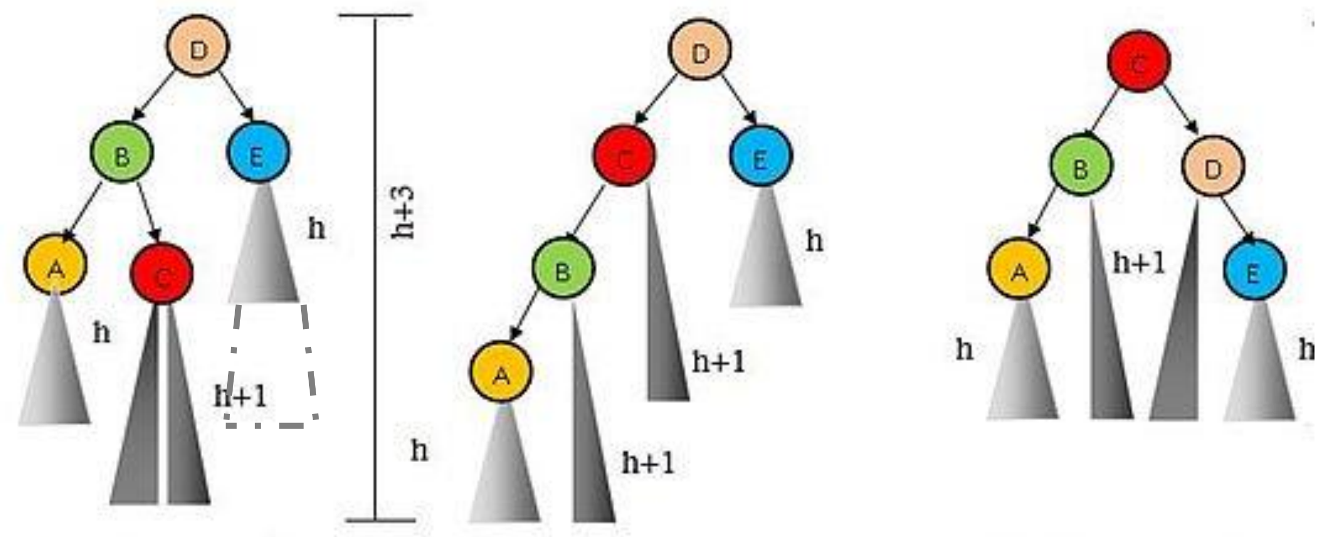
Delete A Node

- Deleted node is on right subtree
- Lost balance on right sub tree: right rotation



Delete A Node

- Deleted node is on right subtree
- Lost balance on right sub tree: left-right rotation



Delete A Node

```
If empty tree then return NULL;

If key is on right subtree
    Delete key recursively on right subtree
    Rebalance on right subtree
Else if key is on left subtree
    Delete key recursively on left subtree
    Rebalance on left subtree
Else
    If there is right subtree
        Find left-most node of right subtree
        Swap key with root
        Delete new key recursively on right subtree
        Rebalance right subtree
    Else
        Move root to left node

Update root height
Return root
```

Example

- Delete following nodes in order:
- 2, 4, 3, -10, 6, 8, 7, 12

AVL Tree Complexity

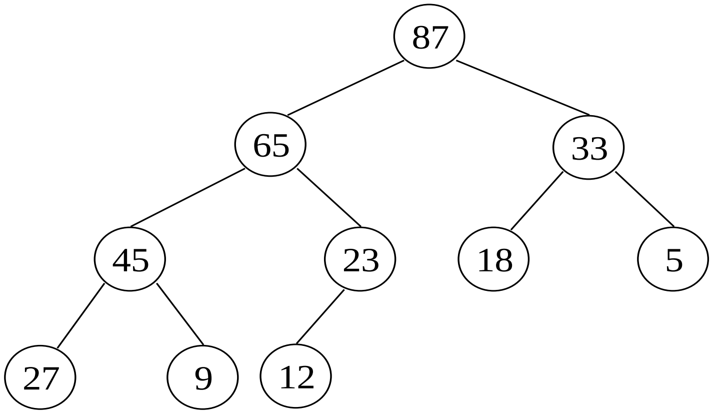
- Complexity:
  - Build tree: O(nlogn)
  - Rotation: O(1)
  - Rebalance: O(1)
  - Insert a node: O(logn)
  - Delete a node: O(logn)
  - Search for a key: O(logn)
- AVL Tree is used when
  - There are few insertion and deletion operations
  - Short search time is needed

Heap Sort

Heap data structure, build a heap, sort by heap

Tree Representation As Array

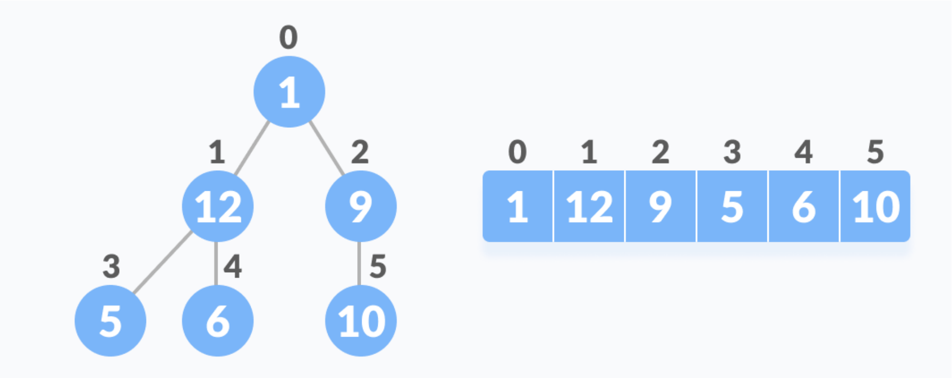
- A complete binary tree is a binary tree whose all levels except the last level are completely filled and all the leaves in the last level are all to the left side.



Tree Representation As Array

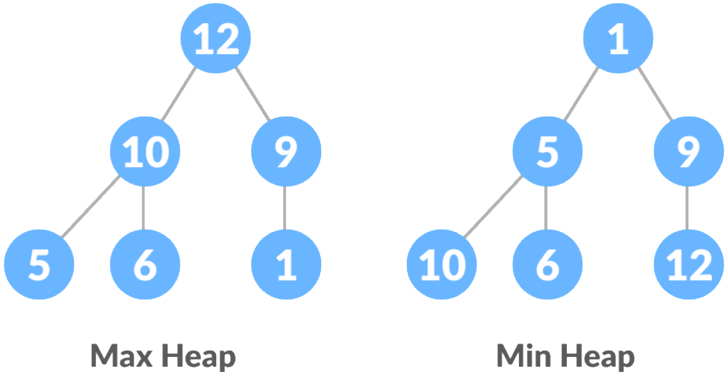
- A complete binary tree can be represented by array
- Node at index [i] will have:
  - left child at [2i + 1]
  - right child at [2i + 2]
  - parent at [(i – 1) / 2]

12 [1]
- left 5 [3]
- right 6 [4]
- parent [0]



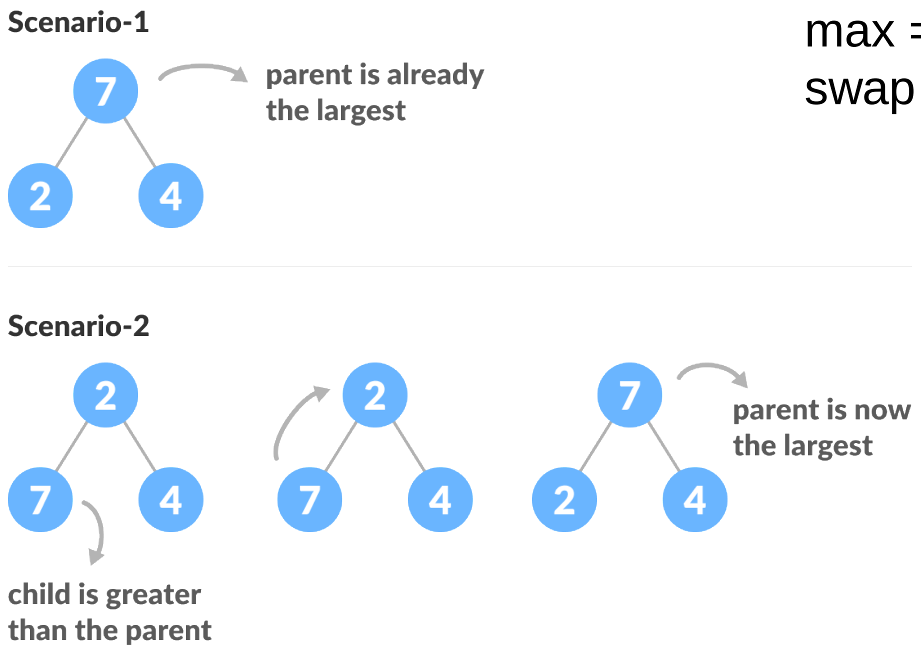
Heap Data Structure

- Heap is a complete binary tree
- Key of root is greater / smaller than all keys of its children
- All subtrees are heap



Heapify A Tree

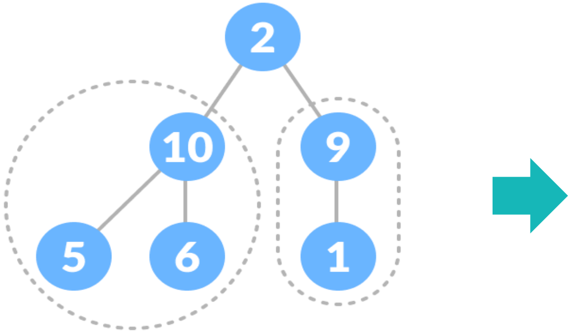
- Heapify a small tree

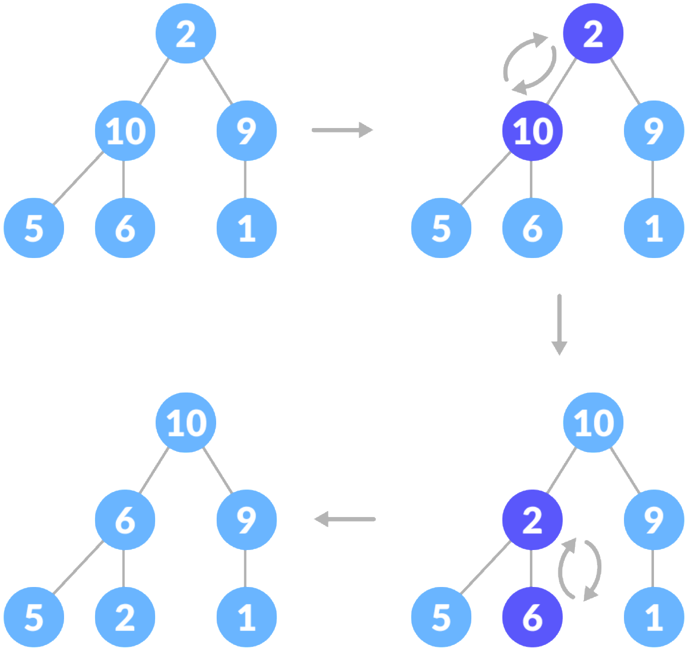


max = max(left, right)
swap root vs max if needed

Heapify A Tree

- Heapify recursively: suppose left / right subtrees are already heaps





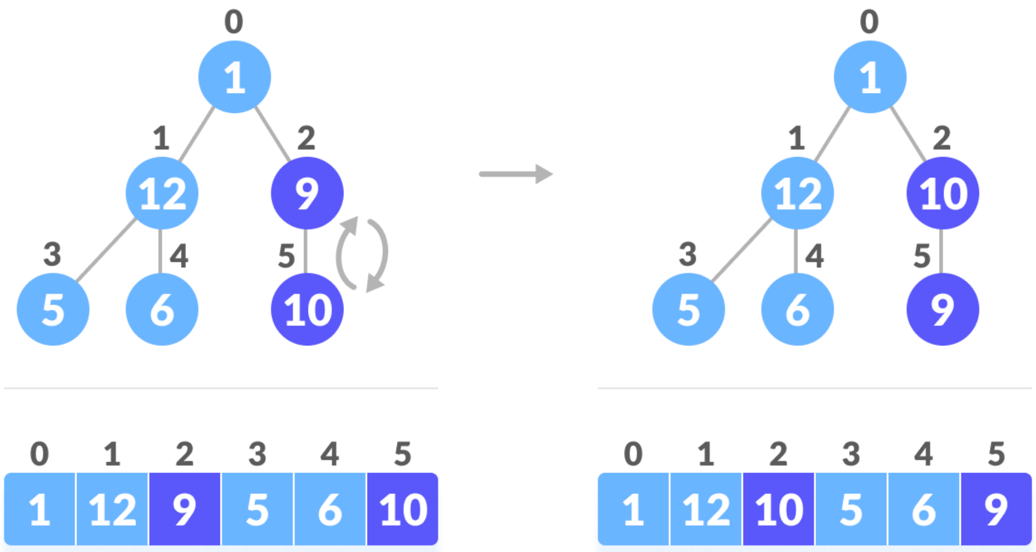
  - Find max (left / right)
  - Swap root vs max if needed
  - Heapify on swapped subtree

Heapify A Tree

- Heapify bottom-up


```
for (int i = n / 2 - 1; i >= 0; i--)
    heapify(arr, n, i);
```

i = 2

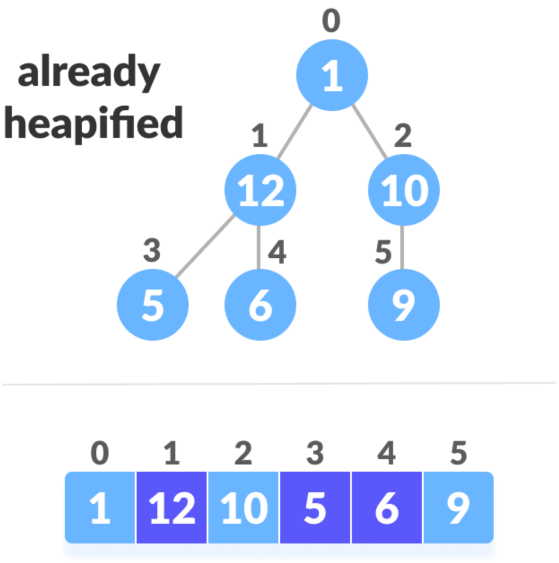


Heapify A Tree

❖ Heapify bottom-up

```
for (int i = n / 2 - 1; i >= 0; i--)
    heapify(arr, n, i);
```

i = 1

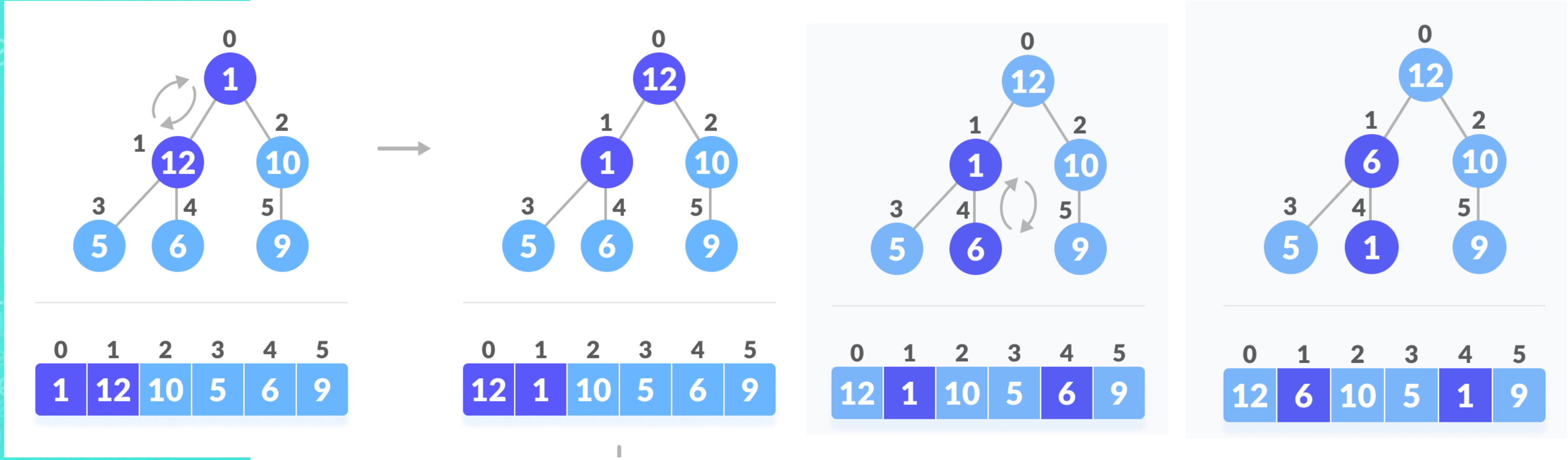


Heapify A Tree

- Heapify bottom-up

```
for (int i = n / 2 - 1; i >= 0; i--)
    heapify(arr, n, i);
```

i = 0



Heap Sort Using Heap

- Since the tree satisfies Max-Heap property, then the largest item is stored at the root node.
- Swap: Remove the root element and put at the end of the array (nth position) Put the last item of the tree (heap) at the root place.
- Remove: Reduce the size of the heap by 1.
- Heapify: Heapify the root element again so that we have the highest element at root.
- The process is repeated until all the items of the list are sorted.

Heap Sort Using Heap

- Heapify complexity: O(logn)
- HeapSort complexity: O(nlogn)

```
for (int i = n / 2 - 1; i >= 0; i--)
    heapify(arr, n, i);

// Heap sort
for (int i = n - 1; i >= 0; i--)
{
    swap(&arr[0], &arr[i]);
    heapify(arr, i, 0);
}
```