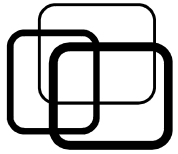
AVL Tree

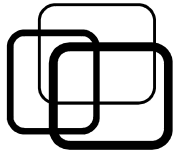
Inst. Nguyễn Minh Huy

Contents



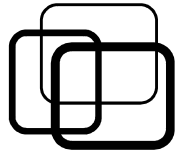
- BS Tree analysis.
- Tree rotation.
- Implementation.

Contents



- **BS Tree analysis.**
- Tree rotation.
- Implementation.

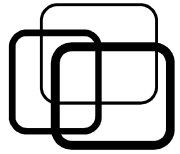
BS Tree analysis



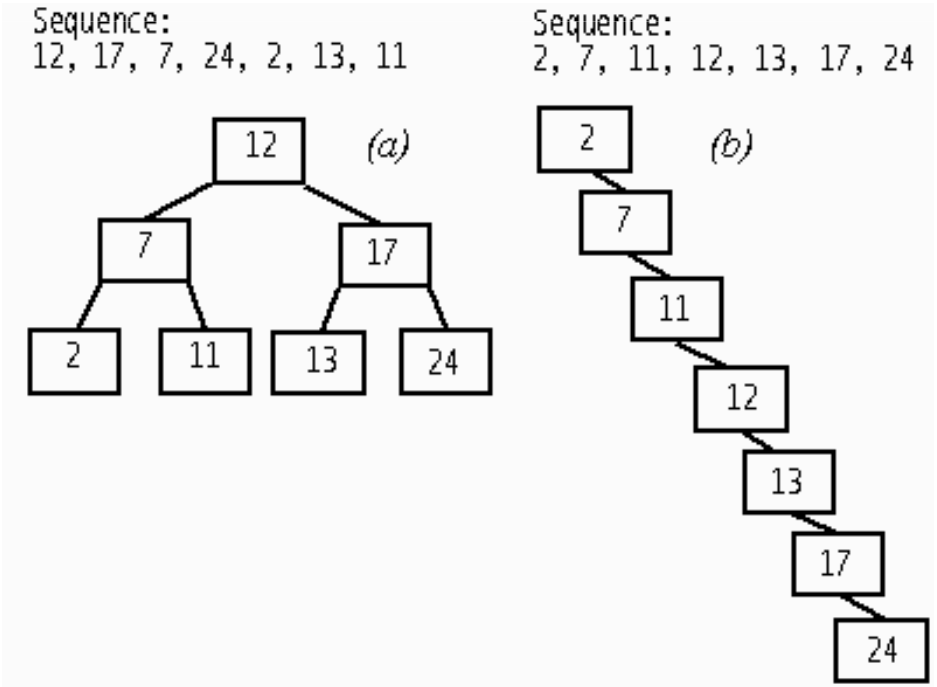
- Binary search tree vs. Singly linked list:

	Binary search tree	**Singly linked List**
Memory storage	Non-contiguous	Non-contiguous
Memory cost	**O(N)**	**O(N)**
Resize	**O(1)**	**O(1)**
Find element	**O(logN) → O(N)**	**O(N)**
Add element	**O(logN) → O(N)**	**O(1)**
Remove element	**O(logN) → O(N)**	**O(1)**

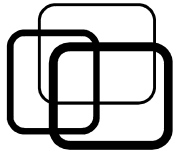
BS Tree analysis



- De-generated tree:
  - Every node has at most 1 child.
  - Binary tree → Linked list.
    - Tree height ~ n.
    - Performance: O(n), same as linked list.



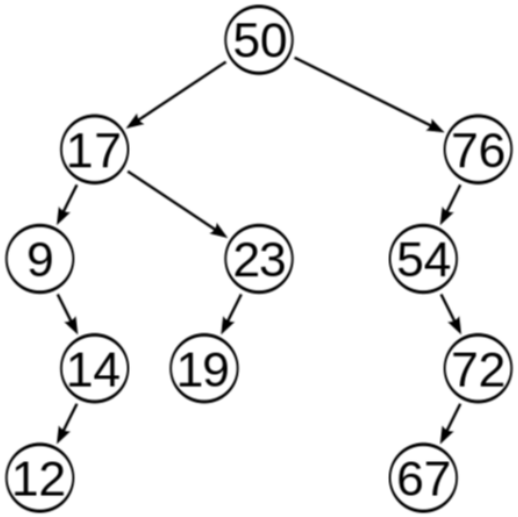
BS Tree analysis



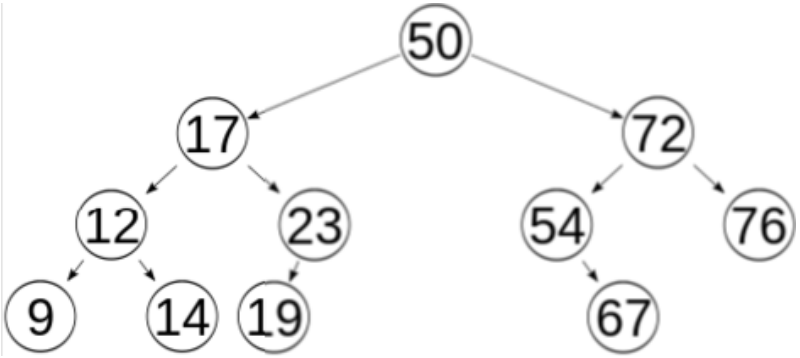
- Balanced tree:
  - For every node:
    - Left height and right height vary slightly.
    - | left height - right height | <= 1.
  - Tree height ~ log(n).

➔ More efficient in performance.

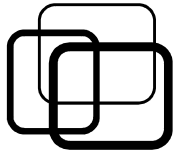
Un-balanced Tree



Balanced Tree

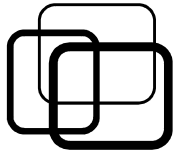


Contents



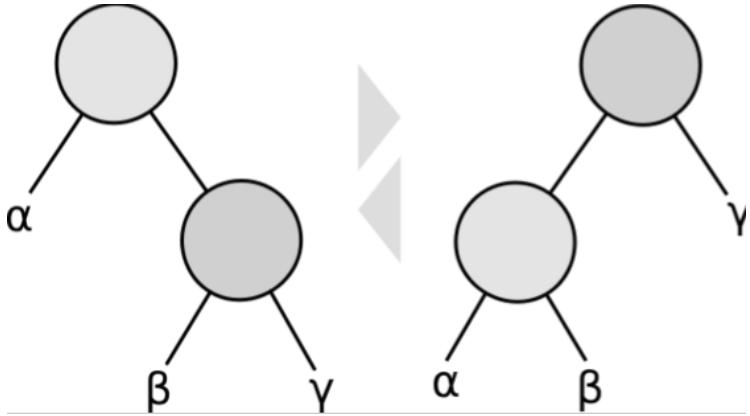
- BS Tree analysis.
- **Tree rotation.**
- Implementation.

Tree rotation



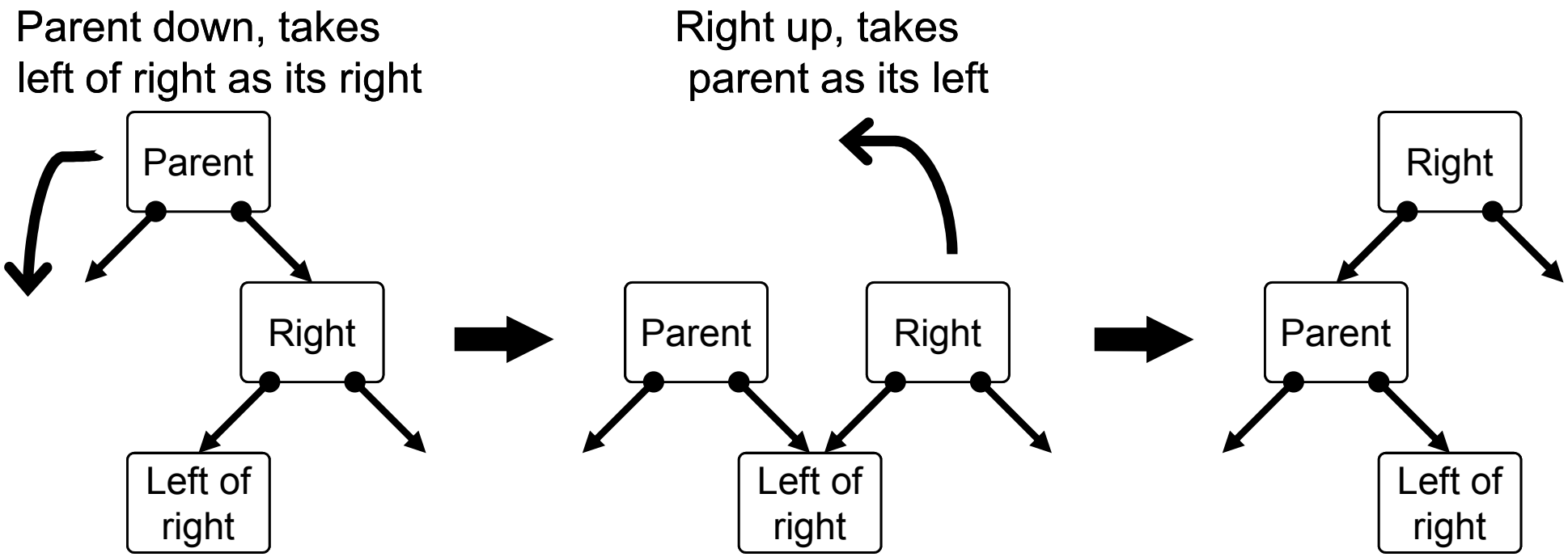
- Concepts:
  - Change structure of tree (keep tree condition).
  - Rotate around a node:
    - Move node down and its child up.
    - Parent → Child.
    - Child → parent.
  - Used to decrease tree height.

  → Make tree more balanced.



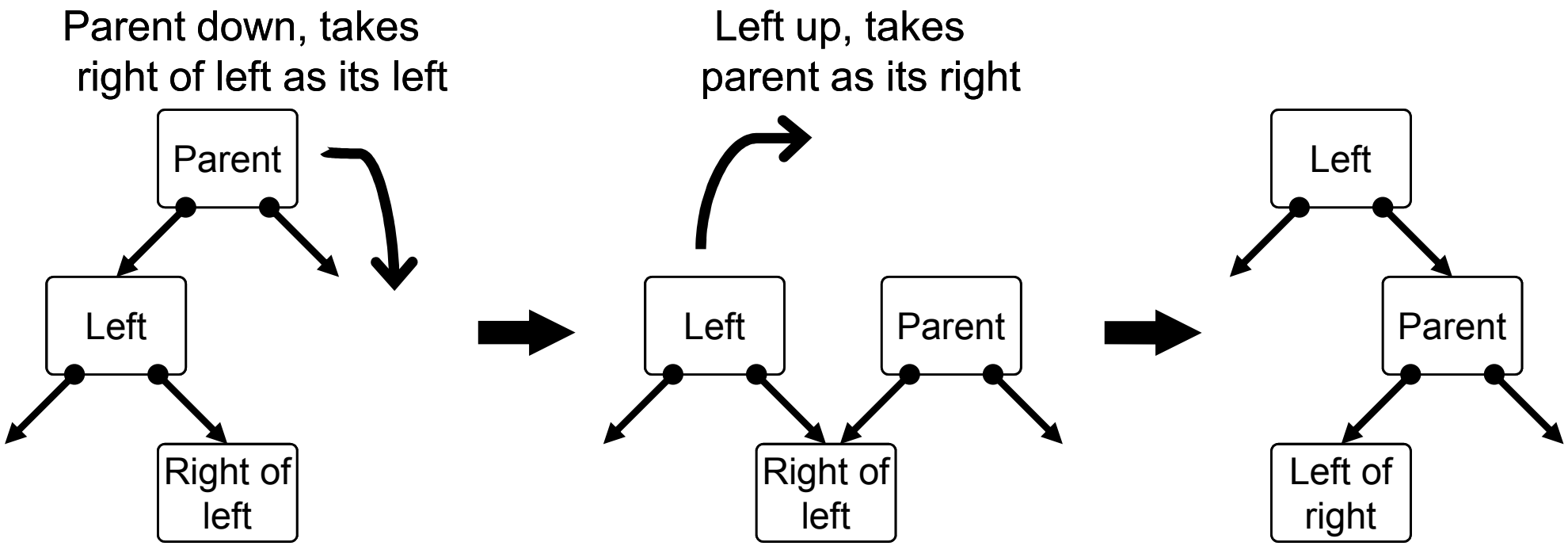
Tree rotation

- Rotate left (at parent):

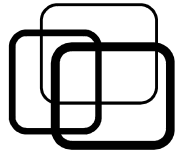


Tree rotation

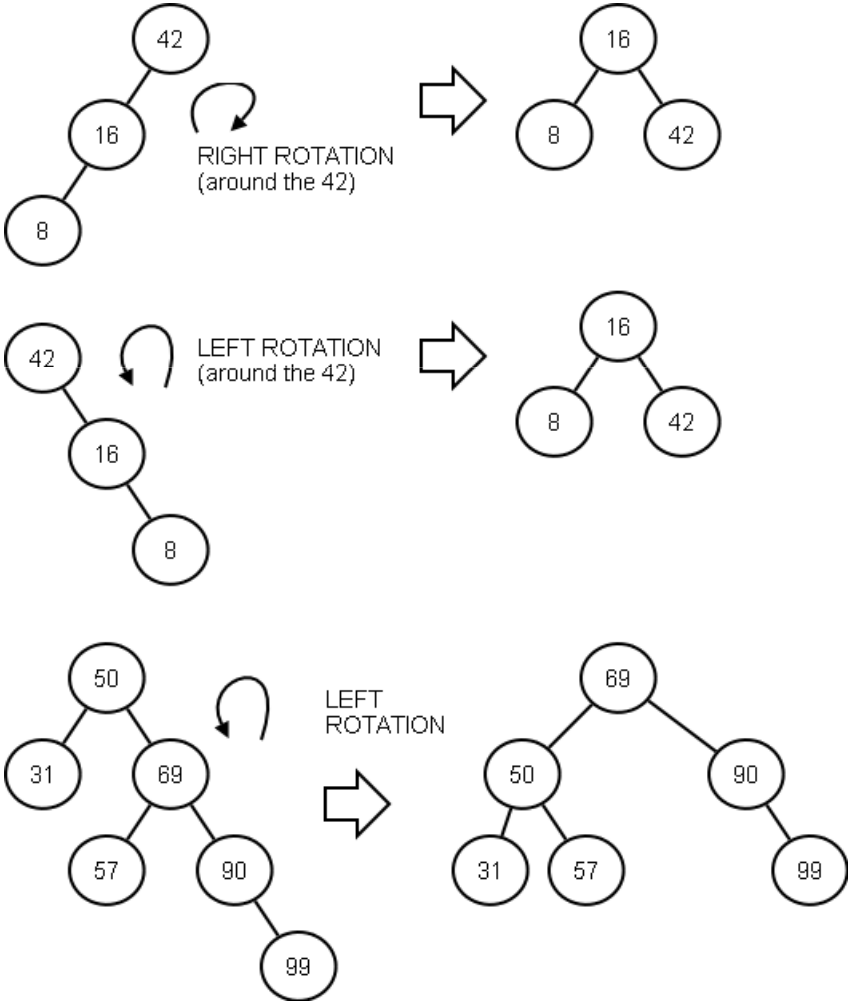
- Rotate right (at parent):



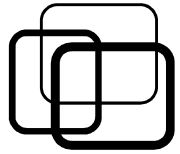
Tree rotation



- Example:

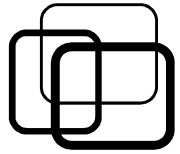


Contents



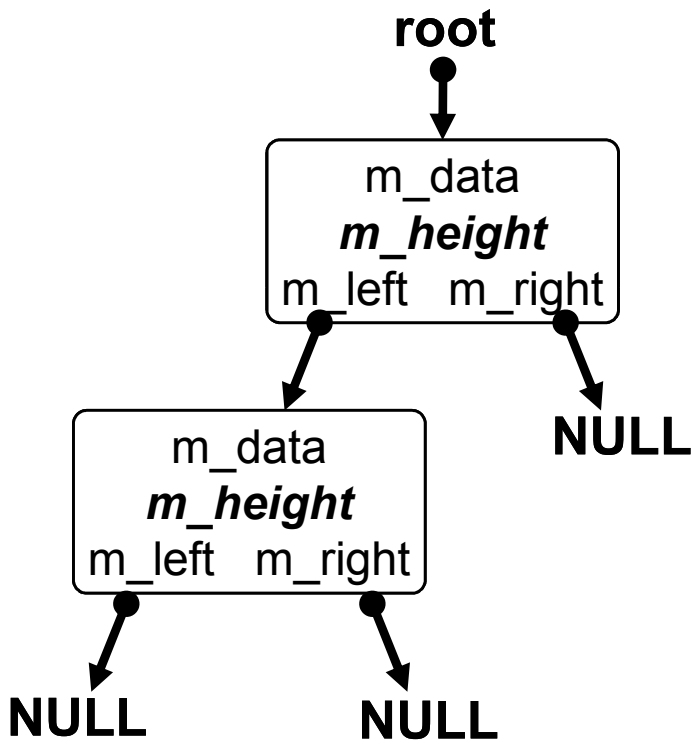
- BS Tree analysis.
- Tree rotation.
- **Implementation.**

Implementation

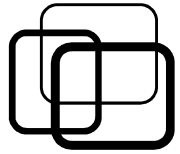


- ADT AVL Tree:
  - Values:

```
struct AVLNode {
        int         m_data;
        TreeNode   *m_left;
        TreeNode   *m_right;
        int         m_height;
};

class AVLTree {
private:
    TreeNode* m_root;
};
```

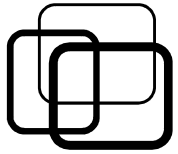


Implementation



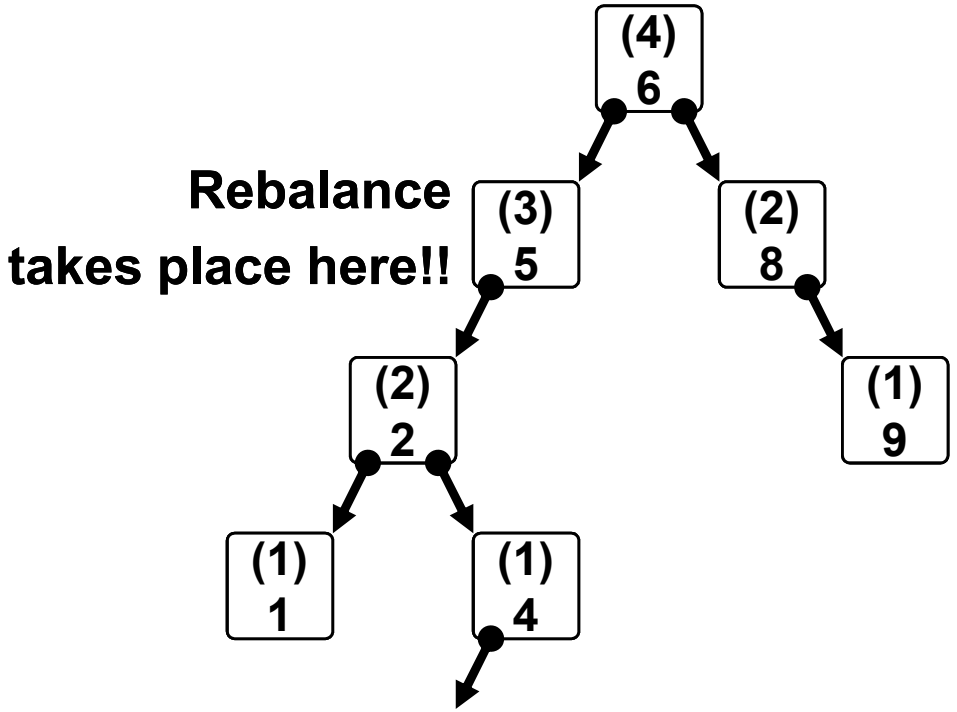
- ADT AVL Tree:
  - Operations:
    - Initialization.
    - Check empty.
    - Find a key.
    - Visit.
    - ***Add a key.***
    - ***Remove a key.***
    - ***Rotate left a node.***
    - ***Rotate right a node.***

Implementation



- Add a key (keep tree condition):

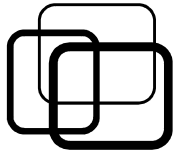
```
Add key to node {
    if node is NULL
        Make node with key
    else if ( key < m_data )
        Add key to left child
    else
        Add key to right child

    Update node height
    Rebalance node
}
```



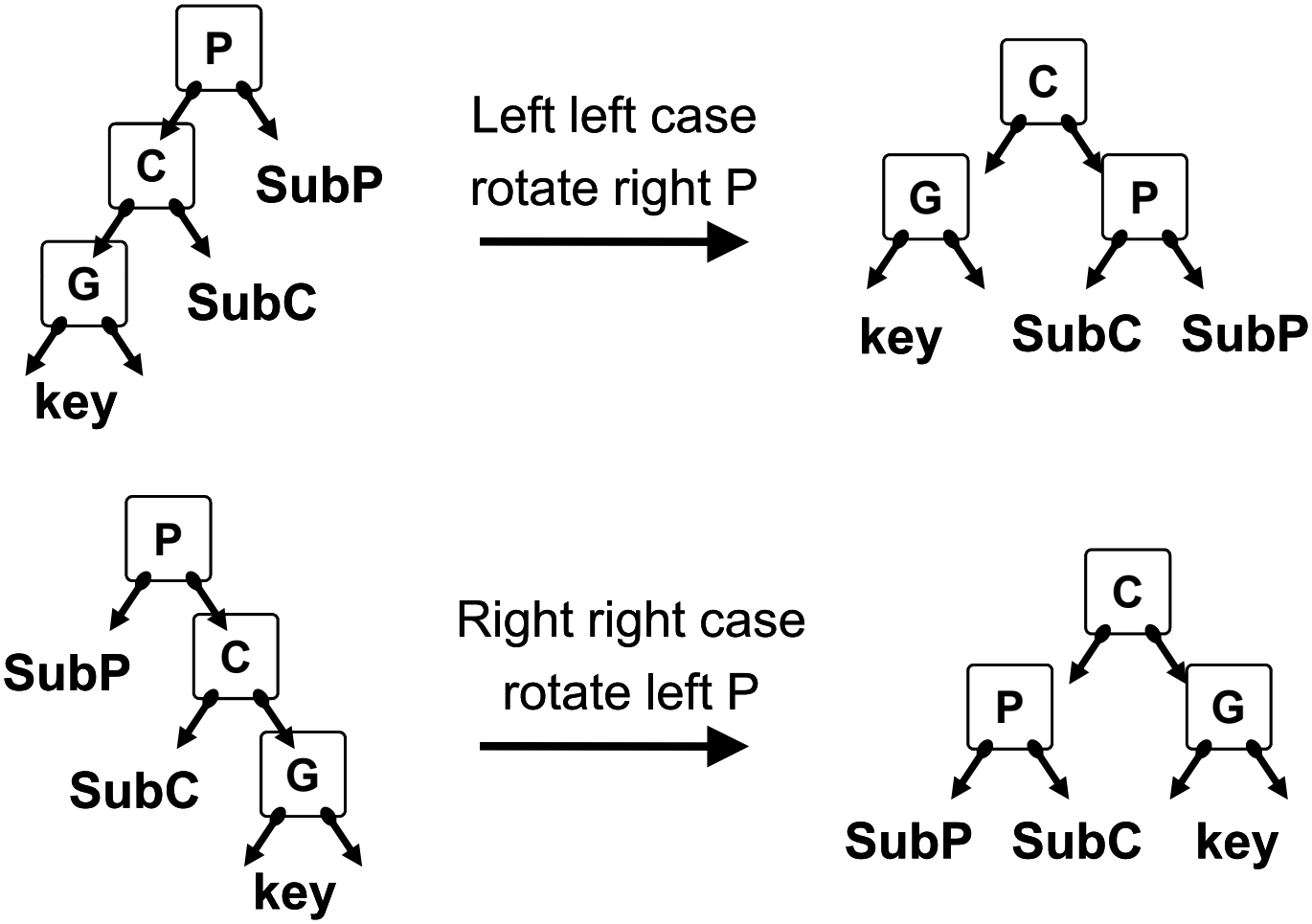
Tracing backward:

- Update height.
- Rebalance if needed.

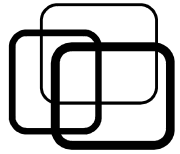
Implementation



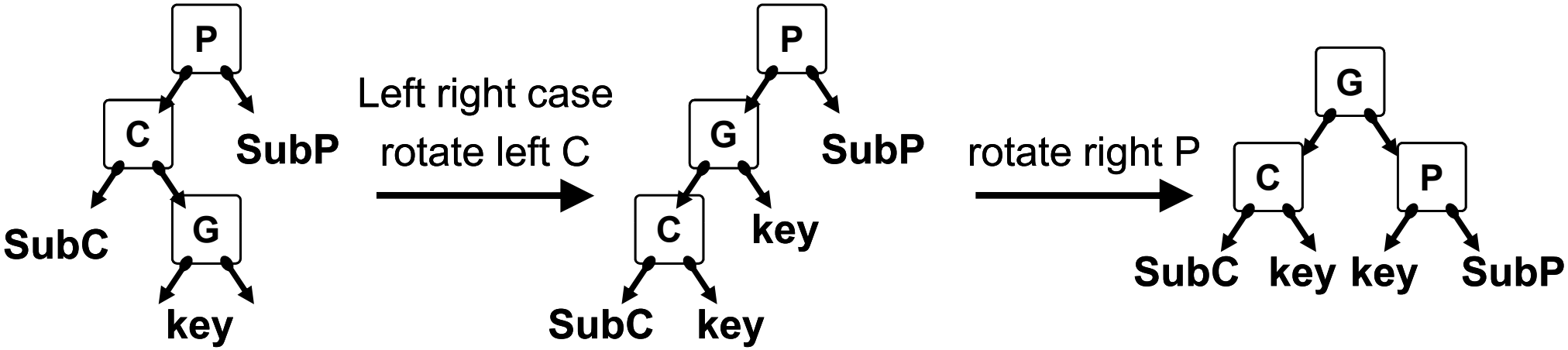
- Add a key (keep tree condition):
  - Rebalancing a node:

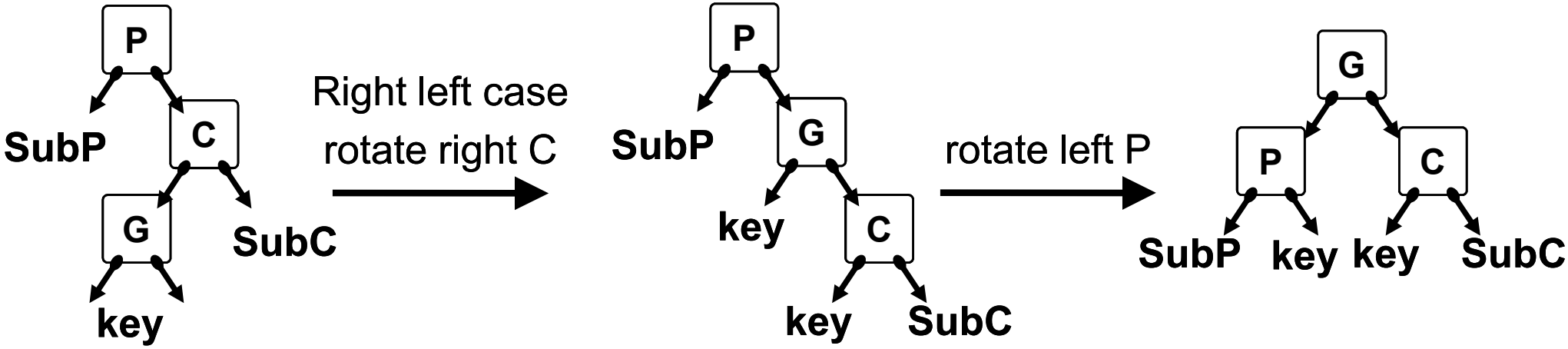


Implementation

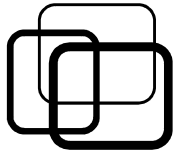


- Add a key (keep tree condition):
  - Rebalancing a node:



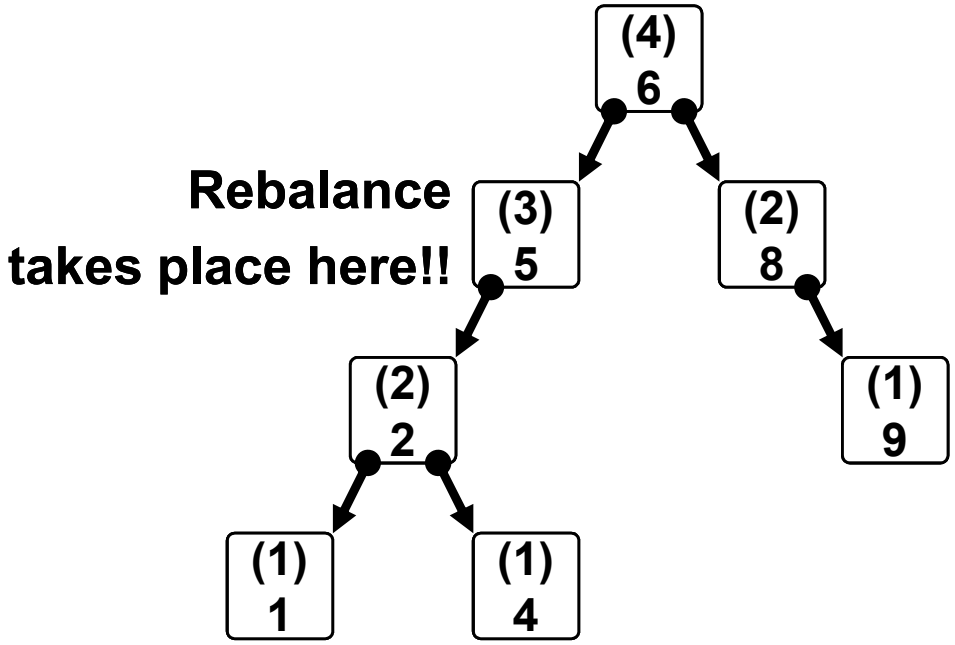


Implementation



- Remove a key (keep tree condition):

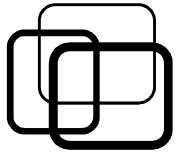
```
Remove key from node {
    if node is NULL
        Stop
    if ( key < m_data )
        Remove key on left child
    else if ( key > m_data )
        Remove key on right child
    else
        Replace node

    Update node height
    Rebalance node
}
```



Tracing backward:

- Update height.
- Rebalance if needed.

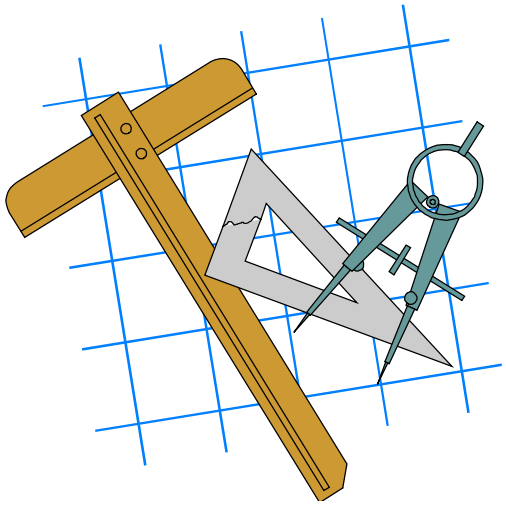
Practice



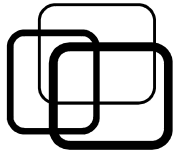
- Practice 9.1:

  Construct class **AVLTree** inherits from **BSTree** and extends the following methods:

  - Add key.
  - Remove key.
  - Rotate left a node.
  - Rotate right a node.
  - Update height of a node.
  - Rebalance a node.



Practice



- Practice 9.2:

  Implement **binary sort** by providing class **AVLTree** with the following methods:

  - Add key from an array of integers.
  - Export LNR all nodes an array.

  Example:

```
int  a [ ] = { 5, 2, 4, 1, 3 };
int  size = 5;
AVLTree t;
t.add( a, size );
t.exportLNR( a, size );
```