Data Structure

실습 8

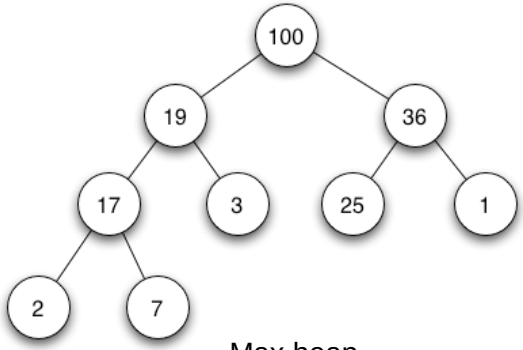
0. 이번 주 실습 내용

- **Heap 복습**
- **AVL Tree**
  - AVL Tree의 정의
  - AVL Tree의 구현 방법 (rotation operation)
- **AVL Tree 실습**
  - AVL Tree 구현 실습
- **과제: AVL Tree 삭제 구현**

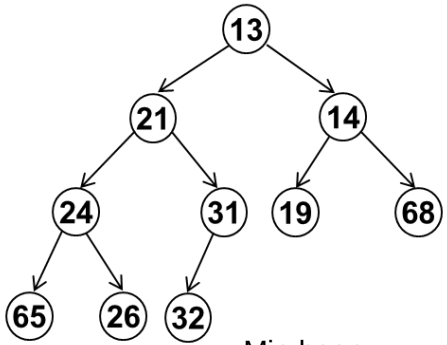
1. Heap

- **Heap (정의): 여러 개의 값들 중에서 가장 큰 값, 또는 가장 작은 값을 빠르게 탐색하도록 만들어진 Binary Tree 자료구조**

- 조건
  - Tree는 **Complete** 해야한다
  - 각 Node의 key 값은 자식 Node 들의 Key 값보다 **크거나 같아야 한다** (Max heap)
    cf.) 각 Node의 key 값이 자식 Node 들의 Key 값보다 **작거나 같아야 한다** (Min heap)



Max heap



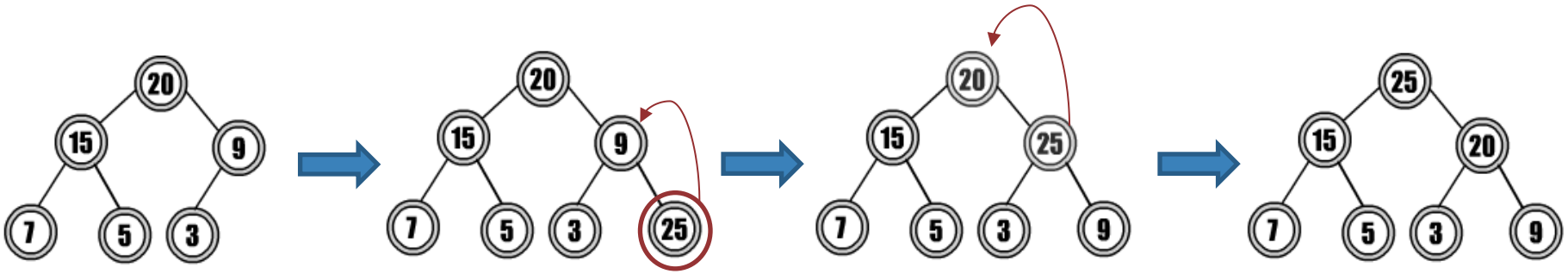
Min heap

1. Heap – 구현 (Max heap)

- **insert (데이터 추가 연산) O(log n)**
  1. Tree에 Node를 하나 확장 시켜 추가할 데이터를 넣어준다
  2. 부모 Node의 데이터와 값을 비교한다

     2-1. 부모 Node의 데이터보다 값이 클 경우 부모와 위치를 바꿔준 다음 다시 2번을 수행한다

     2-2. 부모 Node의 데이터보다 값이 작을 경우 연산을 종료한다.



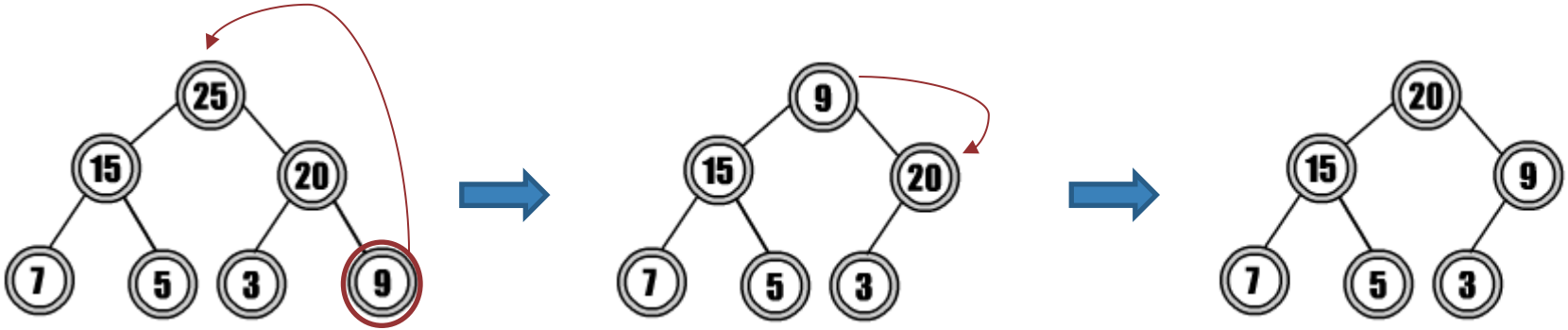
insert 25

1. Heap – 구현 (Max heap)

- **delete (데이터 삭제 연산) O(log n)**
  1. root Node의 데이터를 삭제하고 마지막 Node를 root Node로 이동시킨다
  2. 자식 Node들의 데이터 값을 비교한다

     2-1. 자식 Node들의 데이터 값 중 가장 큰 값을 가진 Node와 위치를 바꾸고 다시 2번을 수행한다

     2-2. 이동이 이루어지지 않았을 경우 연산을 종료한다

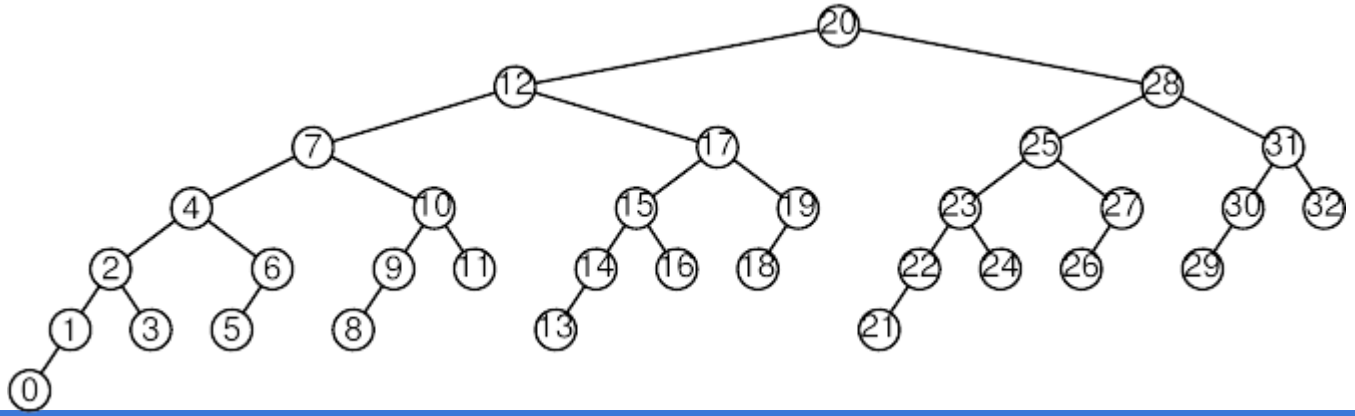


delete 25

2. AVL Tree

- **AVL Tree (정의): Left Sub Tree 의 높이와 Right Sub Tree 의 높이 차이가 1 이하인 Binary Search Tree 자료구조**
  - **Balanced Binary Search Tree**

  - Node의 삽입/삭제 연산 등으로 트리가 비 균형 상태가 되면 스스로 균형 상태를 유지할 수 있도록 트리의 구조를 변경
  - 비 균형 상태: |왼쪽 서브 트리 높이 – 오른쪽 서브 트리 높이| > 1



2. AVL Tree

- **Insertion Operation (삽입 연산)**
  - Binary Search Tree와 같은 방법으로 노드를 추가

  - 균형이 깨진 상태가 될 경우는 4가지가 존재함
    - N: 새로 추가된 노드
    - A: N과 가장 가까우면서 균형이 깨진 조상 노드

    - **LL Type**: N이 A의 왼쪽 서브 트리의 왼쪽 서브 트리에 추가되었을 때
    - **LR Type**: N이 A의 왼쪽 서브 트리의 오른쪽 서브 트리에 추가되었을 때
    - **RR Type**: N이 A의 오른쪽 서브 트리의 오른쪽 서브 트리에 추가되었을 때
    - **RL Type**: N이 A의 오른쪽 서브 트리의 왼쪽 서브 트리에 추가되었을 때

➔Node 들의 회전 연산을 통해 균형을 유지시키는 것이 AVL Tree의 특징!

2. AVL Tree

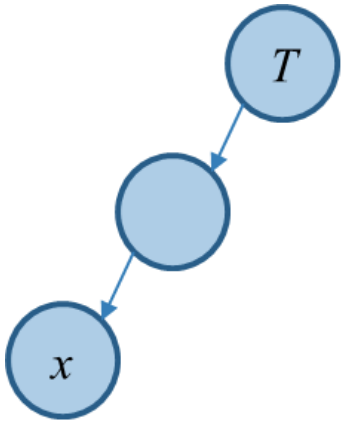
- **Rotation Operation (회전 연산)**
  - 균형을 맞추기 위해 부모 노드와 자식 노드를 바꾸는 연산
    - N: 새로 추가된 노드
    - A: N과 가장 가까우면서 균형이 깨진 조상 노드

    - **LL 회전**: N부터 A까지 경로상의 노드들을 오른쪽으로 회전
    - **LR 회전**: N부터 A까지 경로상의 노드들을 왼쪽-오른쪽으로 회전
    - **RR 회전**: N부터 A까지 경로상의 노드들을 왼쪽으로 회전
    - **RL 회전**: N부터 A까지 경로상의 노드들을 오른쪽-왼쪽으로 회전

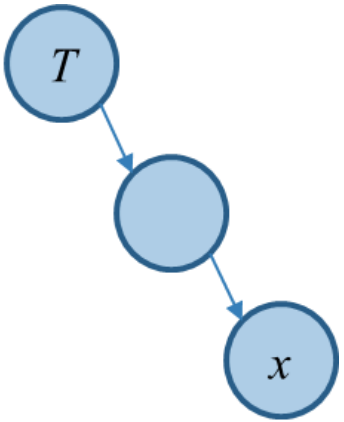
  - **단순 회전(Single Rotation): LL, RR 회전**
  - **이중 회전(Double Rotation): LR, RL 회전**

2. AVL Tree

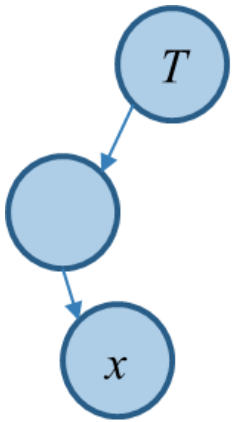
- **Rotation Operation (회전 연산)**



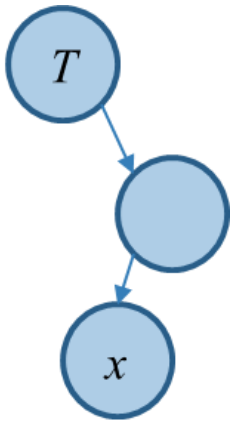
LL Rotation



RR Rotation



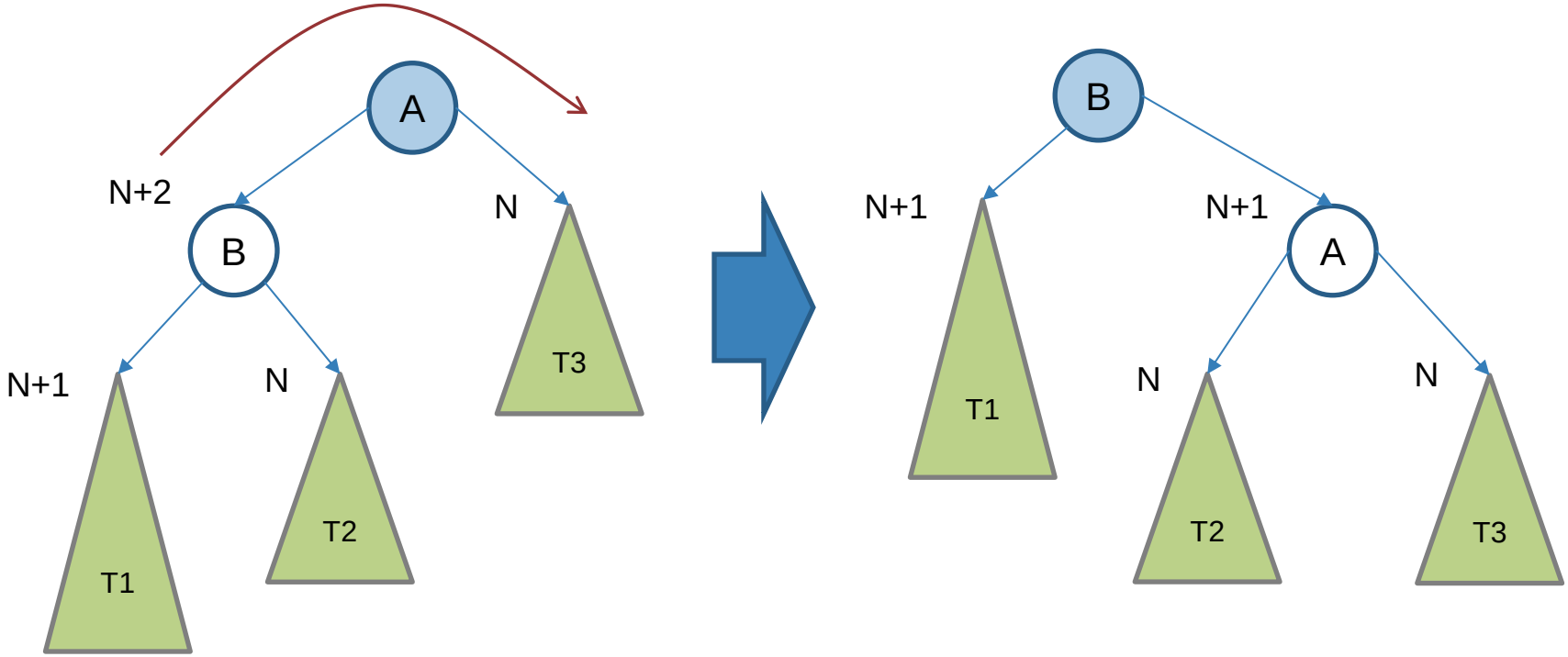
LR Rotation
(RR Rotation → LL Rotation)



RL Rotation
(LL Rotation → RR Rotation)

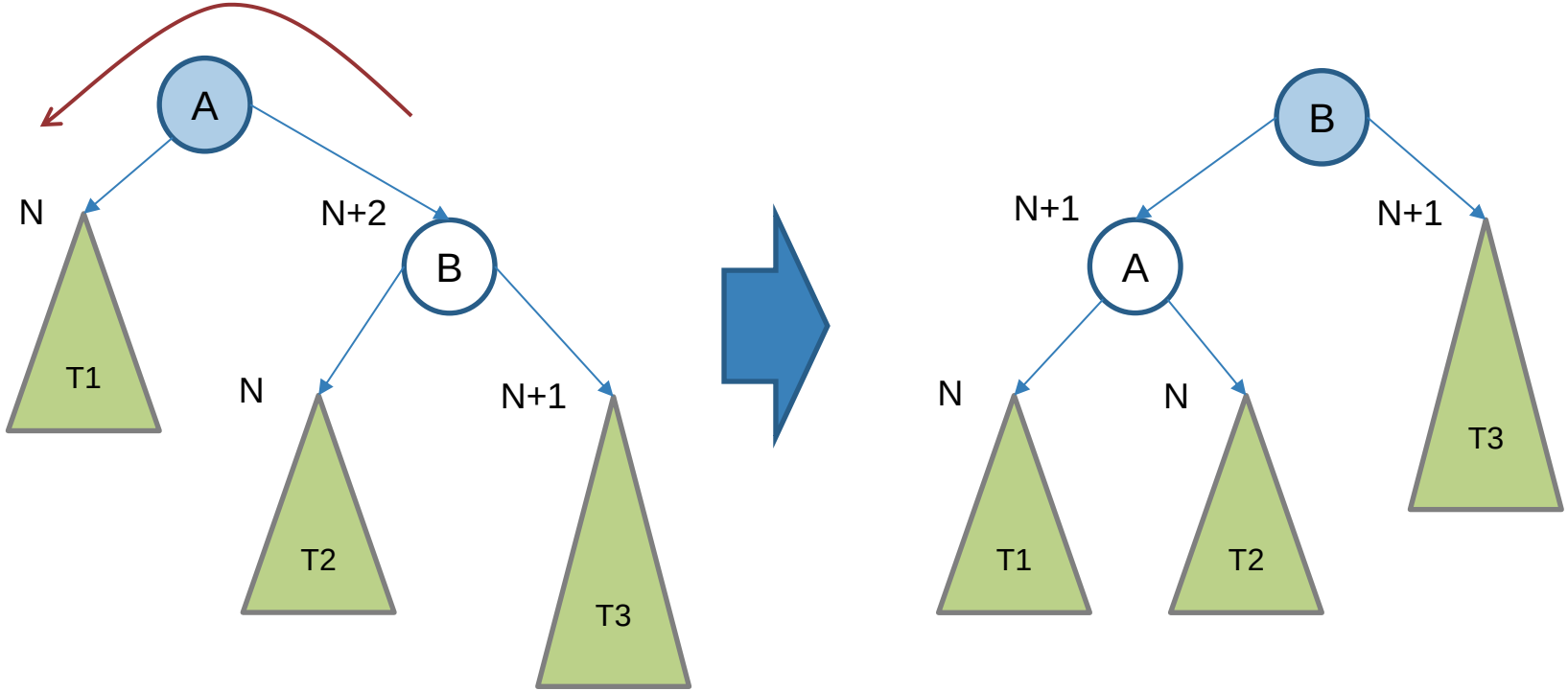
2. AVL Tree

- **Single Rotation – LL Rotation**



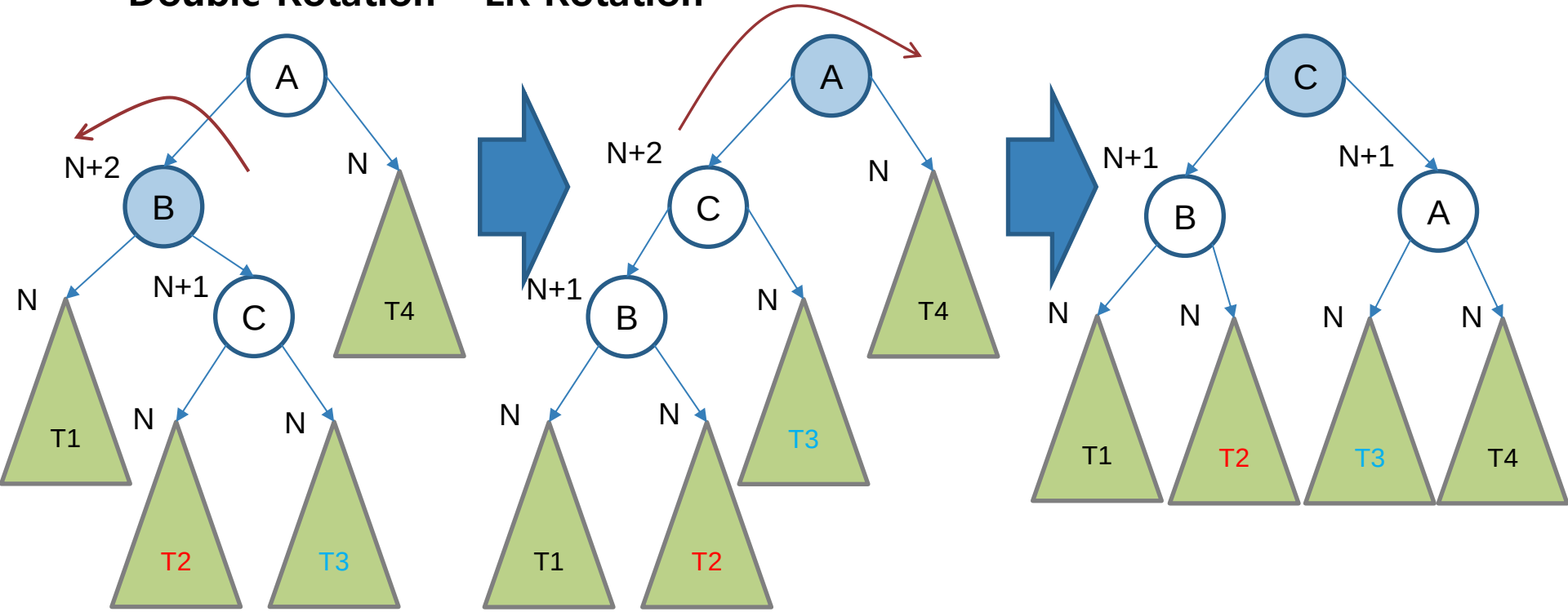
2. AVL Tree

- **Single Rotation – RR Rotation**



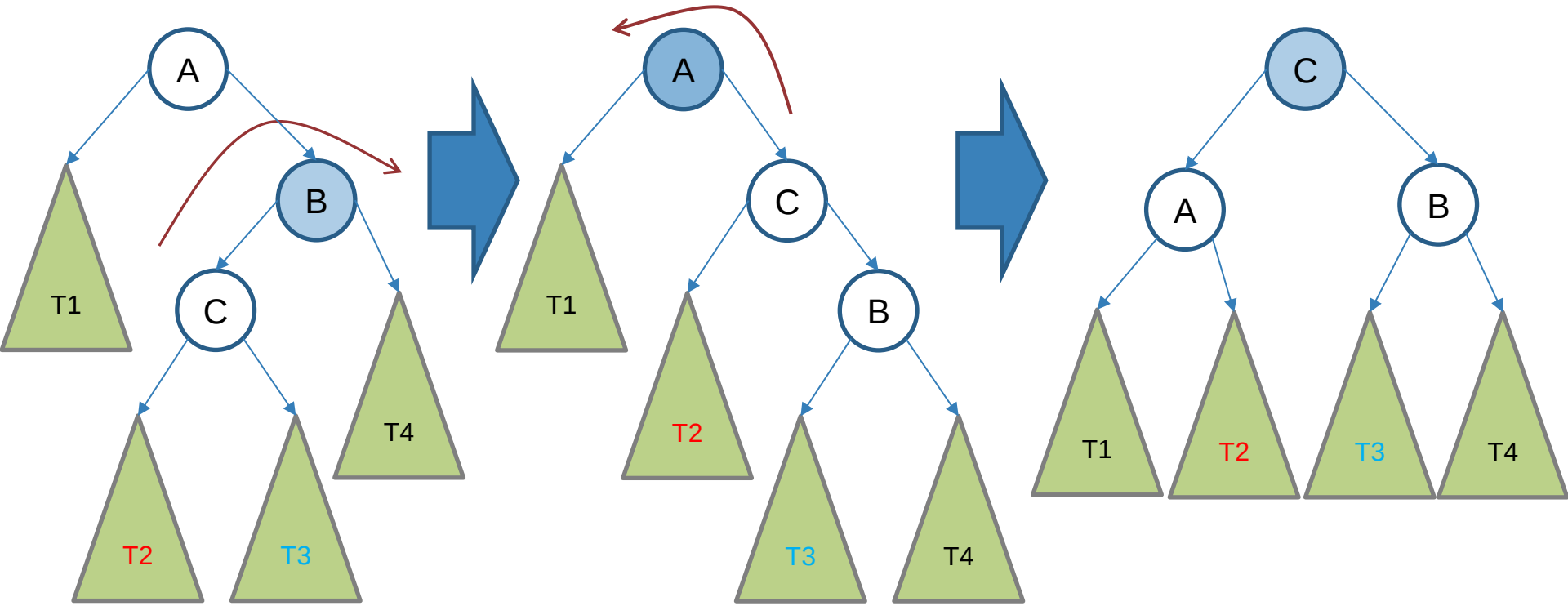
2. AVL Tree

- **Double Rotation – LR Rotation**



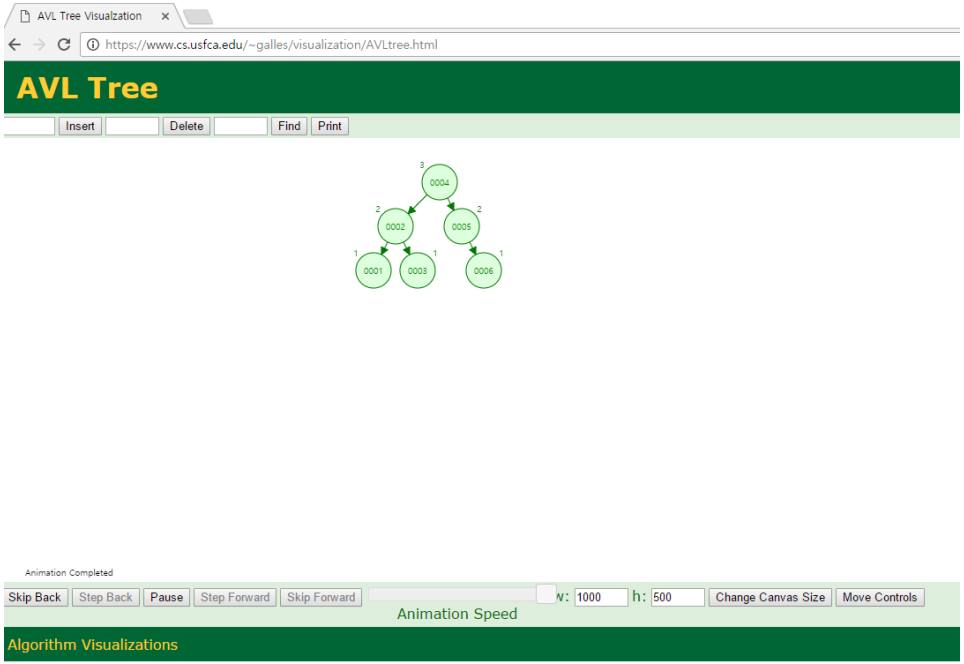
2. AVL Tree

- **Double Rotation – RL Rotation**



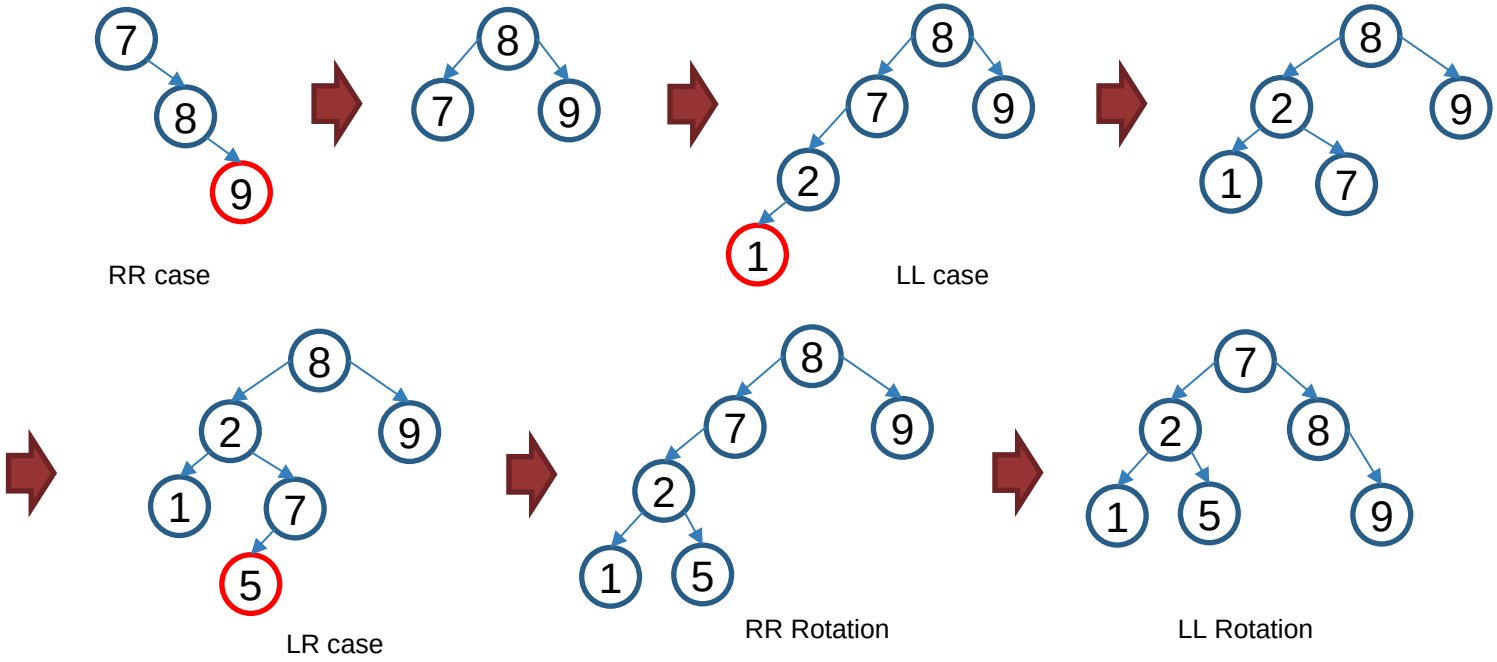
2. AVL Tree

- Insertion Exercise (AVL Tree Visualization)
  - AVL Tree 시각화 사이트: https://www.cs.usfca.edu/~galles/visualization/AVLtree.html



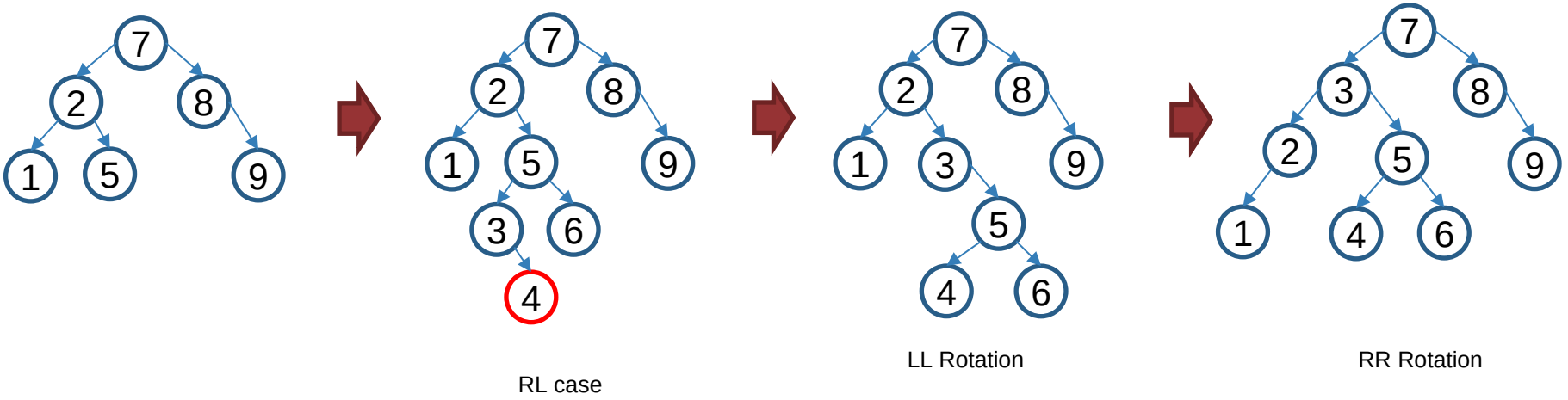
2. AVL Tree

- Insertion Exercise (AVL Tree Visualization)
  - Insert: 7, 8, 9, 2, 1, 5, 3, 6, 4



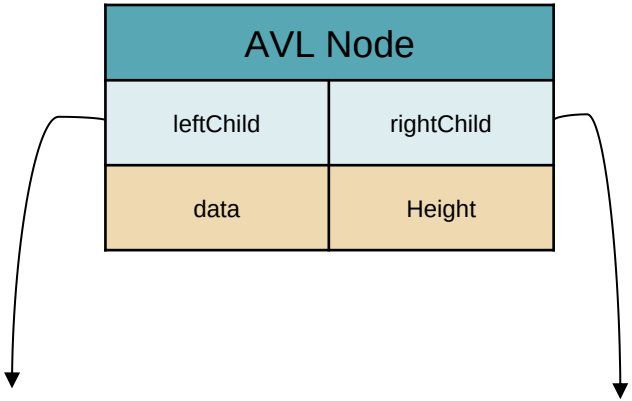
2. AVL Tree

- Insertion Exercise (AVL Tree Visualization)
  - Insert: 7, 8, 9, 2, 1, 5, 3, 6, 4



2. AVL Tree – 실습

- **AVL Tree Data Type**

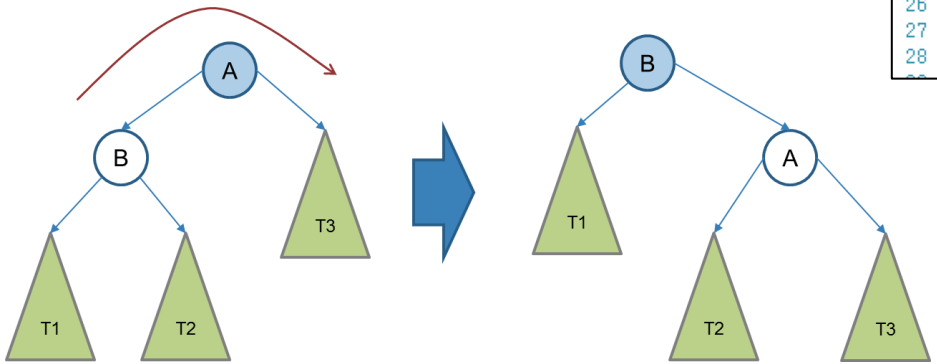



```c
#include <stdio.h>
#include <stdlib.h>
 #define Max(x,y) ((x)>(y))?(x):(y)

typedef struct AvlNode {
    int data;
    struct AvlNode *left, *right;
    int Height;
}AvlNode;

```

```c
int height(AvlNode* node) {
    if (node == NULL)
        return -1;
    else
        return node->Height;
}

```

3. AVL Tree – 실습

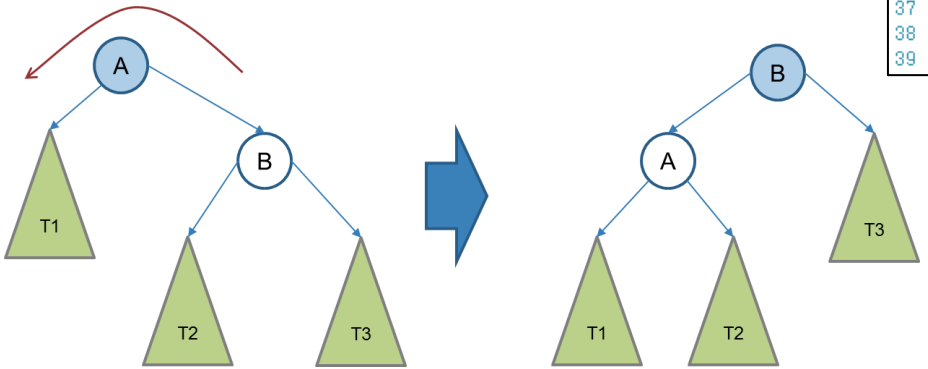
- **LL Rotation**
  - 부모 (node)가 자식 (node)의 오른쪽 sub tree가 됨
  - 기존에 있었던 자식의 오른쪽 sub tree는 부모의 왼쪽 sub tree가 됨
  - 회전 후 부모와 자식의 height를 update




```
19  AvlNode* rotateLL(AvlNode* parent) {
20      AvlNode* child = parent->left;
21
22
23
24
25
26
27      return child;
28  }
```

3. AVL Tree – 실습

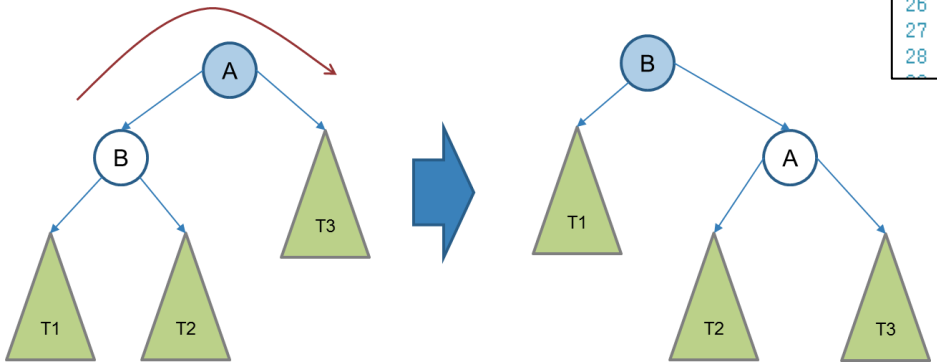
- **RR Rotation**
  - 부모 (node)가 자식 (node)의 왼쪽 sub tree가 됨
  - 기존에 있었던 자식의 왼쪽 sub tree는 부모의 오른쪽 sub tree가 됨
  - 회전 후 부모와 자식의 height를 update




```
30  AvlNode* rotateRR(AvlNode* parent) {
31      AvlNode* child = parent->right;
32
33
34
35
36
37
38      return child;
39  }
```

3. AVL Tree – 실습

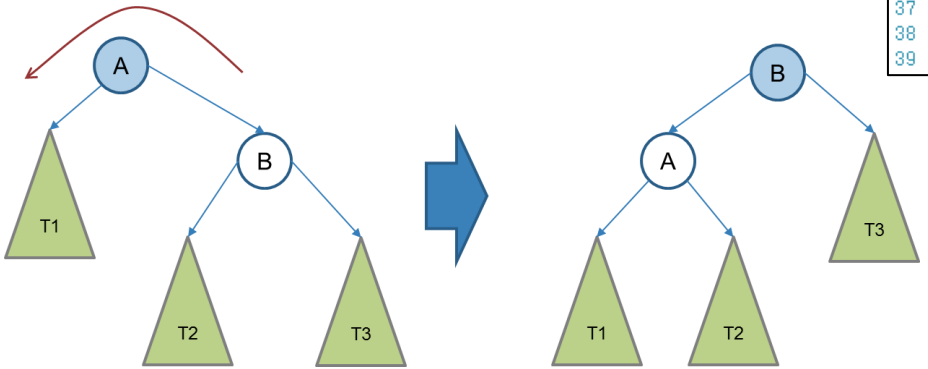
- **LL Rotation**
  - 부모 (node)가 자식 (node)의 오른쪽 sub tree가 됨
  - 기존에 있었던 자식의 오른쪽 sub tree는 부모의 왼쪽 sub tree가 됨
  - 회전 후 부모와 자식의 height를 update




```
19  AvlNode* rotateLL(AvlNode* parent) {
20      AvlNode* child = parent->left;
21      parent->left = child->right;
22      child->right = parent;
23
24      parent->Height = Max(height(parent->left), height(parent->right)) + 1;
25      child->Height = Max(height(child->left), parent->Height) + 1;
26
27      return child;
28  }
```

3. AVL Tree – 실습

- **RR Rotation**
  - 부모 (node)가 자식 (node)의 왼쪽 sub tree가 됨
  - 기존에 있었던 자식의 왼쪽 sub tree는 부모의 오른쪽 sub tree가 됨
  - 회전 후 부모와 자식의 height를 update

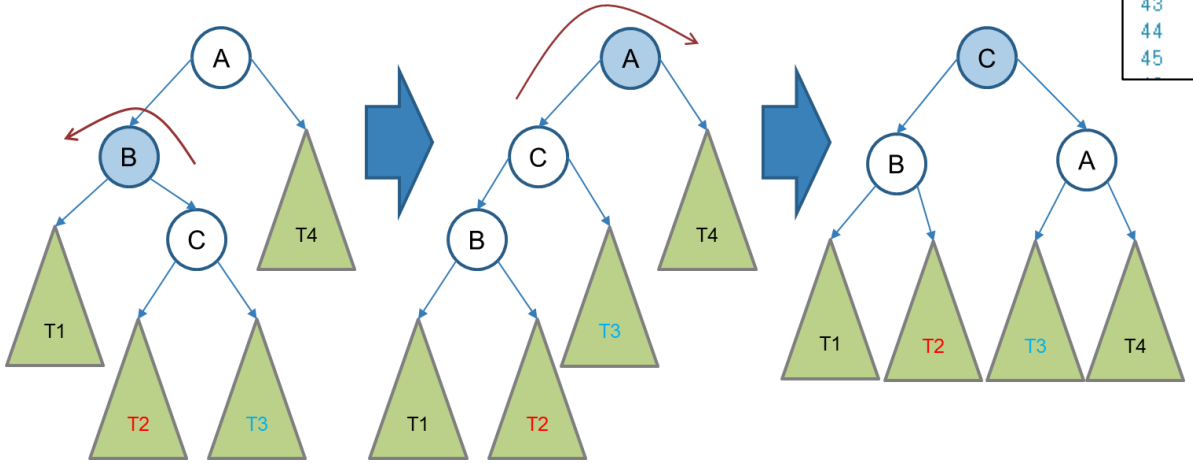



```
30  AvlNode* rotateRR(AvlNode* parent) {
31      AvlNode* child = parent->right;
32      parent->right = child->left;
33      child->left = parent;
34
35      parent->Height = Max(height(parent->left), height(parent->right)) + 1;
36      child->Height = Max(parent->Height, height(child->right)) + 1;
37
38      return child;
39  }
```

3. AVL Tree – 실습

- **LR Rotation**
  - 자식 (node)로 하여금 RR rotation을 수행
  - 그 결과로 LL type 형태가 되므로 부모 (node) 기준으로 LL rotation 수행
  - 회전 후 부모와 자식의 height update는 자동 수행됨

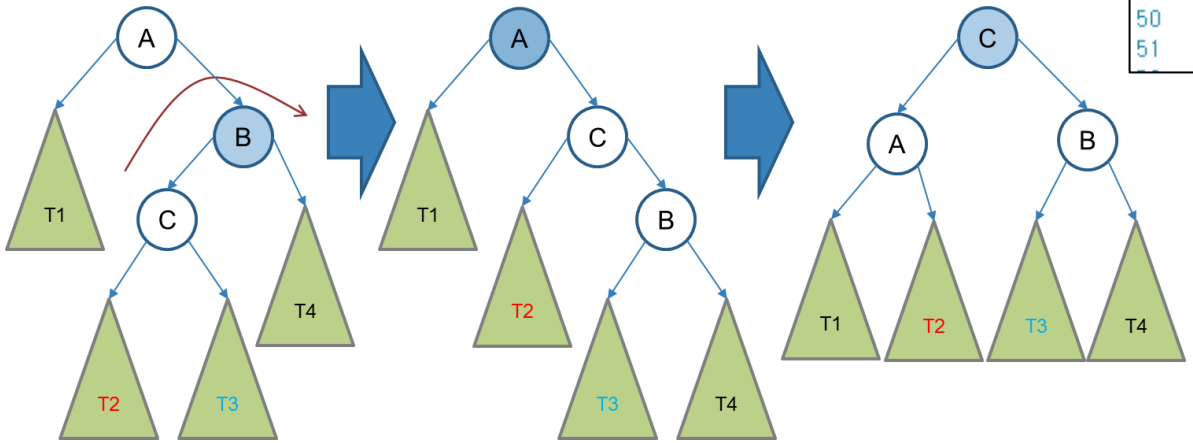
```
41  AvlNode* rotateLR(AvlNode* parent) {
42      AvlNode* child = parent->left;
43
44
45  }
```



3. AVL Tree – 실습

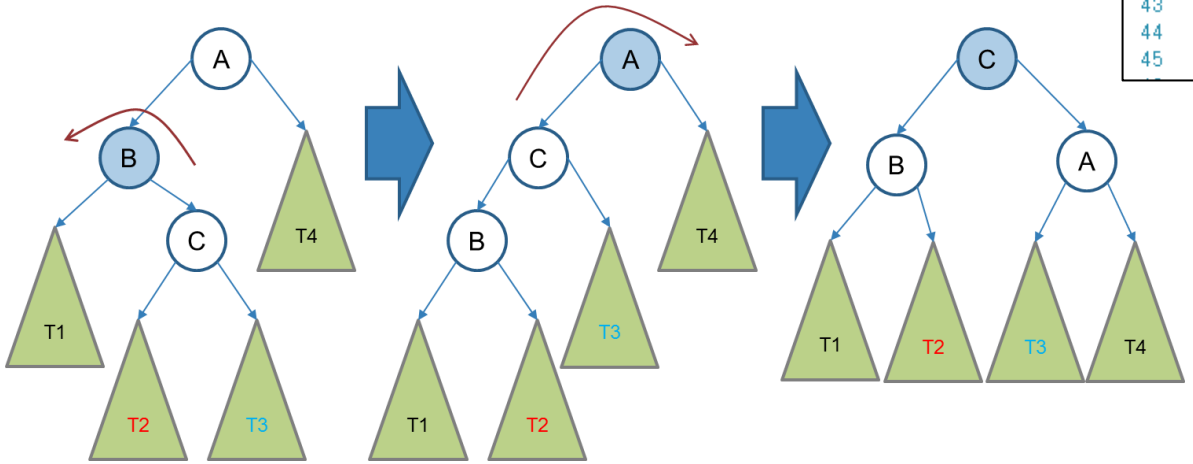
- **RL Rotation**
  - 자식 (node)로 하여금 LL rotation을 수행
  - 그 결과로 RR type 형태가 되므로 부모 (node) 기준으로 RR rotation 수행
  - 회전 후 부모와 자식의 height update는 자동 수행됨

```
47  AvlNode* rotateRL(AvlNode* parent) {
48      AvlNode* child = parent->right;
49
50
51  }
```



3. AVL Tree – 실습

- **LR Rotation**
  - 자식 (node)로 하여금 RR rotation을 수행
  - 그 결과로 LL type 형태가 되므로 부모 (node) 기준으로 LL rotation 수행
  - 회전 후 부모와 자식의 height update는 자동 수행됨

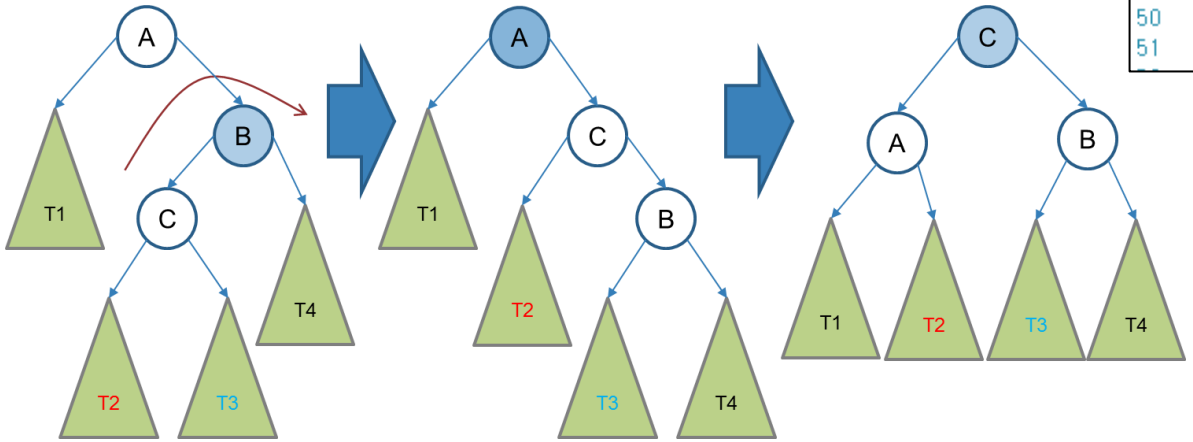



```
41  AvlNode* rotateLR(AvlNode* parent) {
42      AvlNode* child = parent->left;
43      parent->left = rotateRR(child);
44      return rotateLL(parent);
45  }
```

3. AVL Tree – 실습

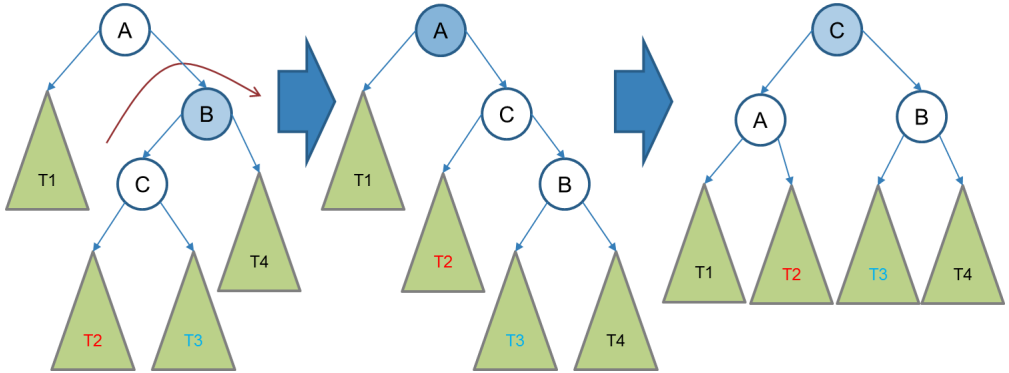
- **RL Rotation**
  - 자식 (node)로 하여금 LL rotation을 수행
  - 그 결과로 RR type 형태가 되므로 부모 (node) 기준으로 RR rotation 수행
  - 회전 후 부모와 자식의 height update는 자동 수행됨

```
47    AvlNode* rotateRL(AvlNode* parent) {
48        AvlNode* child = parent->right;
49        parent->right = rotateLL(child);
50        return rotateRR(parent);
51    }
```



3. AVL Tree – 실습

- **Insertion operation**
  - 기본적으로 BST insertion과 같은 방법으로 진행
    - 현재 node의 data보다 작으면 왼쪽 child, 크면 오른쪽 child node로 탐색
    - leaf node까지 내려와서 node 추가 수행
  - Node가 추가되고 난 다음 현재 위치에서 왼쪽과 오른쪽 sub tree의 height를 비교하여 균형을 맞춰 주도록 함
  - 추가가 완료된 다음 해당 Node의 height를 update




```
55  AvlNode* avlAdd(AvlNode *root, int data) {
56      if (root == NULL) {
57          root = (AvlNode*)malloc(sizeof(AvlNode));
58          if (root == NULL) {
59              exit(1);
60          }
61
62
63
64
65      }
66      else if (data < root->data) {
67          root->left =
68          //root = rebalance(root);
69          if (height(root->left) - height(root->right) == 2)
70          {
71
72
73
74
75          }
76      }
77      else if (data >root->data) {
78          root->right =
79          //root = rebalance(root);
80          if (height(root->right) - height(root->left) == 2)
81          {
82
83
84
85
86          }
87      }
88      else {
89          printf("중복 키 삽입 오류₩n");
90          exit(1);
91      }
92      root->Height = Max(height(root->left), height(root->right)) + 1;
93      return root;
94  }
```

3. AVL Tree – 실습

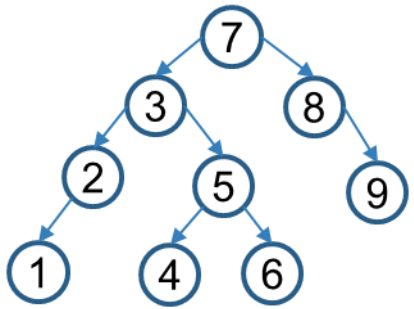
- **Search & inorder traveling**

```
 96  AvlNode *avlSearch(AvlNode * node, int key) {
 97      if (node == NULL) return NULL;
 98      if (key == node->data) return node;
 99      else if (key < node->data)
100          return avlSearch(node->left, key);
101      else
102          return avlSearch(node->right, key);
103  }
104
105  AvlNode* inorderTraveling(AvlNode *root) {
106      if (root != NULL)
107      {
108          inorderTraveling(root->left);
109          printf("[%d] ", root->data);
110          inorderTraveling(root->right);
111      }
112  }
```

3. AVL Tree – 실습

- **main**

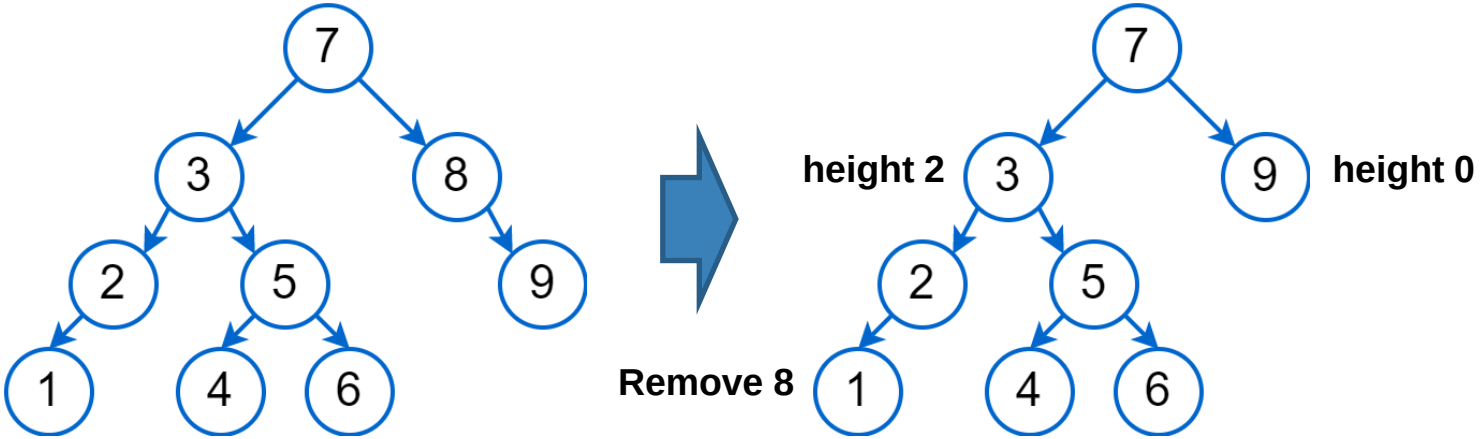
```
C:₩Windows₩system32₩cmd

검색 : 4

[1] [2] [3] [4] [5] [6] [7] [8] [9]
```




```
114  int main() {
115
116      AvlNode *root = NULL;
117
118      root = avlAdd(root, 7);
119      root = avlAdd(root, 8);
120      root = avlAdd(root, 9);
121      root = avlAdd(root, 2);
122      root = avlAdd(root, 1);
123      root = avlAdd(root, 5);
124      root = avlAdd(root, 3);
125      root = avlAdd(root, 6);
126      root = avlAdd(root, 4);
127
128      printf("₩n검색 : %d₩n", avlSearch(root, 4)->data);
129      printf("₩n");
130      inorderTraveling(root);
131      return 0;
132  }
```

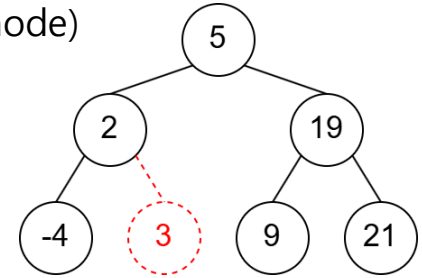
4. AVL Tree – 과제(삭제)

- **Remove Operation in AVL Tree**
  - Remove 8

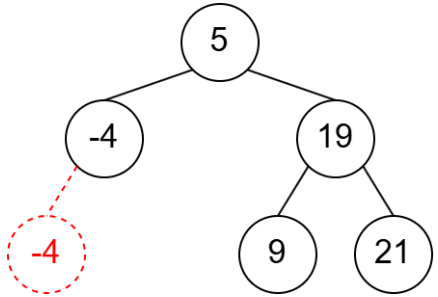


4. AVL Tree – 과제(삭제)

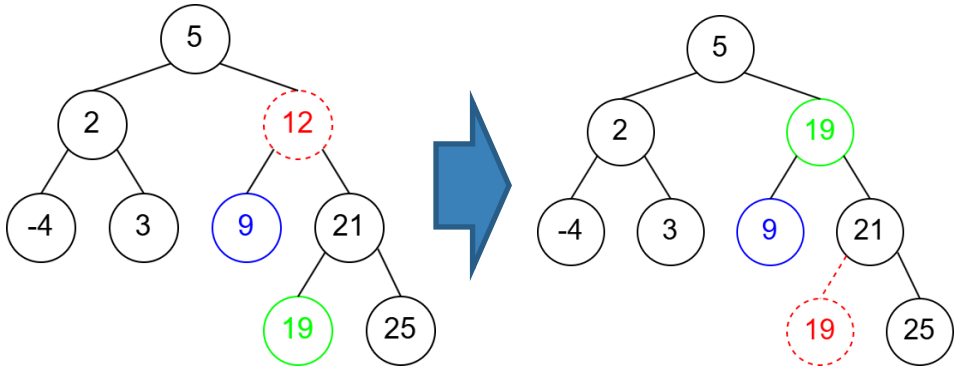
- **Remove Operation in BST Tree**
  - **Case1**. child Node가 하나도 없는 경우(leaf node)
  - **Case2.** child Node가 하나 존재하는 경우
    → 하나 있는 child Node를 부모 노드에 연결
  - **Case3.** child Node가 둘 다 있는 경우
    → 트리에서 successor를 찾아 대체



Case 1



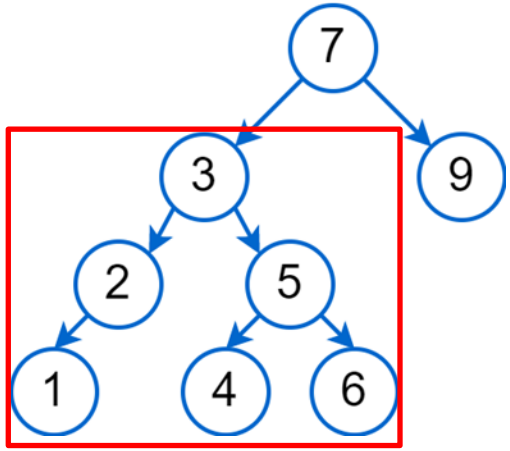
Case 2



Case 3

4. AVL Tree – 과제(삭제)

- **Remove Operation in AVL Tree**
  - Remove 8



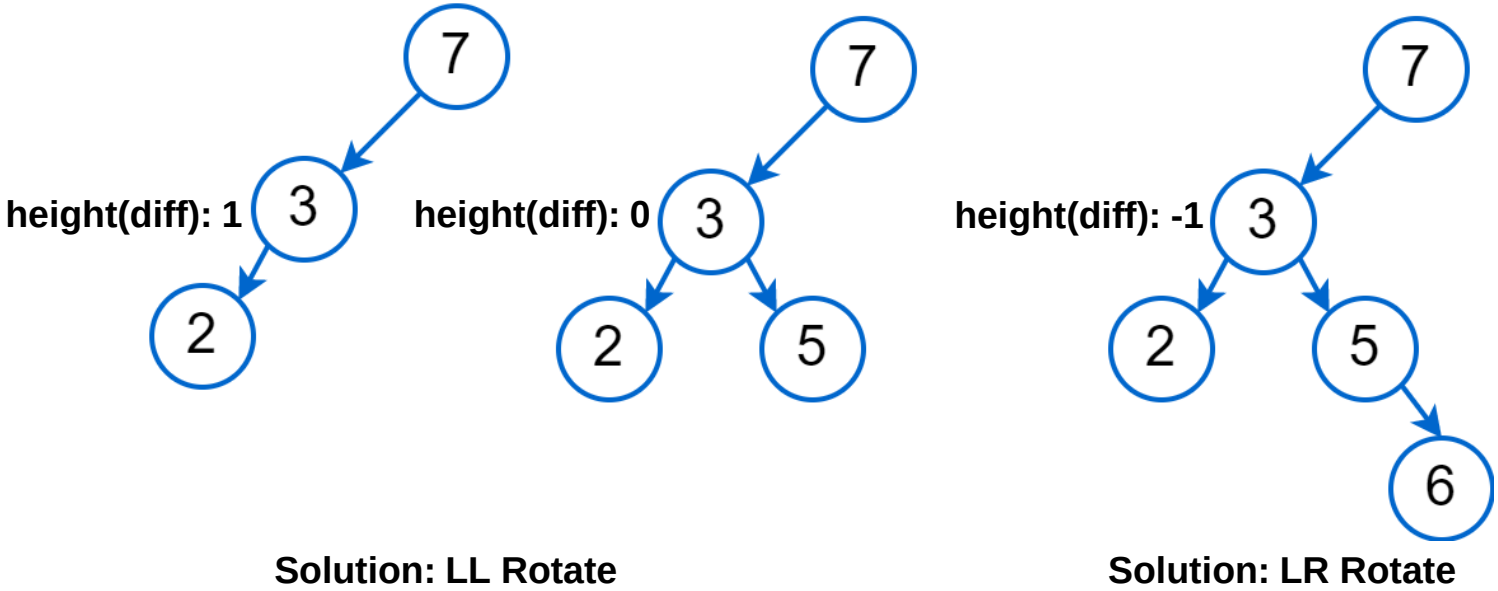
Unbalance!

- **Root 기준 왼쪽 서브 트리에서 문제 발생**
- **왼쪽 서브 트리의 height 차이가 0**

→ ***height(left->left) == height(left->right)***

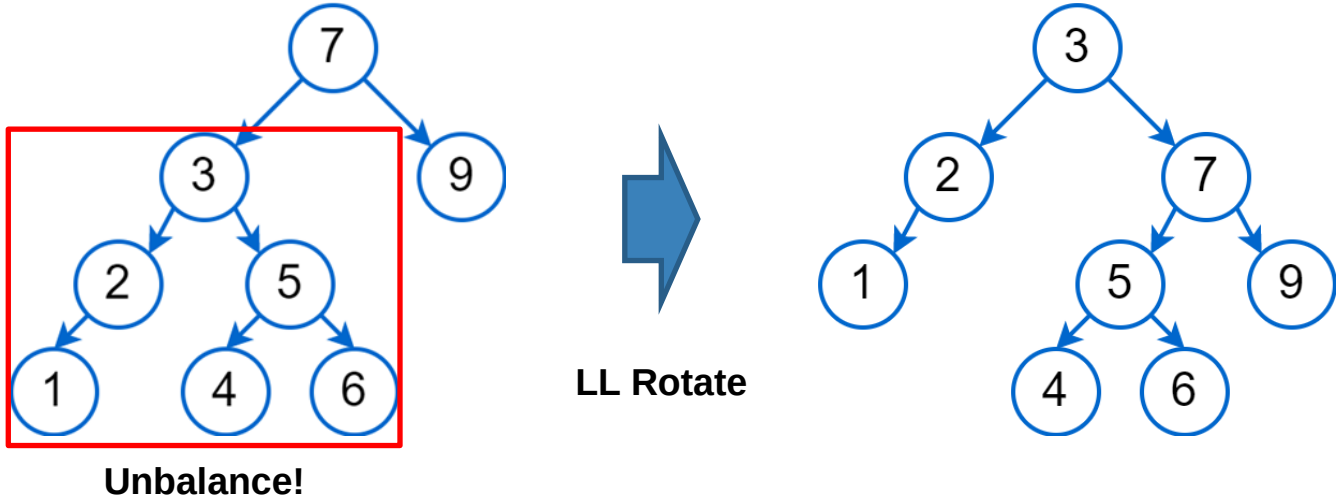
4. AVL Tree – 과제(삭제)

- **Remove Operation in AVL Tree**
  - Compute *height(left) – height(right)*



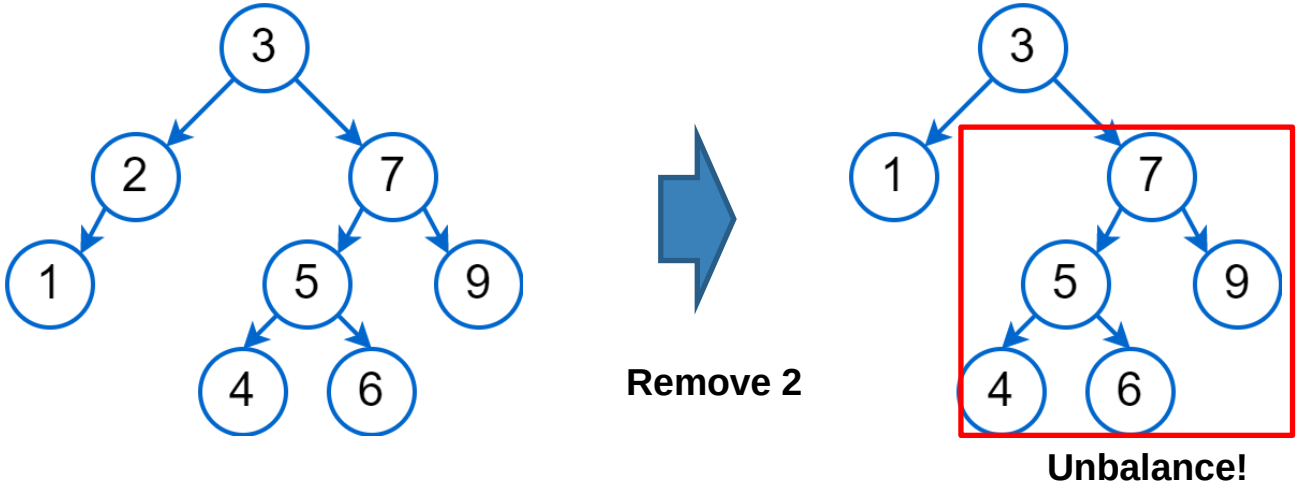
4. AVL Tree – 과제(삭제)

- **Remove Operation in AVL Tree**
  - Remove 8



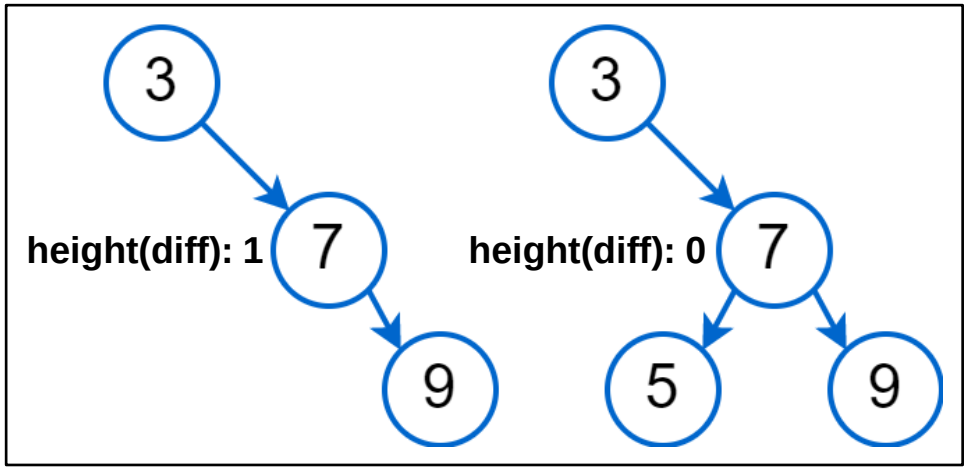
4. AVL Tree – 과제(삭제)

- **Remove Operation in AVL Tree**
  - Remove 2

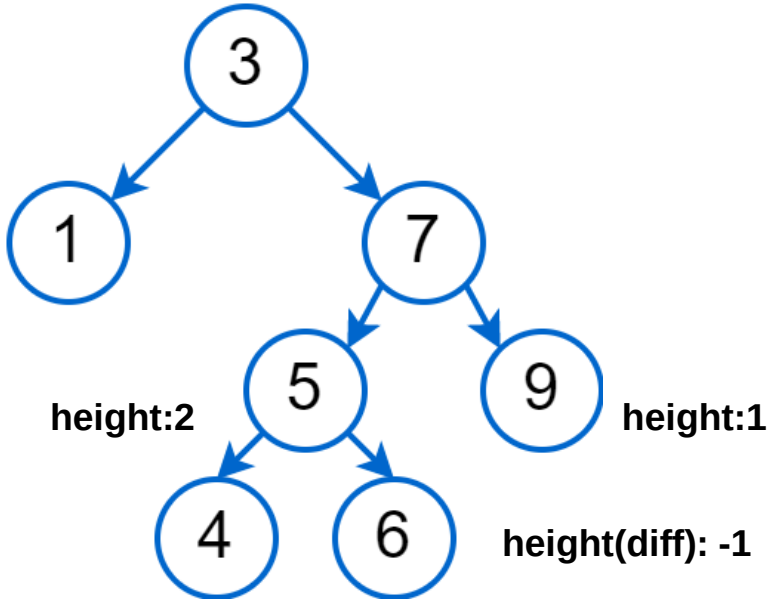


4. AVL Tree – 과제(삭제)

- **Remove Operation in AVL Tree**
  - Remove 2
  - Compute *height(right) – height(left)*



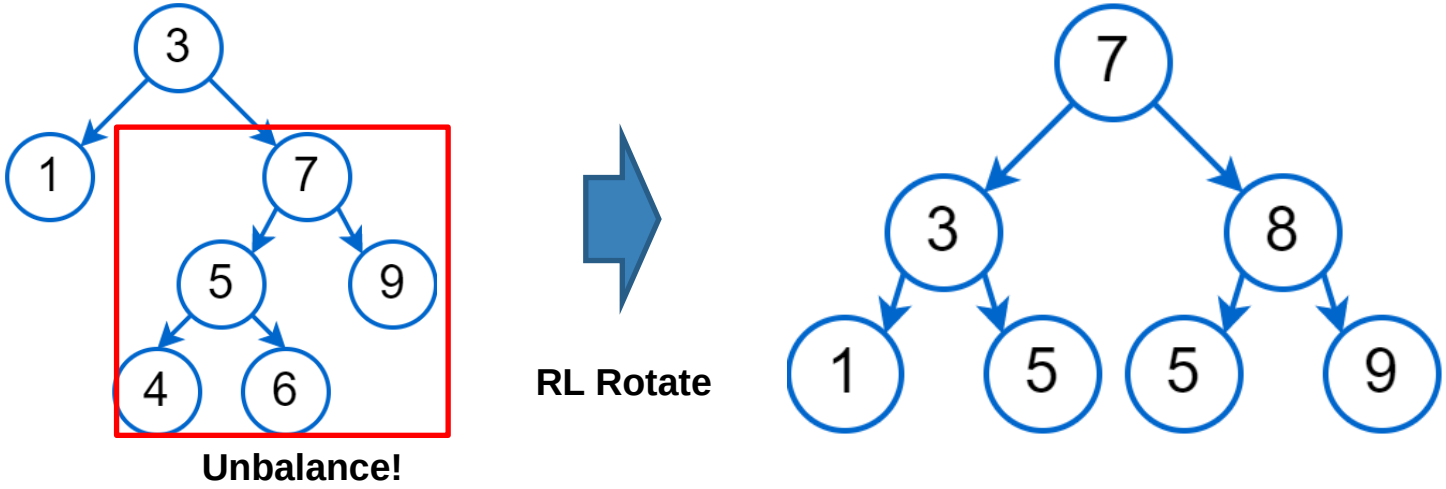
Solution: RR Rotate



Solution: RL Rotate

4. AVL Tree – 과제(삭제)

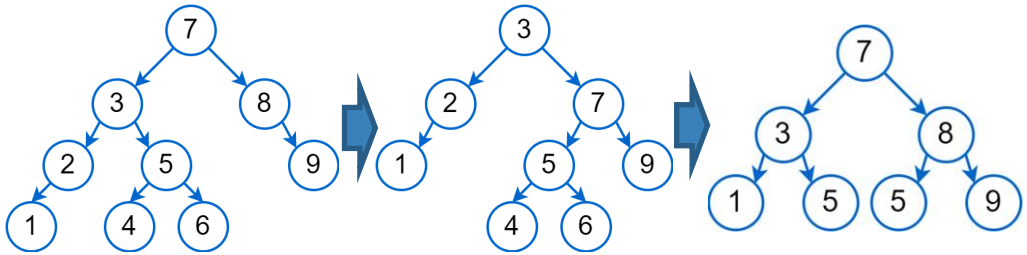
- **Remove Operation in AVL Tree**
  - Remove 2



4. AVL Tree – 과제(삭제)

- 수정된 main함수

```
C:₩Windows₩system32₩cmd
1 2 3 4 5 6 7 8 9
Root node: 7(h=3)
1 2 3 4 5 6 7 9
Root node: 3(h=3)
1 3 4 5 6 7 9
Root node: 5(h=2)
계속하려면 아무 키나 누르십시오 . . .
```



Remove 8 Remove 2

```int main() {
    AvlNode* root = NULL;

    root = avlAdd(root, 7);
    root = avlAdd(root, 8);
    root = avlAdd(root, 9);
    root = avlAdd(root, 2);
    root = avlAdd(root, 1);
    root = avlAdd(root, 5);
    root = avlAdd(root, 3);
    root = avlAdd(root, 6);
    root = avlAdd(root, 4);

    inorderTraveling(root);
    printf("\nRoot node: %d(h=%d)\n", root->data, root->Height);

    root = avlDelete(root, 8);
    inorderTraveling(root);
    printf("\nRoot node: %d(h=%d)\n", root->data, root->Height);

    root = avlDelete(root, 2);
    inorderTraveling(root);
    printf("\nRoot node: %d(h=%d)\n", root->data, root->Height);
    return 0;
}
```

감사합니다.

과제 제출 기한: 2025년 6월 4일 23:59분 (LMS 제출 시간 기준)

제출형식:

- 소스코드의 이름을 `avl_tree.c`로 작성하고 완성된 코드를 LMS 과제 탭에 제출
- 과제 제출 탭은 추후 생성 예정

궁금한 것이 생기면 언제든지 질문하시면 됩니다 ☺

- 공업센터본관 304호로 방문하시거나
- jeongiun@hanyang.ac.kr나 speedpaul@hanyang.ac.kr로 연락 바랍니다.
- 담당조교: 이범기, 이상윤