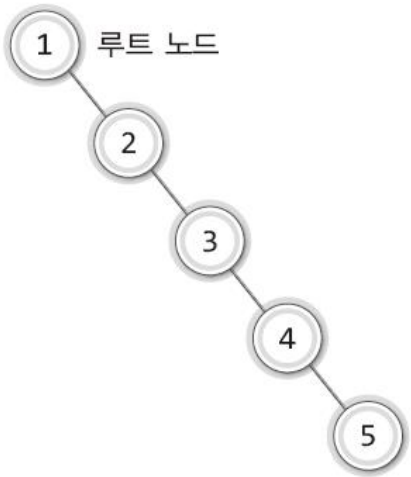
Chapter 12.
탐색(Search) 2

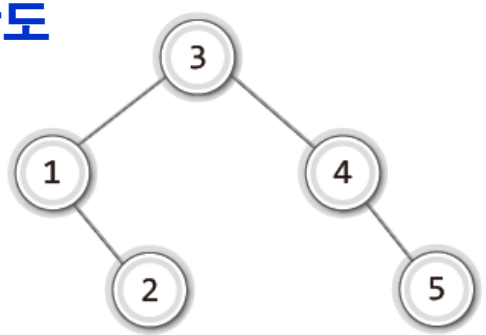
7주차

이진 탐색 트리의 문제점과 AVL 트리

- 1부터 5까지 순서대로 저장이 이뤄진 경우!
- 균형이 맞지 않을수록 $O(n)$에 가까운 시간 복잡도



- 3이 제일 먼저 저장된 경우!
- 탐색 연산은 $O(log_2 n)$의 시간 복잡도



이진 탐색 트리의 균형 문제를 해결한 트리

. AVL 트리 (노드의 추가/삭제 시 스스로 균형을 잡는 트리)

. AVL 트리의 종류

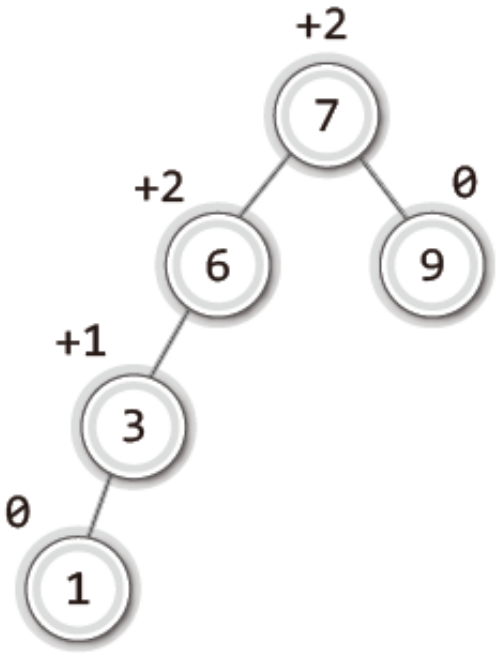
- 2-3 트리
- 2-3-4 트리
- Red-Black 트리
- B-tree

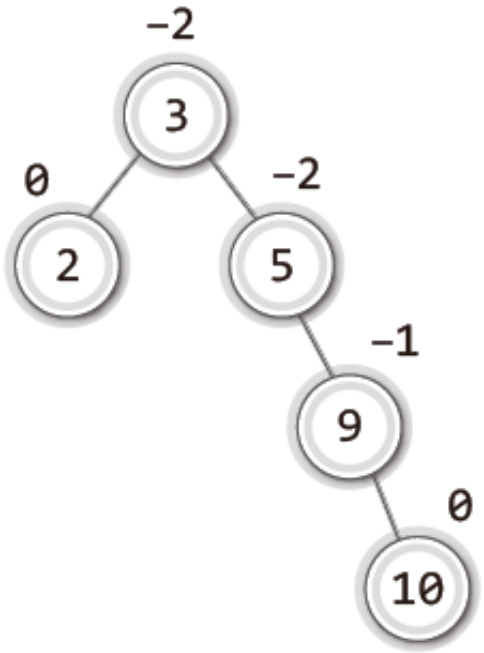
자동으로 균형 잡는 AVL 트리와 균형 인수(balancing factor)

균형 인수 = 왼쪽 서브트리의 높이 - 오른쪽 서브트리의 높이

균형 인수의 절댓값이 2 이상인 경우 리밸런싱(균형을 잡기 위한 트리 구조의 재조정) 진행!

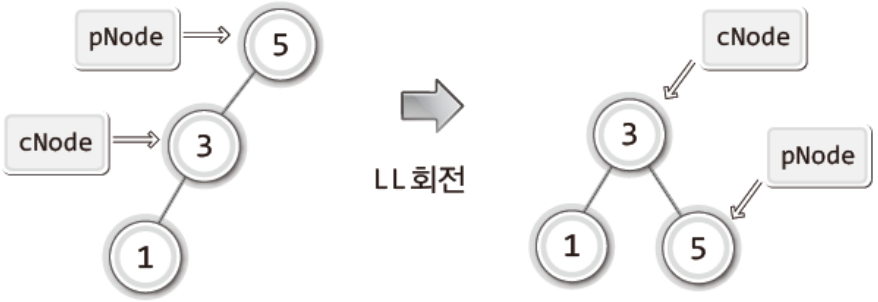
=> AVL 트리의 균형이 무너지는 상태를 4 종류로 나누어 처리.





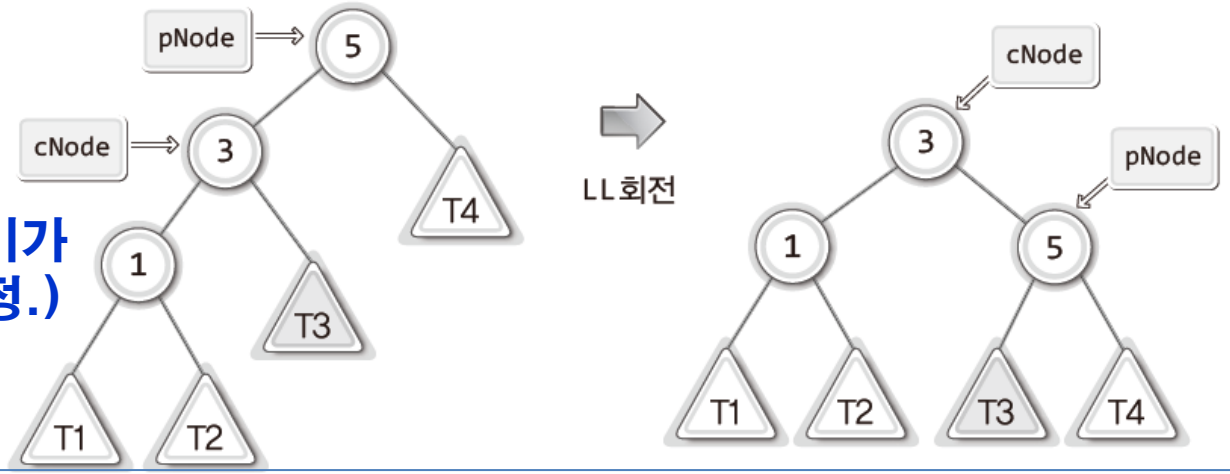
LL 상태와 LL 회전

- LL상태 : 왼쪽으로 연속 2개의 자식이 있어 균형인수 +2 출현.
- LL회전 : 왼쪽 자식을 부모 자리로 옮기고 부모는 그의 오른쪽 자식이 됨. (균형인수가 +2인 노드를 자식의 오른쪽 자식으로 만든다.)



단순화한 예

일반적인 예
(T1, T2, T3, T4 를 높이가 동일한 서브트리라고 가정.)
(T3의 움직임에 유의!)

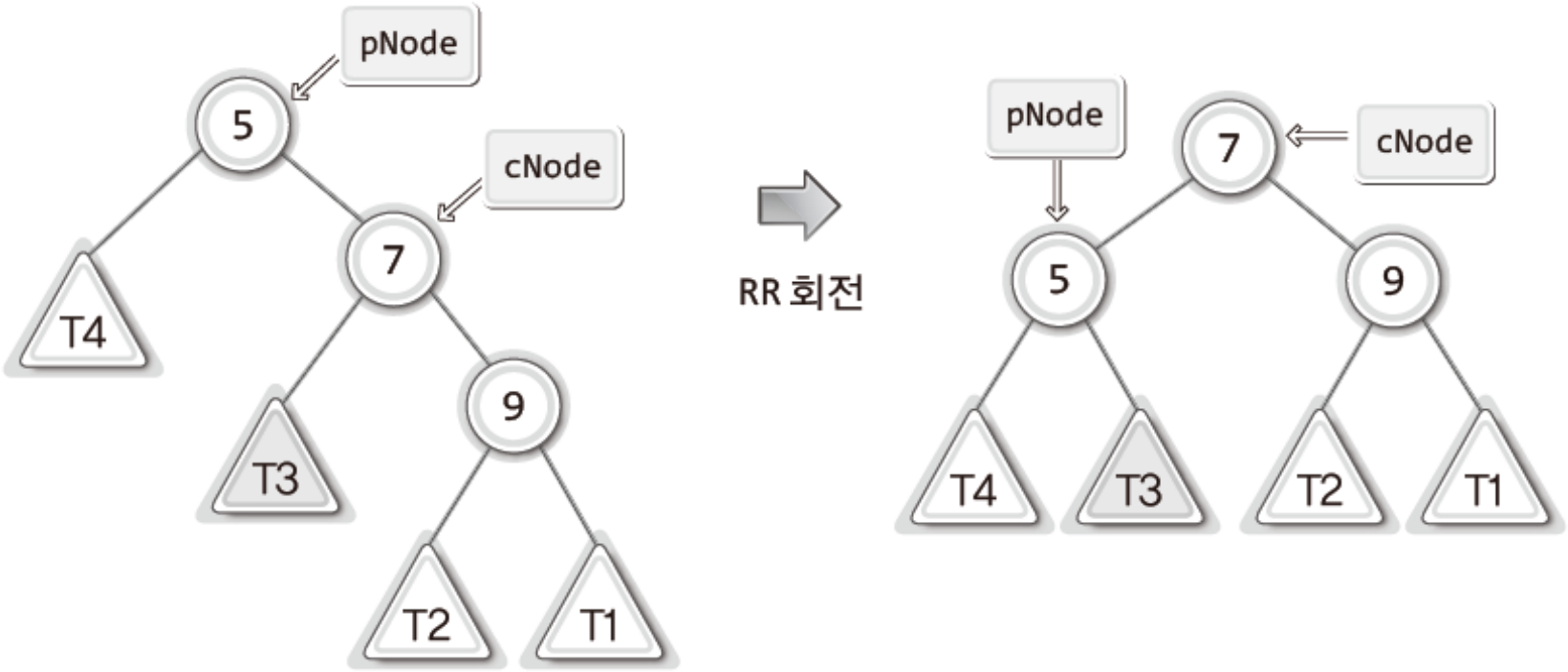


순서 중요!

```
ChangeLeftSubTree( pNode, GetRightSubTree(cNode));
ChangeRightSubTree( cNode, pNode );
```

RR 상태와 RR 회전

- RR상태 : 오른쪽으로 연속 2개의 자식이 있어 균형인수 -2 출현.
- RR회전 : 오른쪽 자식을 부모 자리로 옮기고 부모는 그의 왼쪽 자식이 됨.

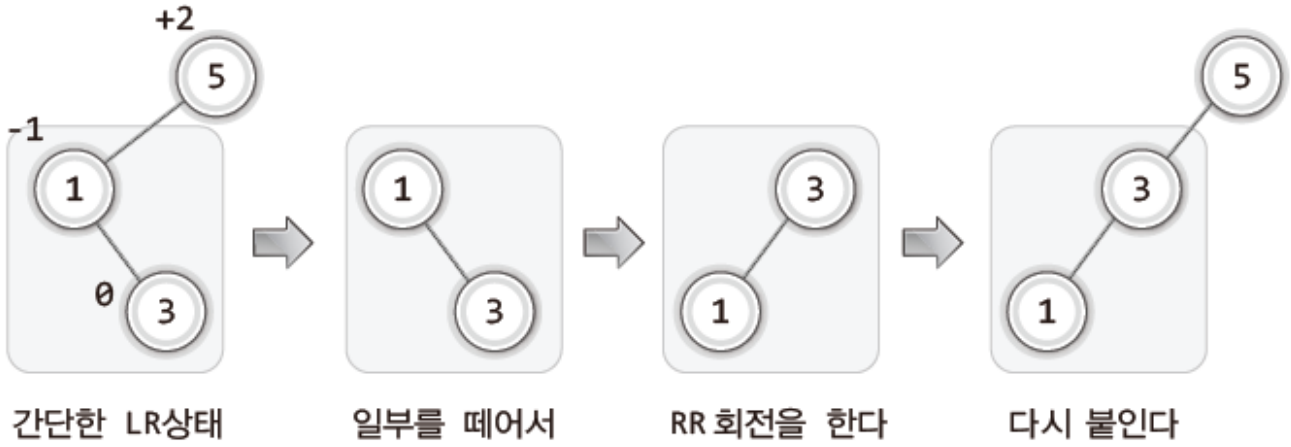


순서 중요!

```
ChangeRightSubTree( pNode, GetLeftSubTree(cNode));
ChangeLeftSubTree( cNode, pNode );
```

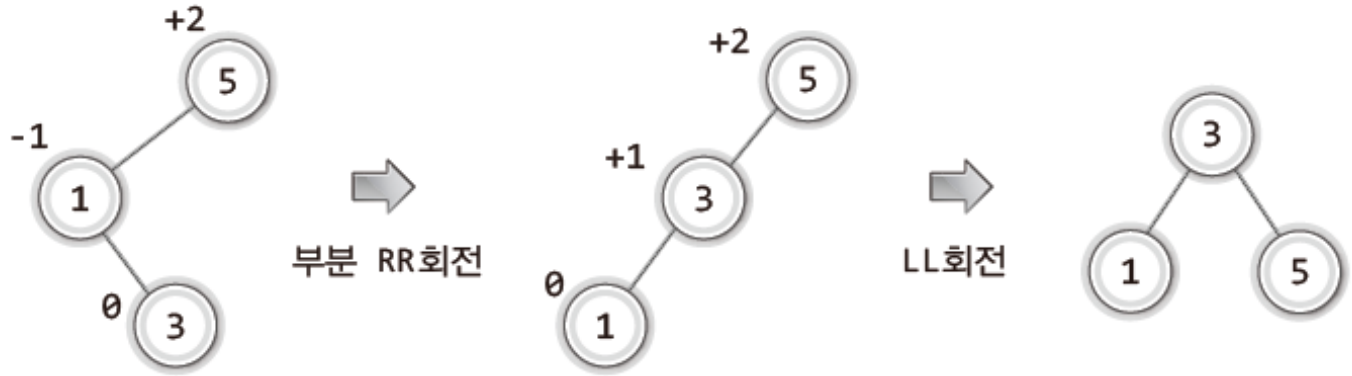
LR 상태와 LR 회전

- LR상태 : 왼쪽 자식과 그의 오른쪽 순으로 연속 2개의 추가적인 자식이 있고 균형인수 +2인 노드가 있는 상태
- LR회전 : LL 또는 RR 상태로 바꾼 후 해결.
  왼쪽 자식에 대해 RR 회전하여 LL상태로 만든 후 LL회전



LR상태를

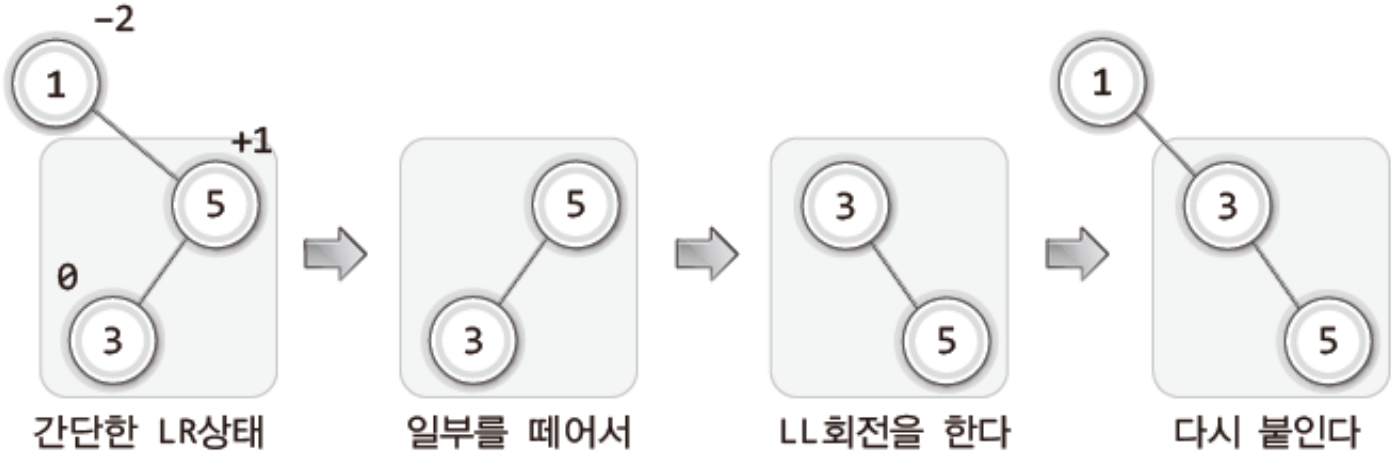
LL상태로



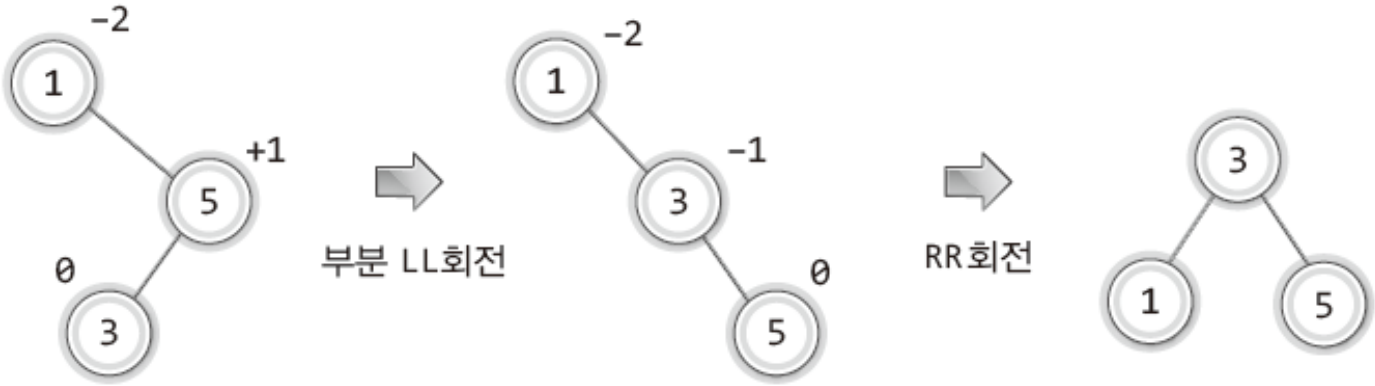
LR회전

RL 상태와 RL 회전

- RL상태 : 오른쪽 자식과 그의 왼쪽 순으로 연속 2개의 추가적인 자식이 있고 균형인수 -2인 노드가 있는 상태
- RL회전 : 오른쪽 자식에 대해 LL 회전하여 RR상태로 만든 후 RR회전



RL상태를 RR상태로



RL회전

AVL 트리 구현
(트리의 높이를 계산하여 반환)

```
int GetHeight(BTreeNode* bst) {
        int leftH;              // left height
        int rightH;             // right height

        if (bst == NULL)
                return 0;

        leftH = GetHeight(GetLeftSubTree(bst));
        rightH = GetHeight(GetRightSubTree(bst));

        // 큰 값의 높이를 반환
        if (leftH > rightH)
                return leftH + 1;
        else
                return rightH + 1;
}
```

왼쪽 서브 트리 높이 계산

오른쪽 서브 트리 높이 계산

AVL 트리 구현
(두 서브 트리의 높이의 차를 반환)

```
int GetHeightDiff(BTreeNode* bst)
{
        int lsh;    // left sub tree height
        int rsh;    // right sub tree height

        if (bst == NULL)
                return 0;

        lsh = GetHeight(GetLeftSubTree(bst));
        rsh = GetHeight(GetRightSubTree(bst));

        return lsh - rsh;
}
```

왼쪽 height가 더 크면 양수
오른쪽이 더 크면 음수.

AVL 트리 구현
(LL 회전)

```
BTreeNode* RotateLL(BTreeNode* bst)
{
	BTreeNode* pNode;
	BTreeNode* cNode;

	pNode = bst;
	cNode = GetLeftSubTree(pNode);

	ChangeLeftSubTree(pNode, GetRightSubTree(cNode));
	ChangeRightSubTree(cNode, pNode);
	return cNode;
}
```

AVL 트리 구현
(RR 회전)

```
BTreeNode* RotateRR(BTreeNode* bst)
{
	BTreeNode* pNode;
	BTreeNode* cNode;

	pNode = bst;
	cNode = GetRightSubTree(pNode);

	ChangeRightSubTree(pNode, GetLeftSubTree(cNode));
	ChangeLeftSubTree(cNode, pNode);
	return cNode;
}
```

AVL 트리 구현
(LR 회전)

```
BTreeNode* RotateLR(BTreeNode* bst)
{
        BTreeNode* pNode;
        BTreeNode* cNode;

        pNode = bst;
        cNode = GetLeftSubTree(pNode);

        ChangeLeftSubTree(pNode, RotateRR(cNode));
        return RotateLL(pNode);
}
```

부분적 RR 회전

LL 회전

AVL 트리 구현
(RL 회전)

```
BTreeNode* RotateRL(BTreeNode* bst)
{
        BTreeNode* pNode;
        BTreeNode* cNode;

        pNode = bst;
        cNode = GetRightSubTree(pNode);

        ChangeRightSubTree(pNode, RotateLL(cNode));
        return RotateRR(pNode);
}
```

부분적 LL 회전

RR 회전

AVL 트리 구현
(트리 균형 잡기)

```
BTreeNode* Rebalance(BTreeNode** pRoot) {
        int hDiff = GetHeightDiff(*pRoot);

        if (hDiff > 1) {
                if (GetHeightDiff(GetLeftSubTree(*pRoot)) > 0)
                        * pRoot = RotateLL(*pRoot);
                else
                        *pRoot = RotateLR(*pRoot);
        }

        if (hDiff < -1) {
                if (GetHeightDiff(GetRightSubTree(*pRoot)) < 0)
                        * pRoot = RotateRR(*pRoot);
                else
                        *pRoot = RotateRL(*pRoot);
        }
        return *pRoot;
}
```

왼쪽 서브 트리 방향으로 높이가 2 이상 크다면

오른쪽 서브 트리 방향으로 2 이상 크다면

AVL 트리 구현
(노드 삽입)

```
BTreeNode* BSTInsert(BTreeNode** pRoot, BSTData data) {
        if (*pRoot == NULL) {
                *pRoot = MakeBTreeNode();
                SetData(*pRoot, data);
        }
        else if (data < GetData(*pRoot)) {
                BSTInsert(&((*pRoot)->left), data);
                *pRoot = Rebalance(pRoot);
        }
        else if (data > GetData(*pRoot)) {
                BSTInsert(&((*pRoot)->right), data);
                *pRoot = Rebalance(pRoot);
        }
        else {
                return NULL; // 키의 중복을 허용하지 않는다.
        }
        return *pRoot;
}
```

BSInsert를 재귀호출:
반환되면서 각 레벨에서
Rebalance()를 호출하게 됨.