12장 탐색(Search)

```
// 프로그램 12.1: 순차 탐색
int seq_search(int key, int low, int high)
{
    int i;
    for(i = low; i <= high; i++)
       if(list[i]==key)
           return i;  // 탐색에 성공하면 키 값의 인덱스 반환

    return -1;    // 탐색에 실패하면 -1 반환
}

// 프로그램 12.2: 개선된 순차 탐색
int seq_search2(int key, int low, int high)
{
    int i;
    list[high + 1] = key; // 맨 뒤에 키를 넣는다
    for(i=low; list[i] != key; i++) // 키 값을 찾으면 종료
       ;



}

// 프로그램 12.3: 오름차순으로 정렬된 배열 리스트의 순차탐색
int sorted_seq_search(int key, int low, int high)
{
   int i;
   for(i = low; i <= high; i++){


   }
}
```

```
// 프로그램 12.4: 재귀(순환) 호출을 이용한 이진 탐색
int search_binary(int key, int low, int high)
{
    int middle;
    if (              ){
       middle = (low + high) / 2;

       if(key == list[middle])    // 탐색 성공
           return middle;
       else if(key < list[middle]) // 왼쪽 부분리스트 탐색
           return search_binary(                          );
       else                // 오른쪽 부분리스트 탐색
           return search_binary(                          );
   }
   return -1;       // 탐색 실패
}

// 프로그램 12.5: 반복을 이용한 이진 탐색
int search_binary2(int key, int low, int high)
{
    int middle;
    while









    return -1;   // 발견되지 않음
}
```

12장: 탐색

순환(재귀) revisited

0 부터 3 까지의 경로를 출력하기 위한 print_path 함수를 재귀로 작성하자.

0 을 시작점으로 하여 3 까지의 경로를 찾으려 한다면

	0	2	0	1
예: previous	[0]	[1]	[2]	[3]

경로는 0 → 2 → 1 → 3

6_4_1 (Dijkstra)

previous[ ]

재귀

```
void print_path(int start, int end)
{



}
```

탐색

Binary Search

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
10	20	25	30	40	50	60	70	80	90

mid 계산 = (low+high)/2

60 을 이진탐색으로 탐색
[0]-[9]: mid ← (0+9)/2 = [4]
40 < 60 그러므로 오른쪽을 탐색
[5]-[9]: mid ← (5+9)/2 = [7]
60 < 70 그러므로 왼쪽을 탐색
[5]-[6]: mid ← (5+6)/2 = [5]
50 < 60 그러므로 오른쪽을 탐색
[6]-[6]: mid ← (6+6)/2 = [6]
60 == 60 발견

65 을 이진탐색으로 탐색
[0]-[9]: mid ← (0+9)/2 = [4]
40 < 65 그러므로 오른쪽을 탐색
[5]-[9]: mid ← (5+9)/2 = [7]
65 < 70 그러므로 왼쪽을 탐색
[5]-[6]: mid ← (5+6)/2 = [5]
50 < 65 그러므로 오른쪽을 탐색
[6]-[6]: mid ← (6+6)/2 = [6]
60 != 65 stop: 없음!

Interpolation Search(보간 탐색)

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
10	20	25	30	40	50	60	70	80	90

loc 계산 = (key – list[low])/(list[high]-list[low]) * (high – low) + low

60 을 보간탐색으로 탐색

[0]-[9]: loc ← (60-10)/(90-10)*(9-1)+0 = [5]
50 < 60 그러므로 오른쪽을 탐색
[6]-[9]: loc ← (60-60)/(90-60)*(9-6)+6=[6]
60 == 60 발견!

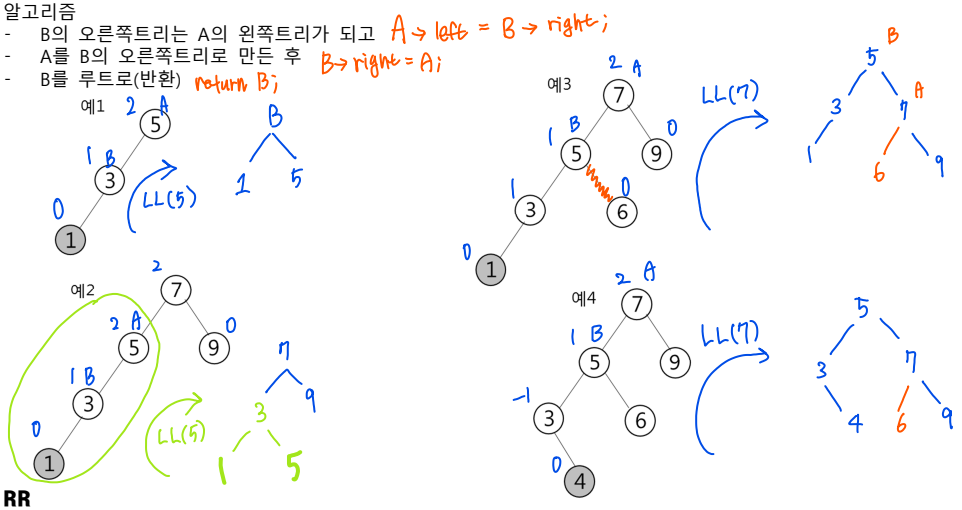
65 를 보간탐색으로 탐색

[0]-[9]: loc ← (65-10)/(90-10)*(9-1)+0 = [6]
60 < 65 그러므로 오른쪽을 탐색
[7]-[9]: list[7] > 65 이므로 stop: 없음!

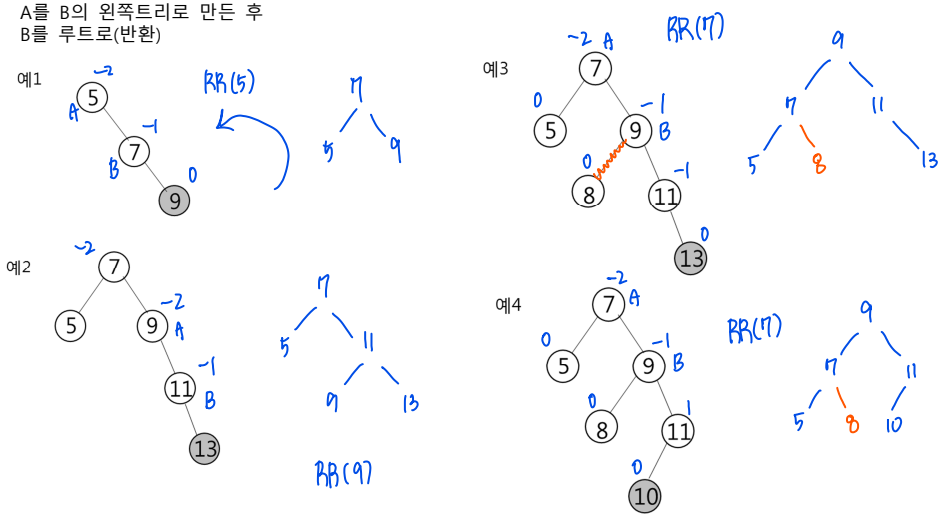
이진 탐색 트리 : 탐색, 삽입, 삭제 O(logn)

AVL: 탐색을 위한 트리
- 탐색 logn, 삽입/삭제도 logn(참고: 배열로 구현한 이진탐색은 탐색은 logn이나 삽입/삭제는 n)
- 균형잡힌 이진탐색 트리
- 균형을 잡는 방법: LL, RR, LR, RL
- 완전 트리일 필요는 없음

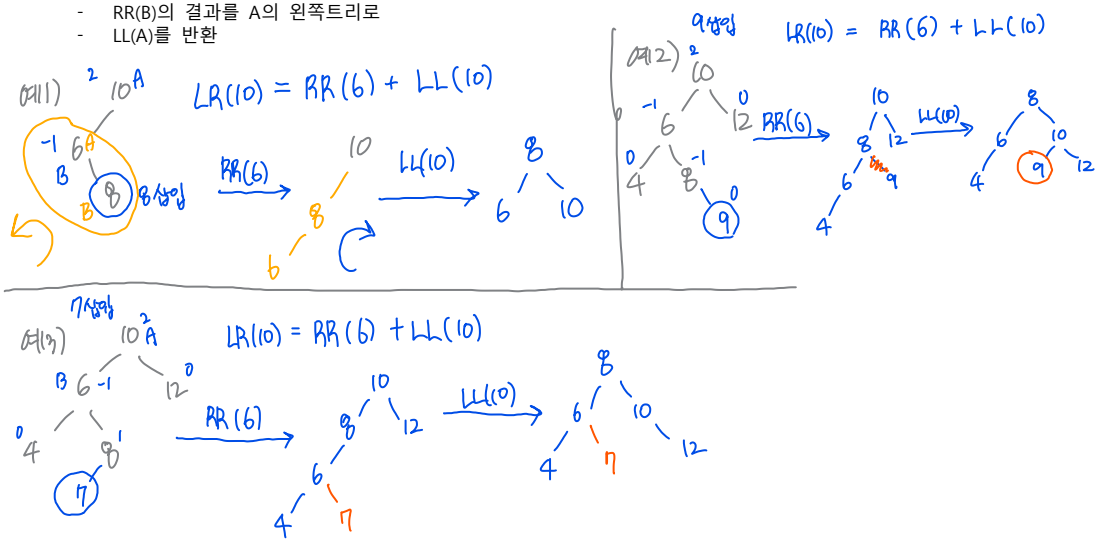
LL
A의 왼쪽트리의 왼쪽트리에 삽입했을 때(B: A의 왼쪽 자식)
A: 균형인수 2
B: 균형인수 1
이면 LL(A): A를 가지고 오른쪽 회전
알고리즘
- B의 오른쪽트리는 A의 왼쪽트리가 되고 A→left = B→right;
- A를 B의 오른쪽트리로 만든 후 B→right = A;
- B를 루트로(반환) return B;



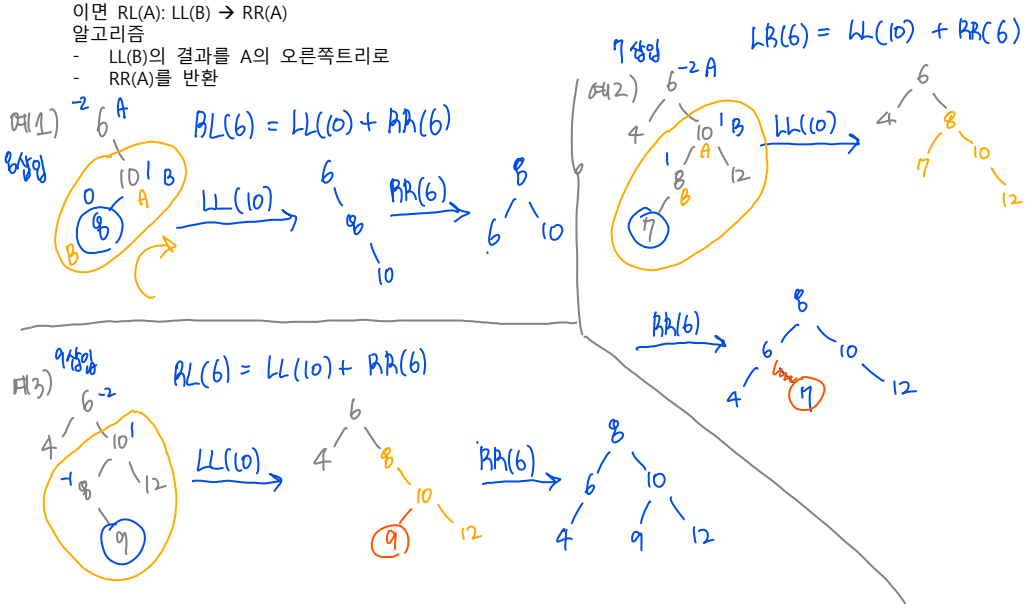
RR
A의 오른쪽 서브트리의 오른쪽 서브트리에 삽입했을 때.(B: A의 오른쪽 자식)
A: 균형인수 -2
B: 균형인수 -1
이면 RR(A): A를 가지고 왼쪽 회전
알고리즘
- B의 왼쪽트리는 A의 오른쪽트리가 되고
- A를 B의 왼쪽트리로 만든 후
- B를 루트로(반환)



LR(RR → LL)
A의 왼쪽트리의 오른쪽트리에 삽입했을 때.(B: A의 왼쪽 자식)
A: 균형인수 2
B: 균형인수 -1
이면 LR(A): RR(B) → LL(A)
알고리즘
- RR(B)의 결과를 A의 왼쪽트리로
- LL(A)를 반환



RL(LL → RR)
A의 오른쪽트리의 왼쪽트리에 삽입했을 때.(B: A의 오른쪽 자식)
A: 균형인수 -2
B: 균형인수 1
이면 RL(A): LL(B) → RR(A)
알고리즘
- LL(B)의 결과를 A의 오른쪽트리로
- RR(A)를 반환



12장 탐색(AVL트리)

```c
#include<stdio.h>
#include<stdlib.h>
#define MAX(a, b) (a)
// AVL 트리 노드 정의
typedef struct AVLNode
{
    int key;
    struct AVLNode *left;
    struct AVLNode *right;
} AVLNode;

// 트리의 높이를 반환
int get_height(AVLNode *node)
{
    int height = 0;

    if (node != NULL)
        height = 1 + max(get_height(node->left),
            get_height(node->right));

    return height;
}
// 노드의 균형인수를 반환
int get_balance(AVLNode* node)
{
    if (node == NULL) return 0;

    return get_height(node->left)
        - get_height(node->right);
}

// 노드를 동적으로 생성하는 함수
AVLNode* create_node(int key)
{
    AVLNode* node = (AVLNode*)malloc(sizeof(AVLNode));
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    return(node);
}

// 오른쪽으로 회전시키는 함수
AVLNode *rotate_right(AVLNode *parent)
{
    AVLNode* child = parent->left;
    parent->left = child->right;
    child->right = parent;

    // 새로운 루트를 반환
    return child;
}

// 왼쪽으로 회전시키는 함수
AVLNode *rotate_left(AVLNode *parent)
{
    AVLNode *child = parent->right;
    parent->right = child->left;
    child->left = parent;

    // 새로운 루트 반환
    return child;
}

// AVL 트리에 새로운 노드 추가 함수
// 새로운 루트를 반환한다.
AVLNode* insert(AVLNode* node, int key)
{
    // 이진 탐색 트리의 노드 추가 수행
    if (node == NULL)
        return(create_node(key));

    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else // 동일한 키는 허용되지 않음
        return node;

    // 노드들의 균형인수 재계산
    int balance = get_balance(node);

    // LL 타입 처리
    if (balance > 1 && key < node->left->key)
        return rotate_right(node);

    // RR 타입 처리
    if (balance < -1 && key > node->right->key)
        return rotate_left(node);

    // LR 타입 처리
    if (balance > 1 && key > node->left->key)
    {
        node->left = rotate_right(node->left);
        return rotate_right(node);
    }

    // RL 타입 처리
    if (balance < -1 && key < node->right->key)
    {
        node->right = rotate_right(node->right);
        return rotate_left(node);
    }
    return node;
}

// 전위 순회 함수
void preorder(AVLNode *root)
{
    if (root != NULL)
    {
        printf("[%d] ", root->key);
        preorder(root->left);
        preorder(root->right);
    }
}

int main(void)
{
    AVLNode *root = NULL;

    // 예제 트리 구축
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 29);

    printf("전위 순회 결과 \n");
    preorder(root);

    return 0;
}
```