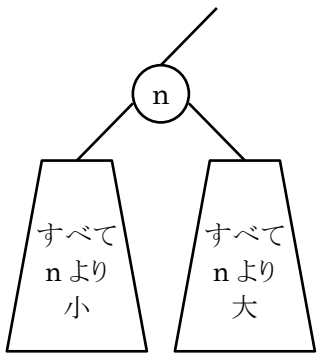
1. **2 分探索木**は，右図に示すように，その中のどのノード n をとっても，その左の部分木には n より小さい要素，右の部分木には n より大きい要素しかないような 2 分木である。このデータ構造は文字通り探索に適する。以下のプログラムは 2 分探索木を構築して，そこから指定のキーを持つノードを探索するものである。空欄を適切に埋めて実行し，動作を理解せよ。

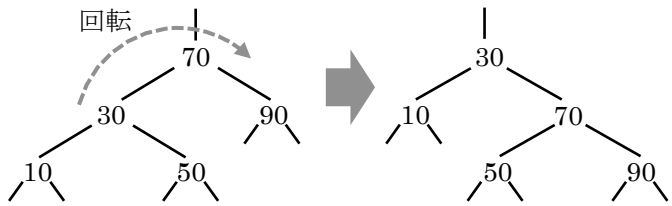



```
#include <stdio.h>
#include <stdlib.h>

struct node {
    int key;
    int data;
    struct node *left;
    struct node *right;
};

struct node *root = NULL;

struct node* make_node(int key) {
    struct node *p;
    p = (struct node *)
        malloc(sizeof(struct node));
    p->key = key;
    p->left = p->right = NULL;
    p->data = 0;
    return p;
}

/* キーを指定してノードを挿入する */
struct node *insert(int key) {
    struct node *p;

    if (root == NULL) {
        root = make_node(key);
        return root;
    }

    /* 挿入位置の探索：ループによる探索方法 */
    p = root;
    while (1) {
        if (key < p->key) {

            if (              == NULL) {

                p->left =
                return p->left;
            }


        } else if (key > p->key) {

            if (               == NULL) {

                p->right =
                return p->right;
            }


        } else {
            return NULL;
        }
    }
}
```

```
/* キーの探索：再帰による探索方法 */
struct node*
search(struct node* p, int key) {
    if (p == NULL) return NULL;

    if (key < p->key)

        return search(              , key);

    if (key > p->key)

        return search(              , key);

    return
}

/* 再帰で部分木の要素を昇順に表示する */
void traverse(struct node *p) {
    if (p == NULL) return;









}

int main(void) {
    int key;
    struct node *p;

    /* 木の構築 */
    while (1) {
        printf("insert number > ");
        scanf("%d", &key);
        if (key < 0) break;
        insert(key);
        traverse(root);
        printf("¥n");
    }

    /* 探索 */
    printf("search number > ");
    scanf("%d", &key);
    p = search(root, key);
    if (p != NULL) printf("Found.¥n");
    else           printf("Not found.¥n");
    return 0;
}
```

2. 単純な 2 分探索木は，データの登録順によっては木の形が左右対称から大きく崩れ，探索速度が低下する。**平衡木**（AVL 木）では，どのノードから見た左右の部分木の高さの差も 1 以内に収まるように，下図に示すような回転操作を利用してなるべく平衡を保つ。

【発展】下記のプログラムに適切な main 関数等を補って実行し（insert(&root, key)でデータ登録），平衡木のしくみと回転操作について理解せよ。




```
#include <stdio.h>
#include <stdlib.h>

typedef struct node {
    int key;
    int data;
    struct node *left;
    struct node *right;
    int depth;  /* この下の部分木の高さ */
} Node;

/* NULL の代わりに「空」を表すノード nil を用いる
   これによっていくつかの条件判断が省略できる */
Node nil = { 0, 0, &nil, &nil, 0 };
#define NIL (&nil)

Node *root = NIL;

Node *make_node(int key) {
    Node *p = (Node *) malloc(sizeof(Node));
    p->key = key;
    p->data = 0;
    p->left = p->right = NIL;
    p->depth = 0;
    return p;
}

/* ノードの depth を計算し直す */
void fix_depth(Node *p) {
    if (p->left->depth > p->right->depth)
        p->depth = p->left->depth + 1;
    else
        p->depth = p->right->depth + 1;
}

/* 右の子ノードを部分木の新しい根にする回転操作：
   親が持つ部分木の根へのポインタを変更するので，
   そのアドレス（ポインタのポインタ）を引数にする */
void rotate_left(Node **link) {
    Node *p = *link;
    Node *q = p->right;
    p->right = q->left;
    q->left = p;
    *link = q;
    fix_depth(p); fix_depth(q);
}

/* 左の子ノードを部分木の新しい根にする回転操作 */
void rotate_right(Node **link) {
    Node *p = *link;
    Node *q = p->left;
    p->left = q->right;
    q->right = p;
    *link = q;
    fix_depth(p); fix_depth(q);
}

/* 再帰による部分木への新しいノードの挿入：
   回転を行う場合，親が持つ部分木の根へのポインタを
   変更するので，ポインタのポインタを引数にする */
Node *insert(Node **link, int key) {
    struct node *p, *q;
    struct node *newnode = NIL;
    int balance;

    /* 現在位置が葉（NIL）ならそこにノードを挿入し，
       その親ノードが持つリンク先を書き換える */
    if (*link == NIL) {
        *link = make_node(key);
        return *link;
    }

    /* 1.の方法とは異なり，再帰によってリンクを
       たどり，挿入位置（NIL）を探していく */
    p = *link;
    if (key < p->key)
        newnode = insert(&(p->left), key);
    else if (key > p->key)
        newnode = insert(&(p->right), key);
    if (newnode == NIL)
        return NIL;  /* 既に登録済みの場合 */

    /* 回転操作を使って左右のバランスを維持する
       詳しい説明は教科書等を参照 */
    balance = p->left->depth - p->right->depth;
    if (balance >= 2) {
        /* 左の子の下が深すぎる場合 */
        q = p->left;
        if (q->left->depth < q->right->depth) {
            /* その右の孫の下が深いなら二重回転 */
            rotate_left(&(p->left));
        }
        rotate_right(link);
    } else if (balance <= -2) {
        /* 右の子の下が深すぎる場合 */
        q = p->right;
        if (q->left->depth > q->right->depth) {
            /* その左の孫の下が深いなら二重回転 */
            rotate_right(&(p->right));
        }
        rotate_left(link);
    } else {
        fix_depth(p);
    }
    return newnode;
}
```