4.
```
#include <stdio.h>
#include <malloc.h>

#define EMPTY 0

typedef struct node_tree
{
   int data;
   struct node_tree* left;
   struct node_tree* right;
} tree;

tree *get_node()
{
   tree *tmp;
   tmp = (tree *)malloc(sizeof(tree));
   tmp -> left = EMPTY;
   tmp -> right = EMPTY;
   return tmp;
}

void tree_ins(tree **root, int data)
{
   if(*root == EMPTY)
   {
      *root = get_node();
      (*root) -> data = data;
      return ;
   }
   else if(data < (*root) -> data)
     tree_ins(&(*root) -> left, data);
   else if(data > (*root) -> data)
     tree_ins(&(*root) -> right, data);
}

int print_tree(tree *root)
{
   if(root)
   {
     printf("%d\n", root -> data);
     print_tree(root -> left);
     print_tree(root -> right);
   }
   return 0;
}

tree *chg_node(tree *root)
{
    if(!root -> left)
      root = root -> right;
    else if(!root -> right)
```

```
      root = root -> left;

    return root;
}

tree *find_max(tree *root, int *data)
{
    if(root -> right)
      root -> right = find_max(root -> right, data);
    else
    {
      *data = root -> data;
      root = chg_node(root);
    }
    return root;
}

tree *detree(tree *root, int data)
{
    int num;
    if(root == EMPTY)
    {
           printf("Not found\n");
           return 0;
    }
    else if(data < root -> data)
      root -> left = detree(root -> left, data);
    else if(data > root -> data)
      root -> right = detree(root -> right, data);
    else if(root -> left && root -> right)
    {
        root -> left = find_max(root -> left, &num);
        root -> data = num;
    }
    else
     root = chg_node(root);


    return root;
}


int main(void)
{
    tree *root = EMPTY;
    int i;
    int arr[10] = {50, 30, 40, 60, 10, 33, 44, 45, 62, 25};
    for(i = 0; i < 10; i++)
    {
            tree_ins(&root, arr[i]);
    }
    print_tree(root);
```

```
    root = detree(root, 30);

    printf("Now U delete, 30\n");
    print_tree(root);
    return 0;
}
```

6.

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

#define EMPTY 0

typedef enum _rot
{
   RR,
   RL,
   LL,
   LR
} rot;


typedef struct avl_node
{
   int data;
   int lev;
   struct avl_node *left;
   struct avl_node *right;
} avl;

avl *get_node()
{
   avl *tmp;
   tmp = (avl *)malloc(sizeof(avl));
   tmp -> left = EMPTY;
   tmp -> right = EMPTY;
   tmp -> lev = 1;
   return tmp;
}

int update_level(avl *root)
{
     int left = root -> left ? root -> left -> lev :0;
     int right = root -> right ? root -> right -> lev :0;

     if(left > right)
       return left +1;

     return right + 1;
}
```

```
int rotation_check(avl *root)
{
    int left = root -> left ? root -> left -> lev :0;
    int right = root -> right ? root -> right -> lev :0;

    return right - left;
}

int kinds_of_rot(avl *root, int data)
{
   if(rotation_check(root) > 1)
   {
        if(root -> right -> data > data)
              return RL;
        return RR;
   }
   else if(rotation_check(root) < -1)
   {
        if(root -> left -> data > data)
              return LL;
        return LR;
   }
}

avl *rr_rot(avl *parent, avl *child)
{
    parent -> right = child -> left;
    child -> left = parent;
    parent -> lev = update_level(parent);
    child -> lev = update_level(child);
    return child;
}

avl *ll_rot(avl *parent, avl *child)
{
    parent -> left = child -> right;
    child -> right = parent;
    parent -> lev = update_level(parent);
    child -> lev = update_level(child);
    return child;
}

avl *rl_rot(avl *parent, avl *child)
{
    child = ll_rot(child, child -> left);
    return rr_rot(parent, child);
}

avl *lr_rot(avl *parent, avl *child)
{
    child = rr_rot(child, child -> right);
```

```
    return ll_rot(parent, child);
}

avl *rotation(avl *root, int ret)
{
    switch(ret)
    {
        case RR:
            printf("RR Rotation\n");
            return rr_rot(root, root -> right);
        case RL:
            printf("RL Rotation\n");
            return rl_rot(root, root -> right);
        case LL:
            printf("LL Rotation\n");
            return ll_rot(root, root -> left);
        case LR:
            printf("LR Rotation\n");
            return lr_rot(root, root -> left);
    }
}

void avl_ins(avl **root, int data)
{
    if(*root == EMPTY)
    {
        *root = get_node();
        (*root) -> data = data;
        return ;
    }
    else if(data < (*root) -> data)
      avl_ins(&(*root) -> left, data);
    else if(data > (*root) -> data)
      avl_ins(&(*root) -> right, data);

    (*root) -> lev = update_level(*root);

    if(abs(rotation_check(*root)) > 1)
    {
        printf("Insert Rotation, %d\n", data);
        *root = rotation(*root, kinds_of_rot(*root, data));
    }
}

int print_tree(avl *root)
{
    if(root)
    {
        printf("date = %d, level = %d\n", root -> data, root -> lev);
        print_tree(root -> left);
        print_tree(root -> right);
    }
```

```
    return 0;
}

int main(void)
{
    avl *root = EMPTY;
    int i;
    int arr[10] = {50, 30, 60, 40, 44, 55, 54, 25, 59, 11};
    for(i = 0; i < 10; i++)
    {
        avl_ins(&root, arr[i]);
    }
    print_tree(root);

    return 0;
}
```

7.

AVL tree 는 완벽한 이진트리의 형태를 띠고있다. 그리하여 원하는 자료를 검색하는 속도가 매우 빠르다. 하지만 입출력시 빈번하게 일어나는 회전으로 인하여 입,출력의 속도는 낮아진다. 그에 반하여 RB tree 는 완벽한 이진트리의 형태가 아니여서 자료를 검색하는 속도는 상대적으로 낮지만, 입출력시 회전이 빈번하게 일어나지 않아 입출력의 속도 또한 빠르다.