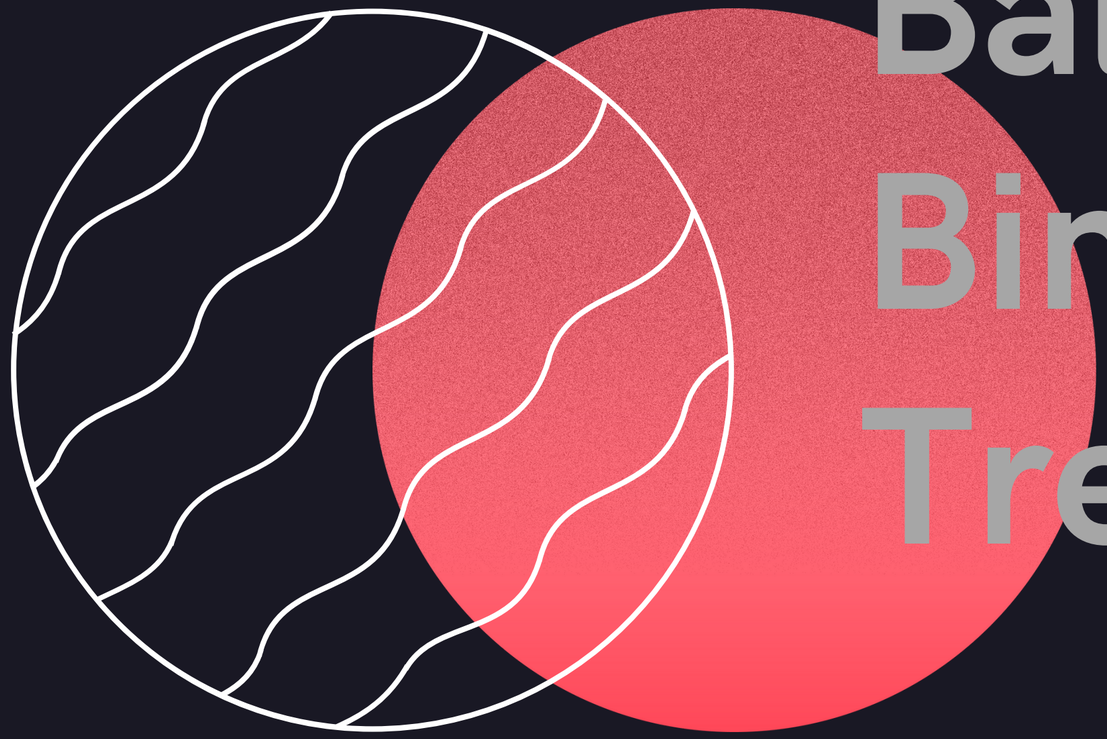
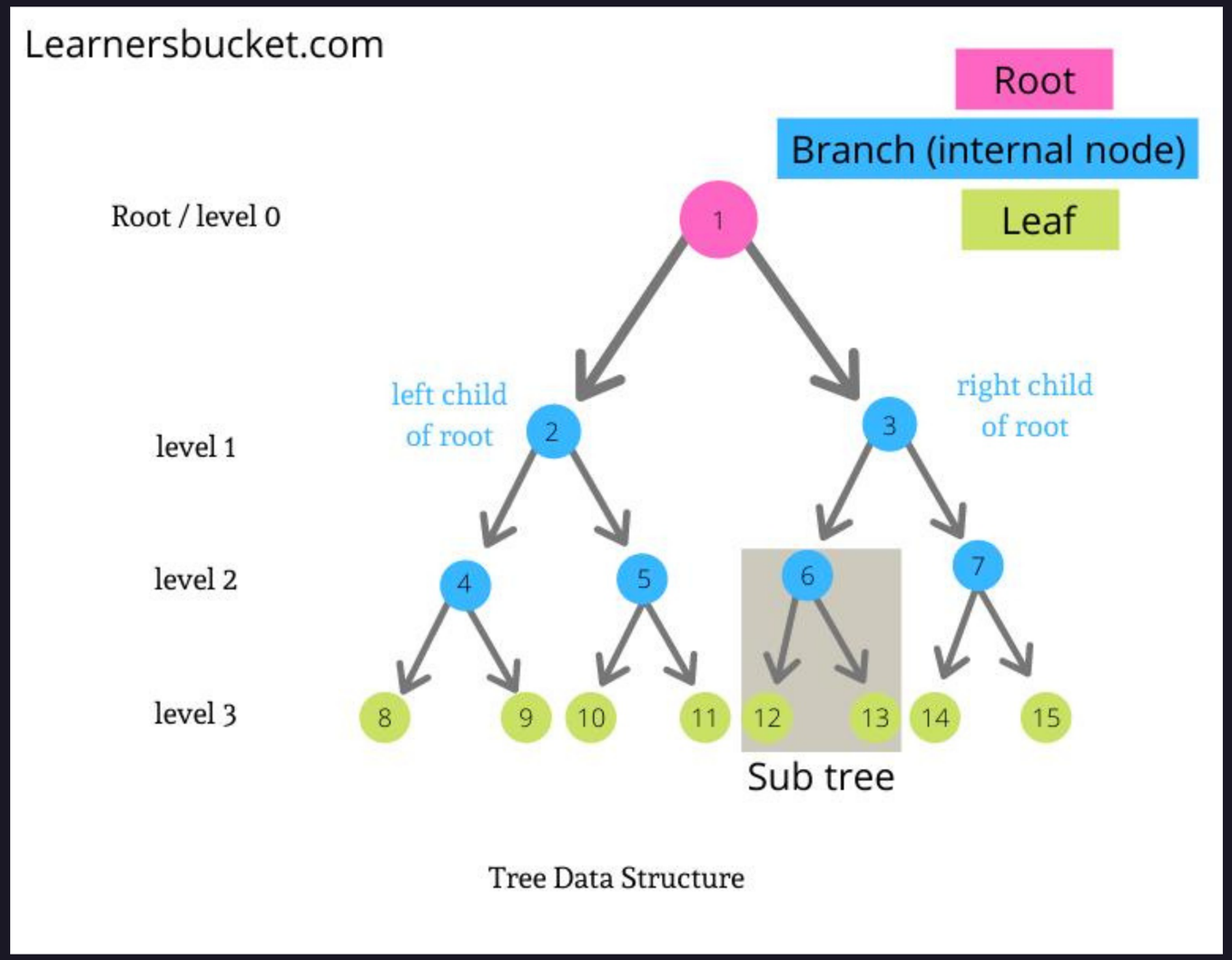


Balanced Binary Search Tree Project

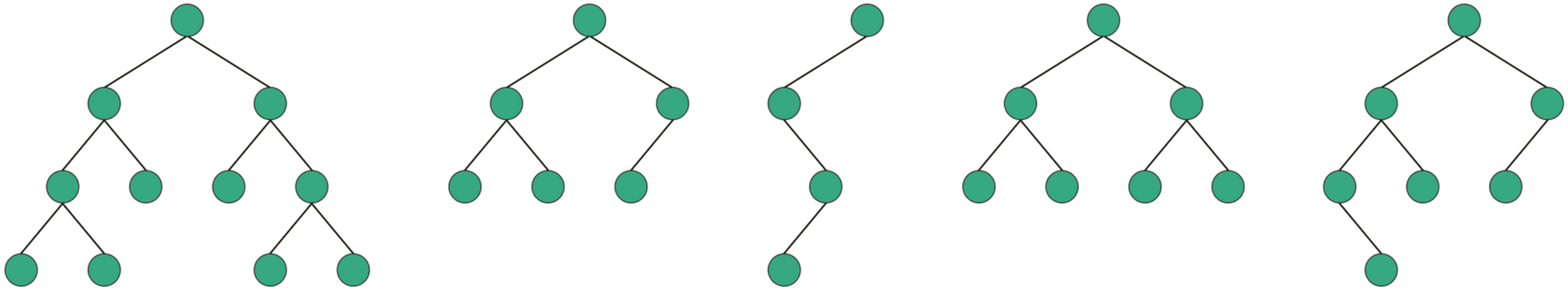
ENGS115 Data Structure and Algorithms
Mariam Mkrtichyan, Maria Apitonian

Binary tree



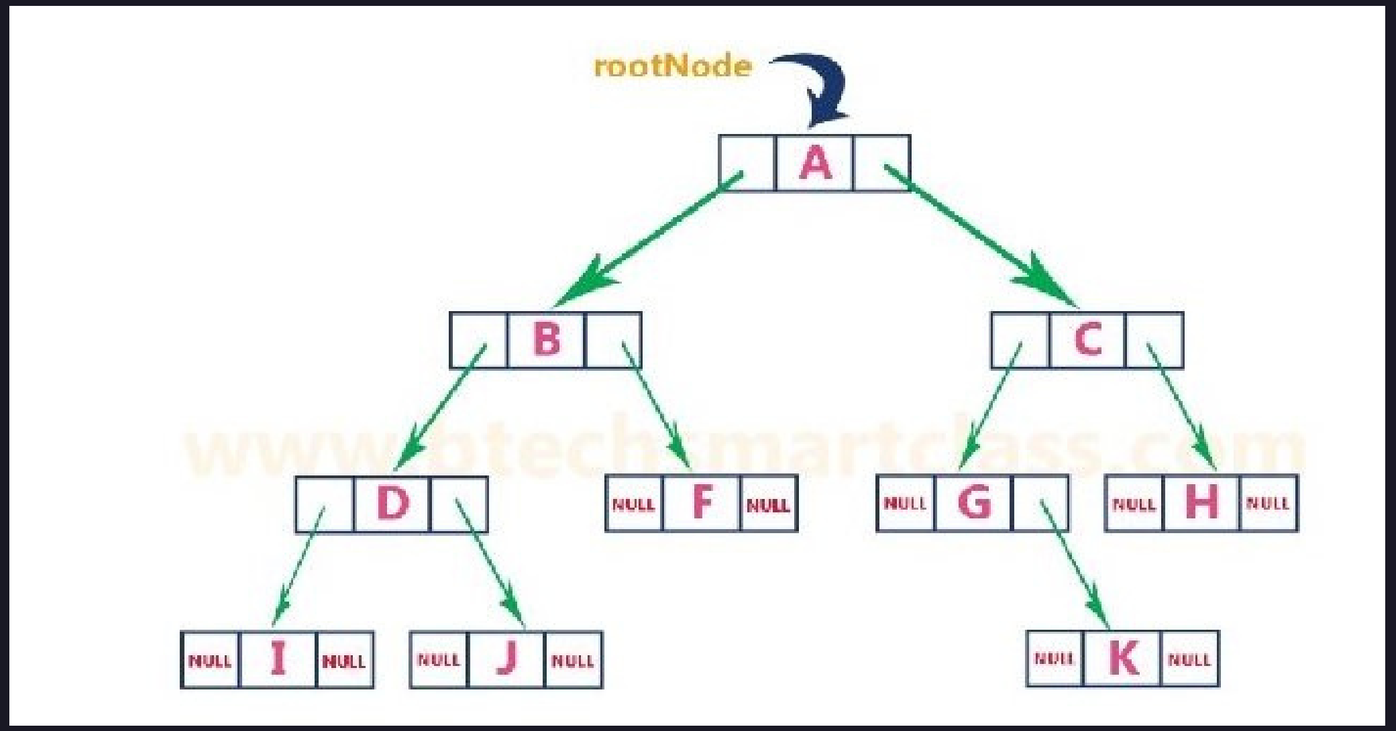
A hierarchical data structure with a set of linked nodes. Trees have a starting value (root) and subtrees of children with linked nodes to their parents. Each parent node has a greater key than the child node below.

Binary tree types
Balanced binary tree



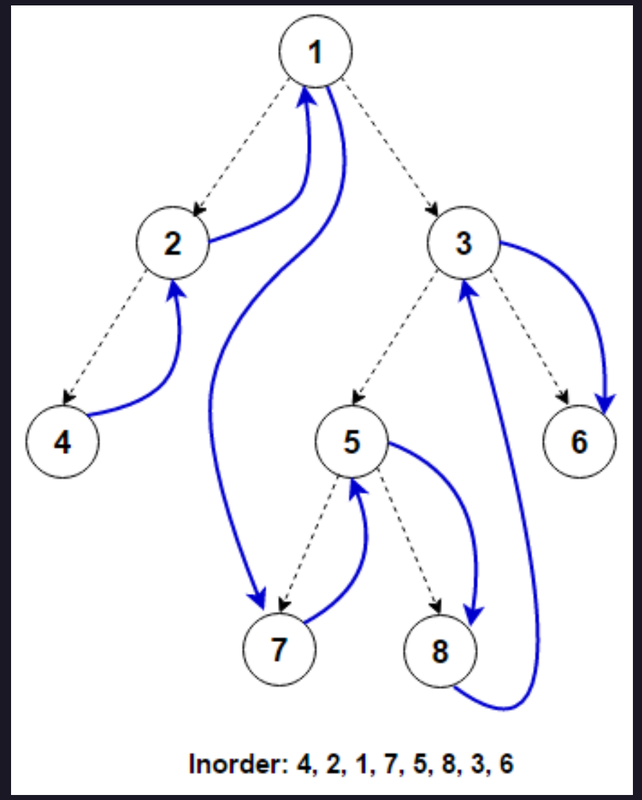
Full | Complete | Degenerate | Perfect | Balanced

Double linked list for the binary tree structure implementation




```c
struct node
{
    element_type key;
    struct node* left;
    struct node* right;
};

struct tree
{
    size_type n;
    struct node* start;
};
```

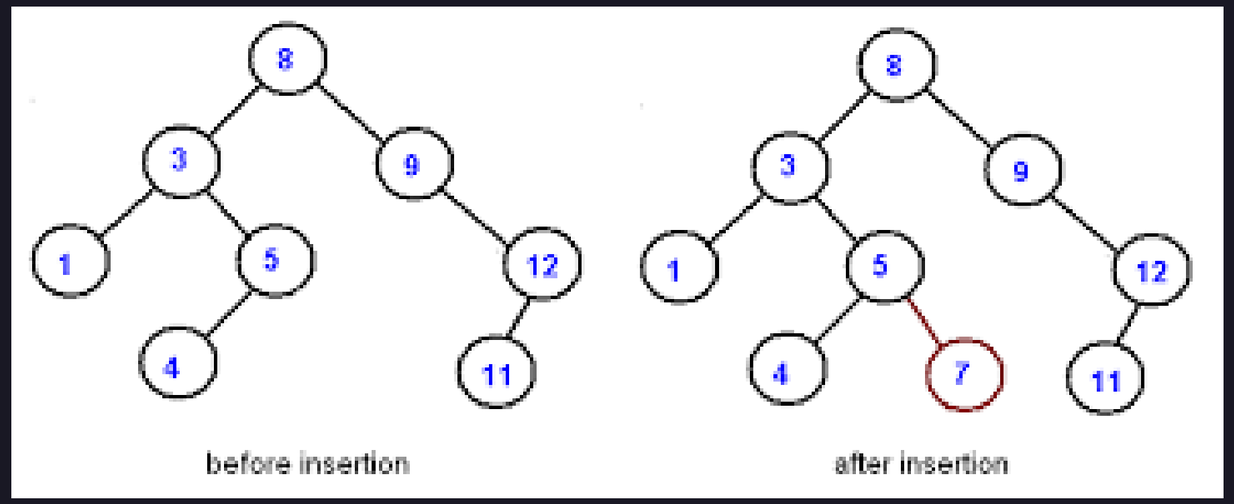
Inorder traversal



Tree traversal - is the process of visiting each node in the tree exactly once. Visiting each node in a graph should be done in a systematic manner (inorder, postorder, preorder).

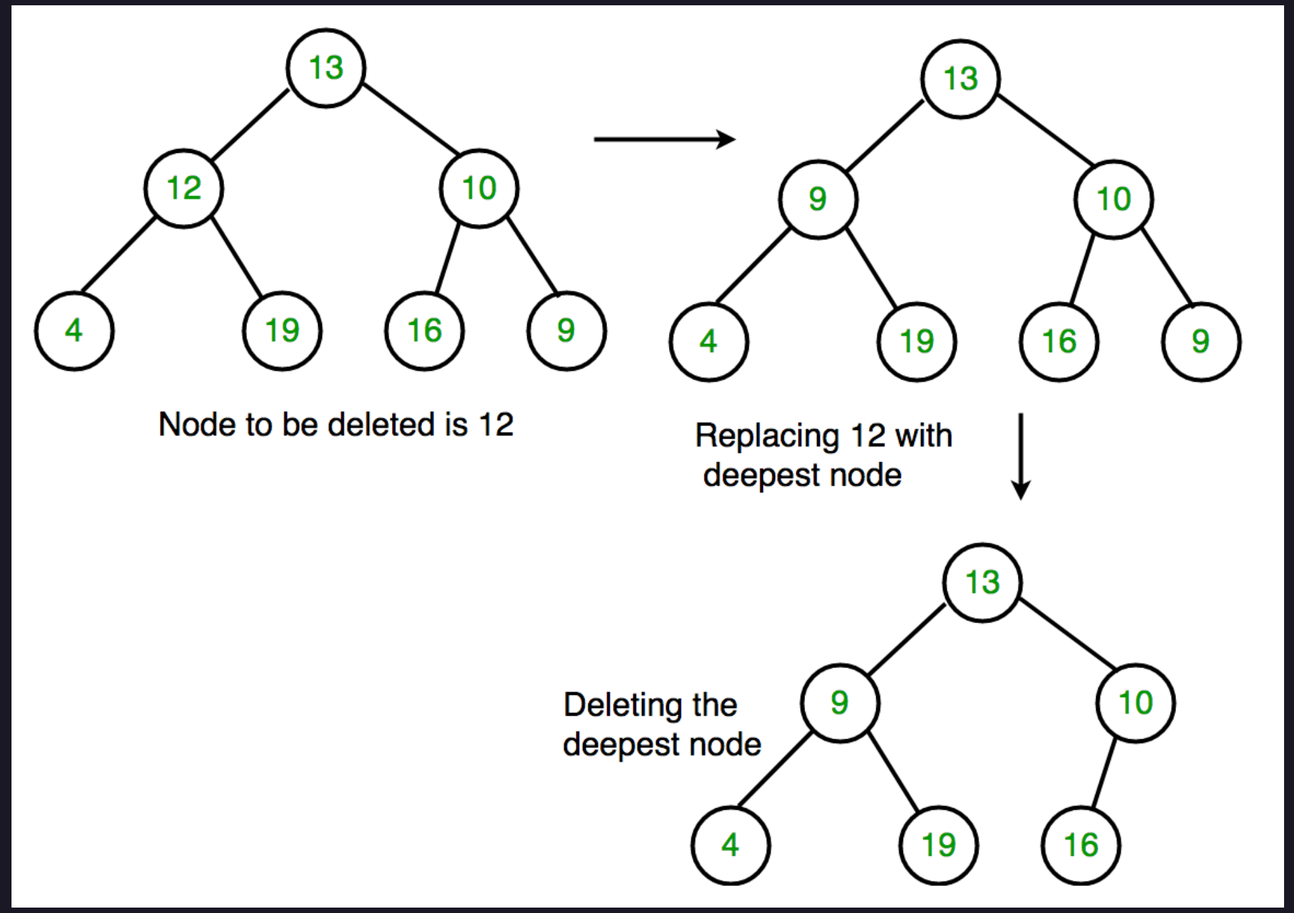
```
void inorder_helper (struct node* n ) //(struct node* n)
/*inorder tree traversal*/
{
    printf("print tree \n");
    assert(NULL != n);
    if(n != NULL ){
        inorder_helper(n->left);
        printf("%d \n",n->key);
        inorder_helper(n->right);
    }
}

void tree_inorder(struct tree* t)
{
    assert (NULL !=t);
    inorder_helper(t->start);
}
```

Node insert




```c
int tree_insert(struct tree* t, element_type value)
{
        if(search_value(t, value) == 1){
                printf("Value cannot be inserted as it already exists.");
                return 0;
        }

        struct node* insertpos = t->start;
        struct node* newnode = node_create(value, NULL, NULL);

        for(int i=0; i<tree_height(t->start); i++) {
                if(insertpos->key > value){
                        if(insertpos->right==NULL){
                                insertpos->right==newnode;
                                break;
                        }
                        else{
                                insertpos=insertpos->right;
                        }
                }
                else if(insertpos->key < value){
                        if(insertpos->left==NULL){
                                insertpos->left==newnode;
                                break;
                        }
                        else{
                                insertpos=insertpos->left;
                        }
                }
        }
        rebalance(t);
        return 1;
}
```

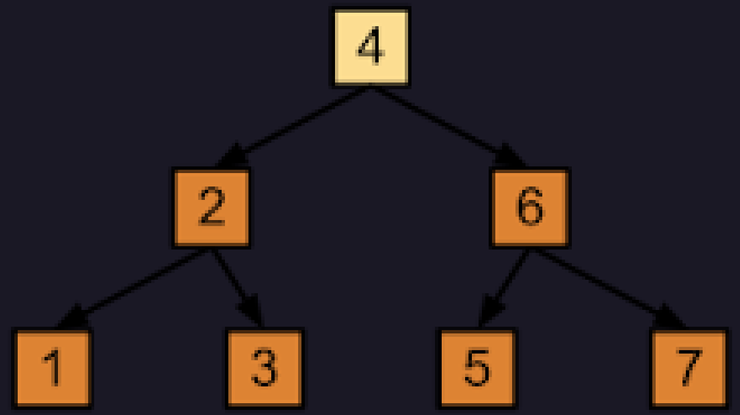
Node Delete



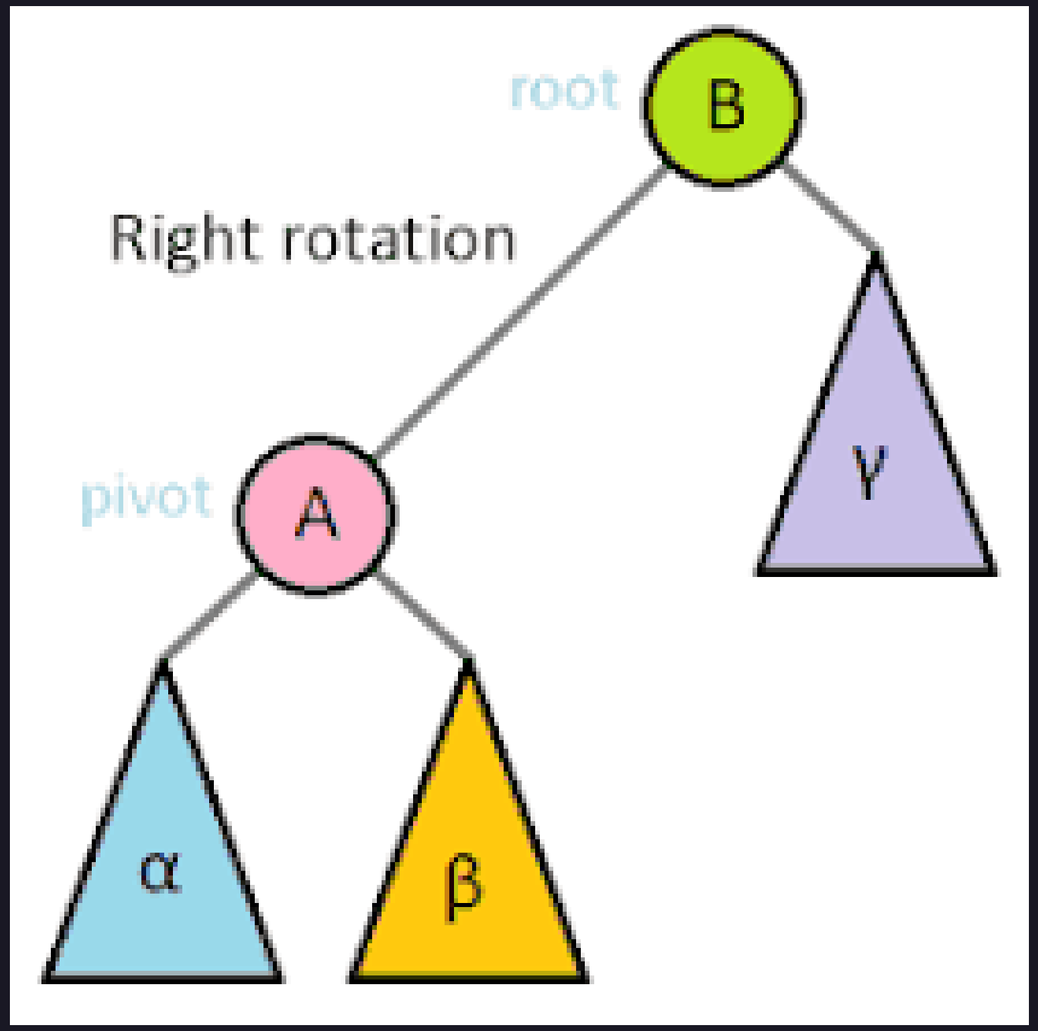

```c
int tree_delete(struct tree* t, element_type value)
{
        if(search_value(t, value) == false){
                printf("The value cannot be removed as it does not exist");
                return 0;
        }

        if(path[0] == "root"){
                int hl = tree_height(t->start->left);
                int hr = tree_height(t->start->right);
                struct node* newrootparent = NULL;
                if(hr>hl){
                        struct node* newroot = tree_minimum(t->start->right);
                        struct node* newrootparent = find_parent(t, newroot->key);
                        newrootparent->left = NULL;
                        struct node* leftsubtree = t->start->left;
                        struct node* rightsubtree = t->start->right;
                        t->start = newroot;
                        newroot->right = rightsubtree;
                        newroot->left = leftsubtree;
                }
                else{
                        struct node* newroot = tree_maximum(t->start->left);
                        struct node* newrootparent = find_parent(t, newroot->key);
                        newrootparent->right = NULL;
                        struct node* leftsubtree = t->start->left;
                        struct node* rightsubtree = t->start->right;
                        t->start = newroot;
                        newroot->right = rightsubtree;
                        newroot->left = leftsubtree;
                }
                rebalance(t);//root replaced, old leaf deleted, rebalanced
                return 1;
        }
```

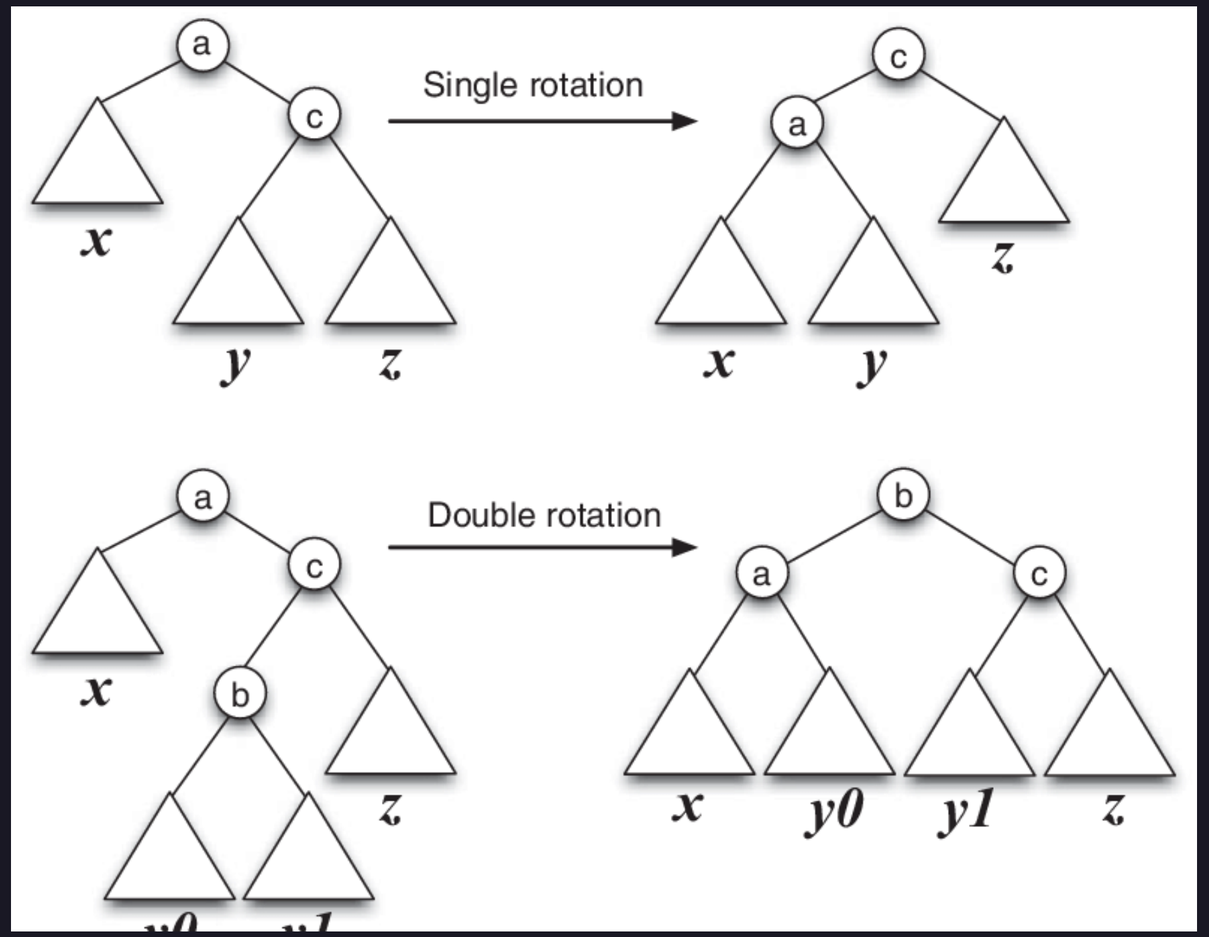
Non-Balanced vs. Balanced

Built from the sequence [1,2,3,4,5,6,7]



Right / Left Rotation





Single rotation left/right

```
void single_rotation_rhelper(struct node* n)
/* singleRotateRight(T) performs a singlerotation from left to right at the root of T*/
{
    struct node* l  = n->left;
    n->left  = l->right;
    l->right = l;
    n = l;
}
```

```
void single_rotation_rhelper(struct node* n)
/* singleRotateRight(T) performs a singlerotation from left to right at the root of T*/
{
    struct node* l  = n->left;
    n->left  = l->right;
    l->right = l;
    n = l;
}
```