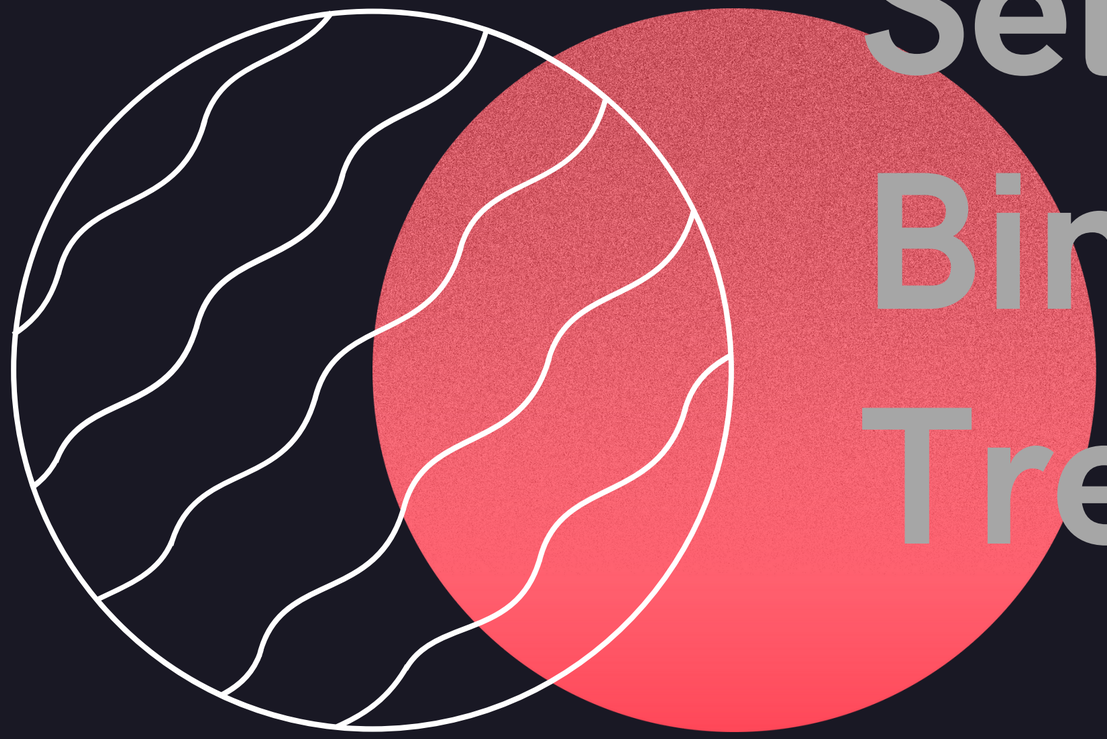
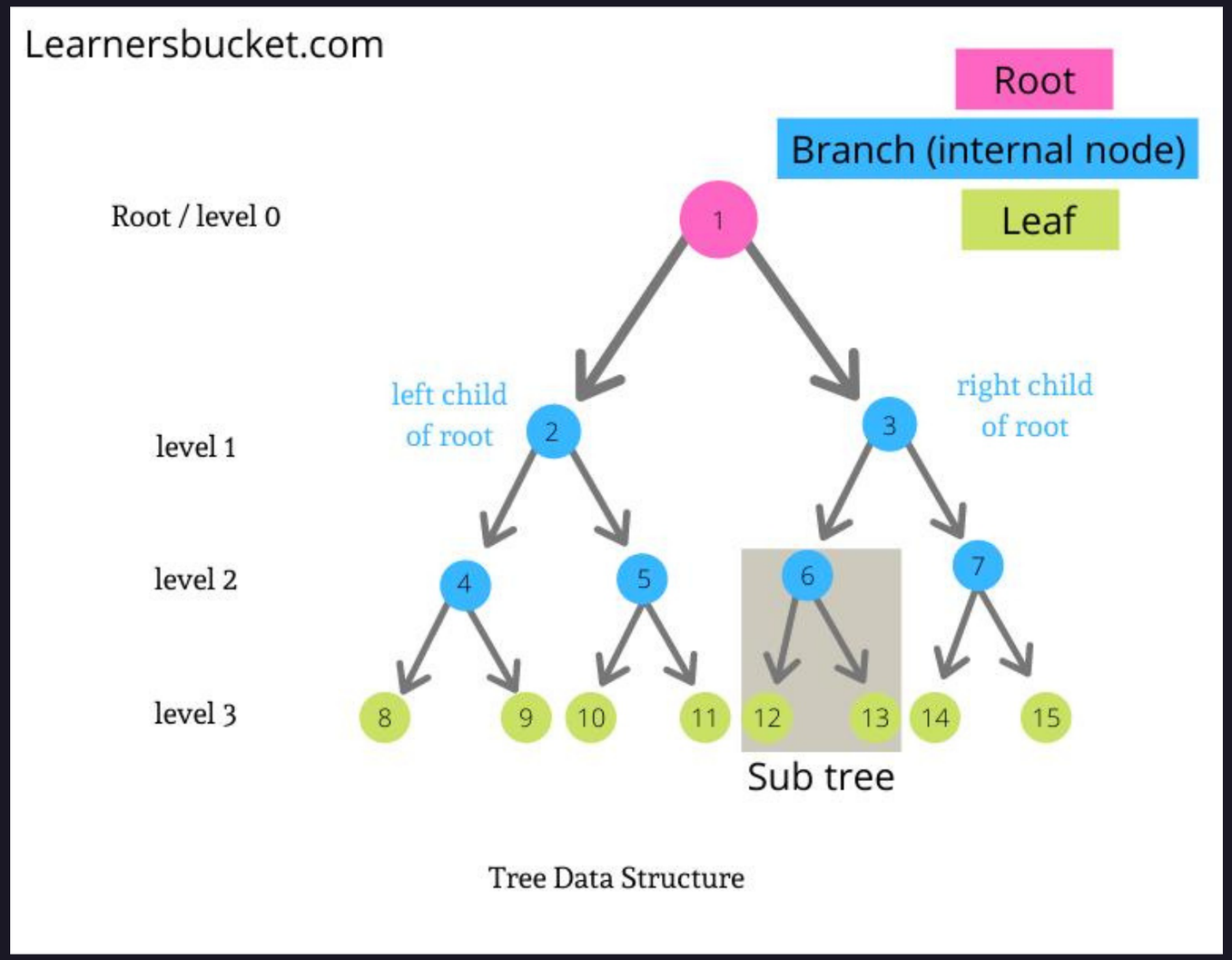


Self-Balancing Binary Search Tree Project

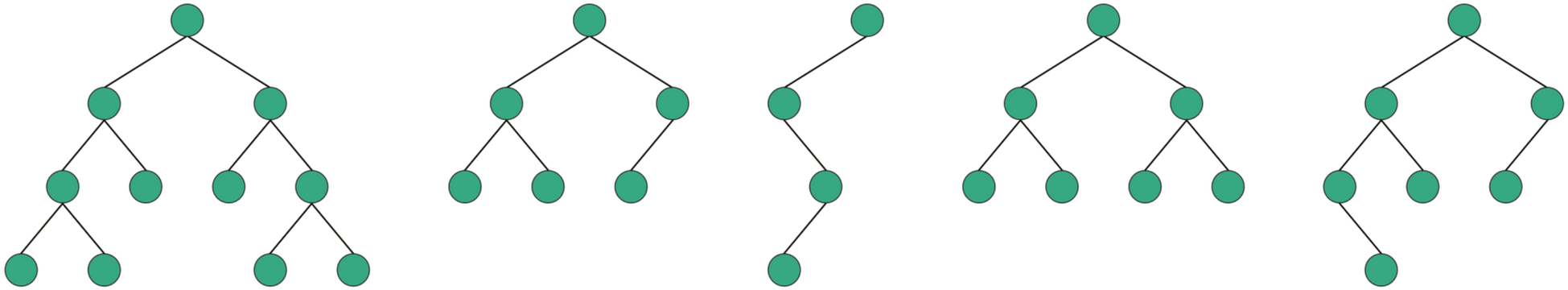
ENGS115 Data Structures and Algorithms
Mariam Mkrtichyan, Maria Apitonian

Binary tree



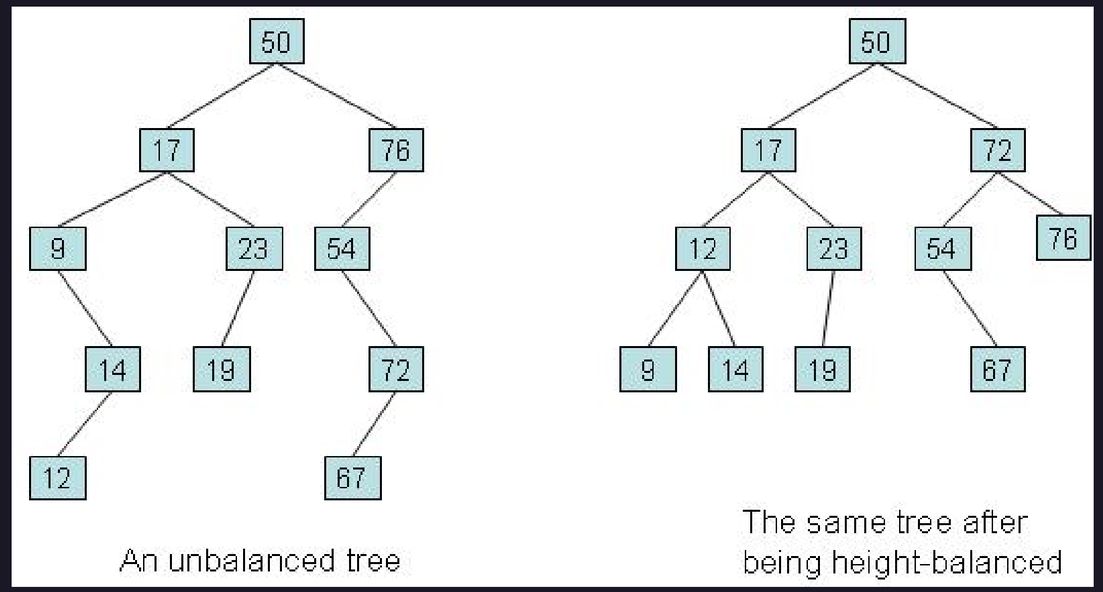
A hierarchical data structure with a set of linked nodes. Trees have a starting value (root) and subtrees of children with linked nodes to their parents. Each parent node has a greater key than the left child node and a smaller key than the right child node.

Binary tree types
Balanced binary tree

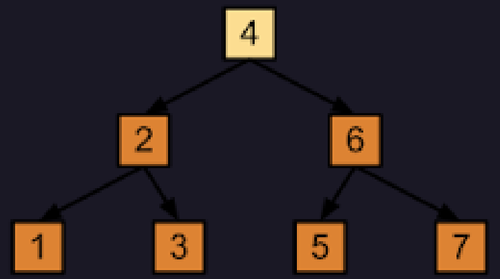


Full | Complete | Degenerate | Perfect | Balanced

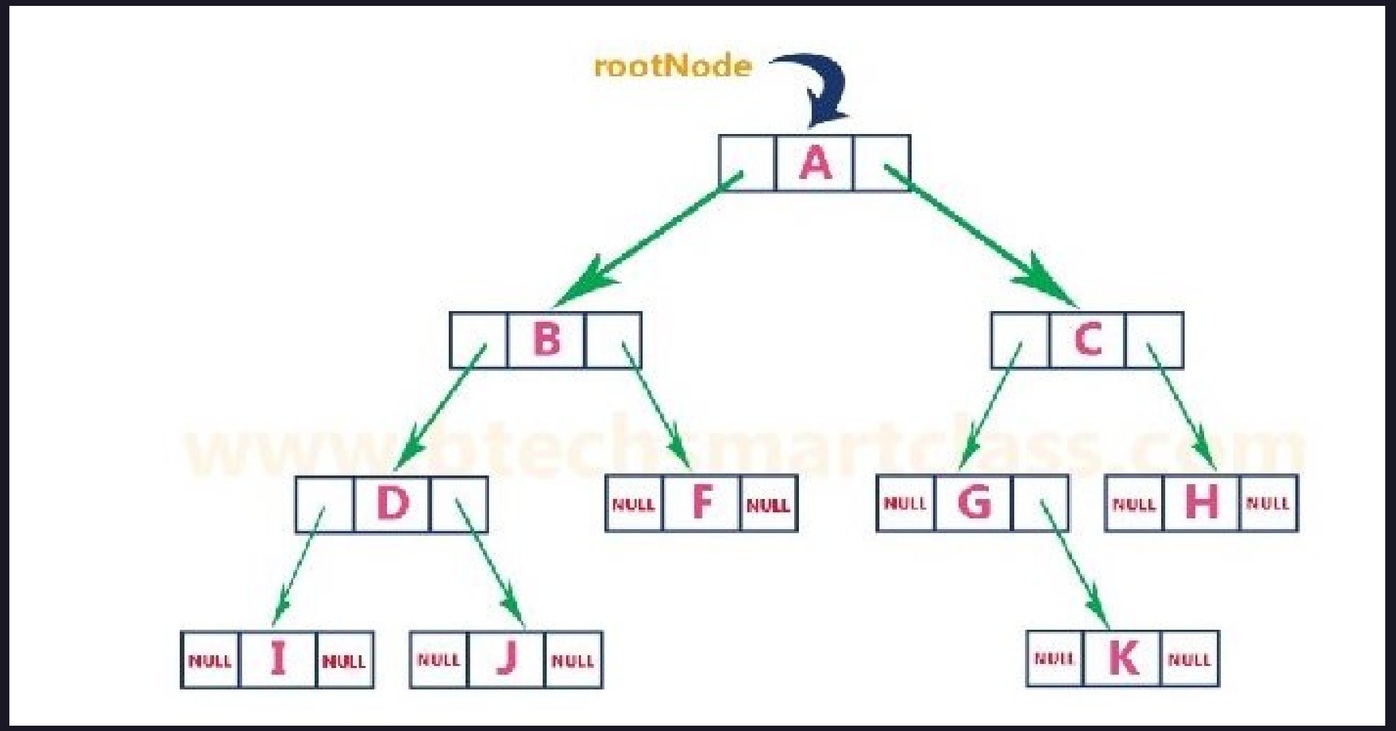
Non-Balanced vs. Balanced



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



Double linked implementation for the binary tree structure

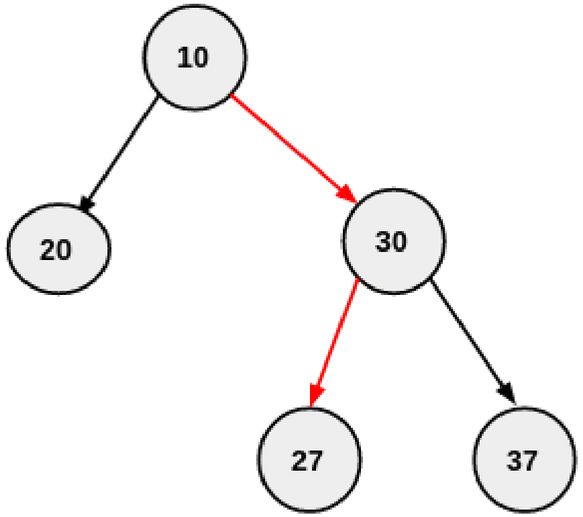



```c
struct node
{
    element_type key;
    struct node* left;
    struct node* right;
};

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

Binary tree height

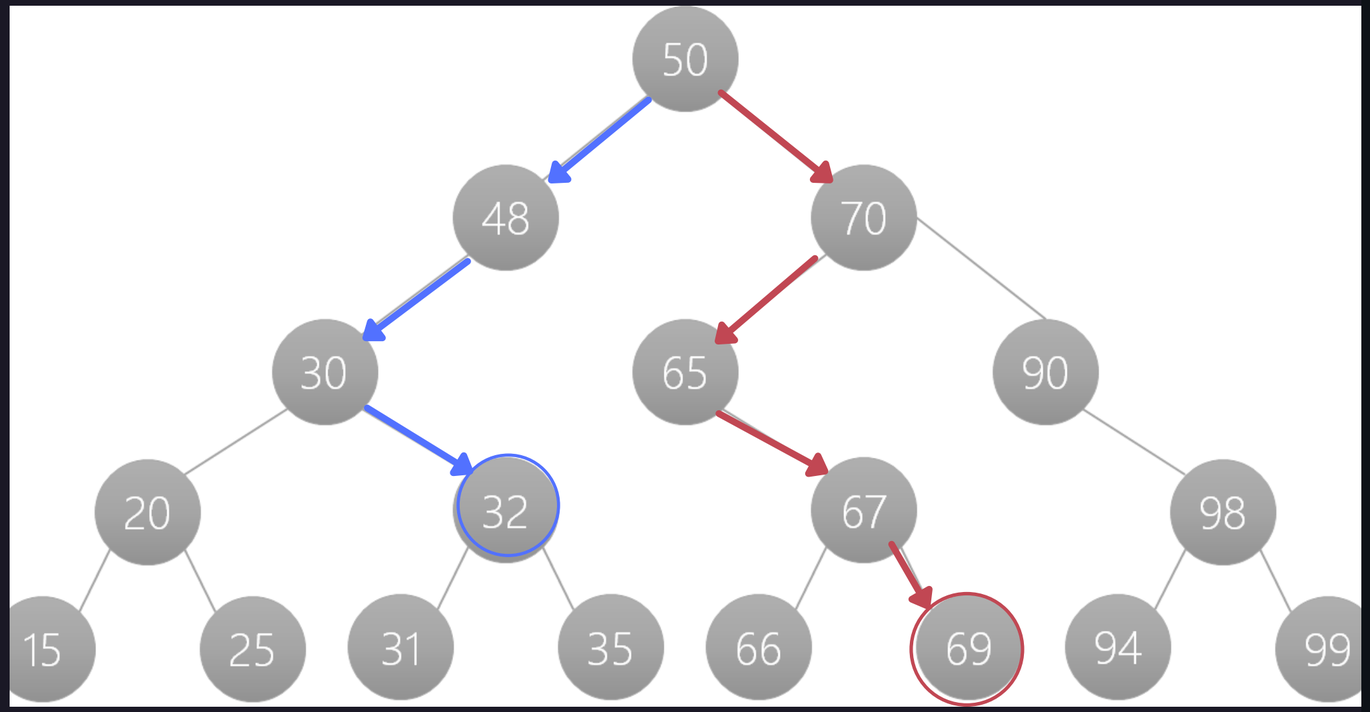
The height of the binary tree is the longest path from root node to any leaf node in the tree.

Number of edges between root and it's furthest leaf node =2.
Hence,
Height of tree = 2.




```c
int tree_height(struct node* root)
/*function that calculates the tree's/subtree's height starting from passed node*/
{
        if(root == NULL){
                return 0;
        }

        int maxleft = 1;
        int maxright = 1;
        struct node* leftsub = node_create(1, NULL, NULL);
        struct node* rightsub = node_create(1, NULL, NULL);
        leftsub = root;
        rightsub = root;

        while(true){
                if(leftsub->left != NULL){
                        leftsub = leftsub->left;
                        maxleft++;
                }
                else if(leftsub->right != NULL){
                        leftsub = leftsub->right;
                        maxleft++;
                }
                else{
                        break;
                }
        }
        while(true){
                if(rightsub->right != NULL){
                        rightsub = rightsub->right;
                        maxright++;
                }
                else if(rightsub->left != NULL){
                        rightsub = rightsub->left;
                        maxright++;
                }
                else{
                        break;
                }
        }

        if(maxleft>maxright){
                return maxleft;

        }
        else{
                return maxright;
        }
}
```

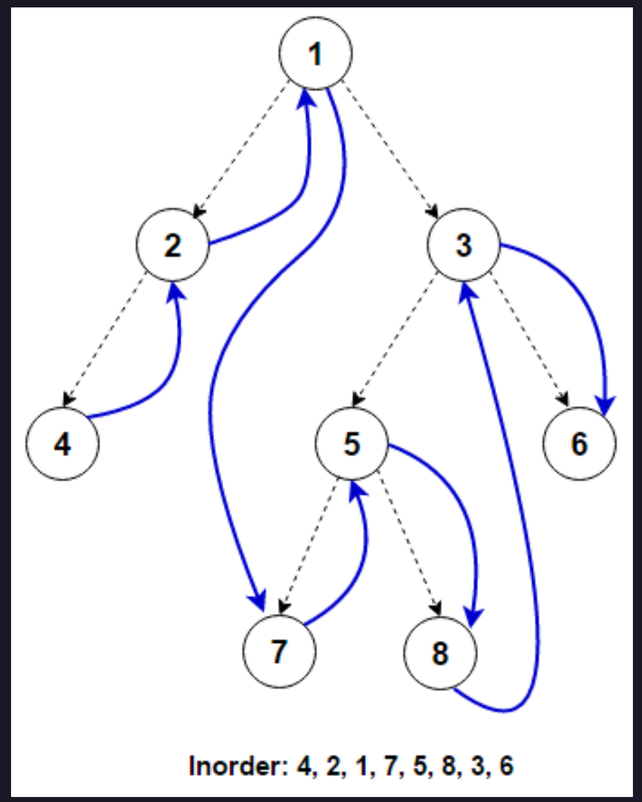
Searching in a BST



$$n \leq 2^h - 1 \quad \rightarrow \quad h \leq \log_2(n + 1)$$

```c
char* search_value_path(struct tree* t, element_type value)
/*returns the path to any node in the tree. Useful for other functions like boolean search, delete,
{
    struct node* root = t->start;
    char path[tree_height(t->start)] = {};
    int steps = 0;
    for(int i=0; i<tree_height(t->start); i++) {
        if(root->key==value){
            if(steps=0){
                printf("The values exists at the root position");
                path[steps]='r';
            }
            else{
                printf("The value exists at position %s", path);
            }
            return path;
        }
        else if((root->key>value) && (root->left != NULL)){
            root=root->left;
            path[steps]='l';
            steps++;
        }
        else if((root->key<value) && (root->right != NULL)){
            root=root->right;
            path[steps]='r';
            steps++;
        }
        else{
            printf("The value does not exist in the binary search tree");
            return path;
                }
        }

}
```

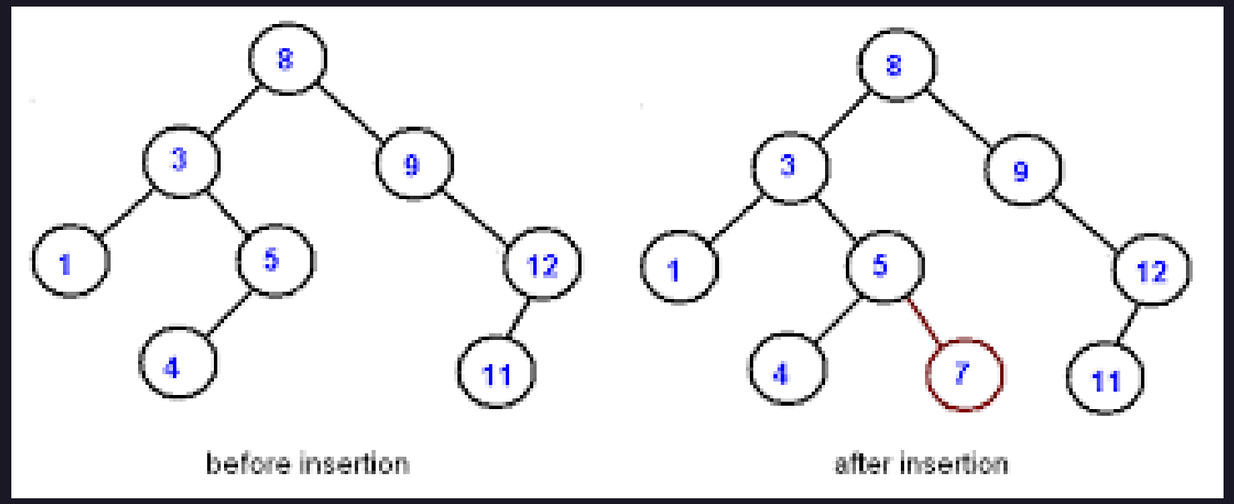
Inorder traversal



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).

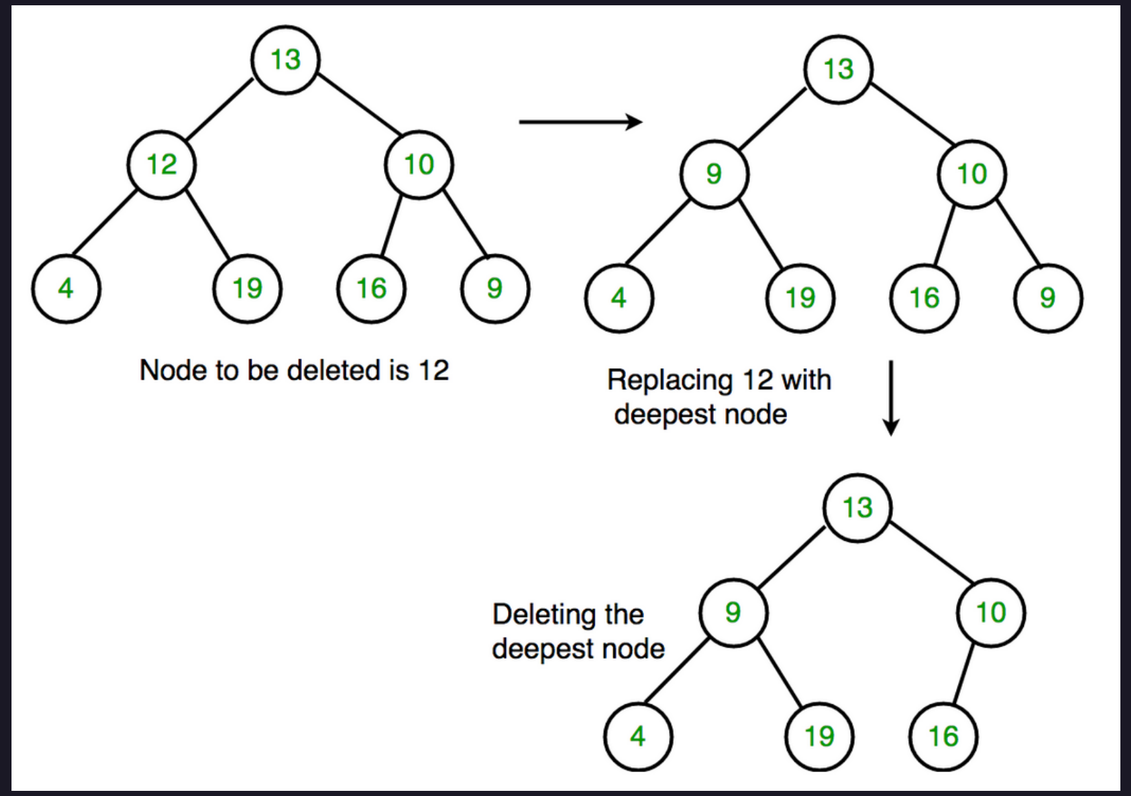
```
void inorder_helper (struct node* n ) //(struct node* n)
/*inorder tree traversal*/
{
    printf("print tree \n");
    assert(NULL != n);
    if(n != NULL ){
        inorder_helper(n->left);
        printf("%d \n",n->key);
        inorder_helper(n->right);
    }
}

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

Node insert



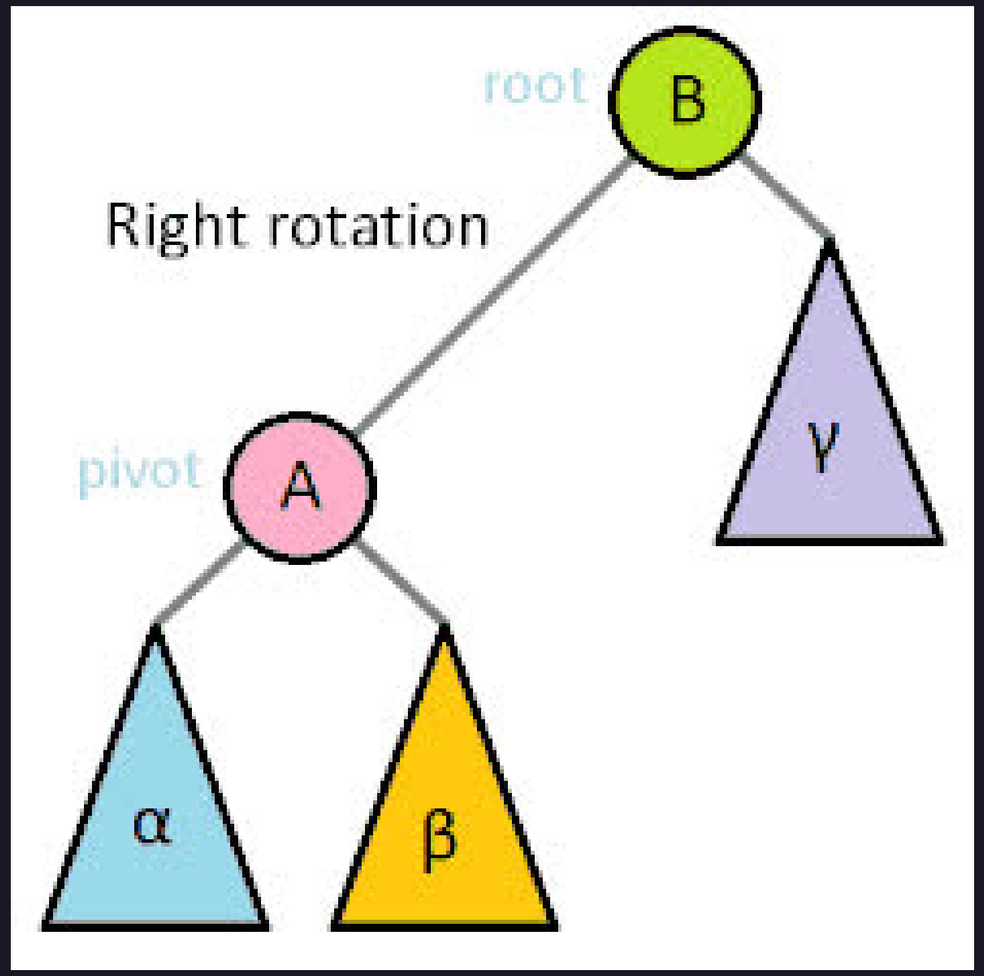

```c
int tree_insert(struct tree* t, element_type value)
{
        if(search_value(t, value) == 1){
                printf("Value cannot be inserted as it already exists.");
                return 0;
        }

        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;
}
```

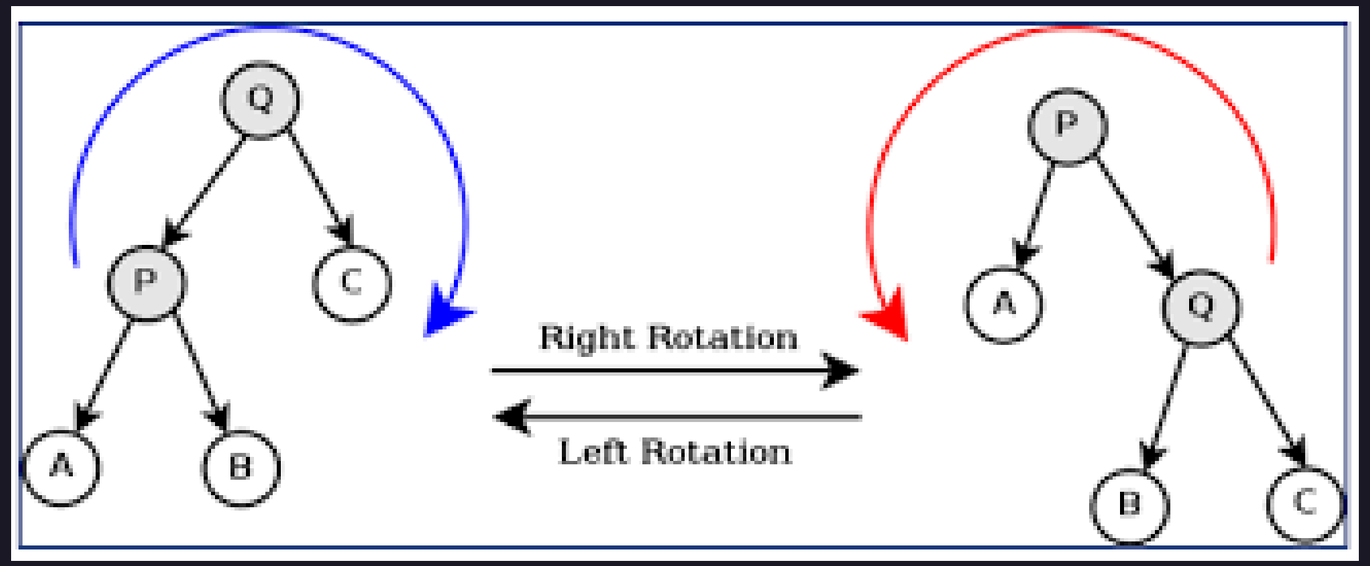
Node Delete




```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;
        }

        char* path = search_value_path(t, value);
        if(path[0] == 'r'){
                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;
        }
```

Right / Left Single Rotation





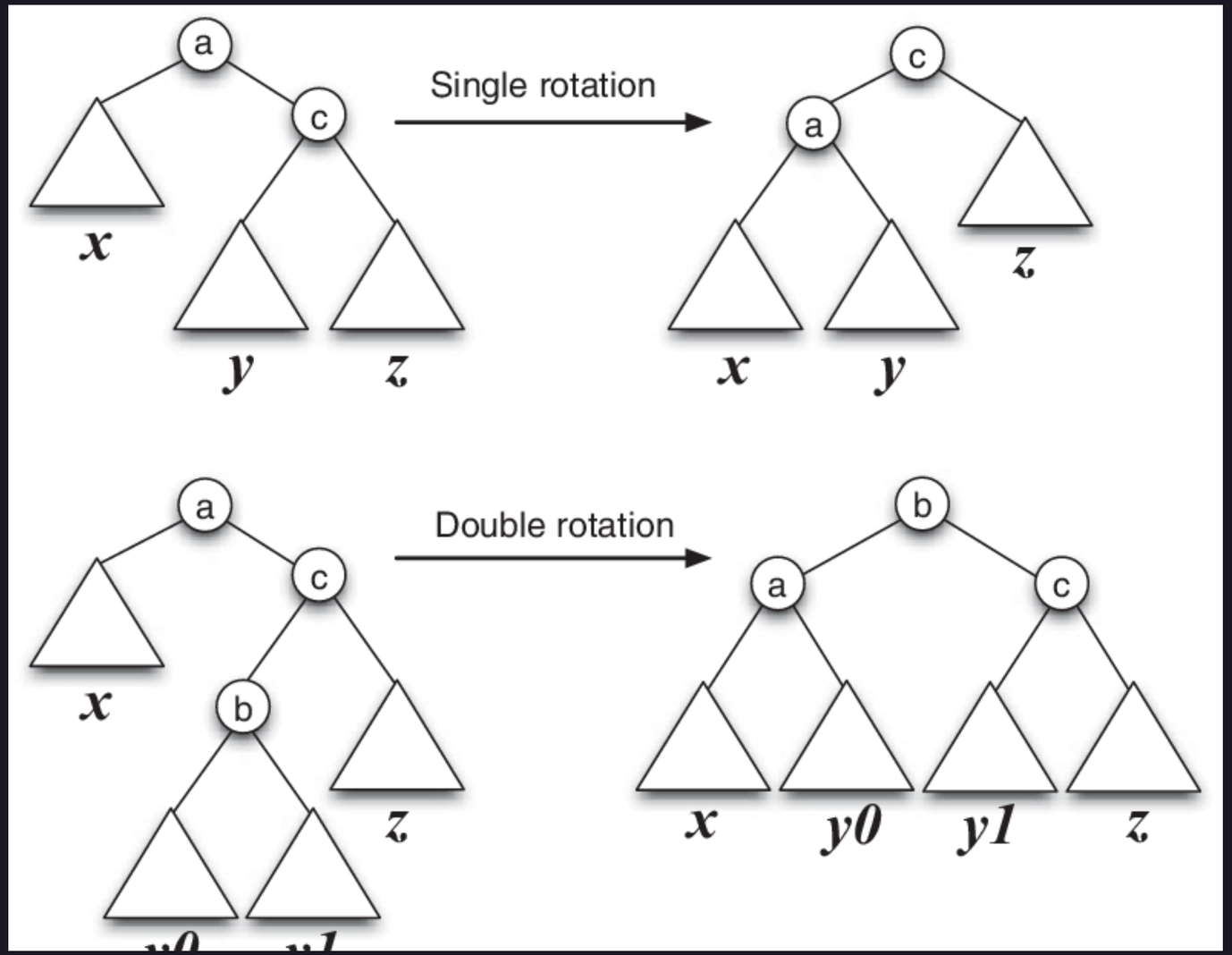
Single rotation left/right

```
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_lhelper(struct node* n)/* a single rotation from right to left at the root of T*/
{
        struct node* r  = n->right;
        n->right = r->left;
        r->left  = n;
        n = r;
}
```

Double left rotation

```c
/*Double rotation*/
void double_rotate_lhelper(struct node* n)
{
        single_rotation_rhelper(n->right);
        single_rotation_lhelper(n);
}
void double_rotate_left(struct tree* t)
{
        assert (NULL !=t);
        double_rotate_lhelper(t->start);
}

void double_rotate_rhelper(struct node* n)
{
        single_rotation_lhelper(n->right);
        single_rotation_rhelper(n);
}
void double_rotate_right(struct tree* t)
{
        assert (NULL !=t);
        double_rotate_rhelper(t->start);
}
```



Other functions in the implementation

- int is_empty(struct tree* t)
- struct node* minimum(struct node* n)
- struct node* maximum(struct node* n)
- struct node* find_parent(struct tree* t, element_type value)

RESOURCES

Anderson-Freed, S, E. Horowitz and S. Sahni (2007). Fundamentals of Data Structures in C. Computer Science Press.

Kataria, A. (2018, June 11). Traversal technique for Binary Tree. Retrieved from www.includehelp.com/data-structure-tutorial/traversal-technique-for-binary-tree.aspx.

Srivastava, A. K. (2019). A Practical Approach to Data Structure and Algorithm with Programming in C (pp 337-392). Retrieved from EBSCO Publishing.

SPARTA, T. E. G., Gerasch, T. E., Sparta, Center, N. A. S. A. A. R., & Metrics, O. M. V. A. (1988, May 1).An insertion algorithm for a minimal internal path length binary search tree. Communications of the ACM. Retrieved December 3, 2021, from https://dlnext.acm.org/doi/10.1145/42411.42421.