Laboratory Module 4

Height Balanced Trees

Purpose:

- ***understand the notion of height balanced trees***
- ***to build, in C, a height balanced tree***

1 Height Balanced Trees

1.1 General Presentation

Height balanced trees (or AVL trees) is named after its two inventors, G.M. Adelson-Velskii and E.M. Landis, who published it in their 1962 paper "An algorithm for the organization of information." As the name sugests AVL trees are used for organizing information.

AVL trees are used for performing search operations on high dimension external data storage. For example, a phone call list may generate a huge database which may be recorded only on external hard drives, hard-disks or other storage devices. AVL is a data structure that allows storing data such that it may be used efficiently regarding the operations that may be performed and the resources that are needed.

AVL trees are binary search trees, wich have the balance propriety. The balance property is true for any node and it states: “the height of the left subtree of any node differs from the height of the right subtree by 1”.

The structure of the nodes of a balanced tree can be represented like:

```
struct NodeAVL{
   int key;
   int ech;
   nod *left, *right;
};
```

Where:

- ***key*** represents the tag of the node(integer number),
- ***ech*** represents the balancing factor
- ***left*** and ***right*** represent pointers to the left and right children.

Here are some important notions:

[1] The lenght of the longest road from the root node to one of the terminal nodes is what we call the ***height of a tree***.

[2] The difference between the height of the right subtree and the height of the left subtree is what we call the ***balancing factor***.

[3] The binary tree is balanced when all the balancing factors of all the nodes are **-1,0,+1**.

Formally, we can translate this to this: $| h_d - h_s | \leq 1$, node X being any node in the tree, where h_s and h_d represent the heigts of the left and the right subtrees.

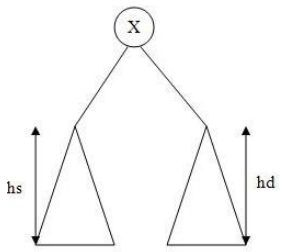


Figure 1. Balancing factor equals $h_d - h_s$

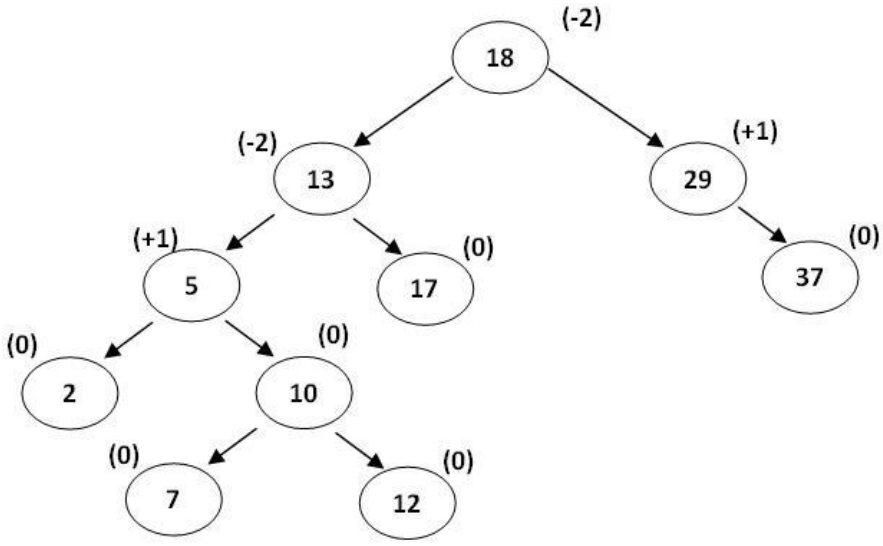


Figure 2. Binary search tree with computed baancing facors

Here are some xxamples of procedures for the calculus of the height of a subtree and of the balancing factor for the above binary search tree.

- the height of the tree is 4, meaning the lenght of the longest path from the root to a leaf node.

- the height of the left subtree of the root is 3, meaning that the lenght of the lognest path from the node 13 to one of the leaf nodes (2, 7 or 12).

-for finding the balancing factor of the root we substract the height of the right subtree and the left subtree : 1-3 = - **2**.

-the balancing factor of the node with the key 12 is very easy to determine.We notice that the node has no children so the balancing factor is 0.

-for finding the balancing factor of the node with key 5 we substract the height of the right subtree from the height of the left subtree: 1 - 0 = **+1**.

We present below the function drum, wich calculates the lognest road frome the current node , meaning the height of a subtree:

```
void path(NodeAVL* p,int &max, int length){
    if (p!=NULL){
            path(p->right,max,length+1);
            if ((p->left==NULL)&&(p->right==NULL)&&(max<length))
                max=length;
            path(p->left,max,length+1);
    }
}
```

Using this function we can determine the balancing indicatior of each node of the tree with the function *balanceFactor*:

```
void balanceFactor(NodeAVL *p){
    int max1,max2;
    max1=1; max2=1;
    if(p->left!=NULL)
            path(p->left,max1,1);
    else
            max1=0;
    if(p->right!=NULL)
```

```
            path(p->right,max2,1);
        else
            max2=0;
        p->ech=max2-max1;
    }
```

1.2 Operations on AVL Trees

1.2.1 Insertion into AVL Trees

An insertion into the tree can lead to the umbalance of some nodes wich occurs in the case of the formula: $|h_d - h_s| \leq 1$ not being respected.

Basically, a key is inserted like in an ordinary binary search tree, meaning that we start from the root following the left or right nodes according to the relation between the key we insert and the key of the nodes we are passing, untill we reach a node which can become parent of the new node. After insertion we go recursively upwords and we search the first node that is not balanced, meaning the first node whose subtrees heights difference differ by 2 units. This node must be balanced and will be in 1 of the 4 cases wich follow.

Case 1. Simple right rotate

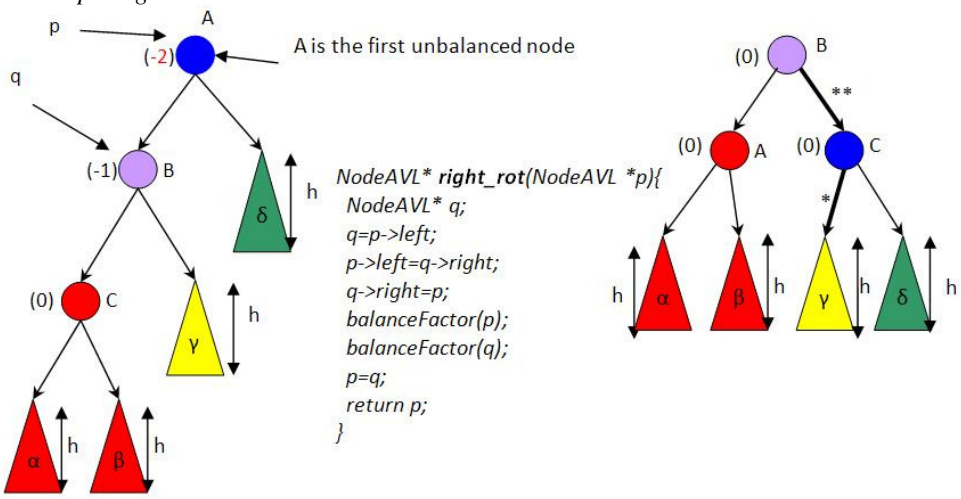


Figure 3. Simple right rotate

Case 2. Simple left rotate

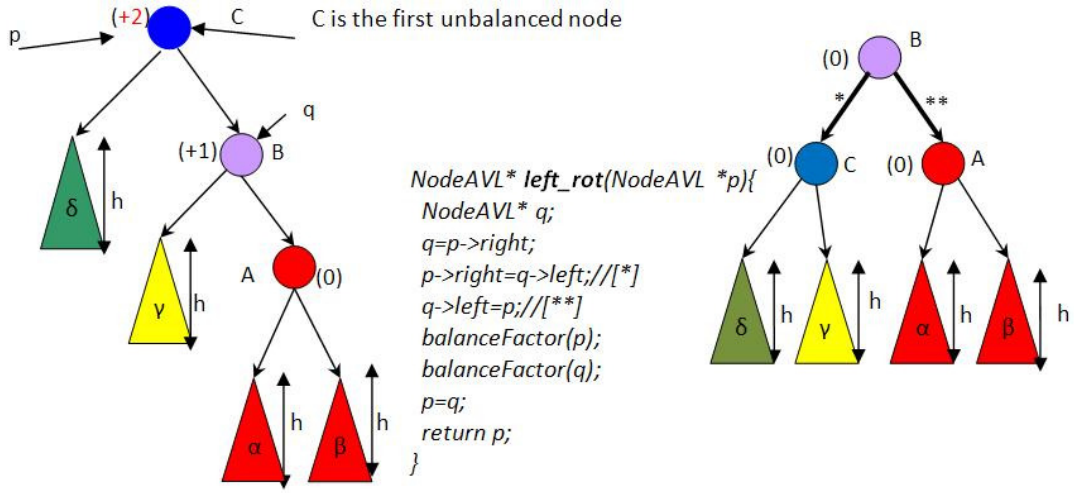


Figure 4. Simple left rotate

Case 3. Double right rotate

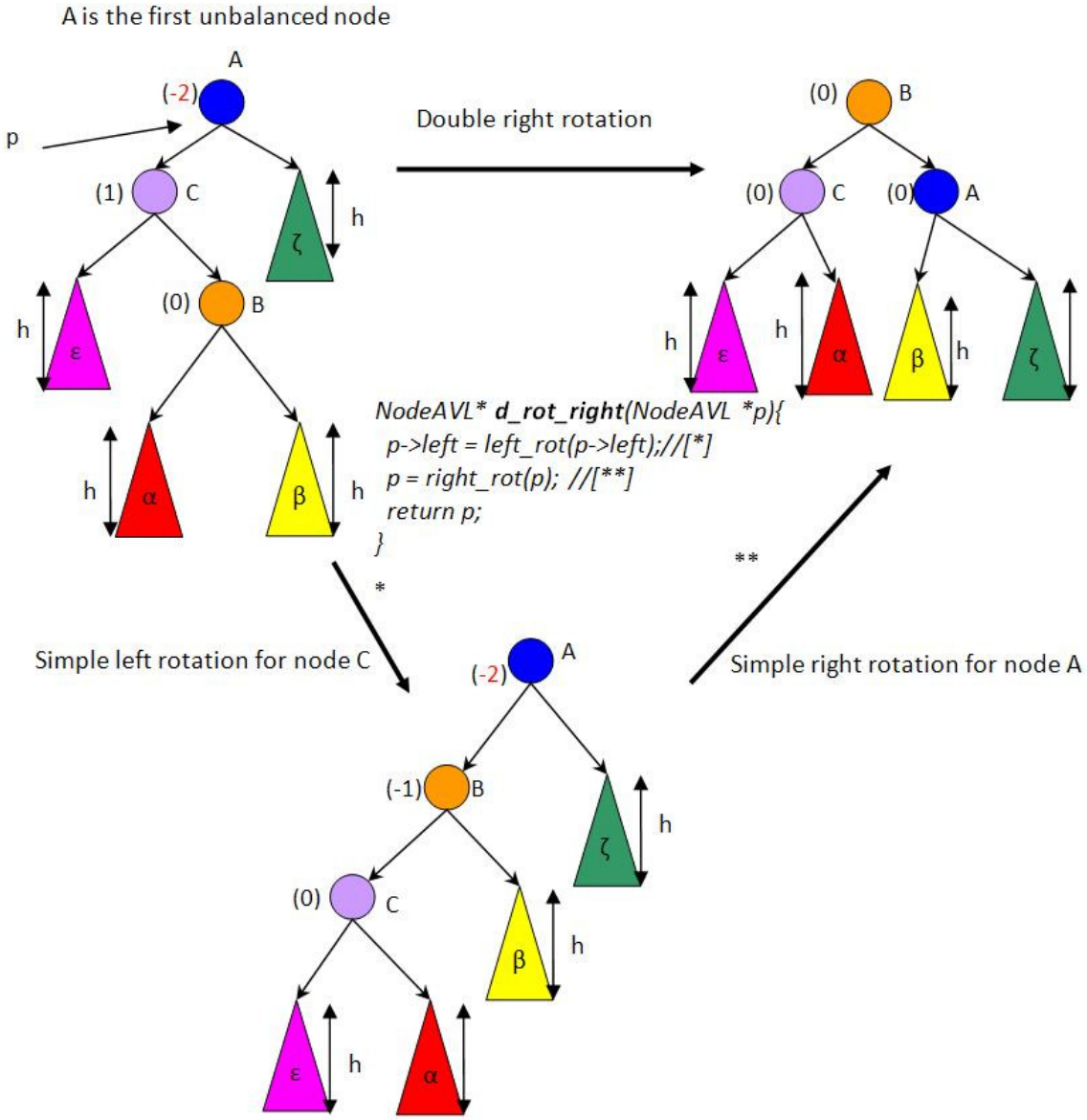


Figure 5. Double right rotate

Case 4. Double left rotate

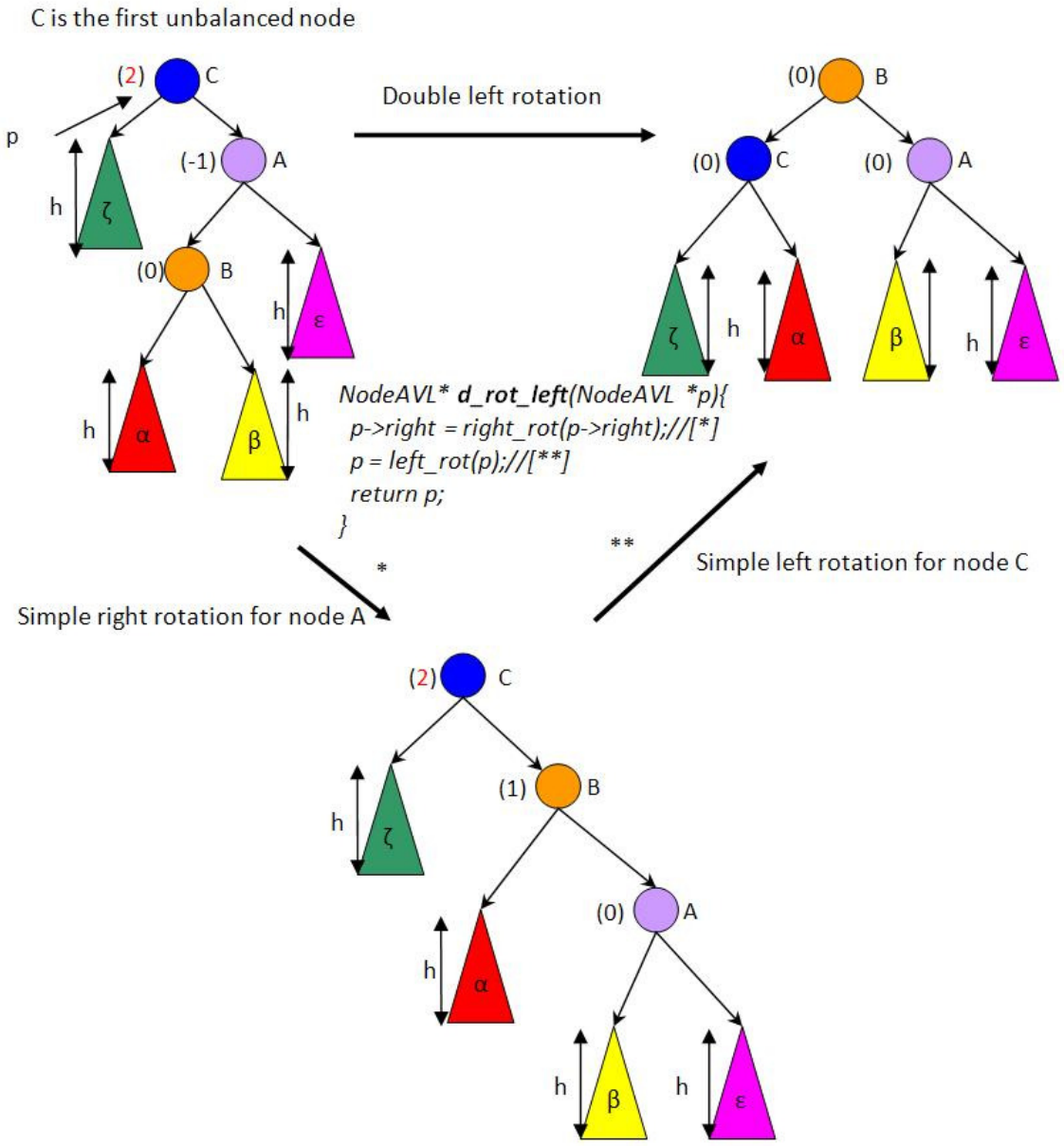


Figure 6. Double left rotate

Here is an example of building an AVL tree by inserting keys 4, 5, 7, 2, 1, 3, 6. We start from an empty tree.

INSERT KEYS 4 and 5

The nodes with the keys 4 and 5 will be inserted like in the binary search trees. Since we do not have balance factor greater than 1 no rotation is needed.

INSERT KEY 7

The node with the 7 key should be inserted in the right of the node with the key 5. In this case the tree is umbalanced since the balance factor of root node becomes 2. Figure 7 presents the simple left rotation that is performed the the final shape of the tree.

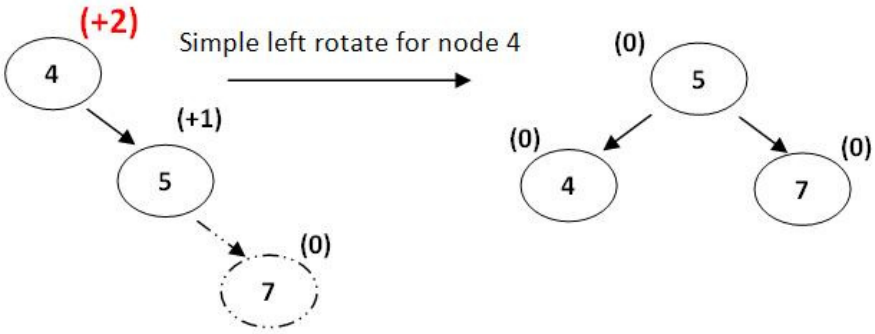


Figure 7. Simple left rotation for root node

INSERT KEY 2

The node with the 2 key will be inserted at the left of the node with the 4 key without the tree becoming umbalanced.

INSERT KEY 1

The node with the 1 key will be inserted at the left of the node with the key 2, but in this case the tree is unbalanced because key 4 has balance factor -2. Figure 8 presents the simple right rotation that is performed the the final shape of the tree.

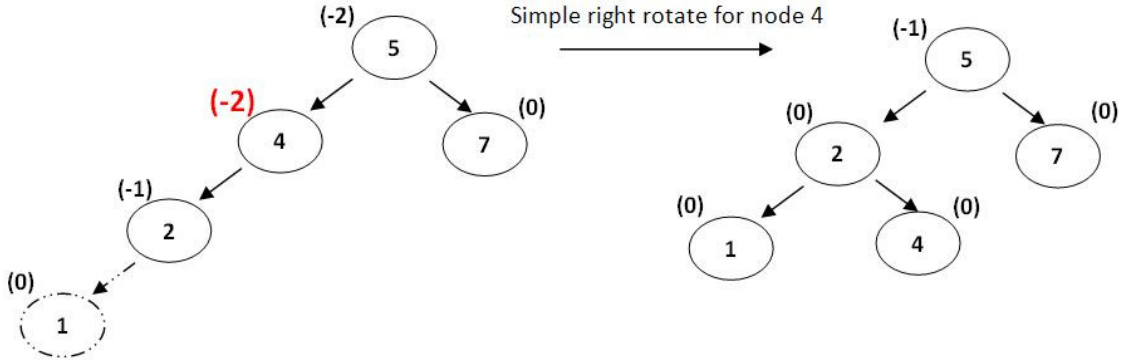


Figure 8. Simple right rotation for node with key 4

INSERT KEY 3

The node with the 3 key will be inserted at the left of the node with the key 4, but in this case the tree is unbalanced because key 5 has balance factor -2. Figure 9 presents the double right rotation that is performed the the final shape of the tree.

INSERT KEY 6

The node with the 6 key will be inserted at the left of the node with the key 7, but in this case the tree is unbalanced because key 5 has balance factor +2. Figure 10 presents the double left rotation that is performed the the final shape of the tree.

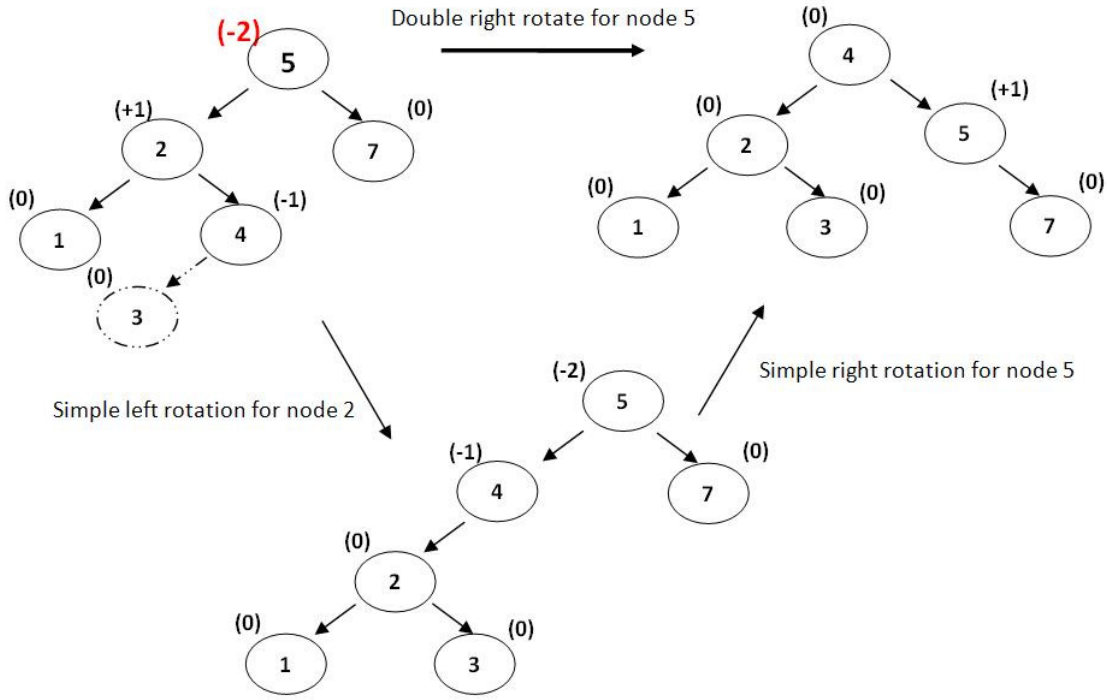


Figure 9. Double right rotation for node with key 5

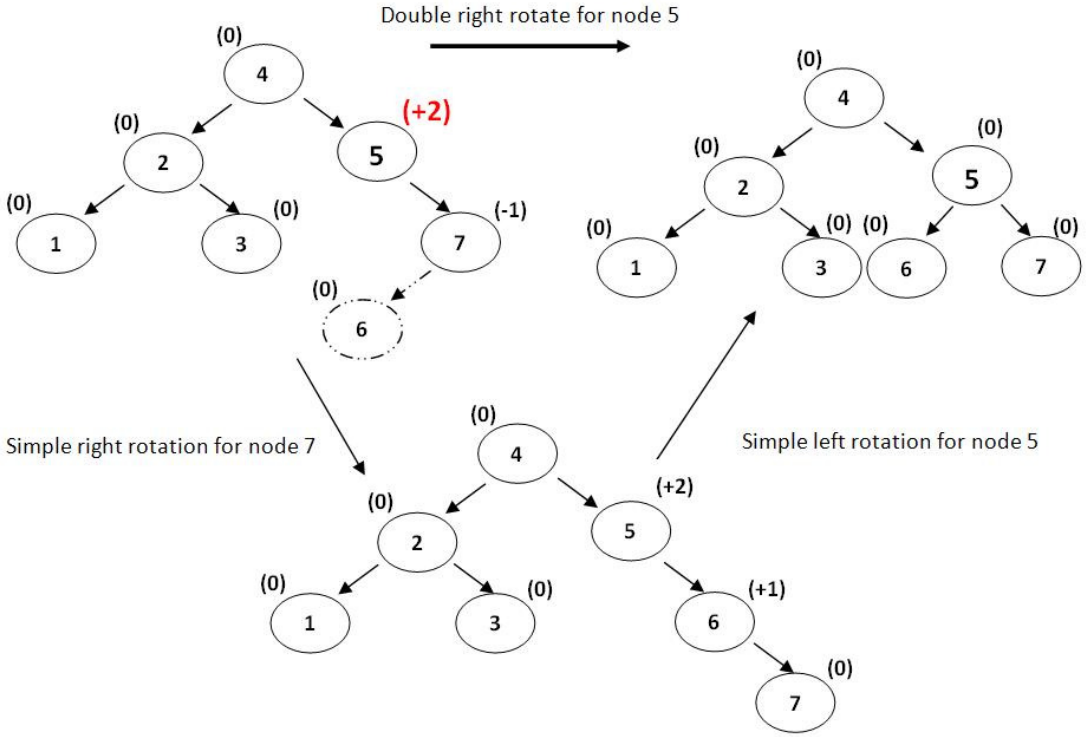


Figure 10. Double left rotation for node with key 5

The insert function is below:

```
NodeAVL* insert(NodeAVL *p, int x){
        if(p==NULL){
         // if the current node is NULL then
                p=new NodeAVL;          // allocate space for it in the heap area
                p->key=x;               // the information becomes x
                p->ech=0;                // the factor balance is 0 - the node is balanced
                p->right=NULL;           // the node is inserted as a leaf node
                p->left=NULL;           // so that its children are referenced NULL
                return p;
        }else
                if (x<p->key)
         // if the key that wish to be inserted is smaller than the information value of the current node
                        p->left = insert(p->left,x);
        // then is inserted into the current node's left subtree
                else
                        if (x>p->key)
        // if the key that wish to be inserted is greater than the information value of the current node
                                p ->right = insert(p->right,x);
        // then is inserted  //into the current node's rigt subtree
                        else
                                cout<<"The node already exist "<<endl;
        p = balance(p);
        // if there will be cases of imbalance it will balance the tree or even the subtree
        return p;
}
```

We notice that inside the insert function we call an balancing function, in case that the tree becomes umbalanced. The balance function acts on the node with balance factor outside the range -1, 0, +1. The balance function is:

```
NodeAVL* balance(NodeAVL *p){
        NodeAVL *w;
        balanceFactor(p);               // Calculate the balance of the current node p
        if(p->ech==-2){                 // if p is a critical node with balance facor -2
                w=p->left;              // then w is the left child of p
                if (w->ech==1)           // and if it has a balance factor of 1
                        p = d_rot_right(p);         // then is made a double right rotation
                else                                // otherwise a simple right rotation
                        p = right_rot(p);
        }
        else
                if(p->ech==2){                  // if p is a critical node with balance facor +2
                        w=p->right;             // then w is the right child of p
                        if (w->ech==-1)         // and if it has a balance factor of -1
                                p = d_rot_left(p);          // then is made a double left rotation
                        else                                // otherwise a simple left rotation
                                p = left_rot(p);
                }
                return p;
}
```

1.2.2 Deletion from AVL Trees

Deletion from AVL trees may be treated as deletion of a node for which *left* **or** *right* pointer are NULL. From this point of view the procedure is similar with deletion of nodes from a binary search tree. The difference consists in the fact that the path from the root node to the node that is to be deleted needs ro be recorded. The steps necessary for deleting nodes from an AVL tree are:

Step1. Find the node that needs to be deleted. The path from root node to deleted node needs to be recorded.

Step2. Find the predecessor/successor of the deleted node. This path needs also to be recorded.

Step3. Replace the key of the deleted node with the key of the predecessor/successor. At successor level make the proper indirections (same as in a binary search tree).

Step4. Update the balance factors on the backward path from the predecessor/successor node involved in deletion at step 3 to the root node. When necessary perform rotations.

Let us suppose we have an AVL tree as the one in the next figure where node with key 12 needs to be deleted.

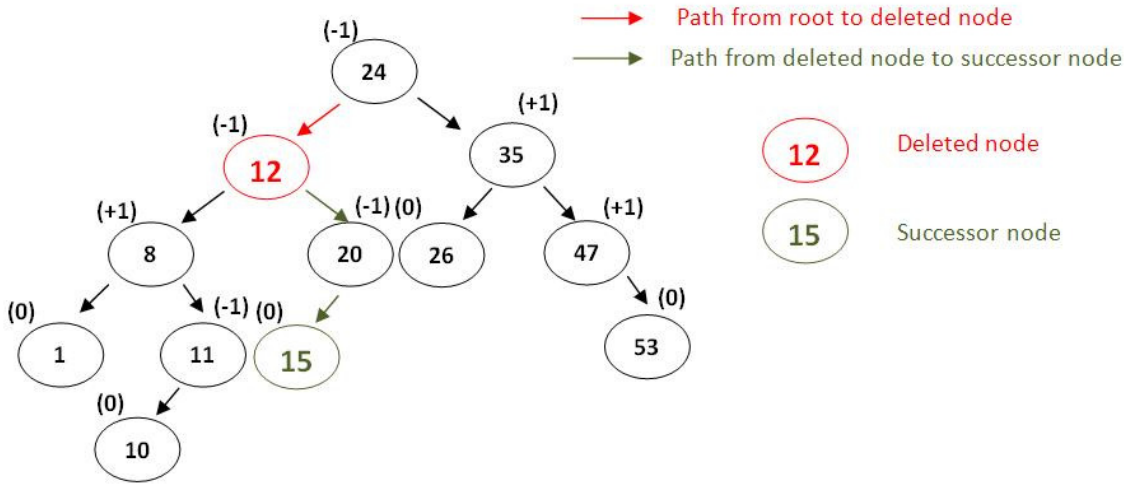


Figure 11. Deletion of node with key 12 – initial situation

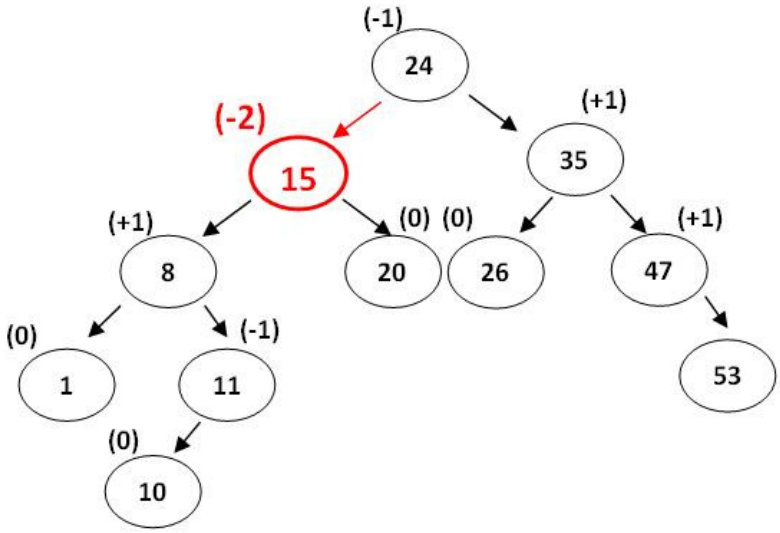


Figure 12. Deletion of node with key 12 – after replacing key 12 with key 15

The node with key 15 (the successor for the node with key 12) has replaced the key 12. After this operation is performed, the balance factor of node with key 20 becomes 0 and the balance factor of node with key 15 becomes -2. This means a rebalancing procedure for node 15 is necessary. Since node 15 has balance factor -2 and the left child (node with key 8) has balance factor of +1 a double right rotation for node 15 is necessary. After this rotation the tree will look like in the next figure. The balance factor of node with key 24 is also increased thus becoming 0.

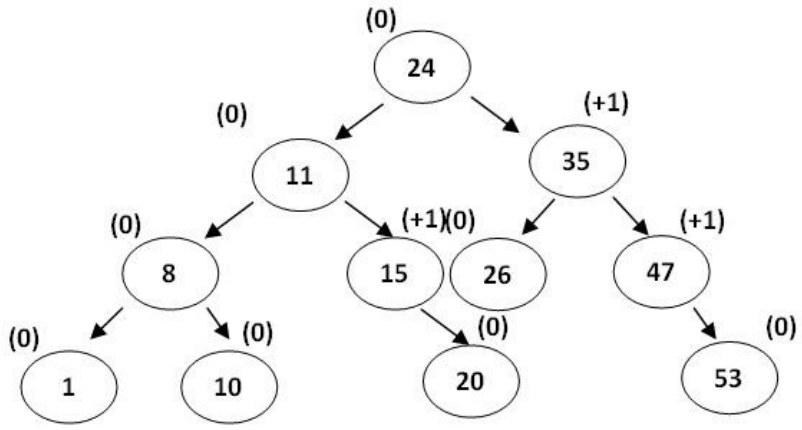


Figure 13. Deletion of node with key 12 – final shape, after rebalancing

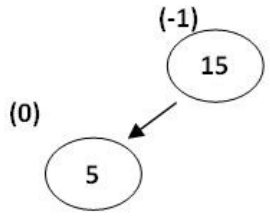
The deletion function is the following:

```
NodeAVL* deletion(NodeAVL *p,int x){
        NodeAVL *q,*r,*w;
        if (p!=NULL)                    // if the current node is not NULL then
                if (x<p->key)
         // the key that wish to be removed is less than the node information
                        p->left = deletion(p->left,x);
                // the key is searches in the left subtree of current the node
                else
                        if (x>p->key)           // if the key is greater
                                p->right = deletion(p->right,x);// searches in the right subtree
                        else{
                                // if the key is equal with the information from the current node
                                q=p;                    // a node q becomes p
                                if (q->right==NULL)     // if q's right child is NULL
                                        p=q->left;      // then p becomes q->stanga
                                else
                                        if (q->left==NULL)      // if q's left child is NULL
                                                p=q->right;     // p becomes q->dreapta
                                        else{
                                                w=q->left;      // else w is q's left child
                                                r=q;            // r becomes q
                                                if (w->right!=NULL) { //if w's right child is not NULL
                                                        while (w->right!=NULL){
                                                                r=w;
                                                                w=w->right;
                                                        }
                                                        p->key=w->key;
                                                        q=w;
                                                        r->right=w->left;
```

```
                                        r=p->left;
                                        w=w->left;
                                        if (r!=NULL)
                                                while ((r!=w)&&(r!=NULL)){
                                                        r = balance(r);
                                                        r=r->right;
                                                }
                                }
                                else{
                                        p->key=w->key;
                                        p->left=w->left;
                                        q=w;
                                }
                        }
                        delete(q);      // q is deleted
                }
                if (p!=NULL)
                        p = balance(p);      // p is balanced if is not NULL
                return p;
}
```

2. Assignments

1) Write a program that creates and manages an AVL tree. The program must implement the following operations: creation, insertion, deletion, update and tree display. The program should present a meniu where user may choose from implemented options.
2) It is given the AVL tree from the following figure.



Present the shape of the tree after each of the following operations:

- INSERT 3
- INSERT 24
- INSERT 27
- INSERT 37
- DELETE 31
- DELETE 25
-

For each insertion operation there must be presented the following:

- Initial position of the key into the AVL tree
- The updated balance factors
- The critical nodes, with balance factor different from -1, 0 or 1
- The rotation type that needs to be performed and how the rotation will perform
- The final shape of the tree

For each deletion operation there must be presented the following:

- The search of the tree for the key that needs to be deleted
- The identification of the predecessor/successor key

- The replacement of the deleted key with predecessor/successor key and the proper pointer indirection at predecessor/successor level
- The updated balance factors
- The critical nodes, with balance factor different from -1, 0 or 1
- The rotation type that needs to be performed and how the rotation will perform
- The final shape of the tree

References:

Adelson-Velskii, G.; E. M. Landis (1962). "An algorithm for the organization of information". Proceedings of the USSR Academy of Sciences 146: 263–266. (Russian) English translation by Myron J. Ricci in Soviet Math. Doklady, 3:1259–1263, 1962.