Laboratory Module 5

Splay Trees

Purpose:

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

1 Splay Trees

1.1 General Presentation

Splay trees are binary search trees that have implemented a self-adjusting mechanism. This mechanism performs in the following way: every time we access a node of the tree, whether for insertion or retrieval, we perform rotations (like in AVL trees), lifting the newly inserted/accessed node all the way up, so that it becomes the root of the modified tree.

The nodes on the way are rotated such that the tree becomes more balanced. The splay tree is NOT a height balanced tree since there are situations when a node may have the balance factor different from -1, 0 or +1. Nodes that are frequently accessed will frequently be lifted up to become the root, and they will never drift too far from the top position. Inactive nodes, on the other hand, will slowly be pushed farther and farther from the root. It is possible that splay trees can become highly unbalanced, so that a single access to a node of the tree can be quite expensive. Later in this section, however, we shall prove that, over a long sequence of accesses, splay trees are not at all expensive and are guaranteed to require not many more operations even than AVL amortized analysis trees.

The analytical tool used is called amortized algorithm analysis, since, like insurance calculations, the few expensive cases are averaged in with many less expensive cases to obtain excellent performance over a long sequence of operations. The operations that are performed are rotations of a similar form to those used for AVL trees, but now with many rotations done for every insertion or retrieval in the tree. In fact, rotations are done all along the path from the root to the target node that is being accessed. Let us now discuss precisely how these rotations proceed.

The structure of a splay node is similar with the one that was used fr binary search trees. Note the appearance of the *parent* pointer.

```
struct NodeSplay{
   int key;
   nod *left, *right, *parent;
};
```

Where:

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

1.2 Domains of Usage for Splay Trees

Splay trees are typically used in the implementation of caches, memory allocators, routers, garbage collectors, data compression, ropes (replacement of string used for long text strings), in Windows NT (in the virtual memory, networking, and file system code) etc.

A splay tree is an efficient implementation of a balanced binary search tree that takes advantage of locality in the keys used in incoming lookup requests. For many applications, there is excellent key locality. A good example is a network router. A network router receives network packets at a high rate from incoming connections and must quickly decide on which outgoing wire to send each packet, based on the IP address in the packet. The router needs a big table (a map) that can be used to look up an IP address and find out which outgoing connection to use. If an IP address has been used once, it is likely to be used again, perhaps many times. Splay trees can provide good performance in this situation.

Another example of a domain in which splay trees are used are Intrusion detection systems (IDS) which are an essential part of the security infrastructure. They are used to detect, identify and stop intruders. The administrators can rely on them to find out successful attacks and prevent a future use of known exploits. IDS are also considered as a complementary solution to firewall technology by recognizing attacks against the network that are missed by the firewall. Nevertheless, IDS are plagued with false positive alerts, letting security professionals to be overwhelmed by the analysis tasks. Therefore, IDS employ several techniques in order to increase the detection probability of suspect threats while reducing the risk of false positives. This leads to the construction of a decision tree such as the splay tree used to verify the traffic. The proposed strategy to detect intrusions using protocol analysis consists in extracting data from each received packet and then traversing a pre-built decision tree.

The process of classifying packets into "flows" in routers, firewalls, packet filters etc., is called packet classification. Packet classification is used in a variety of applications such as security, monitoring, multimedia applications etc. These applications operate on packet flows or set of flows. Therefore these nodes must classify packets traversing through it in order to assign a flow identifier, called as Flow ID. Fast and efficient packet classification techniques are essential for the design of high-speed routers, and various other applications. In order to achieve this, a so called Splay Tree based Packet Classification (ST-PC) technique is used, which consists in using simple primitives in the design, namely splay tree models and prefix conversion methods.

1.3 Operations on Splay Trees

MEMBER (i, S) - finds out if the element "i" is in the "S" splay tree;
INSERT (i, S) - if the element "i" does not belong to the "S" splay tree, then the function inserts the element in the tree.
DELETE (i, S) - if the element "i" is in the "S" splay tree, then the function deletes the element.
JOIN (S, SP) - supposing that x < y, for any x of S and any y of SP, the function merges the "S" and "SP" splay trees.
SPLIT (i, S) - splits the "S" splay tree in two different trees "SP" and "SPP" such that the keys respect the following : x <= i <= y for any x of SP and any y of SPP.
SPLAY (i, S) - organizes the splay tree in such way that "i" element is brought to the root of the tree if "i" belongs to "S"

All these operations can be executed in a constant number of SPLAY operations and furthermore with a constant number of other low level operations such as pointers update and keys comparison.

1.3.1 Splaying Steps

When a single rotation is performed in a binary search tree, some nodes move higher in the tree and some lower. In a left rotation, the parent node moves down and its right child moves up one level. A double rotation, is made up of two single rotations, and one node moves up two levels, while all the others move up or down by at most one level. By beginning with the just-accessed target node and working up the path to the root, we could do single rotations at each step, thereby lifting the target node all the way to the root. This method would achieve the goal of making the target node into the root, but, it turns out, the performance of the tree amortized over many accesses may not be good.

Instead, the key idea of splaying is to move the target node two levels up the tree at each step. First some simple terminology: Consider the path going from the root down to the accessed node. Each time we move left going down this path, we say that we zig, and each time we move right we say that we zag. A move of two steps left (going down) is then called zig-zig, two steps right zag-zag, left then right zig-zag, and right then left zag-zig. These four cases are the only possibilities in moving two steps down the path. If the length of the path is odd, however, there will be one more step needed at its end, either a zig (left) move, or a zag (right) move.

The rotations done in splaying for each of zig-zig, zig-zag, and zig moves are shown following figures. The other three cases, zag-zag, zag-zig, and zag are mirror images of these.

Zig-Zig

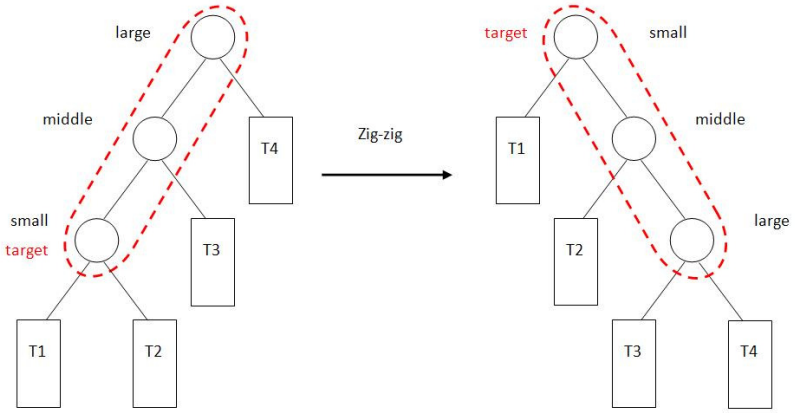


Figure 1. Zig-Zig move

Zig-Zag

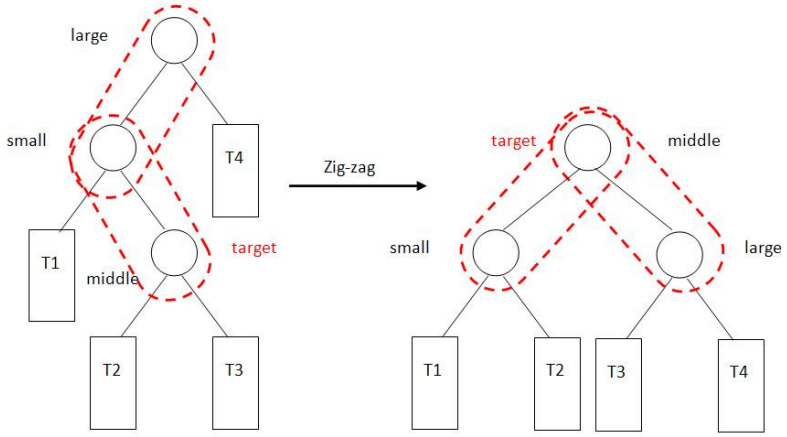


Figure 2. Zig-Zag move

Zig

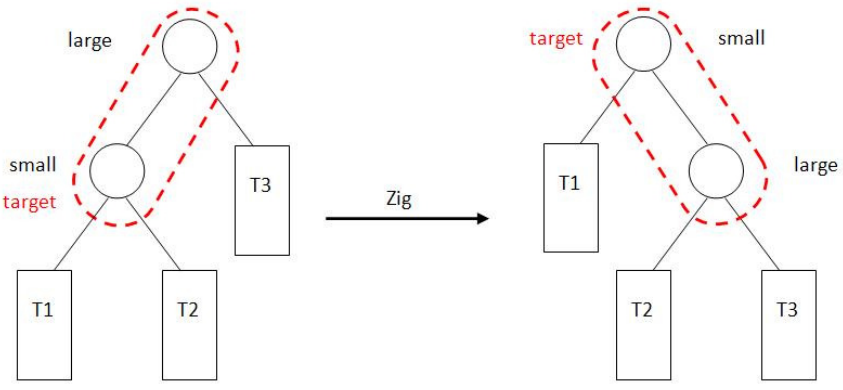


Figure 3. Zig move

1.3.2 Pseudocode for Operations in Splay Trees

```
MEMBER (S, k) {
        if  ( *S == NIL )
                then return 0
        current = *S
        pred = NIL
        while  ( current != NIL ){
                if  (k == curent->element)
                        then    break
                pred = curent
                if  ( k < curent->element )
                        then     current = curent->left
                else
                        current = curent->right
        }//end while
        if  ( current != NIL )
                then{
                        splay(S, curent)
                        return 1
                }
                else{
                        splay(S, pred)
                        return 0
}// end MEMBER function

INSERT  (S, k){
        if  (*S == NIL )
                then     (*S)->left = (*S)->right = (*S)->parent=NIL
                        (*S)->element = k
                        return
        current = *S
        anterior = NIL
        while  ( current != NIL ){
                if  ( k == curent->element )
                        return
                anterior = curent
                if  ( k  < curent->element )
                        then     directie=LEFT
                                current = curent->left
                else
                        directie = RIGHT
                        current = curent->right
        curent->element = k
        curent->left = curent->right = NIL
        curent->parent = anterior
        if  ( directie == LEFT )
                then     anterior->left = curent
        else
                anterior->right = curent
        splay(S, curent)
}
```

```
DELETE (S, k) {
        if ( MEMBER(S, k) ){
                then        SL = (*S)->left
                if ( SL )
                        then SL->parent = NIL
                SR = (*S)->right
                if ( SR )
                        then        SR->parent = NIL
                join(SL, SR, S)
        }
}

JOIN (SL, SR,  S){
        if ( SL )
                then        member( &SL, PLUS_INFINIT)
                            SL->right = SR
                            if ( SR )
                                    then    SR->parent = SL
                            *S = SL
        else
                if ( SR )
                        then        *S = SR
                else
                            *S = NIL
        return
}

SPLIT (S, SL, SR, in){
        If ( S )
                then        MEMBER( &S, in)
                            *SL = S
                            *SR = S->right
                            (*SL)->right = NIL
        else
                *SR = *SL = NIL
        return
}


SINGLEROTATE ( x ){
        if  ( x->parent->left  =  x )
                then      zig_left(&x)
        else
                zig_right(&x)
}

DOUBLEROTATE( x ){
        if  ( x->parent->left  ==  x and x->parent->parent->left  ==  x->parent)
                then       zig_zig_left(&x)
                           return
        if  ( x->parent->left  ==  x and x->parent->parent->right ==  x->parent)
                then      zig_zag_left(&x)
                           return
        if   (x->parent->right  ==  x and x->parent->parent->right == x->parent)
                then      zig_zig_right(&x)
                          return
```

```
        if   ( x->parent->right  ==  x and x->parent->parent->left  ==  x->parent)
                then      zig_zag_right(&x)
                          return
}

SPLAY( S,  crt){
        father = crt->parent
        while ( father  !=  NIL){
                if ( father->parent  ==  NIL)
                          then      SINGLEROTATE(crt)
                else
                          DOUBLEROTATE(crt)
                father = crt->parent
        }//end while
        *S = crt
}

ZIG_LEFT(x){
        splay_node *p, *b
        p = (*x)->parent
        b = (*x)->right
        (*x)->right = p
        (*x)->parent = NIL
        if ( b != NIL )
                b->parent = p
        p->left = b
        p->parent = (*x)
}

ZIG_RIGHT(x){
        splay_node *p, *b
        p = (*x)->parent
        b = (*x)->left
        (*x)->left = p
        (*x)->parent = NIL
        if ( b != NIL )
                b->parent = p
        p->right = b
        p->parent = (*x)
}

ZIG_ZIG_LEFT(x){
         splay_node *p, *g, *ggp
         splay_node *b, *c

        p = (*x)->parent
        g = p->parent
        b = (*x)->right
        c = p->right
        ggp = g->parent
        (*x)->right = p
        p->parent = (*x)
        p->right = g
        g->parent = p
        if ( b != NIL )
                b->parent = p
```

```
        p->left = b
        if ( c != NIL )
                c->parent = g
        g->left = c
        (*x)->parent = ggp
        if ( ggp != NIL )
                if ( ggp->left == g )
                        ggp->left = (*x)
                else
                        ggp->right = (*x)
}

ZIG_ZIG_RIGHT(x){
        splay_node *p, *g, *ggp
         splay_node *b, *c

        p = (*x)->parent
        g = p->parent
        b = (*x)->left
        c = p->left
        ggp = g->parent
        (*x)->left = p
        p->parent = (*x)
        p->left = g
        g->parent = p
        if ( b != NIL )
                b->parent = p
        p->right = b
        if ( c != NIL )
                c->parent = g
        g->right = c
        (*x)->parent = ggp
        if ( ggp != NIL )
                if ( ggp->left == g )
                        ggp->left = (*x)
                else
                        ggp->right = (*x)

}
ZIG_ZAG_LEFT (x){
        splay_node *p, *g, *ggp
        splay_node *a, *b

        p = (*x)->parent
        g = p->parent
        ggp = g->parent
        a = (*x)->left
        b = (*x)->right
        (*x)->left = g
        g->parent = (*x)
        (*x)->right = p
        p->parent = (*x)
        if ( a != NIL )
                a->parent = g
        g->right = a
        if ( b != NIL )
```

```
                b->parent = p
        p->left = b
        (*x)->parent = ggp
        if ( ggp != NIL )
                if ( ggp->left == g )
                        ggp->left = (*x)
                else
                        ggp->right = (*x)
}

ZIG_ZAG_RIGHT(x){
        splay_node *p, *g, *ggp
        splay_node *a, *b

        p = (*x)->parent
        g = p->parent
        ggp = g->parent
        a = (*x)->left
        b = (*x)->right
        (*x)->right = g
        g->parent = (*x)
        (*x)->left = p
        p->parent = (*x)
        if ( a != NIL )
                a->parent = p
        p->right = a
        if ( b != NIL )
                b->parent = g
        g->left = b
        (*x)->parent = ggp
        if ( ggp != NIL)
                if ( ggp->left == g)
                        ggp->left = (*x)
                else
                        ggp->right = (*x)
}
```

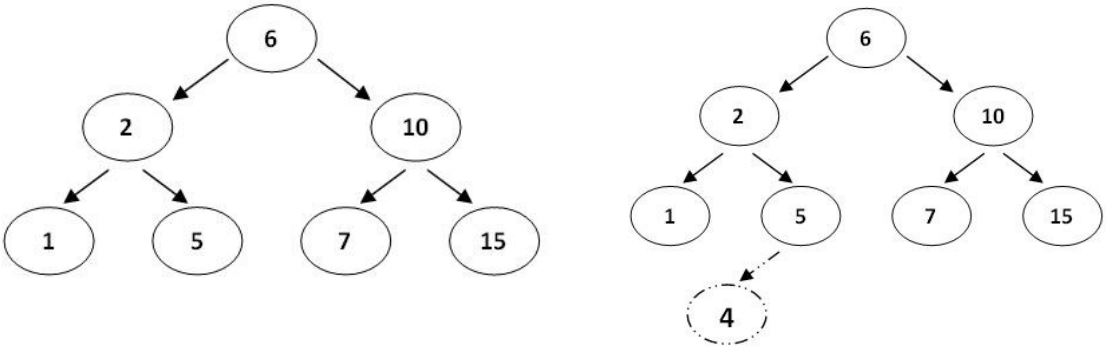
1.3.3 Sample Operations in a Splay Tree

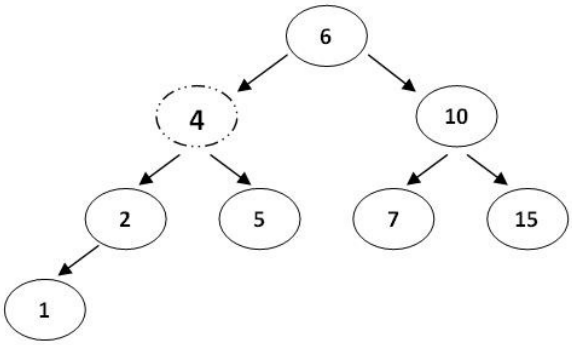
1.3.3.1 Insertion Example

Figure 4.a presents the initial splay tree in which it will be performed the insertion of key 4. At first step, key 4 is inserted as leaf node. Figure 4.b presents the shape of the splay tree after setting node with key 4 as left child of node with key 5. After this step there will be performed rotations that will push the node with key 4 up to the root position.

Figure 4.c presents the splay tree after performing a *zig-zag* rotation on node with key 4. The effect of this rotation is lifting up the node with key 4 with two levels.

Figure 4.d presents the splay tree after performing a *zig* rotation on node with key 4. The effect of this rotation is lifting up the node with key 4 with one level in the root position.





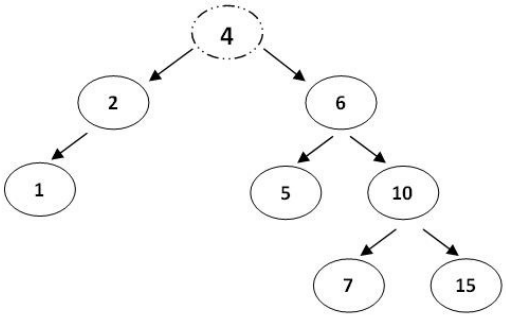


Figure 4. Insertion steps in a splay tree

1.3.3.2 Join Example

There are given two splay trees. They are presented in figures 5.a and 5.b. The figure 5.c presents the resulted tree.

The first step is to perform a MEMBER operation on the largest key in the first splay tree. This operation will bring node with key 15 in the root position for the first splay tree.

The second step is to link the root of the second splay tree as right child of the previously obtained root for the first slay tree. After this step the tree from figure 5.c is obtained.

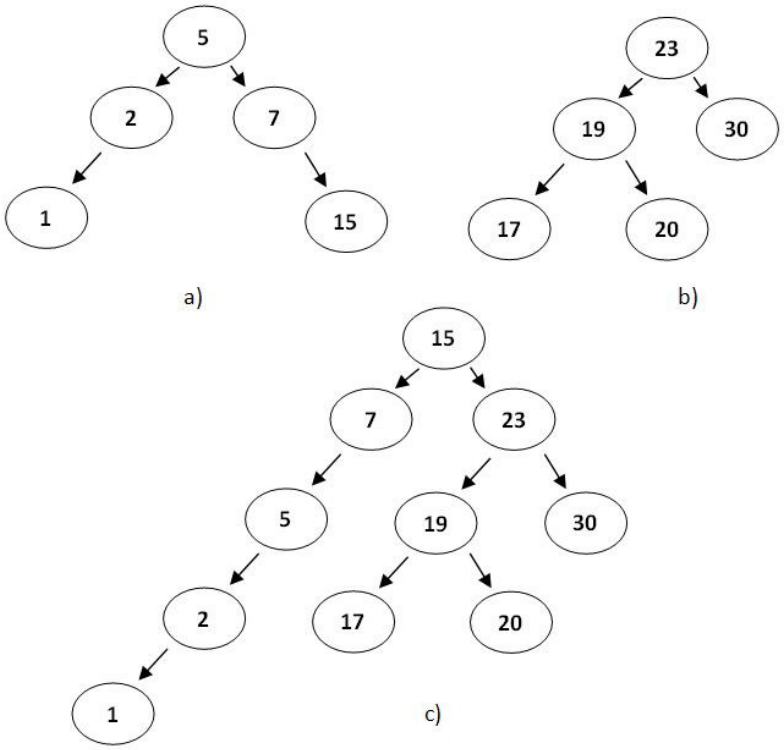


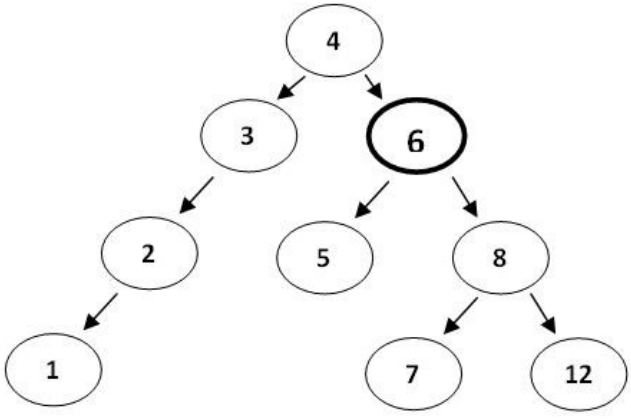
Figure 5. Join example

1.3.3.3 Split Example

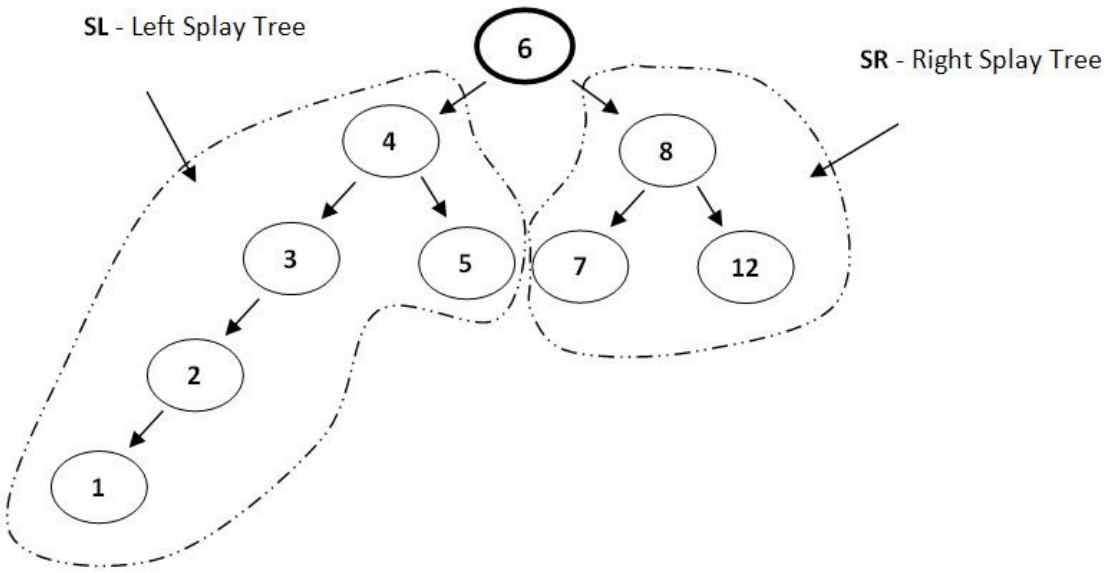
There is given a splay tree and a key that belong to it. Figure 6.a presents this initial splay tree and the splitting key is 6. The split function will produce two splay trees: one with keys smaller than 6 and one with keys greater than 6.

The first step is to perform a MEMBER operation on node with key 6. After this operation is performed the splay tree will look like in figure 6.b.

Once the splitting key has been lifted up as root node the needed left splay tree and right splay tree are represented by the left and right children of the root.



a)

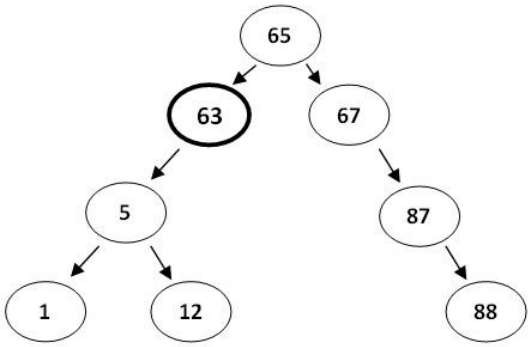


b)

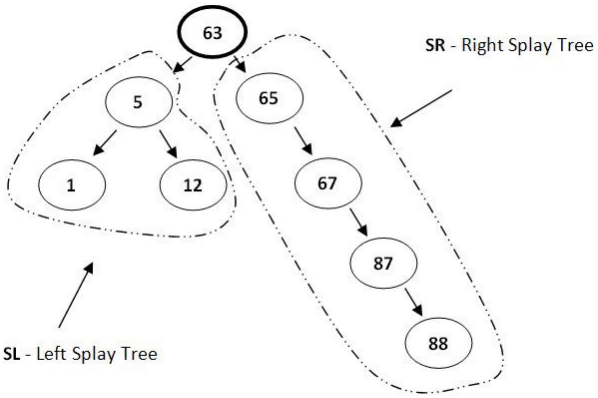
Figure 6. Split example

1.3.3.4 Deletion Example

There is given a splay tree as presented in figure 7.a. The task is to delete node with key 63 from the splay tree. The first step is to perform a MEMBER operation on node with key 63. This step will bring node with key 63 as root node. After this operation the splay tree will look like in figure 7.b

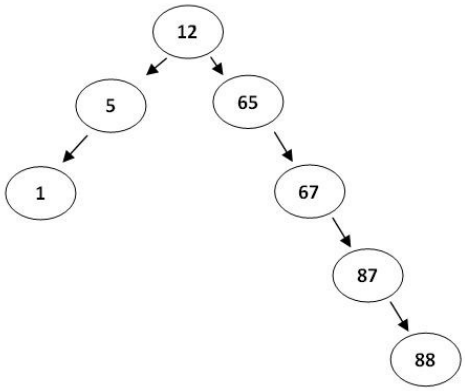


a)



b)

The second step is to perform a join operation on left and right subtrees of node 63. These subtrees are marked in figure 7.b. The outcome of this procedure is presented in figure7.c.

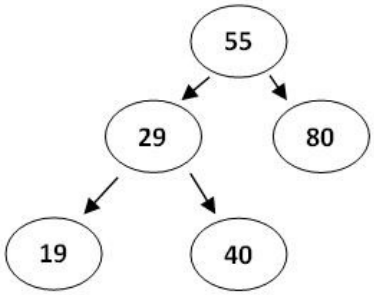


c)

Figure 7. Deletion example

2. Assignments

1) Write a program that creates and manages a splay tree. The program must implement the following operations: creation, insertion, deletion, join, split, and tree display. The program should present a meniu where user may choose from implemented options.
2) It is given the Splay 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 24
- DELETE 55
- SPLIT 40

For each insertion operation there must be presented the following:
- Initial position of the key into the Splay tree
- Each 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
- Each rotation type that needs to be performed and how the rotation will perform
- The Join operation of the left and right subtrees
- The final shape of the tree

For the split operation there must be presented the following:
- The search of the tree for the key used for the split
- Each rotation type that needs to be performed and how the rotation will perform
- The obtained left and right subtrees

3) Perform a join operation on the following splay trees.