INFORMATIKA UMN

UMN
UNIVERSITAS
MULTIMEDIA
NUSANTARA

IF232
ALGORITHMS
&
DATA STRUCTURES

09

EFFICIENT BINARY TREES

DENNIS GUNAWAN

REVIEW

Trees:

Basic Terminology

Types of Trees

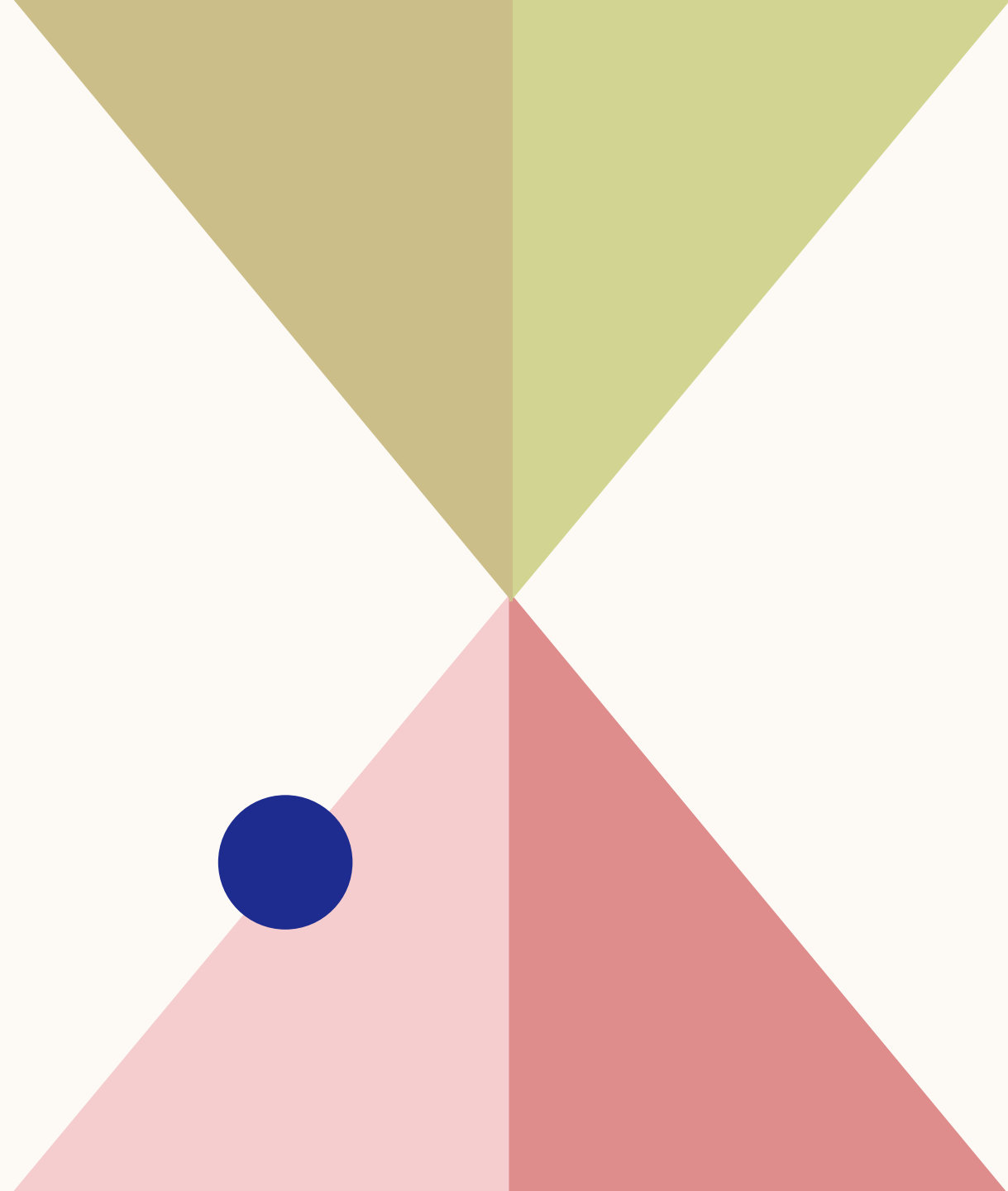
Traversing a Binary Tree

Applications of Trees

OUTLINE

Binary Search Trees

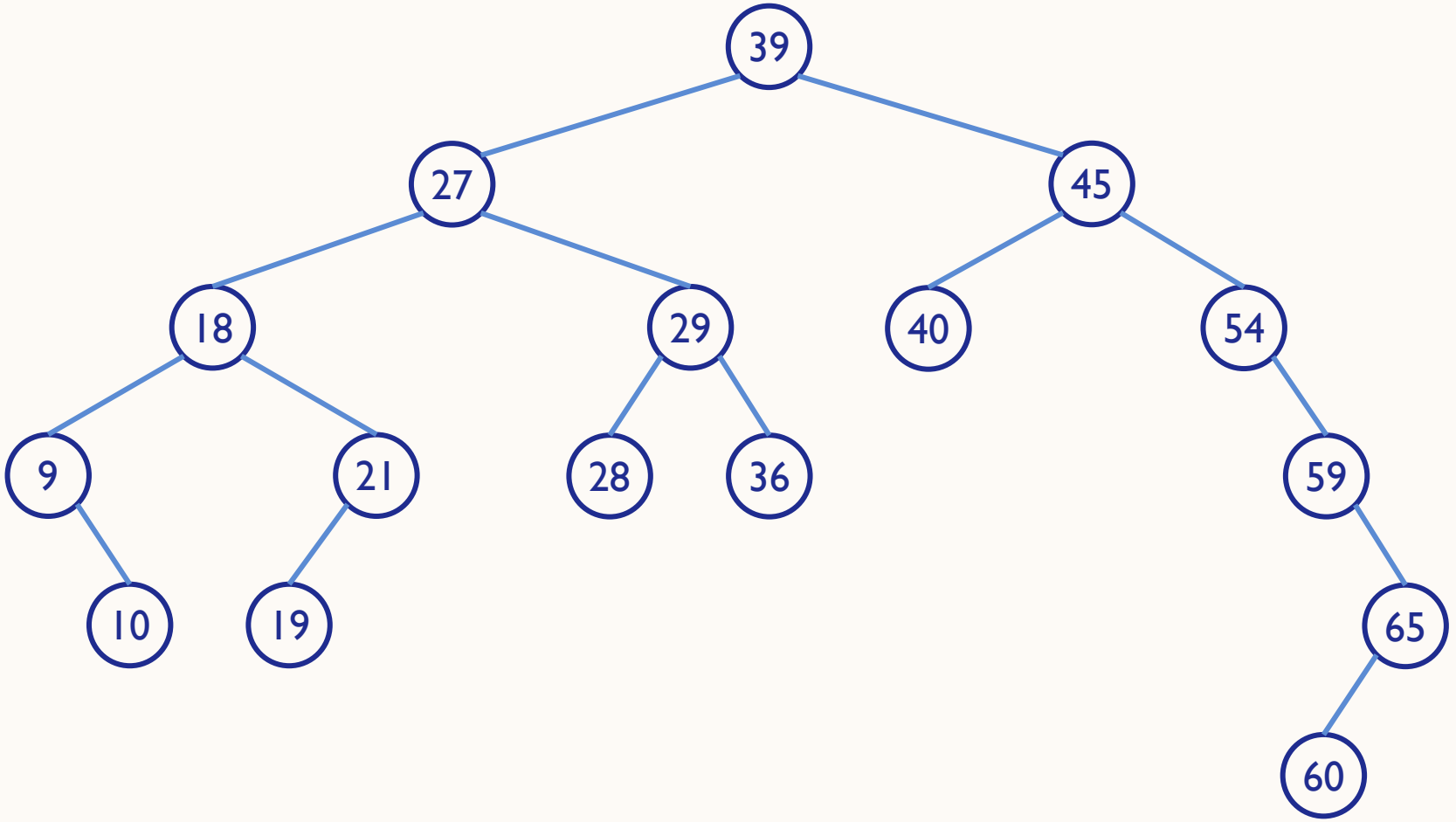
AVL Trees



BINARY SEARCH TREES

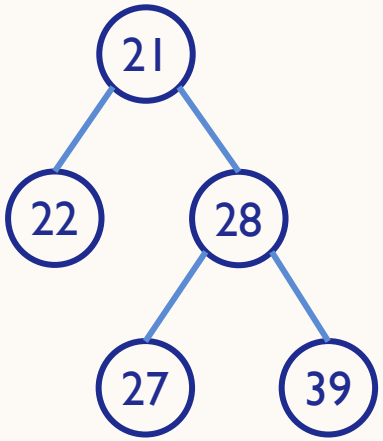
- A binary search tree, also known as an ordered binary tree, is a variant of binary trees in which the nodes are arranged in an order
- Since the nodes in a binary search tree are ordered, the time needed to search an element in the tree is greatly reduced
- Binary search trees also speed up the insertion and deletion operations
  - The tree has a speed advantage when the data in the structure changes rapidly
- Binary search trees are considered to be efficient data structures especially when compared with sorted linear arrays and linked lists

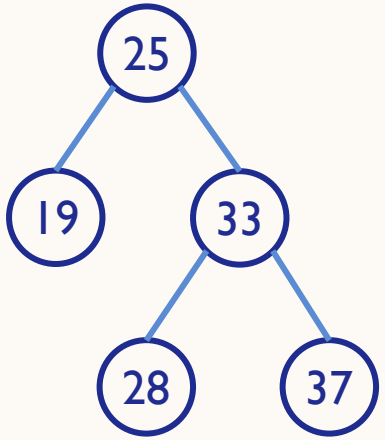
BINARY SEARCH TREES

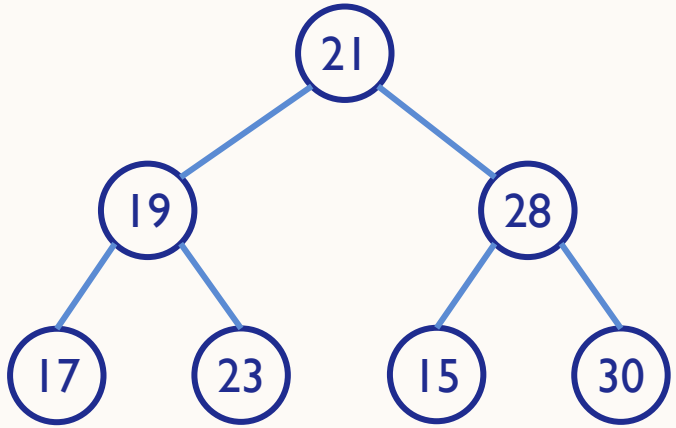


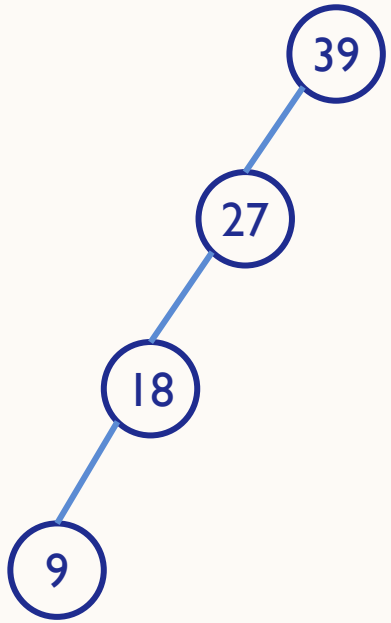
- The **left** sub-tree of a node N contains values that are **less** than N's value
- The **right** sub-tree of a node N contains values that are **greater** than N's value

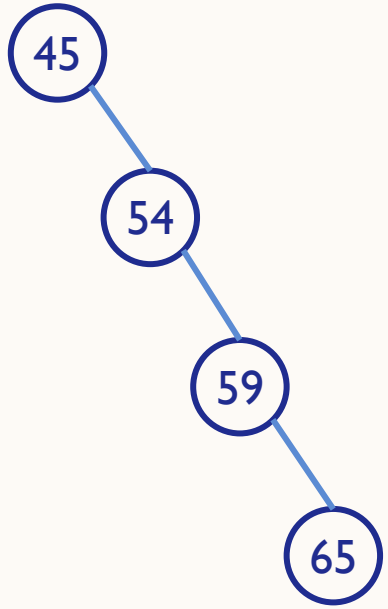
BINARY SEARCH TREES?







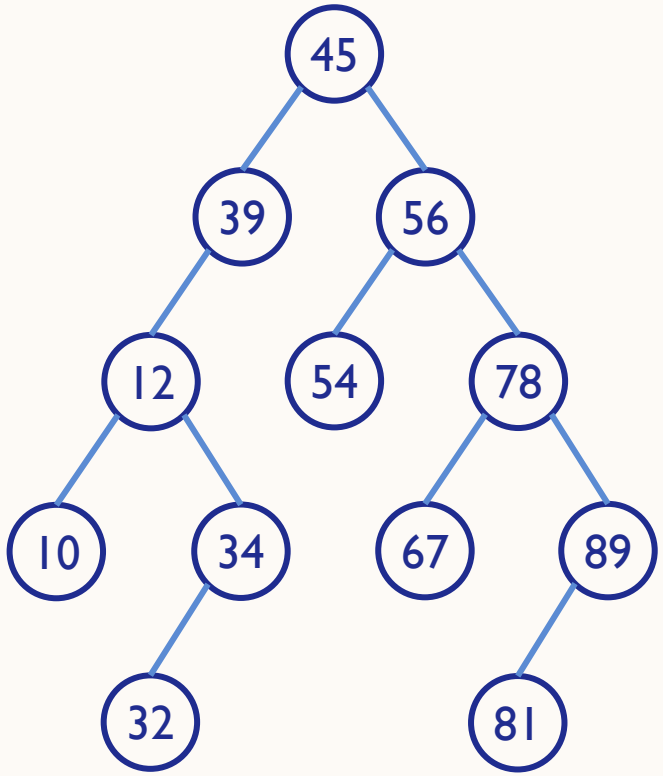




BINARY SEARCH TREES

Create a binary search tree using the following data elements:

`45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67, 81`



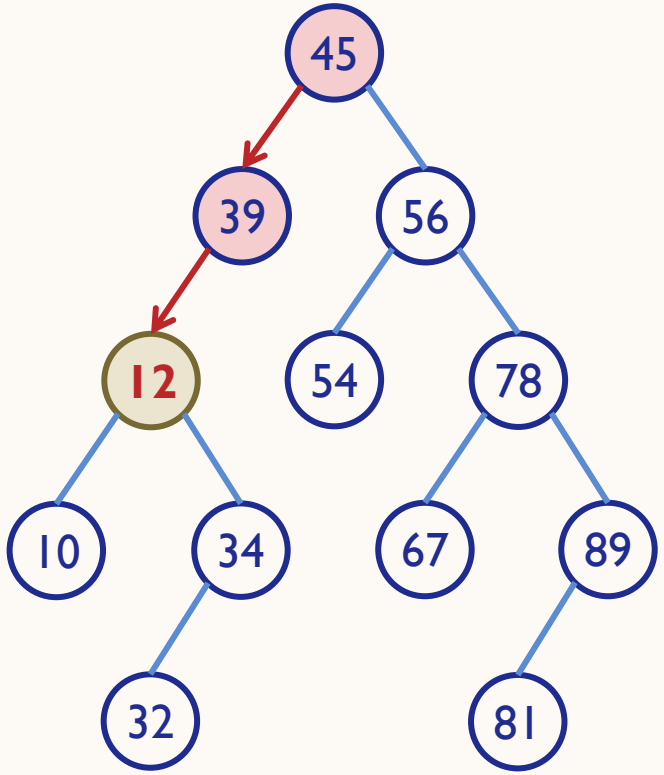
DECLARATION

```
struct tbst{
    struct tbst *left;
    int data;
    struct tbst *right;
};
```

```
int main()
{
    struct tbst *root, *node, *curr, *parent;
    int number;
    ...
    root = NULL;
    ...
}
```

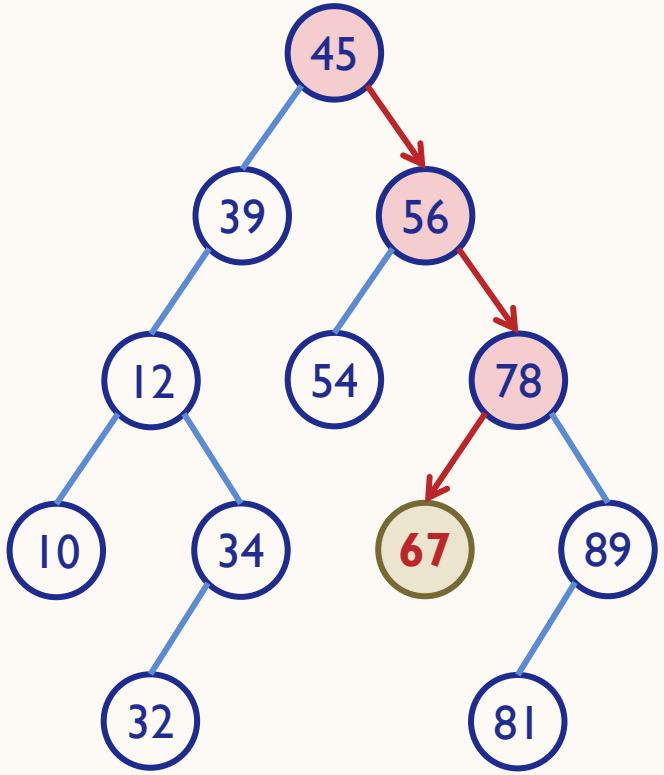
SEARCHING FOR A NODE IN A BINARY SEARCH TREE

```
curr = root;

while(curr != NULL && curr->data != number) //12
    if(curr->data > number)
        curr = curr->left;
    else
        curr = curr->right;
```



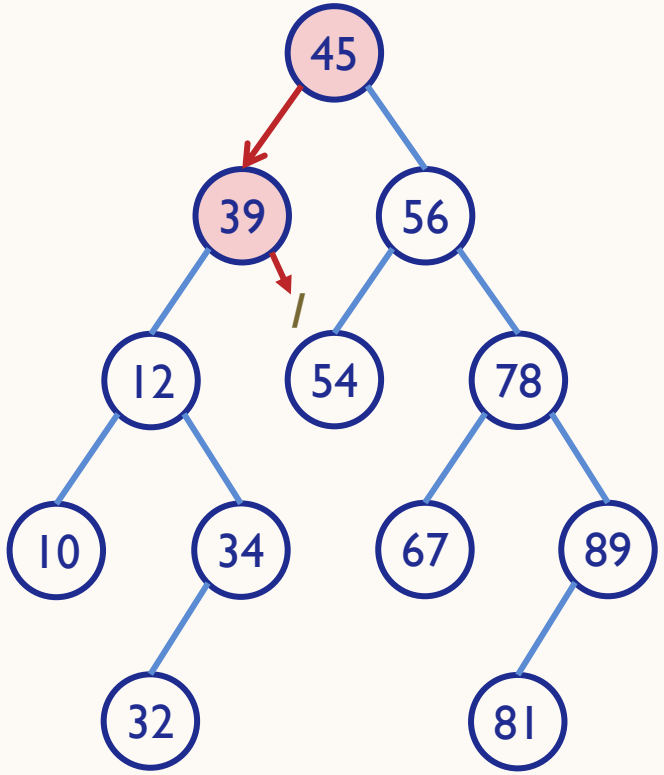
SEARCHING FOR A NODE IN A BINARY SEARCH TREE

```
curr = root;

while(curr != NULL && curr->data != number) //67
    if(curr->data > number)
        curr = curr->left;
    else
        curr = curr->right;
```



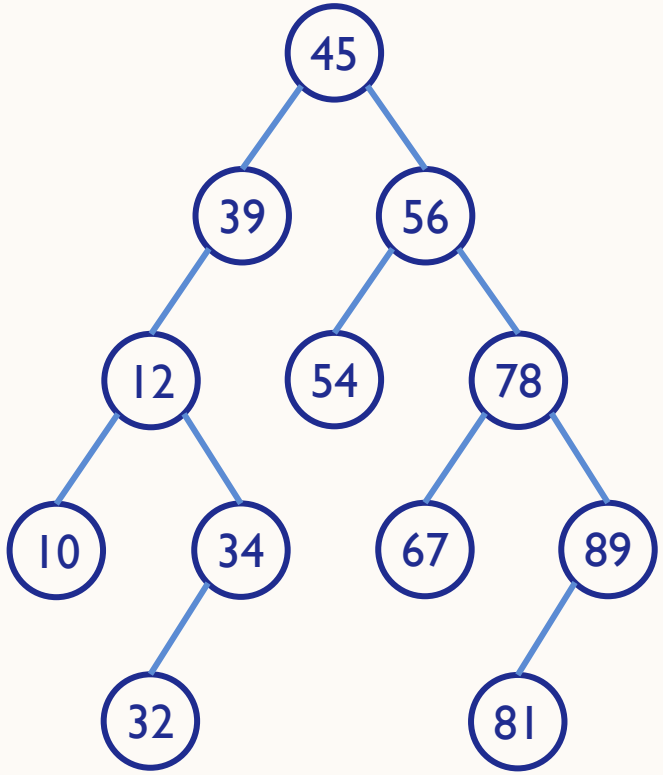
SEARCHING FOR A NODE IN A BINARY SEARCH TREE

```
curr = root;

while(curr != NULL && curr->data != number) //40
    if(curr->data > number)
        curr = curr->left;
    else
        curr = curr->right;
```



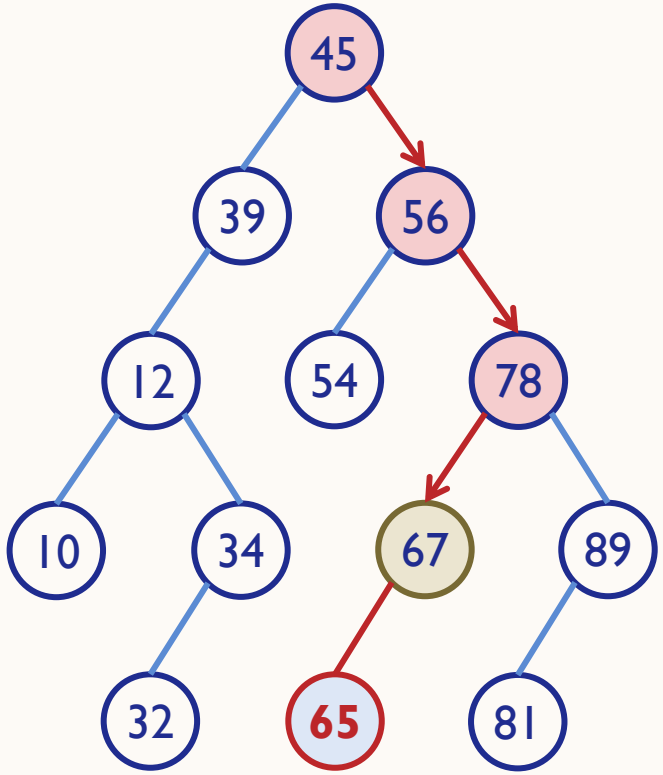
SEARCHING FOR A NODE IN A BINARY SEARCH TREE

```
struct tbst * search(struct tbst *root, int number){
    if(root == NULL || root->data == number)
        return root;
    if(root->data > number)
        return search(root->left, number);
    else
        return search(root->right, number);
}
```



INSERTING A NEW NODE IN A BINARY SEARCH TREE

```
node = (struct tbst *) malloc(sizeof(struct tbst));
node->data = number; //65
node->left = node->right = NULL;

if(root == NULL) root = node;
else{
    curr = root;
    while(curr != NULL){
        parent = curr;
        if(curr->data > number) curr = curr->left;
        else curr = curr->right;
    }
    if(parent->data > number) parent->left = node;
    else parent->right = node;
}
```



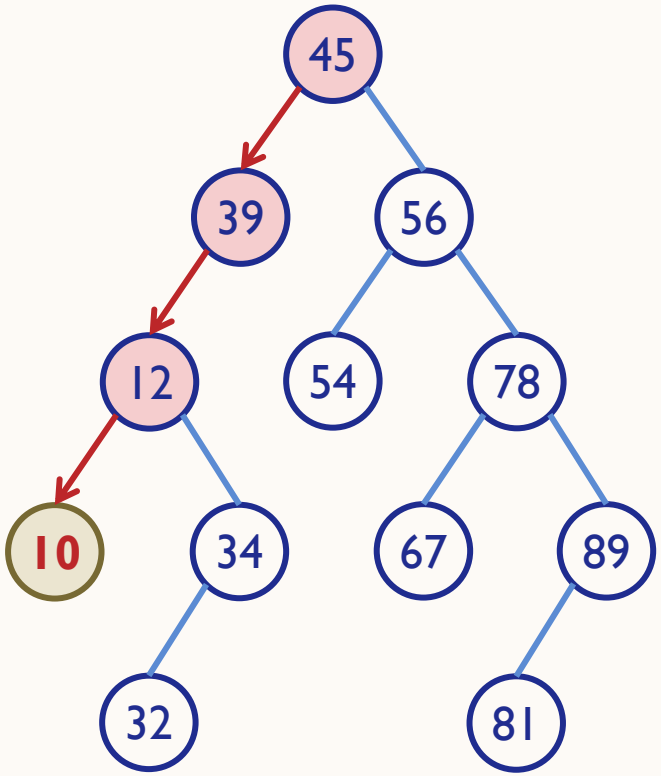
MINIMUM & MAXIMUM

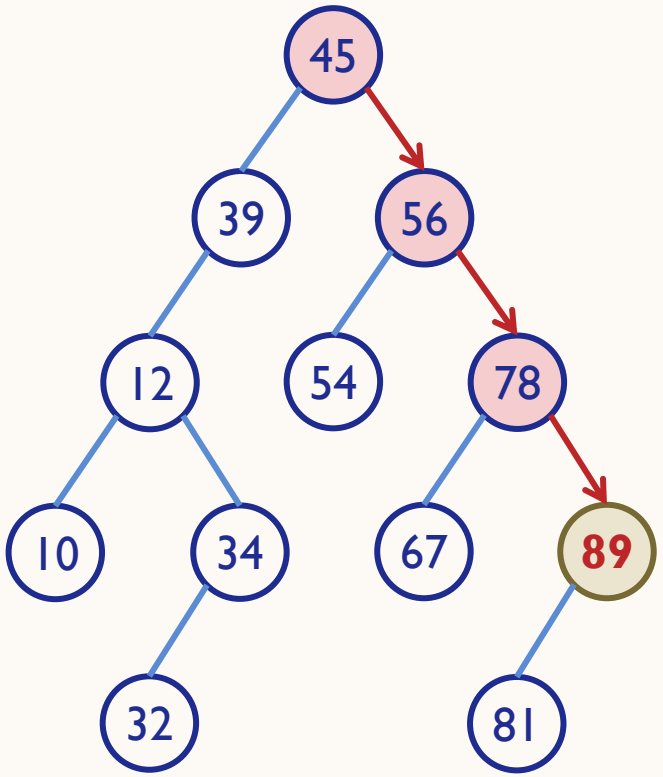
MIN

```
curr = root;
parent = NULL;

while(curr != NULL){
    parent = curr;
    curr = curr->left;
}
```

MAX

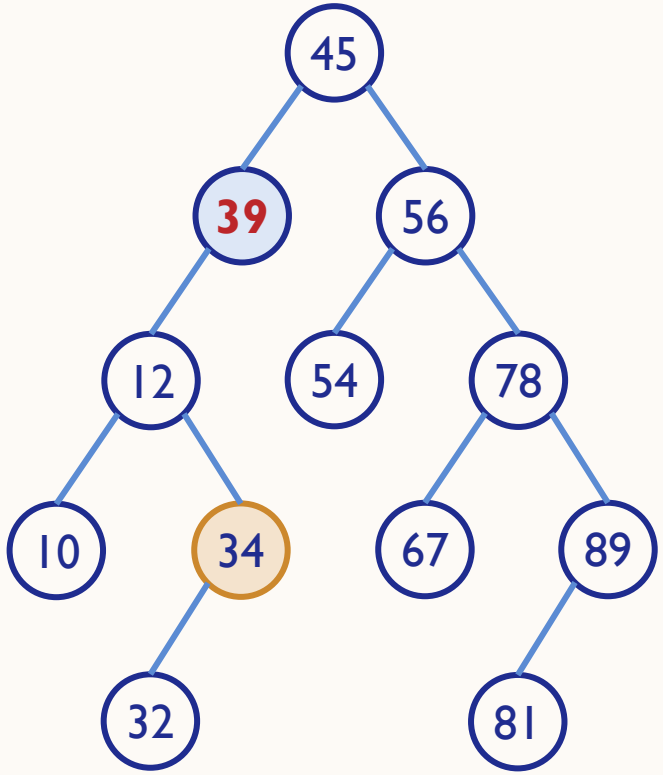
```
curr = root;
parent = NULL;

while(curr != NULL){
    parent = curr;
    curr = curr->right;
}
```





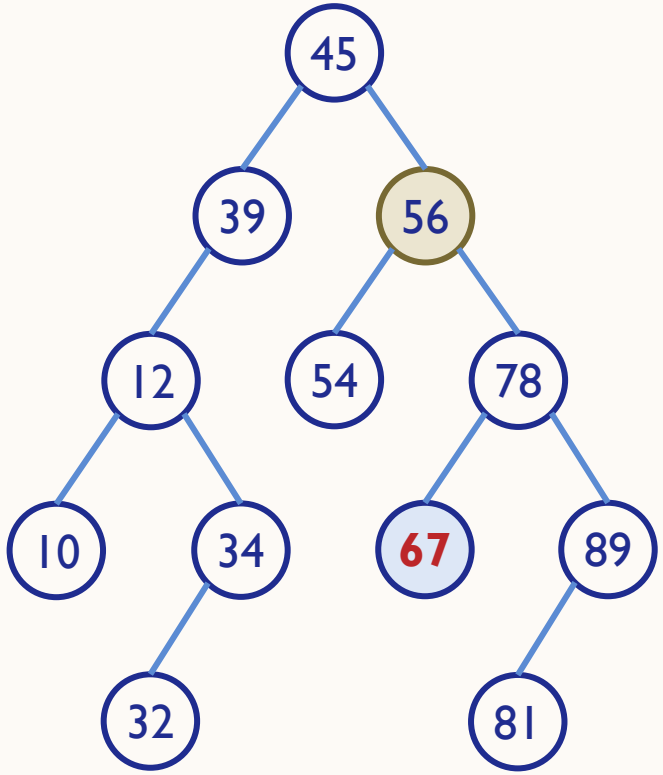
PREDECESSOR

```
struct tbst * pred(struct tbst *root, int number){ //39
    struct tbst *curr, *parent;

    curr = search(root, number);
    if(curr->left != NULL)
        return maximum(curr->left);

    parent = parentSearch(root, curr);
    while(parent != NULL && parent->left == curr){
        curr = parent;
        parent = parentSearch(root, curr);
    }
    return parent;
}
```



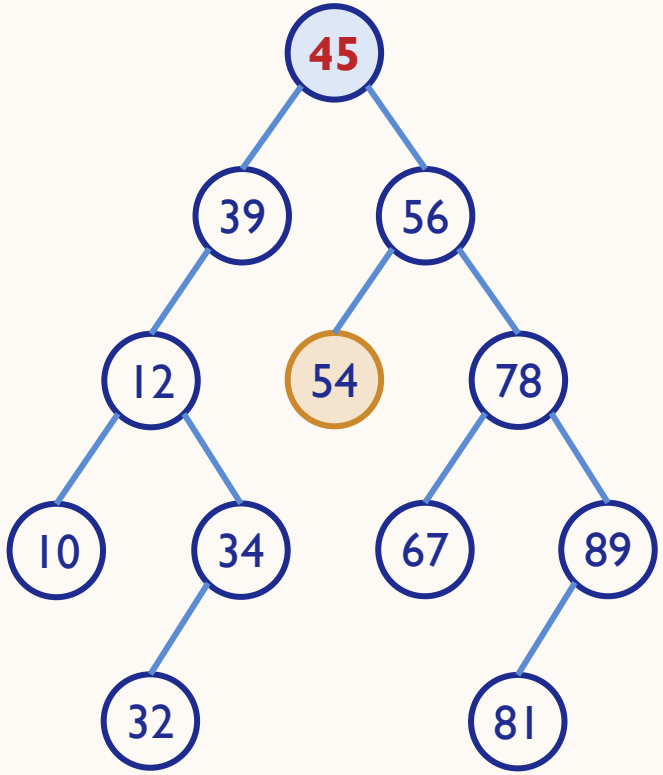
PREDECESSOR

```
struct tbst * pred(struct tbst *root, int number){ //67
    struct tbst *curr, *parent;

    curr = search(root, number);
    if(curr->left != NULL)
        return maximum(curr->left);

    parent = parentSearch(root, curr);
    while(parent != NULL && parent->left == curr){
        curr = parent;
        parent = parentSearch(root, curr);
    }
    return parent;
}
```



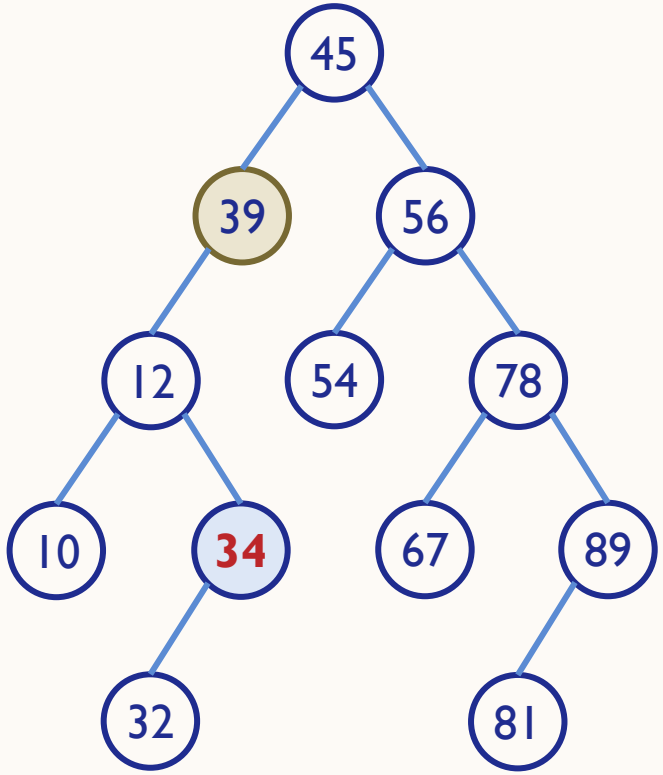
SUCCESSOR

```
struct tbst * succ(struct tbst *root, int number){ //45
    struct tbst *curr, *parent;

    curr = search(root, number);
    if(curr->right != NULL)
        return minimum(curr->right);

    parent = parentSearch(root, curr);
    while(parent != NULL && parent->right == curr){
        curr = parent;
        parent = parentSearch(root, curr);
    }
    return parent;
}
```



SUCCESSOR

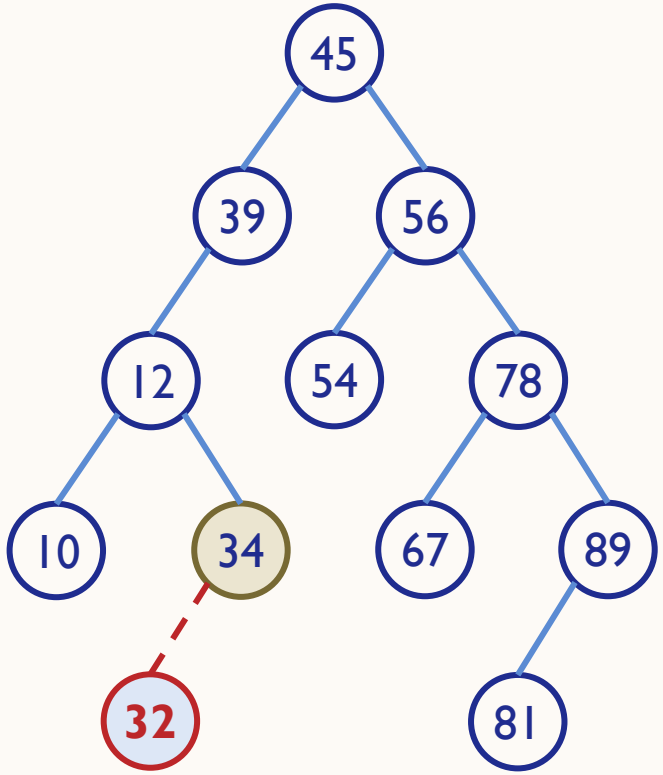
```
struct tbst * succ(struct tbst *root, int number){ //34
    struct tbst *curr, *parent;

    curr = search(root, number);
    if(curr->right != NULL)
        return minimum(curr->right);

    parent = parentSearch(root, curr);
    while(parent != NULL && parent->right == curr){
        curr = parent;
        parent = parentSearch(root, curr);
    }
    return parent;
}
```



DELETING A NODE FROM A BINARY SEARCH TREE

```
curr = search(root, number);

if(curr != NULL){
    if(curr->left == NULL && curr->right == NULL){
        parent = parentSearch(root, curr);
        if(parent->left == curr)
            parent->left = NULL;
        else
            parent->right = NULL;
        free(curr); //32
    }
```

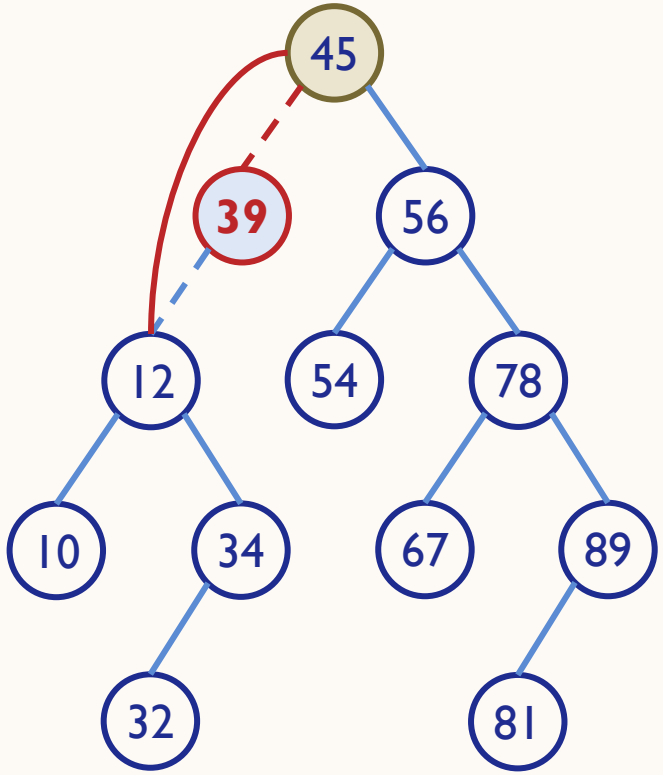
DELETING A NODE THAT HAS NO CHILDREN



DELETING A NODE FROM A BINARY SEARCH TREE

```
else if(curr->left == NULL || curr->right == NULL){
    parent = parentSearch(root, curr);
    if(parent->left == curr)
        if(curr->left == NULL)
            parent->left = curr->right;
        else
            parent->left = curr->left;
    else
        if(curr->left == NULL)
            parent->right = curr->right;
        else
            parent->right = curr->left;
    free(curr); //39
}
```

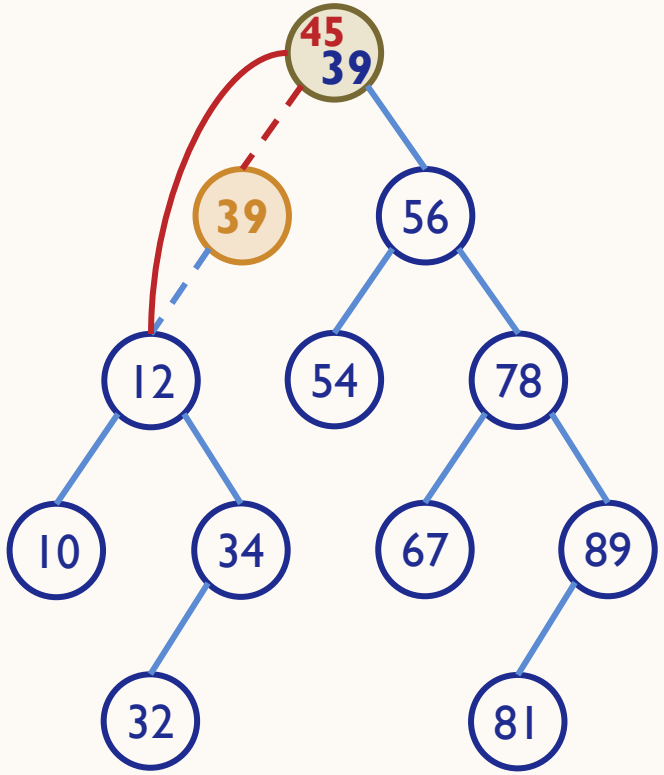
DELETING A NODE WITH ONE CHILD



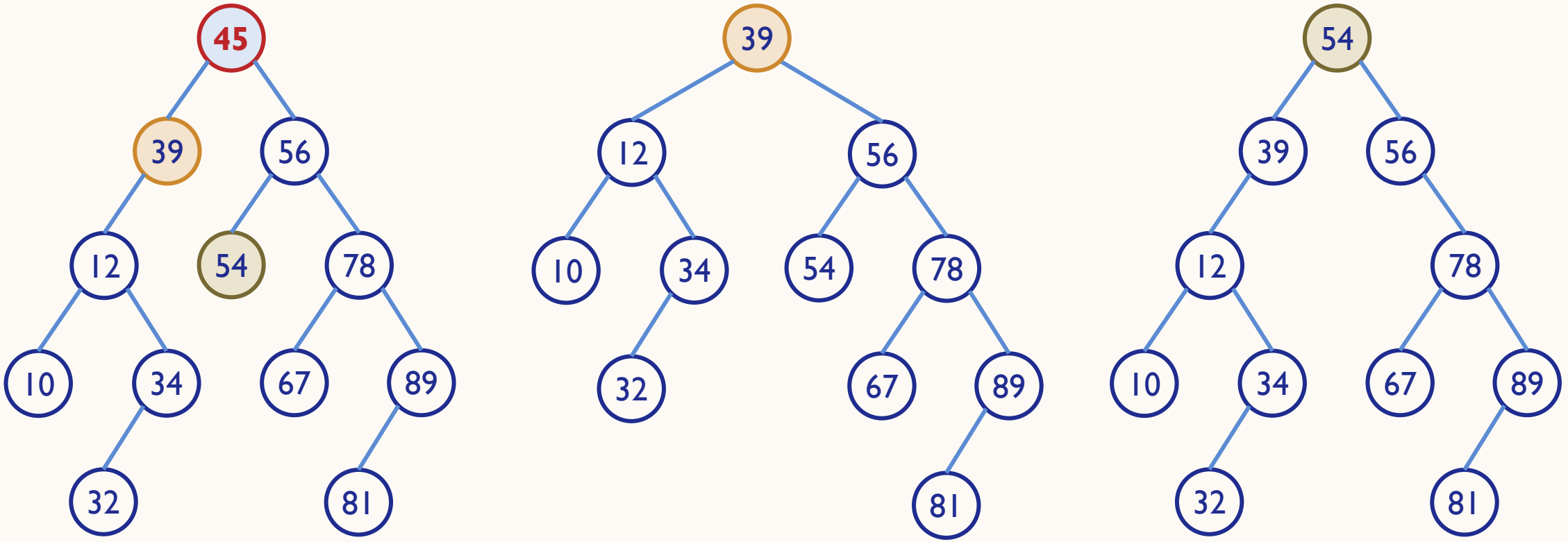
DELETING A NODE FROM A BINARY SEARCH TREE

```
    else if(curr->left != NULL && curr->right != NULL){
        node = pred(root, number);
        parent = parentSearch(root, node);
        curr->data = node->data;
        if(parent != curr)
            parent->right = node->left;
        else
            curr->left = node->left;
        free(node); //45 39
    }
}
```

DELETING A NODE WITH TWO CHILDREN



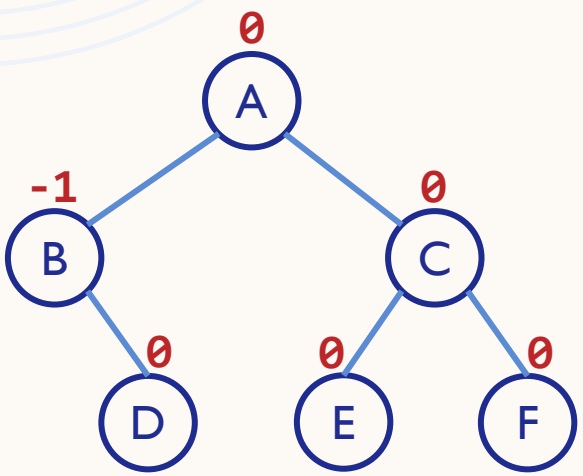
DELETING A NODE WITH TWO CHILDREN

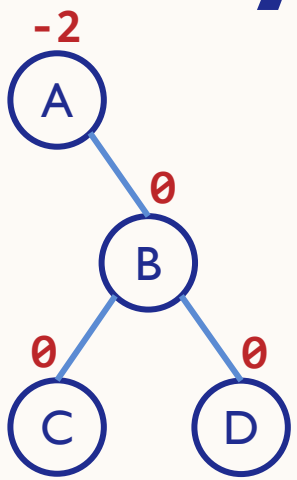


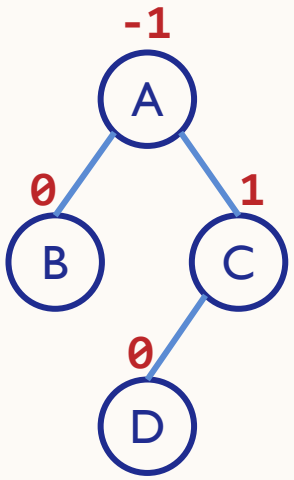
AVL TREES

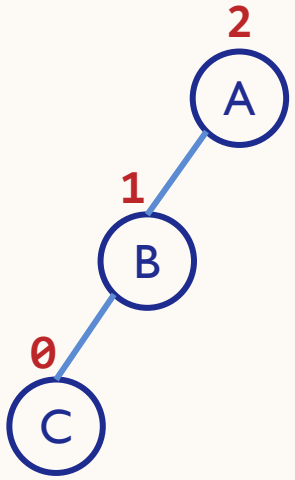
- AVL tree is a **self-balancing binary search tree**
- Also known as a **height-balanced tree**, the **heights** of the two sub-trees of a node may differ by **at most one**
- In its structure, it stores an additional variable called the `BalanceFactor`
  - Every node has a balance factor associated with it

AVL TREES







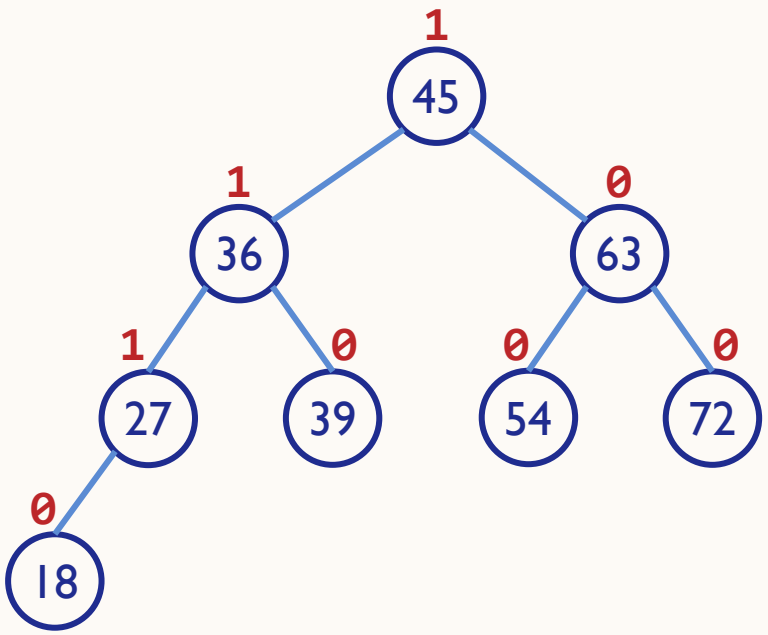


Balance Factor
= Height(left sub-tree) – Height(right sub-tree)

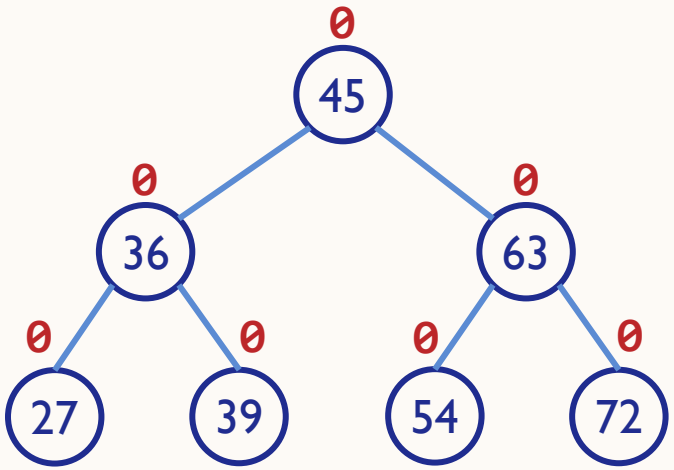
- A binary search tree in which every node has a balance factor of **–1**, **0**, or **1** is said to be height balanced
- A node with any other balance factor is considered to be unbalanced and requires rebalancing of the tree

AVL TREES

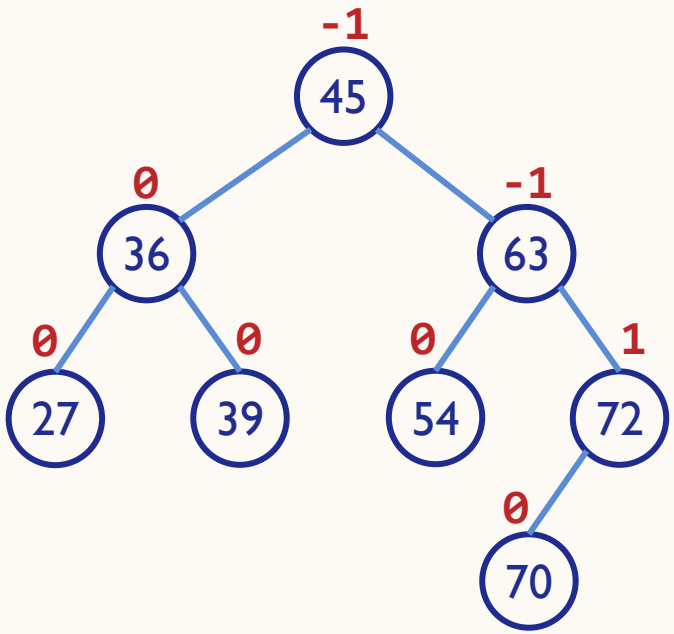
Left-Heavy AVL Tree



Balanced AVL Tree

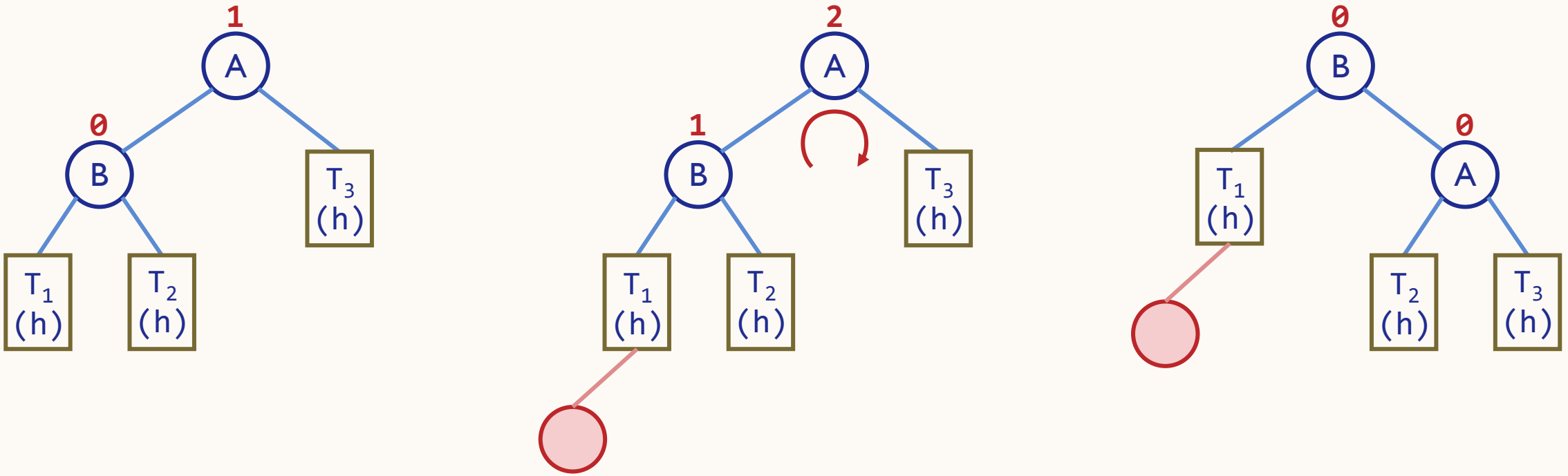


Right-Heavy AVL Tree



INSERTING A NEW NODE IN AN AVL TREE

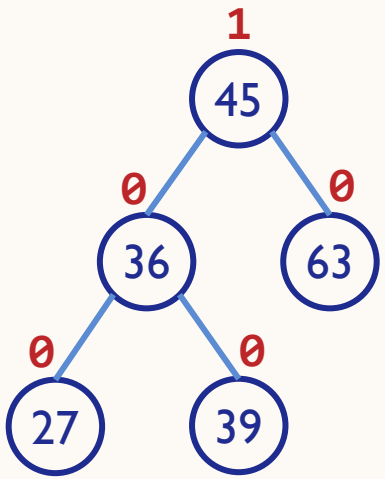
LL ROTATION

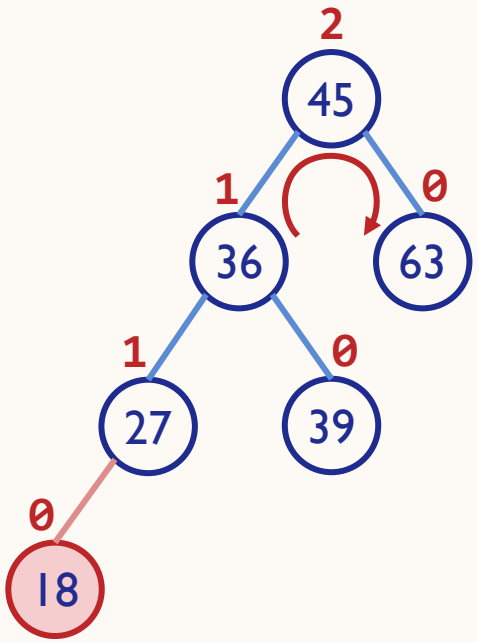


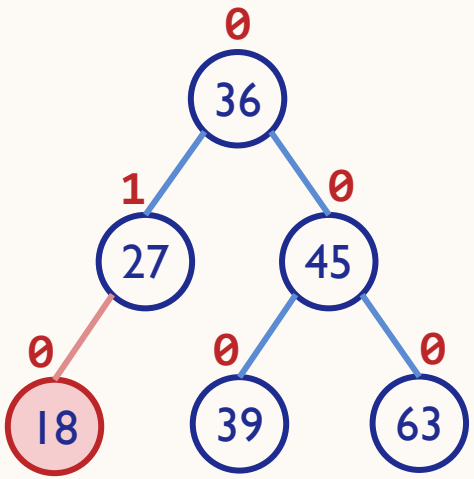
INSERTING A NEW NODE IN AN AVL TREE

LL ROTATION

Consider the AVL tree given below and insert 18 into it.

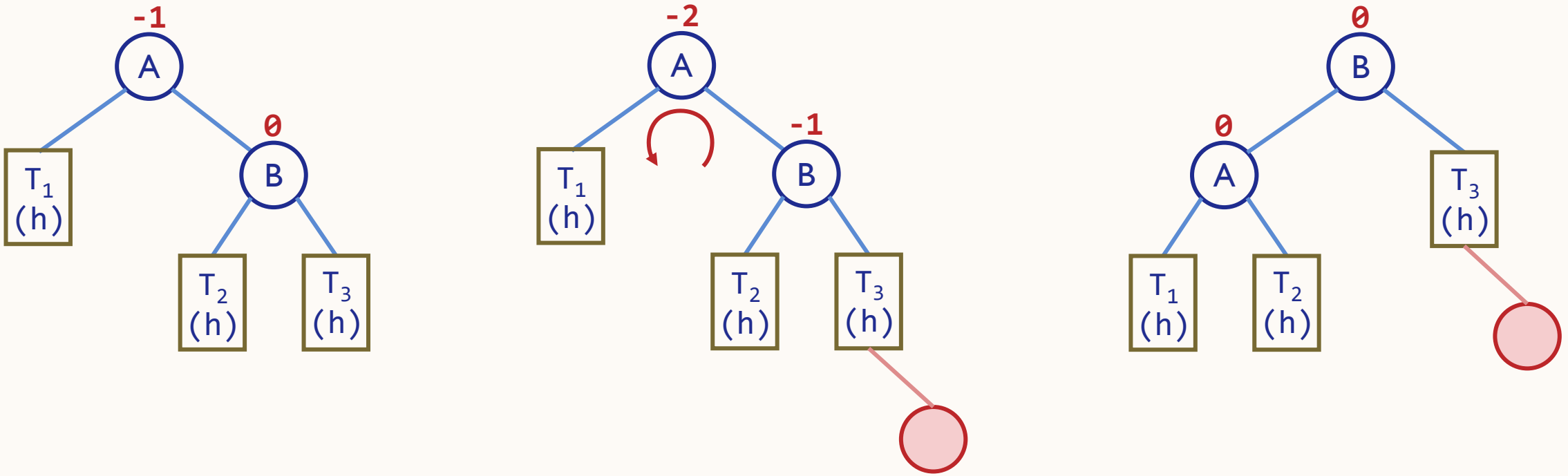






INSERTING A NEW NODE IN AN AVL TREE

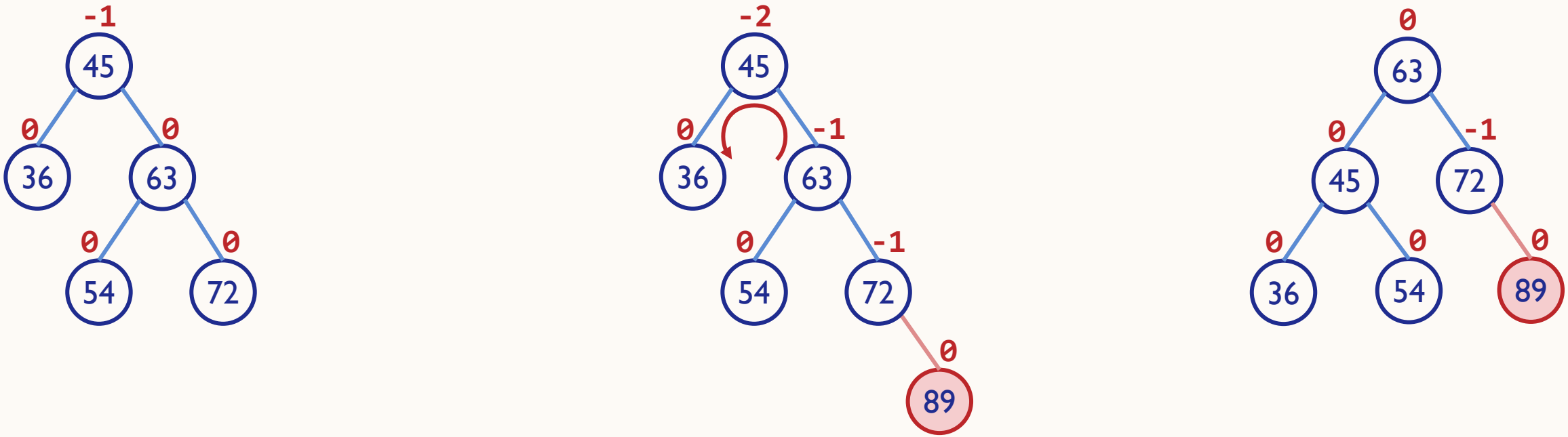
RR ROTATION



INSERTING A NEW NODE IN AN AVL TREE

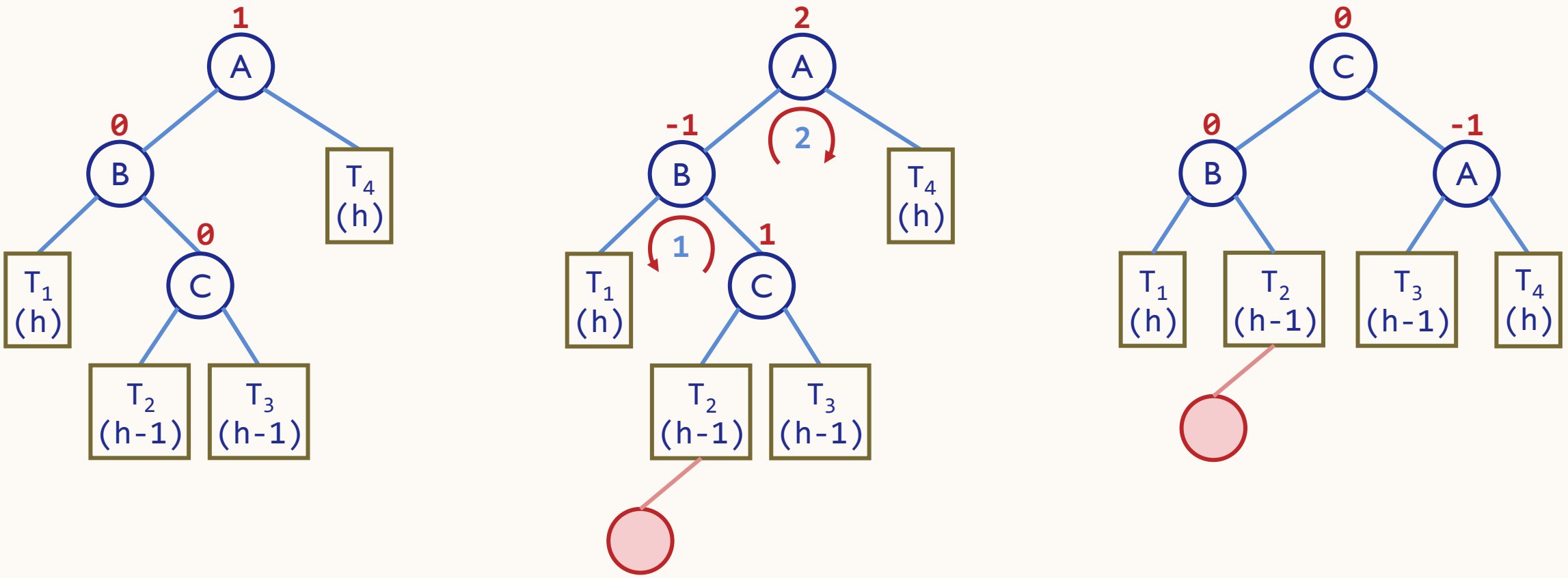
RR ROTATION

Consider the AVL tree given below and insert 89 into it.



INSERTING A NEW NODE IN AN AVL TREE

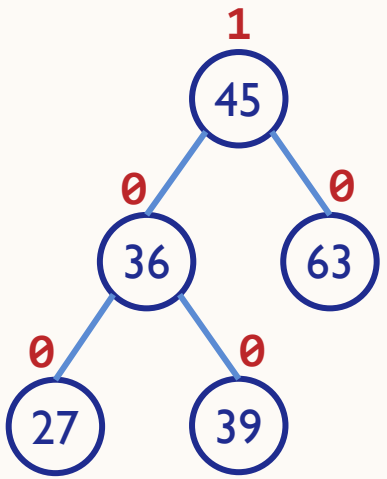
LR ROTATION

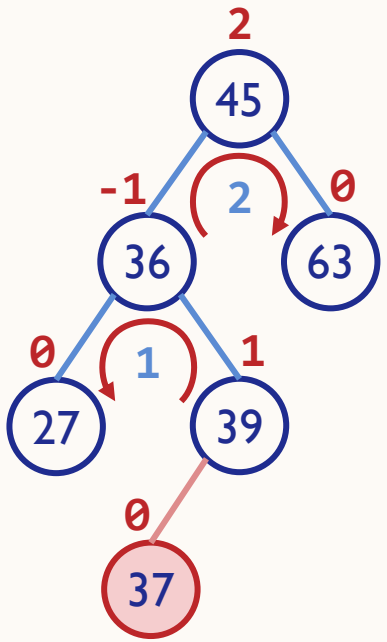


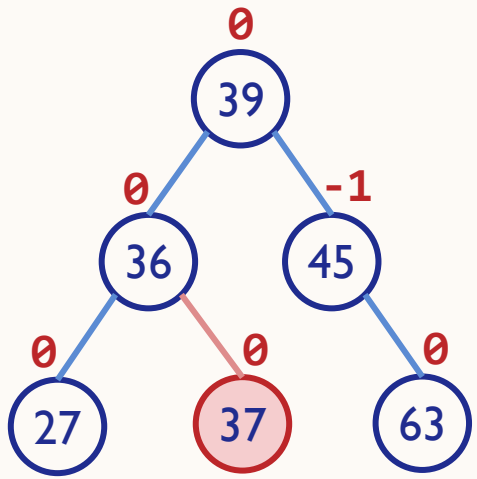
INSERTING A NEW NODE IN AN AVL TREE

LR ROTATION

Consider the AVL tree given below and insert 37 into it.

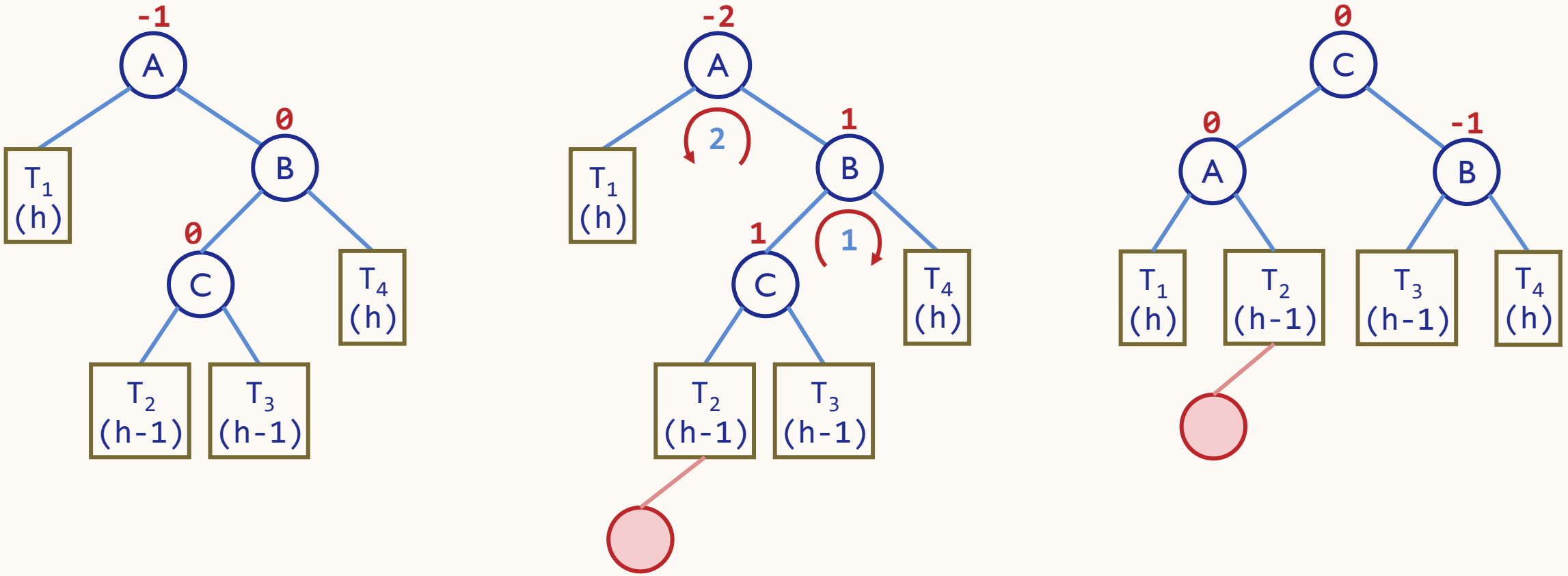






INSERTING A NEW NODE IN AN AVL TREE

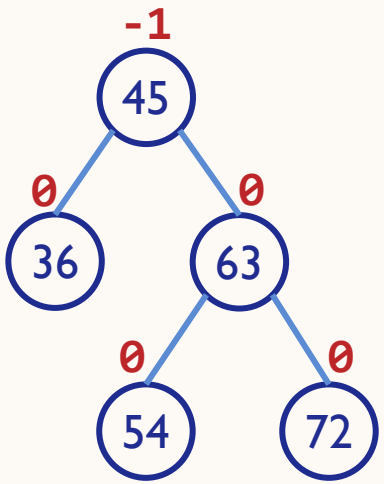
RL ROTATION

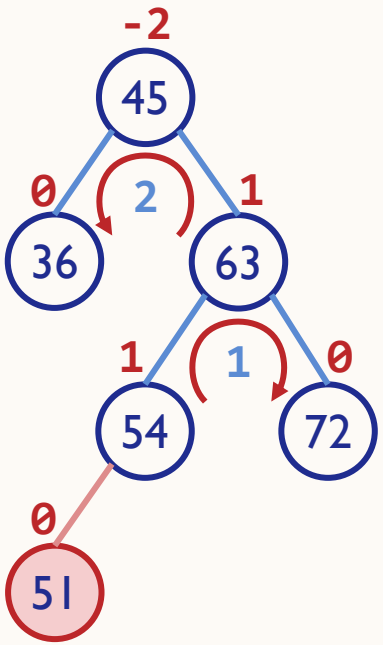


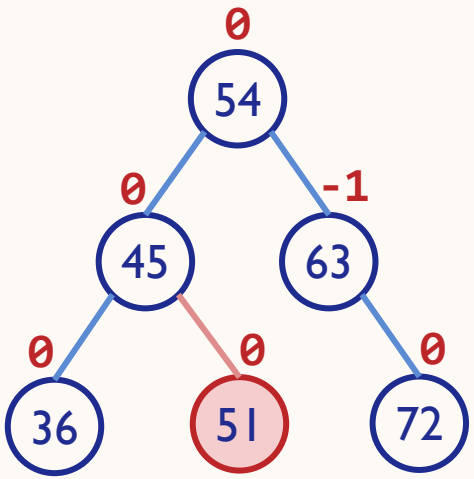
INSERTING A NEW NODE IN AN AVL TREE

RL ROTATION

Consider the AVL tree given below and insert 51 into it.

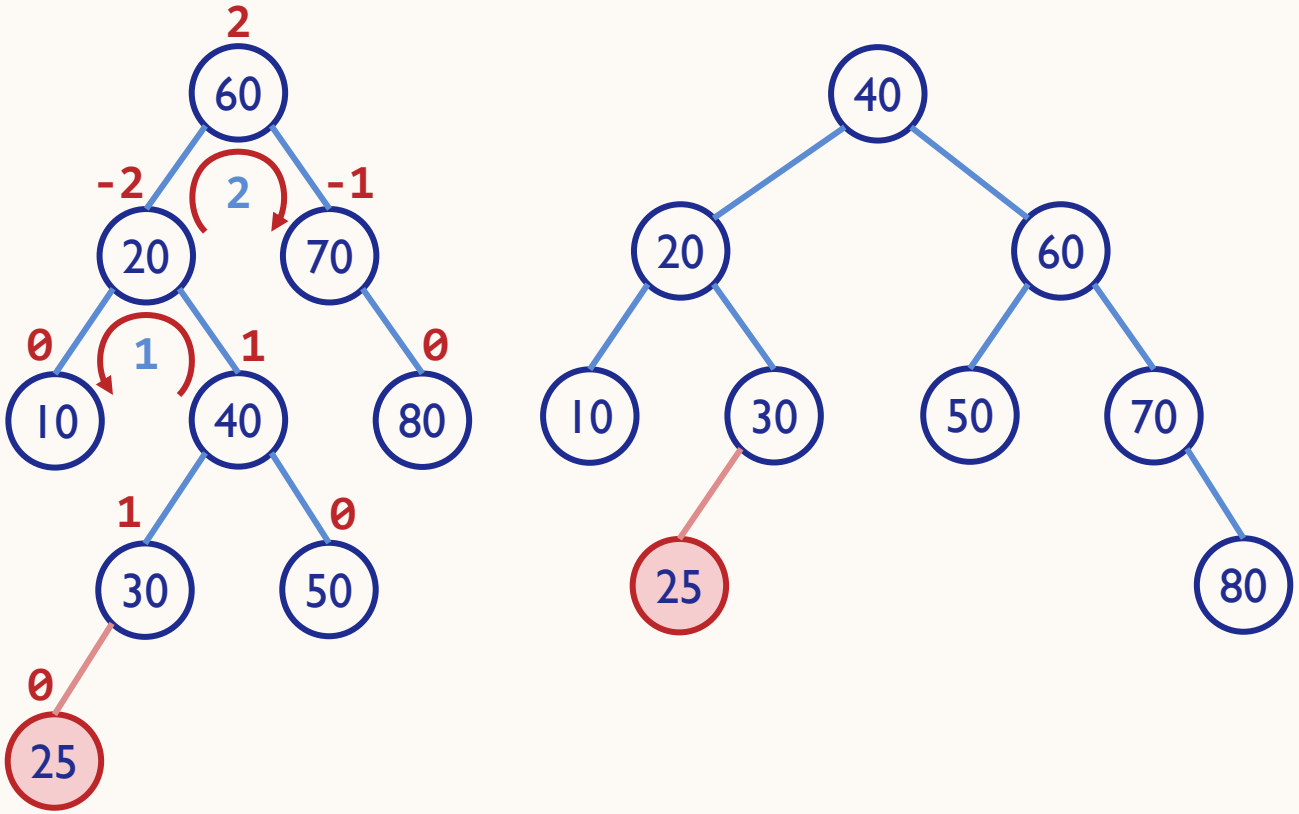


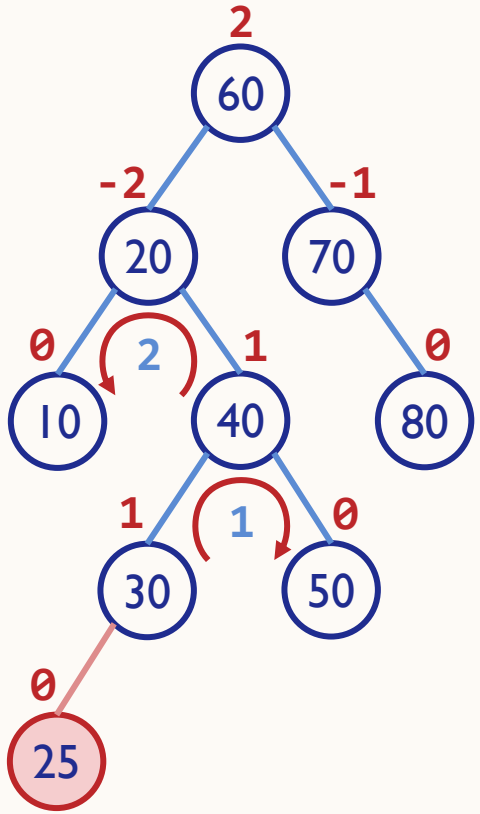


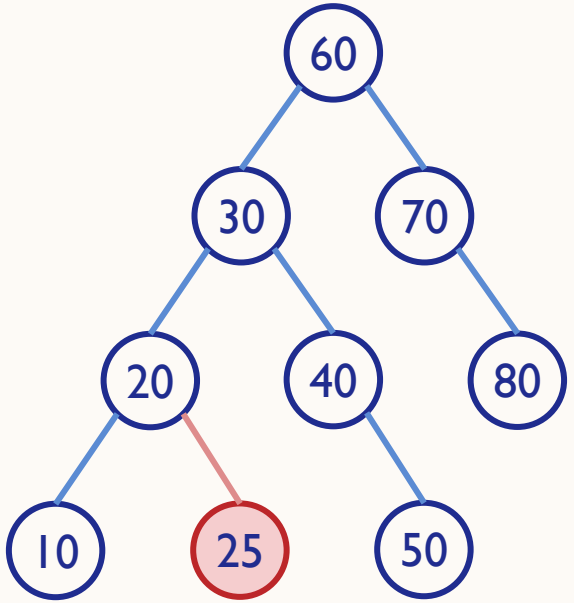


INSERTING A NEW NODE IN AN AVL TREE

LR ROTATION OR RL ROTATION?



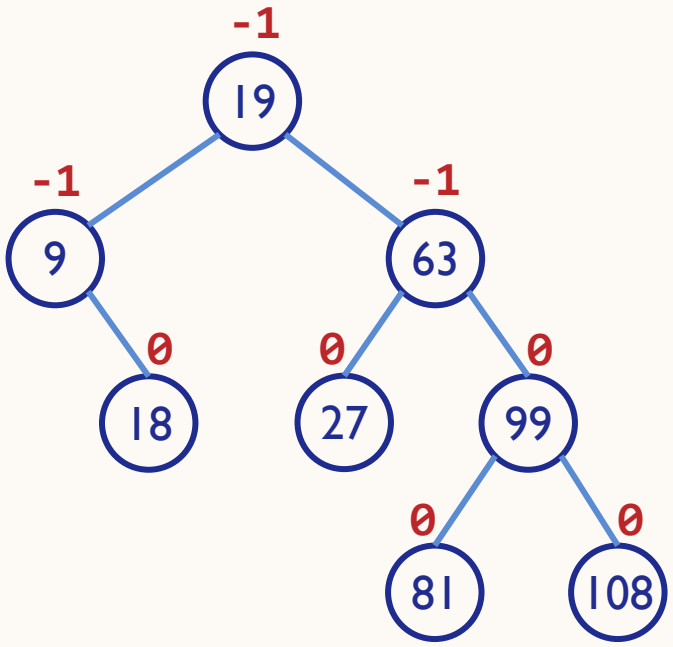




INSERTING A NEW NODE IN AN AVL TREE

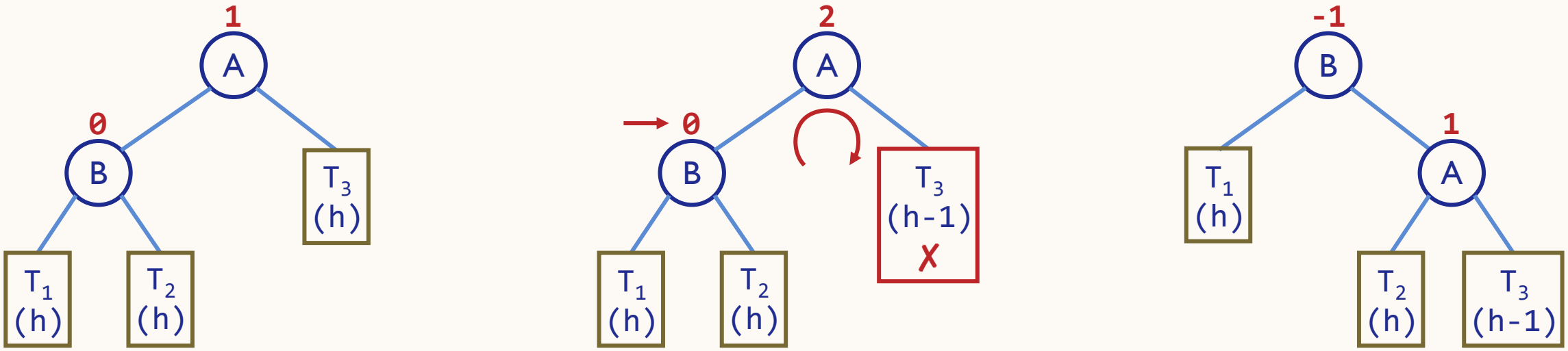
Construct an AVL tree by inserting the following elements in the given order.

`63, 9, 19, 27, 18, 108, 99, 81`



DELETING A NODE FROM AN AVL TREE

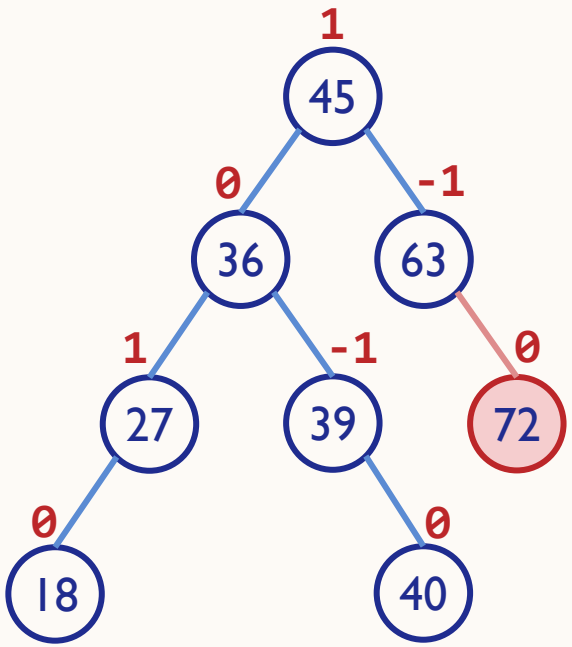
R0 ROTATION

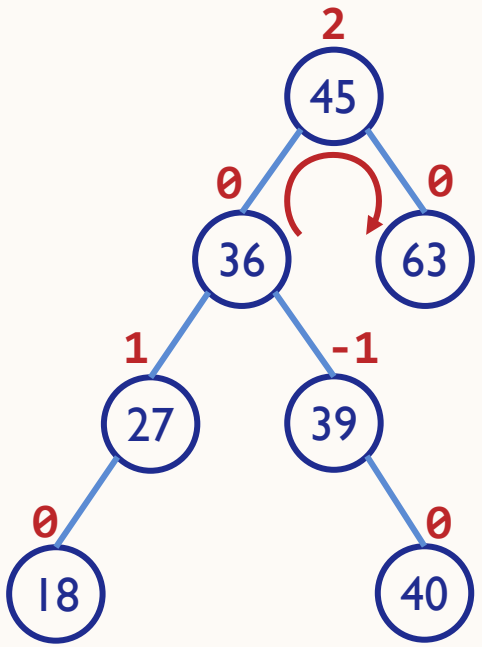


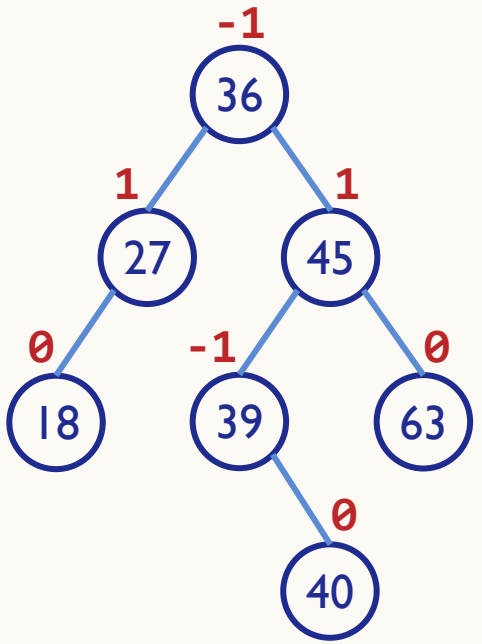
DELETING A NODE FROM AN AVL TREE

R0 ROTATION

Consider the AVL tree given below and delete 72 from it.

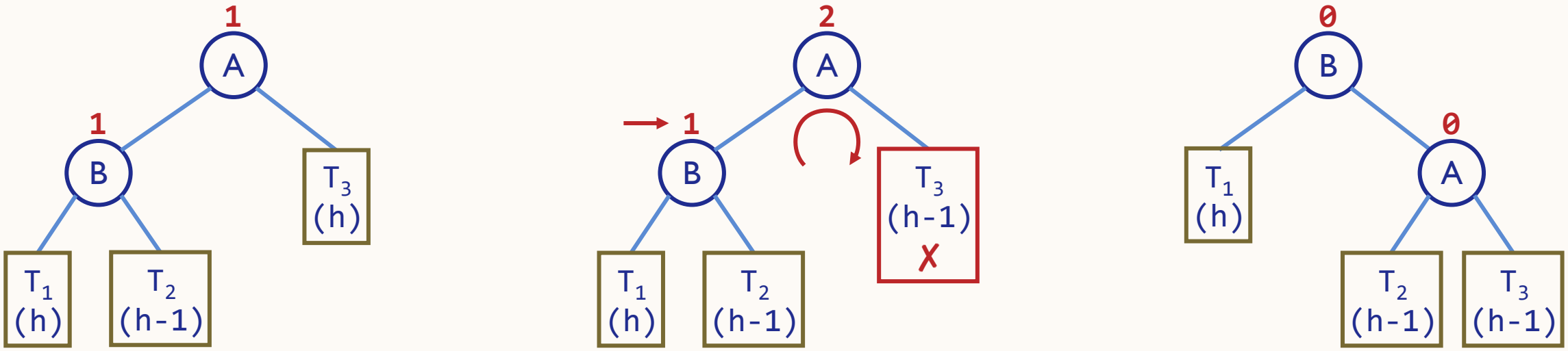






DELETING A NODE FROM AN AVL TREE

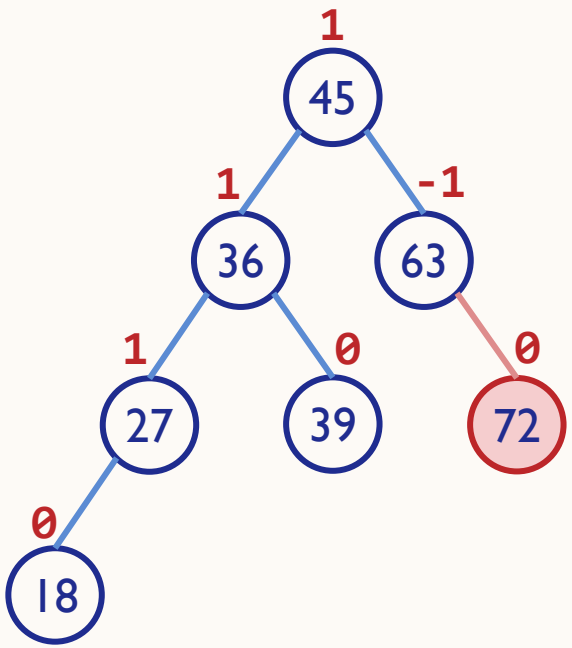
R1 ROTATION

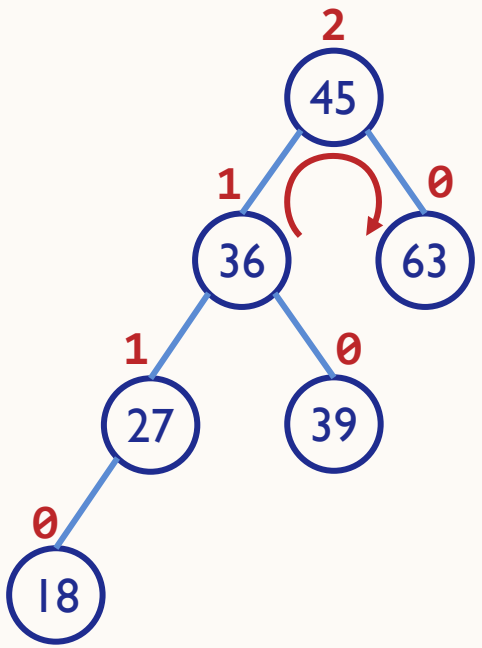


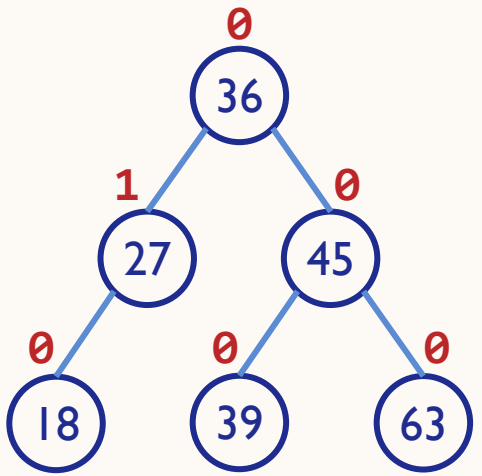
DELETING A NODE FROM AN AVL TREE

R1 ROTATION

Consider the AVL tree given below and delete 72 from it.

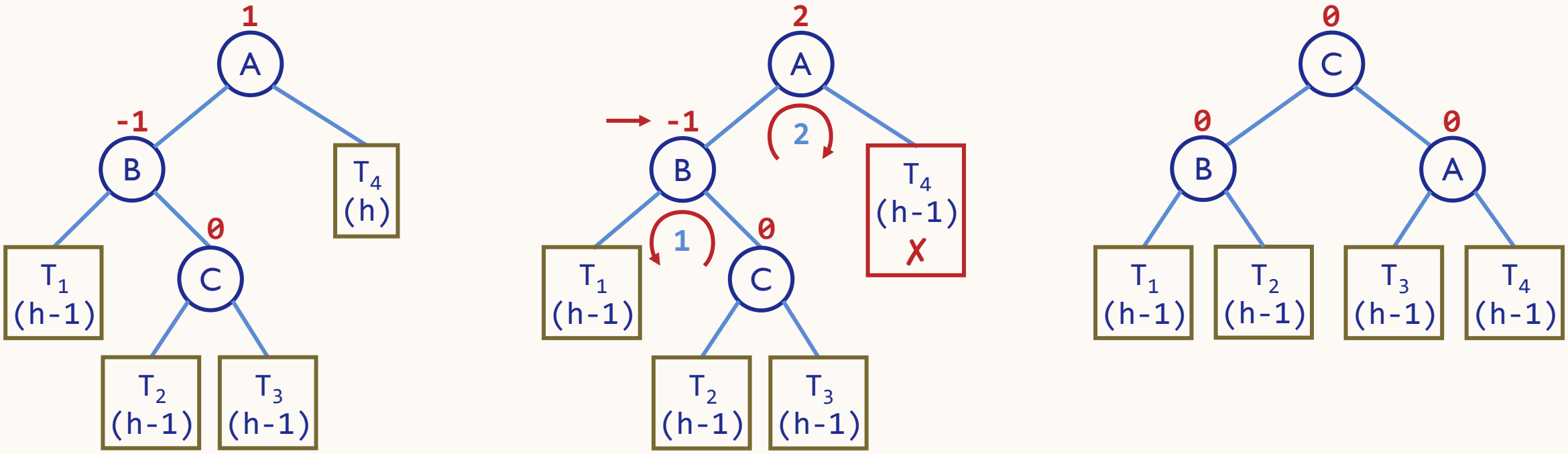






DELETING A NODE FROM AN AVL TREE

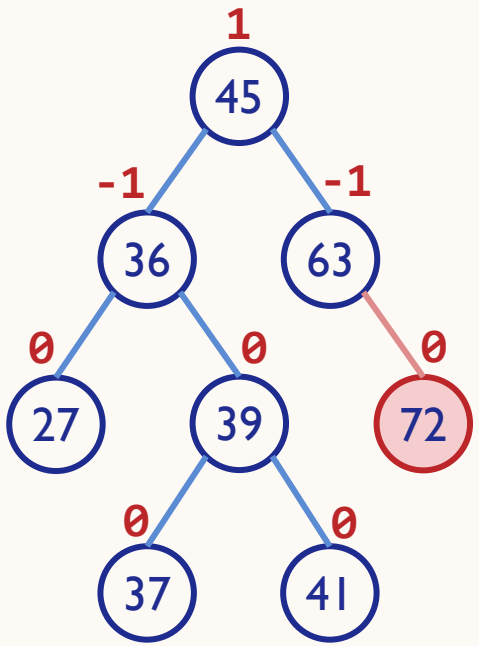
R–1 ROTATION

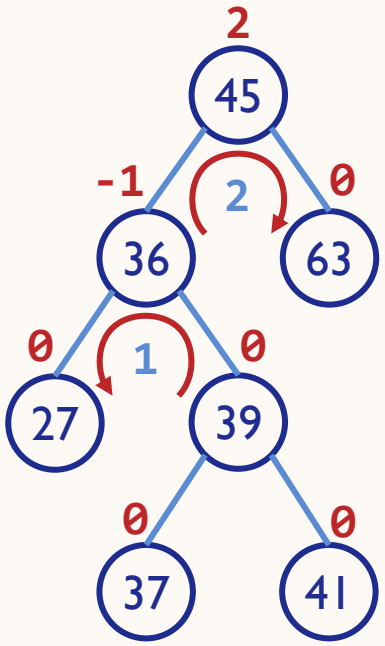


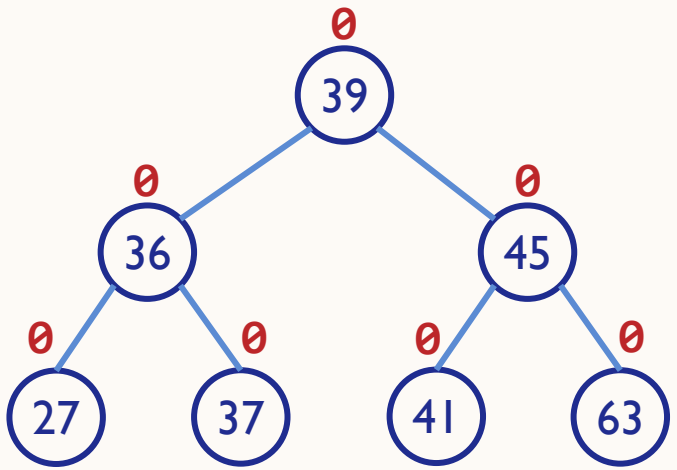
DELETING A NODE FROM AN AVL TREE

R–1 ROTATION

Consider the AVL tree given below and delete 72 from it.

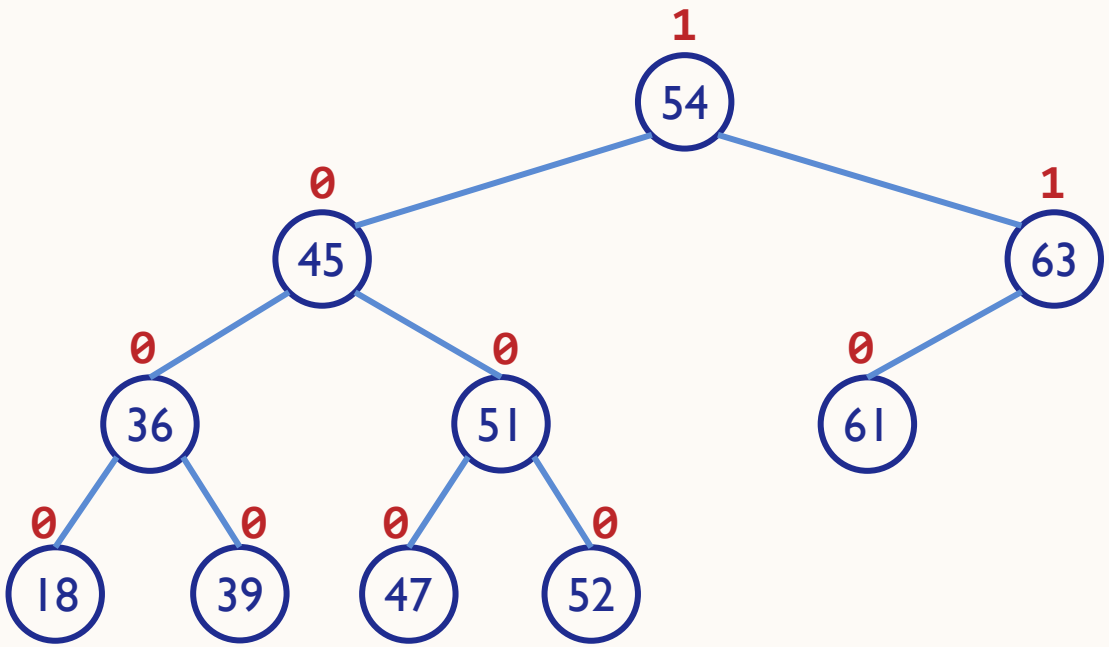


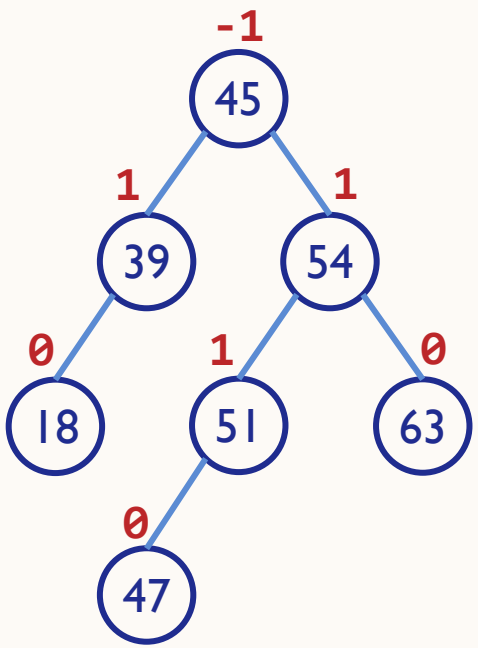




DELETING A NODE FROM AN AVL TREE

Delete nodes 52, 36, and 61 from the AVL tree given below.





PRACTICE

EXERCISES

1. How many nodes will a complete binary tree with 27 nodes have in the last level? What will be the height of the tree?

2. Draw all possible binary search trees of 7, 9, and 11.

EXERCISES

3. Create a binary search tree with the input given below:

   98, 2, 48, 12, 56, 32, 4, 67, 23, 87, 23, 55, 46

   a. Insert 21, 39, 45, 54, and 63 into the tree.

   b. Delete values 23, 56, 2, and 45 from the tree.

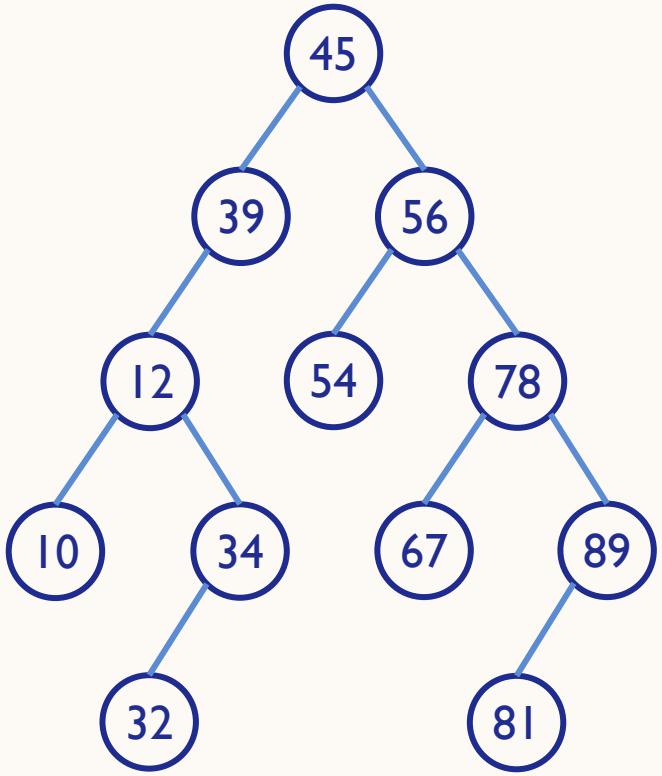
EXERCISES

4. Consider the given binary search tree. Now do the following operations.

    a. Find the result of in-order, pre-order, and post-order traversals.

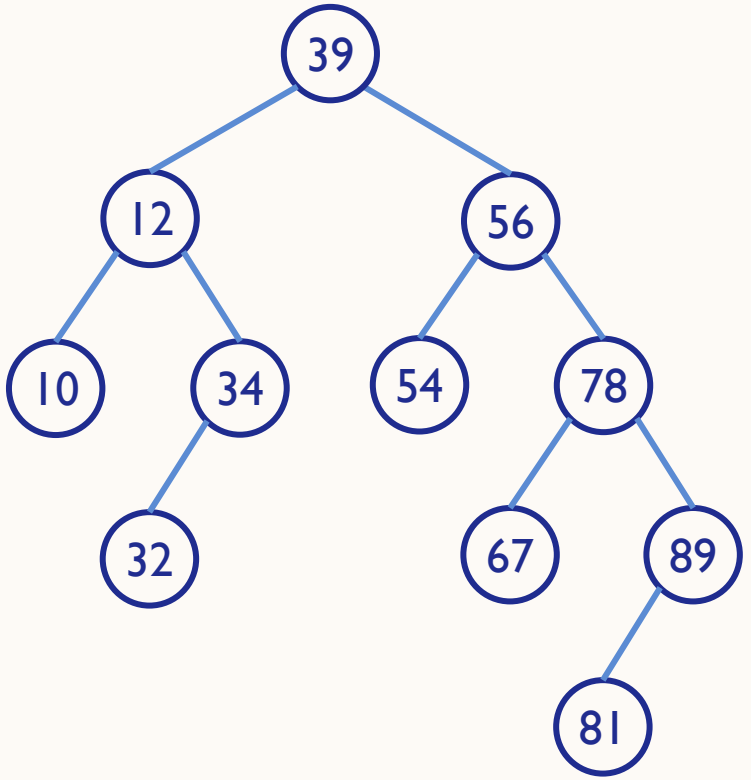
    b. Insert 11, 22, 33, 44, 55, 66, and 77 into the tree.

    c. Delete node 56 from the tree.



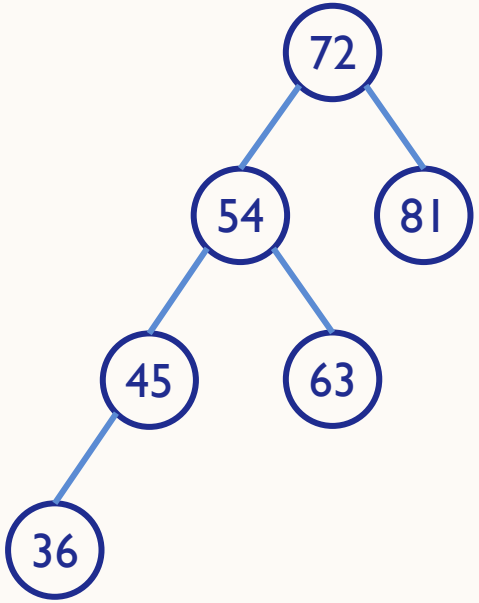
EXERCISES

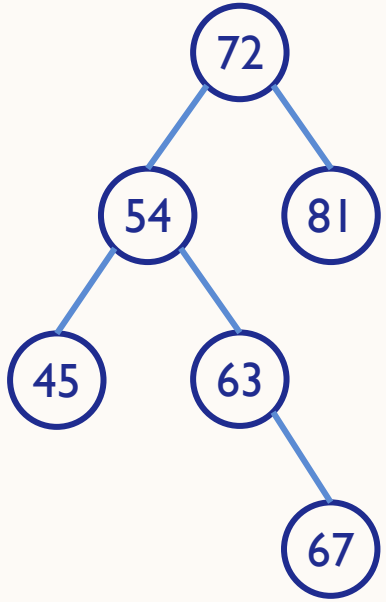
5. Given the binary search tree, show the deletion of the root node.



EXERCISES

6. Balance the AVL trees given below.





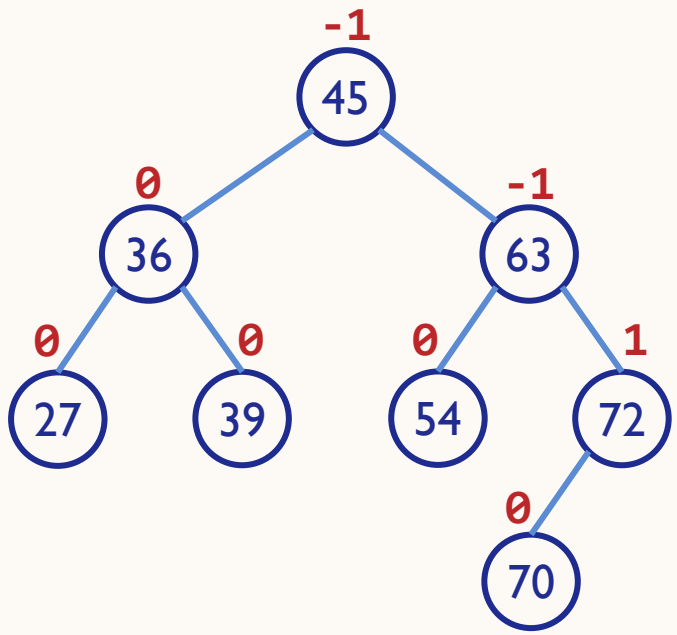
EXERCISES

7. Create an AVL tree using the following sequence of data:

   `16, 27, 9, 11, 36, 54, 81, 63, 72`

EXERCISES

8. Consider the AVL tree given below.



a. Insert 18, 81, 29, 15, 19, 25, 26, and 1 into the tree.

b. Delete nodes 39, 63, 15, and 1 from the AVL tree formed after solving the above question.

REFERENCES

- Deitel, P. and Harvey Deitel (2022), C How to Program (9th Edition), Pearson Education.
- Thareja, R. (2014), Data Structures Using C (2nd Edition), India: Oxford University Press.

NEXT

Heaps:

Binary Heaps

Applications of Heaps

VISION

To become an **outstanding** undergraduate Computer Science program that produces **international-minded** graduates who are **competent** in software engineering and have **entrepreneurial spirit** and **noble character**.

MISSION

1. To conduct studies with the best technology and curriculum, supported by professional lecturer
2. To conduct research in Informatics to promote science and technology
3. To deliver science-and-technology-based society services to implement science and technology

Without hard work,
nothing grows but weeds.

UMN
UNIVERSITAS MULTIMEDIA NUSANTARA

INFORMATIKA UMN

Have patience.
All things are difficult before they become easy.