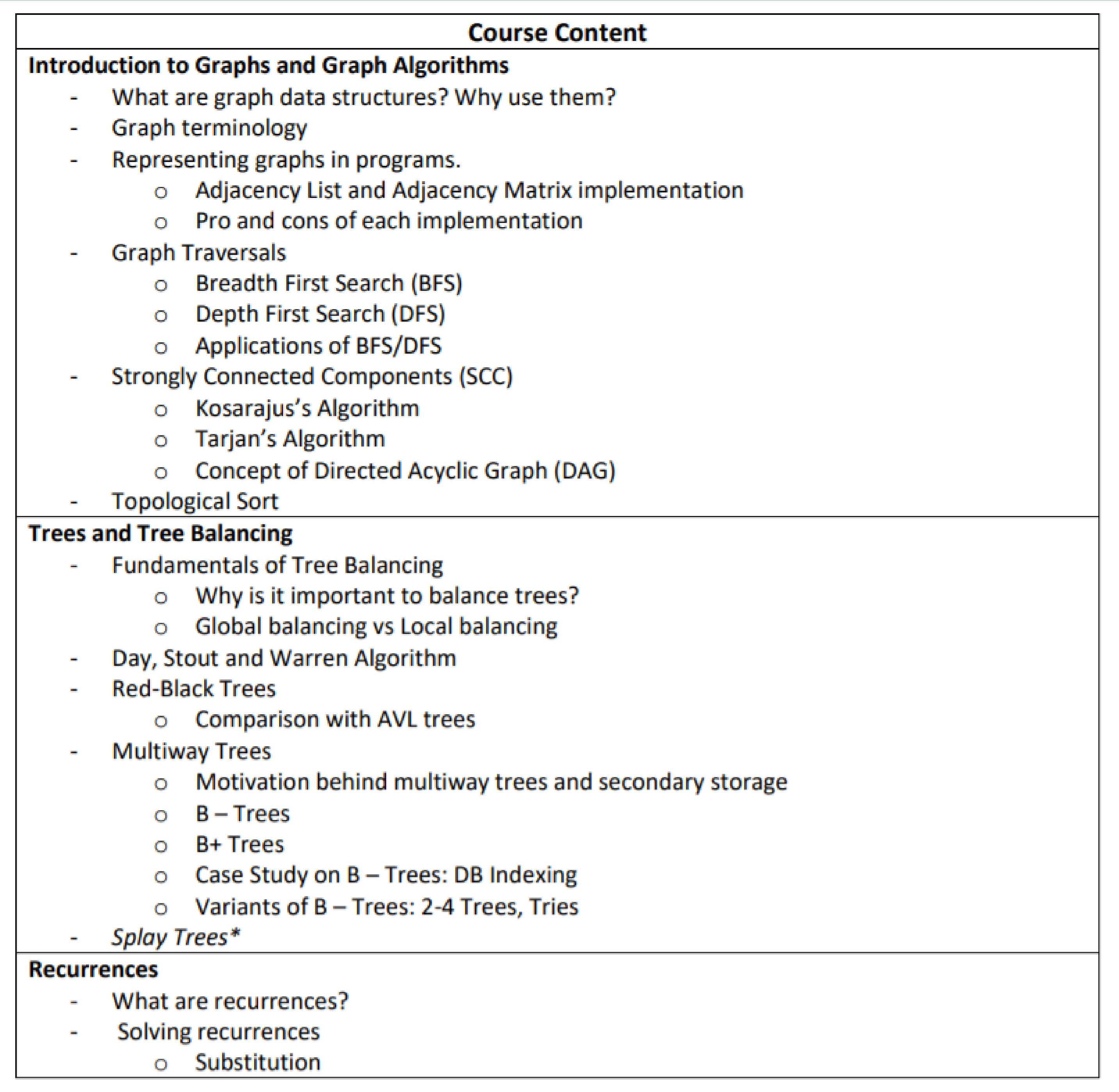
TREES

By Dhanushka sandakelum

You can find out more CS videos over here :- https://www.youtube.com/playlist?list=PLtfHeV0hthugH8IoMU2mtCp3zaBE1j4M5



Course Content

Introduction to Graphs and Graph Algorithms

- What are graph data structures? Why use them?
- Graph terminology
- Representing graphs in programs.
    - Adjacency List and Adjacency Matrix implementation
    - Pro and cons of each implementation
- Graph Traversals
    - Breadth First Search (BFS)
    - Depth First Search (DFS)
    - Applications of BFS/DFS
- Strongly Connected Components (SCC)
    - Kosarajus's Algorithm
    - Tarjan's Algorithm
    - Concept of Directed Acyclic Graph (DAG)
- Topological Sort

Trees and Tree Balancing

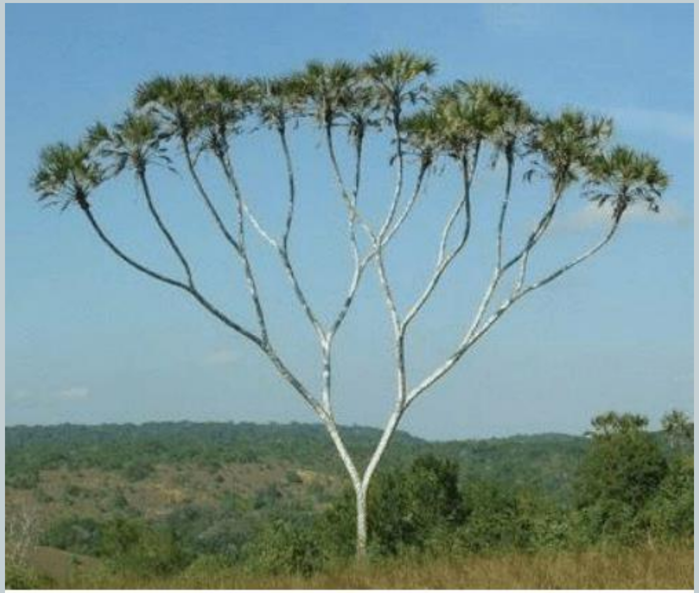
- Fundamentals of Tree Balancing
    - Why is it important to balance trees?
    - Global balancing vs Local balancing
- Day, Stout and Warren Algorithm
- Red-Black Trees
    - Comparison with AVL trees
- Multiway Trees
    - Motivation behind multiway trees and secondary storage
    - B – Trees
    - B+ Trees
    - Case Study on B – Trees: DB Indexing
    - Variants of B – Trees: 2-4 Trees, Tries
- *Splay Trees**

Recurrences

- What are recurrences?
- Solving recurrences
    - Substitution

WHAT IS A TREE

- A tree is recursively defined as a set of one or more nodes where one node is designated as the root of the tree and all the remaining nodes can be partitioned into non-empty sets each of which is a sub-tree of the root.



Binary tree.

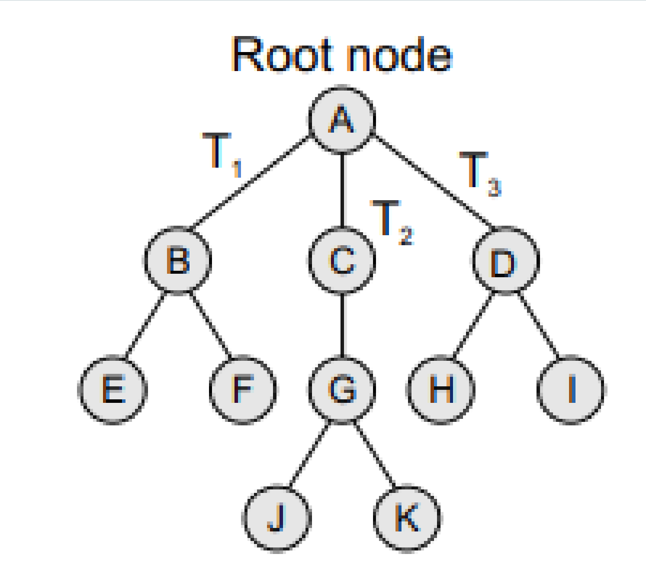


Figure 9.1 Tree

Binary tree.

TYPES OF TREES

- Trees are of following 6 types:
  - ❑ 1. General trees
  - ❑ 2. Forests
  - ❑ 3. Binary trees
  - ❑ 4. Binary search trees
  - ❑ 5. Expression trees
  - ❑ 6. Tournament trees

BINARY TREES

- A binary tree is a data structure that is defined as a collection of elements called nodes. In a binary tree, the topmost element is called the root node, and each node has 0, 1, or at the most 2 children. A node that has zero children is called a leaf node or a terminal node. Every node contains a data element, a left pointer which points to the left child, and a right pointer which points to the right child. The root element is pointed by a 'root' pointer. If root = NULL, then it means the tree is empty.

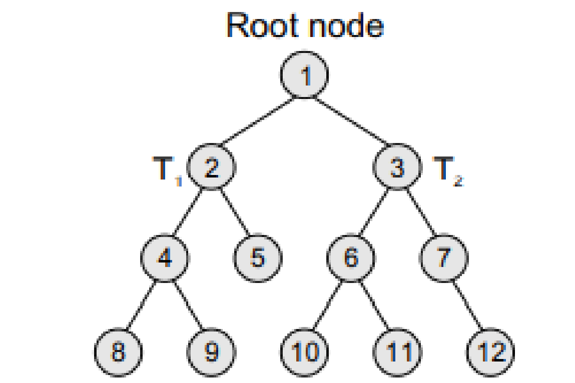


Figure 9.3 Binary tree

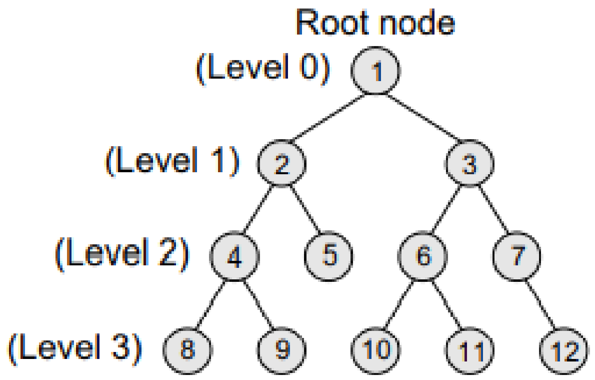


Figure 9.4 Levels in binary tree

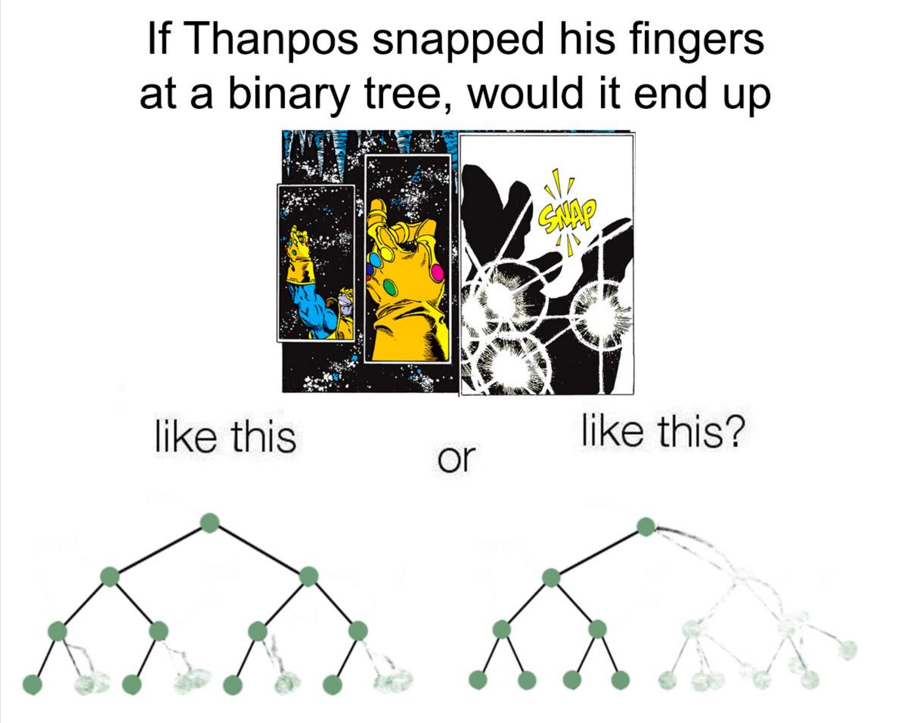
EFFICIENT BINARY TREES

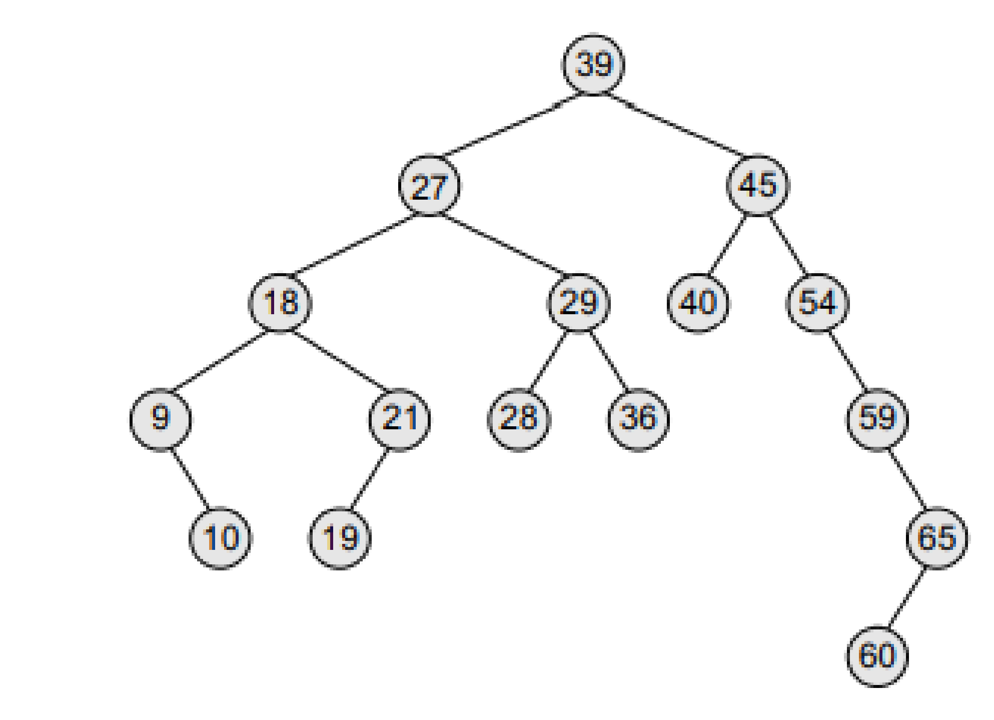
- There are so many variations in binary trees. Following slides we are going to discuss above binary tree types.
- Binary Search Trees
- AVL trees
- Red black trees

BINARY SEARCH TREES

BINARY SEARCH TREES

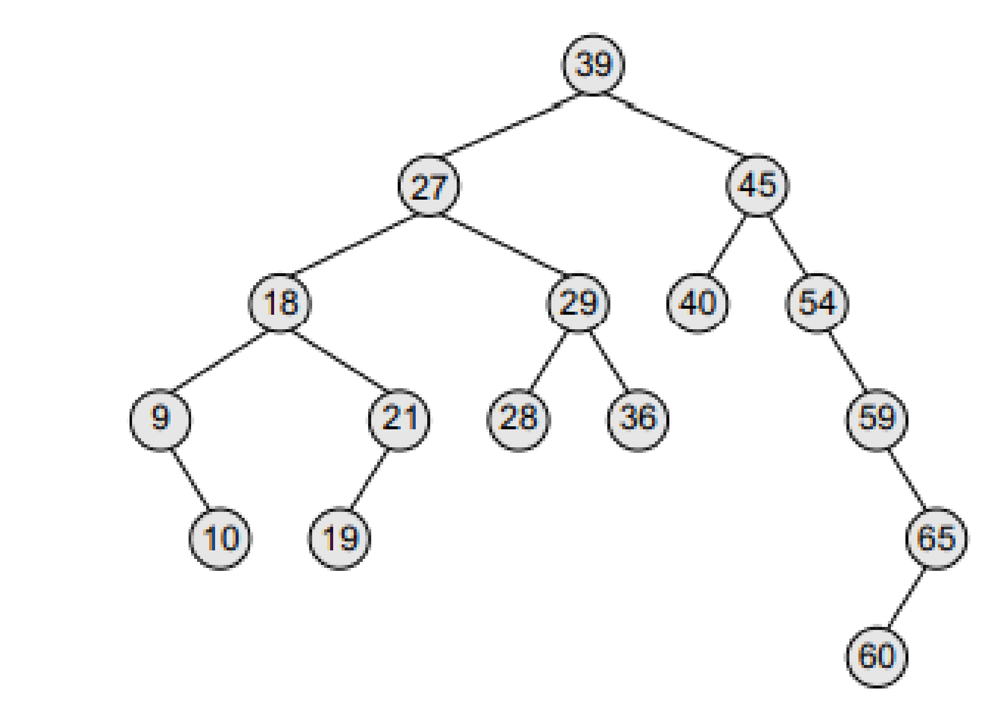
- A binary search tree, also known as an ordered binary tree, is a variant of binary trees in which the nodes are arranged in an order. In a binary search tree, all the nodes in the left sub-tree have a value less than that of the root node. Correspondingly, all the nodes in the right sub-tree have a value either equal to or greater than the root node. The same rule is applicable to every sub-tree in the tree. (Note that a binary search tree may or may not contain duplicate values, depending on its implementation.)


If Thanpos snapped his fingers
at a binary tree, would it end up
SNAP
like this
or
like this?



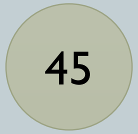
BINARY SEARCH TREES

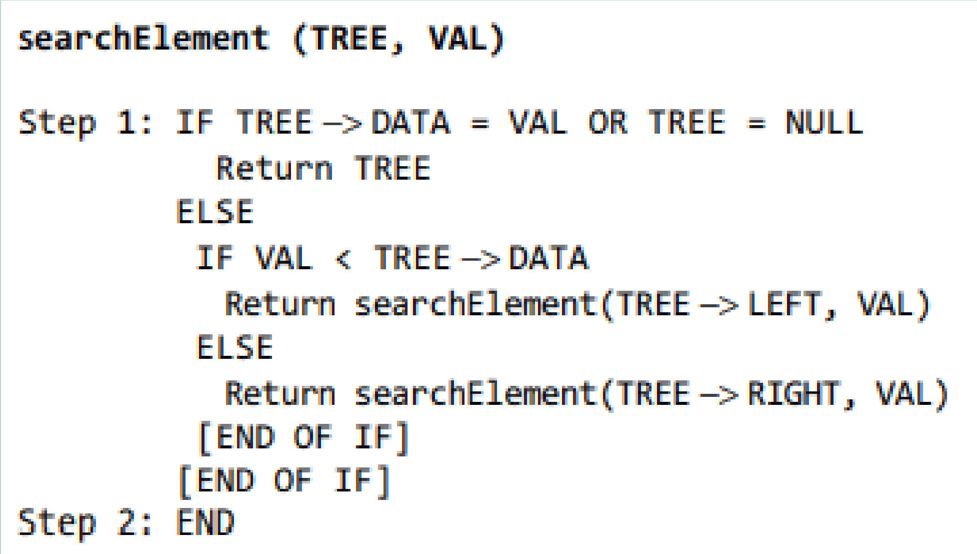
- A binary search tree, also known as an ordered binary tree, is a variant of binary trees in which the nodes are arranged in an order. In a binary search tree, all the nodes in the left sub-tree have a value less than that of the root node. Correspondingly, all the nodes in the right sub-tree have a value either equal to or greater than the root node. The same rule is applicable to every sub-tree in the tree. (Note that a binary search tree may or may not contain duplicate values, depending on its implementation.)



BINARY SEARCH TREES INSERTION

- Create a binary search tree using the following data elements: 45, 39, 56, 12, 34, 78, 32

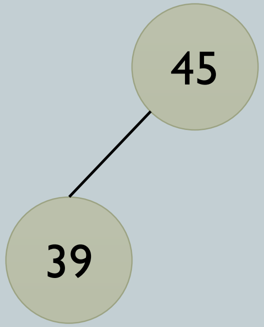


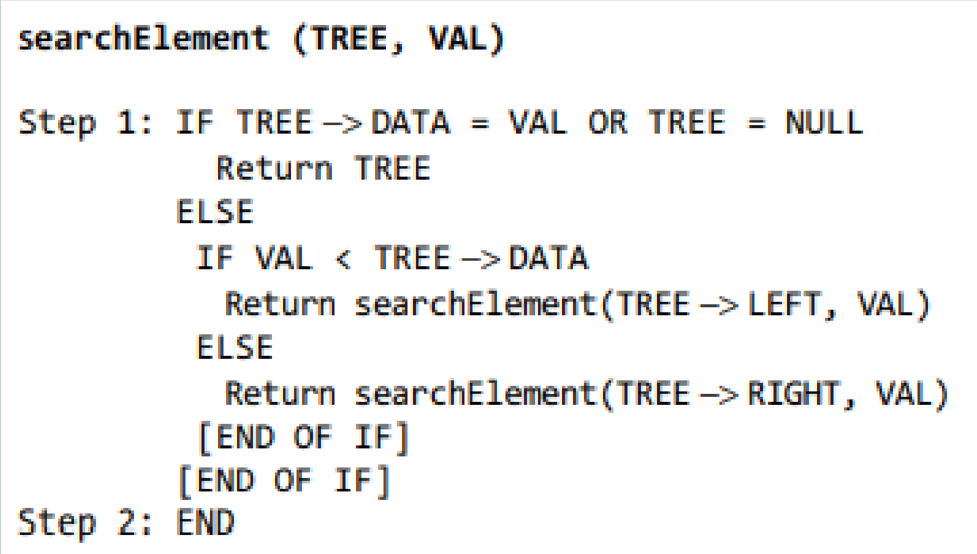


```
searchElement (TREE, VAL)

Step 1: IF TREE -> DATA = VAL OR TREE = NULL
          Return TREE
        ELSE
         IF VAL < TREE -> DATA
          Return searchElement(TREE -> LEFT, VAL)
         ELSE
          Return searchElement(TREE -> RIGHT, VAL)
         [END OF IF]
        [END OF IF]
Step 2: END
```

BINARY SEARCH TREES INSERTION

- Create a binary search tree using the following data elements: 45, 39, 56, 12, 34, 78, 32

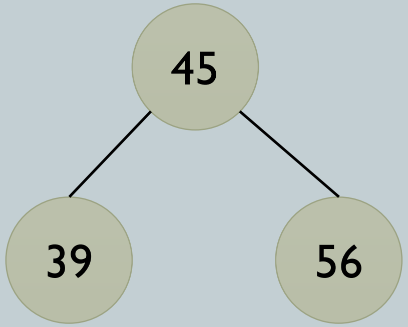


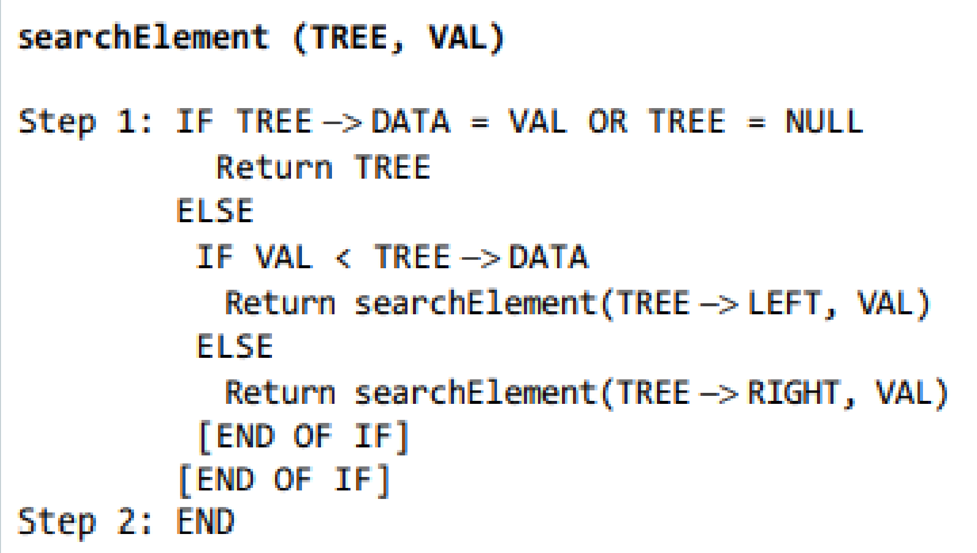


```
searchElement (TREE, VAL)

Step 1: IF TREE -> DATA = VAL OR TREE = NULL
          Return TREE
        ELSE
         IF VAL < TREE -> DATA
          Return searchElement(TREE -> LEFT, VAL)
         ELSE
          Return searchElement(TREE -> RIGHT, VAL)
         [END OF IF]
        [END OF IF]
Step 2: END
```

BINARY SEARCH TREES INSERTION

- Create a binary search tree using the following data elements: 45, 39, 56, 12, 34, 78, 32

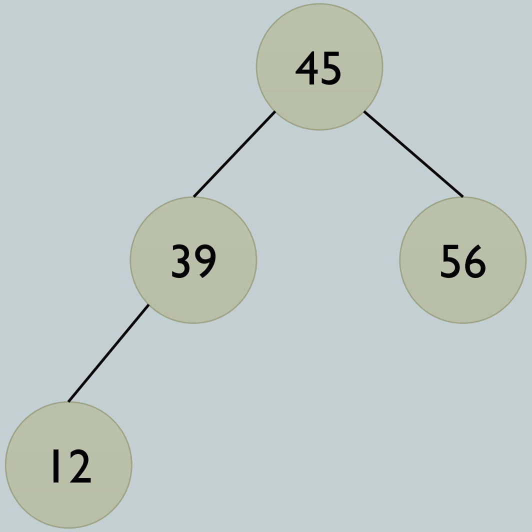


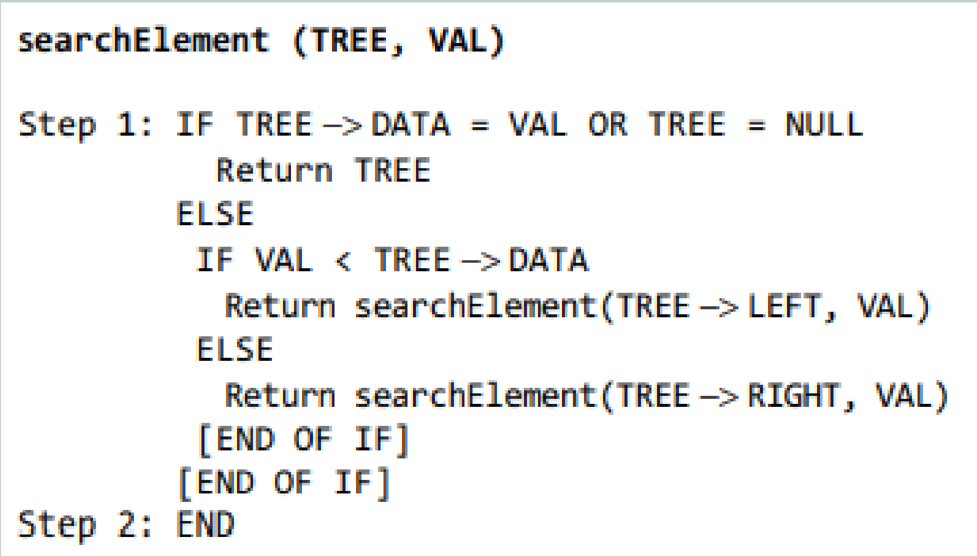


```
searchElement (TREE, VAL)

Step 1: IF TREE -> DATA = VAL OR TREE = NULL
          Return TREE
        ELSE
         IF VAL < TREE -> DATA
          Return searchElement(TREE -> LEFT, VAL)
         ELSE
          Return searchElement(TREE -> RIGHT, VAL)
         [END OF IF]
        [END OF IF]
Step 2: END
```

BINARY SEARCH TREES INSERTION

- Create a binary search tree using the following data elements: 45, 39, 56, 12, 34, 78, 32

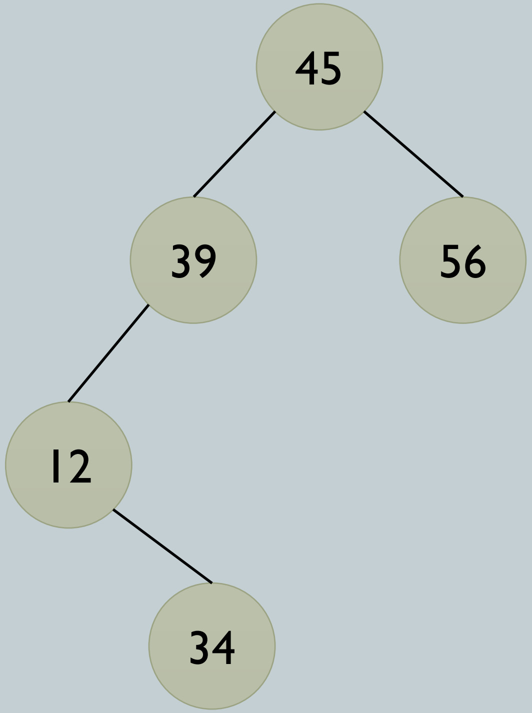


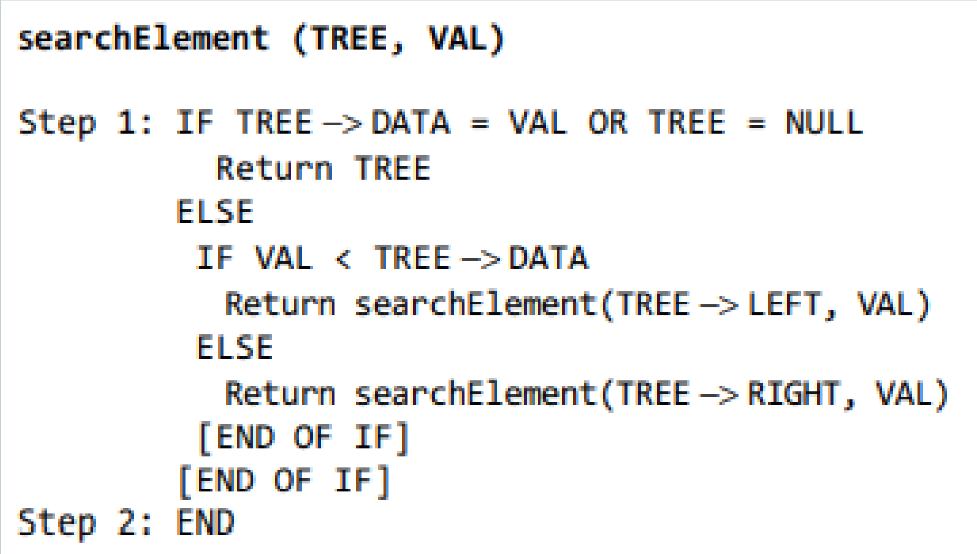


```
searchElement (TREE, VAL)

Step 1: IF TREE -> DATA = VAL OR TREE = NULL
            Return TREE
        ELSE
         IF VAL < TREE -> DATA
           Return searchElement(TREE -> LEFT, VAL)
         ELSE
           Return searchElement(TREE -> RIGHT, VAL)
         [END OF IF]
        [END OF IF]
Step 2: END
```

BINARY SEARCH TREES INSERTION

- Create a binary search tree using the following data elements: 45, 39, 56, 12, 34, 78, 32

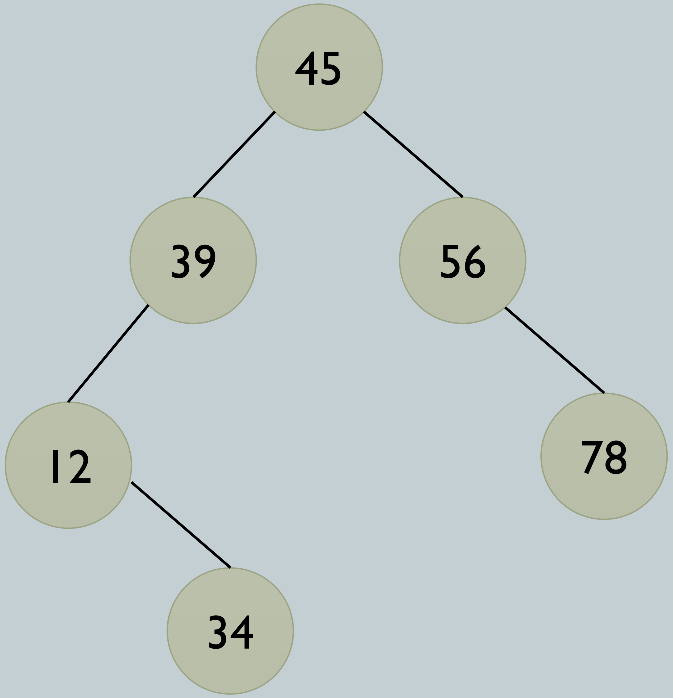


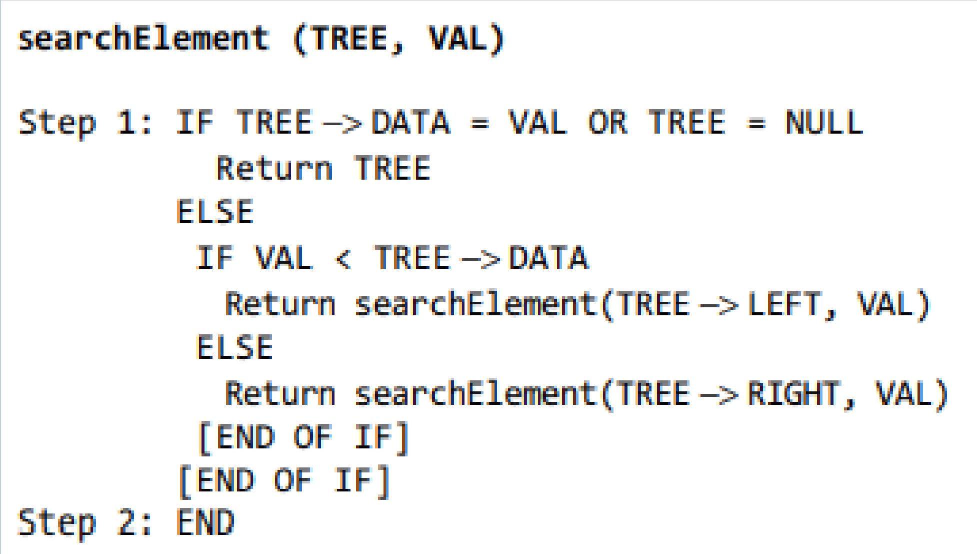


```
searchElement (TREE, VAL)

Step 1: IF TREE -> DATA = VAL OR TREE = NULL
          Return TREE
        ELSE
         IF VAL < TREE -> DATA
          Return searchElement(TREE -> LEFT, VAL)
         ELSE
          Return searchElement(TREE -> RIGHT, VAL)
         [END OF IF]
        [END OF IF]
Step 2: END
```

BINARY SEARCH TREES INSERTION

- Create a binary search tree using the following data elements: 45, 39, 56, 12, 34, 78, 32

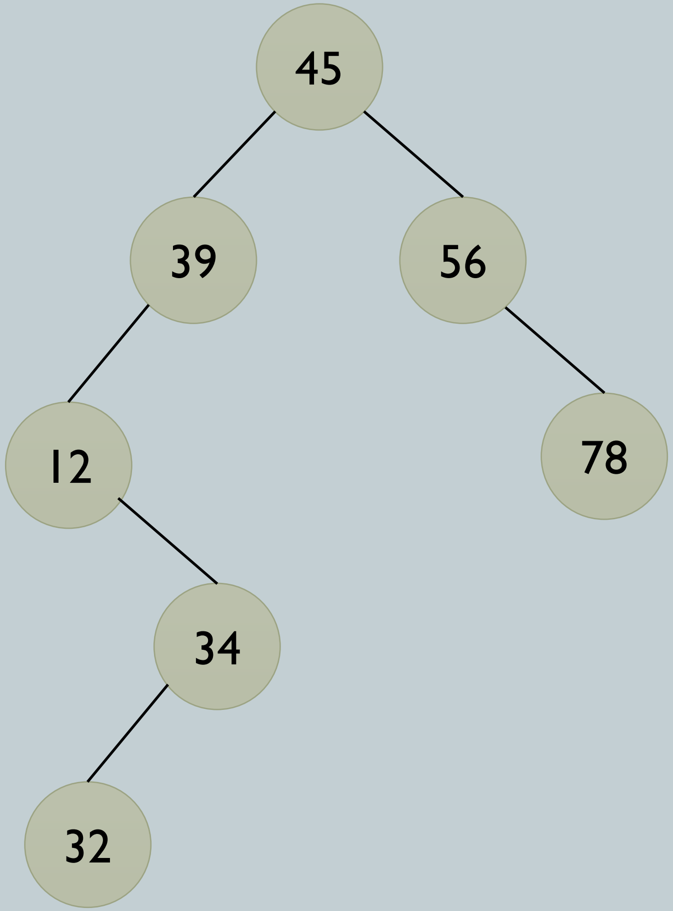


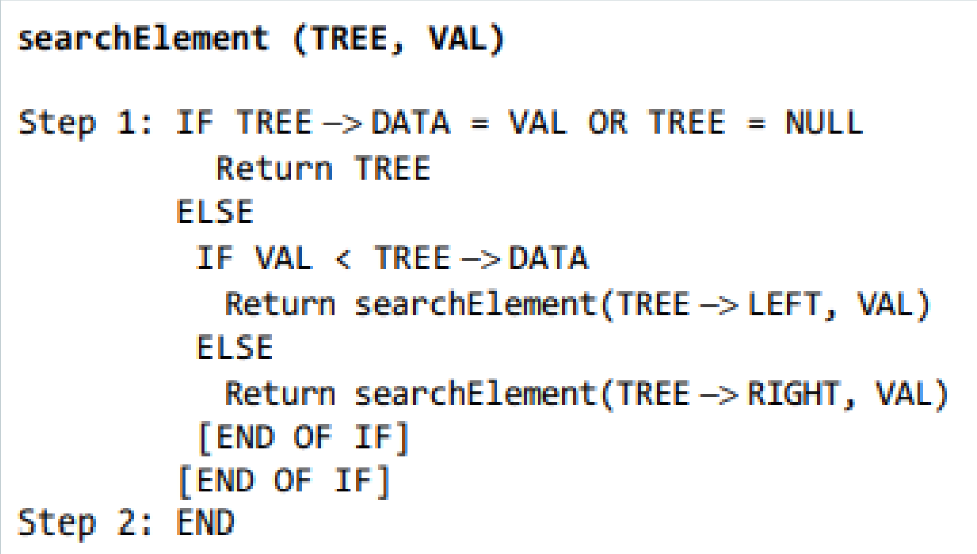


```
searchElement (TREE, VAL)

Step 1: IF TREE -> DATA = VAL OR TREE = NULL
          Return TREE
        ELSE
         IF VAL < TREE -> DATA
          Return searchElement(TREE -> LEFT, VAL)
         ELSE
          Return searchElement(TREE -> RIGHT, VAL)
         [END OF IF]
        [END OF IF]
Step 2: END
```

BINARY SEARCH TREES INSERTION

- Create a binary search tree using the following data elements: 45, 39, 56, 12, 34, 78, 32



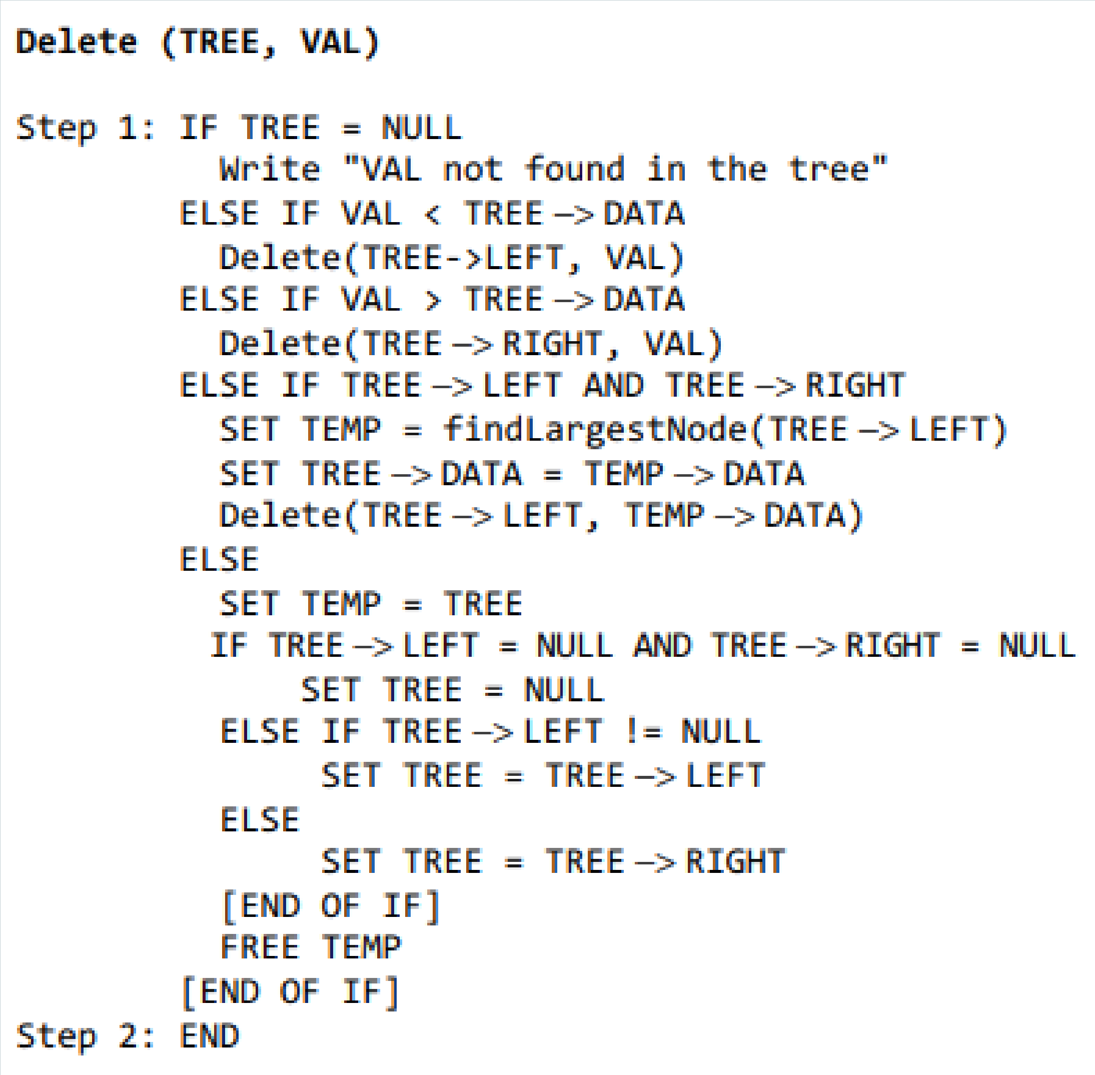


```
searchElement (TREE, VAL)

Step 1: IF TREE -> DATA = VAL OR TREE = NULL
          Return TREE
        ELSE
         IF VAL < TREE -> DATA
          Return searchElement(TREE -> LEFT, VAL)
         ELSE
          Return searchElement(TREE -> RIGHT, VAL)
         [END OF IF]
        [END OF IF]
Step 2: END
```

BINARY SEARCH TREES DELETION

- When it comes to BST deletion it is not simple like insertion. According to the node that we are going to delete, the deletion procedure will be different. First lets see what are possible cases of BST deletion.
  - CASE 1 : Deleting a node that has no children
  - CASE 2 : Deleting a node with one children
  - CASE 3 : Deleting a node with two children

BINARY SEARCH TREES DELETION

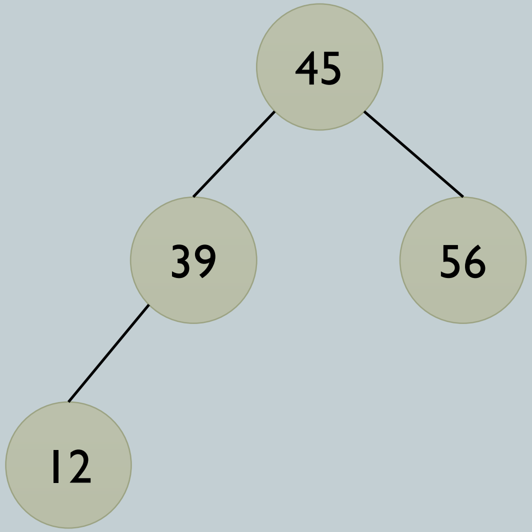


```
Delete (TREE, VAL)

Step 1: IF TREE = NULL
          Write "VAL not found in the tree"
        ELSE IF VAL < TREE -> DATA
          Delete(TREE->LEFT, VAL)
        ELSE IF VAL > TREE -> DATA
          Delete(TREE -> RIGHT, VAL)
        ELSE IF TREE -> LEFT AND TREE -> RIGHT
          SET TEMP = findLargestNode(TREE -> LEFT)
          SET TREE -> DATA = TEMP -> DATA
          Delete(TREE -> LEFT, TEMP -> DATA)
        ELSE
          SET TEMP = TREE
         IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
              SET TREE = NULL
          ELSE IF TREE -> LEFT != NULL
               SET TREE = TREE -> LEFT
          ELSE
               SET TREE = TREE -> RIGHT
          [END OF IF]
          FREE TEMP
        [END OF IF]
Step 2: END
```

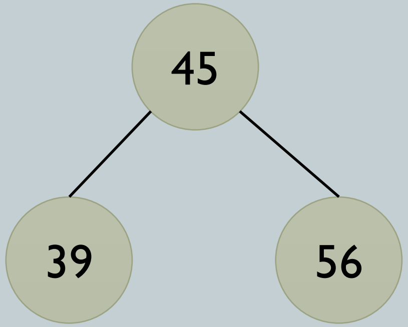
CASE 1 : DELETING A NODE THAT HAS NO CHILDREN

- Simplest case. Just delete the node and make it as a NULL pointer.

- Example :- Delete 12



Before

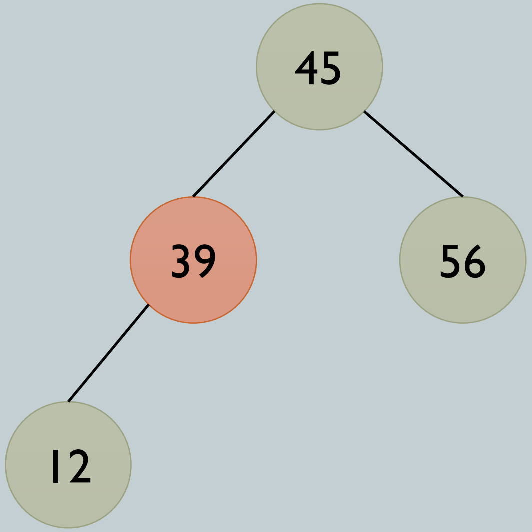


After

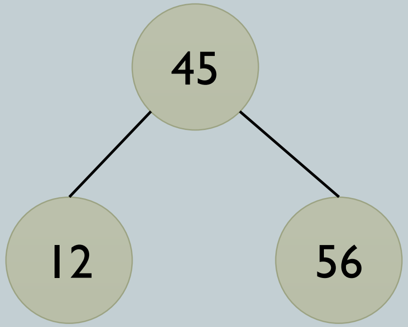
CASE 2 : DELETING A NODE WITH ONE CHILDREN

- In this case the child of node to be deleted is set as the child of the parent of that node to be deleted.
- IF the node is left child of the parent, node’s child become the left child of the parent
- IF the node is right child of the parent, node’s child become the right child of the parent

- Example :- Delete 39 (node is left child of the parent)



Before

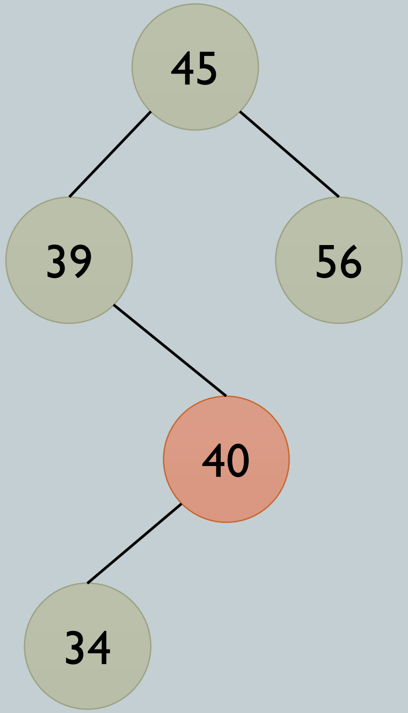


After

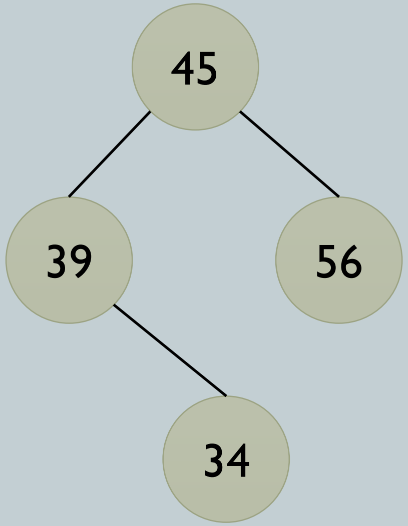
CASE 2 : DELETING A NODE WITH ONE CHILDREN

- In this case the child of node to be deleted is set as the child of the parent of that node to be deleted.
- IF the node is left child of the parent, node's child become the left child of the parent
- IF the node is right child of the parent, node's child become the right child of the parent

- Example :- Delete 12 (node is right child of the parent)



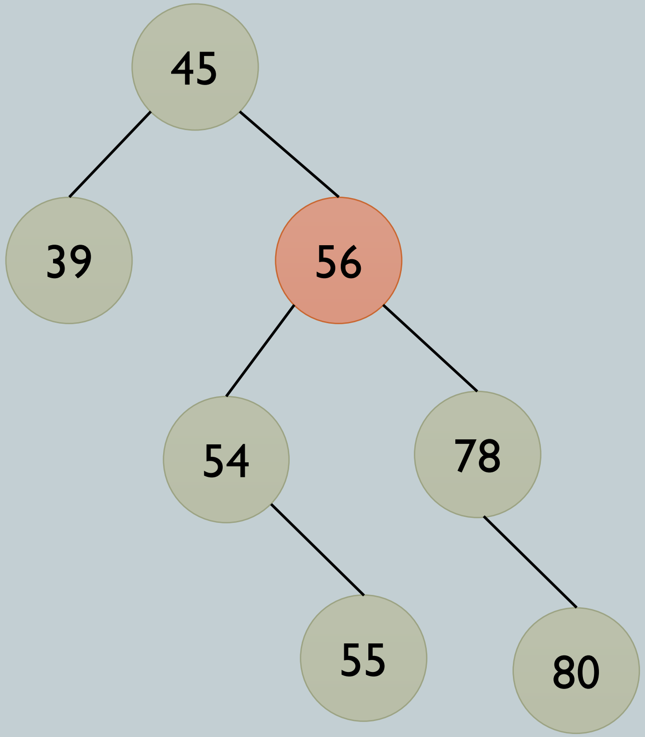
Before



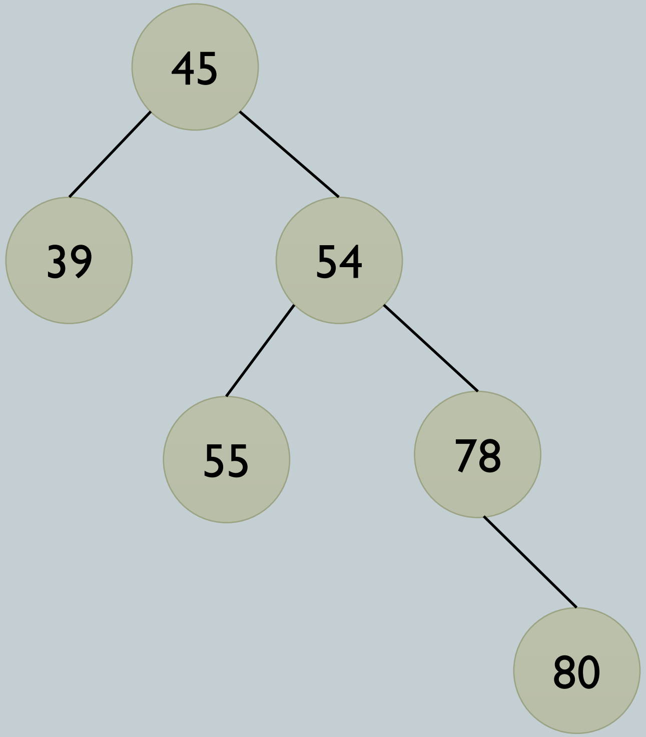
After

CASE 3 : DELETING A NODE WITH TWO CHILDREN

- In this case we replace the node to be deleted with in-order predecessor or in-order successor.
- in-order predecessor – Largest value in the left sub tree
- In-order successor – Smallest value in the right sub tree
- Example :- Delete 56 (following in-order predecessor)



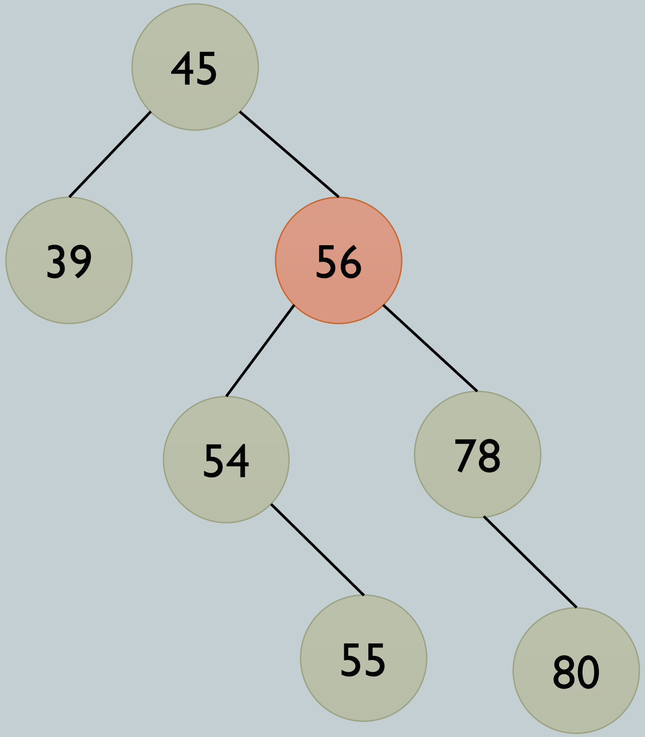
Before



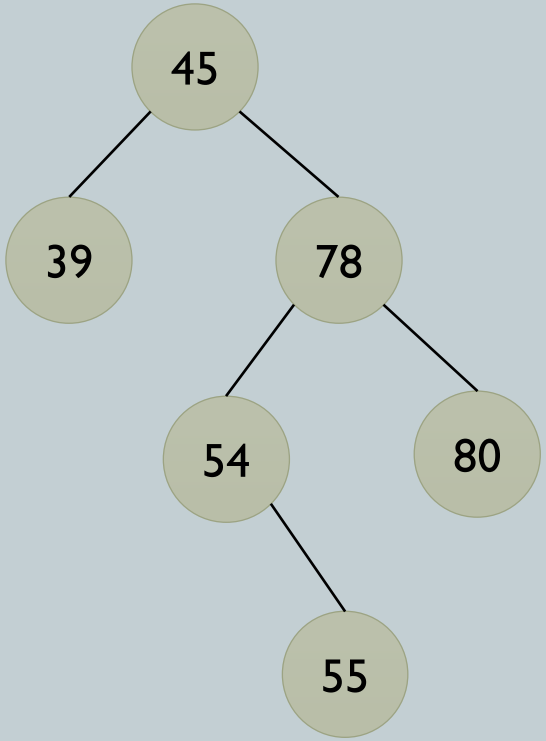
After

CASE 3 : DELETING A NODE WITH TWO CHILDREN

- In this case we replace the node to be deleted with in-order predecessor or in-order successor.
- in-order predecessor – Largest value in the left sub tree
- In-order successor – Smallest value in the right sub tree
- Example :- Delete 56 (following in-order successor)



Before



After

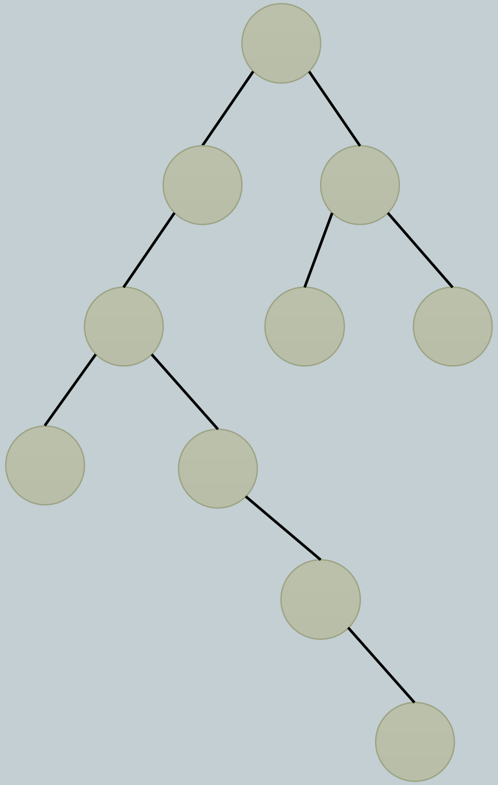
BINARY SEARCH TREES

DSW algorithms

THE DAY STOUT WARREN (DSW) ALGORITHM

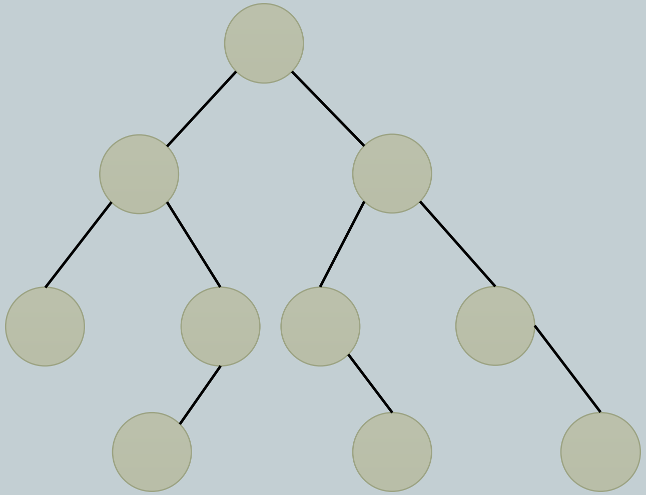
- DSW algorithm balances a binary search tree. This balancing produces a tree that not only has a minimum height, but also forces all the nodes on the bottom most level to be filled from left to right. The running time is O(n) where n is the number of nodes in tree.

Unbalanced tree



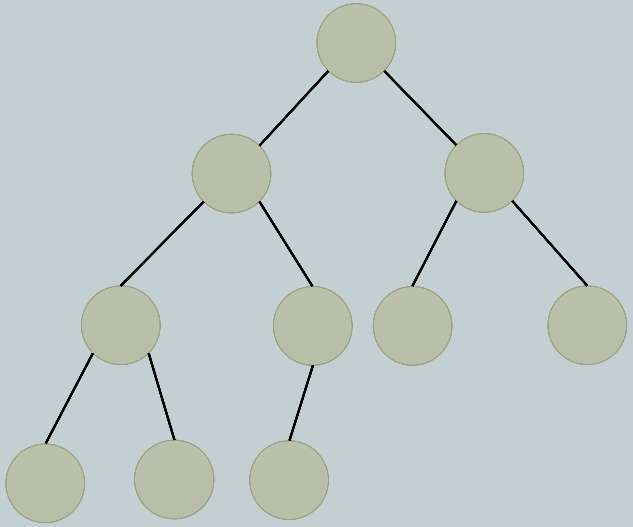
Height 5

Balanced tree



Height 3

DSW Balanced tree



Bottommost nodes
are all to the left.
Height 3

TREE BALANCING TECHNIQUES

- 1. Global tree balancing (DSW algorithm)
  - After creating complete tree, it balances.

2. Local tree balancing (AVL trees, Red black trees)

When we are inserting or deleting a new node, tree balances itself.

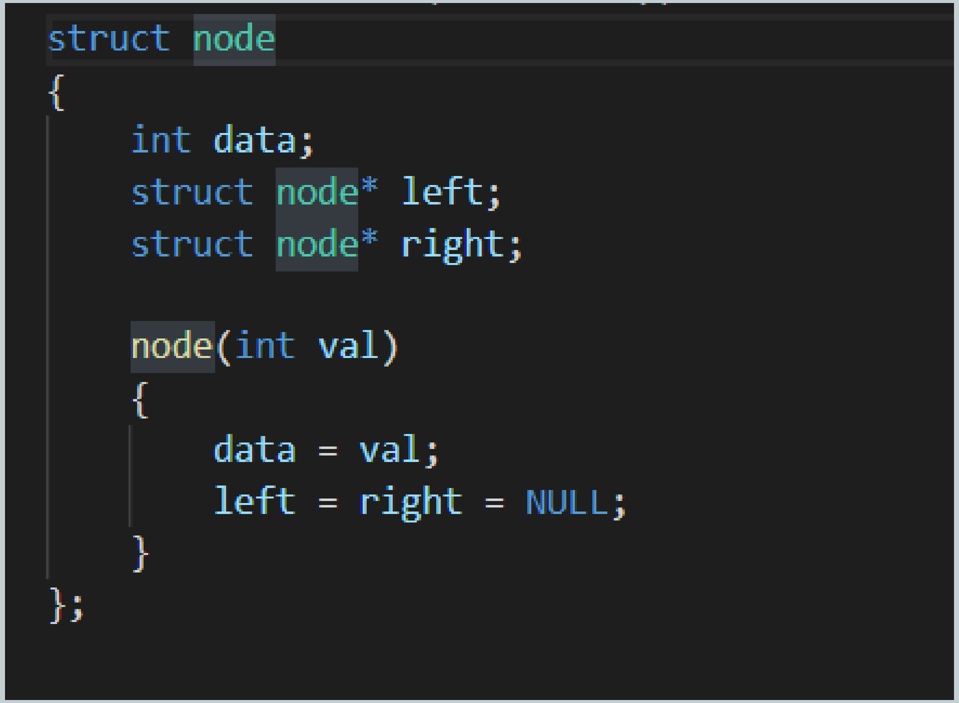
DSW ALGORITHM

1. Create the backbone
2. Balance the backbone

Underline concepts

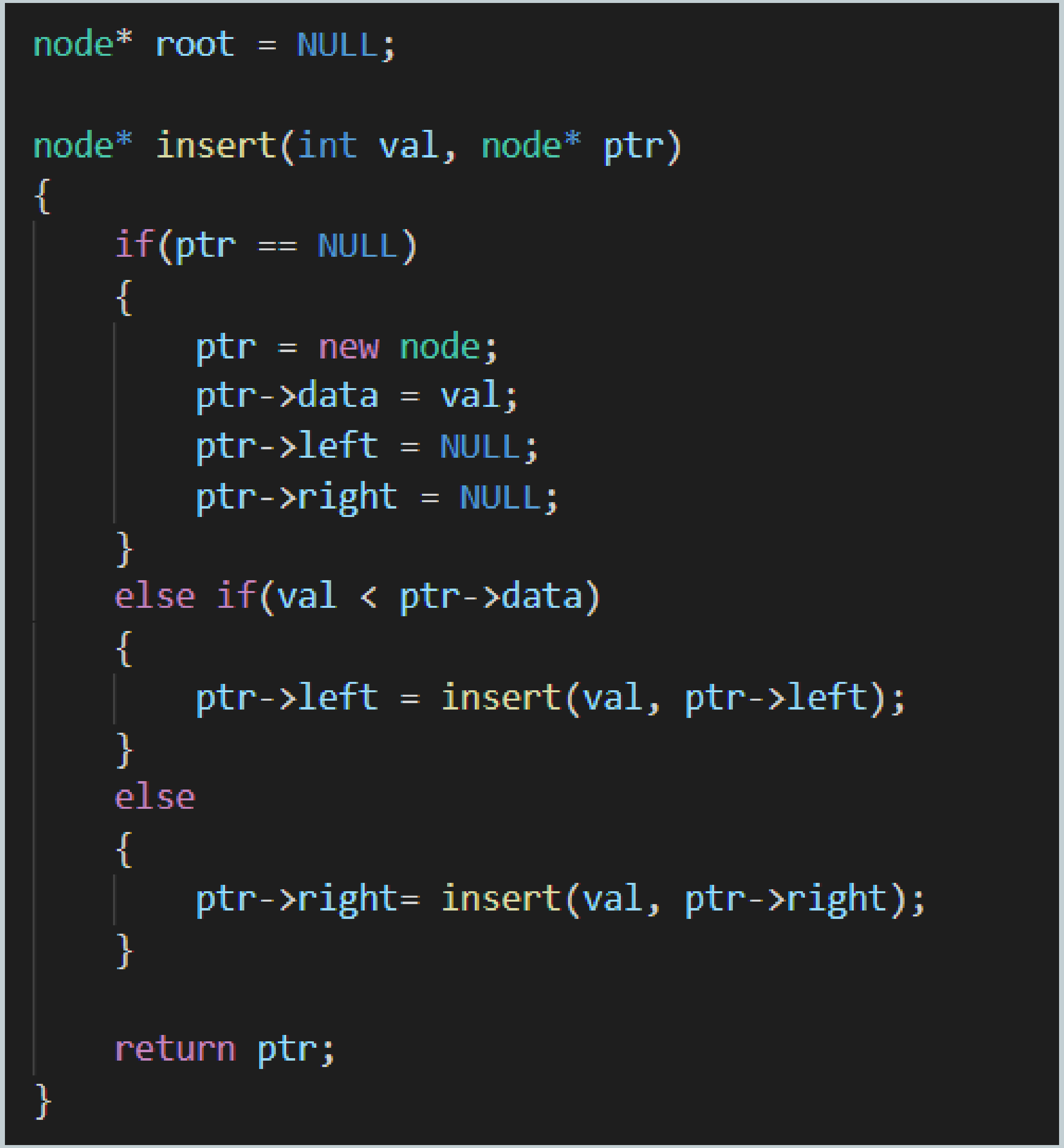
- BST
- Rotations
- Some stuffs

DSW ALGORITHM – NODE STRUCTURE



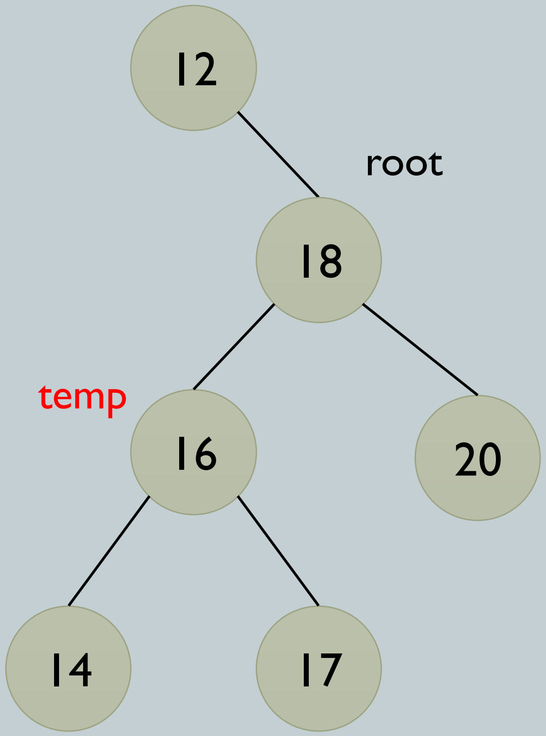
```
struct node
{
    int data;
    struct node* left;
    struct node* right;

    node(int val)
    {
        data = val;
        left = right = NULL;
    }
};
```

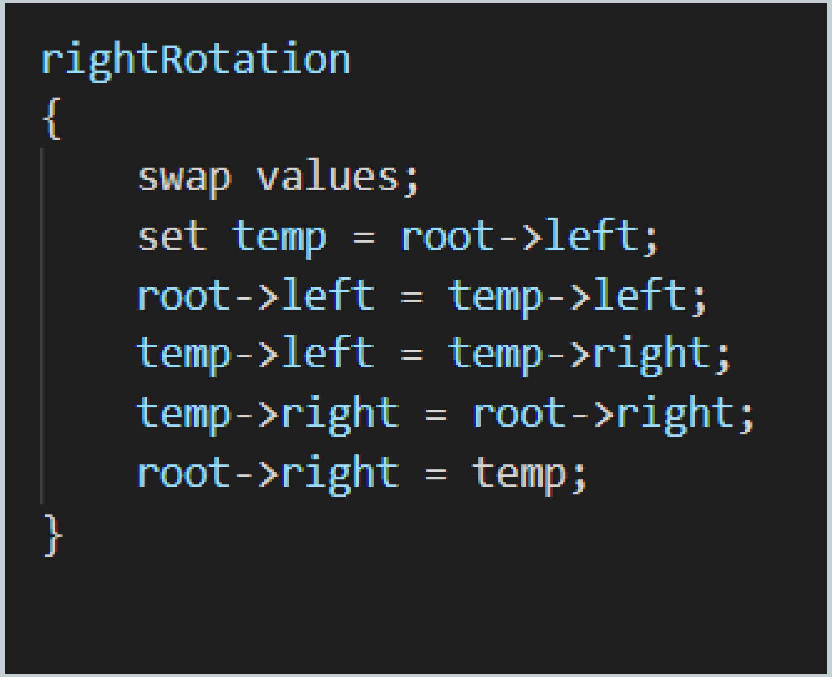
DSW ALGORITHM – INSERTING



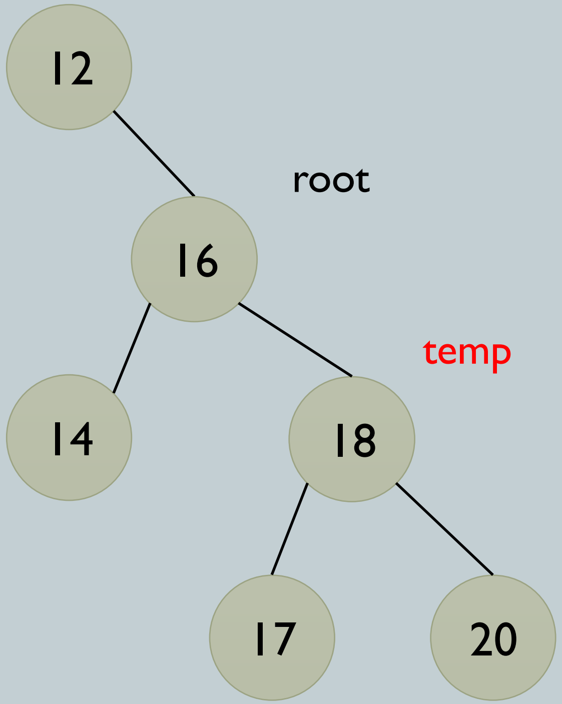
```
node* root = NULL;

node* insert(int val, node* ptr)
{
    if(ptr == NULL)
    {
        ptr = new node;
        ptr->data = val;
        ptr->left = NULL;
        ptr->right = NULL;
    }
    else if(val < ptr->data)
    {
        ptr->left = insert(val, ptr->left);
    }
    else
    {
        ptr->right= insert(val, ptr->right);
    }

    return ptr;
}
```

DSW ALGORITHM – RIGHT ROTATION

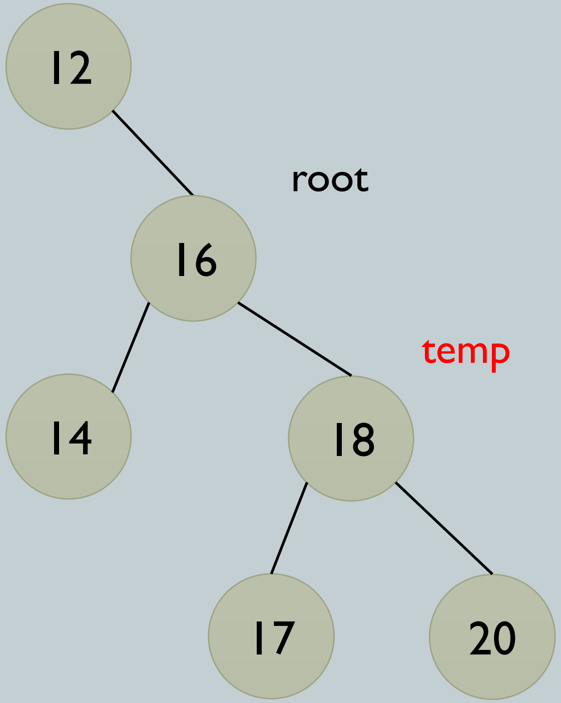


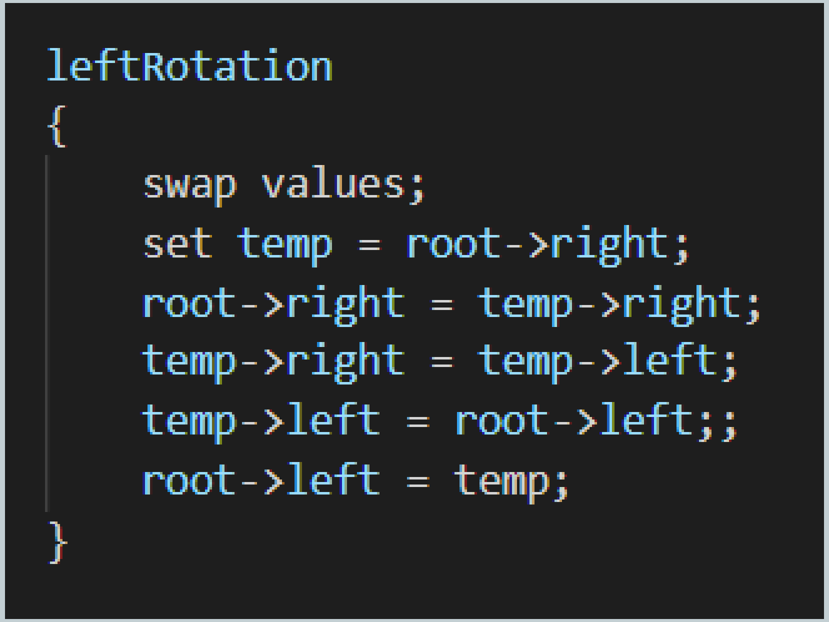


```
rightRotation
{
    swap values;
    set temp = root->left;
    root->left = temp->left;
    temp->left = temp->right;
    temp->right = root->right;
    root->right = temp;
}
```

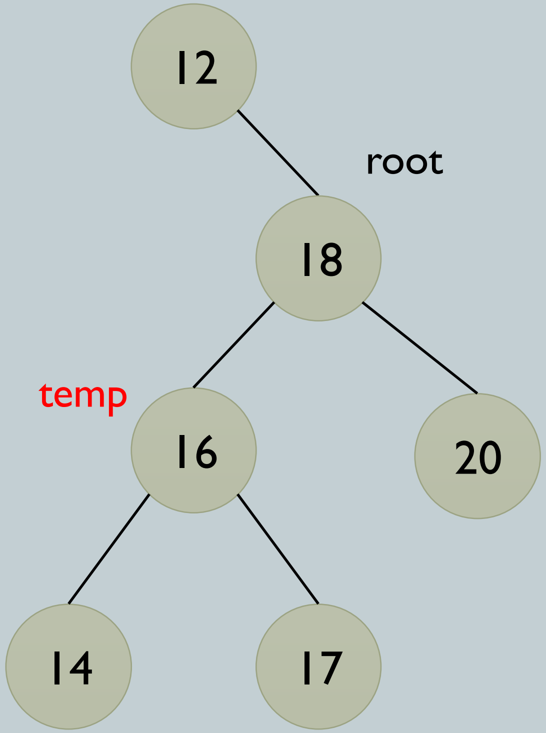


DSW ALGORITHM – LEFT ROTATION





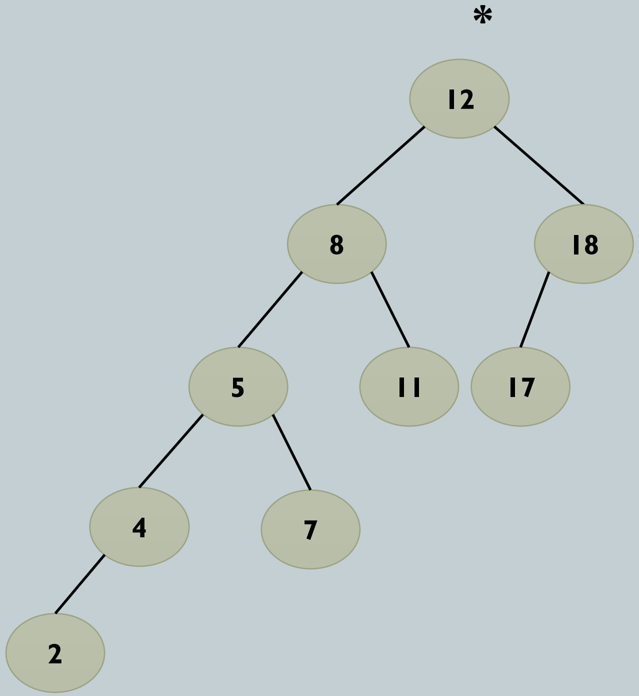
```
leftRotation
{
    swap values;
    set temp = root->right;
    root->right = temp->right;
    temp->right = temp->left;
    temp->left = root->left;;
    root->left = temp;
}
```

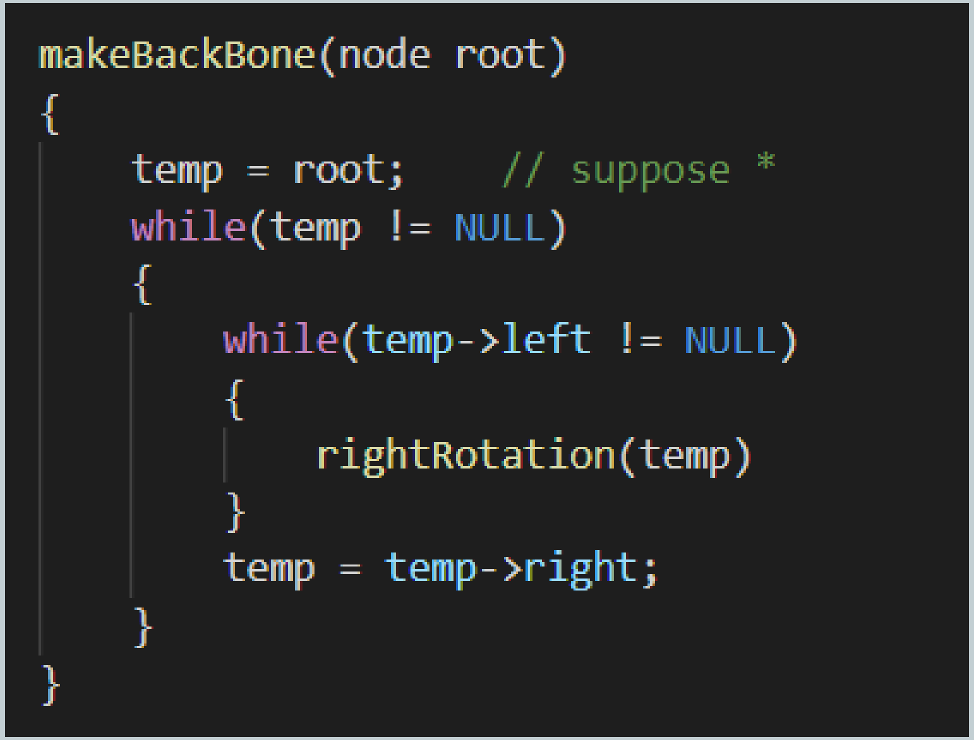


DSW ALGORITHM – MAKE BACKBONE

Starting from the root node

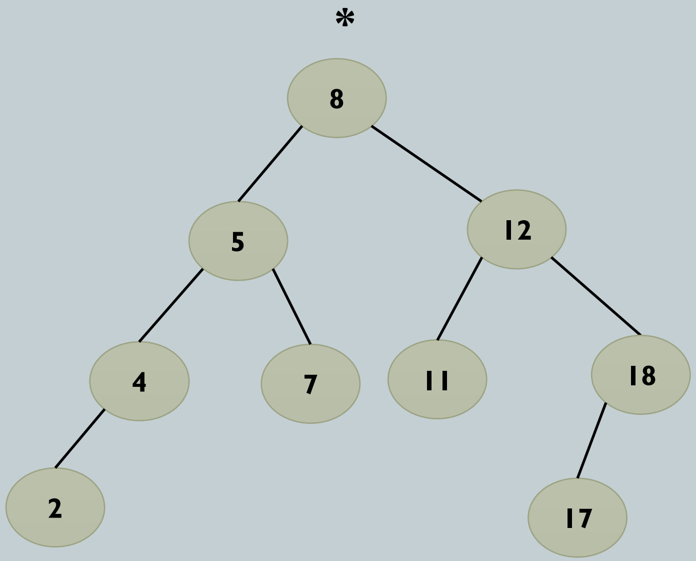
-> Do right rotation on nodes have left childs

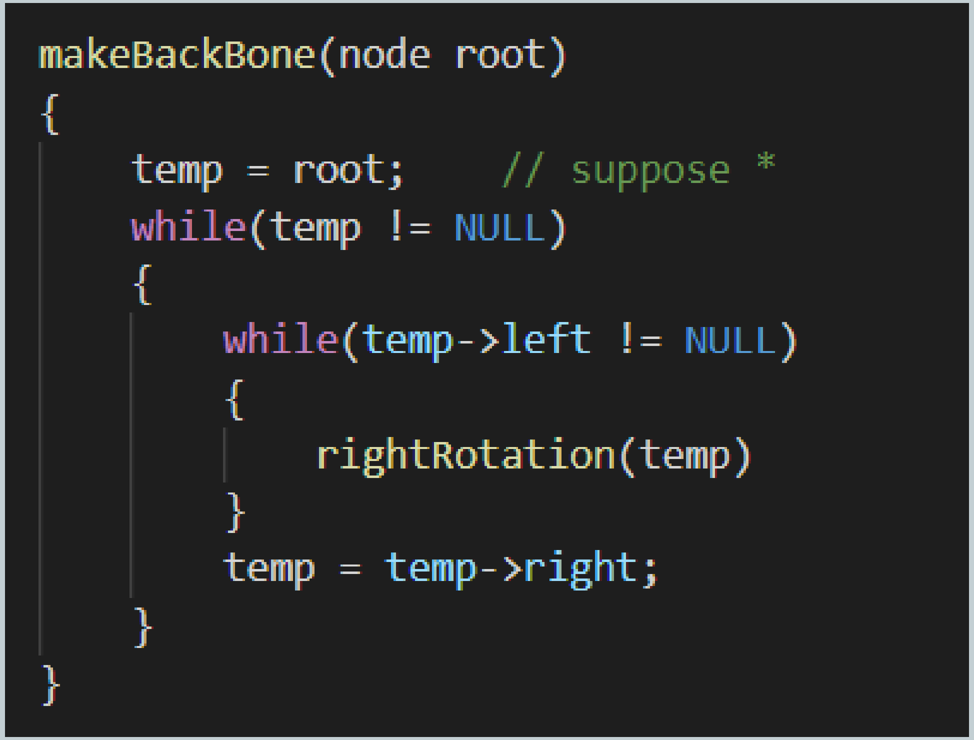




```
makeBackBone(node root)
{
    temp = root;    // suppose *
    while(temp != NULL)
    {
        while(temp->left != NULL)
        {
            rightRotation(temp)
        }
        temp = temp->right;
    }
}
```

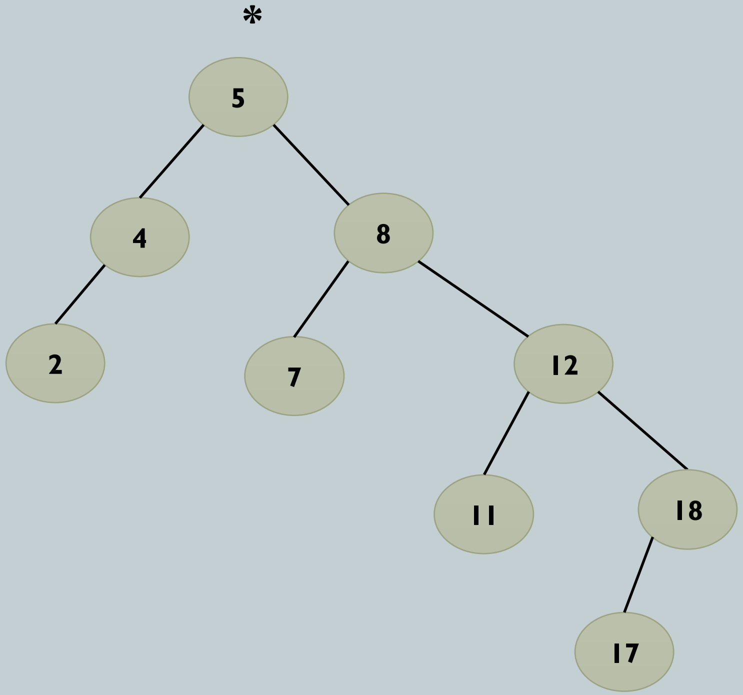
DSW ALGORITHM – MAKE BACKBONE

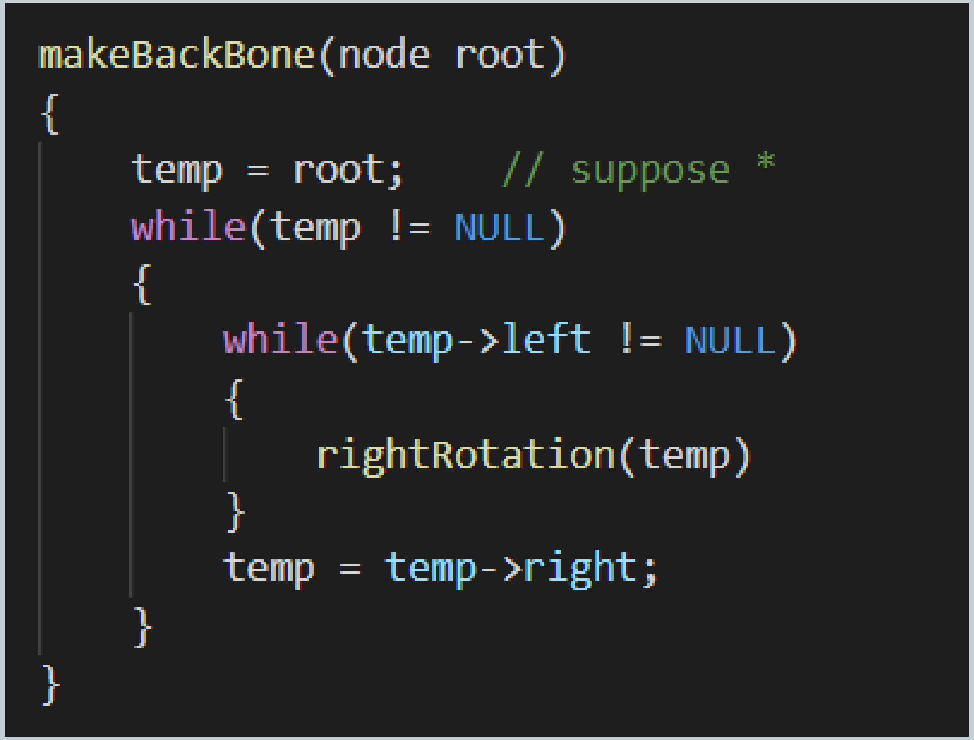




```
makeBackBone(node root)
{
    temp = root;    // suppose *
    while(temp != NULL)
    {
        while(temp->left != NULL)
        {
            rightRotation(temp)
        }
        temp = temp->right;
    }
}
```

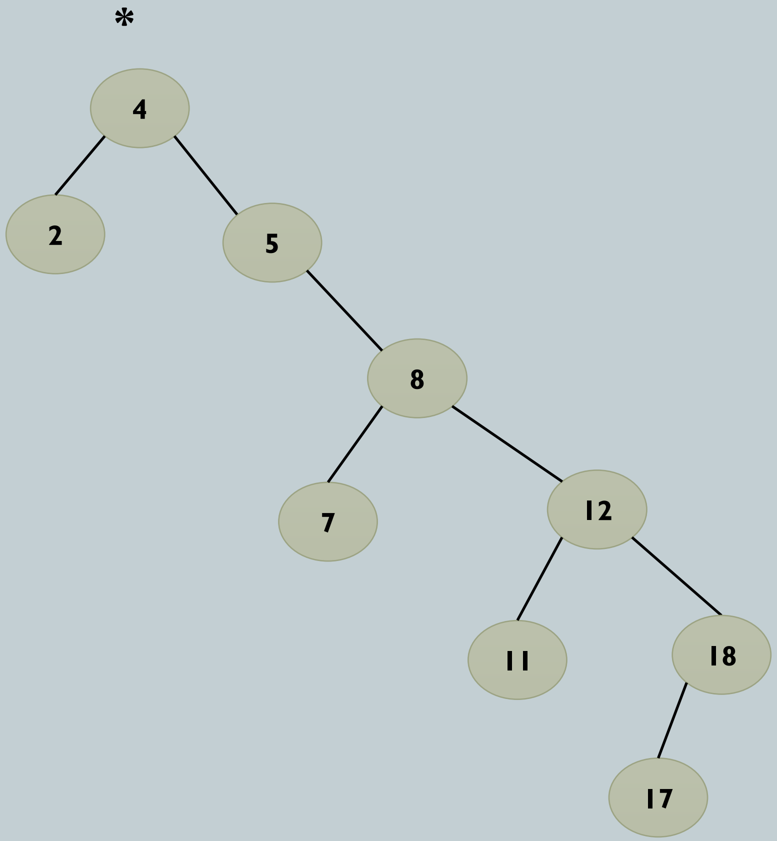
DSW ALGORITHM – MAKE BACKBONE

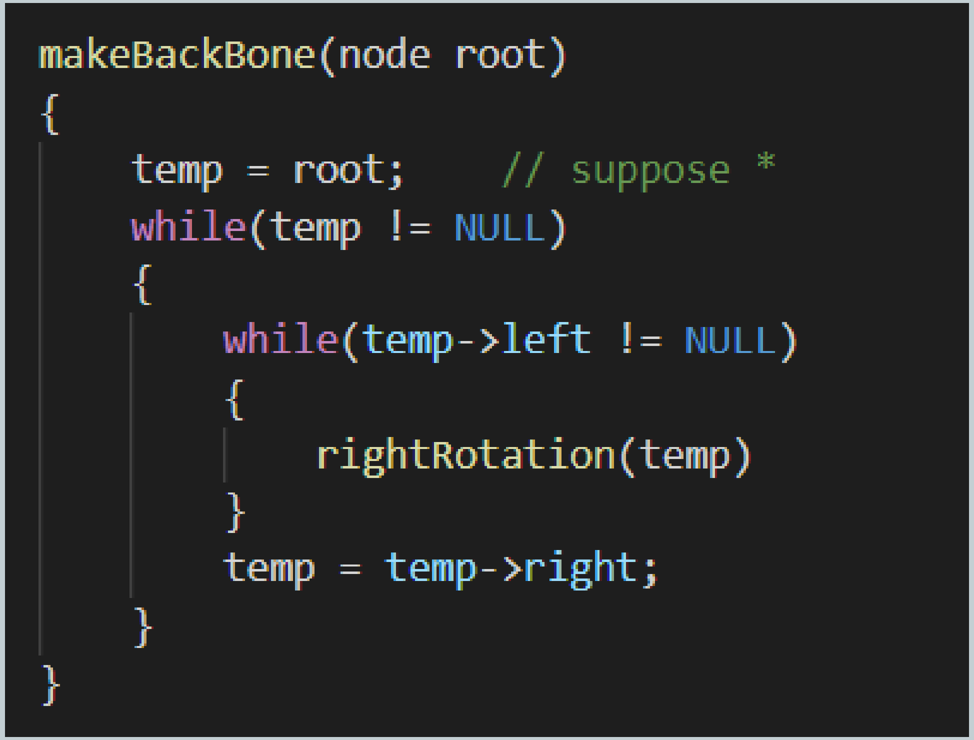




```
makeBackBone(node root)
{
    temp = root;    // suppose *
    while(temp != NULL)
    {
        while(temp->left != NULL)
        {
            rightRotation(temp)
        }
        temp = temp->right;
    }
}
```

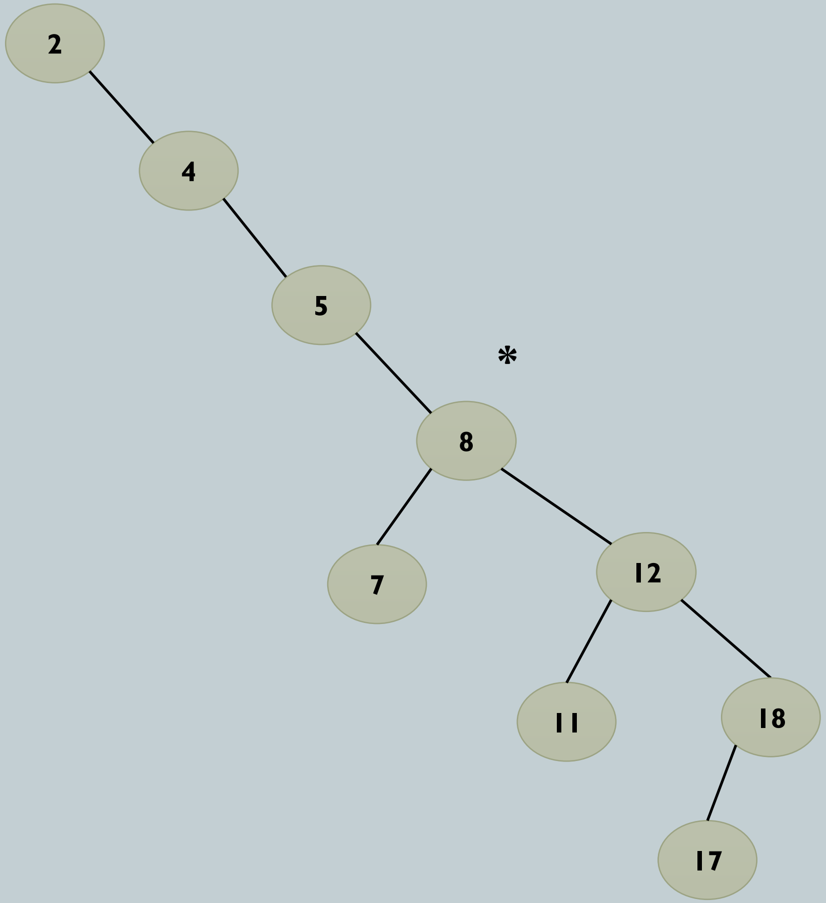
DSW ALGORITHM – MAKE BACKBONE

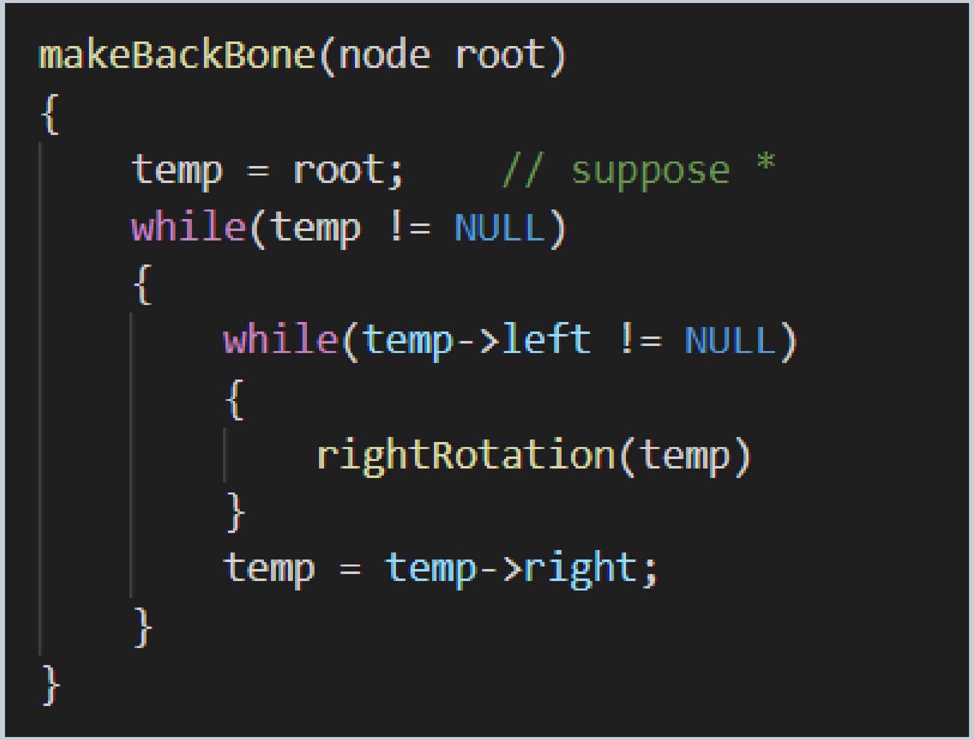




```
makeBackBone(node root)
{
    temp = root;    // suppose *
    while(temp != NULL)
    {
        while(temp->left != NULL)
        {
            rightRotation(temp)
        }
        temp = temp->right;
    }
}
```

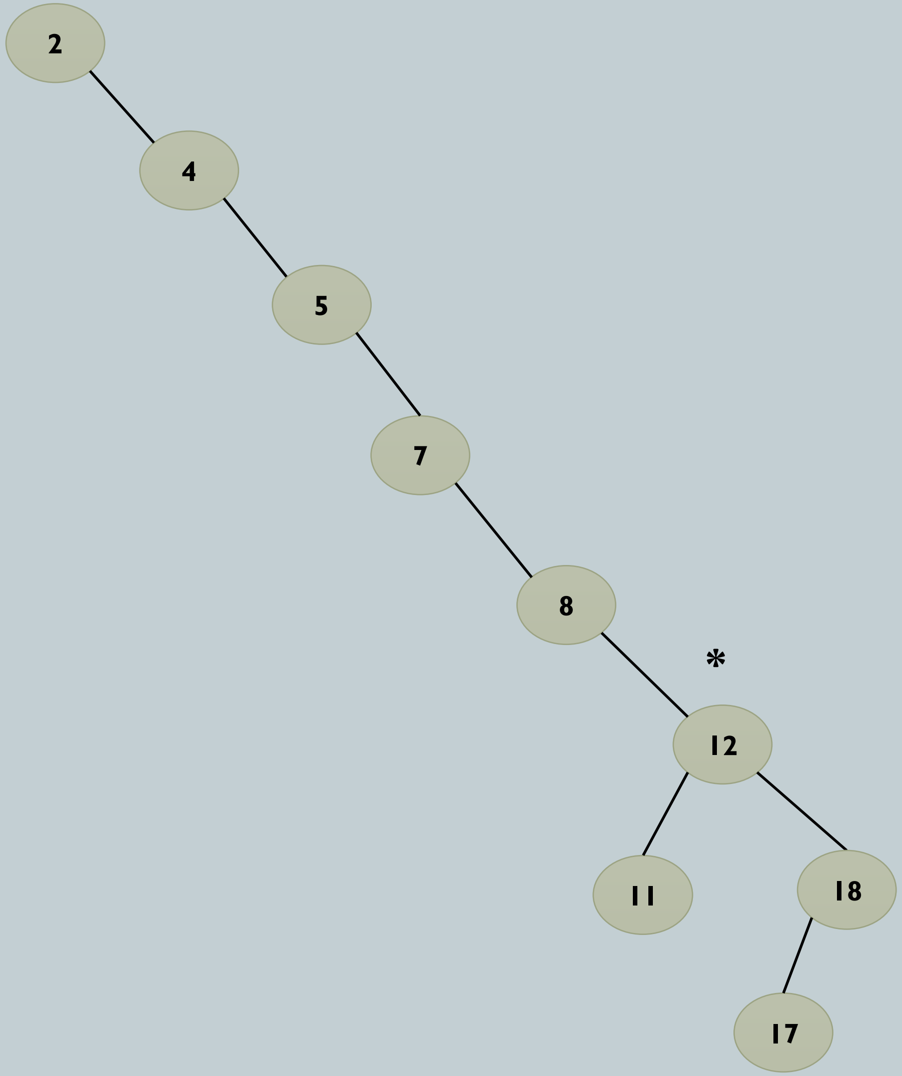
DSW ALGORITHM – MAKE BACKBONE

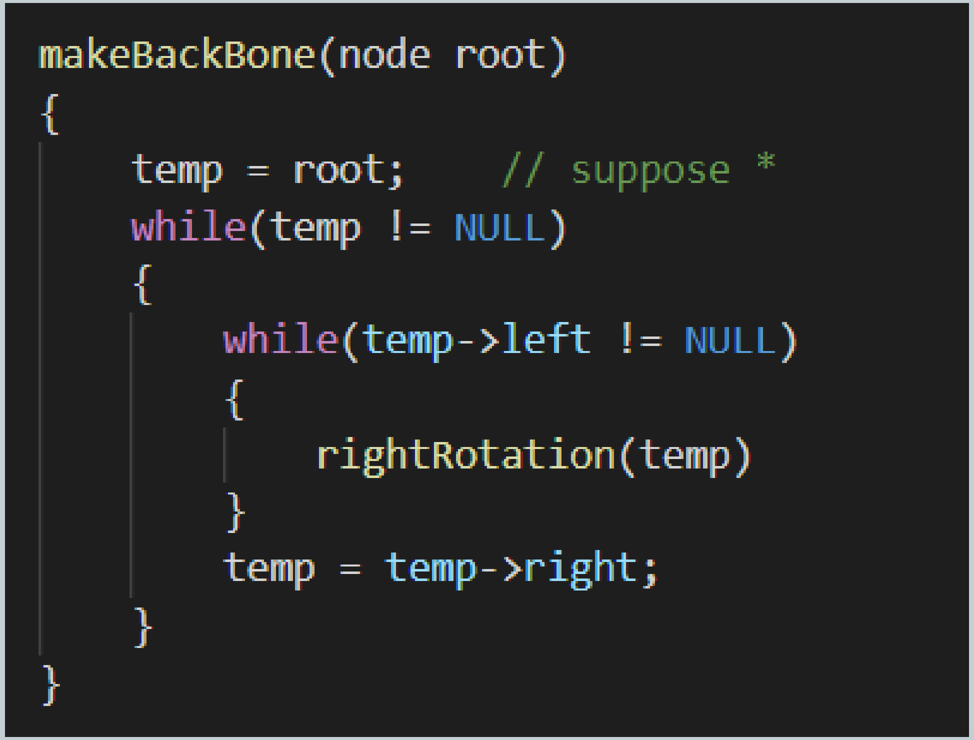




```
makeBackBone(node root)
{
    temp = root;    // suppose *
    while(temp != NULL)
    {
        while(temp->left != NULL)
        {
            rightRotation(temp)
        }
        temp = temp->right;
    }
}
```

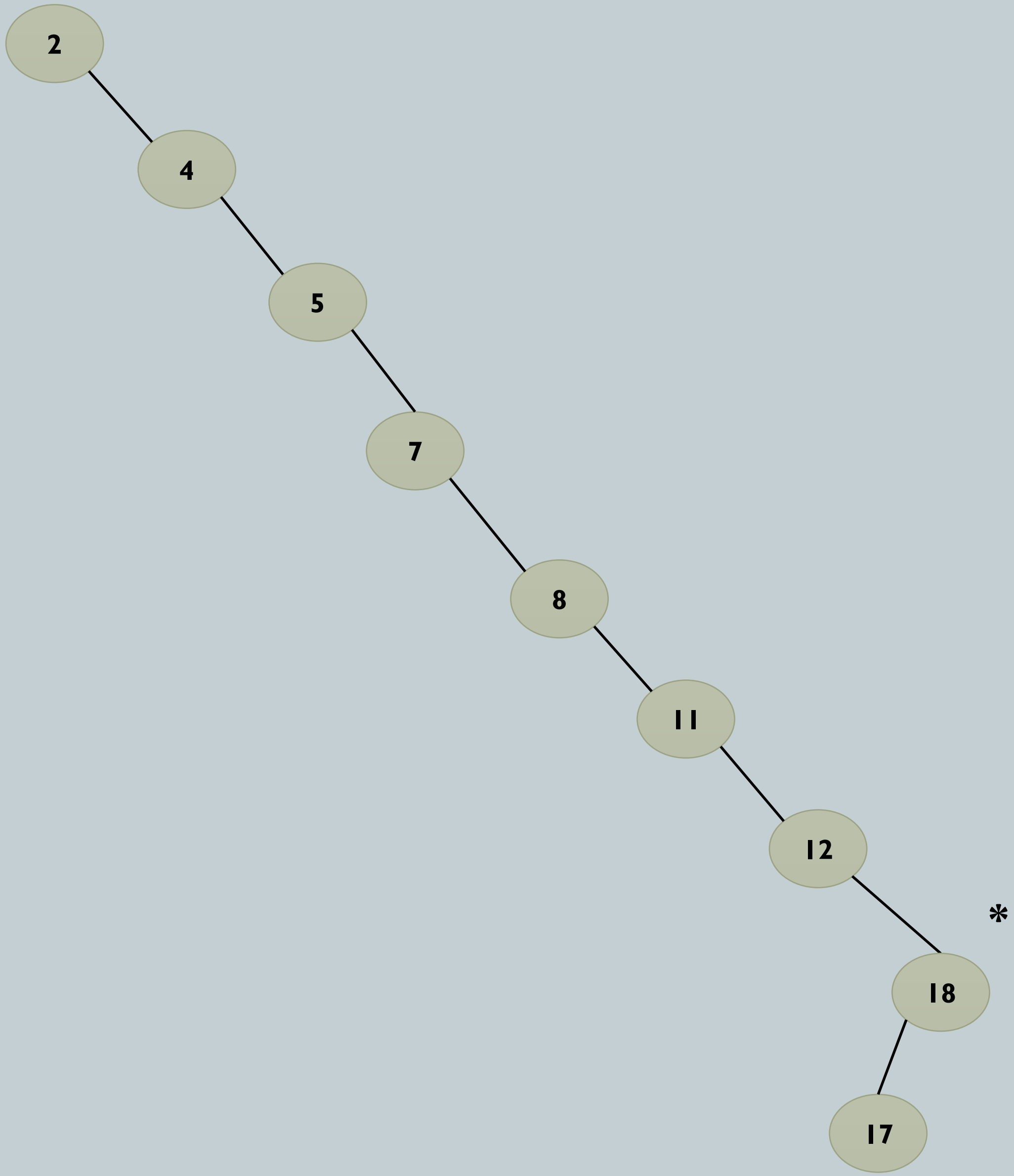
DSW ALGORITHM – MAKE BACKBONE

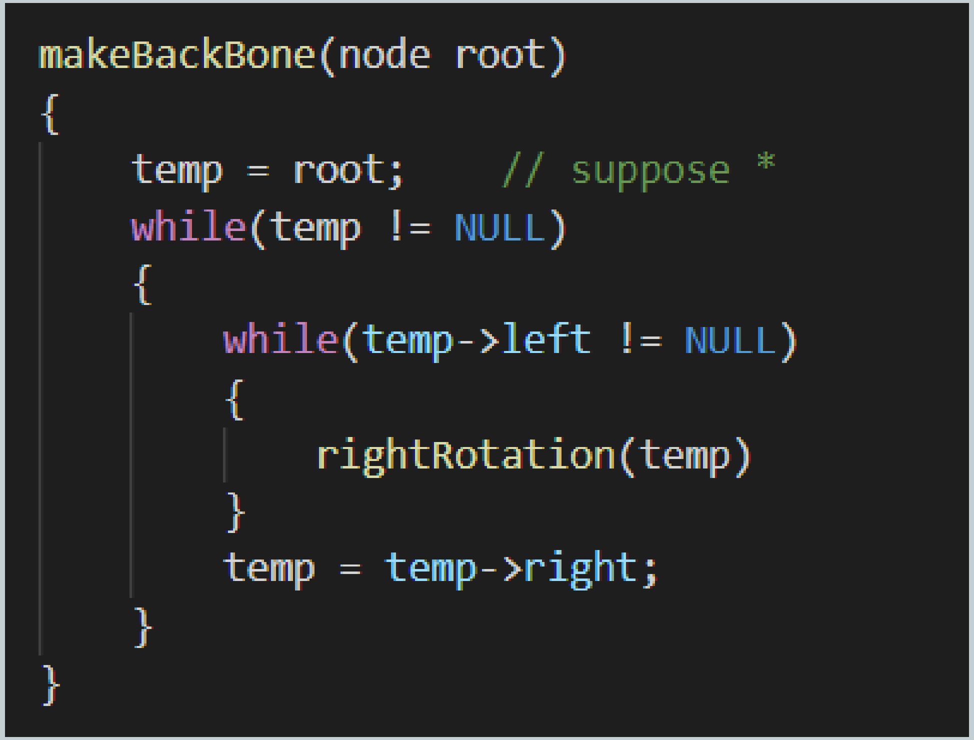




```
makeBackBone(node root)
{
    temp = root;    // suppose *
    while(temp != NULL)
    {
        while(temp->left != NULL)
        {
            rightRotation(temp)
        }
        temp = temp->right;
    }
}
```

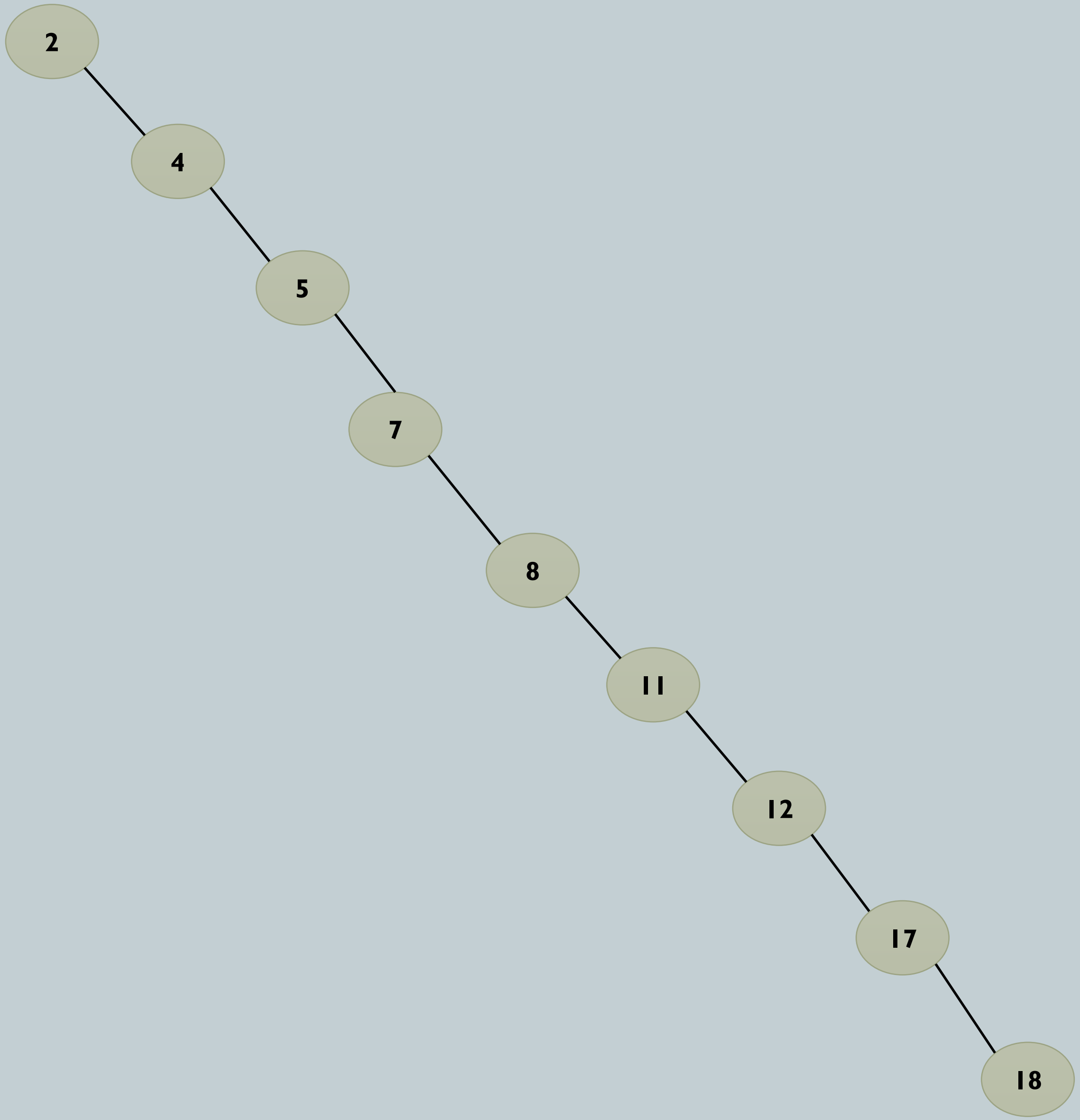
DSW ALGORITHM – MAKE BACKBONE

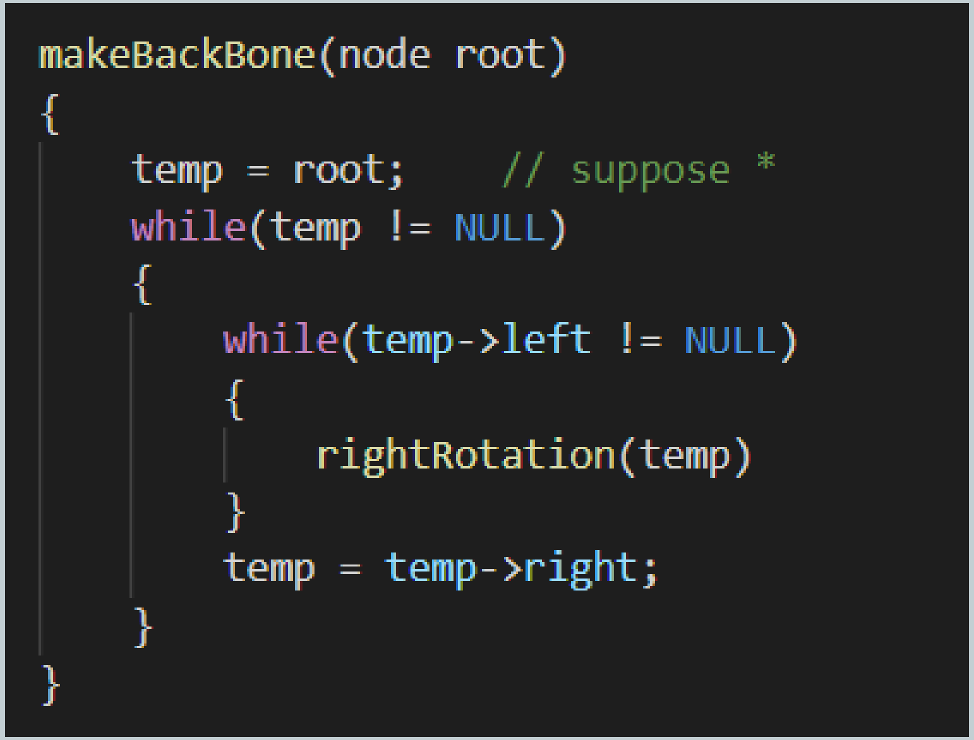




```
makeBackBone(node root)
{
    temp = root;    // suppose *
    while(temp != NULL)
    {
        while(temp->left != NULL)
        {
            rightRotation(temp)
        }
        temp = temp->right;
    }
}
```

DSW ALGORITHM – MAKE BACKBONE





```
makeBackBone(node root)
{
    temp = root;    // suppose *
    while(temp != NULL)
    {
        while(temp->left != NULL)
        {
            rightRotation(temp)
        }
        temp = temp->right;
    }
}
```

Done

DSW ALGORITHM – BALANCE THE BACKBONE

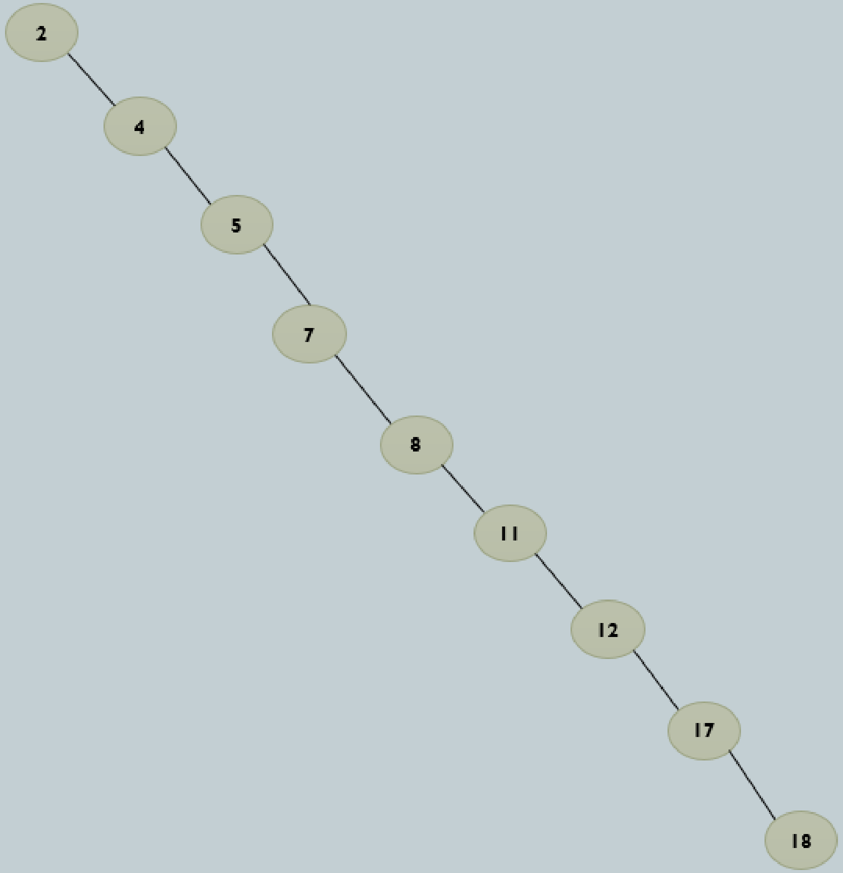
So now we have our backbone, a completely straight tree that is lifted to right. Now we have to perform series of left rotations to create a balanced binary tree.

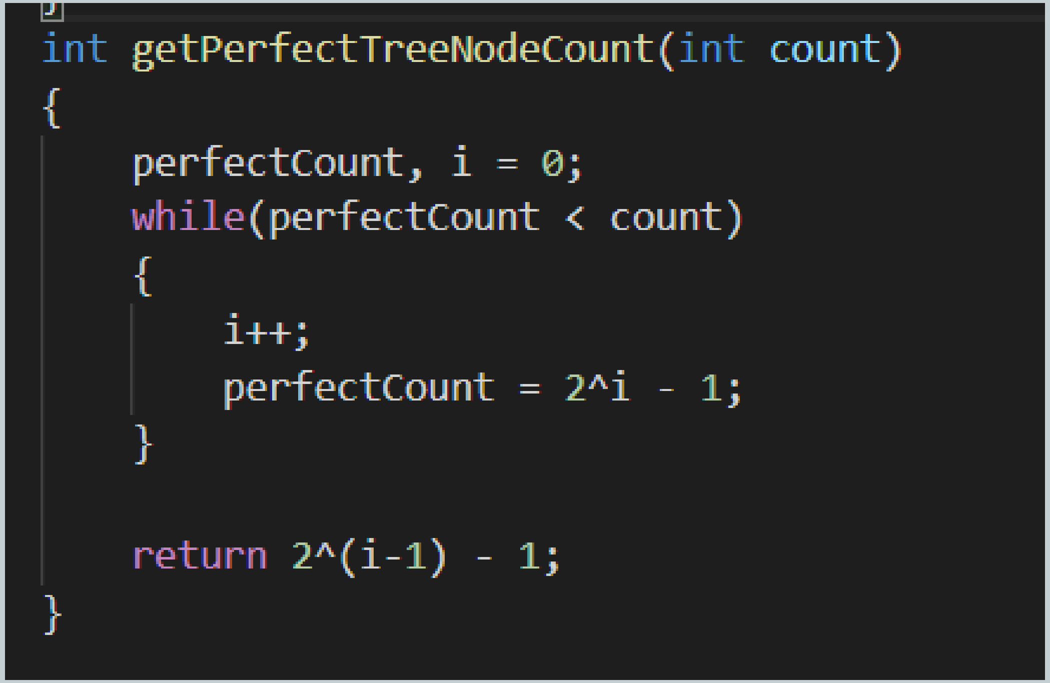
The first thing we have to calculate how many nodes we will be having at the bottommost level.

General balanced binary trees contain 2^n-1 total nodes for each level n

DSW ALGORITHM – BALANCE THE BACKBONE

In our case backbone has 9 nodes, so the balanced binary tree having 7 + 2 nodes which means 2 node at the bottom most level. So now we need perfect tree node count (i.e, to get extra node count)





```
int getPerfectTreeNodeCount(int count)
{
    perfectCount, i = 0;
    while(perfectCount < count)
    {
        i++;
        perfectCount = 2^i - 1;
    }

    return 2^(i-1) - 1;
}
```

DSW ALGORITHM – BALANCE THE BACKBONE

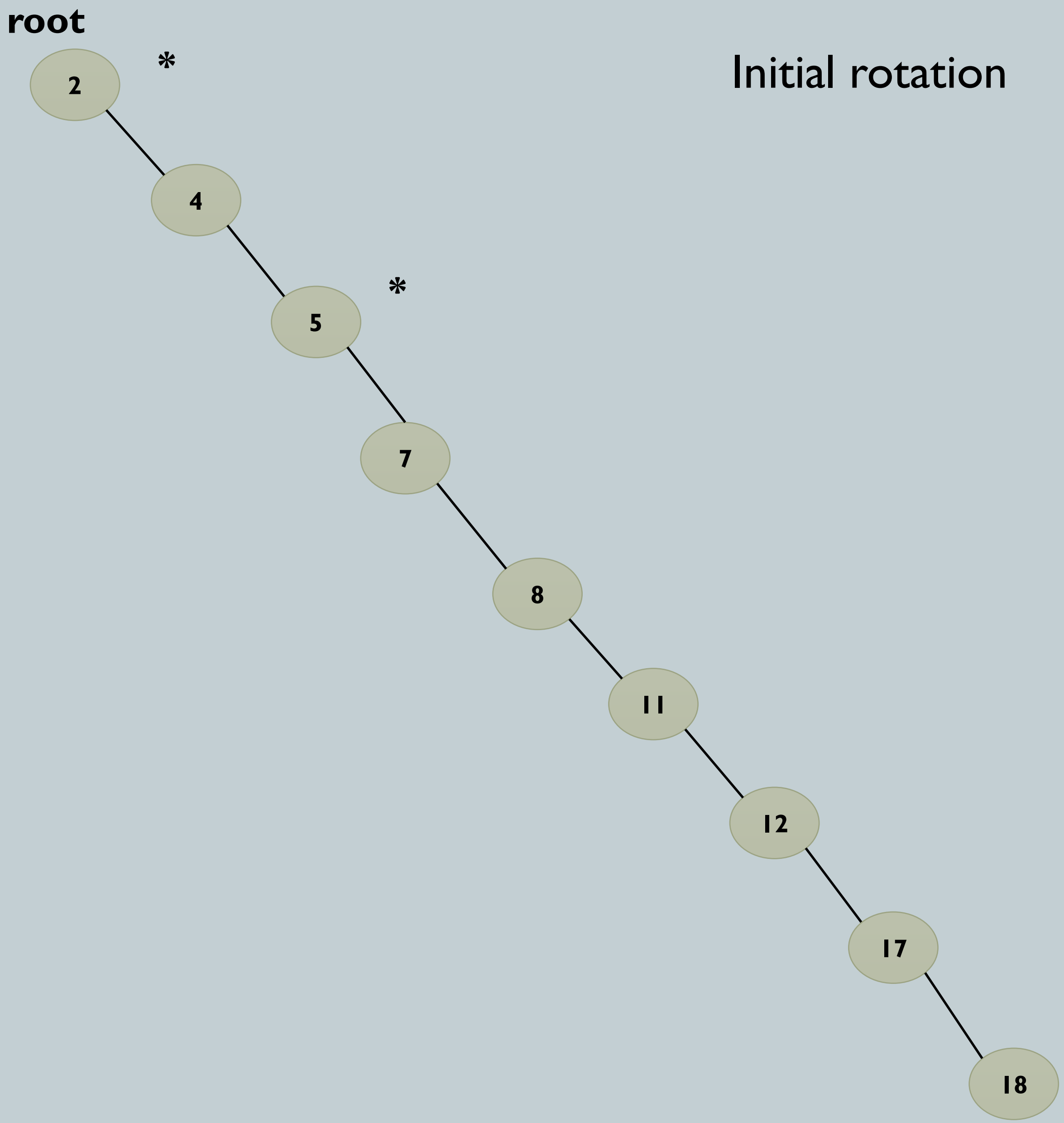
Now we can do the balance rotations,

1). First perform left rotation on the odd nodes, starting from root until extra node count times (initial rotation)

2). Then again left rotate all the odd nodes starting from root until tree balanced

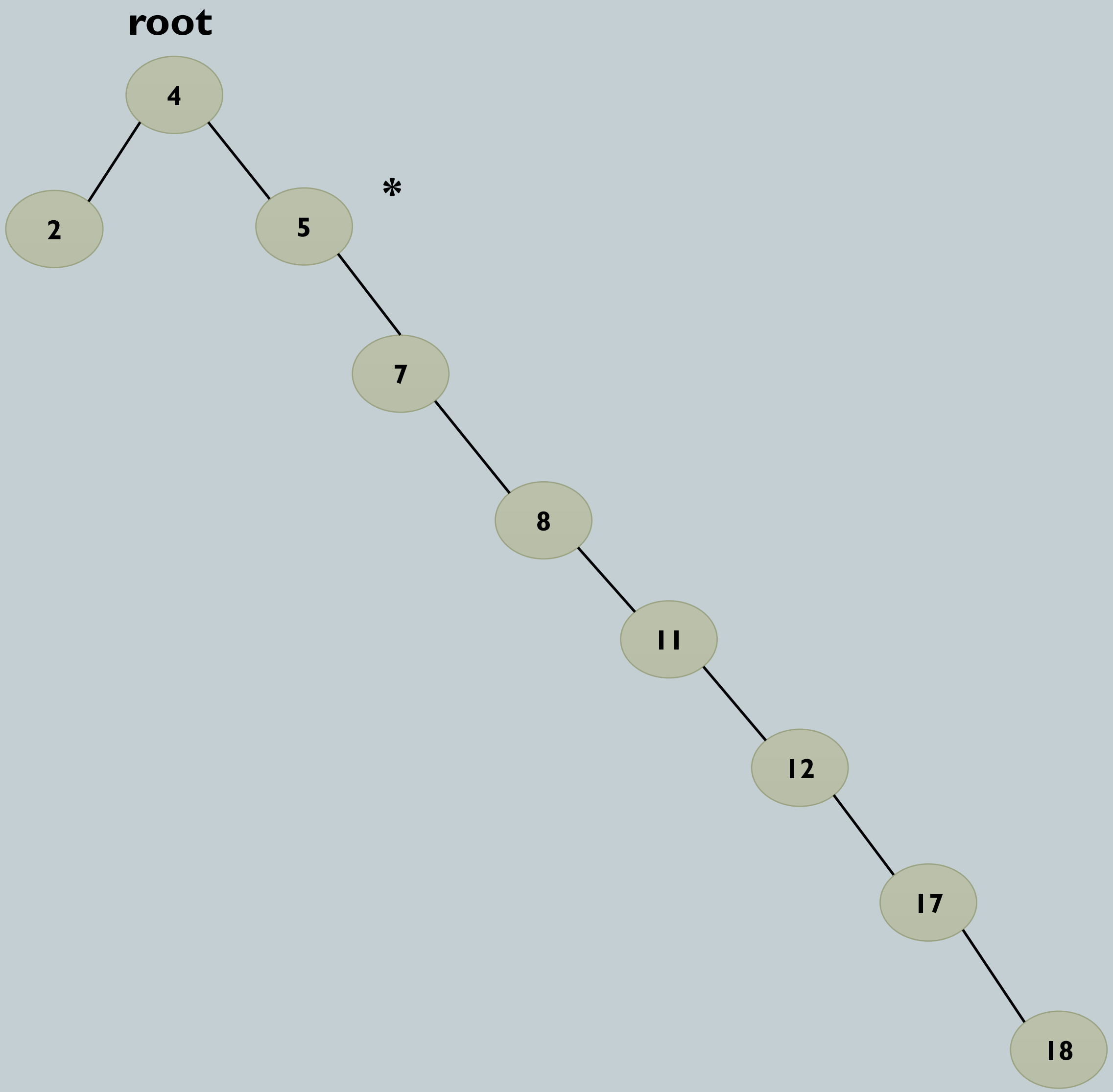
DSW ALGORITHM – BALANCE THE BACKBONE

Initial rotation



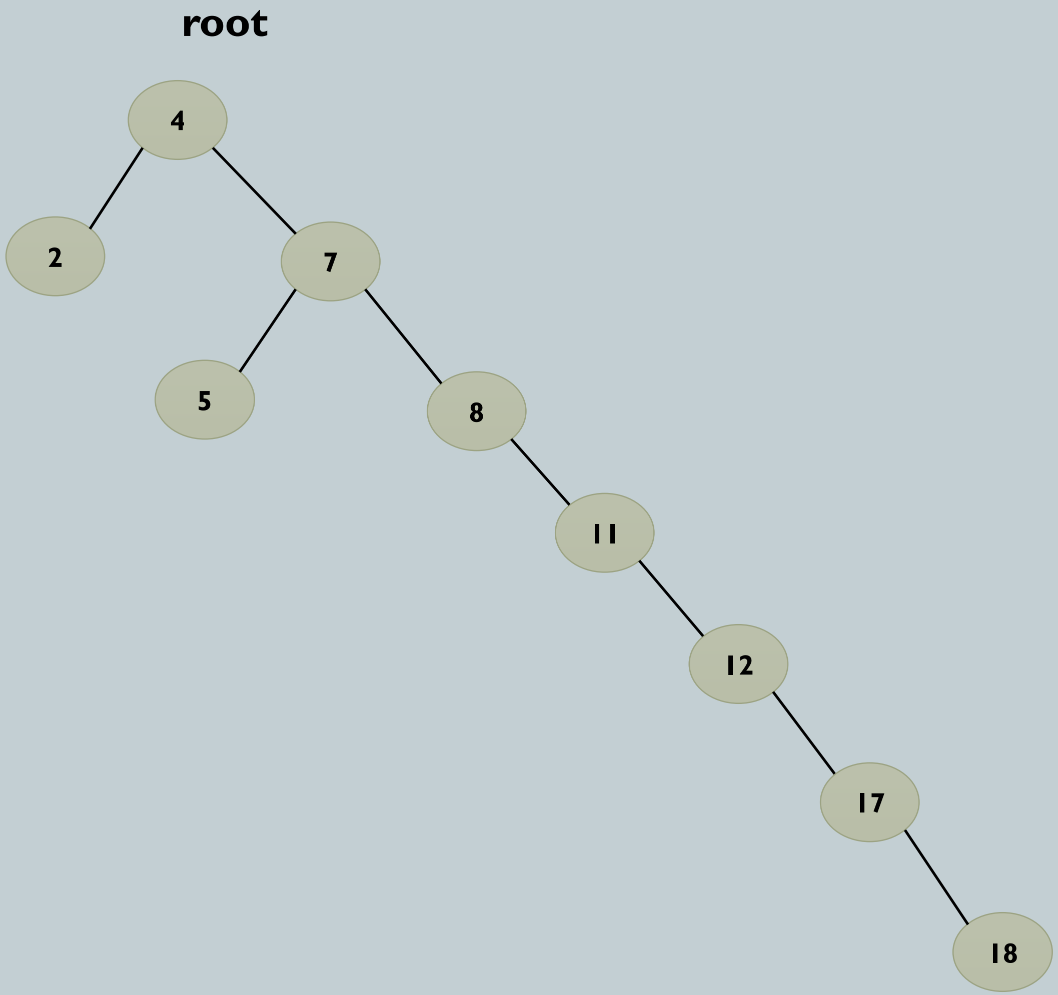
DSW ALGORITHM – BALANCE THE BACKBONE

Initial rotation

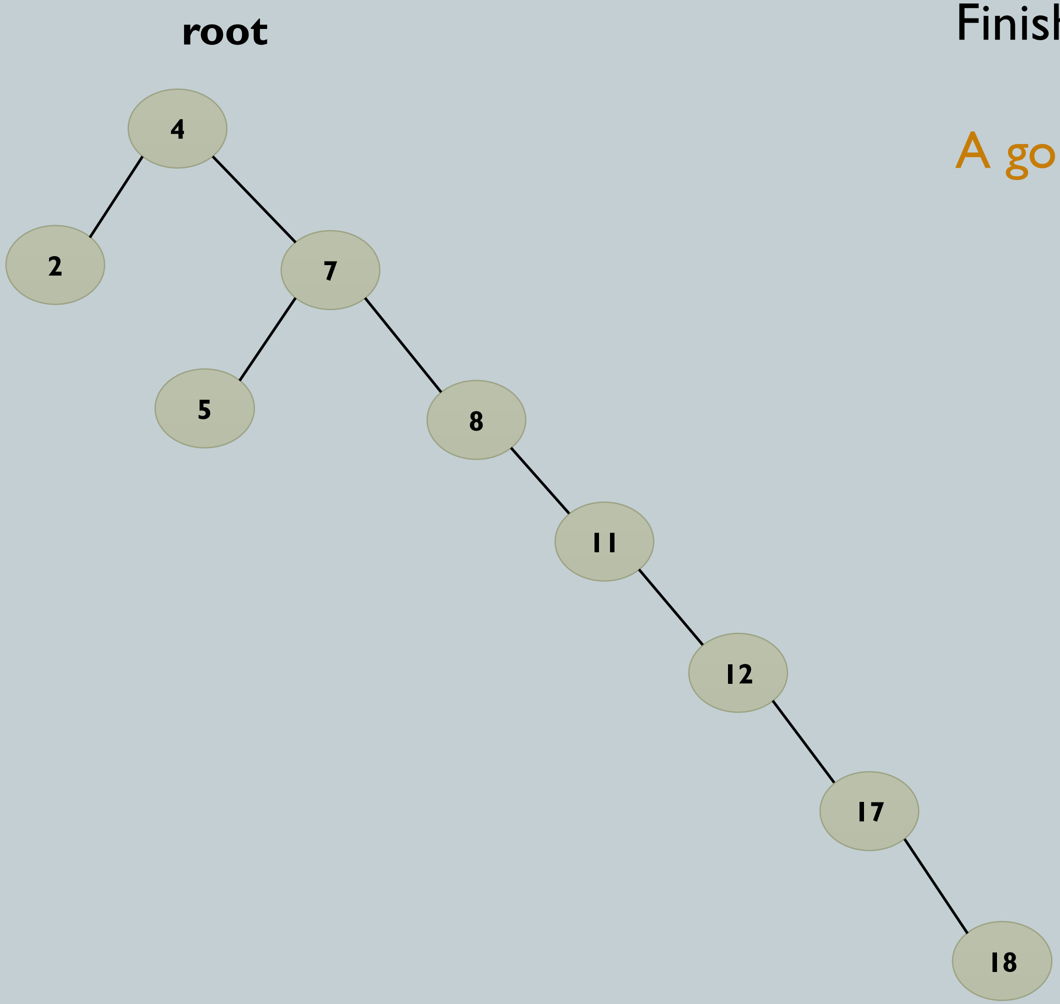


DSW ALGORITHM – BALANCE THE BACKBONE

Initial rotation



DSW ALGORITHM – BALANCE THE BACKBONE

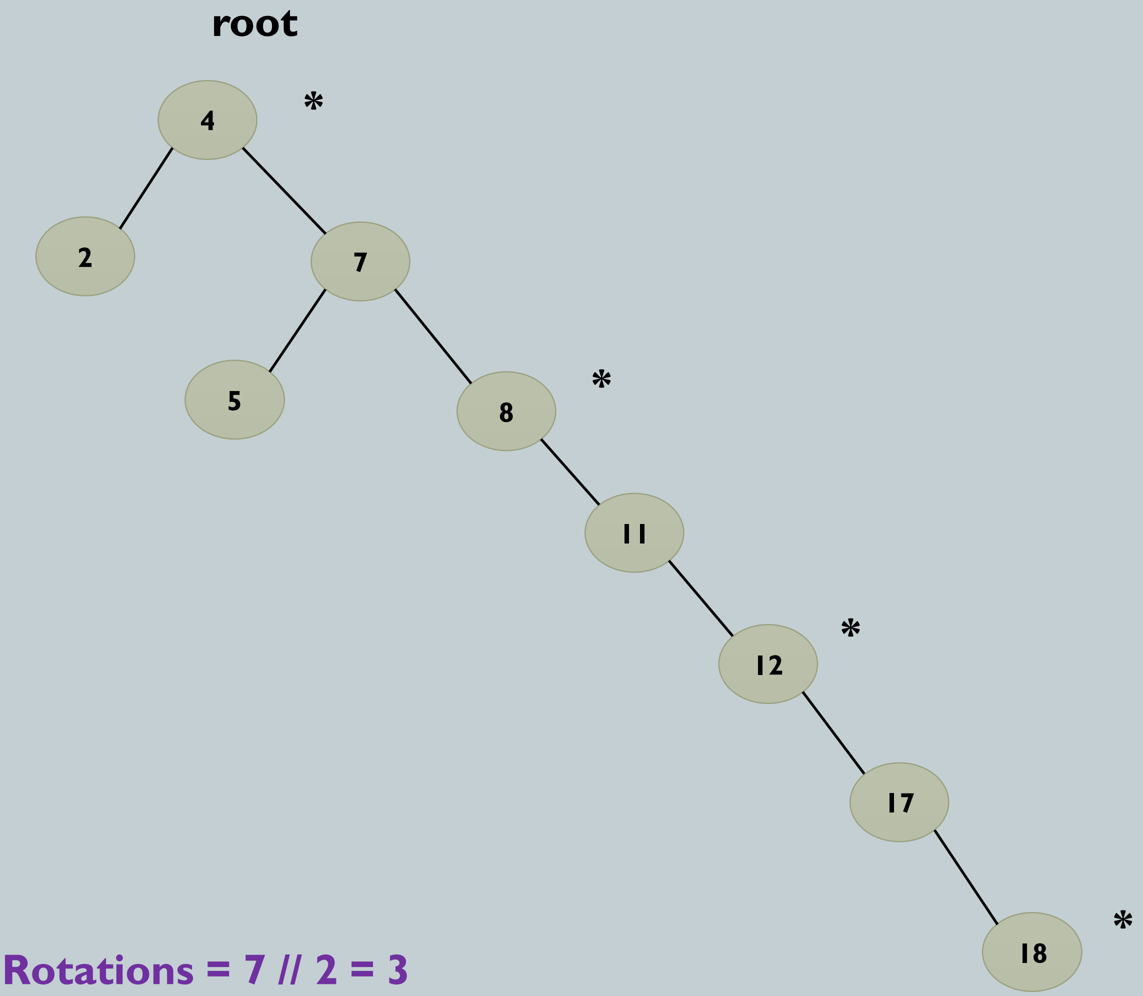


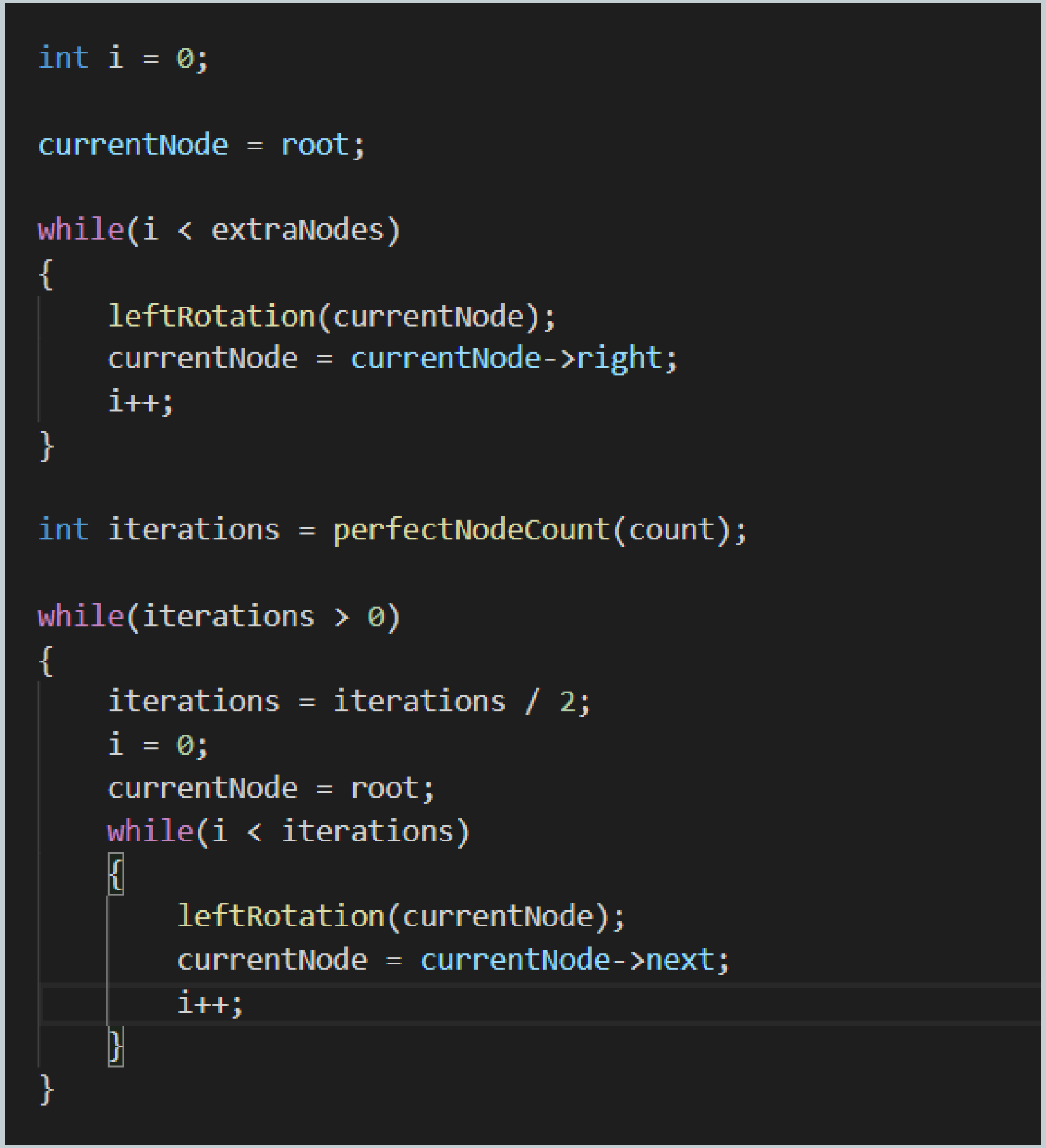
Finished Initial rotation

A gold hint : - Note that, these * marked nodes are the ones can be seen at the bottom most level.

DSW ALGORITHM – BALANCE THE BACKBONE

Second rotation series





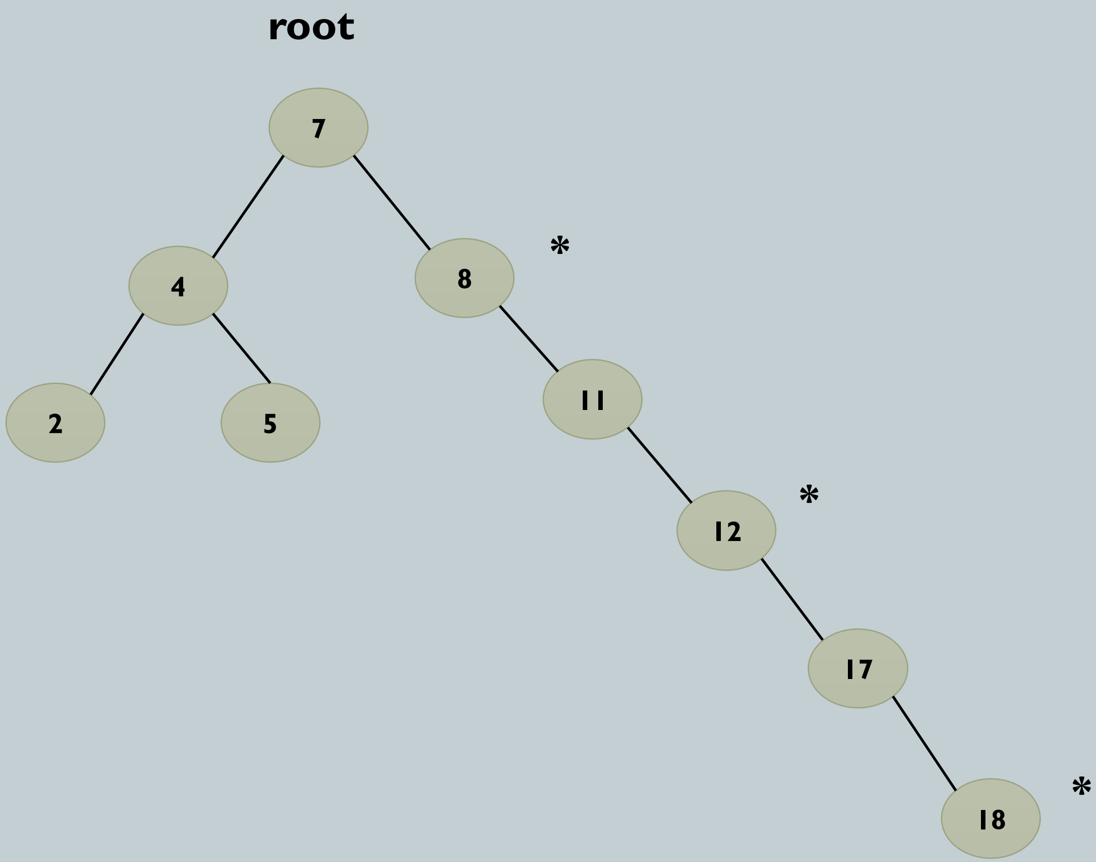
```
int i = 0;

currentNode = root;

while(i < extraNodes)
{
    leftRotation(currentNode);
    currentNode = currentNode->right;
    i++;
}

int iterations = perfectNodeCount(count);

while(iterations > 0)
{
    iterations = iterations / 2;
    i = 0;
    currentNode = root;
    while(i < iterations)
    {
        leftRotation(currentNode);
        currentNode = currentNode->next;
        i++;
    }
}
```

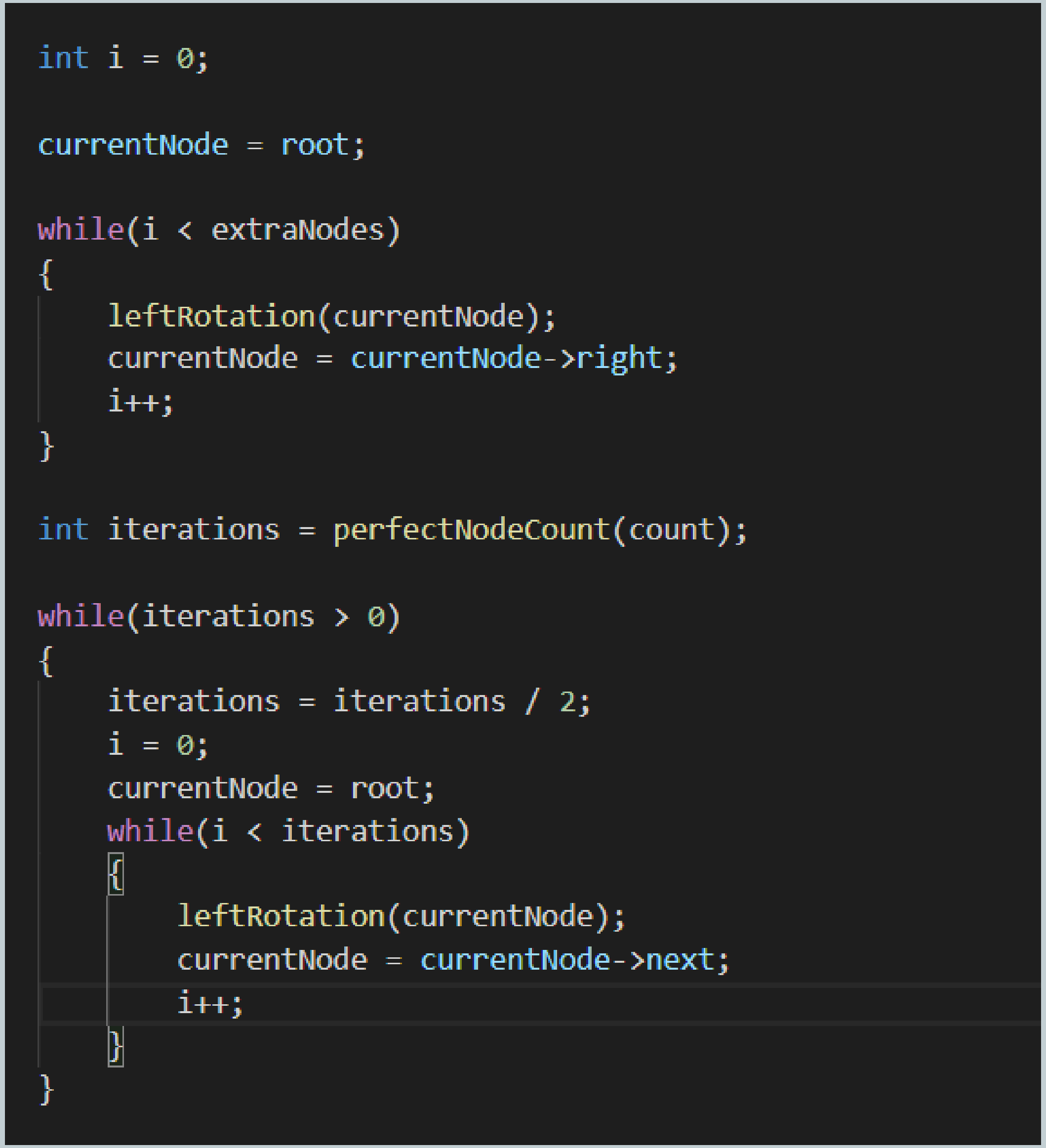
Rotations = 7 // 2 = 3

Mark nodes

DSW ALGORITHM – BALANCE THE BACKBONE

Second rotation series





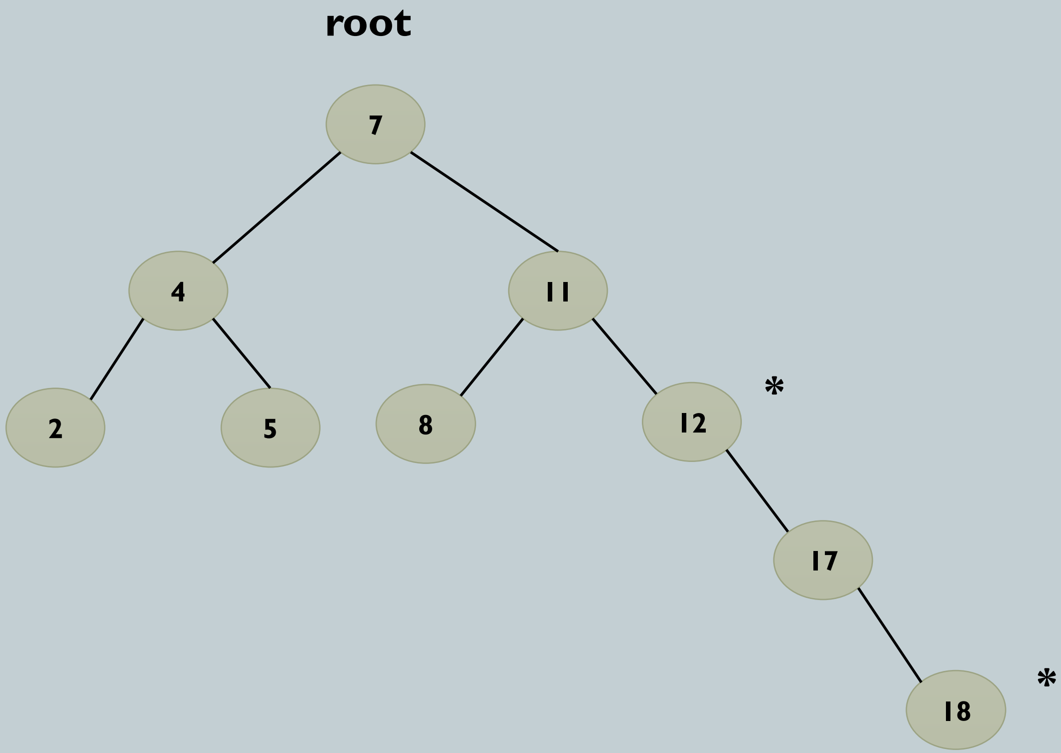
```
int i = 0;

currentNode = root;

while(i < extraNodes)
{
    leftRotation(currentNode);
    currentNode = currentNode->right;
    i++;
}

int iterations = perfectNodeCount(count);

while(iterations > 0)
{
    iterations = iterations / 2;
    i = 0;
    currentNode = root;
    while(i < iterations)
    {
        leftRotation(currentNode);
        currentNode = currentNode->next;
        i++;
    }
}
```

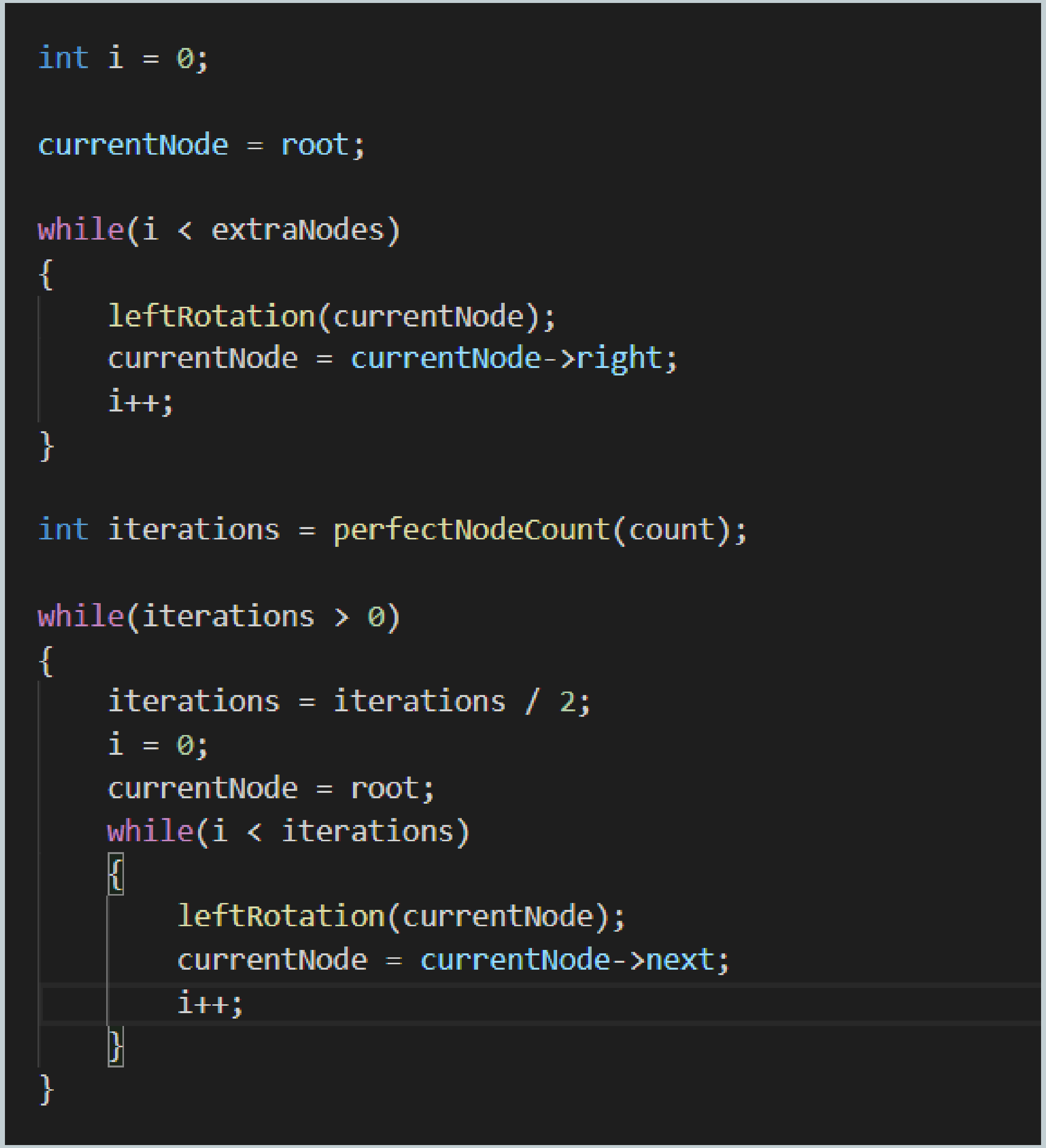
Rotations = 7 // 2 = 3

1 rotation

DSW ALGORITHM – BALANCE THE BACKBONE

Second rotation series





```
int i = 0;

currentNode = root;

while(i < extraNodes)
{
    leftRotation(currentNode);
    currentNode = currentNode->right;
    i++;
}

int iterations = perfectNodeCount(count);

while(iterations > 0)
{
    iterations = iterations / 2;
    i = 0;
    currentNode = root;
    while(i < iterations)
    {
        leftRotation(currentNode);
        currentNode = currentNode->next;
        i++;
    }
}
```

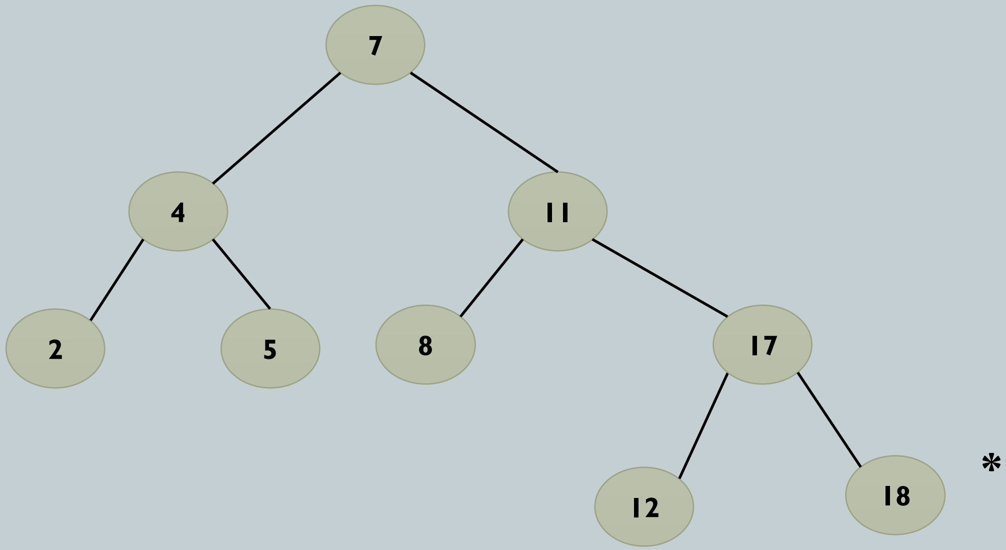
Rotations = 7 // 2 = 3

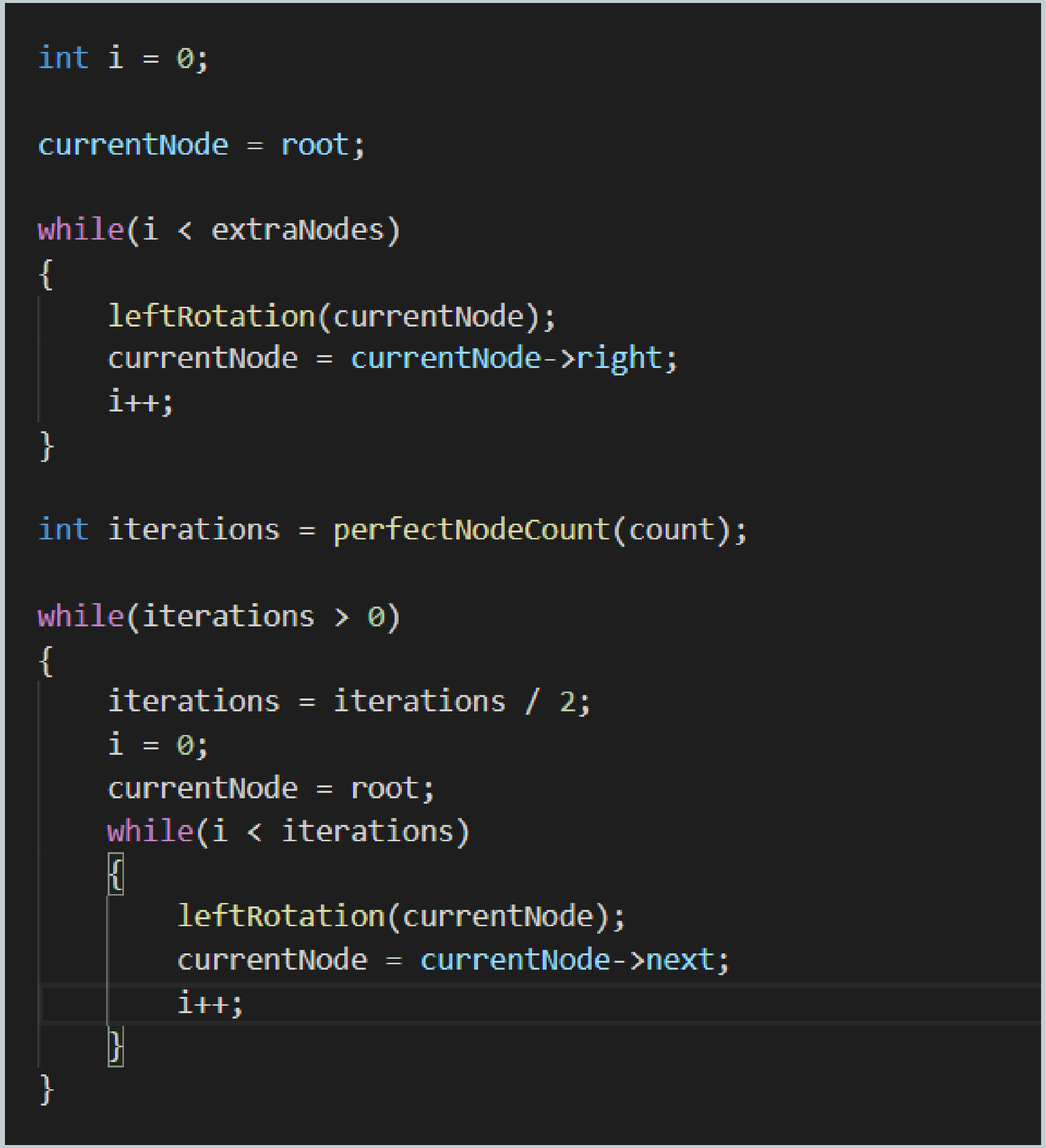
2 rotation

DSW ALGORITHM – BALANCE THE BACKBONE

Second rotation series

root





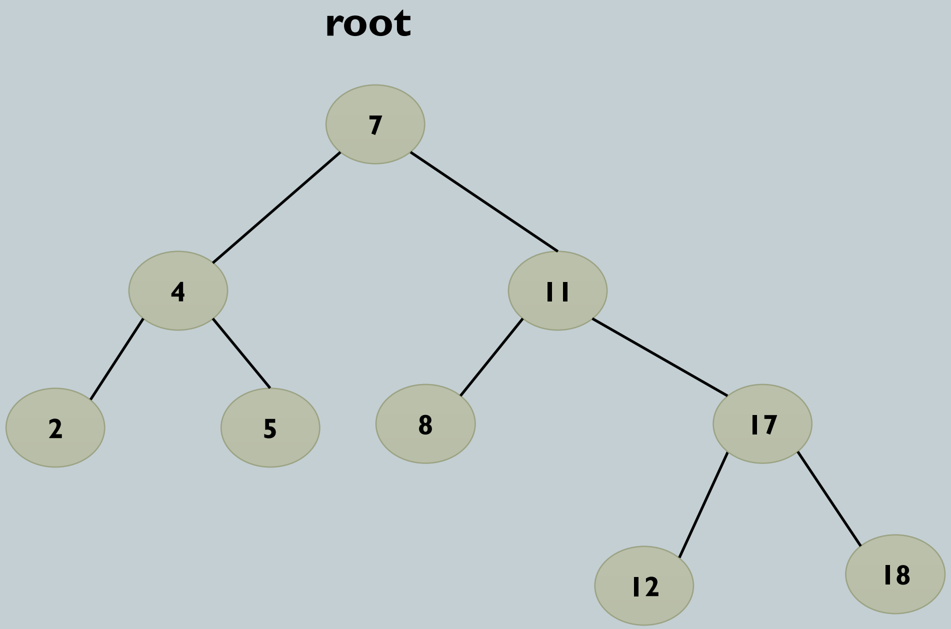
```
int i = 0;

currentNode = root;

while(i < extraNodes)
{
    leftRotation(currentNode);
    currentNode = currentNode->right;
    i++;
}

int iterations = perfectNodeCount(count);

while(iterations > 0)
{
    iterations = iterations / 2;
    i = 0;
    currentNode = root;
    while(i < iterations)
    {
        leftRotation(currentNode);
        currentNode = currentNode->next;
        i++;
    }
}
```

Rotations = 7 // 2 = 3

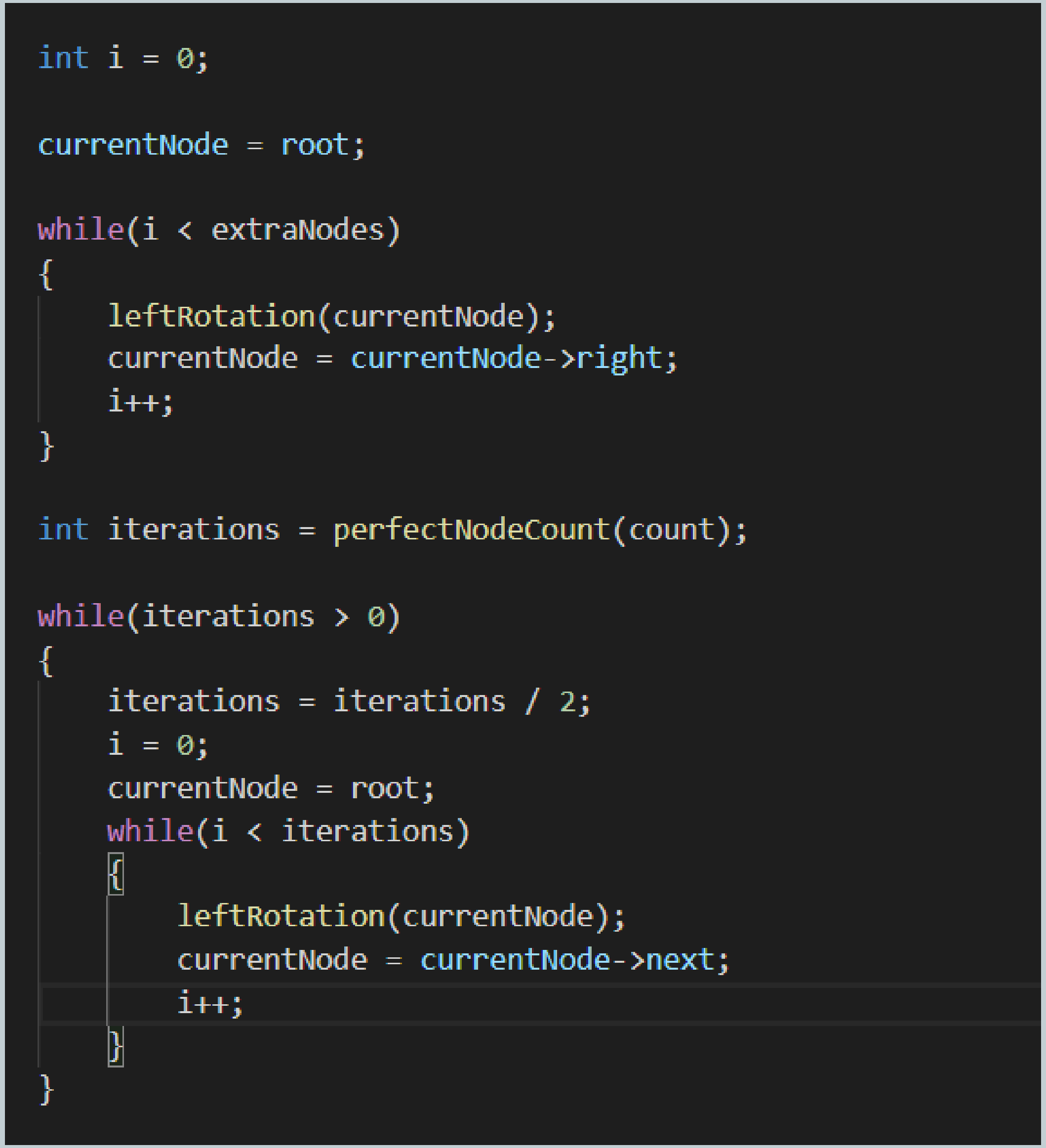
3 rotation

DSW ALGORITHM – BALANCE THE BACKBONE

Second rotation series



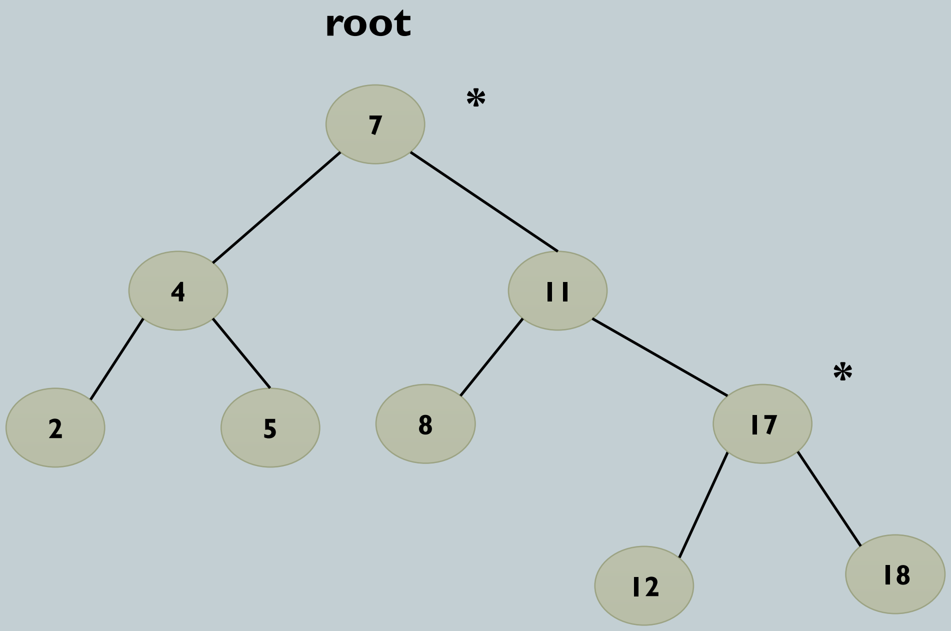
Then mark again



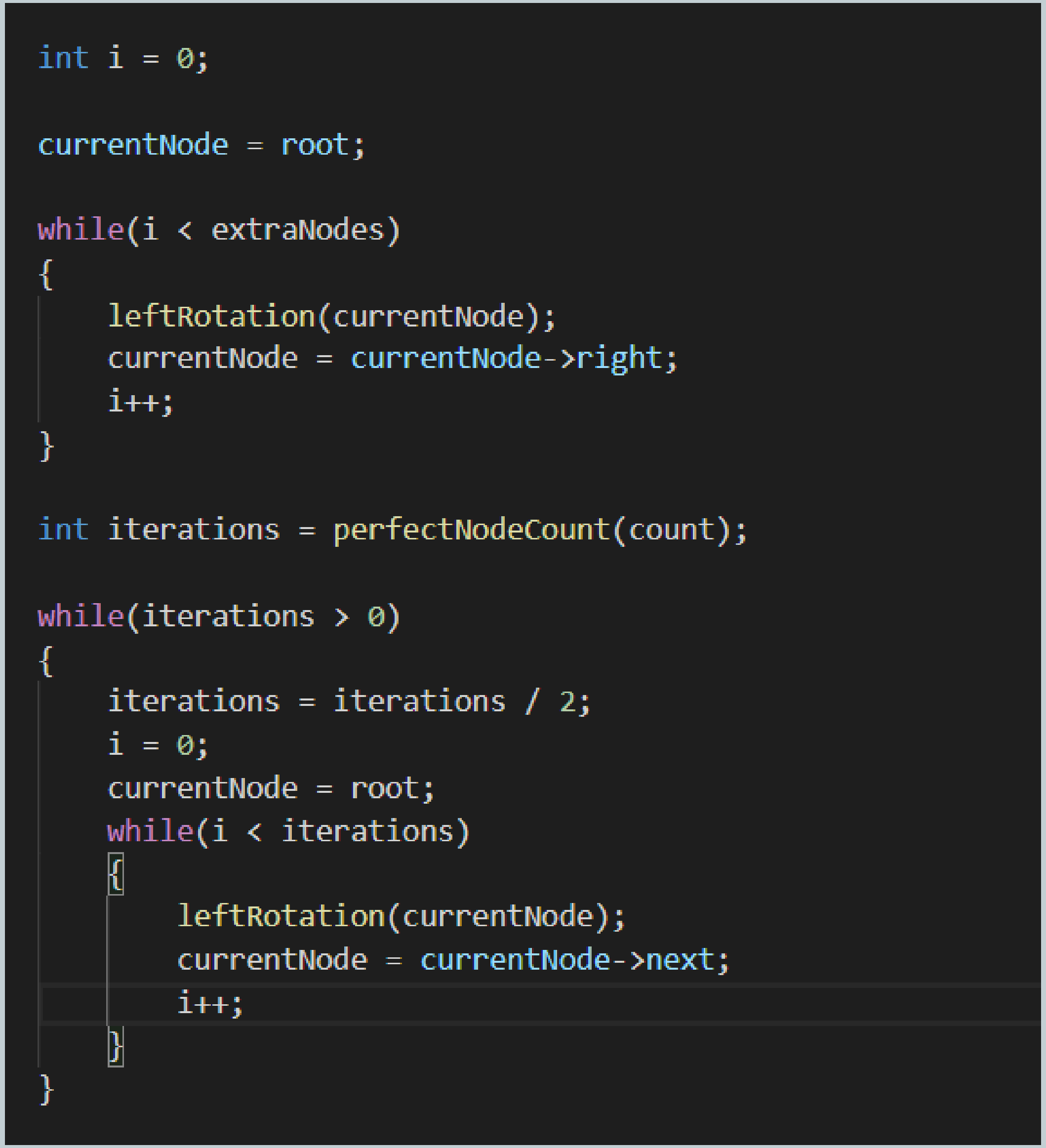
```
int i = 0;

currentNode = root;

while(i < extraNodes)
{
    leftRotation(currentNode);
    currentNode = currentNode->right;
    i++;
}

int iterations = perfectNodeCount(count);

while(iterations > 0)
{
    iterations = iterations / 2;
    i = 0;
    currentNode = root;
    while(i < iterations)
    {
        leftRotation(currentNode);
        currentNode = currentNode->next;
        i++;
    }
}
```

DSW ALGORITHM – BALANCE THE BACKBONE

Second rotation series



Then mark again



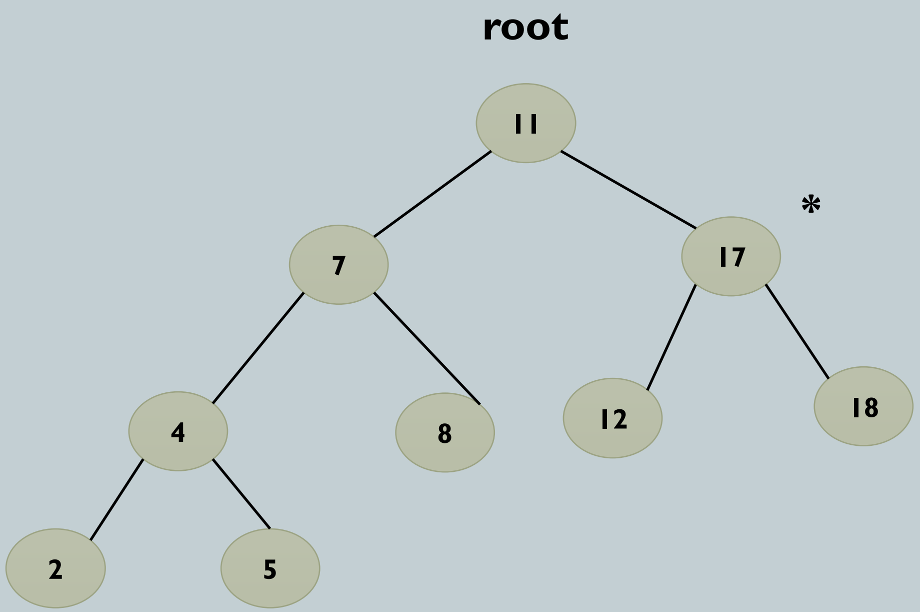
```
int i = 0;

currentNode = root;

while(i < extraNodes)
{
    leftRotation(currentNode);
    currentNode = currentNode->right;
    i++;
}

int iterations = perfectNodeCount(count);

while(iterations > 0)
{
    iterations = iterations / 2;
    i = 0;
    currentNode = root;
    while(i < iterations)
    {
        leftRotation(currentNode);
        currentNode = currentNode->next;
        i++;
    }
}
```

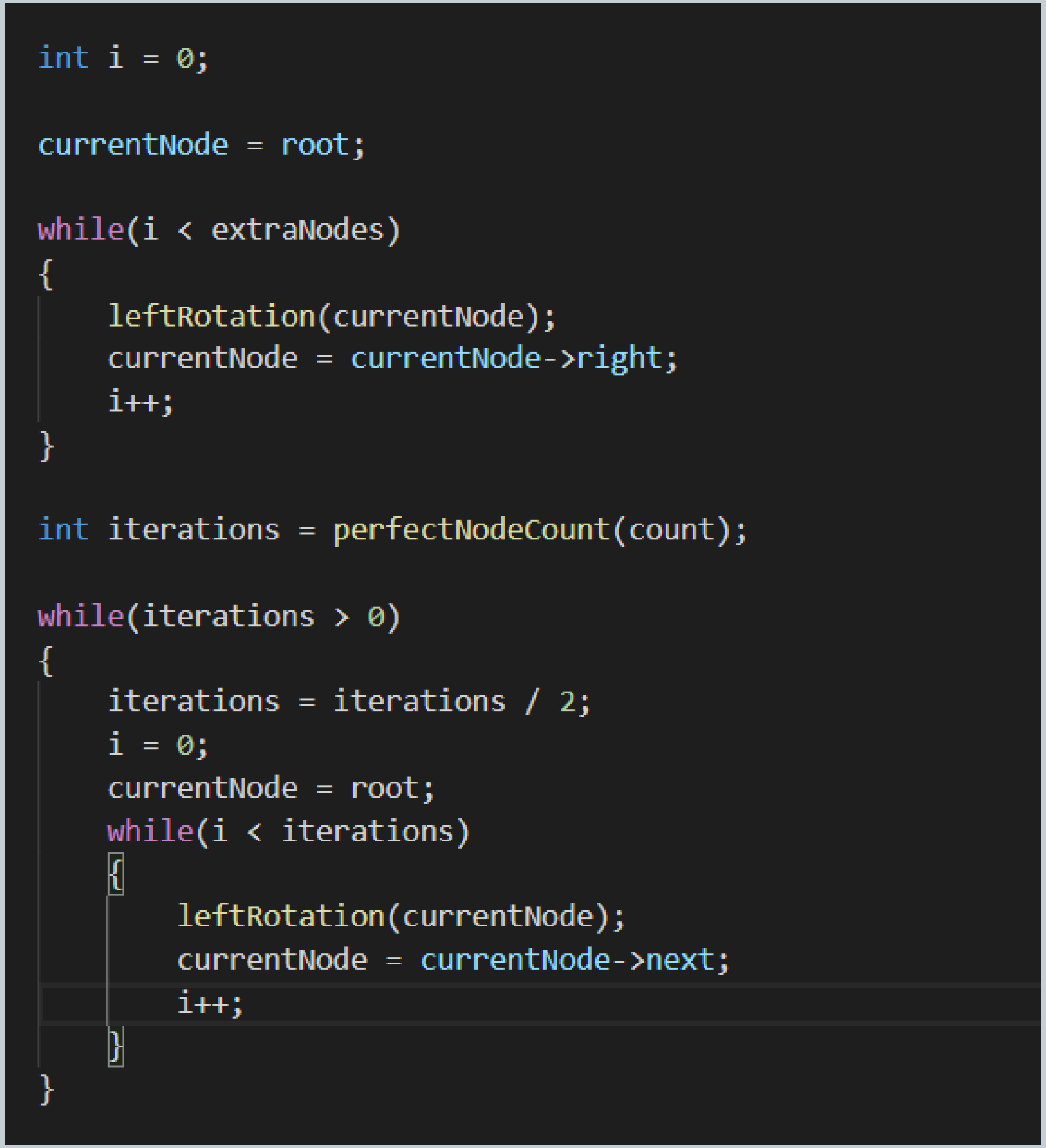
Rotations = 3 // 2 = 1

1 rotation

DSW ALGORITHM – BALANCE THE BACKBONE

Second rotation series





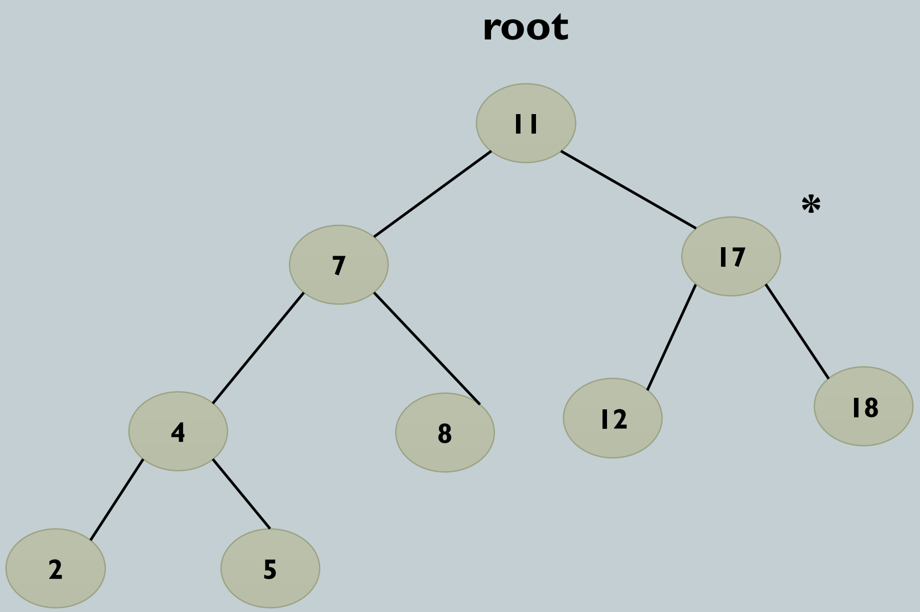
```
int i = 0;

currentNode = root;

while(i < extraNodes)
{
    leftRotation(currentNode);
    currentNode = currentNode->right;
    i++;
}

int iterations = perfectNodeCount(count);

while(iterations > 0)
{
    iterations = iterations / 2;
    i = 0;
    currentNode = root;
    while(i < iterations)
    {
        leftRotation(currentNode);
        currentNode = currentNode->next;
        i++;
    }
}
```

Rotations = 3 // 2 = 1

???????

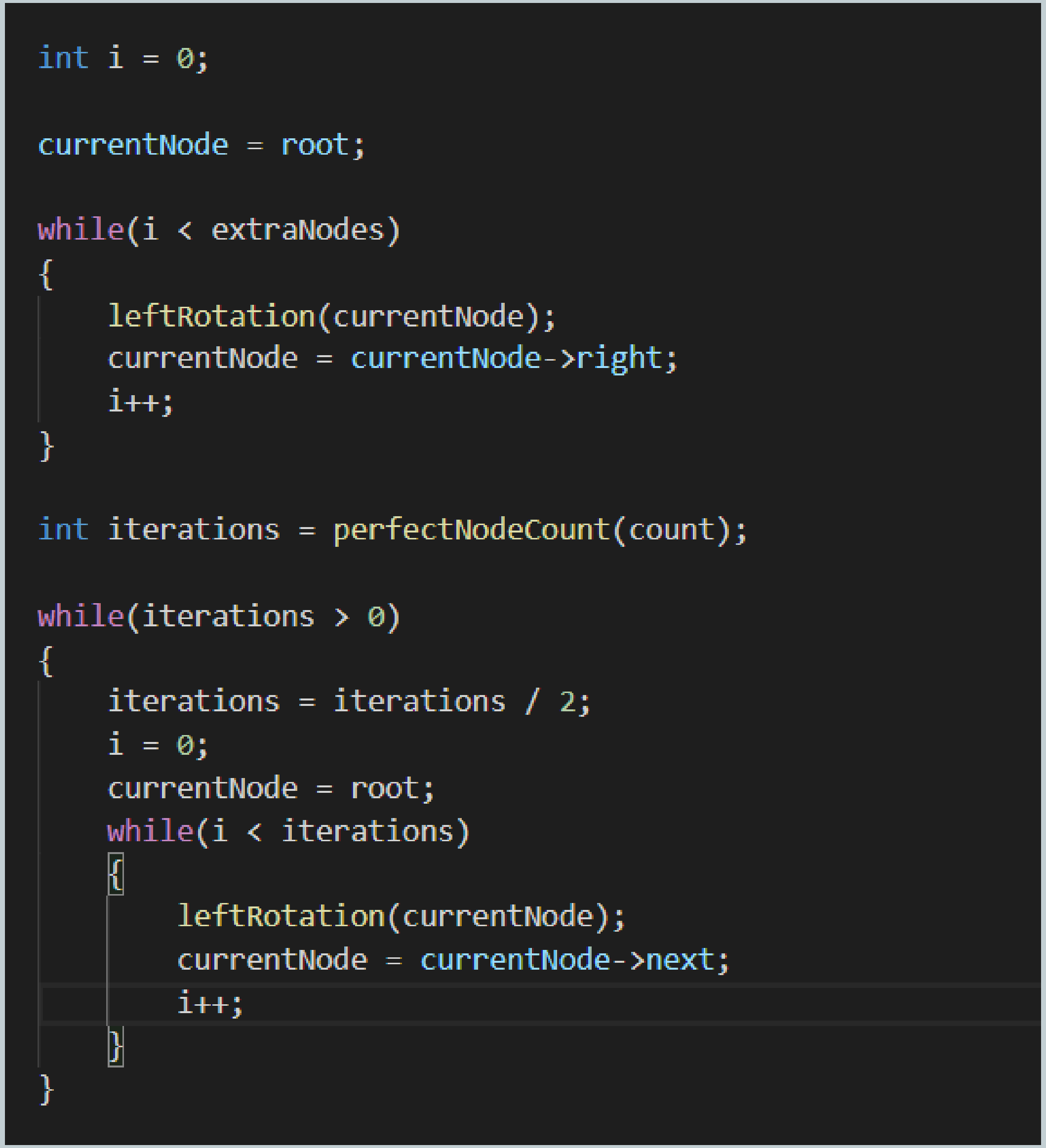
DSW ALGORITHM – BALANCE THE BACKBONE

Second rotation series



Skipped -> Out of the loop

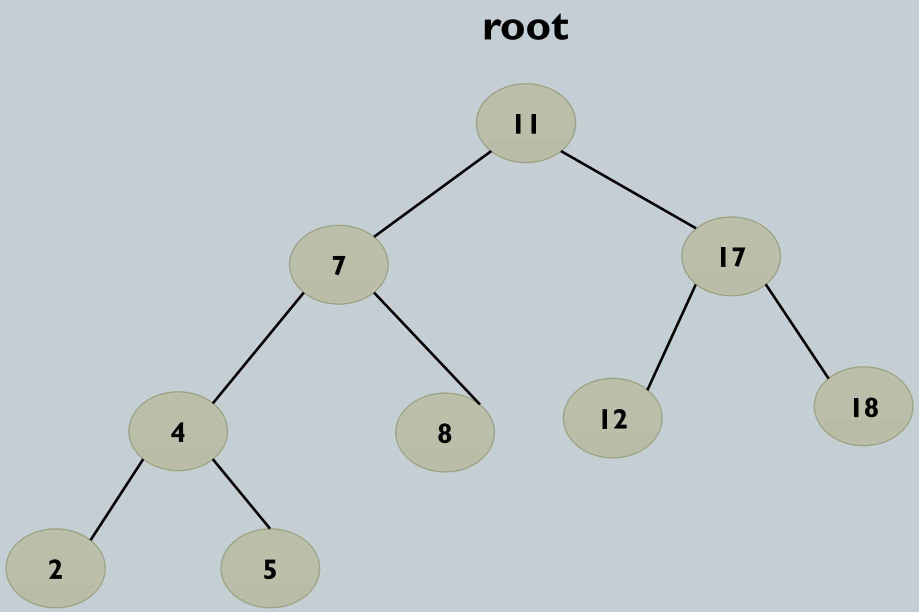
Because Rotations = 3 // 2 = 1



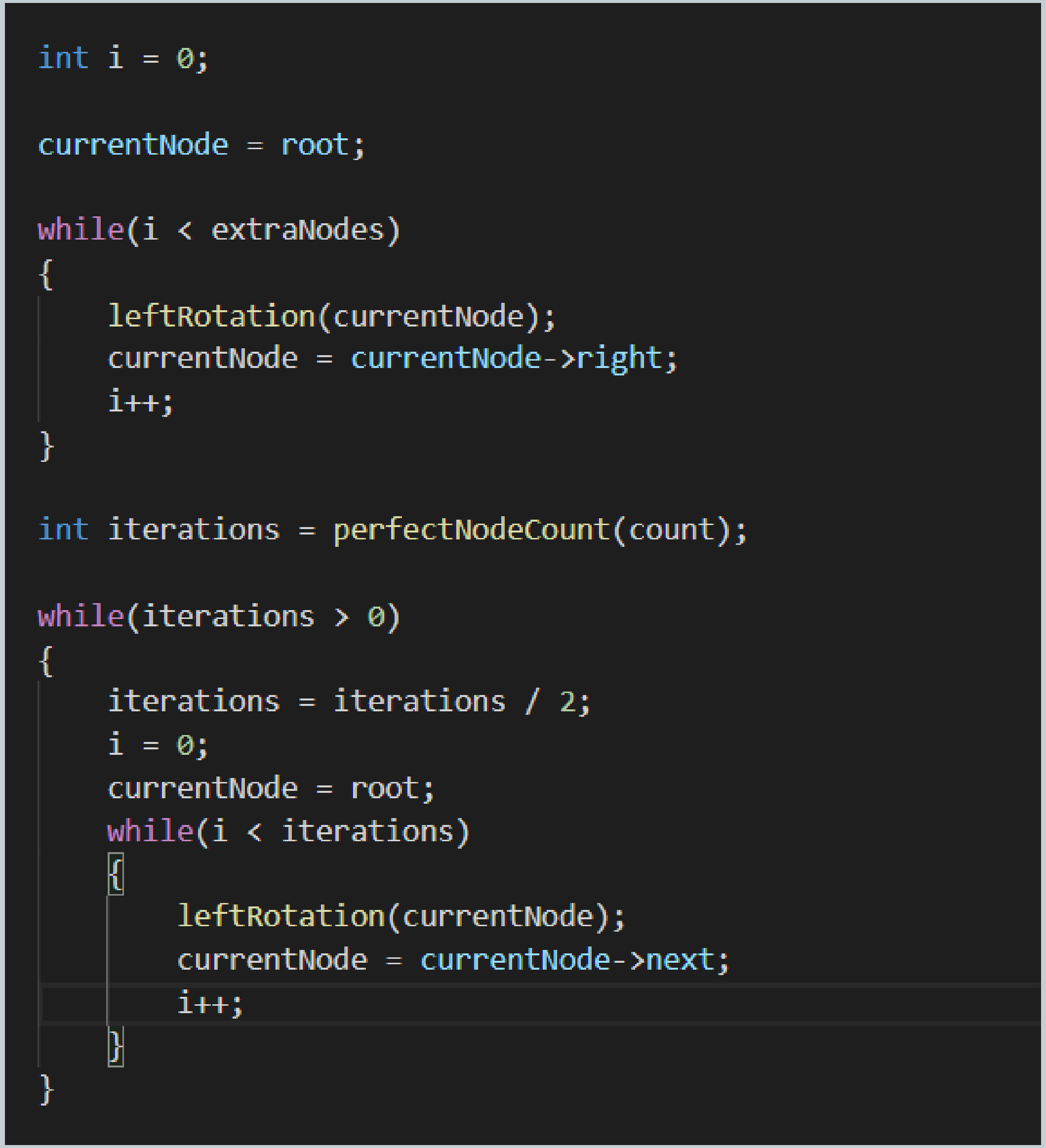
```
int i = 0;

currentNode = root;

while(i < extraNodes)
{
    leftRotation(currentNode);
    currentNode = currentNode->right;
    i++;
}

int iterations = perfectNodeCount(count);

while(iterations > 0)
{
    iterations = iterations / 2;
    i = 0;
    currentNode = root;
    while(i < iterations)
    {
        leftRotation(currentNode);
        currentNode = currentNode->next;
        i++;
    }
}
```

DSW ALGORITHM – BALANCE THE BACKBONE

Second rotation series



Final (DSW)



```
int i = 0;

currentNode = root;

while(i < extraNodes)
{
    leftRotation(currentNode);
    currentNode = currentNode->right;
    i++;
}

int iterations = perfectNodeCount(count);

while(iterations > 0)
{
    iterations = iterations / 2;
    i = 0;
    currentNode = root;
    while(i < iterations)
    {
        leftRotation(currentNode);
        currentNode = currentNode->next;
        i++;
    }
}
```

DSW ALGORITHM – DSW VERDICT

Now our tree is balanced, following global tree balancing.

AVL TREES

AVL TREES

- AVL tree is a self-balancing binary search tree invented by G.M. Adelson-Velsky and E.M. Landis in 1962. The tree is named AVL in honour of its inventors. In an AVL tree, the heights of the two sub-trees of a node may differ by at most one. Due to this property, the AVL tree is also known as a height-balanced tree. The key advantage of using an AVL tree is that it takes O(log n) time to perform search, insert, and delete operations in an average case as well as the worst case because the height of the tree is limited to O(log n).

AVL TREE BALANCE FACTOR

- The structure of an AVL tree is the same as that of a binary search tree but with a little difference. In its structure, it stores an additional variable called the BalanceFactor. Thus, every node has a balance factor associated with it. The balance factor of a node is calculated by subtracting the height of its right sub-tree from the height of its left 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.

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

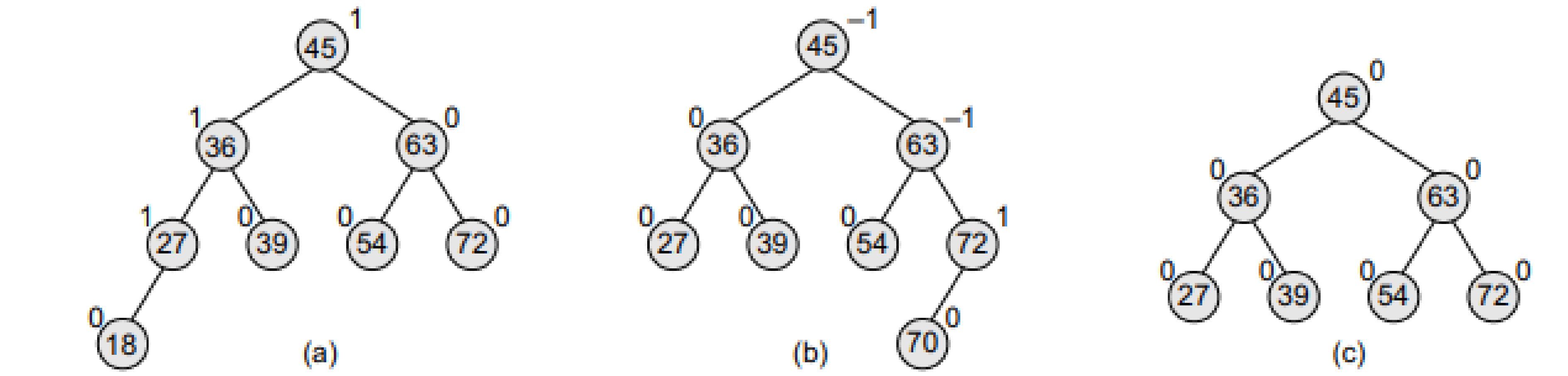


Figure 10.35 (a) Left-heavy AVL tree, (b) right-heavy tree, (c) balanced tree

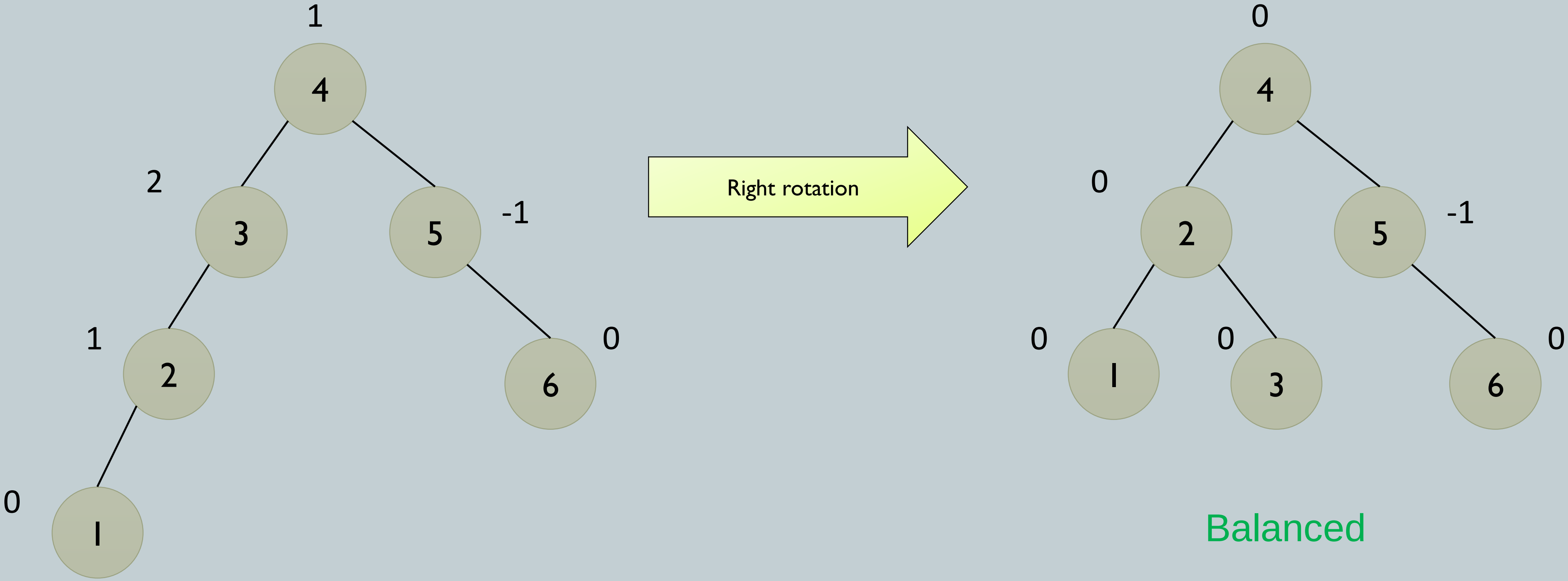
AVL TREE INSERTION

- Insertion in an AVL tree is also done in the same way as it is done in a binary search tree. In the AVL tree, the new node is always inserted as the leaf node. But the step of insertion is usually followed by an additional step of rotation. Rotation is done to restore the balance of the tree. However, if insertion of the new node does not disturb the balance factor, that is, if the balance factor of every node is still –1, 0, or 1, then rotations are not required.

AVL TREE ROTATIONS (QUICK LOOK)

- There are four types of rotations depends on the AVL trees imbalance.

- I. Left- left -> Do right rotation



Imbalance at 3

Balanced

AVL TREE ROTATIONS (GENERAL)

- 1. Left- left -> Do right rotation

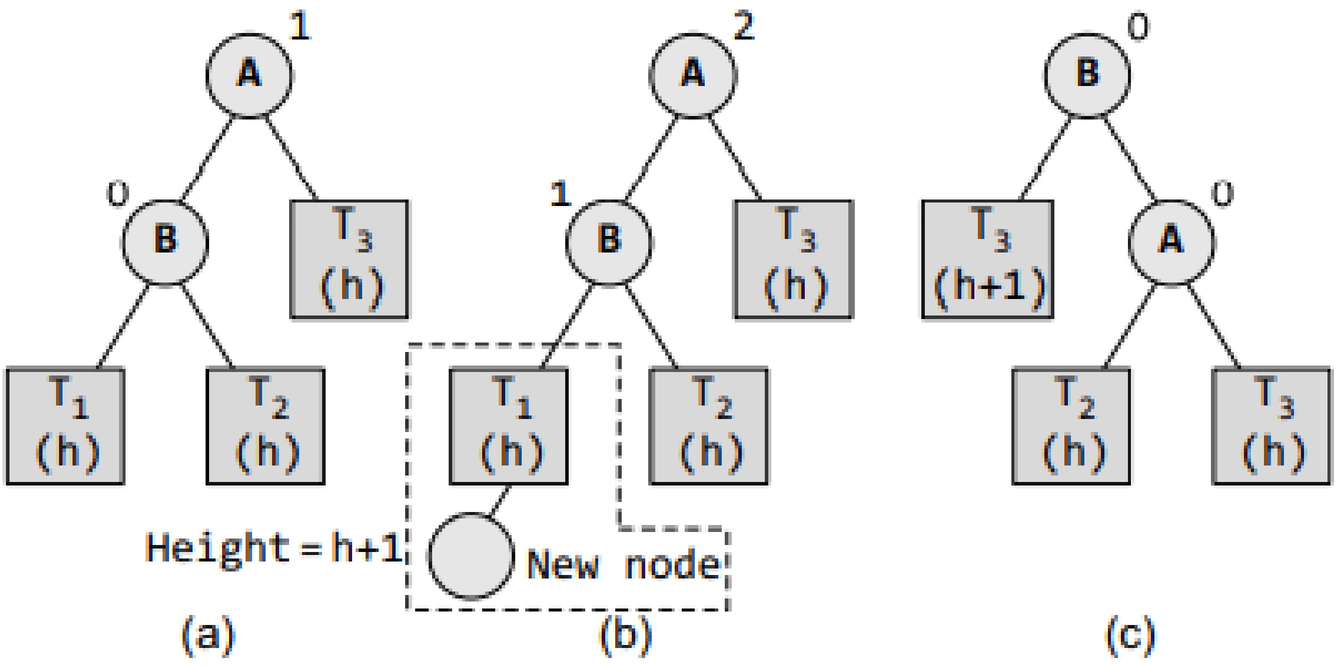
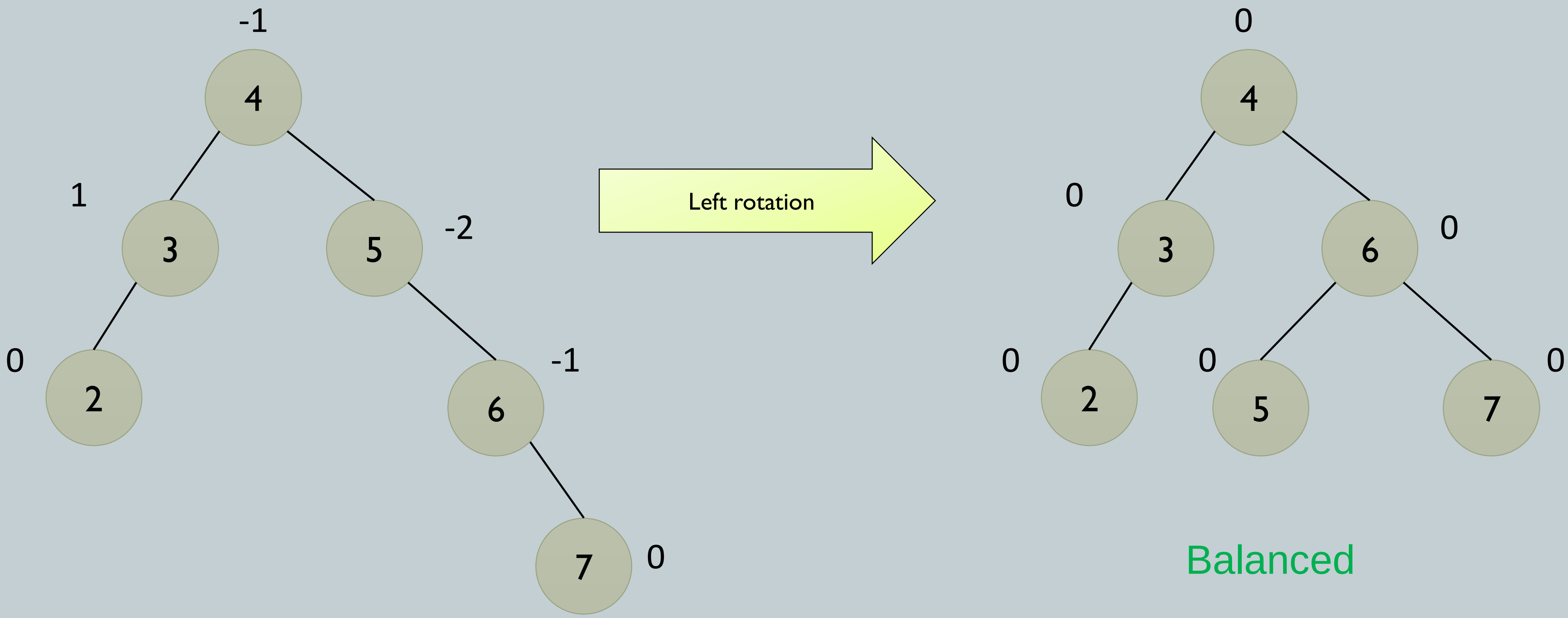


Figure 10.40 LL rotation in an AVL tree

AVL TREE ROTATIONS (QUICK LOOK)

- There are four types of rotations depends on the AVL trees imbalance.

- 2. right right -> Do left rotation



Imbalance at 5

Balanced

AVL TREE ROTATIONS (GENERAL)

- 2 right right -> Do left rotation

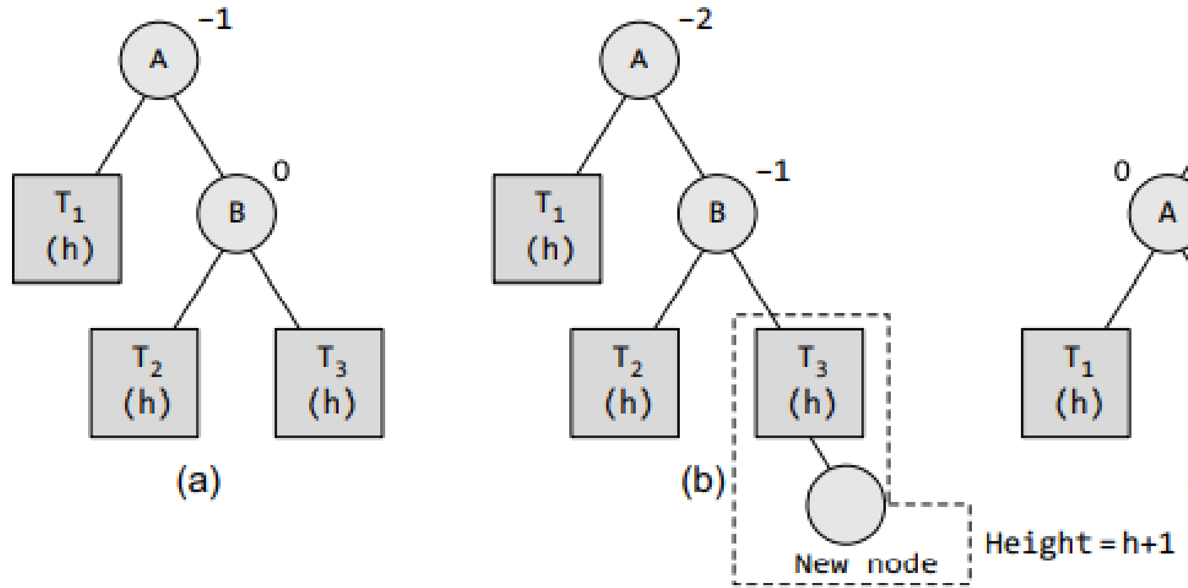
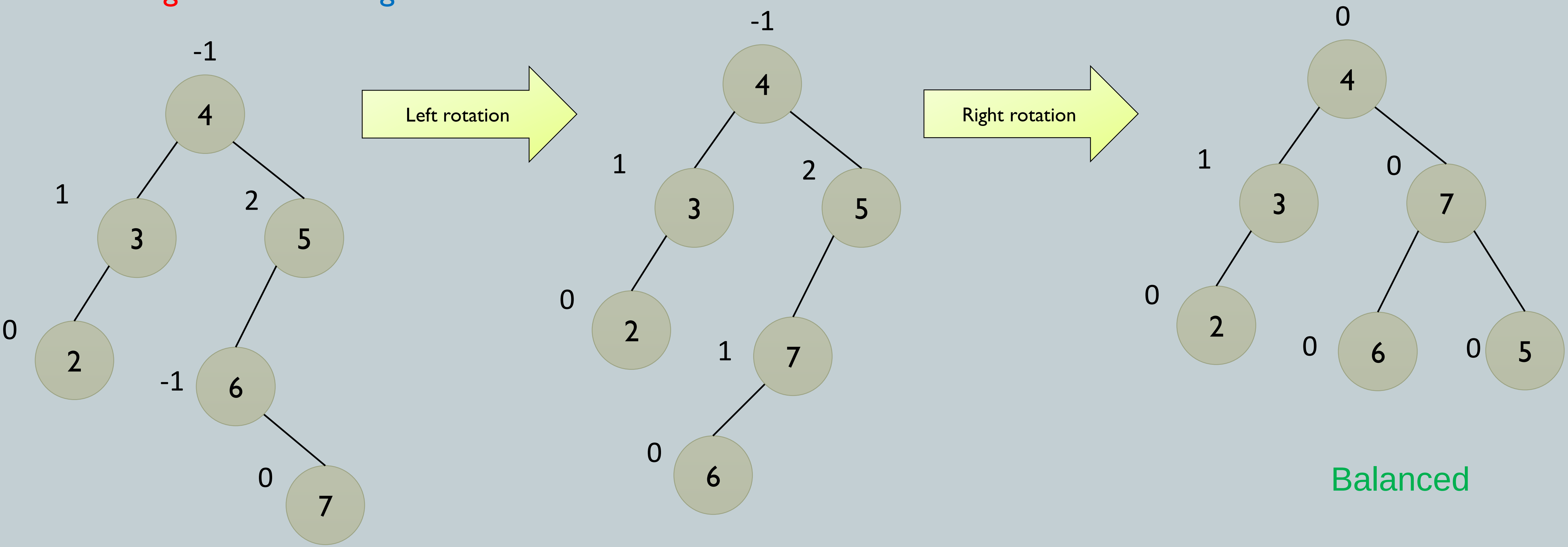


Figure 10.42 RR rotation in an AVL tree

AVL TREE ROTATIONS (QUICK LOOK)

- There are four types of rotations depends on the AVL trees imbalance.

- 3. left right -> Do left right rotation



Imbalance at 5

AVL TREE ROTATIONS (GENERAL)

- 3 left right -> Do left right rotation

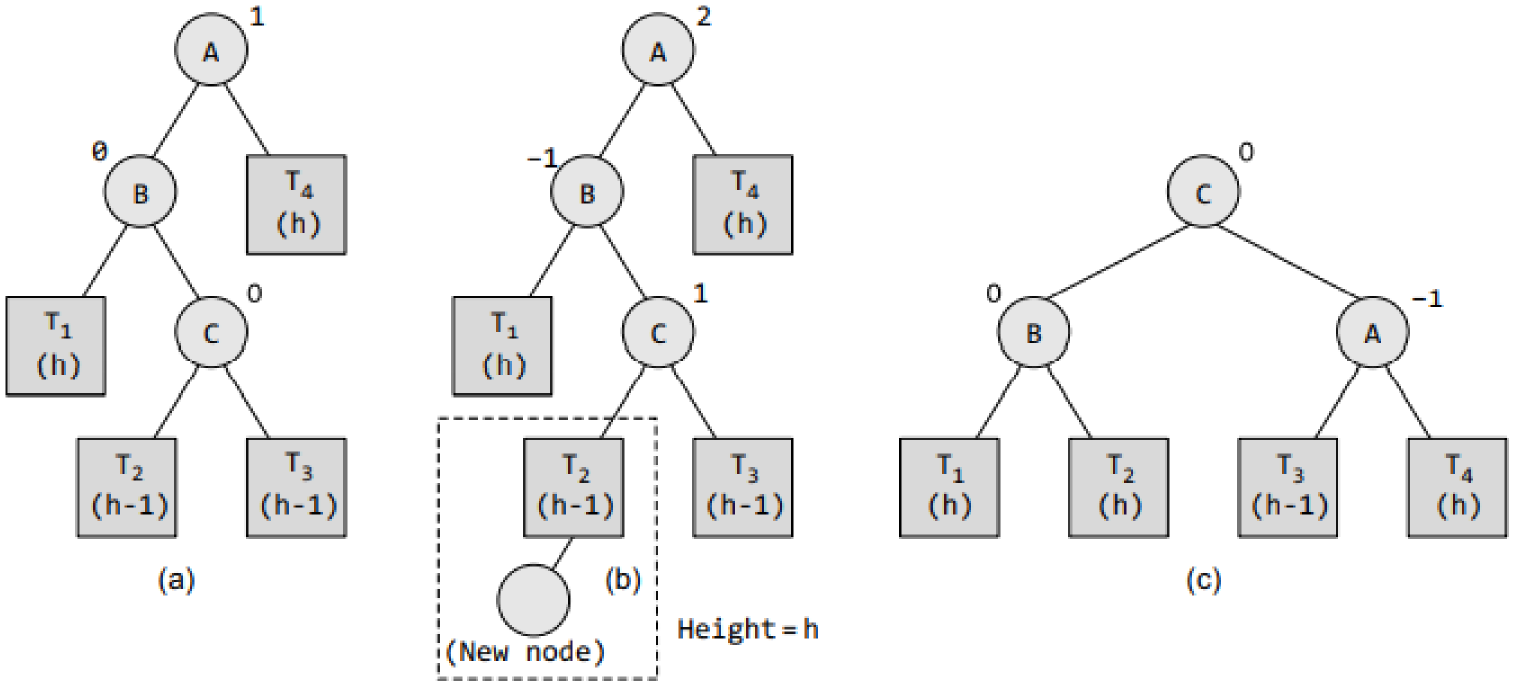
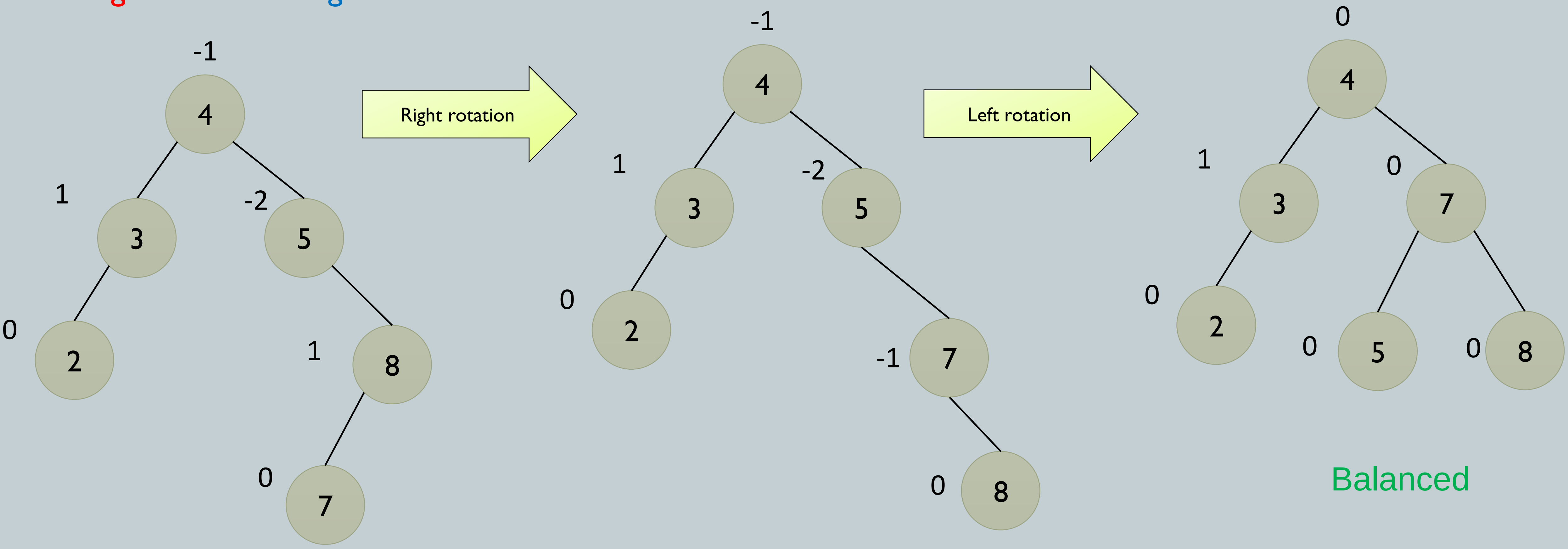


Figure 10.44 LR rotation in an AVL tree

AVL TREE ROTATIONS (QUICK LOOK)

- There are four types of rotations depends on the AVL trees imbalance.

- 4. right left -> Do right left rotation



Imbalance at 5

Balanced

AVL TREE ROTATIONS (GENERAL)

- 4. right left -> Do right left rotation

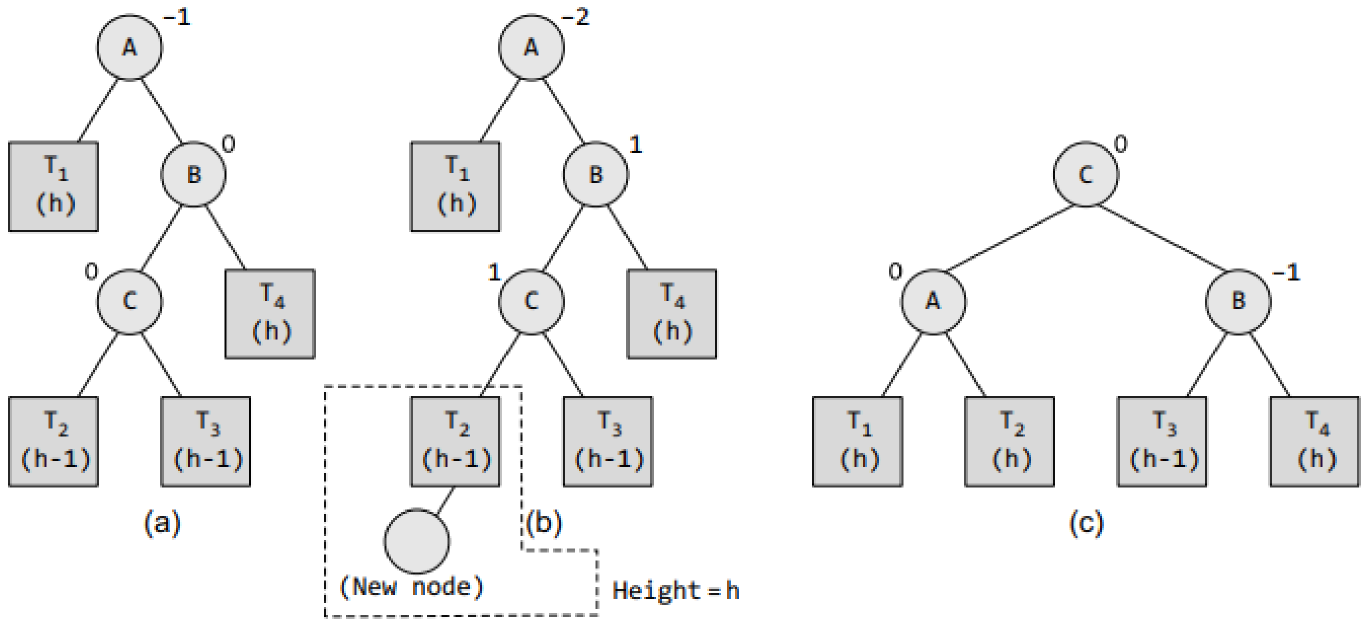
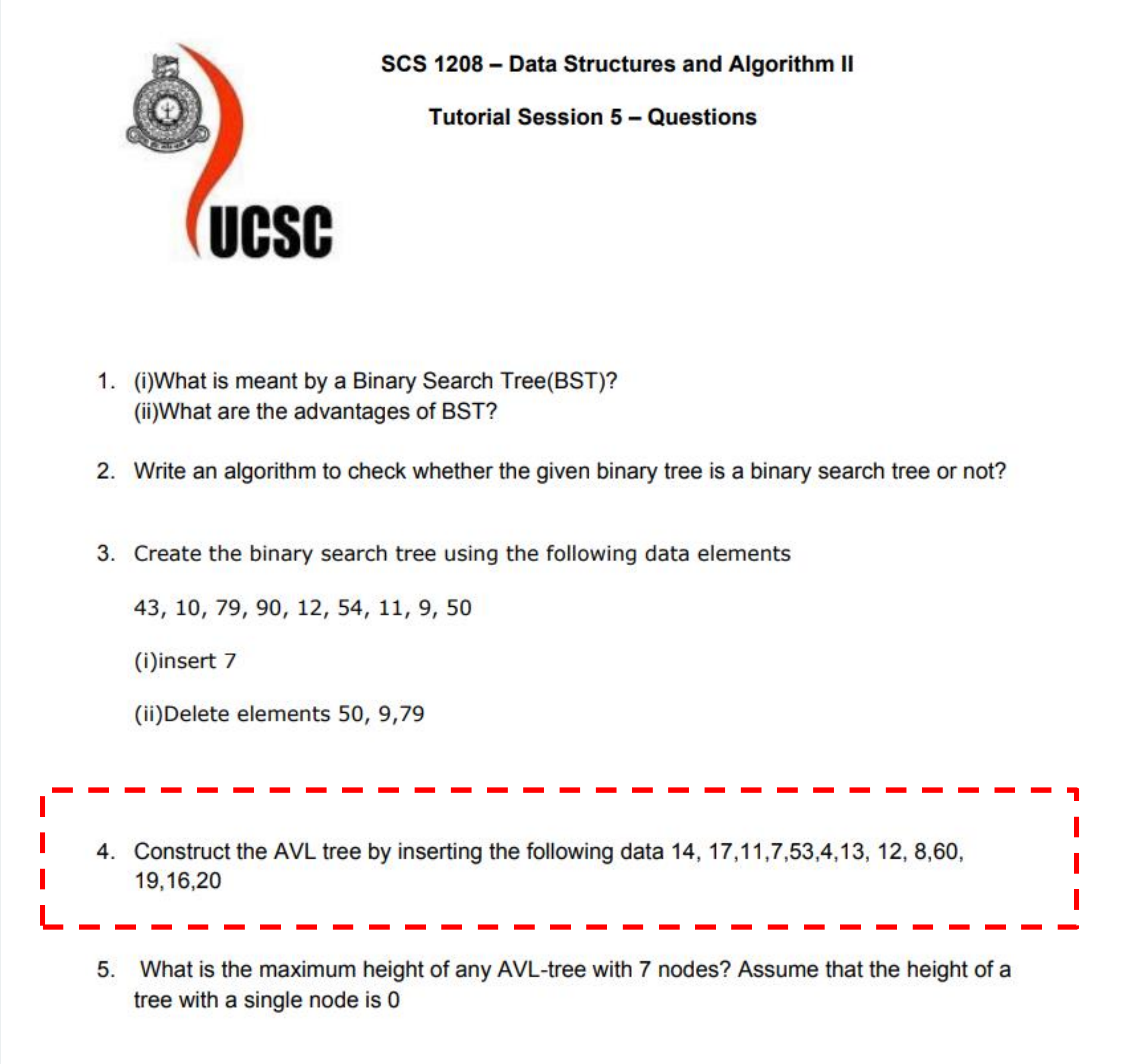


Figure 10.46 RL rotation in an AVL tree

AVL TREE INSERTION



UCSC

SCS 1208 – Data Structures and Algorithm II

Tutorial Session 5 – Questions

1. (i)What is meant by a Binary Search Tree(BST)?
   (ii)What are the advantages of BST?

2. Write an algorithm to check whether the given binary tree is a binary search tree or not?

3. Create the binary search tree using the following data elements

   43, 10, 79, 90, 12, 54, 11, 9, 50

   (i)insert 7

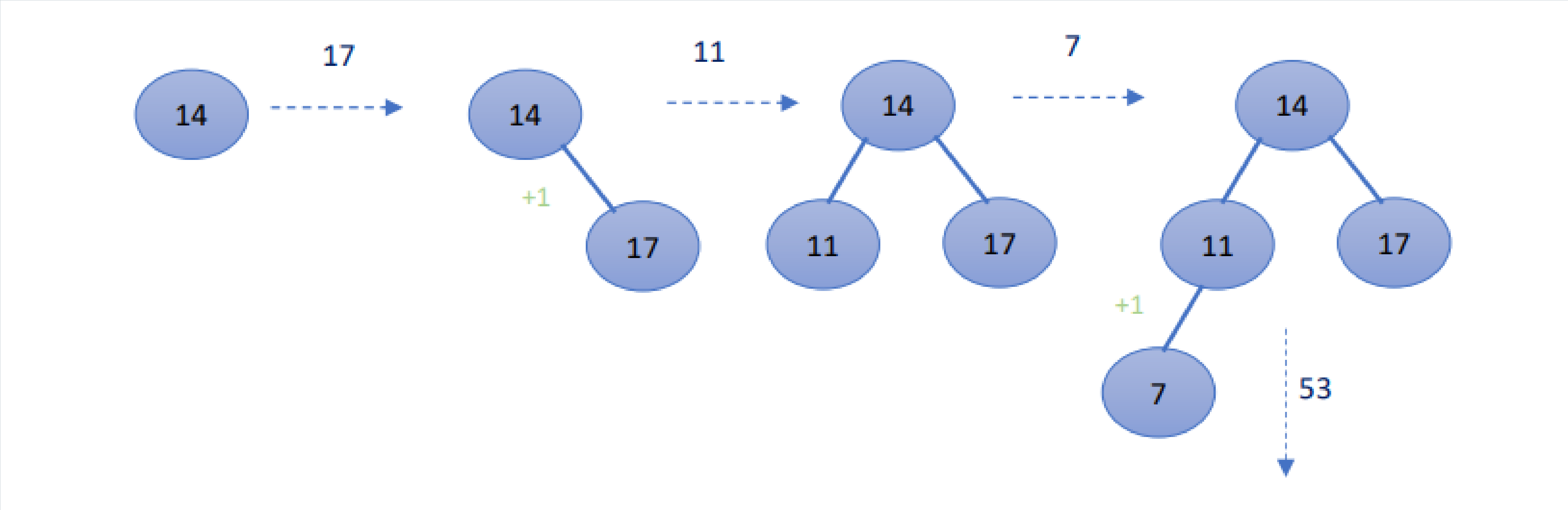
   (ii)Delete elements 50, 9,79

4. Construct the AVL tree by inserting the following data 14, 17,11,7,53,4,13, 12, 8,60, 19,16,20

5. What is the maximum height of any AVL-tree with 7 nodes? Assume that the height of a tree with a single node is 0

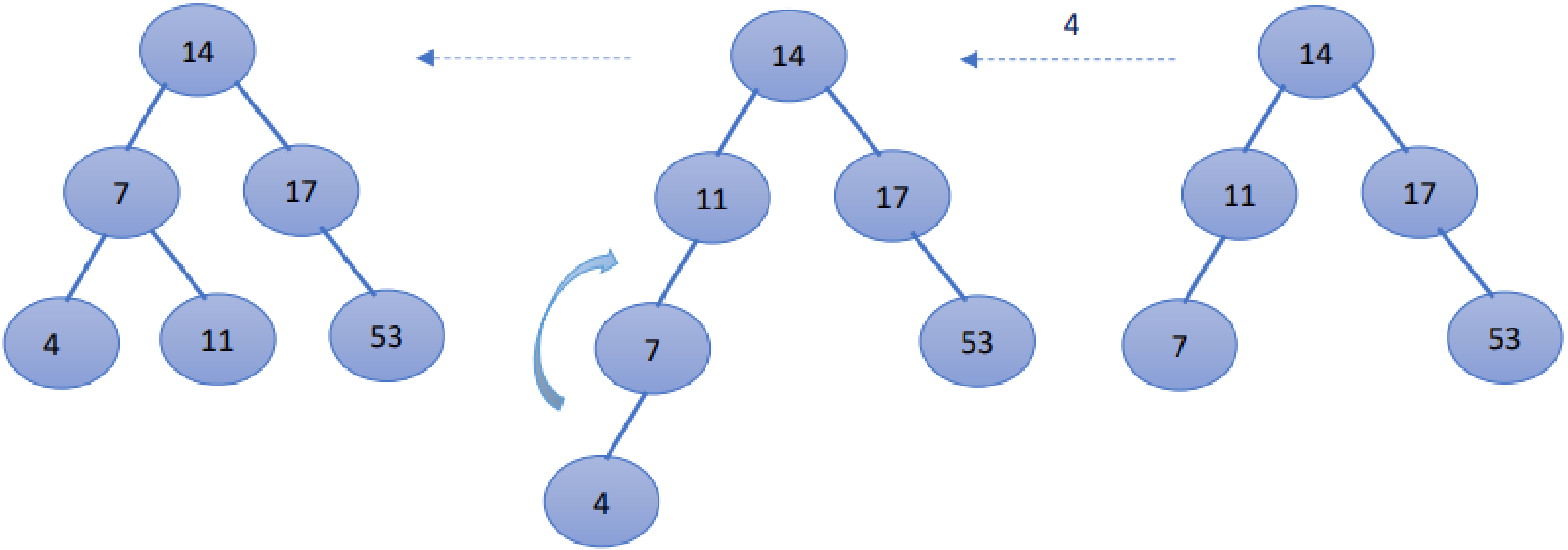
AVL TREE INSERTION

4. Construct the AVL tree by inserting the following data 14, 17,11,7,53,4,13, 12, 8,60, 19,16,20



AVL TREE INSERTION

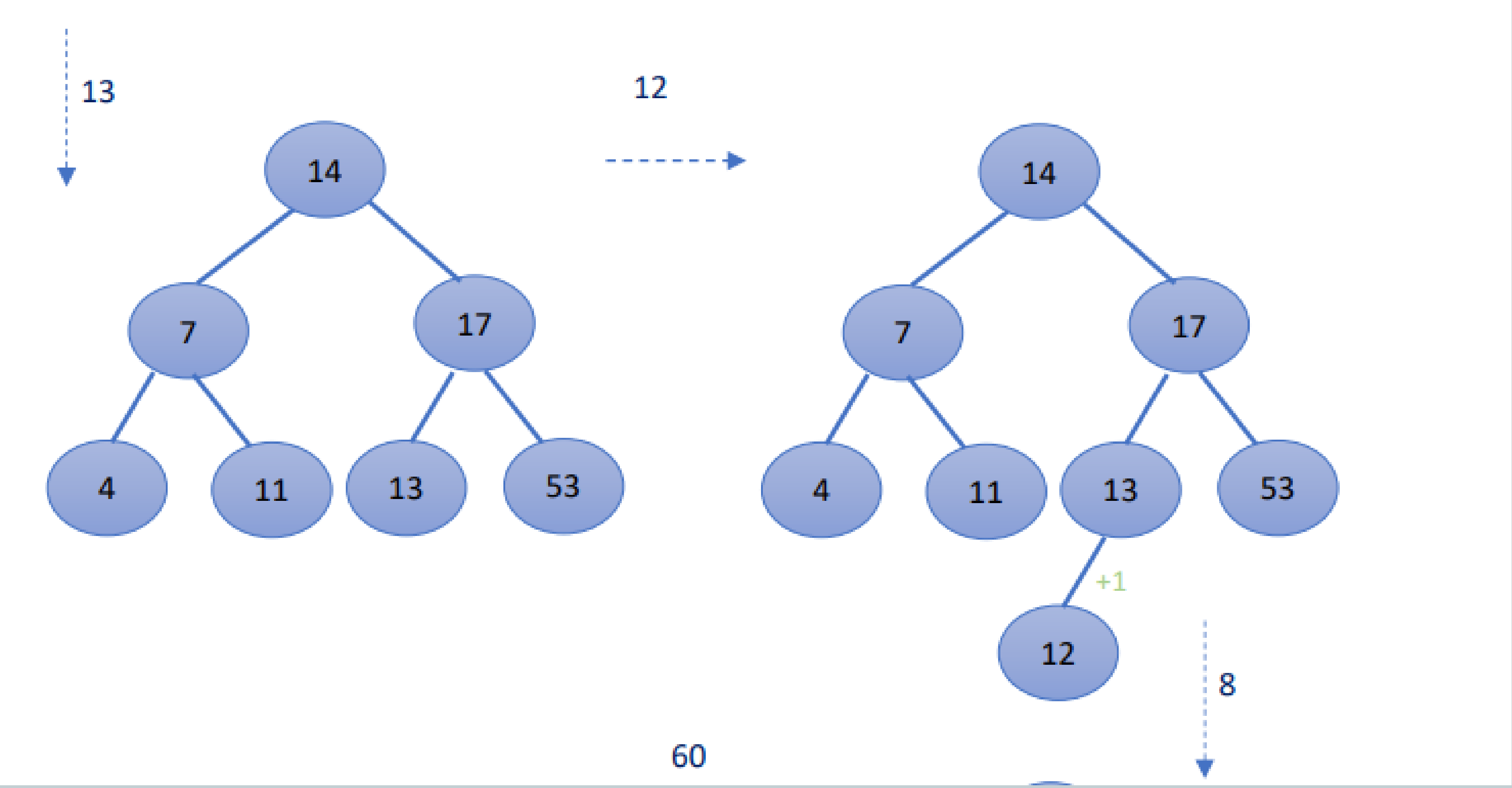
4. Construct the AVL tree by inserting the following data 14, 17,11,7,53,4,13, 12, 8,60, 19,16,20



Imbalance occurred at node 1. Nodes are in a straight line. Therefore single rotation needs.

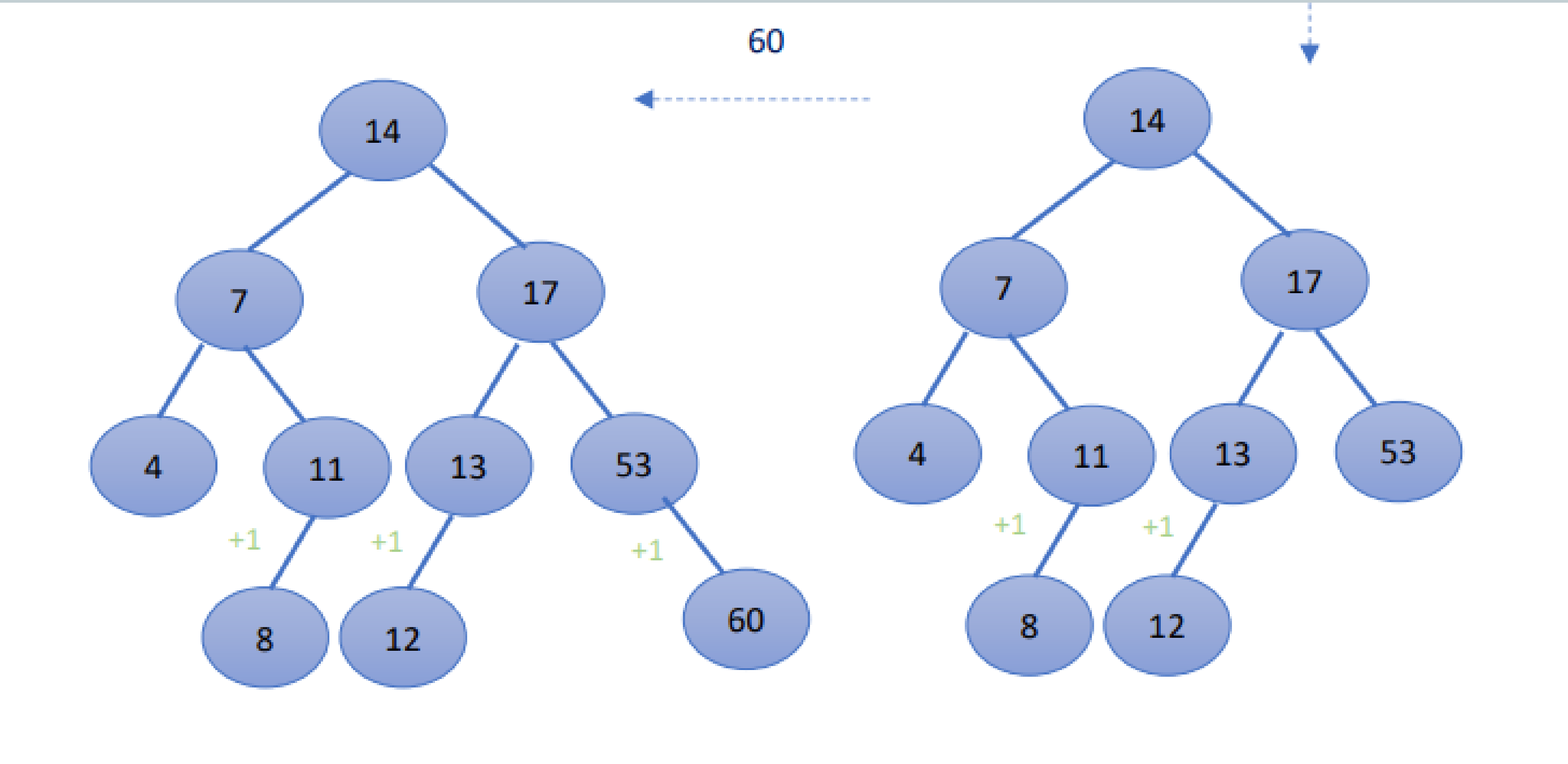
AVL TREE INSERTION

4. Construct the AVL tree by inserting the following data 14, 17,11,7,53,4,13, 12, 8,60, 19,16,20



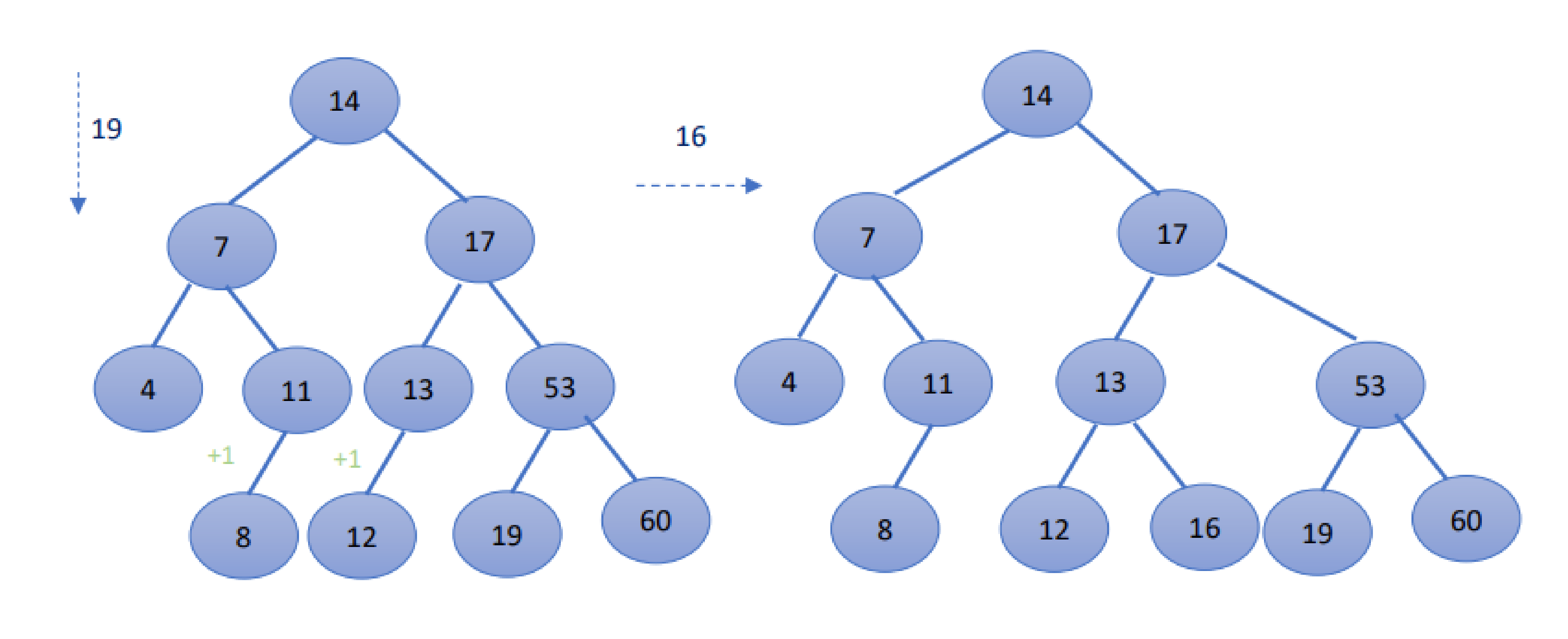
AVL TREE INSERTION

4. Construct the AVL tree by inserting the following data 14, 17,11,7,53,4,13, 12, 8,60, 19,16,20



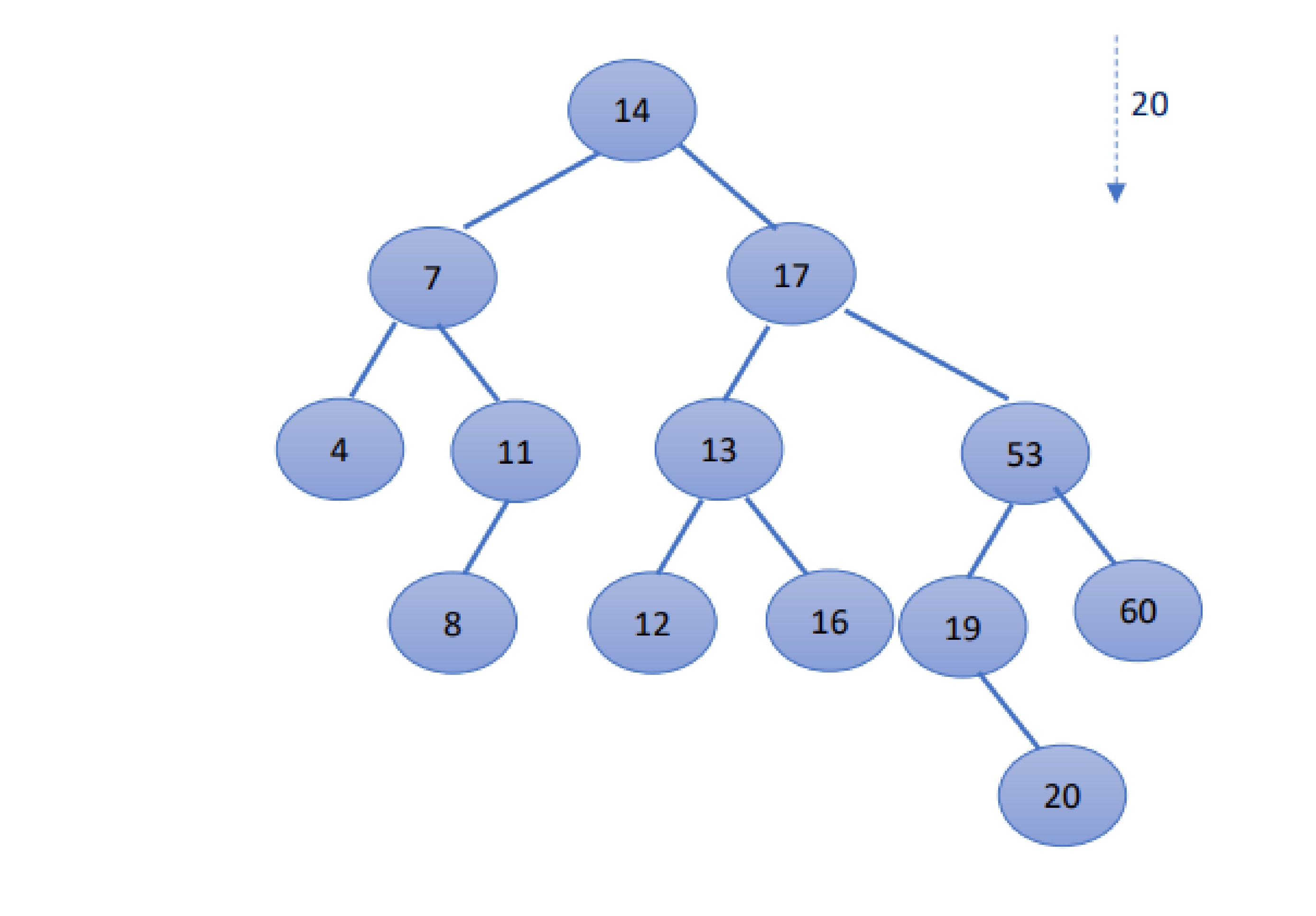
AVL TREE INSERTION

4. Construct the AVL tree by inserting the following data 14, 17,11,7,53,4,13, 12, 8,60, 19,16,20



AVL TREE INSERTION

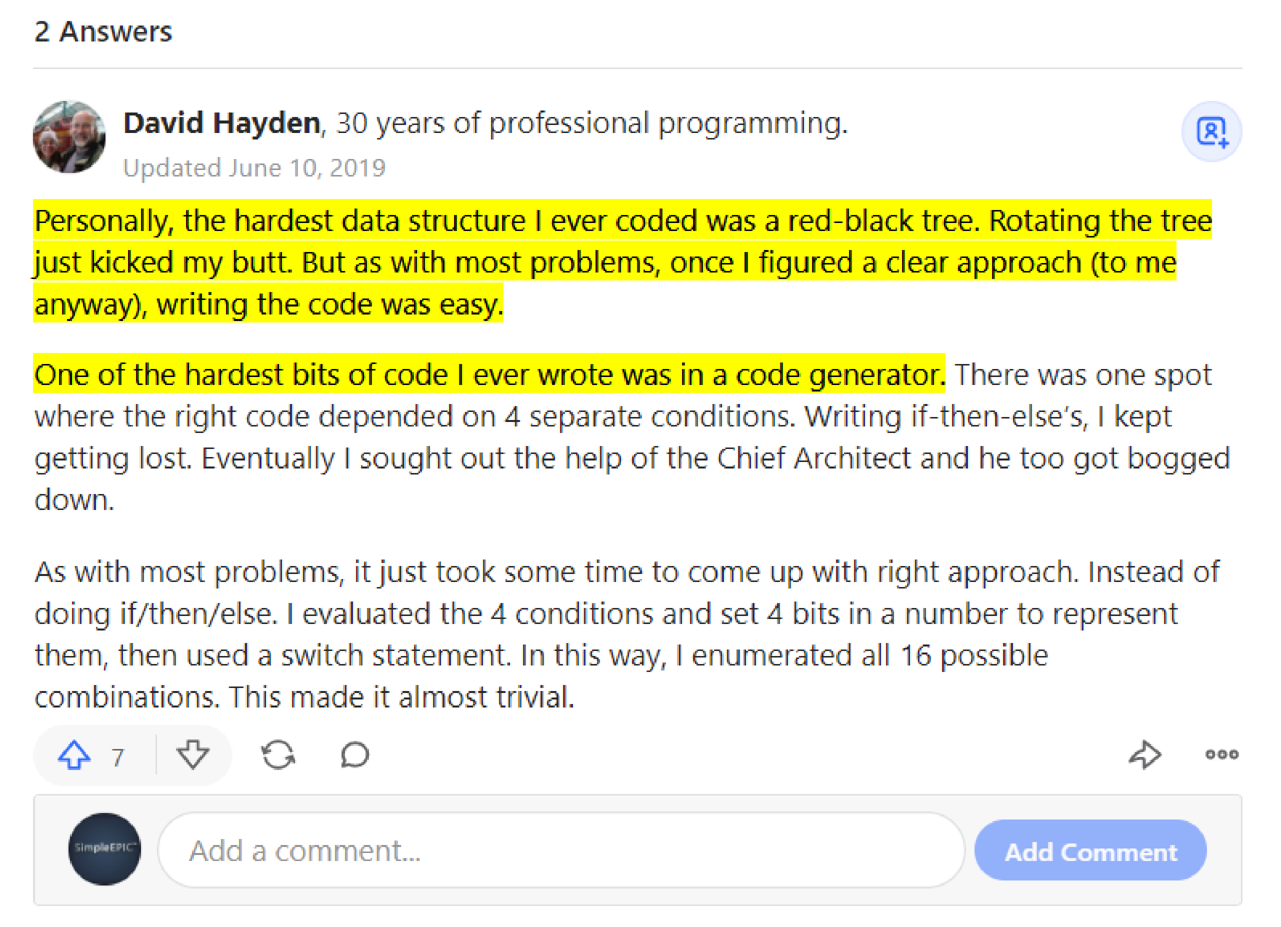
4. Construct the AVL tree by inserting the following data 14, 17,11,7,53,4,13, 12, 8,60, 19,16,20



RED BLACK TREES

Complex data structure. So be awake.

RED BLACK TREES



2 Answers

David Hayden, 30 years of professional programming.

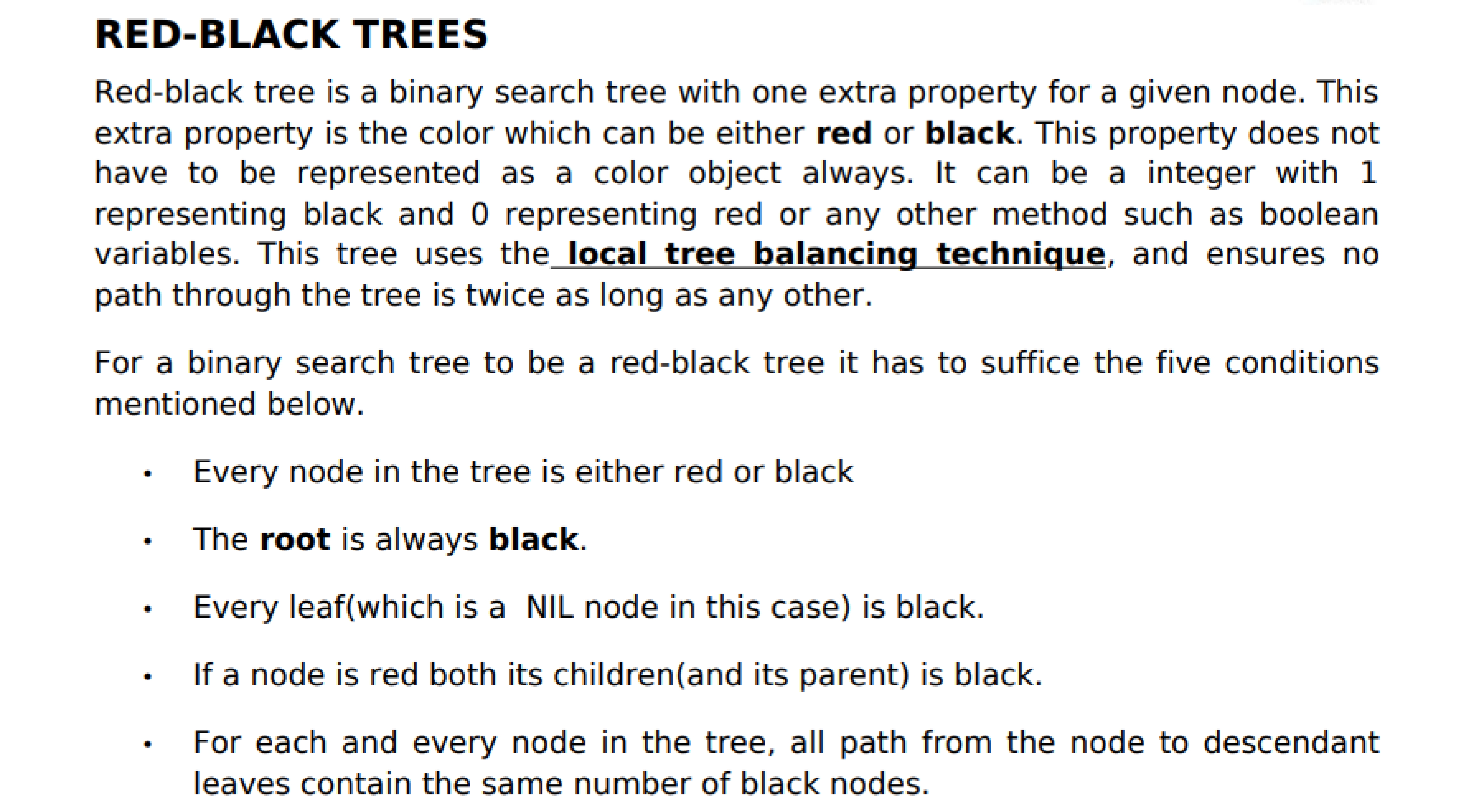
Updated June 10, 2019

Personally, the hardest data structure I ever coded was a red-black tree. Rotating the tree just kicked my butt. But as with most problems, once I figured a clear approach (to me anyway), writing the code was easy.

One of the hardest bits of code I ever wrote was in a code generator. There was one spot where the right code depended on 4 separate conditions. Writing if-then-else's, I kept getting lost. Eventually I sought out the help of the Chief Architect and he too got bogged down.

As with most problems, it just took some time to come up with right approach. Instead of doing if/then/else. I evaluated the 4 conditions and set 4 bits in a number to represent them, then used a switch statement. In this way, I enumerated all 16 possible combinations. This made it almost trivial.

RED BLACK TREES



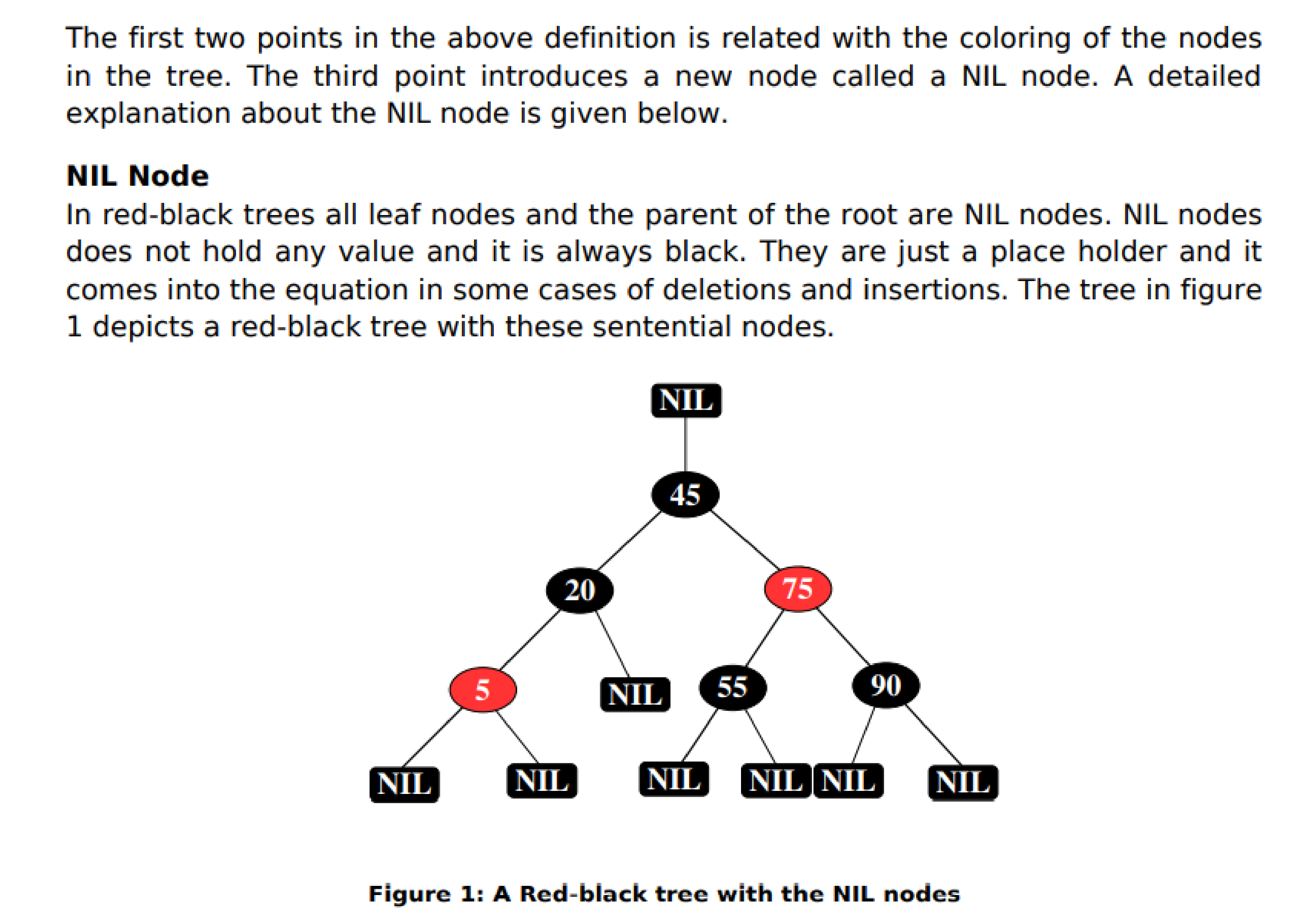
RED-BLACK TREES

Red-black tree is a binary search tree with one extra property for a given node. This extra property is the color which can be either **red** or **black**. This property does not have to be represented as a color object always. It can be a integer with 1 representing black and 0 representing red or any other method such as boolean variables. This tree uses the **local tree balancing technique**, and ensures no path through the tree is twice as long as any other.

For a binary search tree to be a red-black tree it has to suffice the five conditions mentioned below.

- Every node in the tree is either red or black
- The **root** is always **black**.
- Every leaf(which is a NIL node in this case) is black.
- If a node is red both its children(and its parent) is black.
- For each and every node in the tree, all path from the node to descendant leaves contain the same number of black nodes.

RED BLACK TREES



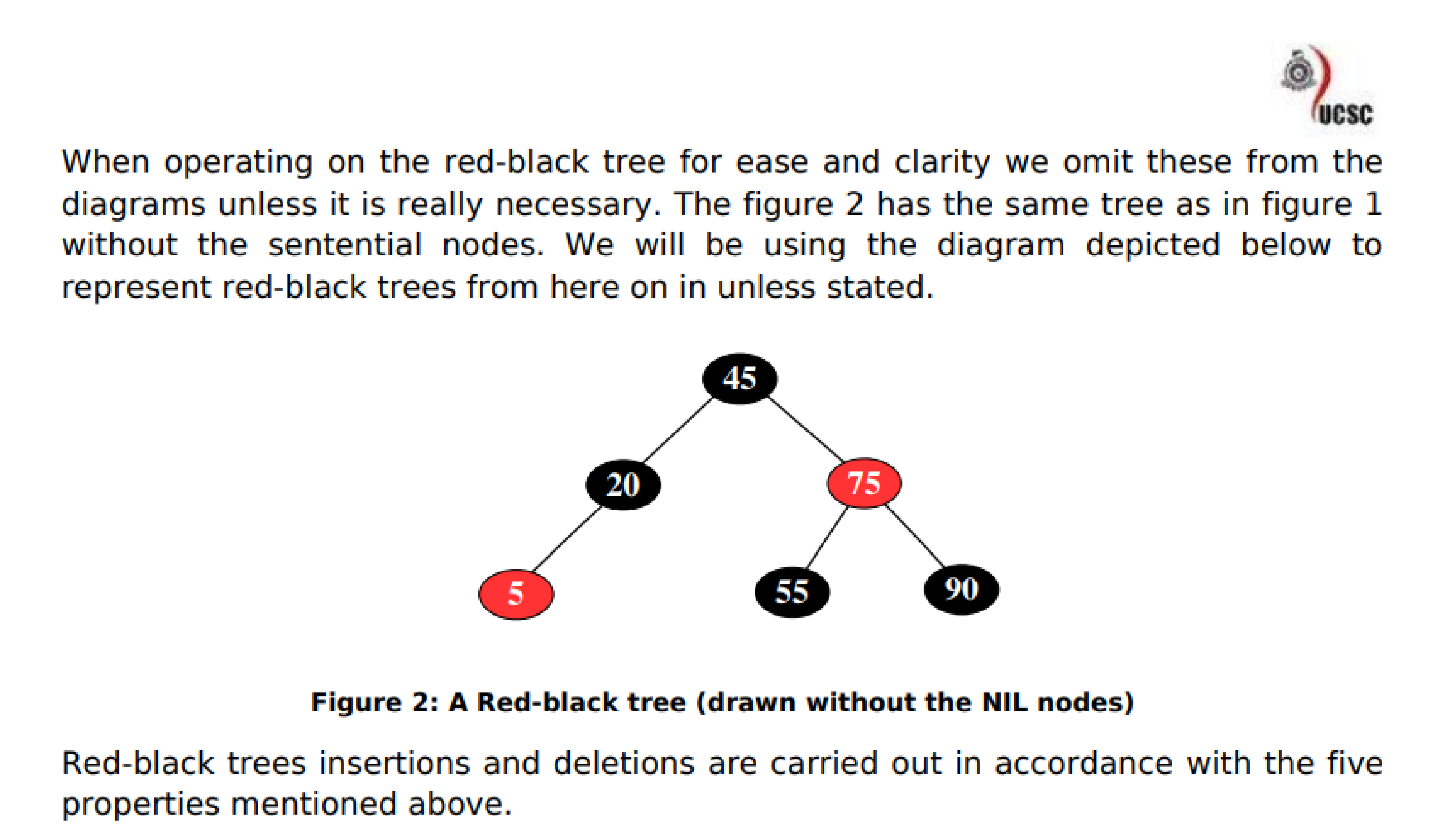
The first two points in the above definition is related with the coloring of the nodes in the tree. The third point introduces a new node called a NIL node. A detailed explanation about the NIL node is given below.

NIL Node

In red-black trees all leaf nodes and the parent of the root are NIL nodes. NIL nodes does not hold any value and it is always black. They are just a place holder and it comes into the equation in some cases of deletions and insertions. The tree in figure 1 depicts a red-black tree with these sentential nodes.

Figure 1: A Red-black tree with the NIL nodes

RED BLACK TREES



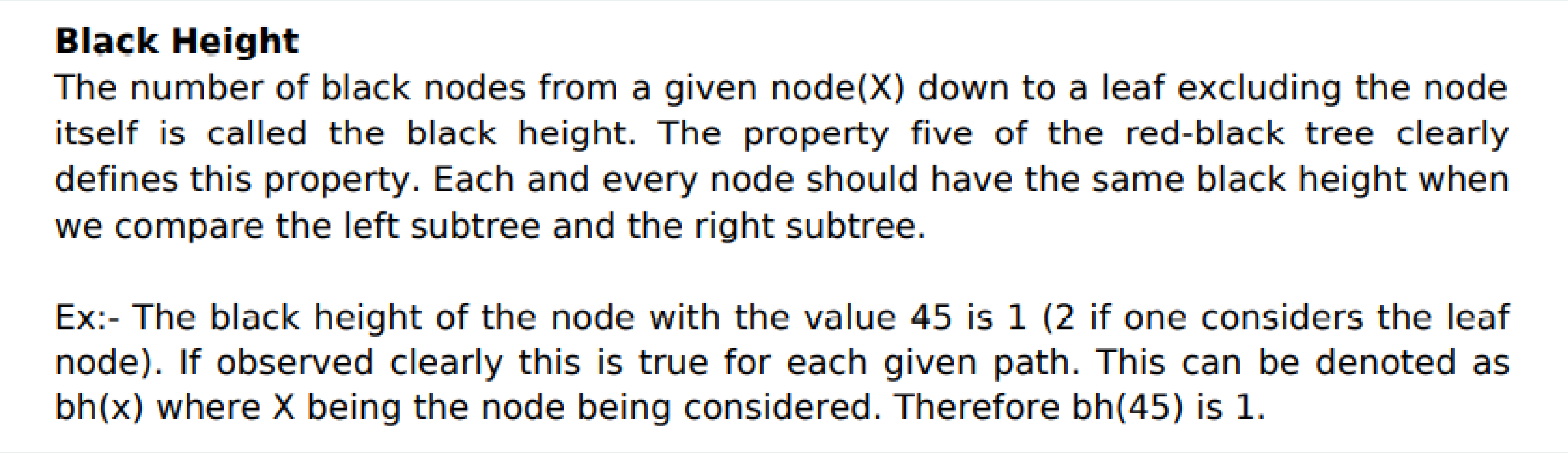
When operating on the red-black tree for ease and clarity we omit these from the diagrams unless it is really necessary. The figure 2 has the same tree as in figure 1 without the sentential nodes. We will be using the diagram depicted below to represent red-black trees from here on in unless stated.

45
20
75
5
55
90

Figure 2: A Red-black tree (drawn without the NIL nodes)

Red-black trees insertions and deletions are carried out in accordance with the five properties mentioned above.

RED BLACK TREES

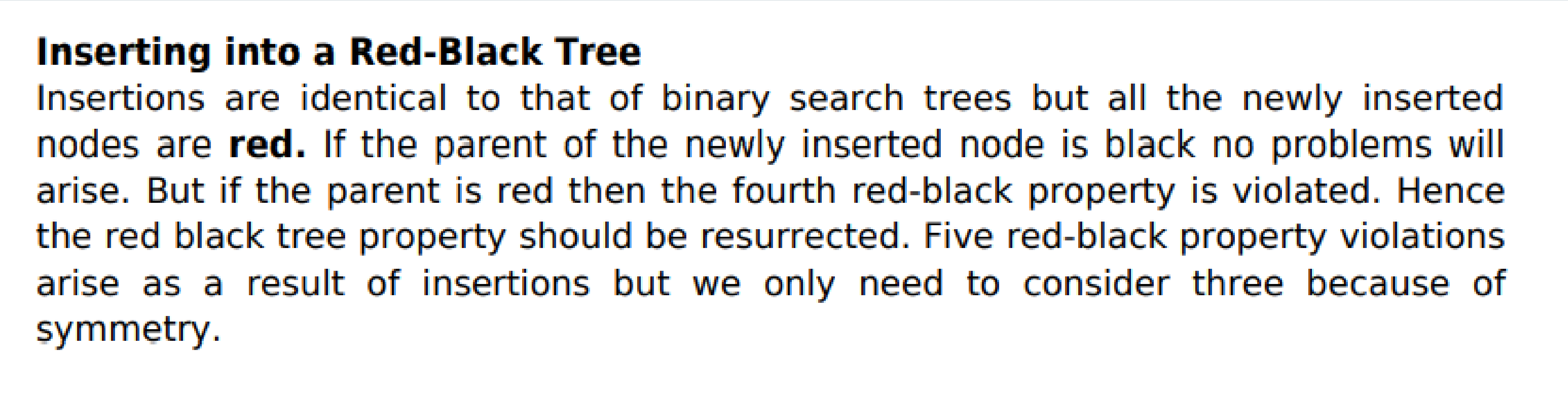


Black Height

The number of black nodes from a given node(X) down to a leaf excluding the node itself is called the black height. The property five of the red-black tree clearly defines this property. Each and every node should have the same black height when we compare the left subtree and the right subtree.

Ex:- The black height of the node with the value 45 is 1 (2 if one considers the leaf node). If observed clearly this is true for each given path. This can be denoted as bh(x) where X being the node being considered. Therefore bh(45) is 1.

RED BLACK TREES - INSERTION



Inserting into a Red-Black Tree

Insertions are identical to that of binary search trees but all the newly inserted nodes are **red.** If the parent of the newly inserted node is black no problems will arise. But if the parent is red then the fourth red-black property is violated. Hence the red black tree property should be resurrected. Five red-black property violations arise as a result of insertions but we only need to consider three because of symmetry.

RED BLACK TREES - INSERTION

(i) ***Newly inserted node's uncle is red:-*** The figure 3 (a) and (b) illustrates this violation after the new node with the value 26 (or 5) is inserted.

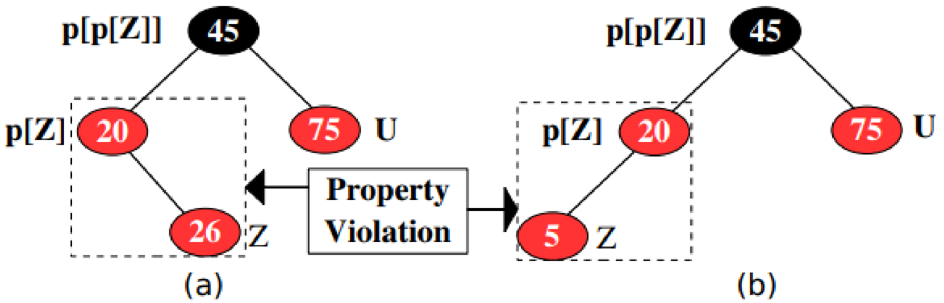


Figure 3: RBT property violation after inserting the value (a) 26 (b) 5

In the figure 3 the *uncle* of the node in question **(z)** is marked as **u** and p[z] and p[p[z]] stands for the parent and the grandparent respectively. This property violation can be fixed by recoloring of the nodes in the following order.

- The color of Z's uncle U and its parent p[Z] is changed from **red** to **black**
- The grandparent (p[p[Z]]) of Z is colored **red**.
- Update Z so now it points to the grandparent (**new Z**) of the initial setting
- Keep repeating the above steps until the red-black property is resurrected or the ***new Z*** reaches the root.
- If the root was colored **red** recolor it to **black**.

RED BLACK TREES - INSERTION

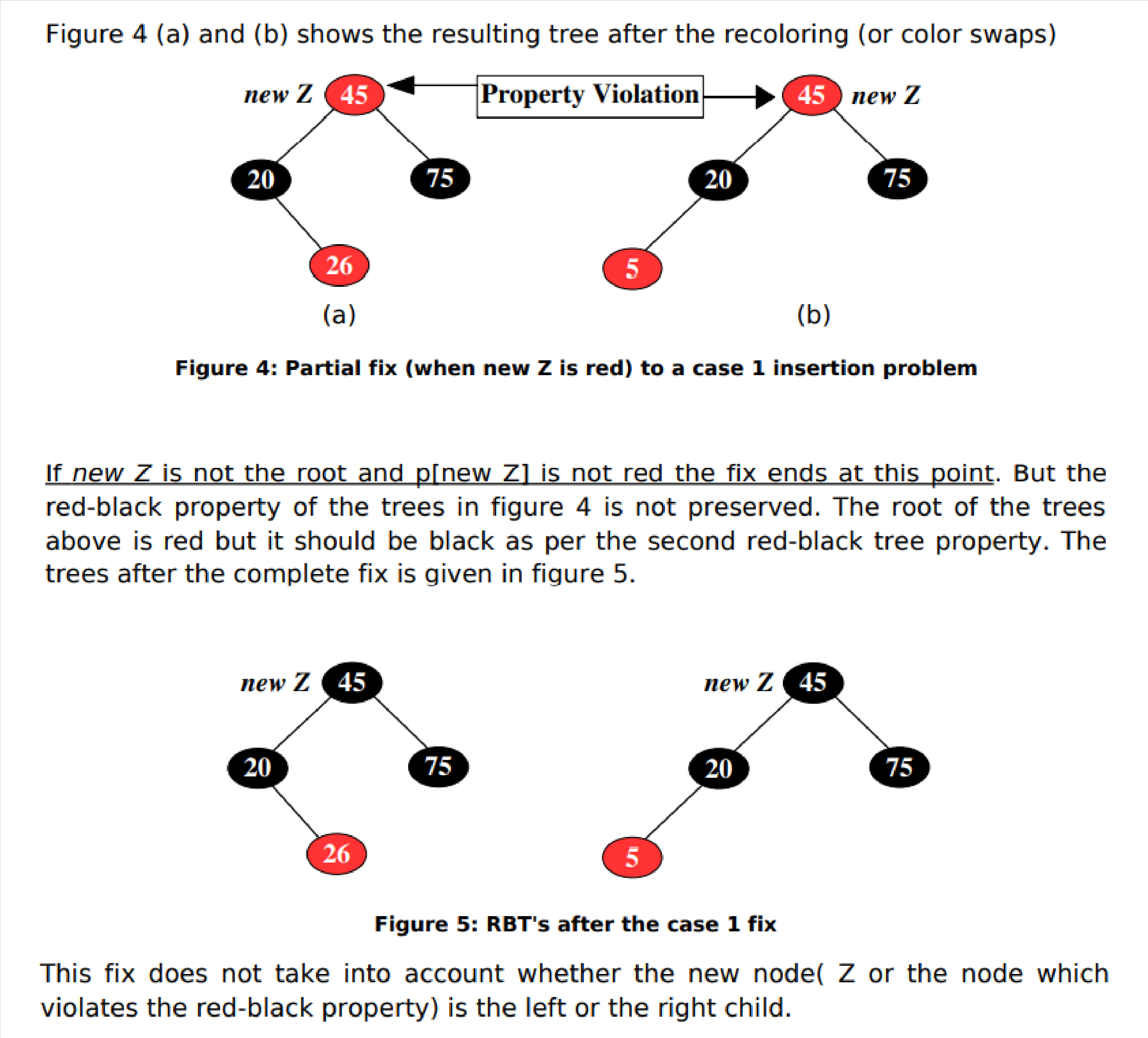


Figure 4 (a) and (b) shows the resulting tree after the recoloring (or color swaps)

new Z
45
Property Violation
45
new Z
20
75
20
75
26
5
(a)
(b)

Figure 4: Partial fix (when new Z is red) to a case 1 insertion problem

If *new Z* is not the root and p[new Z] is not red the fix ends at this point. But the red-black property of the trees in figure 4 is not preserved. The root of the trees above is red but it should be black as per the second red-black tree property. The trees after the complete fix is given in figure 5.

Figure 5: RBT's after the case 1 fix

This fix does not take into account whether the new node(Z or the node which violates the red-black property) is the left or the right child.

RED BLACK TREES - INSERTION

(ii) Newly inserted node's (Z) uncle is black and Z is a right child:- This is a complex case which requires rotations. Figure 6 (a) illustrates this red-black property violation. Keep note that this case has a symmetrical formation where the uncle is black and the Z is a left child. To reduce this complex case in to a simpler case the following steps are carried out in the order mentioned below.

- Left (or right) rotate Z about its parent p[Z] which results in the tree in figure 6 (b).
- Update Z so now it points to the parent of Z in the initial setting.

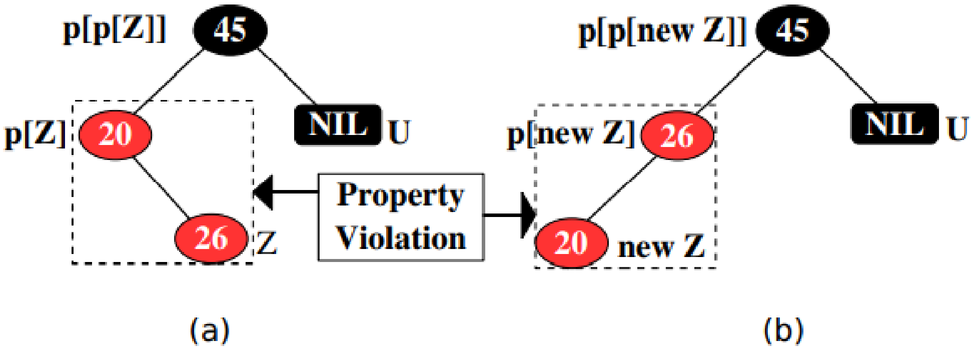
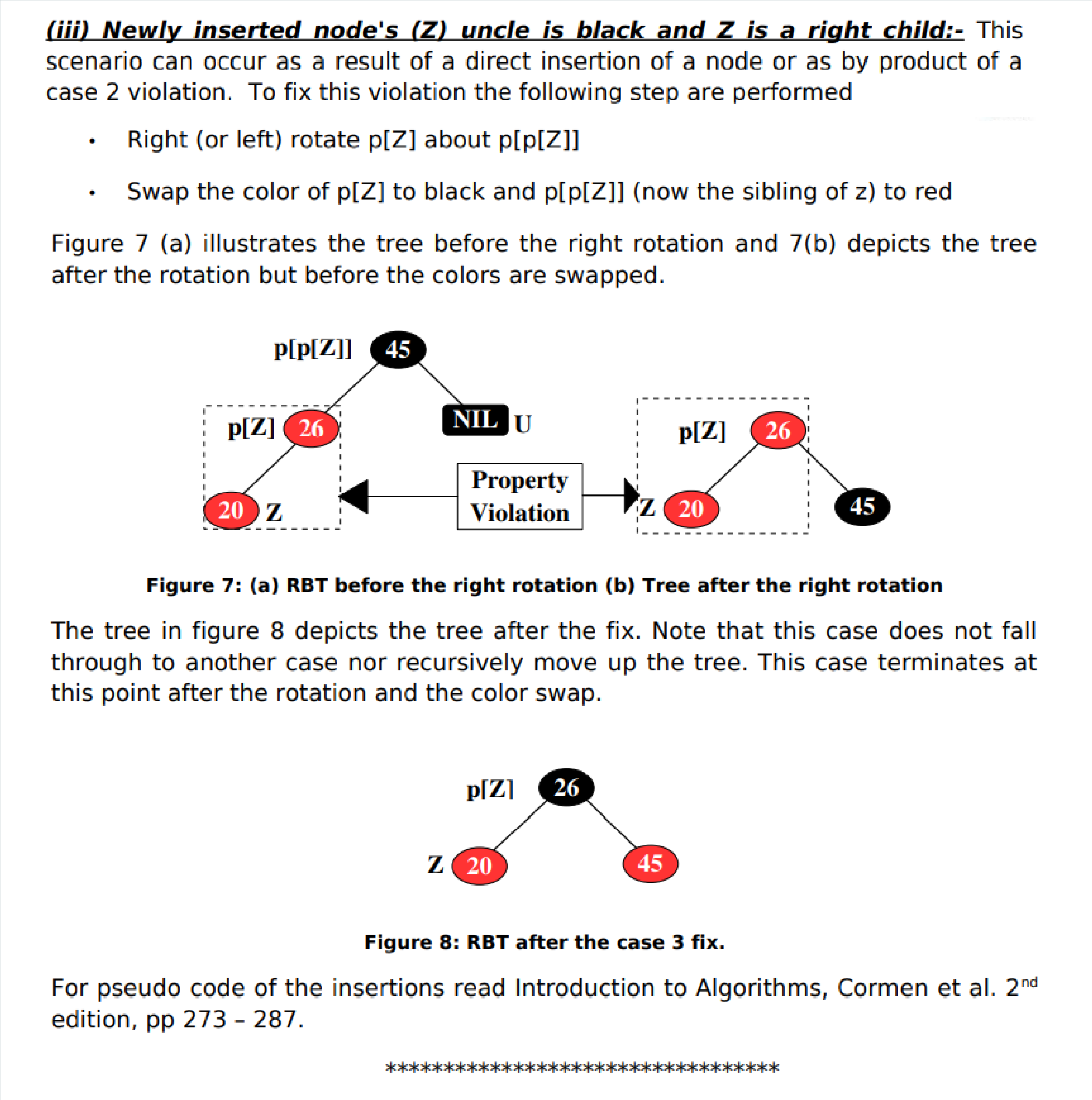


Figure 6: (a) RBT before the left rotation (b) Tree after the left rotation

In the above example the uncle of the newly inserted node 26 is the NIL node. By definition NIL node is always black. So a left rotation is performed on the node containing the value 26 about its parent 20 and the resulting tree is given in figure 6 (b). Note that the red-black property violation still remains after the rotation. The resulting violation is the case 3 violation. So technically the rotation have reduce the complex violation into much simpler case.

RED BLACK TREES - INSERTION



(iii) Newly inserted node's (Z) uncle is black and Z is a right child:- This scenario can occur as a result of a direct insertion of a node or as by product of a case 2 violation. To fix this violation the following step are performed

- Right (or left) rotate p[Z] about p[p[Z]]
- Swap the color of p[Z] to black and p[p[Z]] (now the sibling of z) to red

Figure 7 (a) illustrates the tree before the right rotation and 7(b) depicts the tree after the rotation but before the colors are swapped.

Figure 7: (a) RBT before the right rotation (b) Tree after the right rotation

The tree in figure 8 depicts the tree after the fix. Note that this case does not fall through to another case nor recursively move up the tree. This case terminates at this point after the rotation and the color swap.

Figure 8: RBT after the case 3 fix.

For pseudo code of the insertions read Introduction to Algorithms, Cormen et al. 2nd edition, pp 273 – 287.

****

RED BLACK TREES - INSERTION

When it comes to red black tree insertion there is a right way and easy way. Let us see the easy way.

Red black tree nodes should be either black or red.There are some properties that should be satisfied in order to create a valid red black tree.

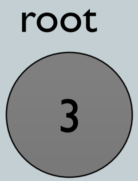
Properties:

- Every node is red or black
- Root is always black
- New insertions are always red
- Every path from root to leaf has the same number of black nodes
- No path can have two consecutive red nodes
- Nulls are black

When she says "Some of your red-black trees donot even have a red color so how is it redblack?"
rsnpj
You wouldn't get it
Laughs in data structures

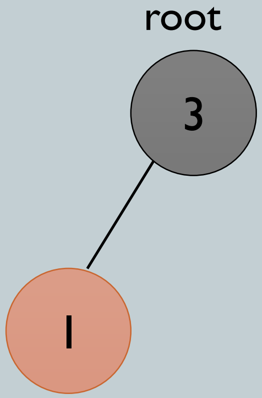
RED BLACK TREES - INSERTION

Insert: 3, 1, 5, 7, 6, 8, 9, 10



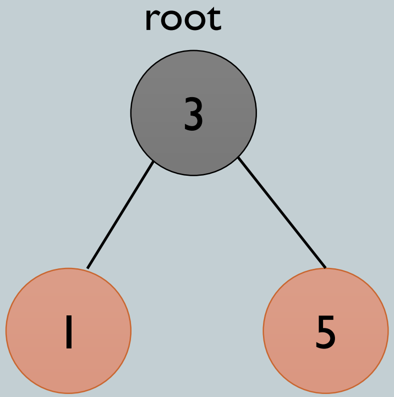
RED BLACK TREES - INSERTION

Insert: 3, 1, 5, 7, 6, 8, 9, 10



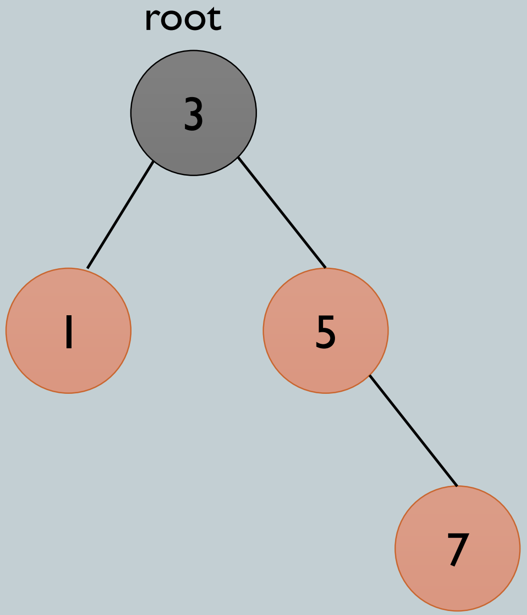
RED BLACK TREES - INSERTION

Insert: 3, 1, 5, 7, 6, 8, 9, 10

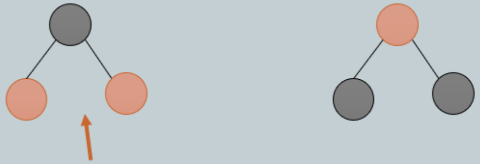


RED BLACK TREES - INSERTION

Insert: 3, 1, 5, 7, 6, 8, 9, 10



Violation (consecutive red nodes)



This the color flip after rotation

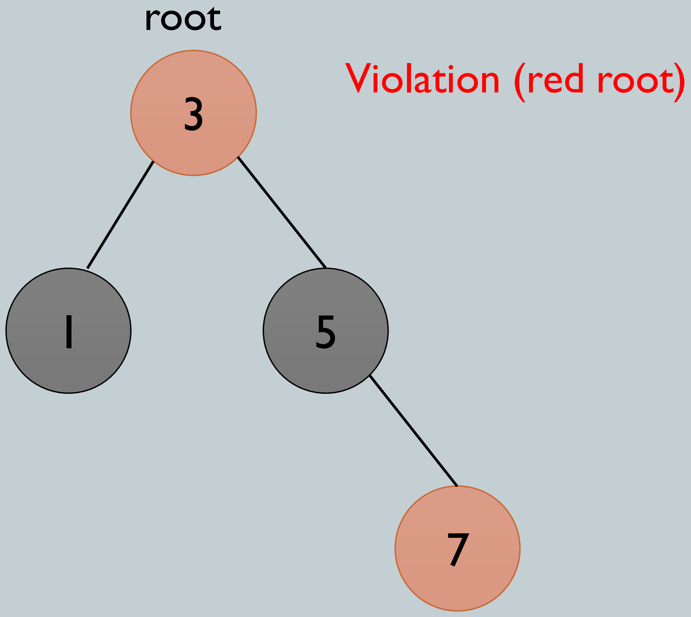
Rebalance conditions:

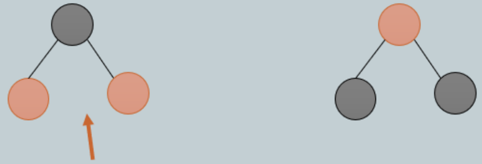
- If we have black uncle we rotate (rotation have 4 cases)
- If we have a red uncle we color flip

  (Note that after rotation we do a color flip as well)

RED BLACK TREES - INSERTION

Insert: 3, 1, 5, 7, 6, 8, 9, 10





This the color flip after rotation

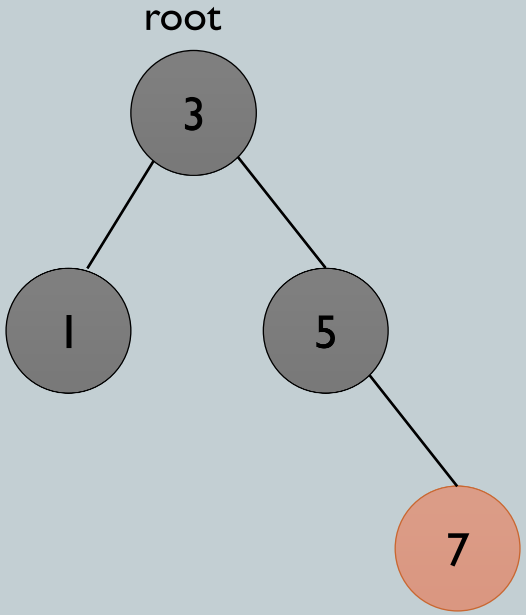
Rebalance conditions:

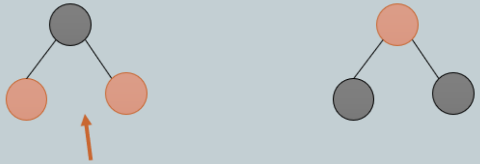
- If we have black uncle we rotate (rotation have 4 cases)
- If we have a red uncle we color flip

  (Note that after rotation we do a color flip as well)

RED BLACK TREES - INSERTION

Insert: 3, 1, 5, 7, 6, 8, 9, 10





This the color flip after rotation

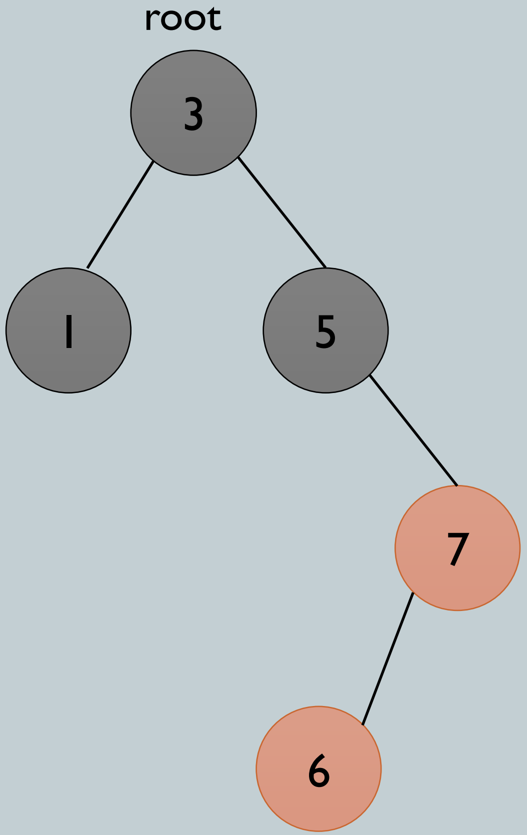
Rebalance conditions:

- If we have black uncle we rotate (rotation have 4 cases)
- If we have a red uncle we color flip

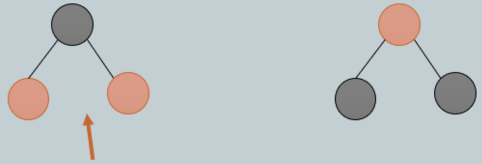
  (Note that after rotation we do a color flip as well)

RED BLACK TREES - INSERTION

Insert: 3, 1, 5, 7, 6, 8, 9, 10



Violation (consecutive red nodes)



This the color flip after rotation

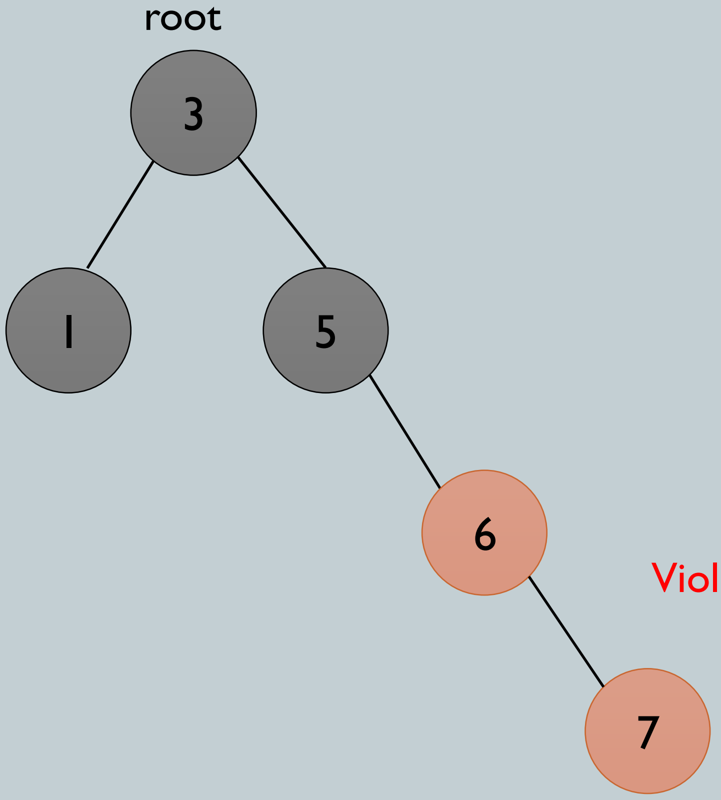
Rebalance conditions:

- If we have black uncle we rotate (rotation have 4 cases)
- If we have a red uncle we color flip

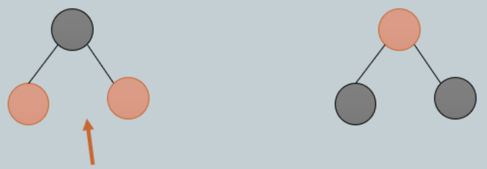
  (Note that after rotation we do a color flip as well)

RED BLACK TREES - INSERTION

Insert: 3, 1, 5, 7, 6, 8, 9, 10



Violation (consecutive red nodes)



This the color flip after rotation

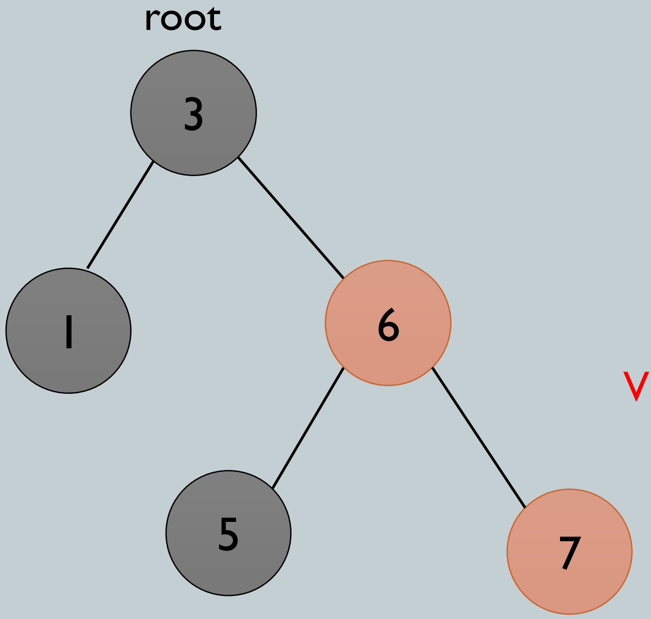
Rebalance conditions:

- If we have black uncle we rotate (rotation have 4 cases)
- If we have a red uncle we color flip

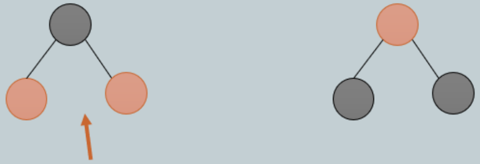
  (Note that after rotation we do a color flip as well)

RED BLACK TREES - INSERTION

Insert: 3, 1, 5, 7, 6, 8, 9, 10



Violation (consecutive red nodes)



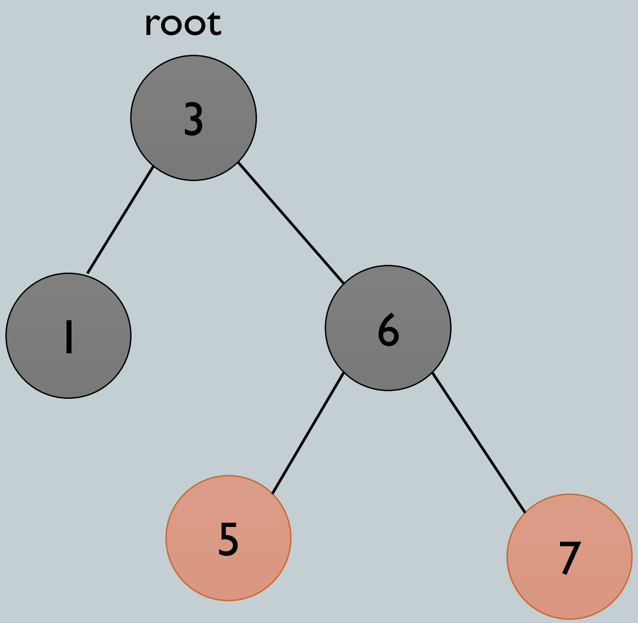
This the color flip after rotation

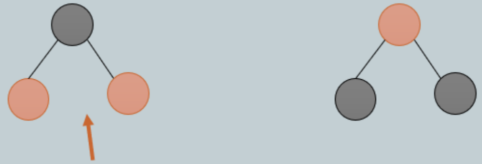
Rebalance conditions:

- If we have black uncle we rotate (rotation have 4 cases)
- If we have a red uncle we color flip
  (Note that after rotation we do a color flip as well)

RED BLACK TREES - INSERTION

Insert: 3, 1, 5, 7, 6, 8, 9, 10





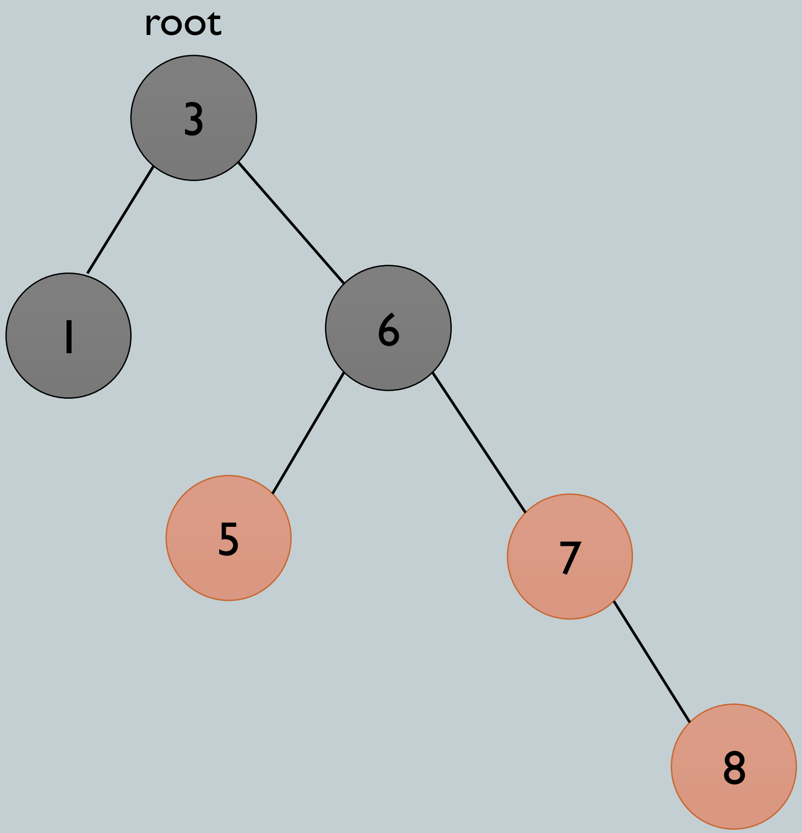
This the color flip after rotation

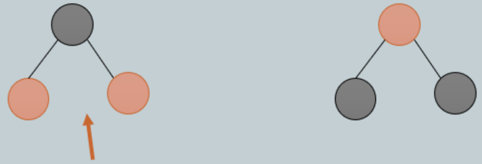
Rebalance conditions:

- If we have black uncle we rotate (rotation have 4 cases)
- If we have a red uncle we color flip
  (Note that after rotation we do a color flip as well)

RED BLACK TREES - INSERTION

Insert: 3, 1, 5, 7, 6, 8, 9, 10





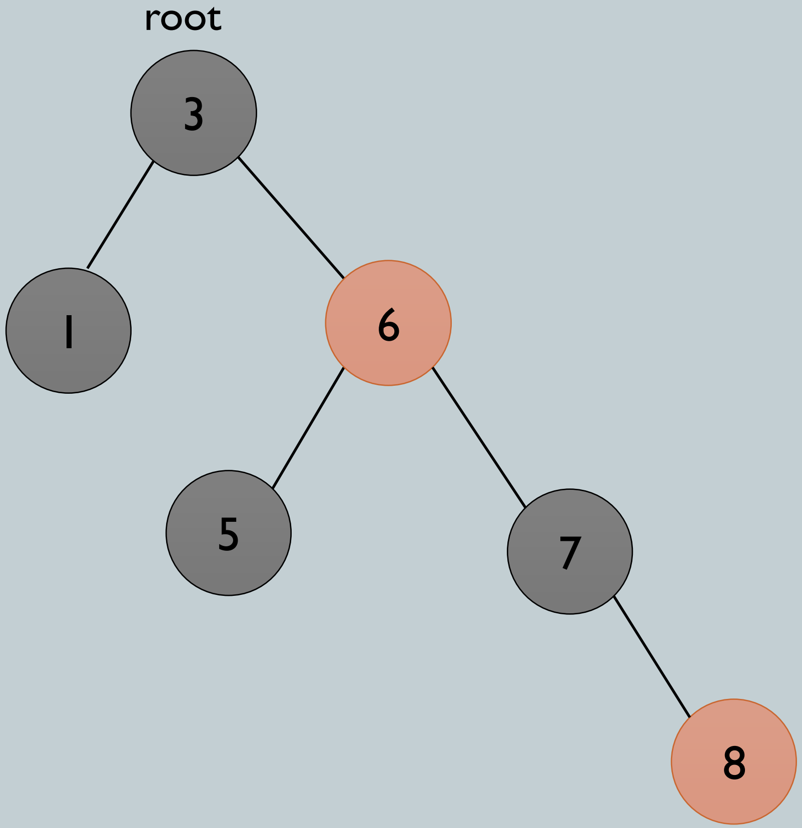
This the color flip after rotation

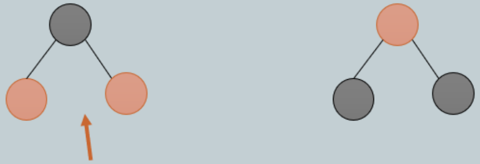
Rebalance conditions:

- If we have black uncle we rotate (rotation have 4 cases)
- If we have a red uncle we color flip
  (Note that after rotation we do a color flip as well)

RED BLACK TREES - INSERTION

Insert: 3, 1, 5, 7, 6, 8, 9, 10





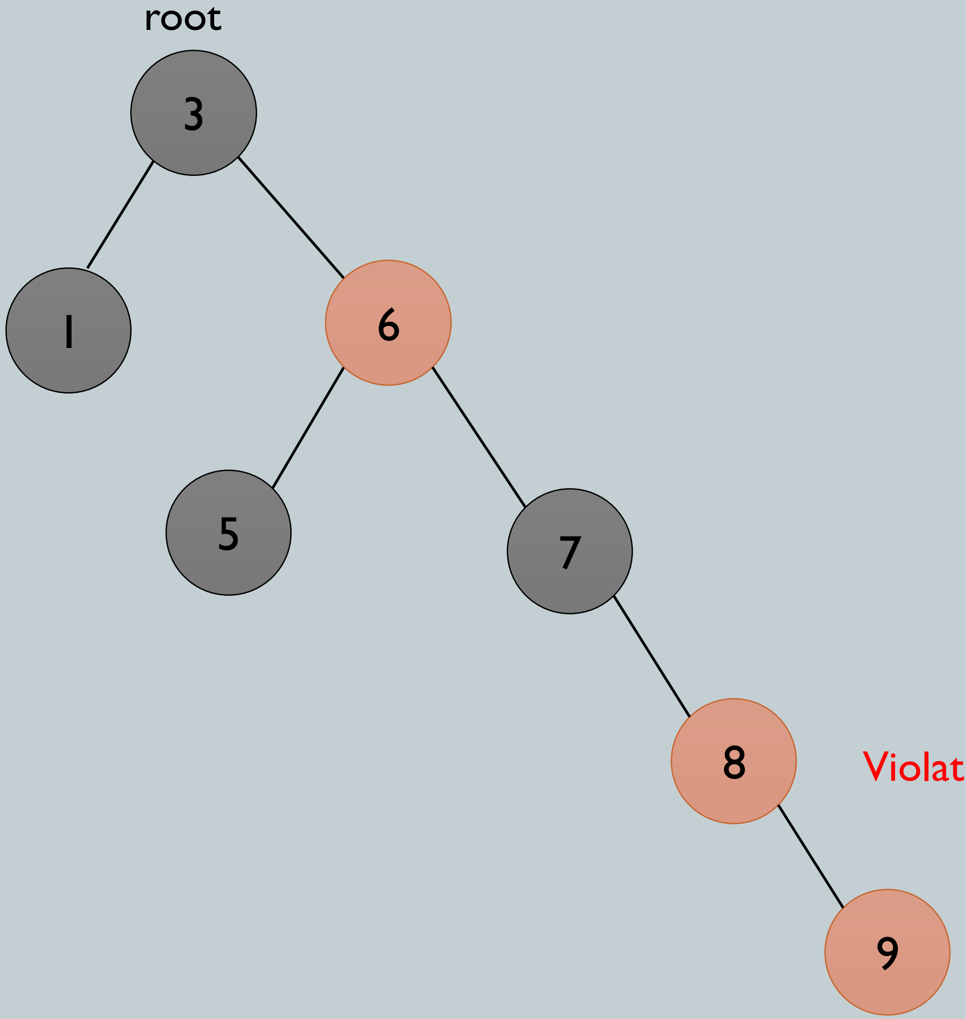
This the color flip after rotation

Rebalance conditions:

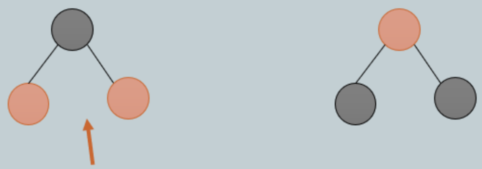
- If we have black uncle we rotate (rotation have 4 cases)
- If we have a red uncle we color flip
  (Note that after rotation we do a color flip as well)

RED BLACK TREES - INSERTION

Insert: 3, 1, 5, 7, 6, 8, 9, 10



Violation (consecutive red nodes)



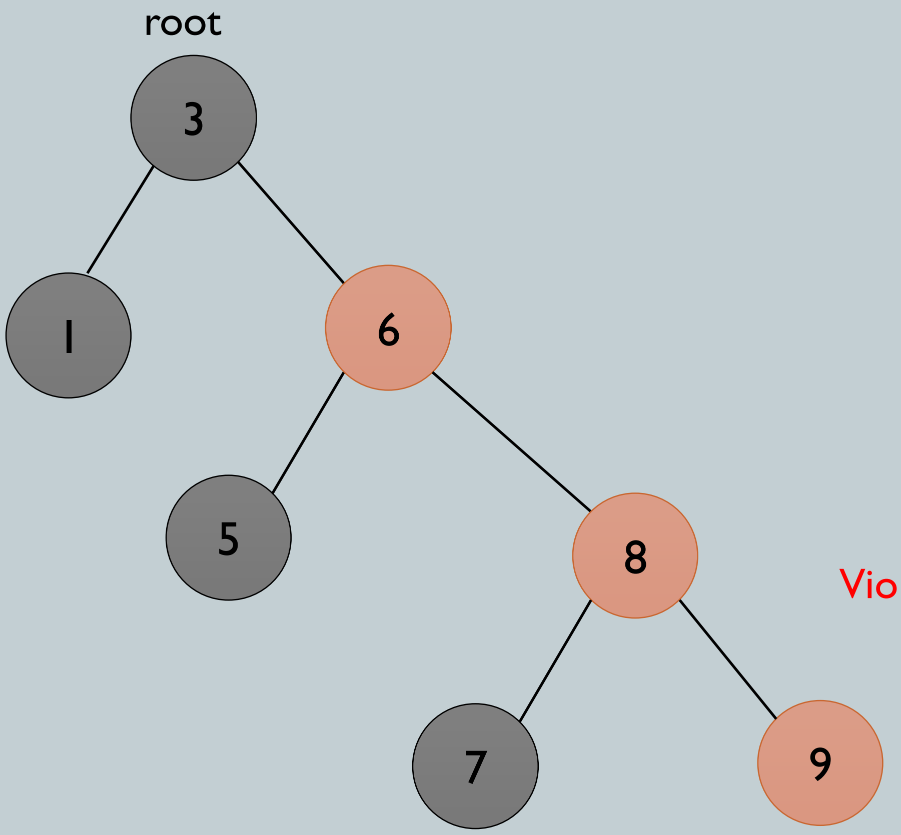
This the color flip after rotation

Rebalance conditions:

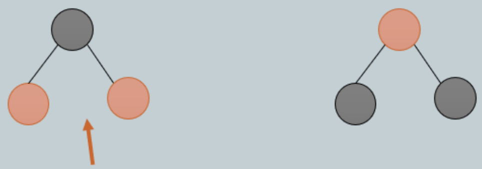
- If we have black uncle we rotate (rotation have 4 cases)
- If we have a red uncle we color flip
  (Note that after rotation we do a color flip as well)

RED BLACK TREES - INSERTION

Insert: 3, 1, 5, 7, 6, 8, 9, 10



Violation (consecutive red nodes)



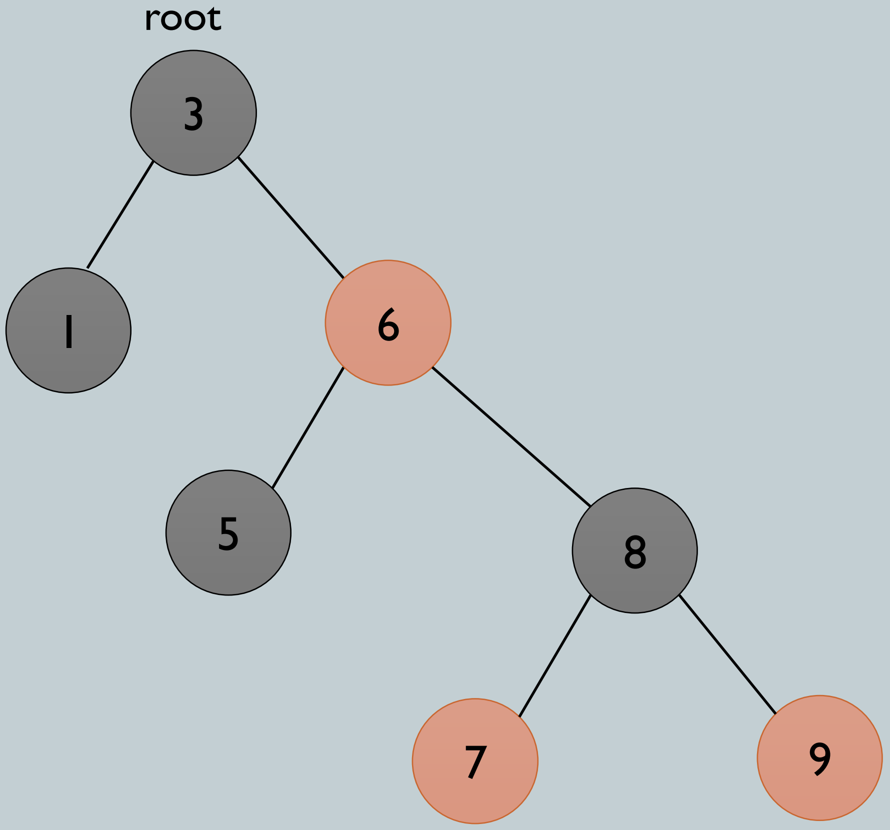
This the color flip after rotation

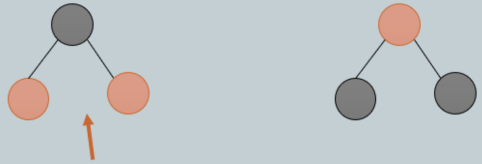
Rebalance conditions:

- If we have black uncle we rotate (rotation have 4 cases)
- If we have a red uncle we color flip
  (Note that after rotation we do a color flip as well)

RED BLACK TREES - INSERTION

Insert: 3, 1, 5, 7, 6, 8, 9, 10





This the color flip after rotation

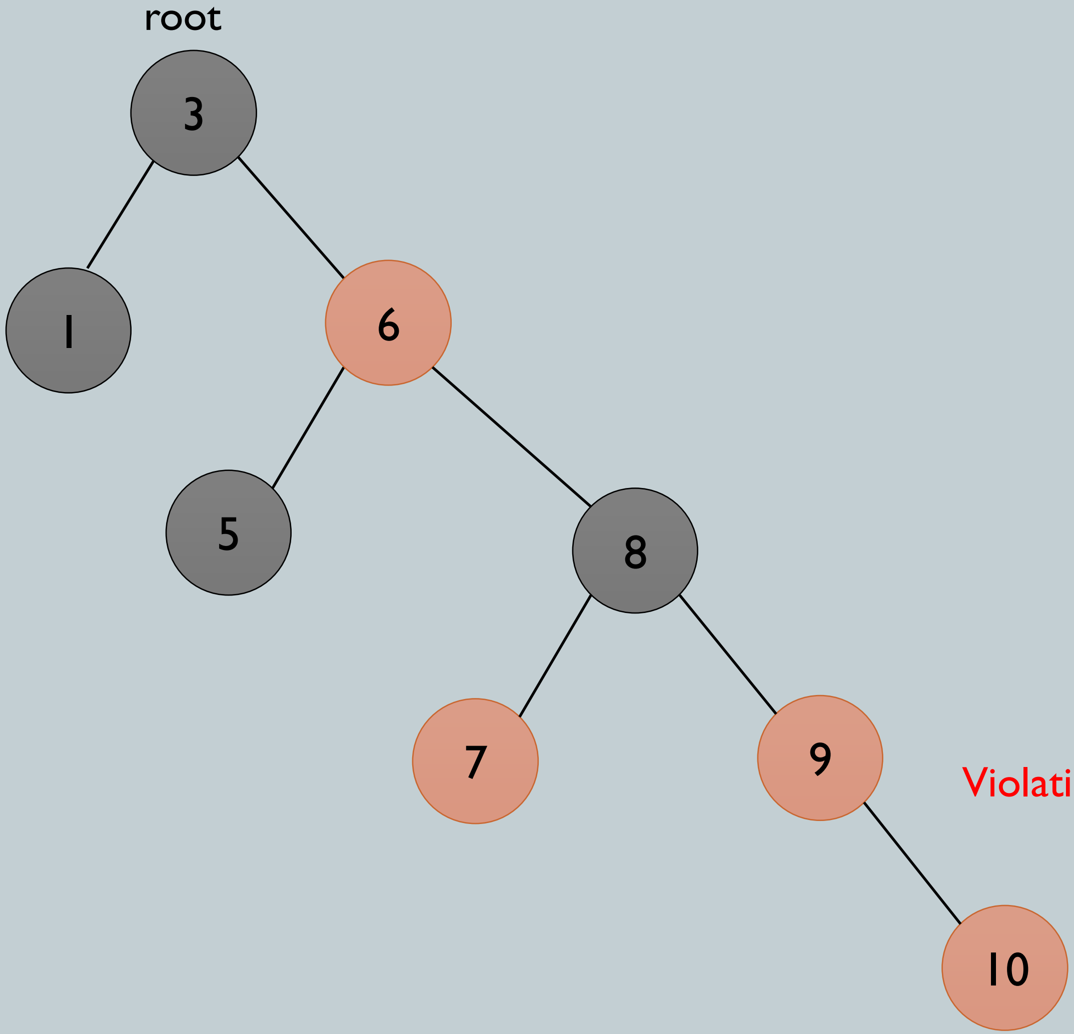
Rebalance conditions:

- If we have black uncle we rotate (rotation have 4 cases)
- If we have a red uncle we color flip

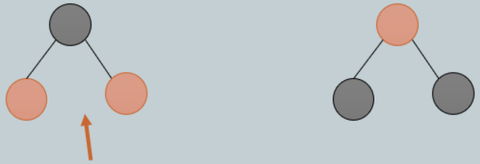
  (Note that after rotation we do a color flip as well)

RED BLACK TREES - INSERTION

Insert: 3, 1, 5, 7, 6, 8, 9, 10



Violation (consecutive red nodes)



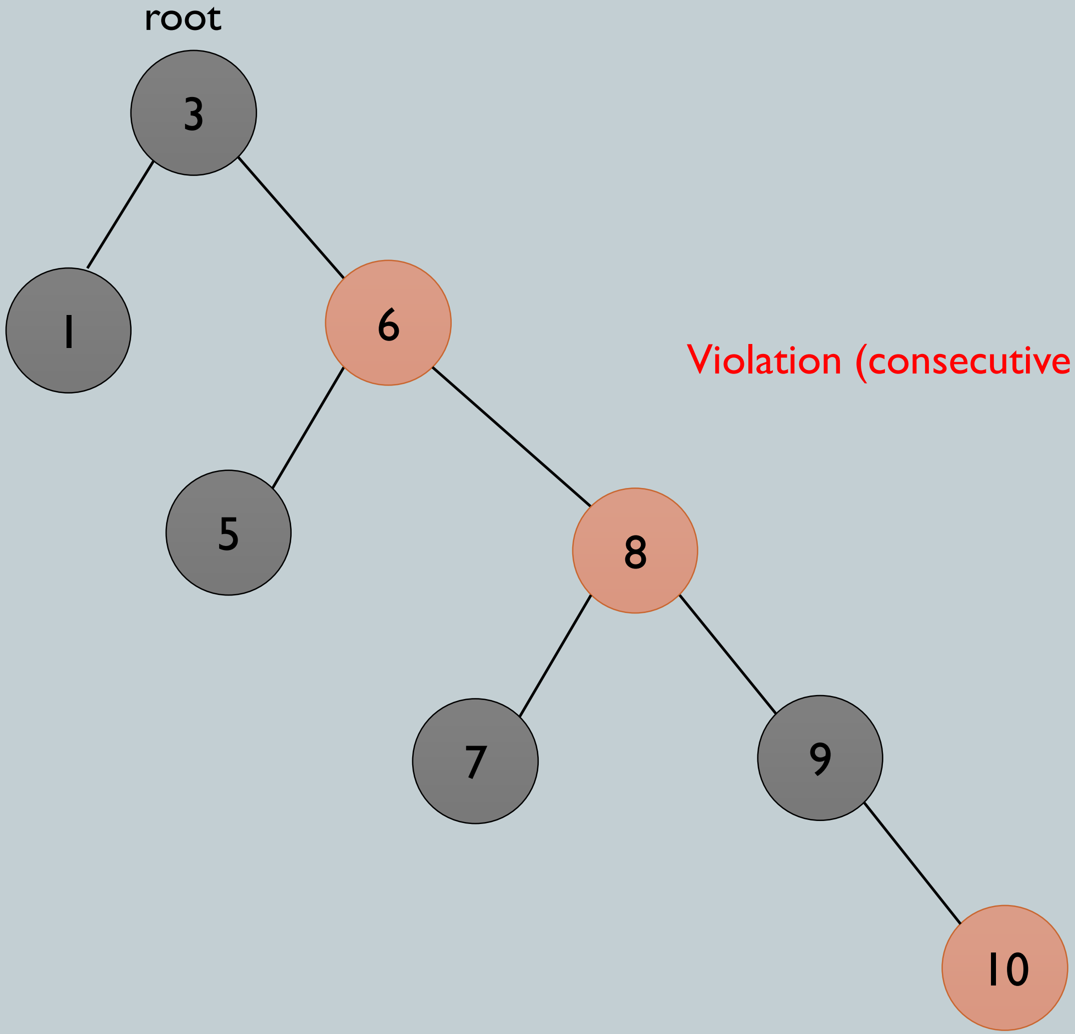
This the color flip after rotation

Rebalance conditions:

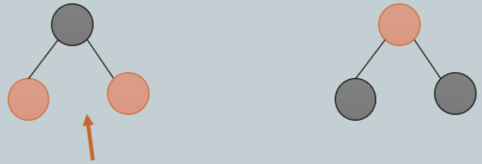
- If we have black uncle we rotate (rotation have 4 cases)
- If we have a red uncle we color flip
  (Note that after rotation we do a color flip as well)

RED BLACK TREES - INSERTION

Insert: 3, 1, 5, 7, 6, 8, 9, 10



Violation (consecutive red nodes)



This the color flip after rotation

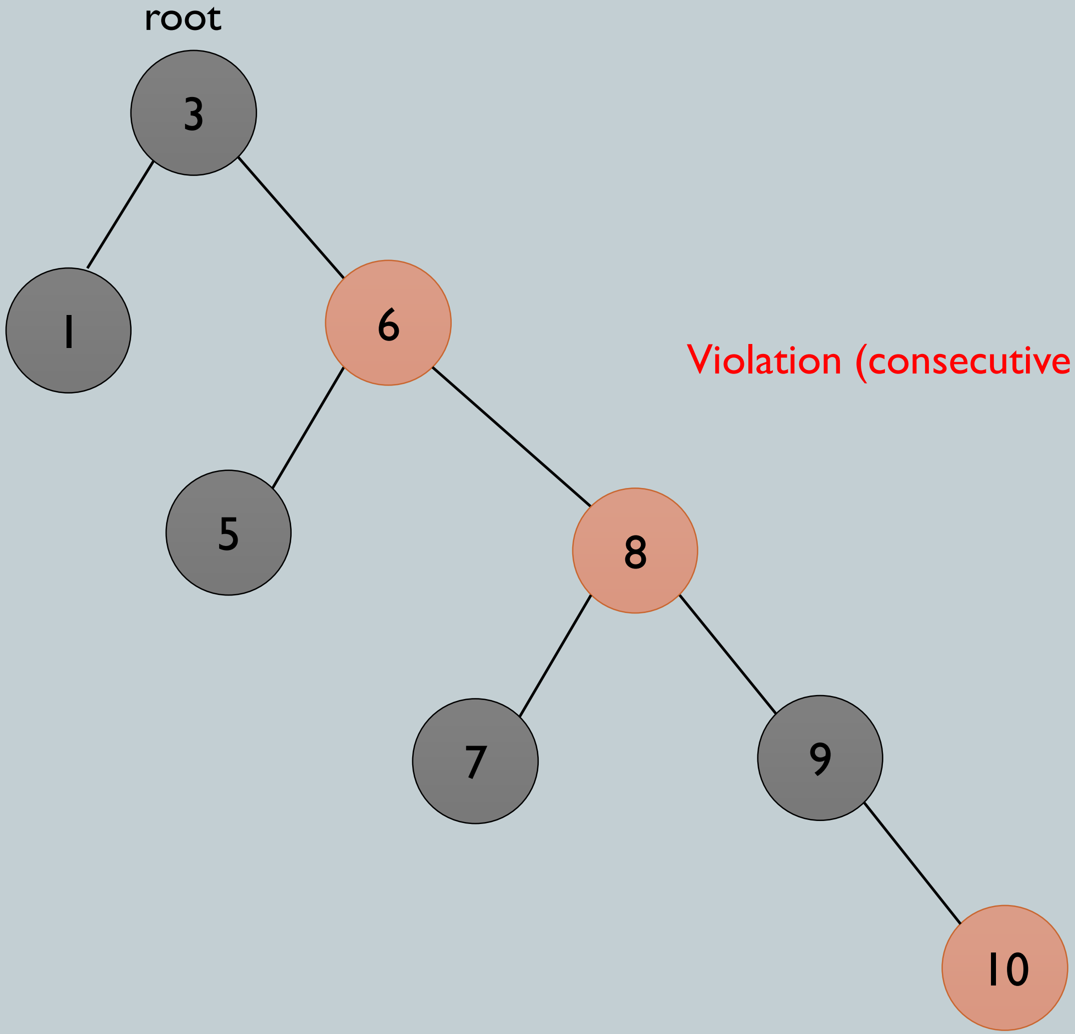
Rebalance conditions:

- If we have black uncle we rotate (rotation have 4 cases)
- If we have a red uncle we color flip

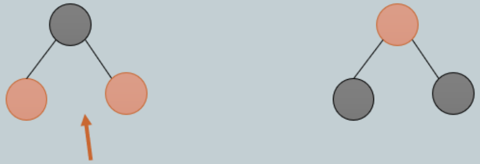
  (Note that after rotation we do a color flip as well)

RED BLACK TREES - INSERTION

Insert: 3, 1, 5, 7, 6, 8, 9, 10



Violation (consecutive red nodes)



This the color flip after rotation

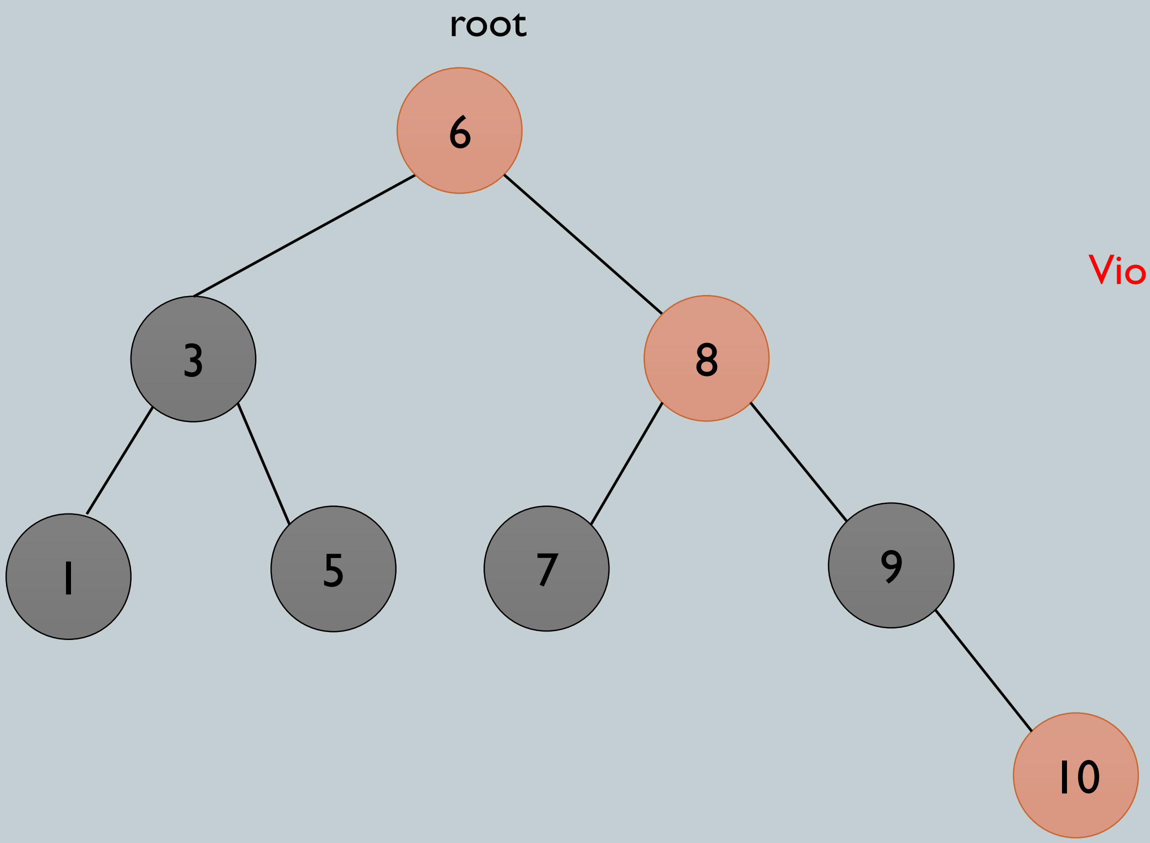
Rebalance conditions:

- If we have black uncle we rotate (rotation have 4 cases)
- If we have a red uncle we color flip

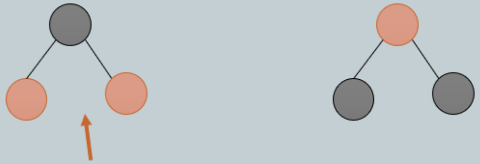
  (Note that after rotation we do a color flip as well)

RED BLACK TREES - INSERTION

Insert: 3, 1, 5, 7, 6, 8, 9, 10



Violation (consecutive red nodes)



This the color flip after rotation

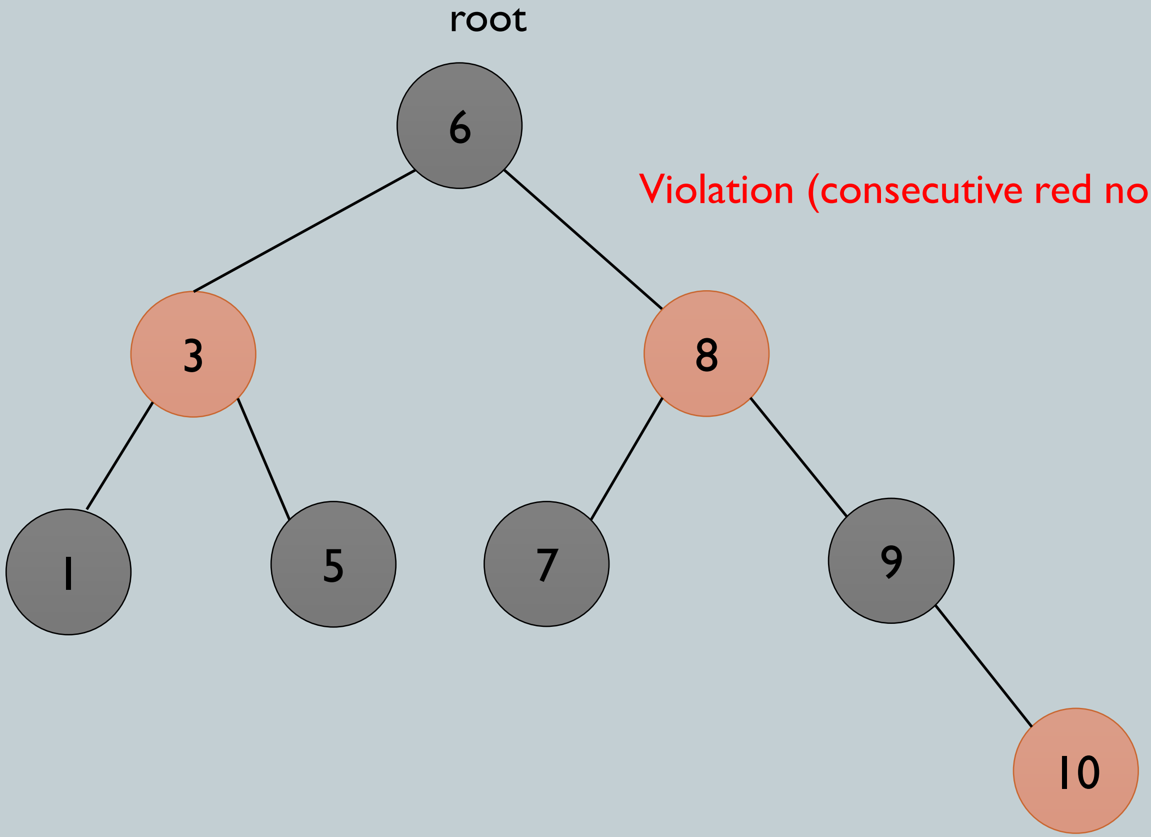
Rebalance conditions:

- If we have black uncle we rotate (rotation have 4 cases)
- If we have a red uncle we color flip

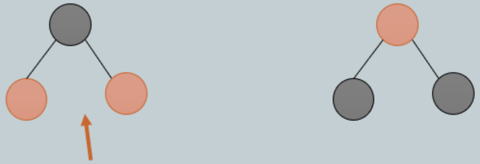
  (Note that after rotation we do a color flip as well)

RED BLACK TREES - INSERTION

Insert: 3, 1, 5, 7, 6, 8, 9, 10



Violation (consecutive red nodes)



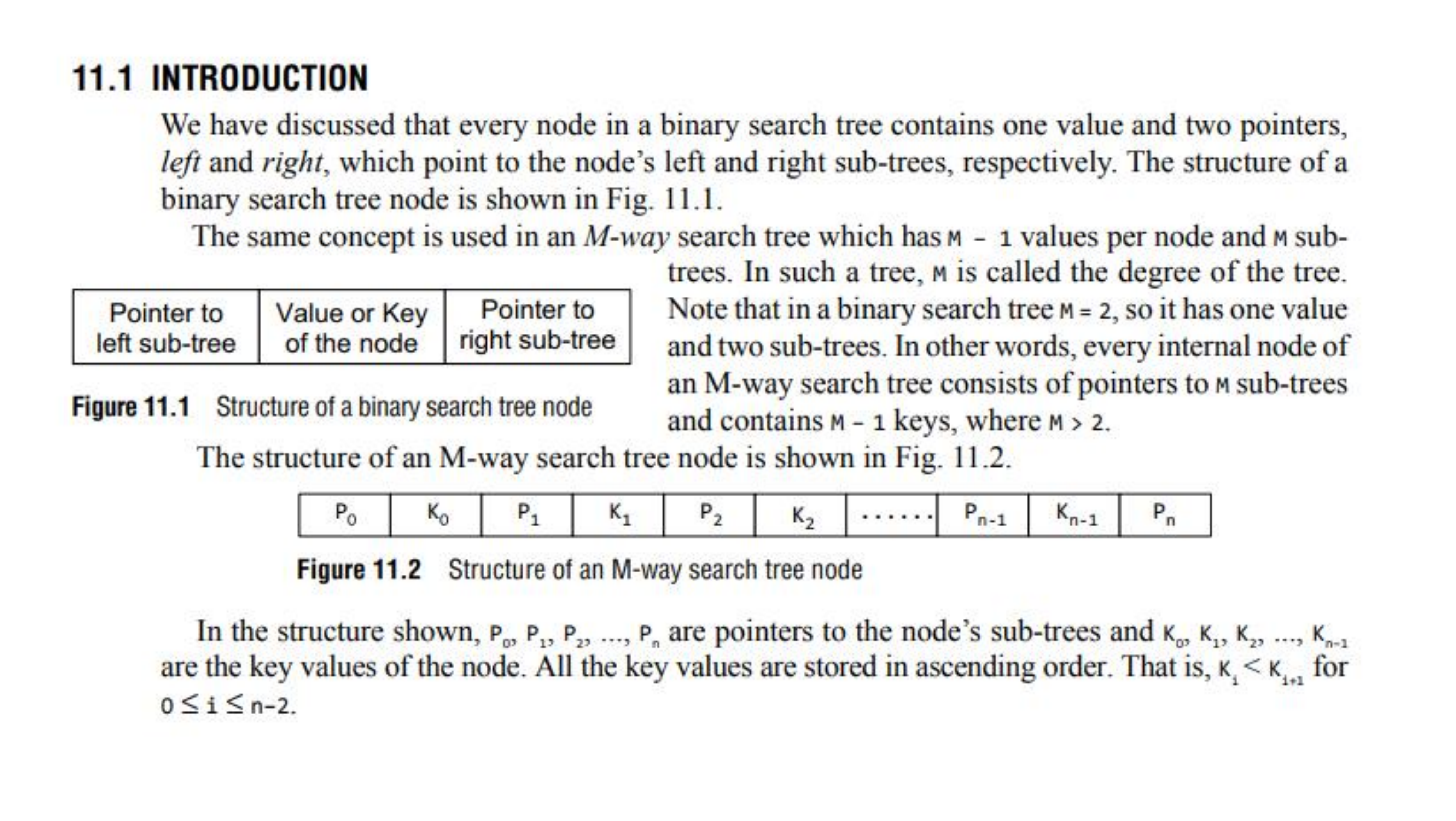
This the color flip after rotation

Rebalance conditions:

- If we have black uncle we rotate (rotation have 4 cases)
- If we have a red uncle we color flip
  (Note that after rotation we do a color flip as well)

MULTIWAY SEARCH TREES

MULTIWAY SEARCH TREES



11.1 INTRODUCTION

We have discussed that every node in a binary search tree contains one value and two pointers, *left* and *right*, which point to the node's left and right sub-trees, respectively. The structure of a binary search tree node is shown in Fig. 11.1.

The same concept is used in an *M-way* search tree which has M - 1 values per node and M sub-trees. In such a tree, M is called the degree of the tree. Note that in a binary search tree M = 2, so it has one value and two sub-trees. In other words, every internal node of an M-way search tree consists of pointers to M sub-trees and contains M - 1 keys, where M > 2.

Pointer to left sub-tree	Value or Key of the node	Pointer to right sub-tree

Figure 11.1 Structure of a binary search tree node

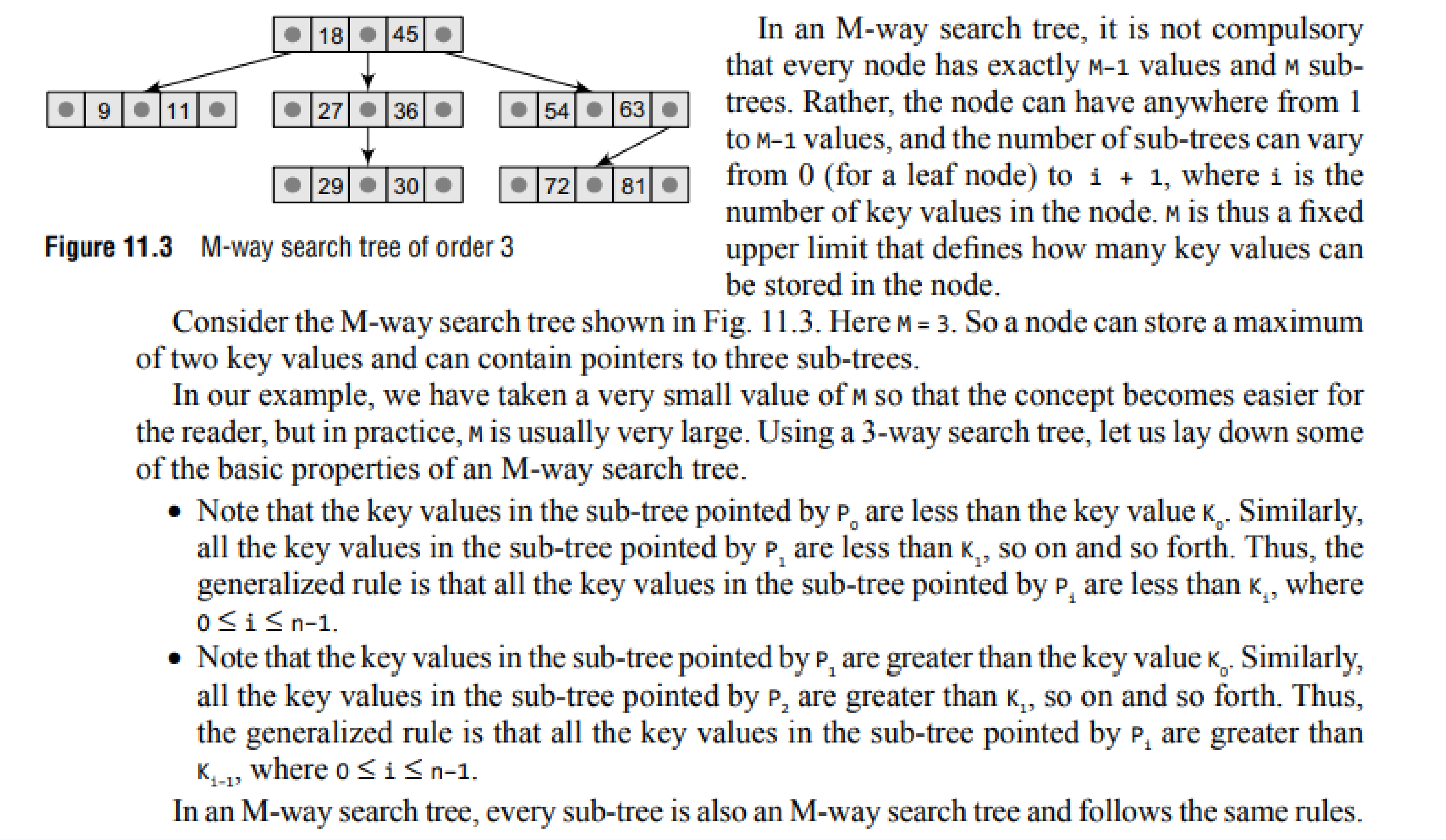
The structure of an M-way search tree node is shown in Fig. 11.2.

P_0	K_0	P_1	K_1	P_2	K_2	......	P_{n-1}	K_{n-1}	P_n

Figure 11.2 Structure of an M-way search tree node

In the structure shown, P_0, P_1, P_2, ..., P_n are pointers to the node's sub-trees and K_0, K_1, K_2, ..., K_{n-1} are the key values of the node. All the key values are stored in ascending order. That is, $K_i < K_{i+1}$ for $0 \le i \le n-2$.

MULTIWAY SEARCH TREES



18
45
9
11
27
36
54
63
29
30
72
81

Figure 11.3 M-way search tree of order 3

In an M-way search tree, it is not compulsory that every node has exactly `M-1` values and `M` sub-trees. Rather, the node can have anywhere from 1 to `M-1` values, and the number of sub-trees can vary from 0 (for a leaf node) to `i + 1`, where `i` is the number of key values in the node. `M` is thus a fixed upper limit that defines how many key values can be stored in the node.

Consider the M-way search tree shown in Fig. 11.3. Here `M = 3`. So a node can store a maximum of two key values and can contain pointers to three sub-trees.

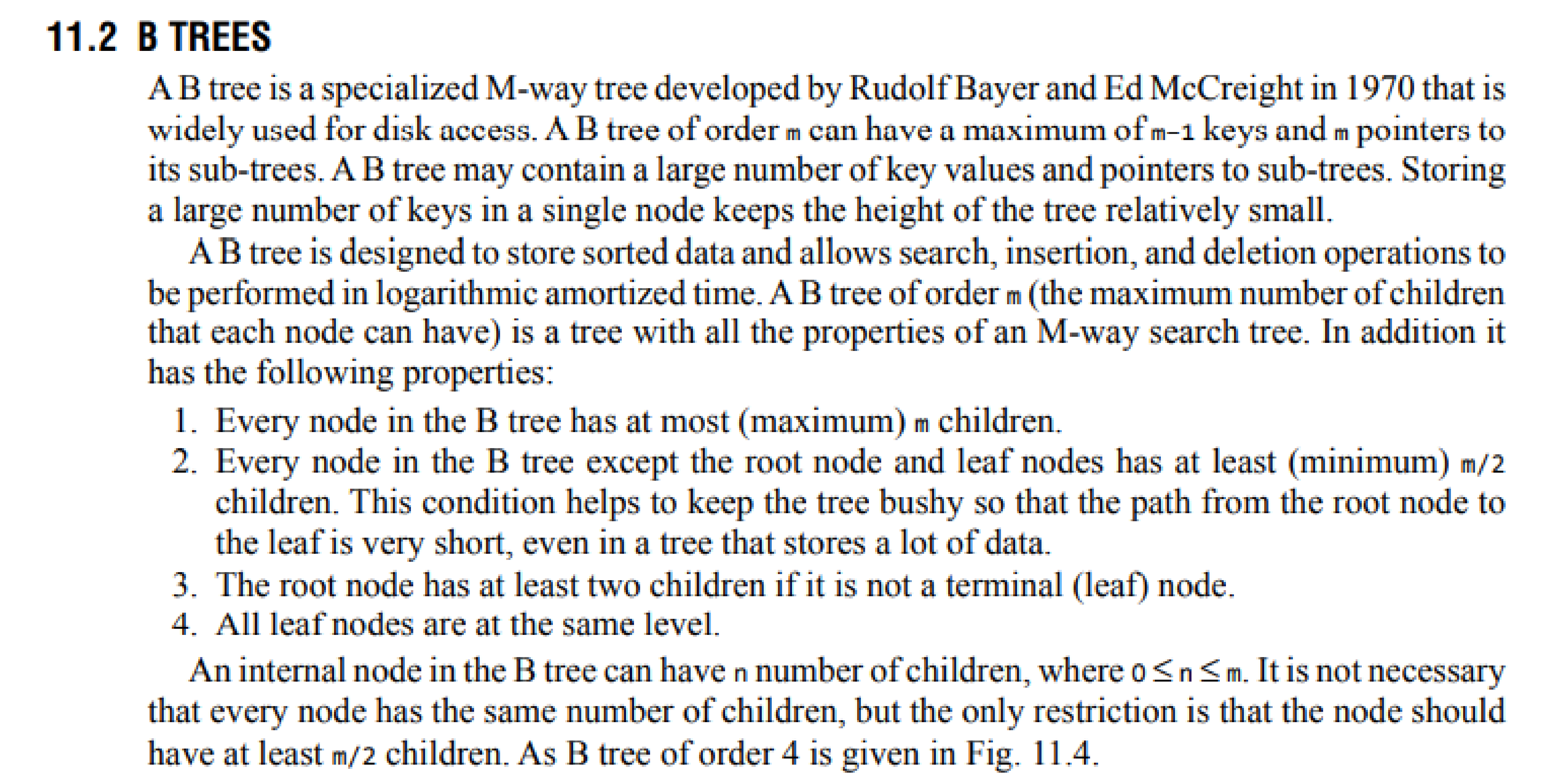
In our example, we have taken a very small value of `M` so that the concept becomes easier for the reader, but in practice, `M` is usually very large. Using a 3-way search tree, let us lay down some of the basic properties of an M-way search tree.

- Note that the key values in the sub-tree pointed by P_0 are less than the key value K_0. Similarly, all the key values in the sub-tree pointed by P_1 are less than K_1, so on and so forth. Thus, the generalized rule is that all the key values in the sub-tree pointed by P_i are less than K_i, where $0 \le i \le n-1$.
- Note that the key values in the sub-tree pointed by P_1 are greater than the key value K_0. Similarly, all the key values in the sub-tree pointed by P_2 are greater than K_1, so on and so forth. Thus, the generalized rule is that all the key values in the sub-tree pointed by P_i are greater than K_{i-1}, where $0 \le i \le n-1$.

In an M-way search tree, every sub-tree is also an M-way search tree and follows the same rules.

B TREES

B TREES



11.2 B TREES

A B tree is a specialized M-way tree developed by Rudolf Bayer and Ed McCreight in 1970 that is widely used for disk access. A B tree of order m can have a maximum of $m-1$ keys and m pointers to its sub-trees. A B tree may contain a large number of key values and pointers to sub-trees. Storing a large number of keys in a single node keeps the height of the tree relatively small.

A B tree is designed to store sorted data and allows search, insertion, and deletion operations to be performed in logarithmic amortized time. A B tree of order m (the maximum number of children that each node can have) is a tree with all the properties of an M-way search tree. In addition it has the following properties:

1. Every node in the B tree has at most (maximum) m children.
2. Every node in the B tree except the root node and leaf nodes has at least (minimum) $m/2$ children. This condition helps to keep the tree bushy so that the path from the root node to the leaf is very short, even in a tree that stores a lot of data.
3. The root node has at least two children if it is not a terminal (leaf) node.
4. All leaf nodes are at the same level.

An internal node in the B tree can have n number of children, where $0 \leq n \leq m$. It is not necessary that every node has the same number of children, but the only restriction is that the node should have at least $m/2$ children. As B tree of order 4 is given in Fig. 11.4.

B TREES

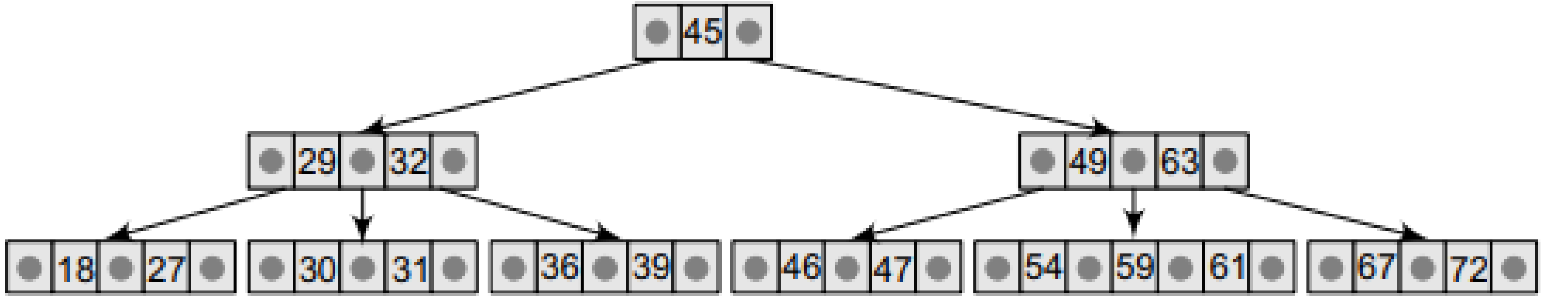
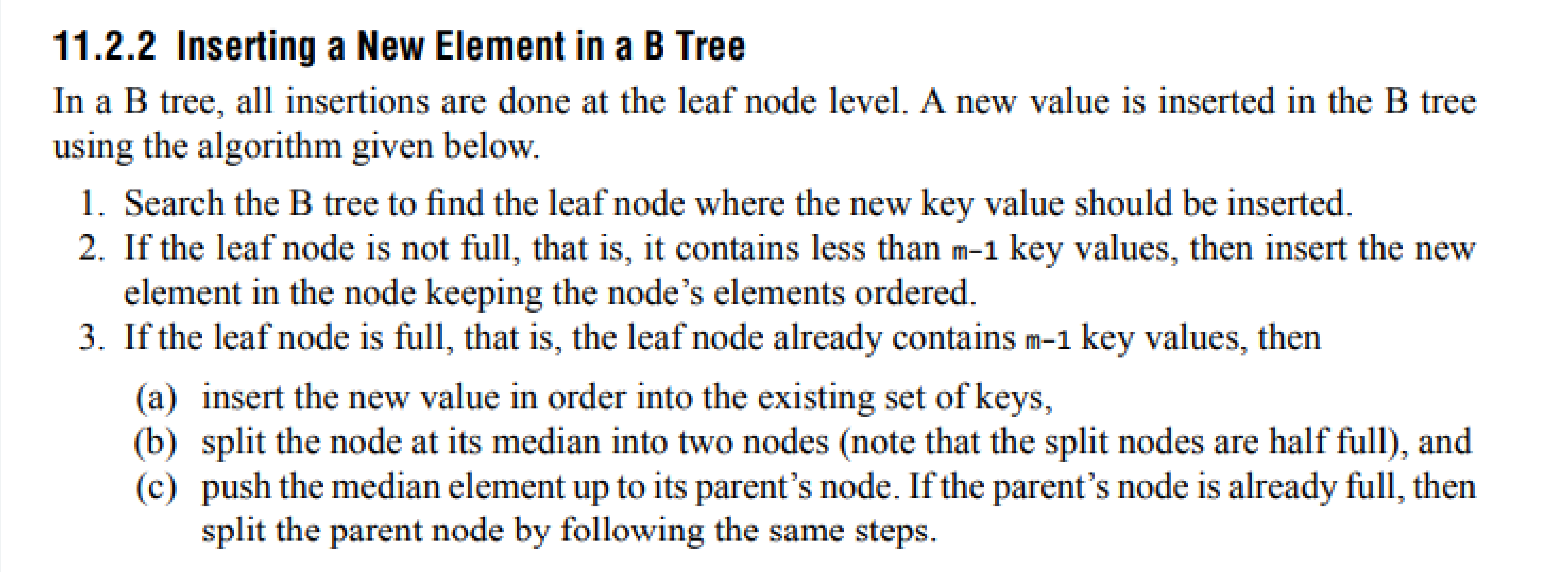


Figure 11.4 B tree of order 4

B TREE INSERTION



11.2.2 Inserting a New Element in a B Tree

In a B tree, all insertions are done at the leaf node level. A new value is inserted in the B tree using the algorithm given below.

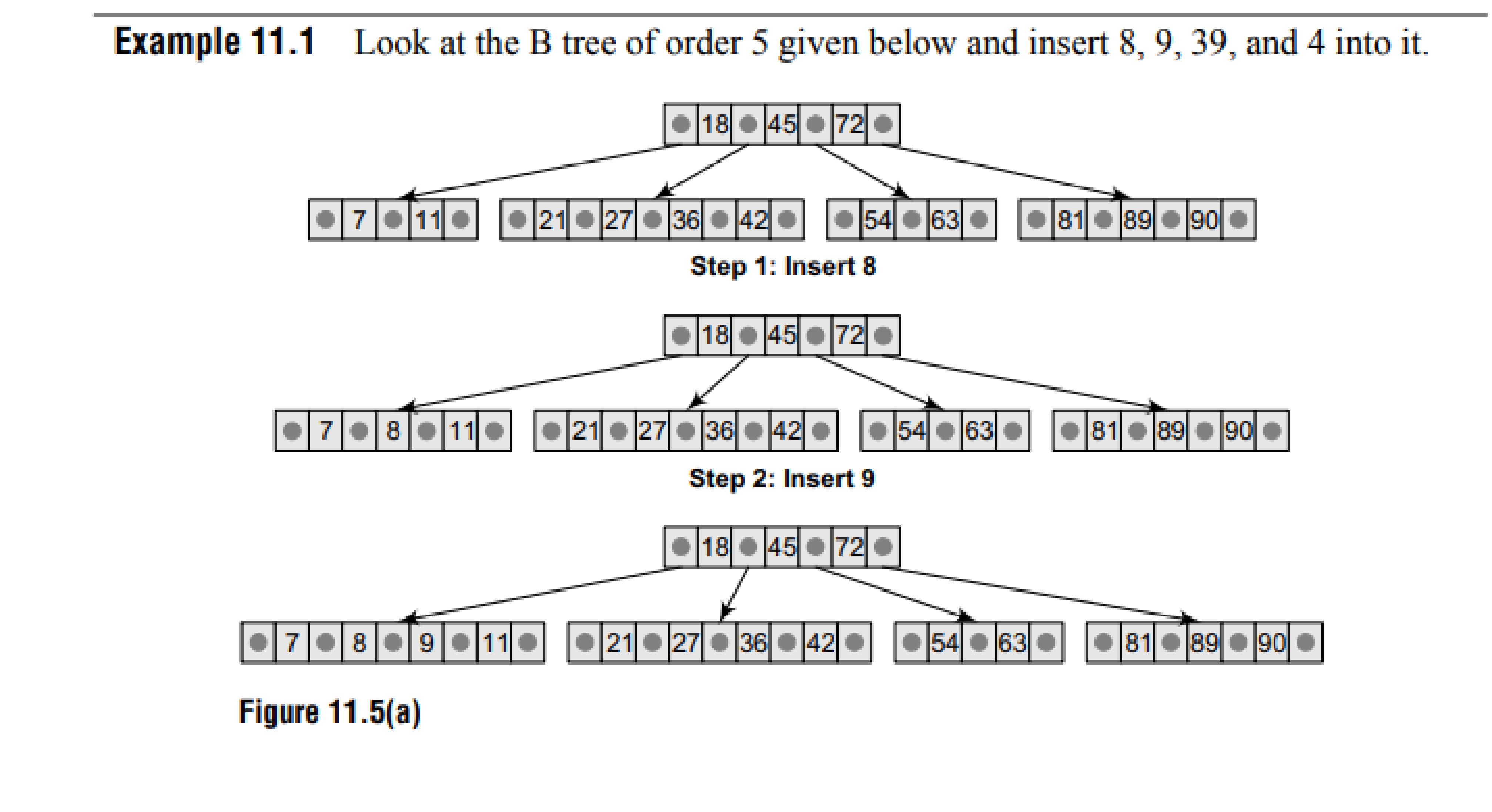
1. Search the B tree to find the leaf node where the new key value should be inserted.
2. If the leaf node is not full, that is, it contains less than `m-1` key values, then insert the new element in the node keeping the node's elements ordered.
3. If the leaf node is full, that is, the leaf node already contains `m-1` key values, then

   (a) insert the new value in order into the existing set of keys,

   (b) split the node at its median into two nodes (note that the split nodes are half full), and

   (c) push the median element up to its parent's node. If the parent's node is already full, then split the parent node by following the same steps.

B TREE INSERTION



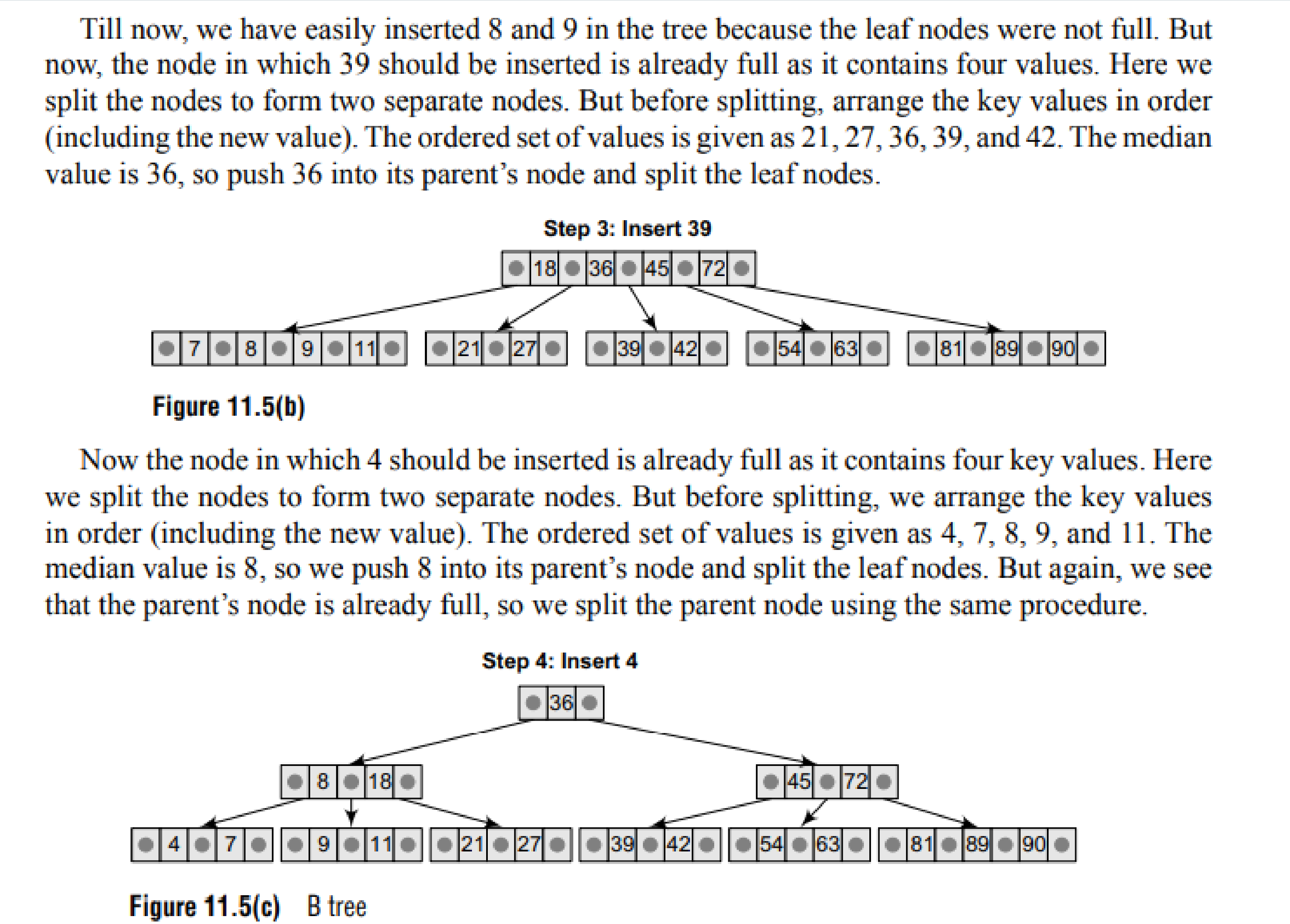
Example 11.1 Look at the B tree of order 5 given below and insert 8, 9, 39, and 4 into it.

Step 1: Insert 8

Step 2: Insert 9

Figure 11.5(a)

B TREE INSERTION



Till now, we have easily inserted 8 and 9 in the tree because the leaf nodes were not full. But now, the node in which 39 should be inserted is already full as it contains four values. Here we split the nodes to form two separate nodes. But before splitting, arrange the key values in order (including the new value). The ordered set of values is given as 21, 27, 36, 39, and 42. The median value is 36, so push 36 into its parent's node and split the leaf nodes.

Step 3: Insert 39

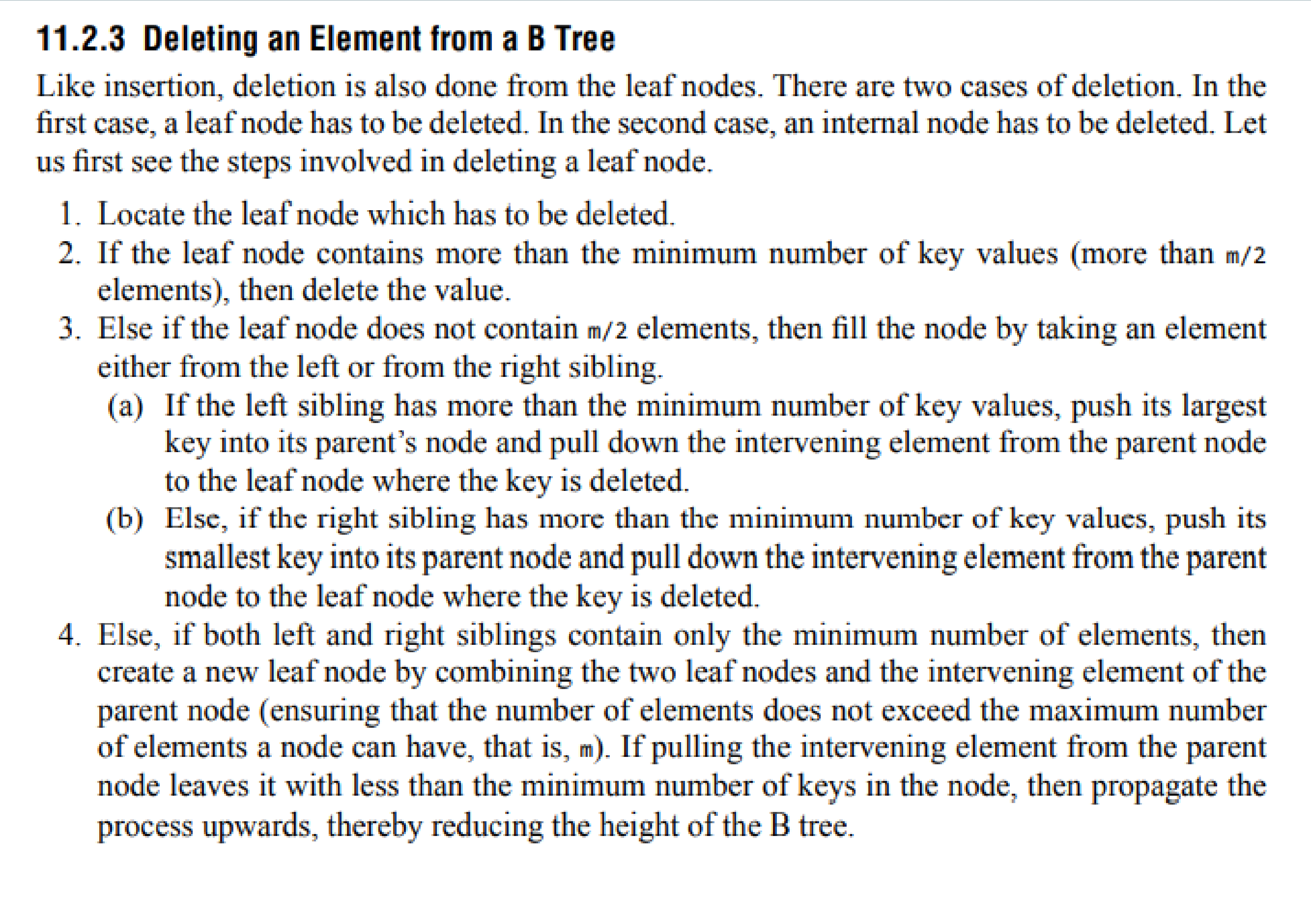
Figure 11.5(b)

Now the node in which 4 should be inserted is already full as it contains four key values. Here we split the nodes to form two separate nodes. But before splitting, we arrange the key values in order (including the new value). The ordered set of values is given as 4, 7, 8, 9, and 11. The median value is 8, so we push 8 into its parent's node and split the leaf nodes. But again, we see that the parent's node is already full, so we split the parent node using the same procedure.

Step 4: Insert 4

Figure 11.5(c) B tree

B TREE DELETION

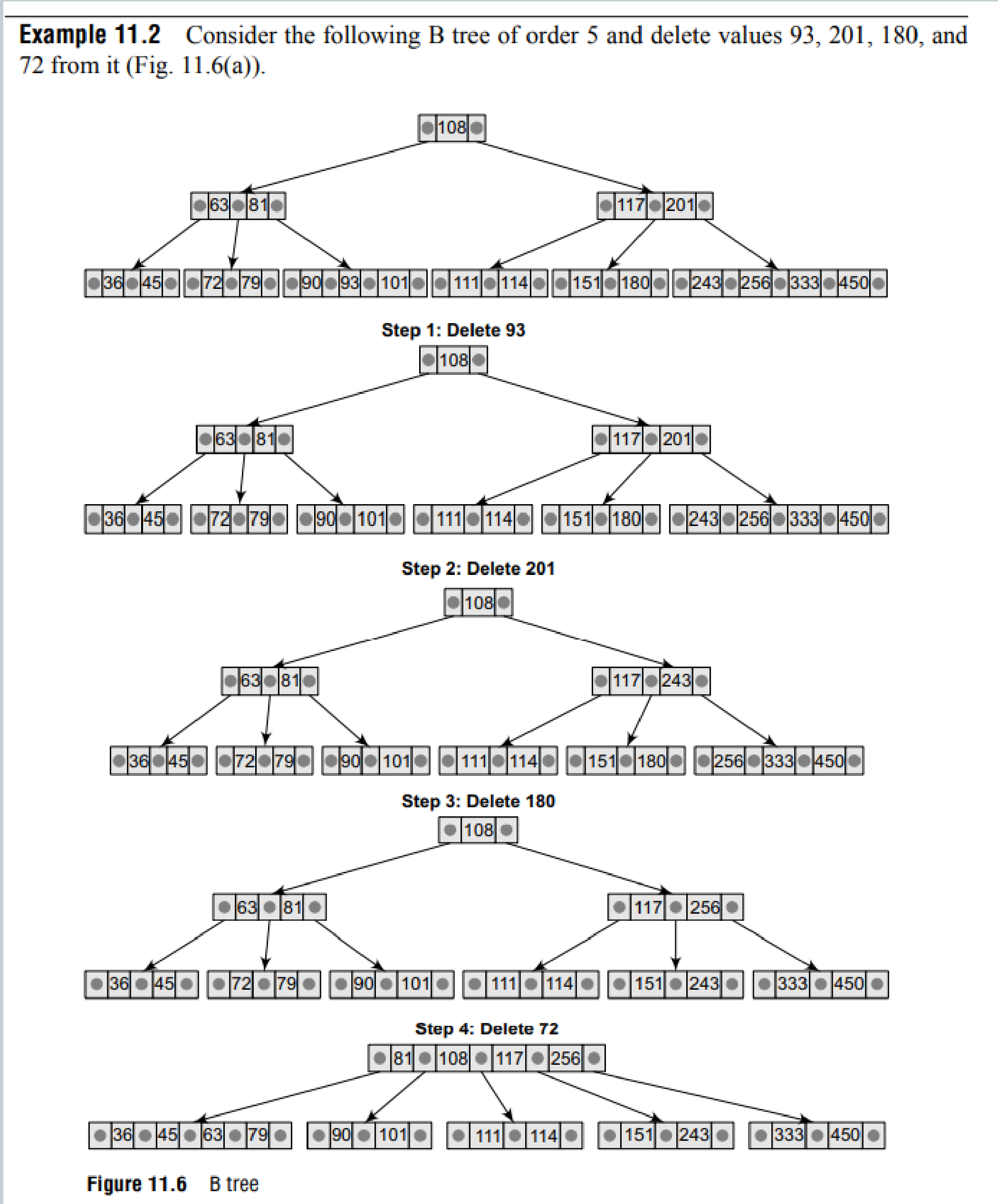


11.2.3 Deleting an Element from a B Tree

Like insertion, deletion is also done from the leaf nodes. There are two cases of deletion. In the first case, a leaf node has to be deleted. In the second case, an internal node has to be deleted. Let us first see the steps involved in deleting a leaf node.

1. Locate the leaf node which has to be deleted.
2. If the leaf node contains more than the minimum number of key values (more than `m/2` elements), then delete the value.
3. Else if the leaf node does not contain `m/2` elements, then fill the node by taking an element either from the left or from the right sibling.
   (a) If the left sibling has more than the minimum number of key values, push its largest key into its parent's node and pull down the intervening element from the parent node to the leaf node where the key is deleted.
   (b) Else, if the right sibling has more than the minimum number of key values, push its smallest key into its parent node and pull down the intervening element from the parent node to the leaf node where the key is deleted.
4. Else, if both left and right siblings contain only the minimum number of elements, then create a new leaf node by combining the two leaf nodes and the intervening element of the parent node (ensuring that the number of elements does not exceed the maximum number of elements a node can have, that is, `m`). If pulling the intervening element from the parent node leaves it with less than the minimum number of keys in the node, then propagate the process upwards, thereby reducing the height of the B tree.

B TREE DELETION



Example 11.2 Consider the following B tree of order 5 and delete values 93, 201, 180, and 72 from it (Fig. 11.6(a)).

Step 1: Delete 93

Step 2: Delete 201

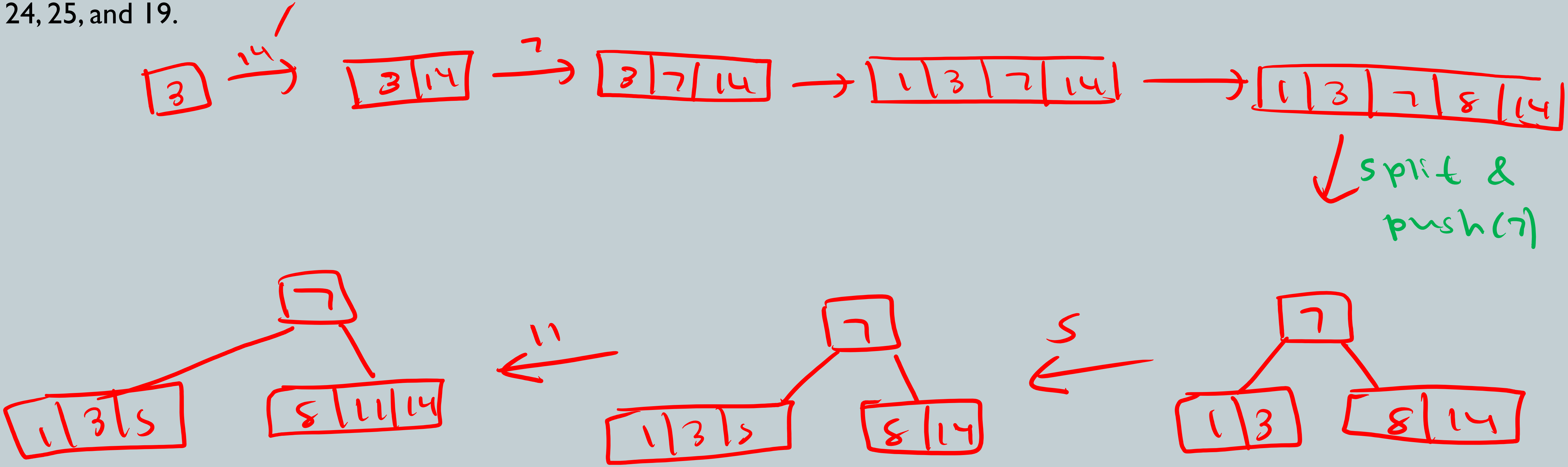
Step 3: Delete 180

Step 4: Delete 72

Figure 11.6 B tree

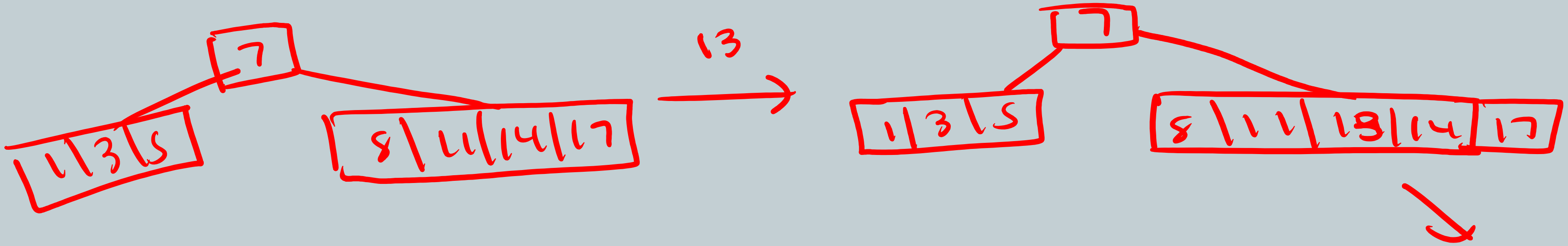
B TREE INSERTION EXAMPLE

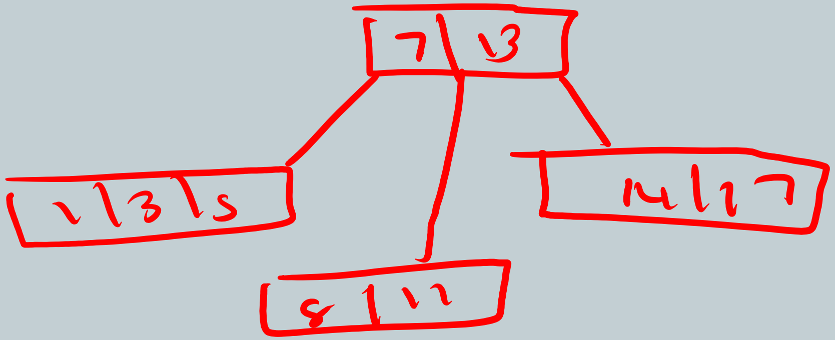
Create a B tree of order 5 by inserting the following elements: 3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25, and 19.



B TREE INSERTION EXAMPLE

Create a B tree of order 5 by inserting the following elements: 3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25, and 19.





keep inserting

facts:-

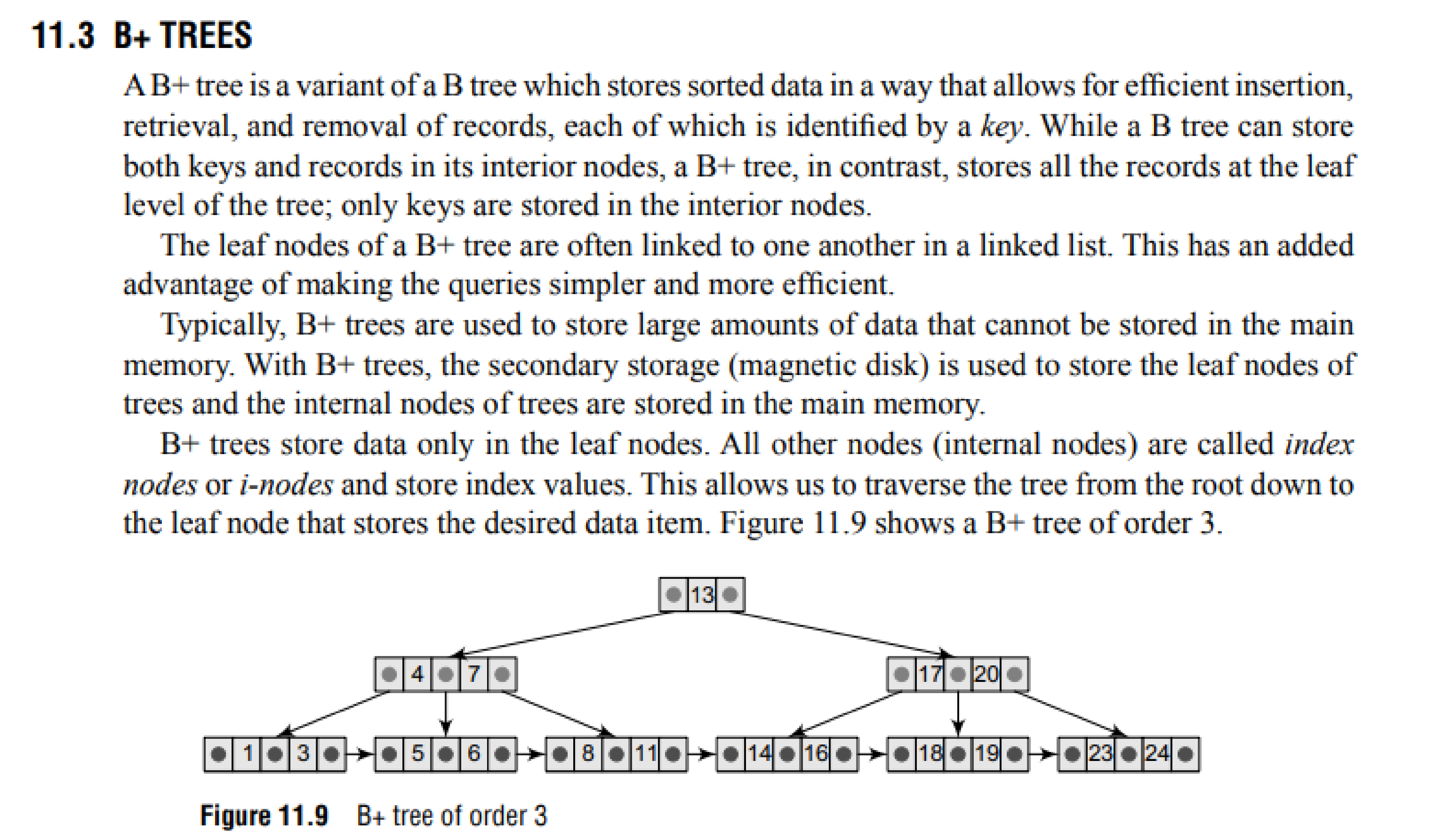
1) Order is 5 (cant exceed 4)

maximum key values per node

2) To keep the bushy level ($\frac{m}{2}$) minimum nodes should be there

B+ TREES

B+ TREES



11.3 B+ TREES

A B+ tree is a variant of a B tree which stores sorted data in a way that allows for efficient insertion, retrieval, and removal of records, each of which is identified by a *key*. While a B tree can store both keys and records in its interior nodes, a B+ tree, in contrast, stores all the records at the leaf level of the tree; only keys are stored in the interior nodes.

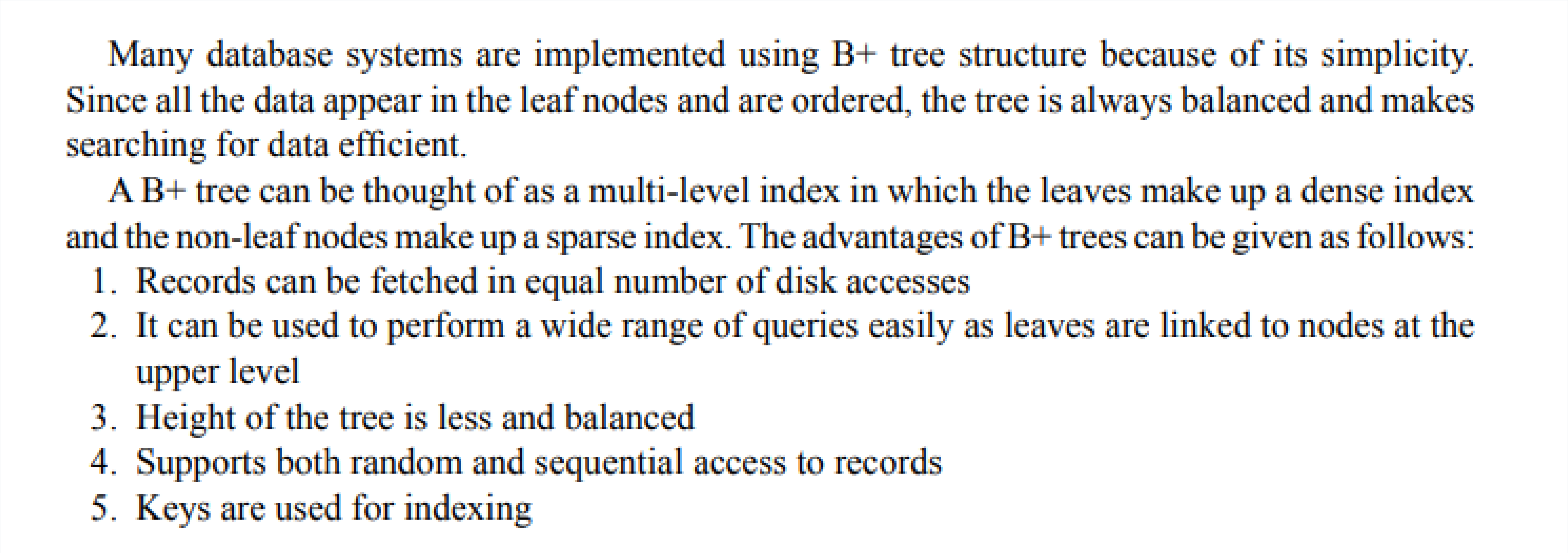
The leaf nodes of a B+ tree are often linked to one another in a linked list. This has an added advantage of making the queries simpler and more efficient.

Typically, B+ trees are used to store large amounts of data that cannot be stored in the main memory. With B+ trees, the secondary storage (magnetic disk) is used to store the leaf nodes of trees and the internal nodes of trees are stored in the main memory.

B+ trees store data only in the leaf nodes. All other nodes (internal nodes) are called *index nodes* or *i-nodes* and store index values. This allows us to traverse the tree from the root down to the leaf node that stores the desired data item. Figure 11.9 shows a B+ tree of order 3.

Figure 11.9 B+ tree of order 3

B+ TREES

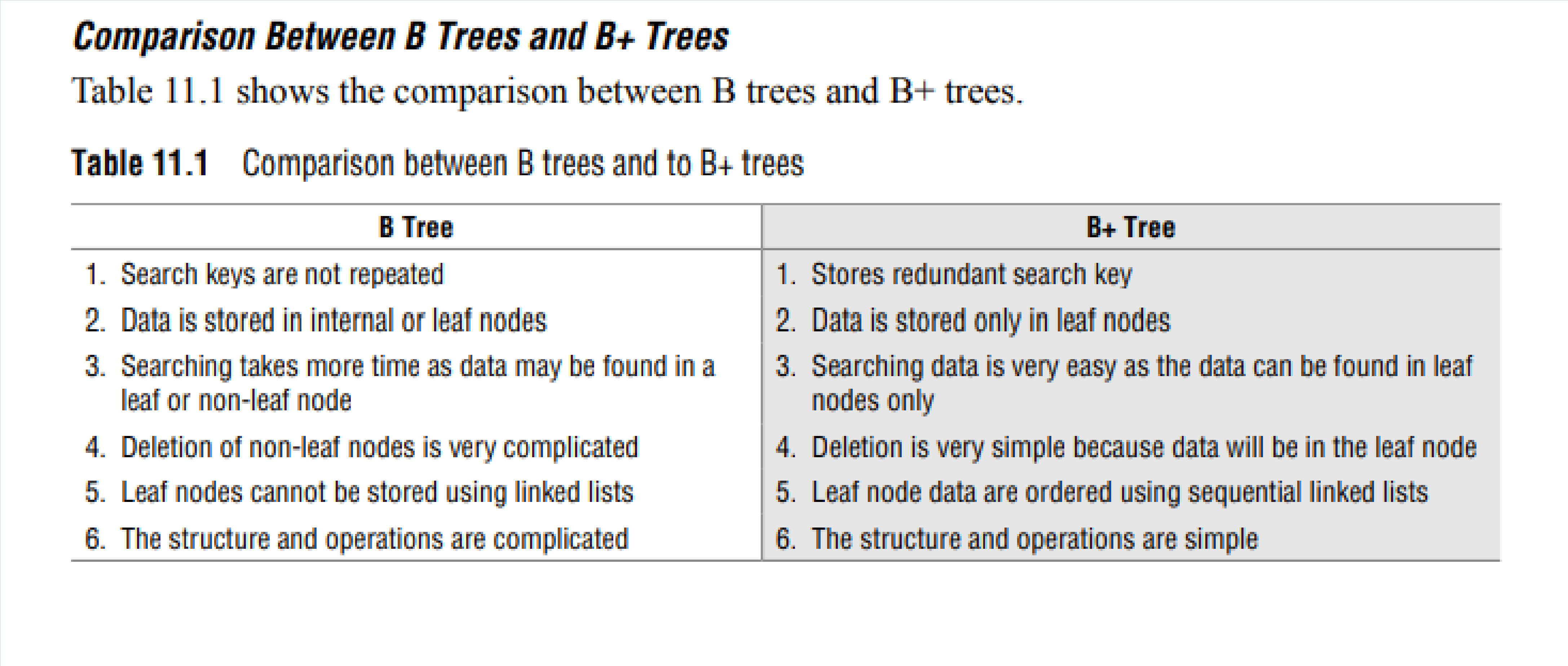


Many database systems are implemented using B+ tree structure because of its simplicity. Since all the data appear in the leaf nodes and are ordered, the tree is always balanced and makes searching for data efficient.

A B+ tree can be thought of as a multi-level index in which the leaves make up a dense index and the non-leaf nodes make up a sparse index. The advantages of B+ trees can be given as follows:

1. Records can be fetched in equal number of disk accesses
2. It can be used to perform a wide range of queries easily as leaves are linked to nodes at the upper level
3. Height of the tree is less and balanced
4. Supports both random and sequential access to records
5. Keys are used for indexing

B+ TREES



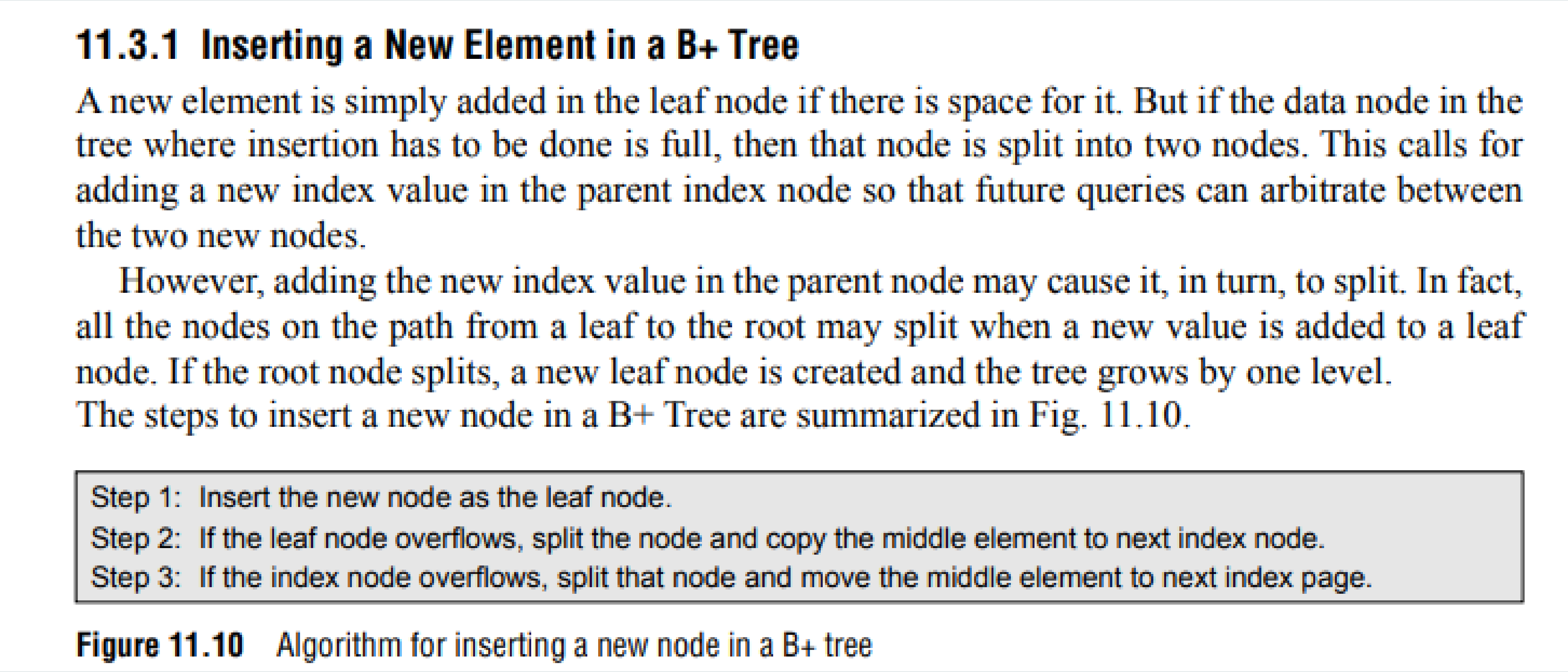
Comparison Between B Trees and B+ Trees

Table 11.1 shows the comparison between B trees and B+ trees.

Table 11.1 Comparison between B trees and to B+ trees

B Tree	B+ Tree
1. Search keys are not repeated	1. Stores redundant search key
2. Data is stored in internal or leaf nodes	2. Data is stored only in leaf nodes
3. Searching takes more time as data may be found in a leaf or non-leaf node	3. Searching data is very easy as the data can be found in leaf nodes only
4. Deletion of non-leaf nodes is very complicated	4. Deletion is very simple because data will be in the leaf node
5. Leaf nodes cannot be stored using linked lists	5. Leaf node data are ordered using sequential linked lists
6. The structure and operations are complicated	6. The structure and operations are simple

B+ TREE INSERTION



11.3.1 Inserting a New Element in a B+ Tree

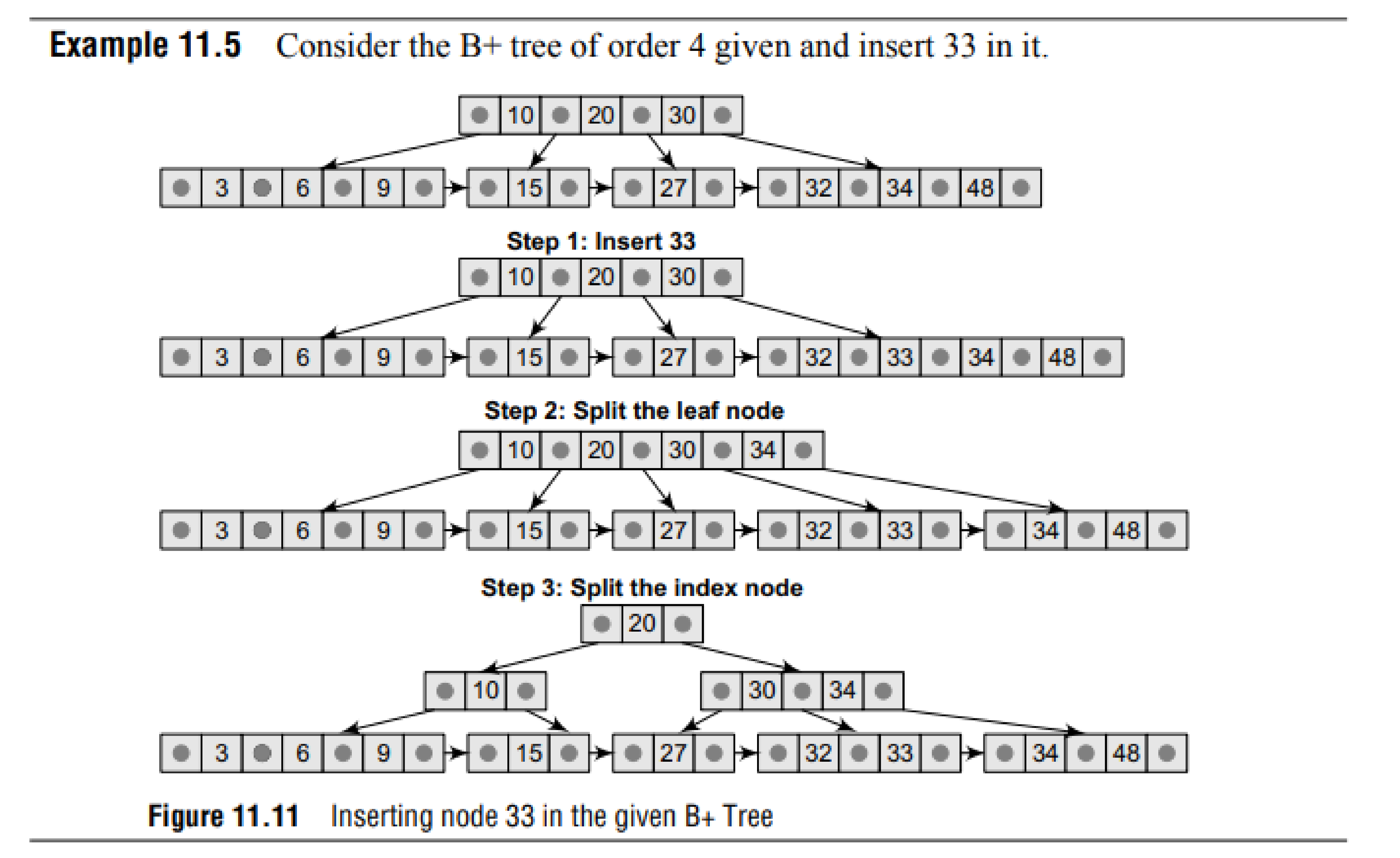
A new element is simply added in the leaf node if there is space for it. But if the data node in the tree where insertion has to be done is full, then that node is split into two nodes. This calls for adding a new index value in the parent index node so that future queries can arbitrate between the two new nodes.

However, adding the new index value in the parent node may cause it, in turn, to split. In fact, all the nodes on the path from a leaf to the root may split when a new value is added to a leaf node. If the root node splits, a new leaf node is created and the tree grows by one level.
The steps to insert a new node in a B+ Tree are summarized in Fig. 11.10.

```
Step 1:  Insert the new node as the leaf node.
Step 2:  If the leaf node overflows, split the node and copy the middle element to next index node.
Step 3:  If the index node overflows, split that node and move the middle element to next index page.
```

Figure 11.10 Algorithm for inserting a new node in a B+ tree

B+ TREE INSERTION



Example 11.5 Consider the B+ tree of order 4 given and insert 33 in it.

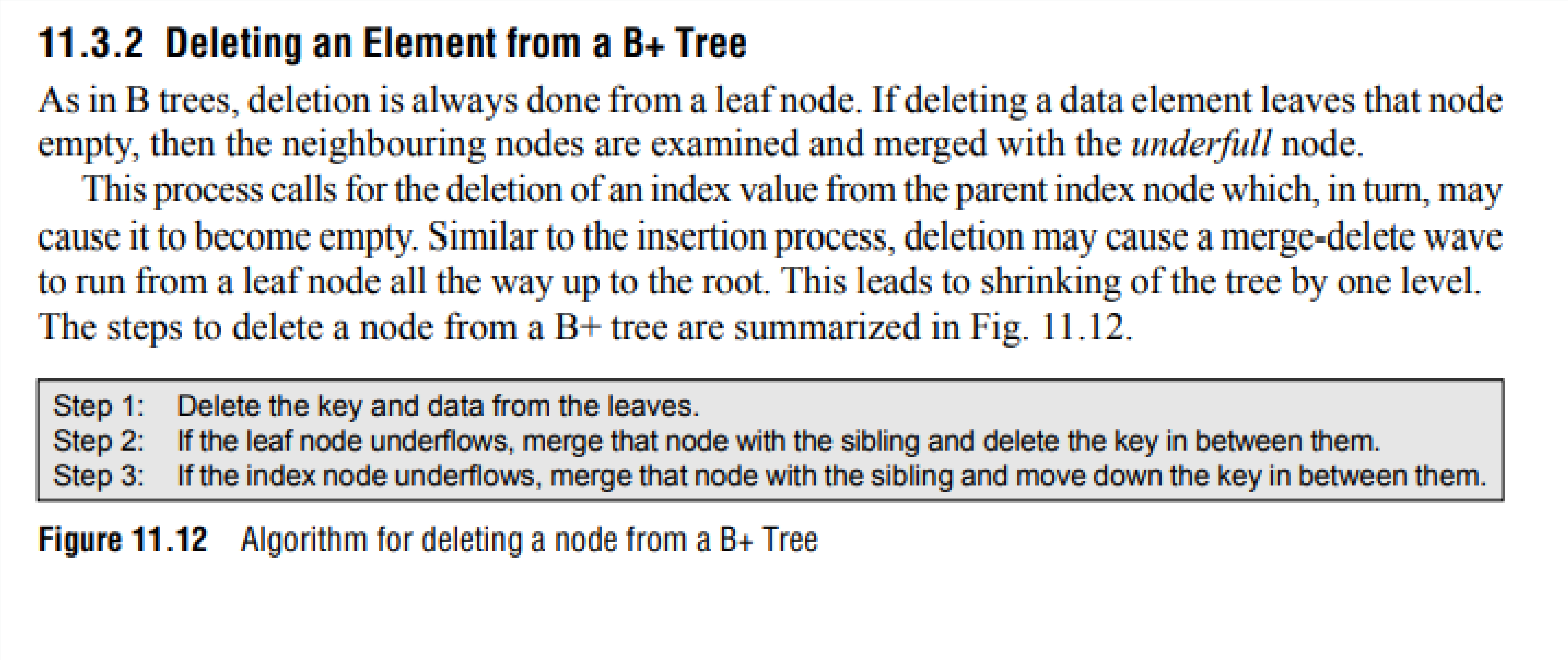
Step 1: Insert 33

Step 2: Split the leaf node

Step 3: Split the index node

Figure 11.11 Inserting node 33 in the given B+ Tree

B+ TREE DELETION



11.3.2 Deleting an Element from a B+ Tree

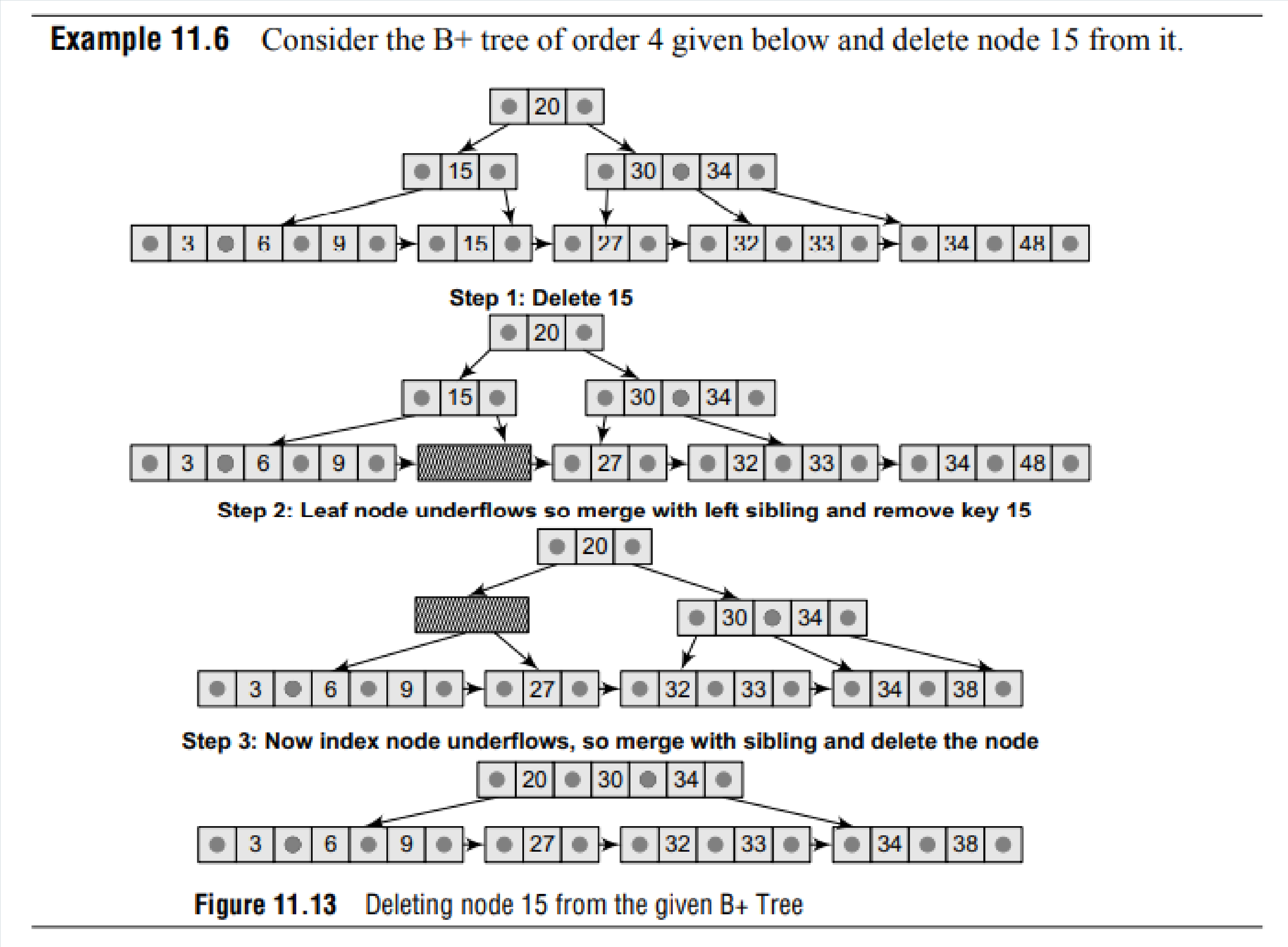
As in B trees, deletion is always done from a leaf node. If deleting a data element leaves that node empty, then the neighbouring nodes are examined and merged with the *underfull* node.

This process calls for the deletion of an index value from the parent index node which, in turn, may cause it to become empty. Similar to the insertion process, deletion may cause a merge-delete wave to run from a leaf node all the way up to the root. This leads to shrinking of the tree by one level. The steps to delete a node from a B+ tree are summarized in Fig. 11.12.

```
Step 1:   Delete the key and data from the leaves.
Step 2:   If the leaf node underflows, merge that node with the sibling and delete the key in between them.
Step 3:   If the index node underflows, merge that node with the sibling and move down the key in between them.
```

Figure 11.12 Algorithm for deleting a node from a B+ Tree

B+ TREE DELETION



Example 11.6 Consider the B+ tree of order 4 given below and delete node 15 from it.

Step 1: Delete 15

Step 2: Leaf node underflows so merge with left sibling and remove key 15

Step 3: Now index node underflows, so merge with sibling and delete the node

Figure 11.13 Deleting node 15 from the given B+ Tree

THANK YOU