MET's Institute of Engineering

Bhujbal Knowledge City, Adgaon, Nashik.

Department of Computer Engineering

"SEARCH TREE"

Prepared By

Prof. Anand N. Gharu

(Assistant Professor)

Computer Dept.

CLASS : SE COMPUTER 2019

SUBJECT : DSA (SEM-II)

UNIT : IV

15 March 2024

SYLLABUS

Symbol Table-Representation of Symbol Tables- Static tree table and Dynamic tree table, Weight balanced tree - Optimal Binary Search Tree (OBST), OBST as an example of Dynamic Programming, Height Balanced Tree- AVL tree. Red-Black Tree, AA tree, K-dimensional tree, Splay Tree

UNIT-IV

SEARCH TREE

SYMBOL TABLE

SYMBOL TABLE MEANS

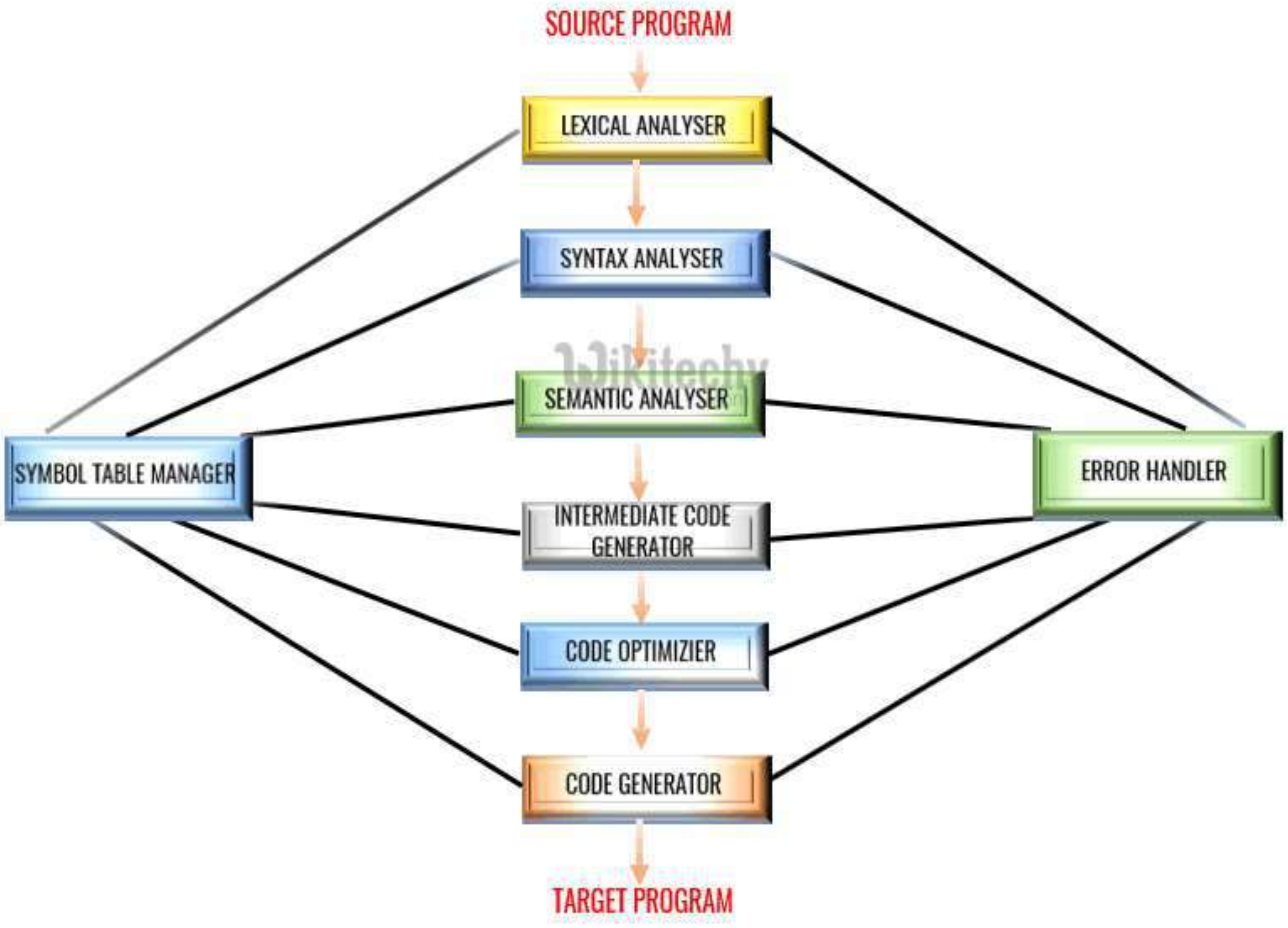
*Symbol tables are **data structures** that are used by compilers to hold information about source-program constructs.*

Asymbol table is a necessary component because

- Declaration of identifiers appears once in a program
- Use of identifiers may appear in many places of the program text

Example : https://youtu.be/P1bQUyl__t0

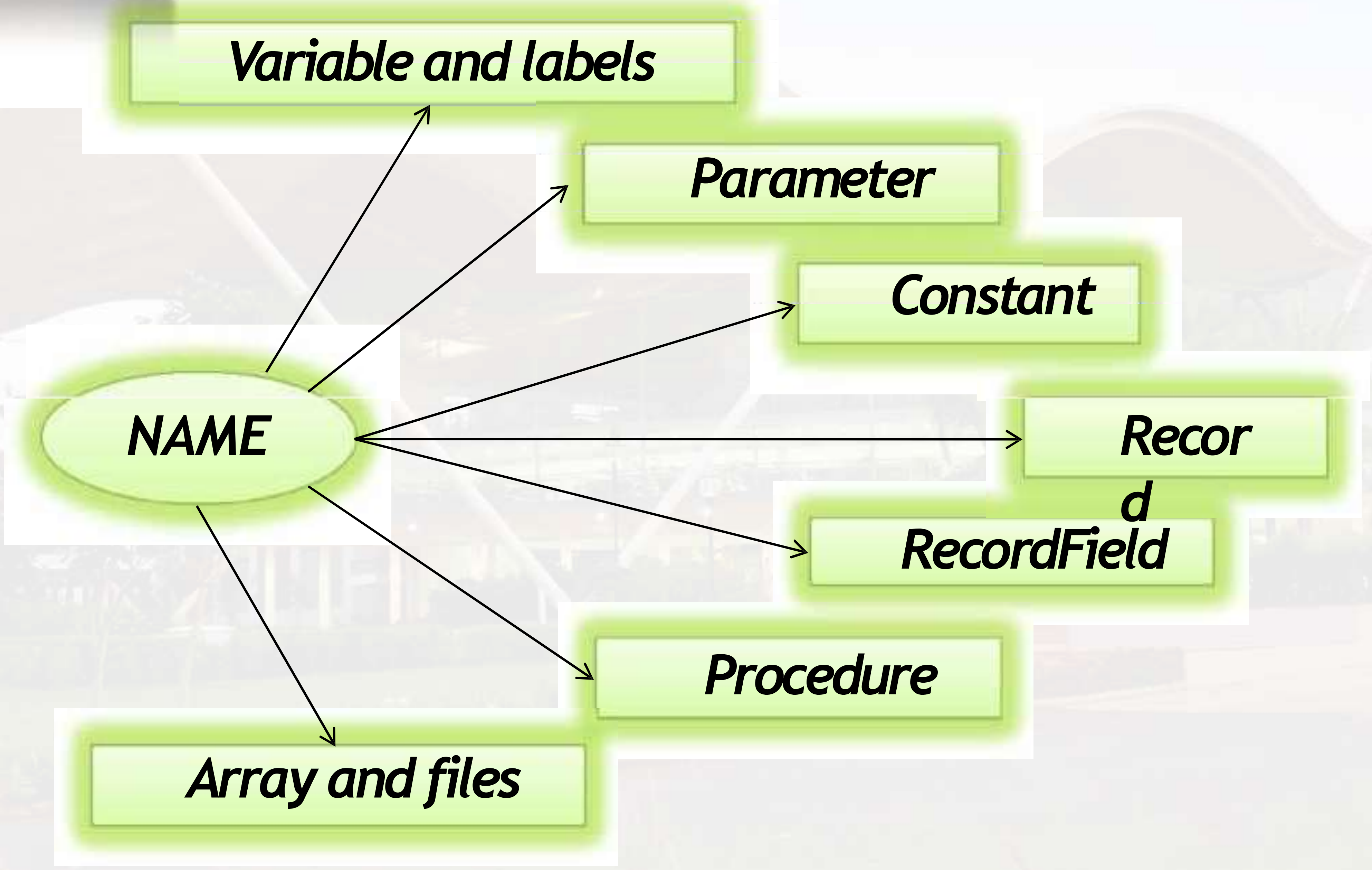
PHASES OF COMPILER



IN FORMATION PROVIDED BY SYMBOL TABLE

- *Given an Identifier which name isit?*
- *What information is to be associated with a name?*
- *How do we access thisinformation?*

SYMBOL TABLE NAMES



WHO CREATES SYMBOL TABLE ?

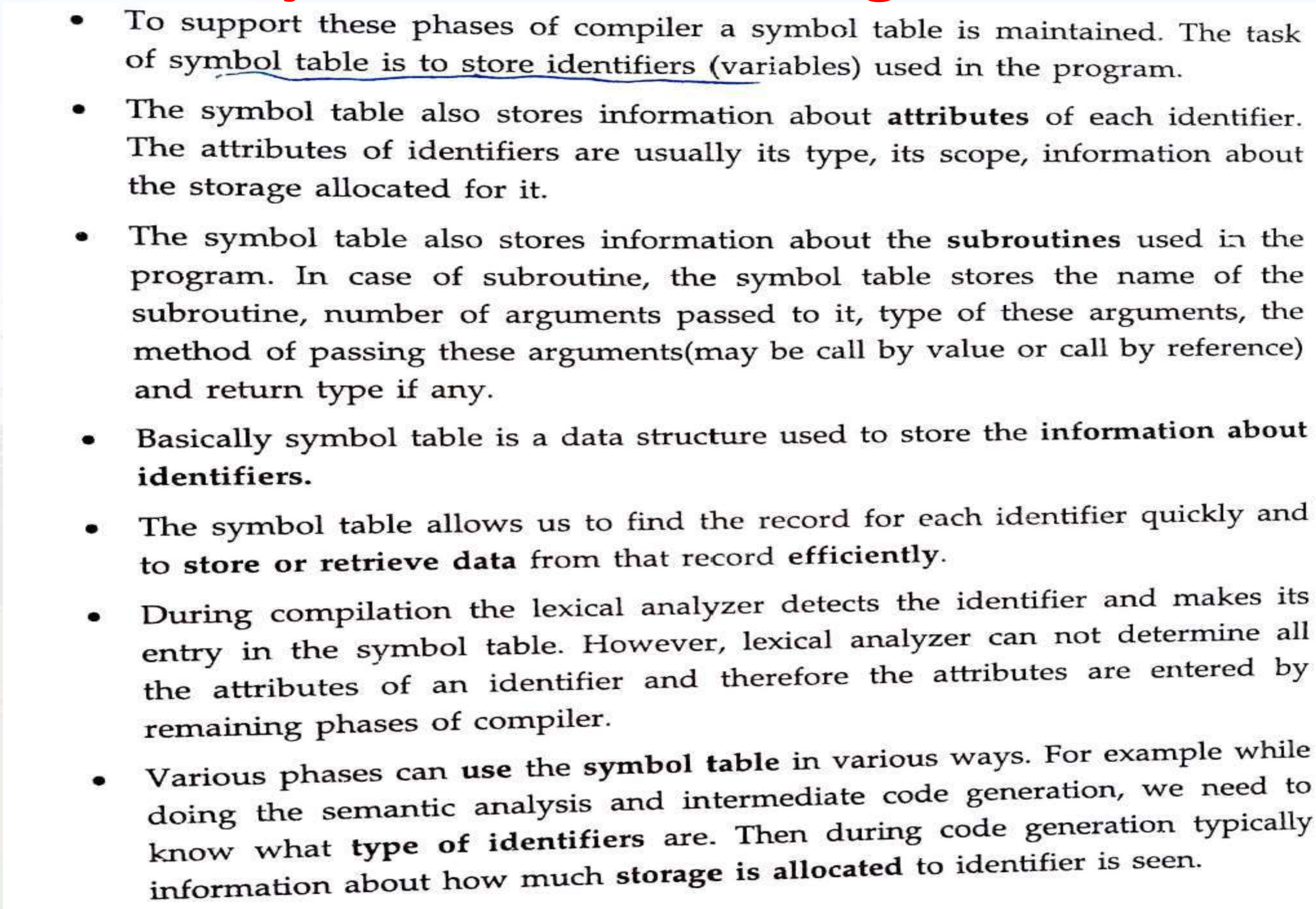
identifiers and attributes are entered by the analysis phases when processing a definition (declaration) of an identifier

- In simple languages with only global variables and implicit declarations:
  - ✓The scanner can enter an identifier into a symbol table if it is not alreadythere
- In block-structured languages with scopes and explicit declarations:
  - ✓ The parser and/or semantic analyzer enter identifiers and corresponding attributes

USE OF SYMBOL TABLE

- Symbol table information is used by the analysis and synthesis phases
- To verify that used identifiers have been defined (declared)
- To verify that expressions assignments and semantically correct – type ar checking e
- To generate intermediate or target code

Symbol Table Management



- To support these phases of compiler a symbol table is maintained. The task of symbol table is to store identifiers (variables) used in the program.
- The symbol table also stores information about **attributes** of each identifier. The attributes of identifiers are usually its type, its scope, information about the storage allocated for it.
- The symbol table also stores information about the **subroutines** used in the program. In case of subroutine, the symbol table stores the name of the subroutine, number of arguments passed to it, type of these arguments, the method of passing these arguments(may be call by value or call by reference) and return type if any.
- Basically symbol table is a data structure used to store the **information about identifiers.**
- The symbol table allows us to find the record for each identifier quickly and to **store or retrieve data** from that record **efficiently**.
- During compilation the lexical analyzer detects the identifier and makes its entry in the symbol table. However, lexical analyzer can not determine all the attributes of an identifier and therefore the attributes are entered by remaining phases of compiler.
- Various phases can **use** the **symbol table** in various ways. For example while doing the semantic analysis and intermediate code generation, we need to know what **type of identifiers** are. Then during code generation typically information about how much **storage is allocated** to identifier is seen.

SYMBOL TABLE

“Symbol table is an important data structure used in a compiler”

Symbol table is used to store the information about the occurrence of various entities such as objects, classes, variable name, interface, function name etc. it is used by both the analysis and synthesis phases.

The symbol table used for following purposes:

1. It is used to store the name of all entities in a structured form at one place.
2. It is used to verify if a variable has been declared.
3. It is used to determine the scope of a name.
4. It is used to implement type checking by verifying assignments and expressions in the source code are semantically correct.

SYMBOL TABLE

A symbol table can either be linear or a hash table. Using the following format, it maintains the entry for each name.

<symbol name, type, attribute>

For example, suppose a variable store the information about the following variable declaration:

static int salary

SYMBOL TABLE

Implementation :

The symbol table can be implemented in the unordered list if the compiler is used to handle the small amount of data.

A symbol table can be implemented in one of the following techniques:

1. Linear (sorted or unsorted) list
2. Hash table
3. Binary search tree

Symbol table are mostly implemented as hash table.

SYMBOL TABLE

OPERATION OF SYMBOL TABLE :

1. Insert ()

- Insert () operation is more frequently used in the analysis phase when the tokens are identified and names are stored in the table.
- The insert() operation is used to insert the information in the symbol table like the unique name occurring in the source code.

SYMBOL TABLE

OPERATION OF SYMBOL TABLE :

1. Insert ()

- In the source code, the attribute for a symbol is the information associated with that symbol. The information contains the state, value, type and scope about the symbol.
- The insert () function takes the symbol and its value in the form of argument.

For example:

int x

insert (x, int)

SYMBOL TABLE

OPERATION OF SYMBOL TABLE :

lookup()

In the symbol table, lookup() operation is used to search a name. It is used to determine:

- The existence of symbol in the table.
- The declaration of the symbol before it is used.
- Check whether the name is used in the scope.
- Initialization of the symbol.
- Checking whether the name is declared multiple times.

The basic format of lookup() function is as follows:

lookup (symbol)

Classification of Symbol Table

Data structure for symbol table

A compiler contains two type of symbol table: **global symbol table and scope symbol table.**

Global symbol table can be accessed by all the procedures and scope symbol table.

The scope of a name and symbol table is arranged in the hierarchy structure as shown below:

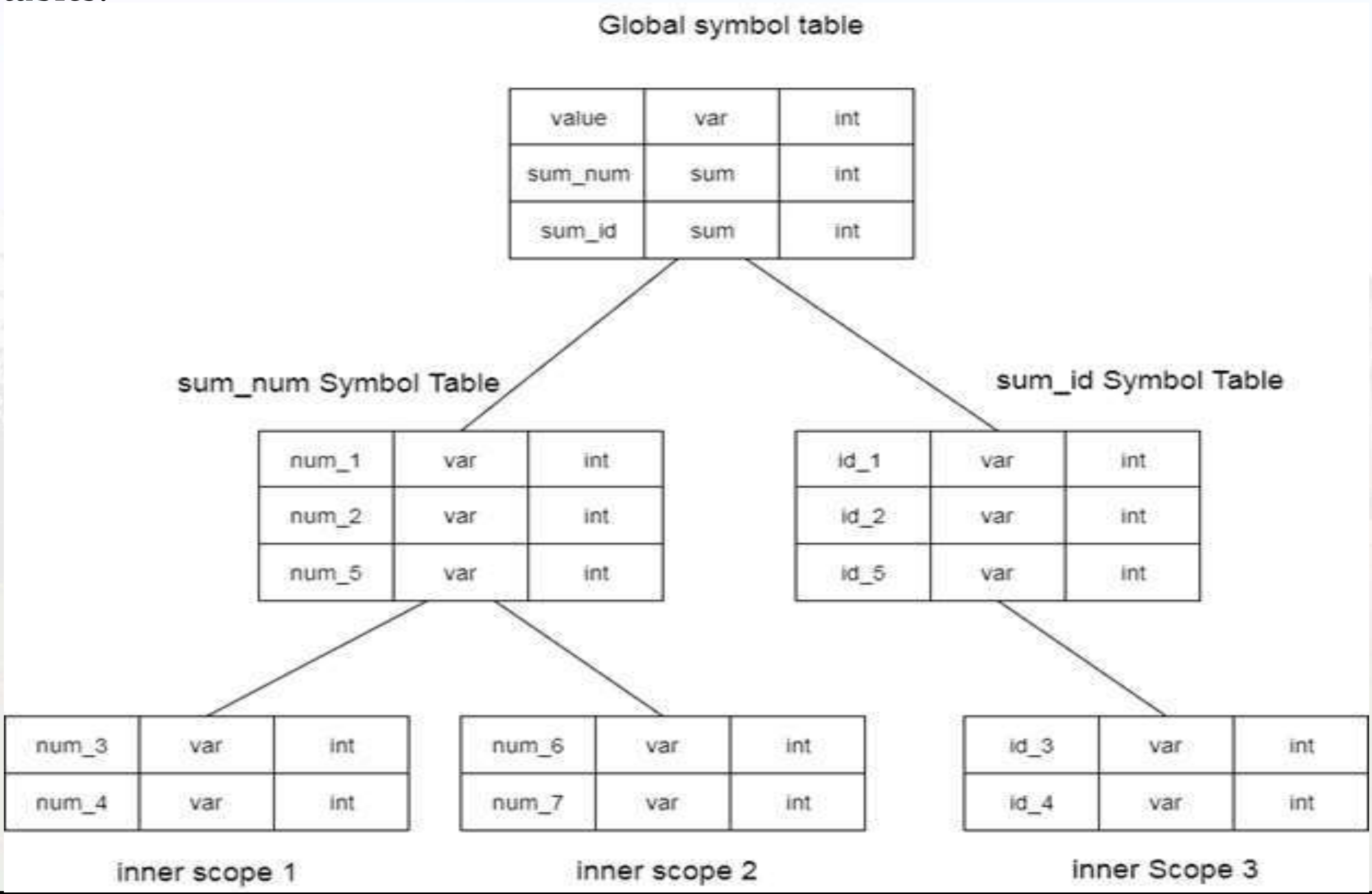
Classification of Symbol Table

```
int value=10;

void sum_num()
   {
     int num_1;
     int num_2;

        {
         int num_3;
         int num_4;
        }

     int num_5;

        {
         int_num 6;
         int_num 7;
        }
   }
```

```
Void sum_id

   {

     int id_1;

     int id_2;


        {

         int id_3;

         int id_4;

        }


     int num_5;

   } :
```

Classification of Symbol Table

The above grammar can be represented in a hierarchical data structure of symbol tables:



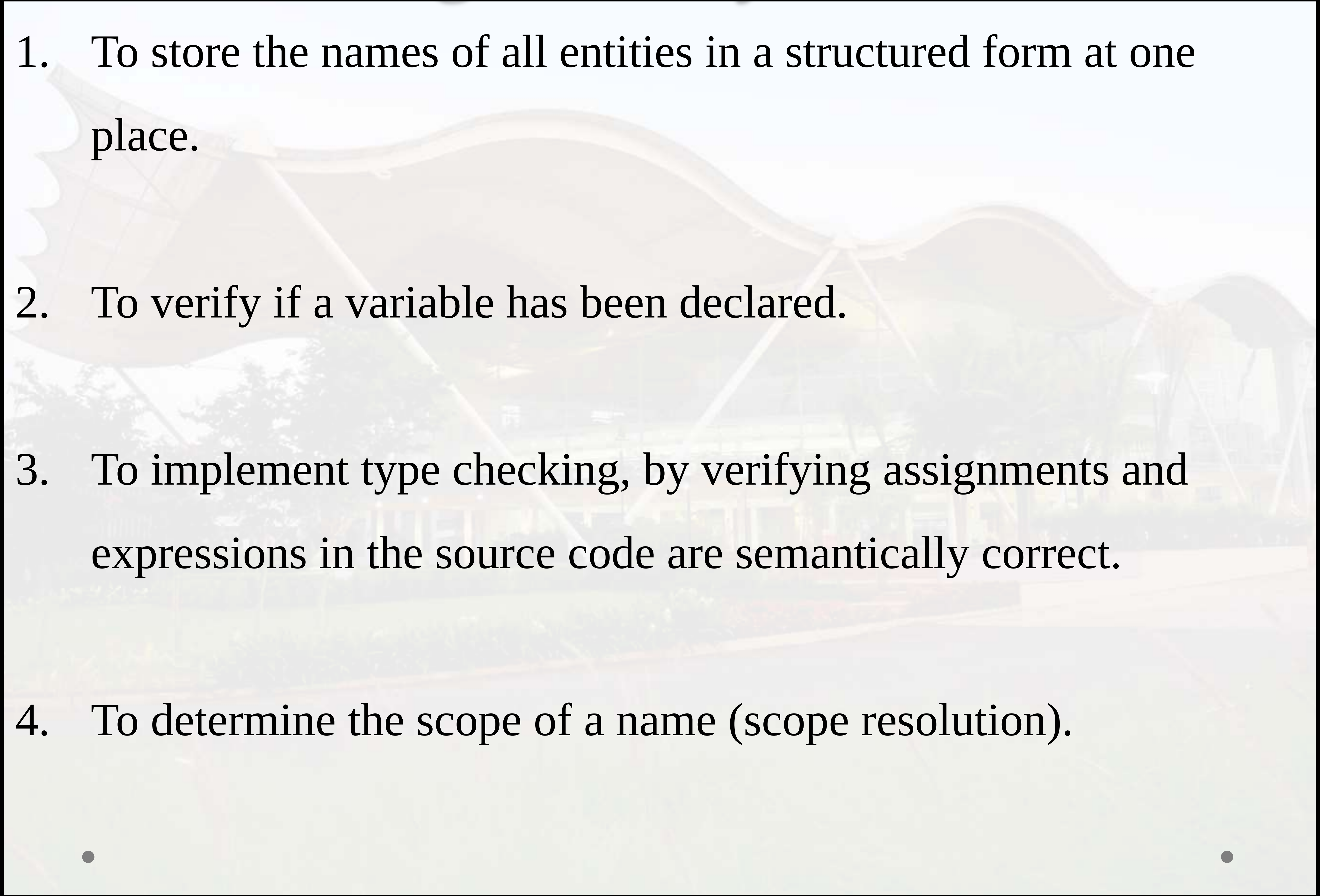
Classification of Symbol Table

The global symbol table contains one global variable and two procedure names. The name mentioned in the sum_num table is not available for sum_id and its child tables.

Data structure hierarchy of symbol table is stored in the semantic analyzer. If you want to search the name in the symbol table then you can search it using the following algorithm:

- First a symbol is searched in the current symbol table.
- If the name is found then search is completed else the name will be searched in the symbol table of parent until,
- The name is found or global symbol is searched

Advantages of Symbol Table



1. To store the names of all entities in a structured form at one place.
2. To verify if a variable has been declared.
3. To implement type checking, by verifying assignments and expressions in the source code are semantically correct.
4. To determine the scope of a name (scope resolution).

Types of Symbol Table

Data structures can be two types :

1. Static Tree table

2. Dynamic tree table

1. Static Tree Table :

In Static data structure the size of the structure is fixed. The content of the data structure can be modified but without changing the memory space allocated to it.

Example of Static tree tables are OBST, Huffman's Coding

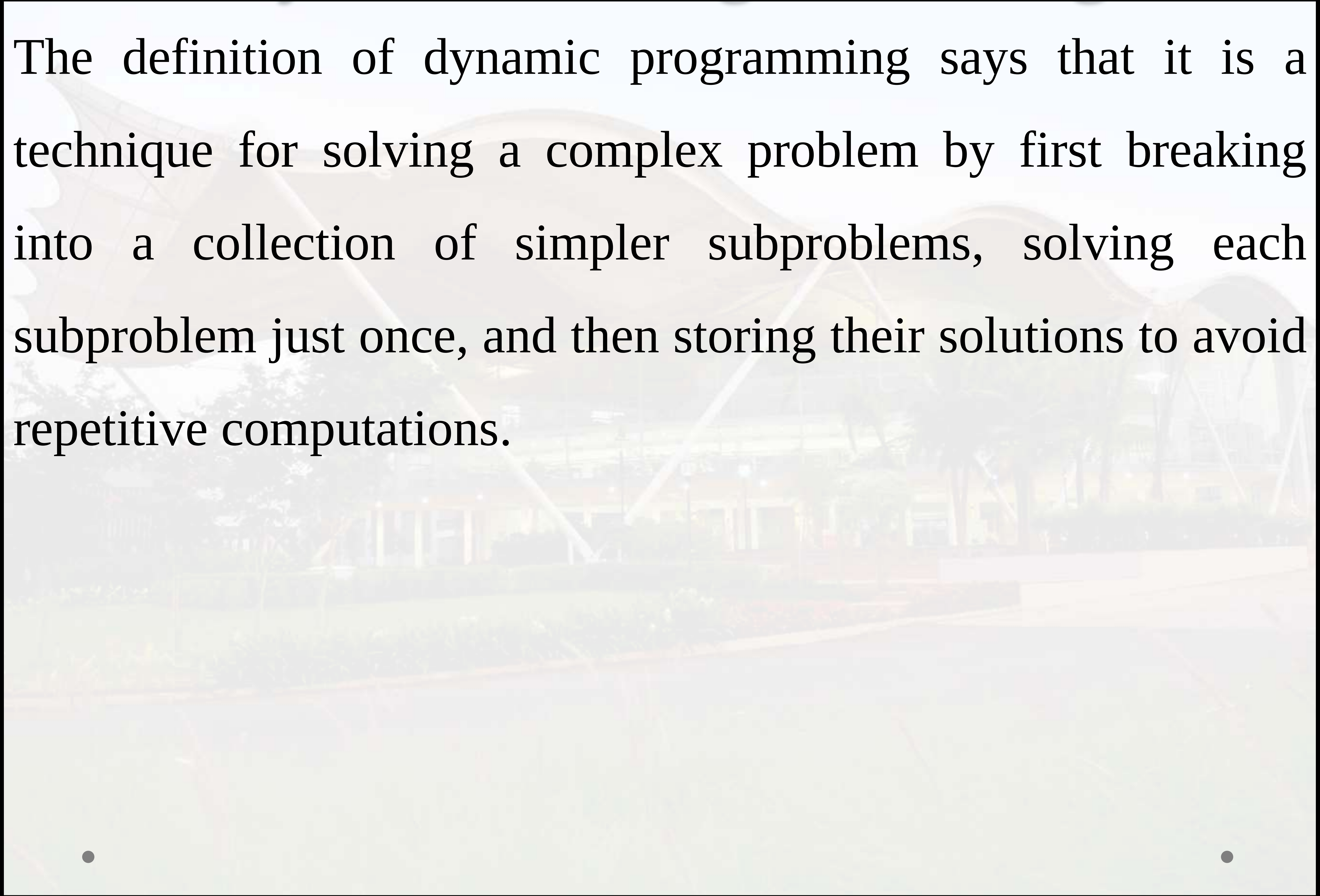
Types of Symbol Table

2**. Dynamic tree table :**

In Dynamic data structure the size of the structure in not fixed and can be modified during the operations performed on it. Dynamic data structures are designed to facilitate change of data structures in the run time.

Example of Dynamic tree table are an AVL Tree

Dynamic Programming

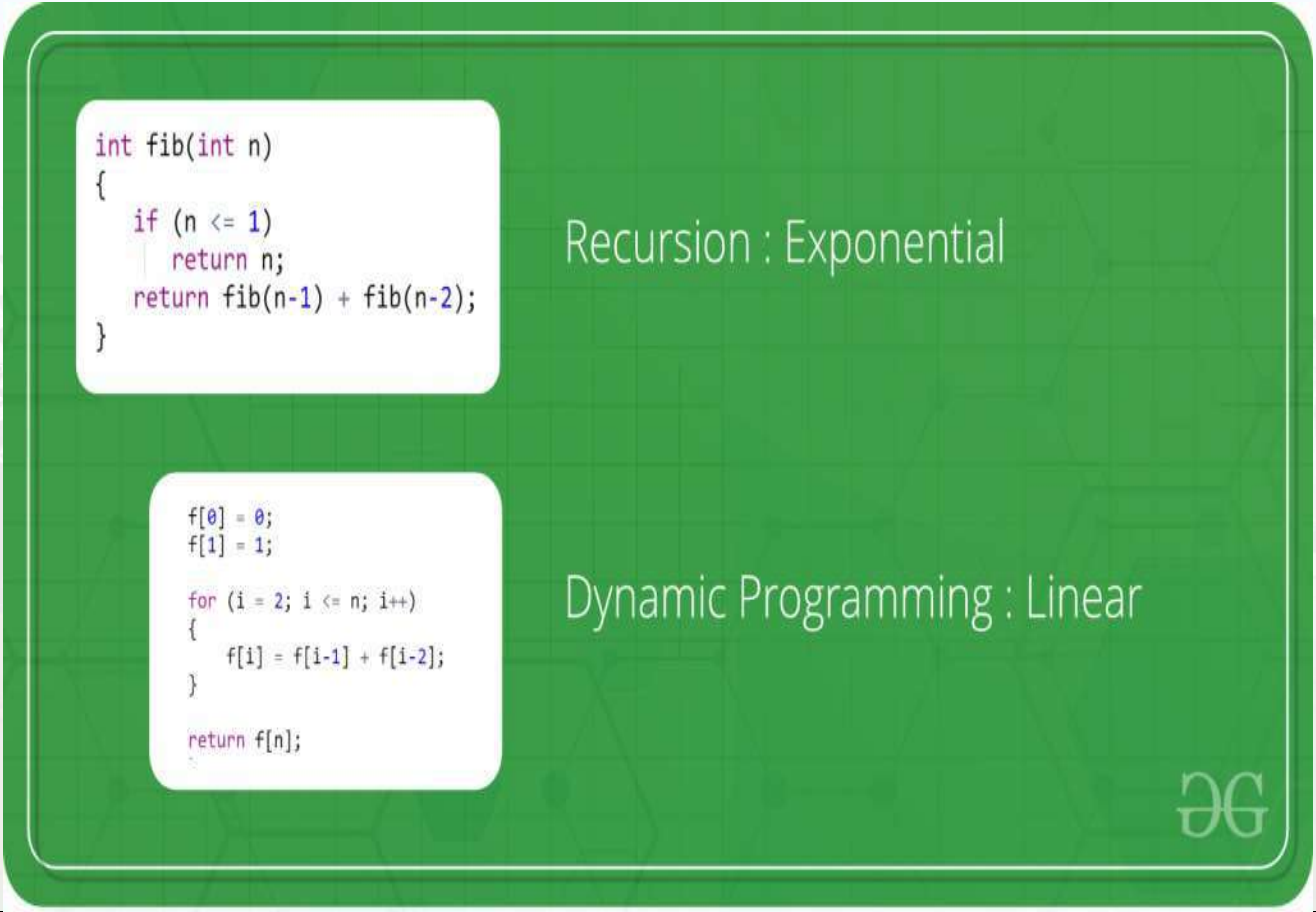


The definition of dynamic programming says that it is a technique for solving a complex problem by first breaking into a collection of simpler subproblems, solving each subproblem just once, and then storing their solutions to avoid repetitive computations.

Dynamic Programming

Dynamic Programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial. For example, if we write simple recursive solution for Fibonacci Numbers, we get exponential time complexity and if we optimize it by storing solutions of subproblems, time complexity reduces to linear.

Dynamic Programming



```
int fib(int n)
{
    if (n <= 1)
        return n;
    return fib(n-1) + fib(n-2);
}
```

Recursion : Exponential

```
f[0] = 0;
f[1] = 1;

for (i = 2; i <= n; i++)
{
    f[i] = f[i-1] + f[i-2];
}

return f[n];
```

Dynamic Programming : Linear

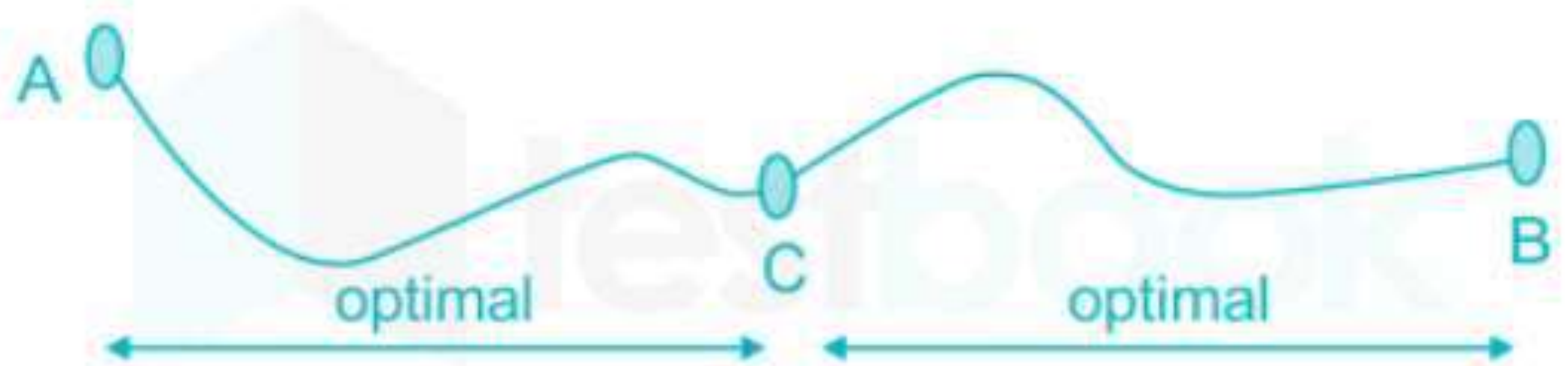
Dynamic Programming

Bellman's Principle of Optimality:

- An **optimal** policy has the property that whatever the **initial state and initial decision are**, the **remaining decisions must constitute** an **optimal** policy with regard to the state resulting from the **first decision**.
- **Dynamic Programming works on the principle of optimality.**
- **Principle of optimality** states that in an optimal **sequence of decisions or choices**, each **subsequences must also be optimal.**

If C belongs to an **optimal path** from **A to B**, then the **sub-path A to C** and **C to B** are also **optimal**. **OR**

All sub-path of an **optimal path is optimal**.



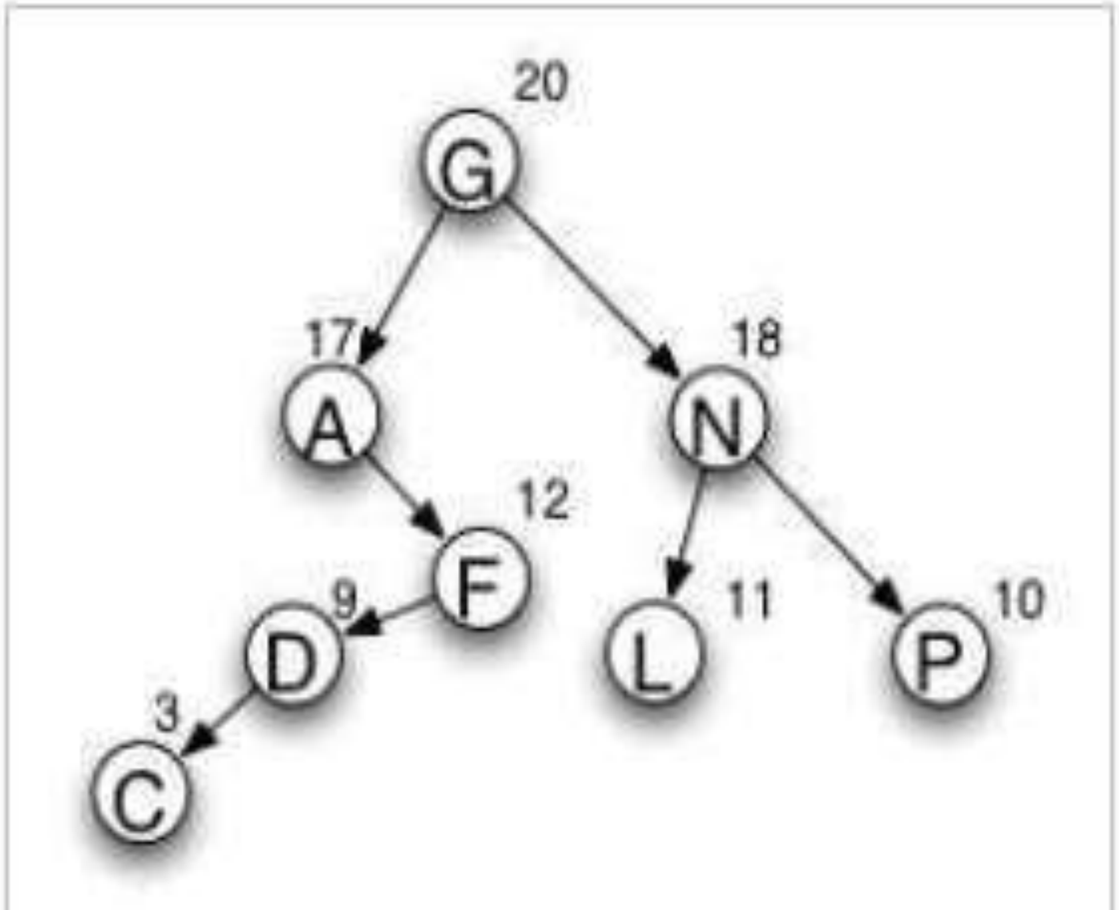
So, we can say that **Bellman's principle of optimality** is related to the **dynamic programming problem.**

Weight Balance Tree

1. A weight-balanced binary tree is a binary tree which is balanced based on knowledge of the probabilities of searching for each individual node.
2. Within each subtree, the node with the highest weight appears at the root.
3. This can result in more efficient searching performance.

Weight Balance Tree

In the diagram to the right, the letters represent node values and the numbers represent node weights. Values are used to order the tree, as in a general binary search tree. The weight may be thought of as a probability or activity count associated with the node. In the diagram, the root is G because its weight is the greatest in the tree. The left subtree begins with A because, out of all nodes with values that come before G, A has the highest weight. Similarly, N is the highest-weighted node that comes after G



Example of weight balanced tree

Optimal Binary Search Tree

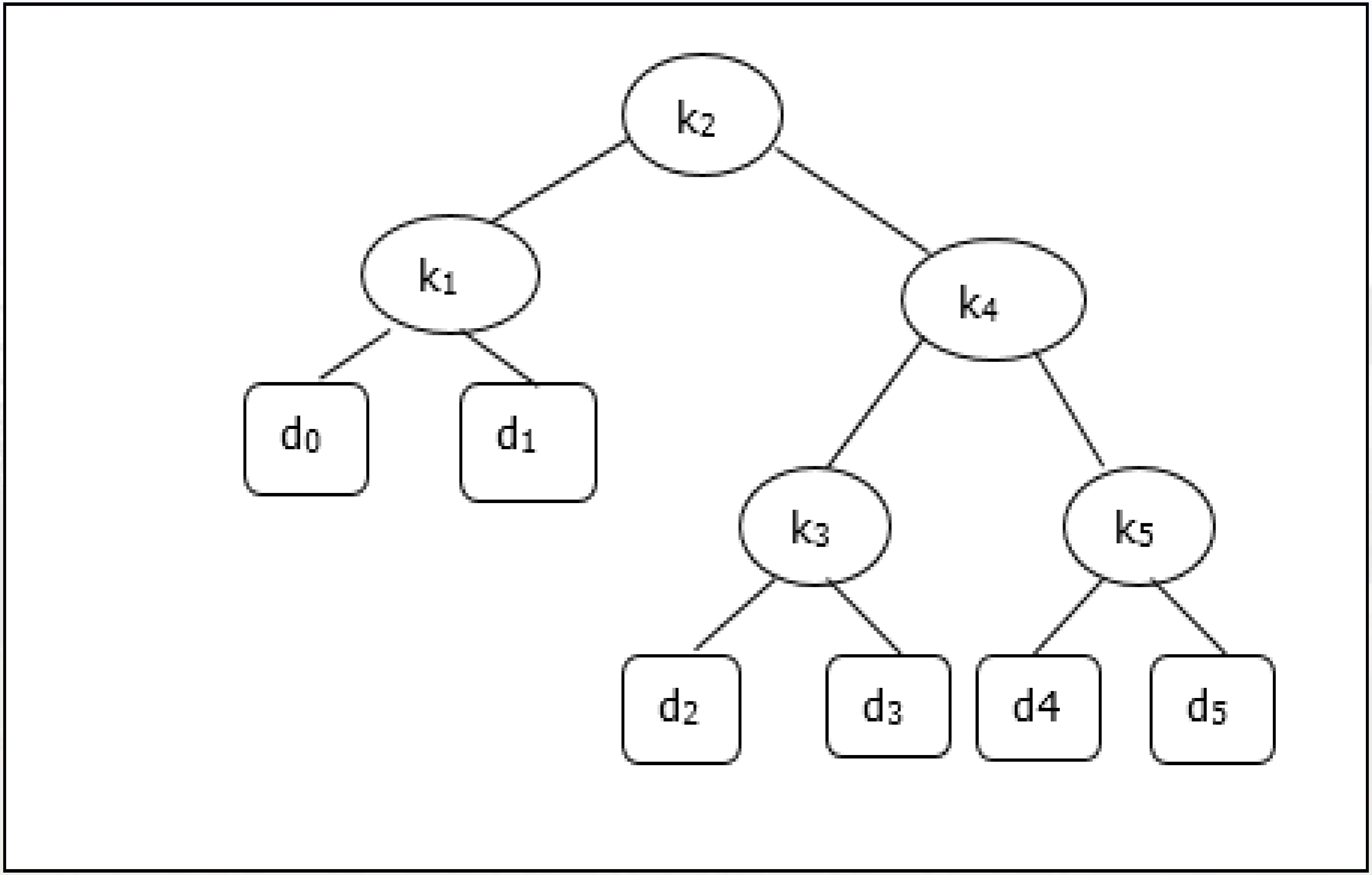
- As we know that in binary search tree, the nodes in the left subtree have lesser value than the root node and the nodes in the right subtree have greater value than the root node.

- We know the key values of each node in the tree, and we also know the frequencies of each node in terms of searching means how much time is required to search a node.
- The frequency and key-value determine the overall cost of searching a node. The cost of searching is a very important factor in various applications.

Optimal Binary Search Tree

- The overall cost of searching a node should be less.
- The time required to search a node in BST is more than the balanced binary search tree as a balanced binary search tree contains a lesser number of levels than the BST.
- There is one way that can reduce the cost of a binary search tree is known as an **optimal binary search tree.(OBST)**

Optimal Binary Search Tree

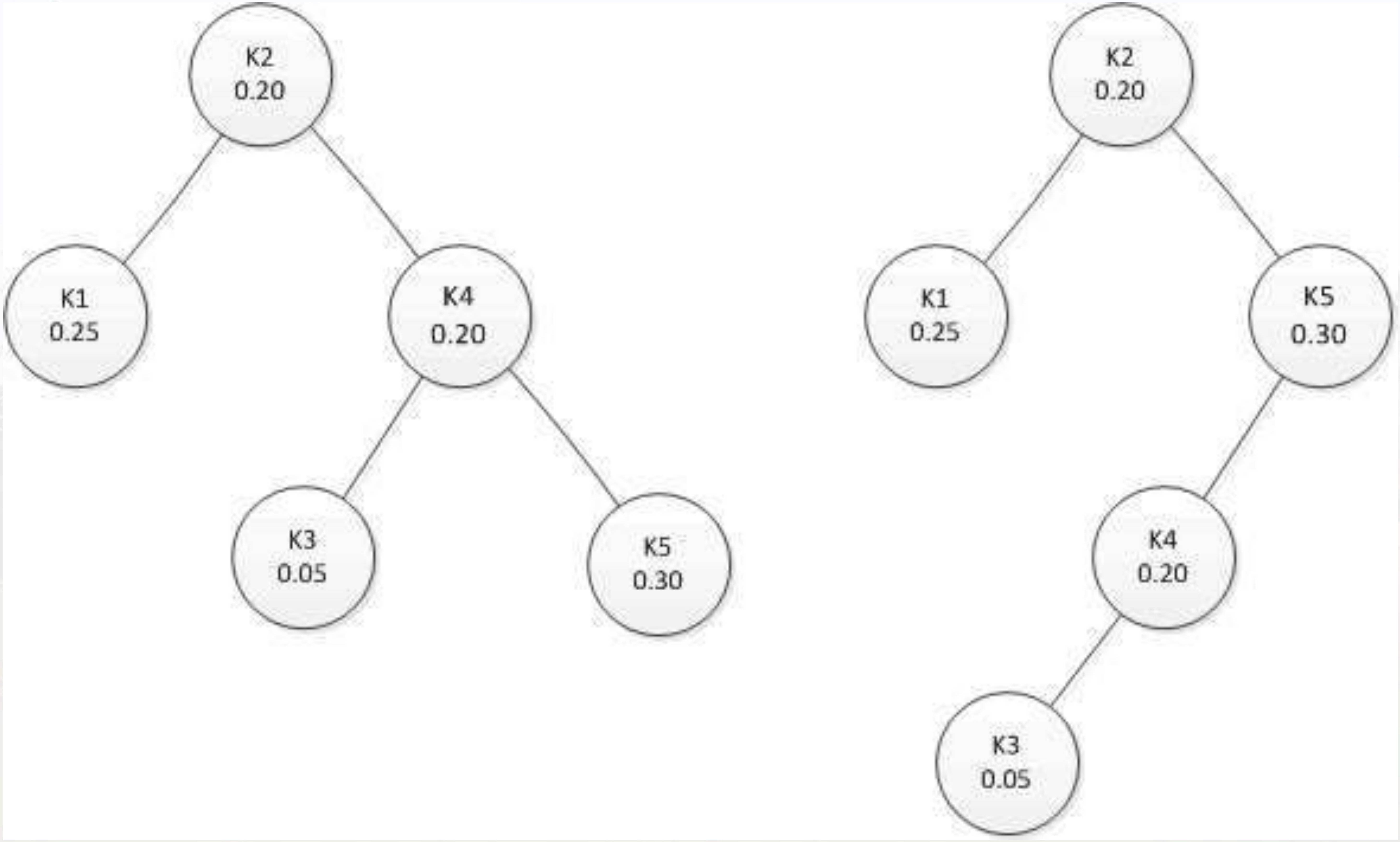


Optimal Binary Search Tree

- Problem:
  - Sorted set of keys $k_1, k_2, \ldots, k_n$
  - Key probabilities: $p_1, p_2, \ldots, p_n$
  - What tree structure has lowest expected cost?
  - Cost of searching for node i: $\text{cost}(k_i) = \text{depth}(k_i) + 1$

$$\begin{aligned}
\text{Expected Cost of tree} &= \sum_{i=1}^{n} \text{cost}(k_i) p_i \\
&= \sum_{i=1}^{n} (\text{depth}(k_i) + 1) p_i \\
&= \sum_{i=1}^{n} \text{depth}(k_i) p_i + \sum_{i=1}^{n} p_i \\
&= \left(\sum_{i=1}^{n} \text{depth}(k_i) p_i \right) + 1
\end{aligned}$$

Optimal Binary Search Tree



Optimal Binary Search Tree

- Given: k1<k2<k3<k4<k5

Tree 1:

k2/[k1,k4]/[nil,nil],[k3,k5]

cost = 0(0.20) + 1(0.25+0.20) +2(0.05+0.30) + 1 = 1.15 + 1

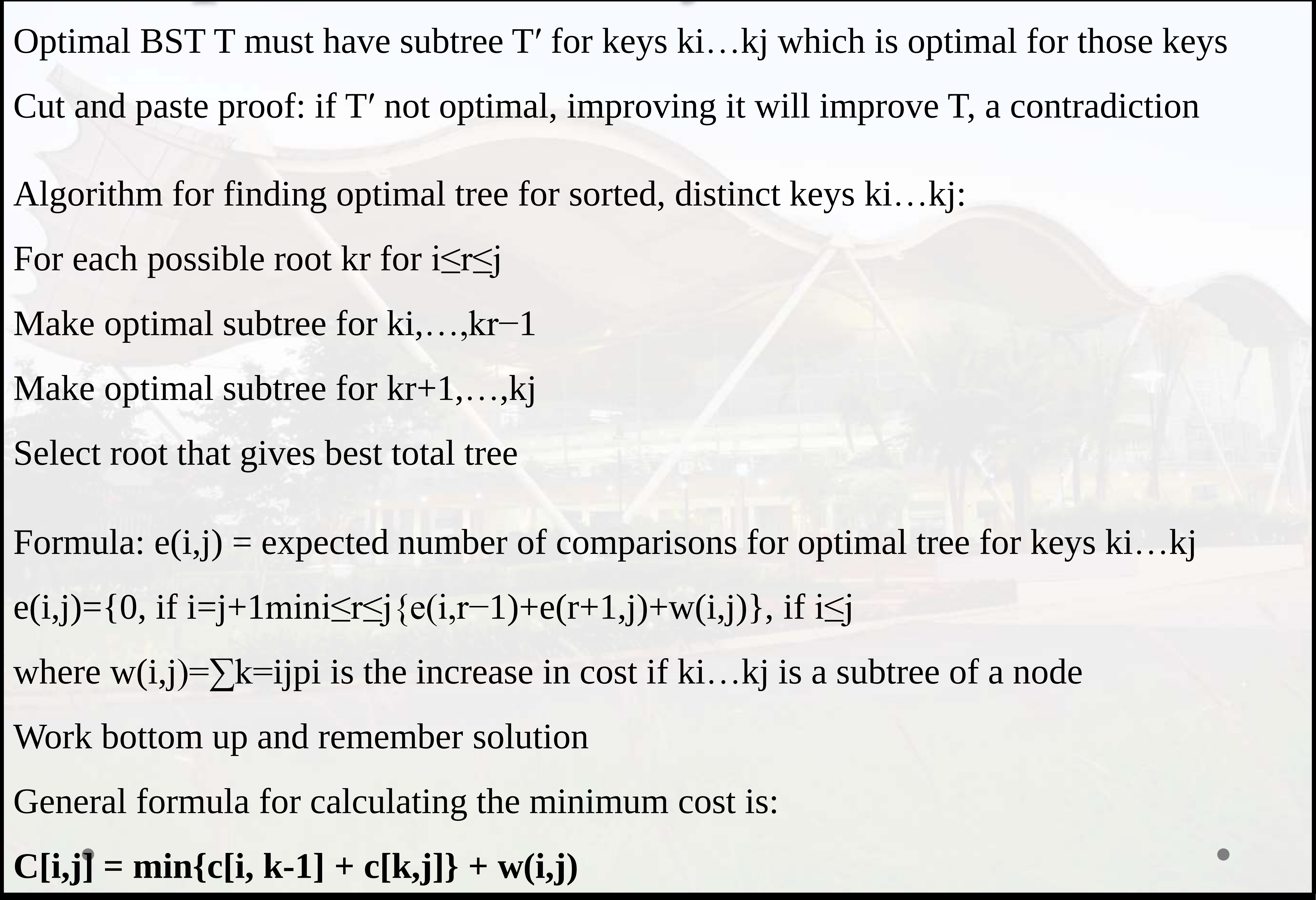
Tree 2:

k2/[k1,k5]/[nil,nil],[k4,nil]/[nil,nil],[nil,nil],[k3,nil],[nil,nil]

cost = 0(0.20) + 1(0.25+0.30) +2(0.20) + 3(0.05) + 1 = 1.10 + 1

Notice that a deeper tree has expected lower cost

Optimal Binary Search Tree



Optimal BST T must have subtree T′ for keys ki…kj which is optimal for those keys

Cut and paste proof: if T′ not optimal, improving it will improve T, a contradiction

Algorithm for finding optimal tree for sorted, distinct keys ki…kj:

For each possible root kr for i≤r≤j

Make optimal subtree for ki,…,kr−1

Make optimal subtree for kr+1,…,kj

Select root that gives best total tree

Formula: e(i,j) = expected number of comparisons for optimal tree for keys ki…kj

e(i,j)={0, if i=j+1mini≤r≤j{e(i,r−1)+e(r+1,j)+w(i,j)}, if i≤j

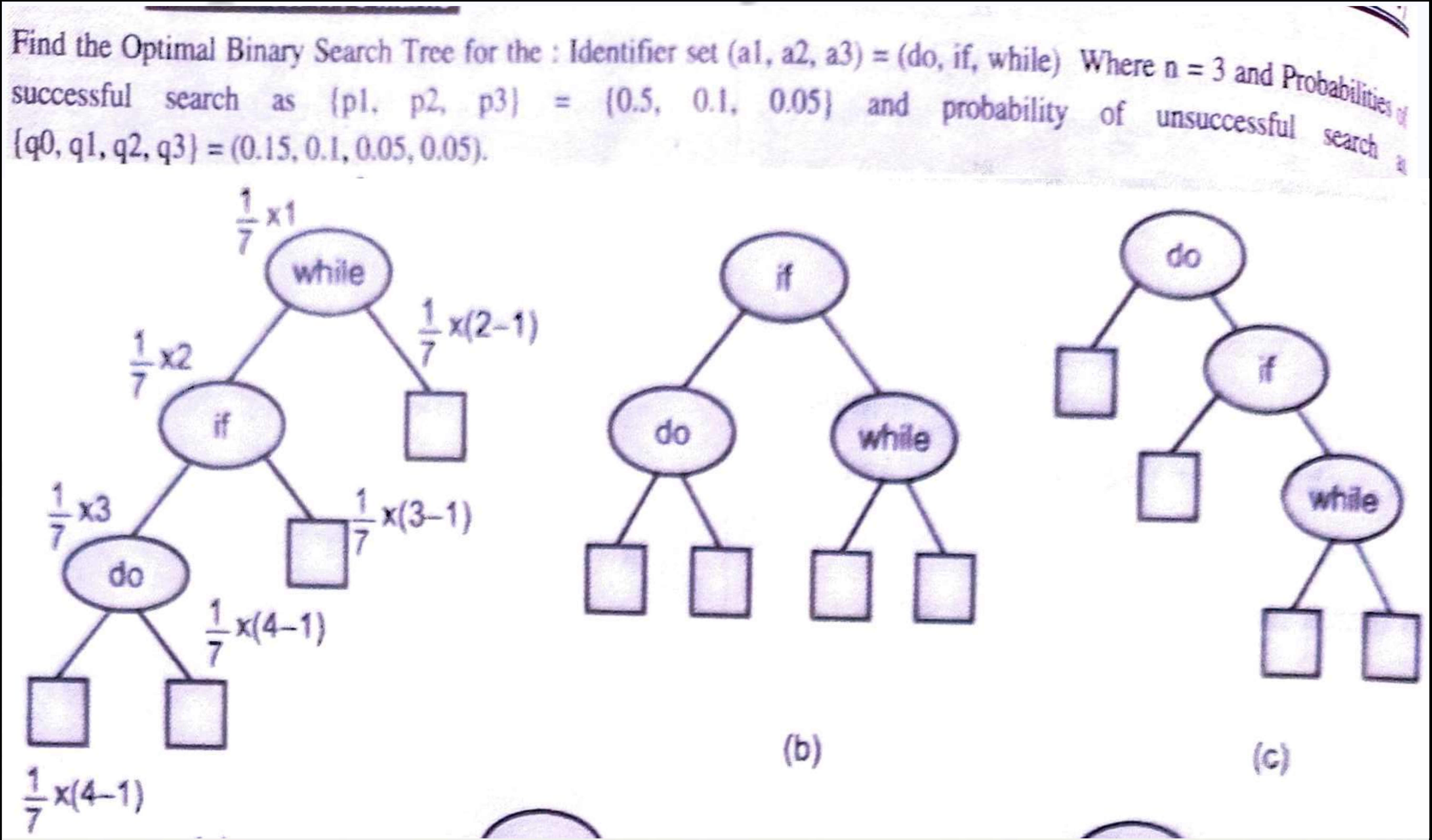
where w(i,j)=∑k=ijpi is the increase in cost if ki…kj is a subtree of a node

Work bottom up and remember solution

General formula for calculating the minimum cost is:

C[i,j] = min{c[i, k-1] + c[k,j]} + w(i,j)

Optimal Binary Search Tree

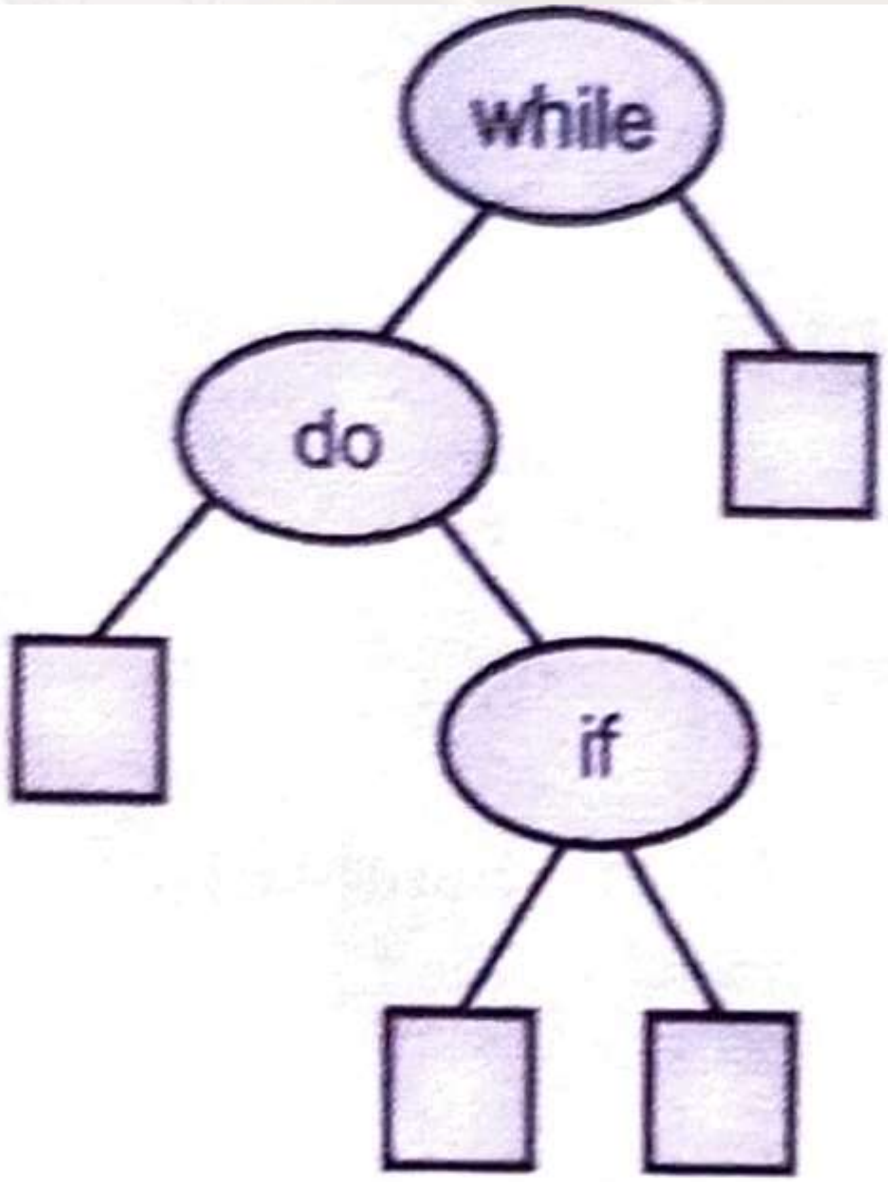


Find the Optimal Binary Search Tree for the : Identifier set (a1, a2, a3) = (do, if, while) Where n = 3 and Probabilities of successful search as {p1, p2, p3} = {0.5, 0.1, 0.05} and probability of unsuccessful search is {q0, q1, q2, q3} = (0.15, 0.1, 0.05, 0.05).

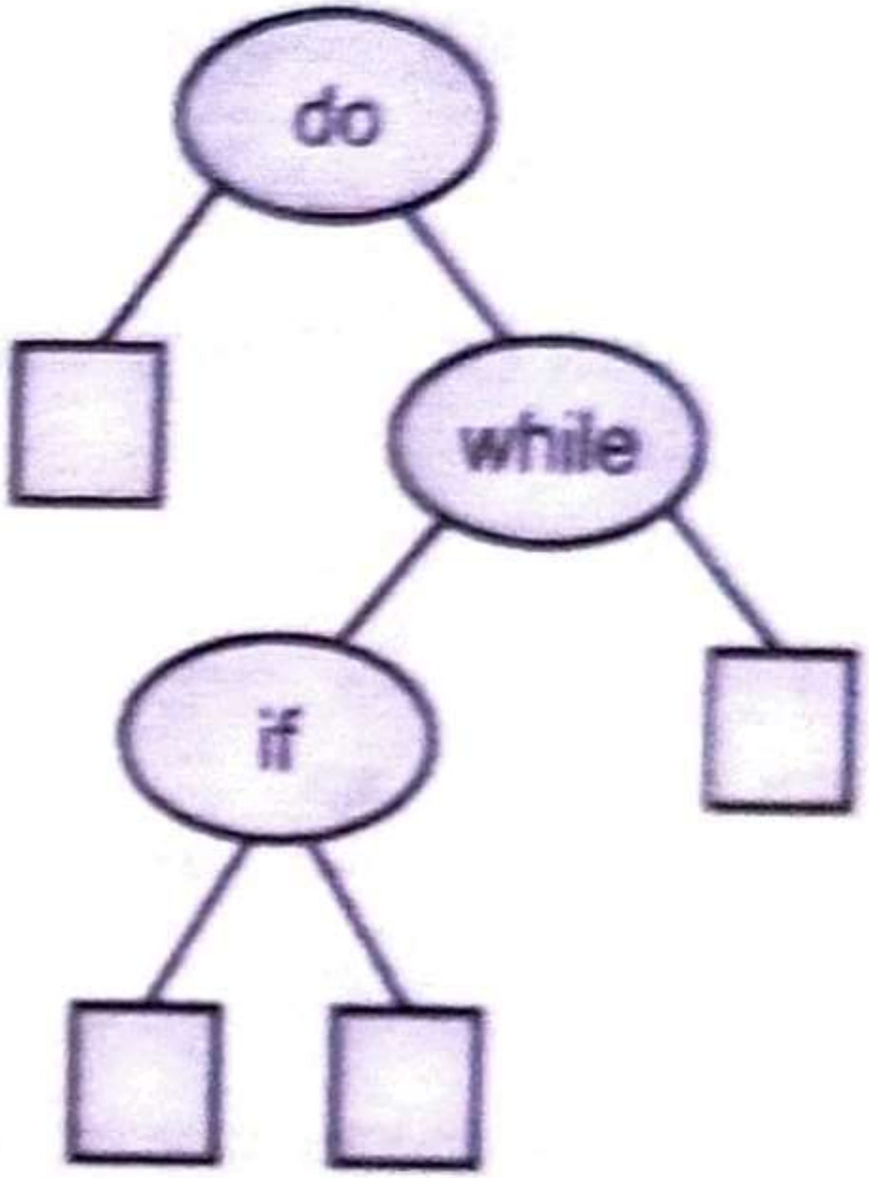
https://www.javatpoint.com/optimal-binary-search-tree

Optimal Binary Search Tree

Find the Optimal Binary Search Tree for the : Identifier set (a1, a2, a3) = (do, if, while) Where n = 3 and Probabilities of successful search as {p1, p2, p3} = {0.5, 0.1, 0.05} and probability of unsuccessful search {q0, q1, q2, q3} = (0.15, 0.1, 0.05, 0.05).

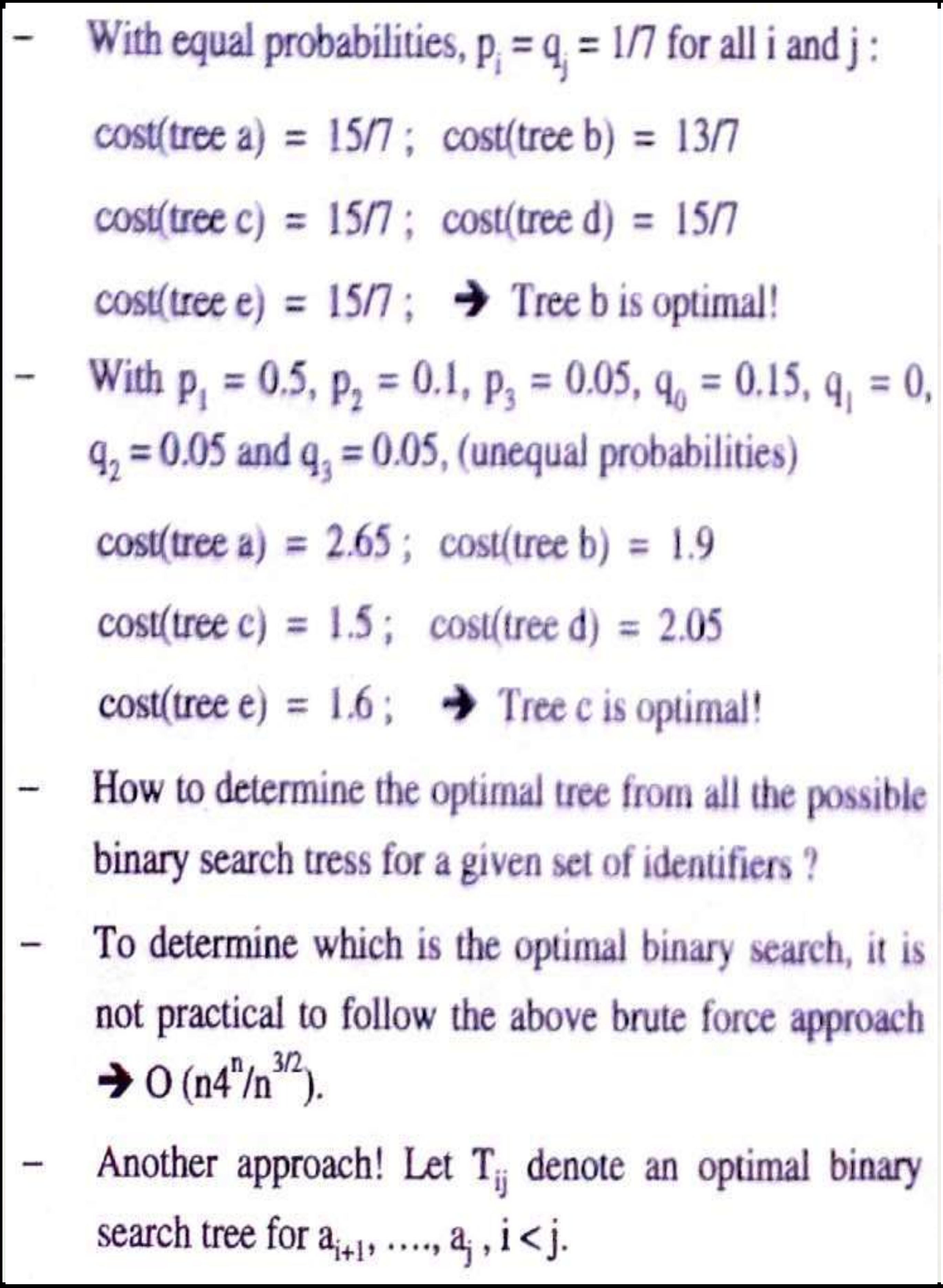


(d)



(e)

Optimal Binary Search Tree



- With equal probabilities, $p_i = q_j = 1/7$ for all i and j :

  cost(tree a) = 15/7 ; cost(tree b) = 13/7

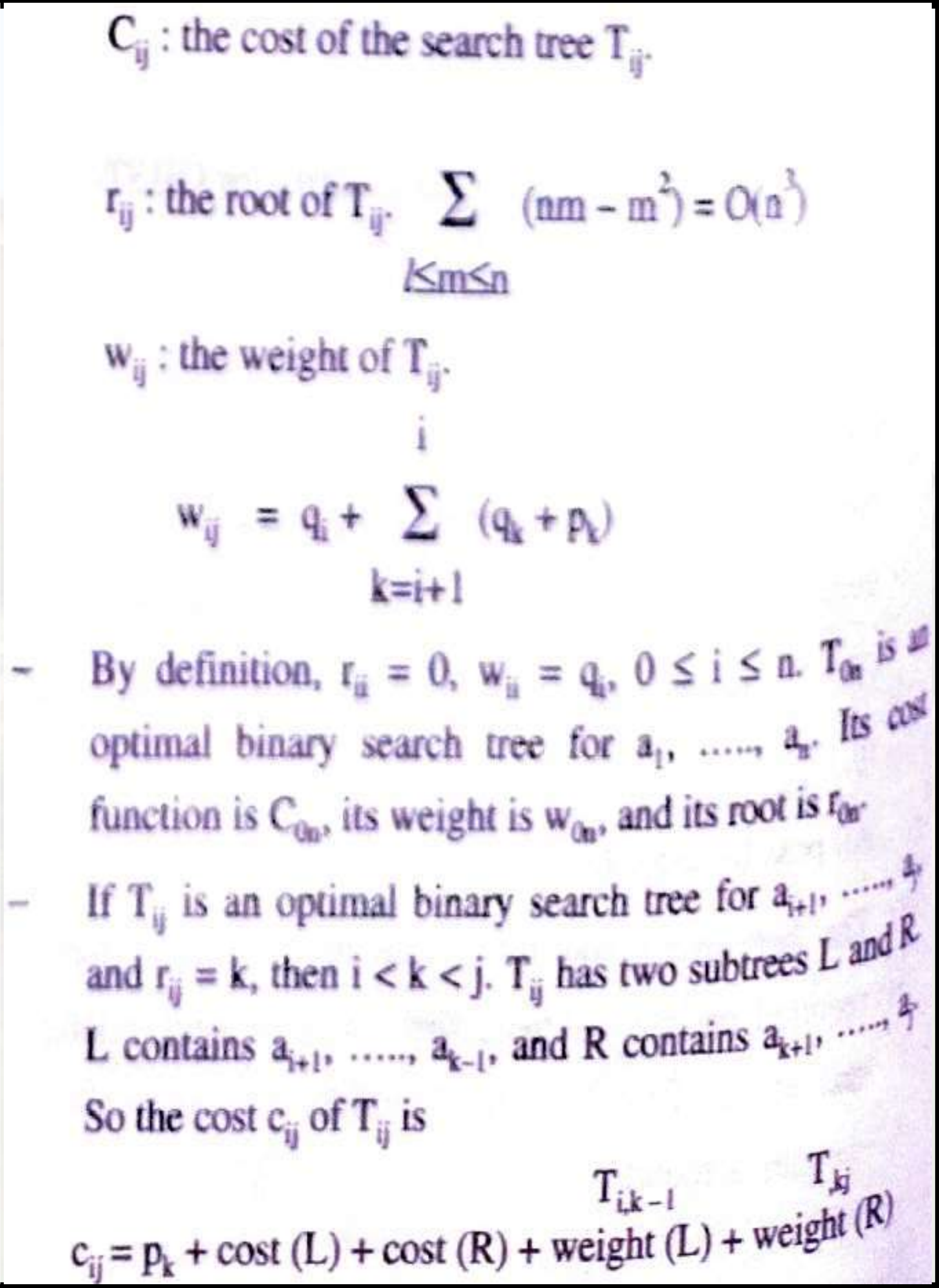
  cost(tree c) = 15/7 ; cost(tree d) = 15/7

  cost(tree e) = 15/7 ; ➔ Tree b is optimal!
- With $p_1 = 0.5$, $p_2 = 0.1$, $p_3 = 0.05$, $q_0 = 0.15$, $q_1 = 0$, $q_2 = 0.05$ and $q_3 = 0.05$, (unequal probabilities)

  cost(tree a) = 2.65 ; cost(tree b) = 1.9

  cost(tree c) = 1.5 ; cost(tree d) = 2.05

  cost(tree e) = 1.6 ; ➔ Tree c is optimal!
- How to determine the optimal tree from all the possible binary search tress for a given set of identifiers ?
- To determine which is the optimal binary search, it is not practical to follow the above brute force approach ➔ $O(n4^n/n^{3/2})$.
- Another approach! Let T_{ij} denote an optimal binary search tree for $a_{i+1}, \ldots, a_j$, $i < j$.



C_{ij} : the cost of the search tree T_{ij}.

r_{ij} : the root of T_{ij}. $\sum_{l \le m \le n} (nm - m^2) = O(n^3)$

w_{ij} : the weight of T_{ij}.

$$w_{ij} = q_i + \sum_{k=i+1}^{i} (q_k + p_k)$$

- By definition, $r_{ii} = 0$, $w_{ii} = q_i$, $0 \le i \le n$. T_{0n} is a optimal binary search tree for $a_1, \ldots, a_n$. Its cos function is C_{0n}, its weight is w_{0n}, and its root is r_{0n}.
- If T_{ij} is an optimal binary search tree for $a_{i+1}, \ldots, a_j$ and $r_{ij} = k$, then $i < k < j$. T_{ij} has two subtrees L and R. L contains $a_{i+1}, \ldots, a_{k-1}$, and R contains $a_{k+1}, \ldots, a_j$. So the cost c_{ij} of T_{ij} is

$$c_{ij} = p_k + \text{cost}(L) + \text{cost}(R) + \overset{T_{i,k-1}}{\text{weight}(L)} + \overset{T_{kj}}{\text{weight}(R)}$$

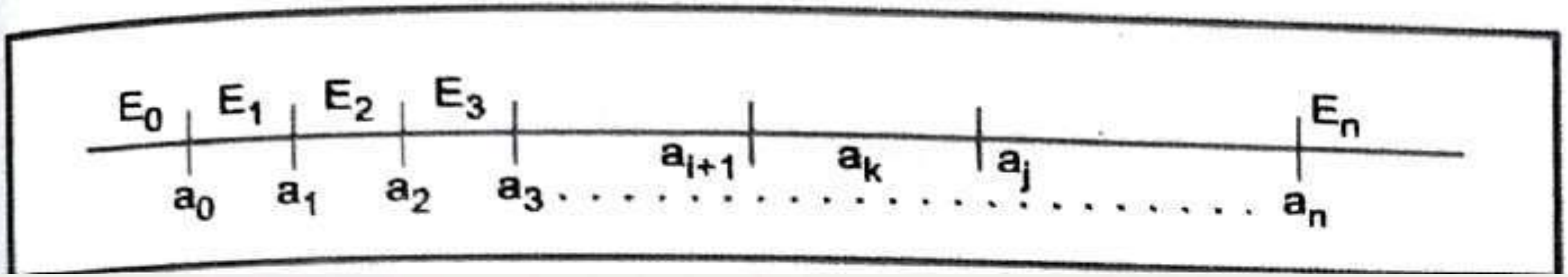
Optimal Binary Search Tree

$$c_{ij} = p_k + c_{i,\,k-1} + c_{kj} + w_{i,k-1} + w_{kj} = w_{ij} + c_{i,\,k-1} + c_{kj}$$

Since T_{ij} is optimal, we have

$$w_{ii} + c_{i\cdot k-1} + c_{ki} = \min_{i \le l \le j} \{ w_{ij} + c_{i,l-1} + c_{lj} \}$$

$$c_{i,k-1} + c_{kj} = \min_{i \le l \le j} \{ c_{i,l-1} + c_{lj} \}$$



[Example OBST : https://youtu.be/vLS-zRCHo-Y](https://youtu.be/vLS-zRCHo-Y)

Optimal Binary Search Tree

- $n = 4$, $(a_1, a_2, a_3, a_4) = (\mathbf{10, 15, 20, 25})$.
  $16\times(p_1, p_2, p_3, p_4) = (3, 3, 1, 1)$
  $16\times(q_0, q_1, q_2, q_3, q_4) = (2, 3, 1, 1, 1)$.
- Initially $w_{ii} = q_i$, $c_{ii} = 0$, and $r_{ii} = 0$, $0 \le i \le 4$

$$w_{01} = p_1 + w_{00} + w_{11} = p_1 + q_0 + q_1 = 8$$
$$c_{01} = w_{01} + \min\{c_{00} + c_{11}\} = 8, \quad r_{01} = 1$$

$$w_{12} = p_2 + w_{11} + w_{22} = p_2 + q_1 + q_2 = 7$$
$$c_{12} = w_{12} + \min\{c_{11} + c_{22}\} = 7, \quad r_{12} = 2$$

$$w_{14} = 11$$
$$c_{14} = w_{14} + \min\{c_{11} + c_{24}, c_{12} + c_{34}, c_{13} + c_{44}\}$$
$$= 11 + c_{11} + c_{24} = 19, \qquad r_{14} = 2$$

$$w_{04} = 16$$
$$c_{04} = w_{04} + \min\{c_{00} + c_{14}, c_{01} + c_{24}, c_{02} + c_{34}, c_{03} + c_{44}\}$$
$$= 16 + c_{01} + c_{24} = 32, \qquad r_{04} = 2$$

Optimal Binary Search Tree

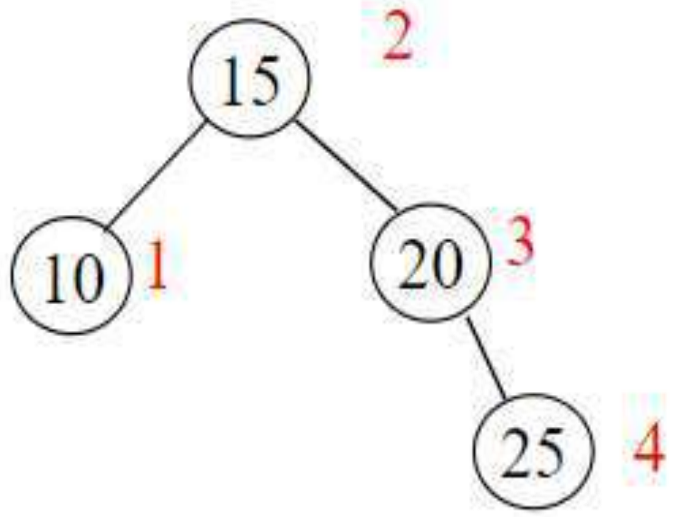
- $w_{ii} = q_i$
- $w_{ij} = p_k + w_{i,k-1} + w_{kj}$
- $c_{ij} = w_{ij} + \min\limits_{i<l\leq j}\{c_{i,l-1} + c_{lj}\}$
- $c_{ii} = 0$
- $r_{ii} = 0$
- $r_{ij} = l$

$(a_1, a_2, a_3, a_4) = (10, 15, 20, 25)$

$(p_1, p_2, p_3, p_4) = (3, 3, 1, 1)$

$(q_0, q_1, q_2, q_3, q_4) = (2, 3, 1, 1, 1)$

	0	1	2	3	4
0	$w_{00}=2$ $c_{00}=0$ $r_{00}=0$	$w_{11}=3$ $c_{11}=0$ $r_{11}=0$	$w_{22}=1$ $c_{22}=0$ $r_{22}=0$	$w_{33}=1$ $c_{33}=0$ $r_{33}=0$	$w_{44}=1$ $c_{44}=0$ $r_{44}=0$
1	$w_{01}=8$ $c_{01}=8$ $r_{01}=1$	$w_{12}=7$ $c_{12}=7$ $r_{12}=2$	$w_{23}=3$ $c_{23}=3$ $r_{23}=3$	$w_{34}=3$ $c_{34}=3$ $r_{34}=4$	
2	$w_{02}=12$ $c_{02}=19$ $r_{02}=1$	$w_{13}=9$ $c_{13}=12$ $r_{13}=2$	$w_{24}=5$ $c_{24}=8$ $r_{24}=3$		
3	$w_{03}=14$ $c_{03}=25$ $r_{03}=2$	$w_{14}=11$ $c_{14}=19$ $r_{14}=2$			
4	$w_{04}=16$ $c_{04}=32$ $r_{04}=2$				

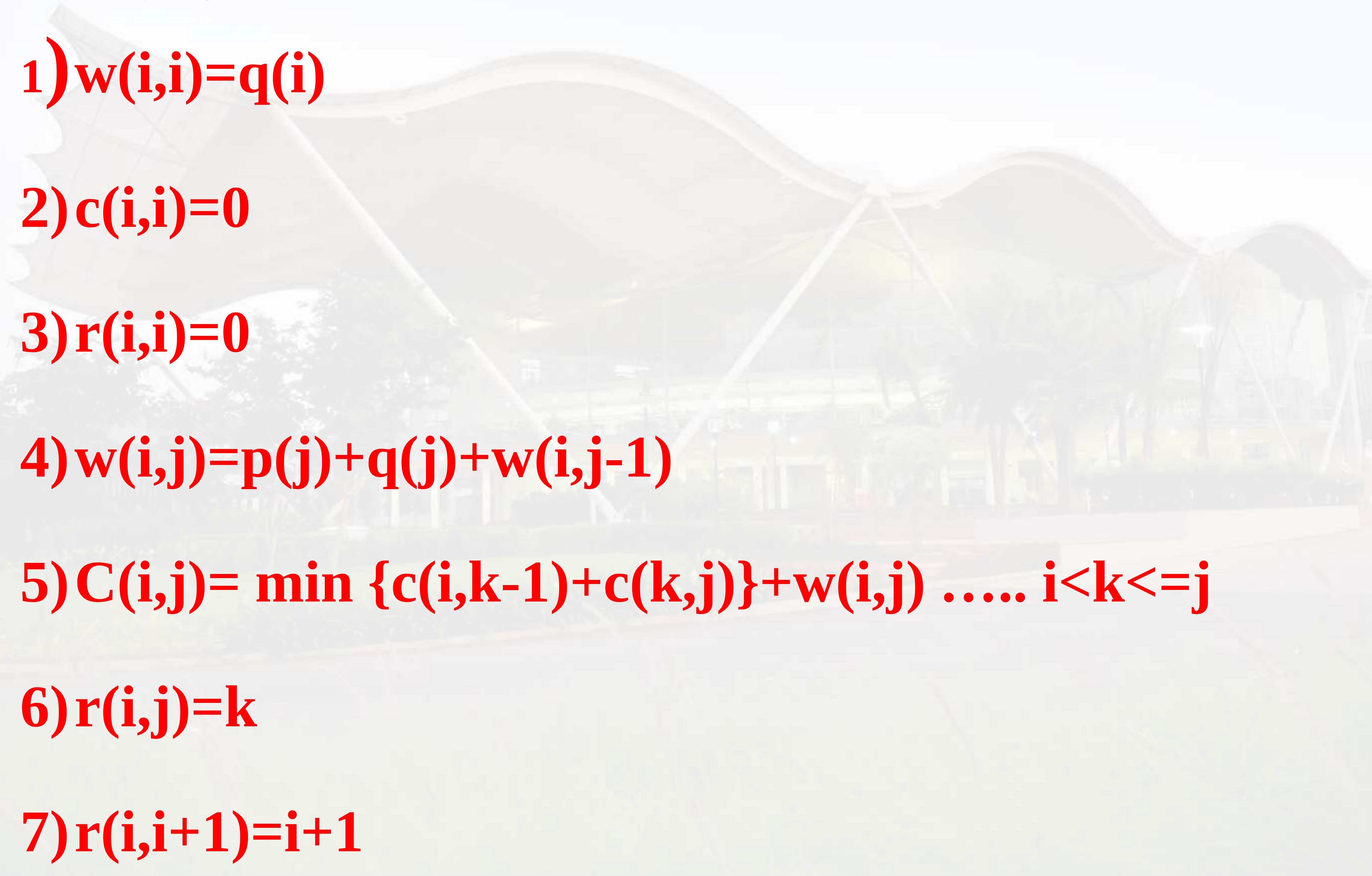


The optimal binary search tree

Computation is carried out row-wise from row 0 to row 4.

Optimal Binary Search Trees(OBST)

Formula:-



1) $w(i,i)=q(i)$

2) $c(i,i)=0$

3) $r(i,i)=0$

4) $w(i,j)=p(j)+q(j)+w(i,j-1)$

5) $C(i,j)= \min \{c(i,k-1)+c(k,j)\}+w(i,j)$ ….. $i<k<=j$

6) $r(i,j)=k$

7) $r(i,i+1)=i+1$

OBST Example

Example 1:

Let n = 4, and (a_1, a_2, a_3, a_4) = (do, if, need, while) Let P (1: 4) = (3, 3, 1, 1) and Q (0: 4) = (2, 3, 1, 1, 1)

Solution:

Table for recording W (i, j), C (i, j) and R (i, j):

Column Row	**0**	**1**	**2**	**3**	**4**
0	2, 0, 0	3, 0, 0	1, 0, 0	1, 0, 0,	1, 0, 0
1	8, 8, 1	7, 7, 2	3, 3, 3	3, 3, 4	
2	12, 19, 1	9, 12, 2	5, 8, 3		
3	14, 25, 2	11, 19, 2			
4	16, 32, 2				

OBST Example

Solution:-

This computation is carried out row-wise from row 0 to row 4. Initially, W (i, i) = Q

(i)and C (i, i) = 0 and R (i, i) = 0, 0 < i < 4.

Start with i = 0; so j = 1; as i < k ≤ j, so the possible value for k = 1

W (0, 1) = P (1) + Q (1) + W (0, 0) = 3 + 3 + 2 = 8

C (0, 1) = W (0, 1) + min {C (0, 0) + C (1, 1)} = 8

R (0, 1) = 1 (value of 'K' that is minimum in the above equation).

Next with i = 1; so j = 2; as i < k ≤ j, so the possible value for k = 2

W (1, 2) = P (2) + Q (2) + W (1, 1) = 3 + 1 + 3 = 7

C (1, 2) = W (1, 2) + min {C (1, 1) + C (2, 2)} = 7

R (1, 2) = 2

Next with i = 2; so j = 3; as i < k ≤ j, so the possible value for k = 3

W (2, 3) = P (3) + Q (3) + W (2, 2) = 1 + 1 + 1 = 3
C (2, 3) = W (2, 3) + min {C (2, 2) + C (3, 3)} = 3 + [(0 + 0)] = 3
R (2, 3) = 3

Next with i = 3; so j = 4; as i < k ≤ j, so the possible value for k = 4

W (3, 4) = P (4) + Q (4) + W (3, 3) = 1 + 1 + 1 = 3
C (3, 4) = W (3, 4) + min {[C (3, 3) + C (4, 4)]} = 3 + [(0 + 0)] = 3
R (3, 4) = 4

Start with i = 0; so j = 2; as i < k ≤ J, so the possible values for k = 1 and 2.

W (0, 2) = P (2) + Q (2) + W (0, 1) = 3 + 1 + 8 = 12
C (0, 2) = W (0, 2) + min {(C (0, 0) + C (1, 2)), (C (0, 1) + C (2, 2))}
= 12 + min {(0 + 7, 8 + 0)} = 19
R (0, 2) = 1

Next, with i = 1; so j = 3; as i < k ≤ j, so the possible value for k = 2 and 3.

W (1, 3) = P (3) + Q (3) + W (1, 2) = 1 + 1+ 7 = 9
C (1, 3) = W (1, 3) + min {[C (1, 1) + C (2, 3)], [C (1, 2) + C (3, 3)]}
= W (1, 3) + min {(0 + 3), (7 + 0)} = 9 + 3 = 12
R (1, 3) = 2

Next, with i = 2; so j = 4; as i < k ≤ j, so the possible value for k = 3 and 4.

W (2, 4) = P (4) + Q (4) + W (2, 3) = 1 + 1 + 3 = 5

C (2, 4) = W (2, 4) + min {[C (2, 2) + C (3, 4)], [C (2, 3) + C (4, 4)]

= 5 + min {(0 + 3), (3 + 0)} = 5 + 3 = 8

R (2, 4) = 3

Start with i = 0; so j = 3; as i < k ≤ j, so the possible values for k = 1, 2 and 3.

W (0, 3) = P (3) + Q (3) + W (0, 2) = 1 + 1 + 12 = 14

C (0, 3) = W (0, 3) + min {[C (0, 0) + C (1, 3)], [C (0, 1) + C (2, 3)],[C (0, 2) + C (3, 3)]}

= 14 + min {(0 + 12), (8 + 3), (19 + 0)} = 14 + 11 = 25

R (0, 3) = 2

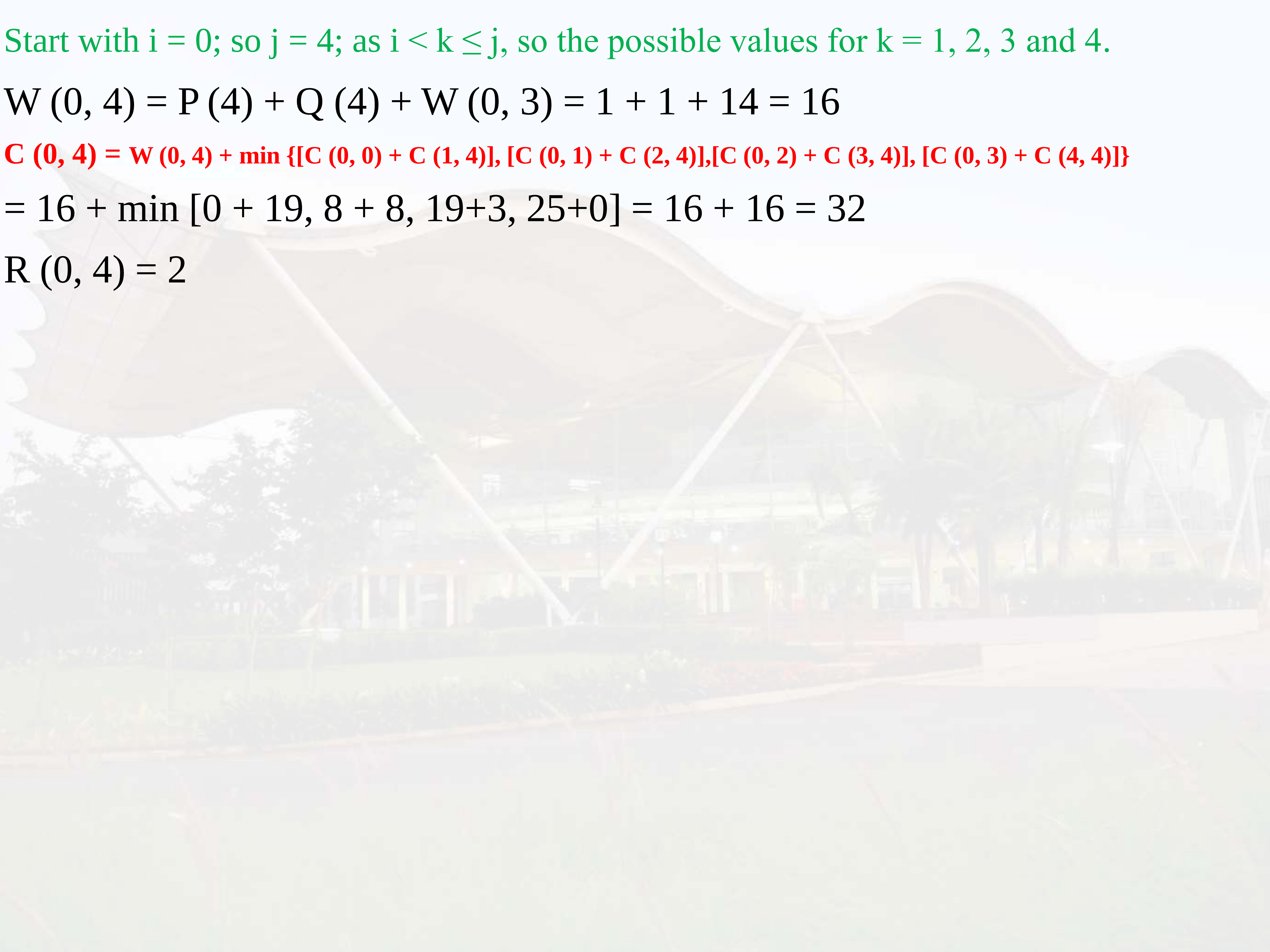
Start with i = 1; so j = 4; as i < k ≤ j, so the possible values for k = 2, 3 and 4.

W (1, 4) = P (4) + Q (4) + W (1, 3) = 1 + 1 + 9 = 11

C (1, 4) = W (1, 4) + min {[C (1, 1) + C (2, 4)], [C (1, 2) + C (3, 4)],[C (1, 3) + C (4, 4)]}

= 11 + min {(0 + 8), (7 + 3), (12 + 0)} = 11 + 8 = 19

R (1, 4) = 2



Start with $i = 0$; so $j = 4$; as $i < k \le j$, so the possible values for $k = 1, 2, 3$ and 4.

$W(0, 4) = P(4) + Q(4) + W(0, 3) = 1 + 1 + 14 = 16$

$C(0, 4) = W(0, 4) + \min\{[C(0, 0) + C(1, 4)], [C(0, 1) + C(2, 4)], [C(0, 2) + C(3, 4)], [C(0, 3) + C(4, 4)]\}$

$= 16 + \min[0 + 19, 8 + 8, 19+3, 25+0] = 16 + 16 = 32$

$R(0, 4) = 2$

☆ How Draw OBST Tree :—

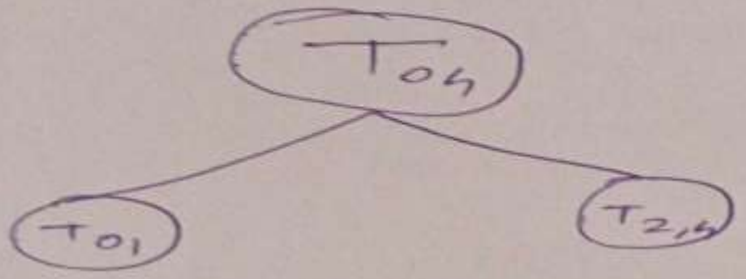
1> $\boxed{T_{04}}$

4th row & 0th column
$r_{04} = 2$ where $k = 2$

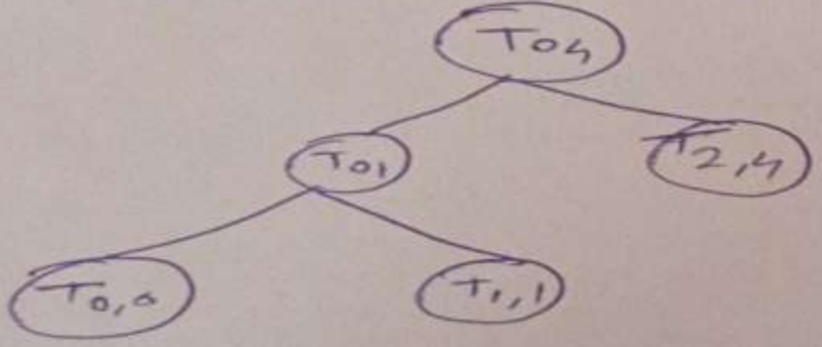
2> $r_{\underset{i}{0}\underset{j}{4}} = 2$ & $k = 2$

$T_{i,j} = T_{i,k-1}, T_{k,j}$ to generate OBST.

$T_{04} = T_{0,2-1}, T_{2,4}$
$= T_{0,1}, T_{2,4}$



3> $T_{0,1}$ simple check $r_{0,1} = 1$
hence $k = 1$
$T_{i,j} = T_{i,k-1}, T_{k,j}$
$= T_{0,0}, T_{1,1}$

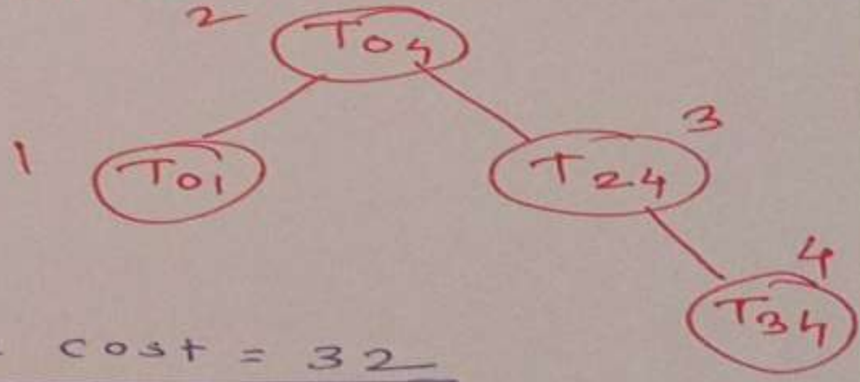


4> $T_{2,4} \longrightarrow r_{2,4} = 3 \longrightarrow k = 3.$
$T_{24} = T_{22}, T_{34}$

5> $T_{3,4} \longrightarrow r_{3,4} \longrightarrow 4 \longrightarrow k = 4$
$T_{3,4} = T_{3,3}, T_{4,4}$

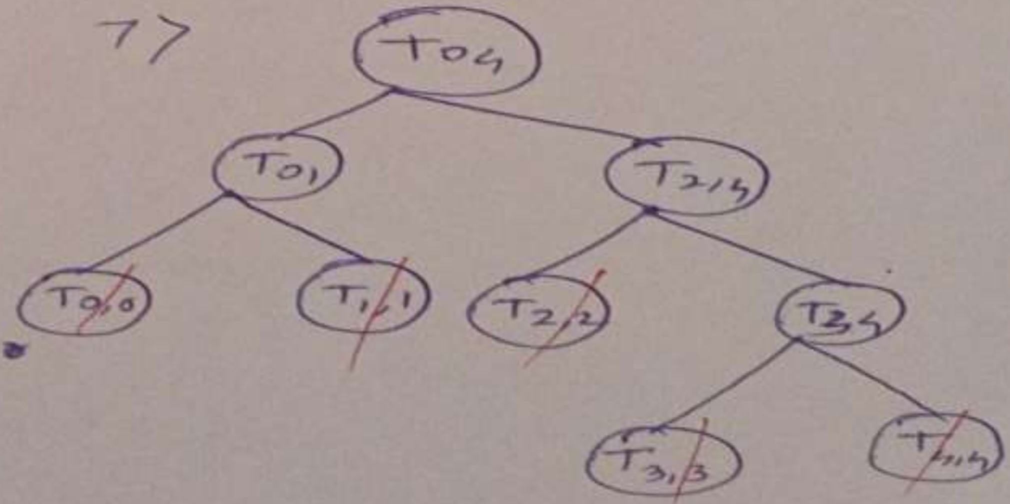
6> $r_{0,0} = r_{1,1} = r_{2,2} = r_{3,3} = r_{4,4} = 0$
Hence Discard all Nodes.

8> Final Answer —



Final cost = 32

7>



	0	1	2	3	4
0	ω(0,0)=1 c(0,0)=0 ε(0,0)=0	ω(1,1)=2 c(1,1)=0 ε(1,1)=0	ω(2,2)=1 c(2,2)=0 ε(2,2)=0	ω(3,3)=1 c(3,3)=0 ε(3,3)=0	ω(4,4)=3 c(4,4)=0 ε(4,4)=0
1	ω(0,1)=4 c(0,1)=4 ε(0,1)=1	ω(1,2)=6 c(1,2)=6 ε(1,2)=2	ω(2,3)=3 c(2,3)=3 ε(2,3)=3	ω(3,4)=7 c(3,4)=7 ε(3,4)=4	
2	ω(0,2)=8 c(0,2)=12 ε(0,2)=2	ω(1,3)=8 c(1,3)=11 ε(1,3)=2	ω(2,4)=9 c(2,4)=12 ε(2,4)=4		
3	ω(0,3)=10 c(0,3)=17 ε(0,3)=2	ω(1,4)=14 c(1,4)=25 ε(1,4)=4			
4	ω(0,4)=16 c(0,4)=32 ε(0,4)=2				

$$P(1:4) = (\underset{p_1}{1}, \underset{p_2}{3}, \underset{p_3}{1}, \underset{p_4}{3})$$

$$q(0:4) = (\underset{q_0}{1}, \underset{q_1}{2}, \underset{q_2}{1}, \underset{q_3}{1}, \underset{q_4}{3})$$

1) $\omega(0,1) = p(1) + q(1) + \omega(0,0)$
$= 1 + 2 + 1$
$= 4$

$c(0,1) = \min_{0<1\le 1} (c(0,0) + c(1,1) + w(0,1))$
$k=1$
$= 0 + 0 + 4$
$= 4$
$r(0,1) = 1$

2) $w(1,2) = p(2) + q(2) + w(1,1)$
$= 3 + 1 + 2$
$= 6$
$c(1,2) = \min_{0<2\le 2} \{ c(1,1) + c(2,2) + w(1,2)$
$k=2$
$= 0 + 0 + 6$
$= 6$
$r(1,2) = 2$

3) $w(2,3) = p(3) + q(3) + w(2,2)$
$= 1 + 1 + 1$
$= 3$
$c(2,3) = \min_{2<3\le 3} \{ c(2,2) + c(3,3) + w(2,3) \}$
$k=3$
$= 0 + 0 + 3$
$= 3$
$r(2,3) = 3$

4) $w(3,4) = p(4) + q(4) + w(3,3)$
$= 3 + 3 + 1$
$= 7$
$c(3,4) = \min_{3<4\le 4} \{ c(3,3) + c(4,4) + w(3,4) \}$
$k=4$
$= 0 + 0 + 7$
$= 7$
$r(3,4) = 4$

5) $\omega(0,2) = P(2) + q(2) + \omega(0,1)$

$= 3 + 1 + 4$

$= 8$

$c(0,2) = \min\{c(0,0) + c(1,2) + \omega(0,2)\}$

$k=1 \quad 0<1\leq 2$

$= 0 + 6 + 8$

$= 14$

$c(0,2) = \min\{c(0,1) + c(2,2) + \omega(0,2)\}$

$k=2 \quad 0<2\leq 2$

$= 4 + 0 + 8$

$= 12$

$r(0,2) = 2$

6) $\omega(1,3) = P(3) + q(3) + \omega(2,2)$

$= 1 + 1 + 6$

$= 8$

$c(1,3) = \min\{c(1,1) + c(2,3) + \omega(1,3)\}$

$k=2 \quad 1<2\leq 3$

$= 0 + 3 + 8$

$= 11$

$c(1,3) = \min\{c(1,2) + c(3,3) + \omega(1,3)\}$

$k=3 \quad 1<3\leq 3$

$= 6 + 0 + 8$

$= 14$

$r(1,3) = 2$

7) $\omega(2,4) = P(4) + q(4) + \omega(2,3)$

$= 3 + 3 + 3$

$= 9$

$c(2,4) = \min\{c(2,2) + c(3,4) + \omega(2,4)\}$

$k=3 \quad 2<3\leq 4$

$= 0 + 7 + 9$

$= 16$

$c(2,4) = \min \{ c(2,3) + c(4,4) + w(2,4) \}$

$K = 4 \quad 2 < 4 \leq 4$

$= 3 + 0 + 9$

$= 12$

$K(2,4) = 4$

8) $w(0,3) = p(3) + q(3) + w(0,2)$

$= 1 + 1 + 8$

$= 10$

$c(0,3) = \min \{ c(0,0) + c(1,3) + w(0,3) \}$

$K = 1 \quad 0 < 1 \leq 3$

$= 0 + 11 + 10 = 21$

$c(0,3) = \min \{ c(0,1) + c(2,3) + w(0,3) \}$

$K = 2 \quad 0 < 2 \leq 3$

$= 4 + 3 + 10 = 17$

$c(0,3) = \min \{ c(0,2) + c(3,3) + w(0,3) \}$

$K = 3 \quad 0 < 3 \leq 3$

$= 12 + 0 + 10 = 22$

$r(0,3) = 2$

9) $w(1,4) = p(4) + q(4) + w(1,3)$

$= 3 + 3 + 8$

$= 14$

$c(1,4) = \min \{ c(1,1) + c(2,4) + w(1,4) \}$

$K = 2 \quad 1 < 2 \leq 4$

$= 0 + 12 + 14 = 26$

$c(1,4) = \min \{ c(1,2) + c(3,4) + w(1,4) \}$

$K = 3 \quad 1 < 3 \leq 4$

$= 6 + 7 + 14 = 27$

$c(1,4) = \min \{ c(1,3) + c(4,4) + w(1,4) \}$

$K = 4 \quad 1 < 4 \leq 4$

$= 11 + 0 + 14 = 25$

$r(1,4) = 4$

10) $w(0,4) = p(4) + q(4) + w(0,3)$

$= 3 + 3 + 10$

$= 16$

$c(0,4) = \min\limits_{0<1\leq 4} \{ c(0,0) + c(1,4) + w(0,4) \}$

$k=1$

$= 0 + 25 + 16 = 41$

$c(0,4) = \min\limits_{0<2\leq 4} \{ c(0,1) + c(2,4) + w(0,4) \}$

$k=2$

$= 4 + 12 + 16 = 32$

$c(0,4) = \min\limits_{0<3\leq 4} \{ c(0,2) + c(3,4) + w(0,4) \}$

$k=3$

$= 12 + 7 + 16 = 35$

$c(0,4) = \min\limits_{0<4\leq 4} \{ c(0,3) + c(4,4) + w(0,4) \}$

$k=4$

$= 17 + 0 + 16 = 33$

$r(0,4) = 2$

AVL Tree

- **An AVL tree is a type of tree that is a self-balancing binary search tree.**
- **Properties**
- Follows all properties of the tree data structure.
- Self-balancing.
- Each node stores a value called a balanced factor, which is the difference in the height of the left sub-tree and right sub-tree.
- All the nodes in the AVL tree must have a balance factor of -1, 0, and

AVL Tree

AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honour of its inventors.

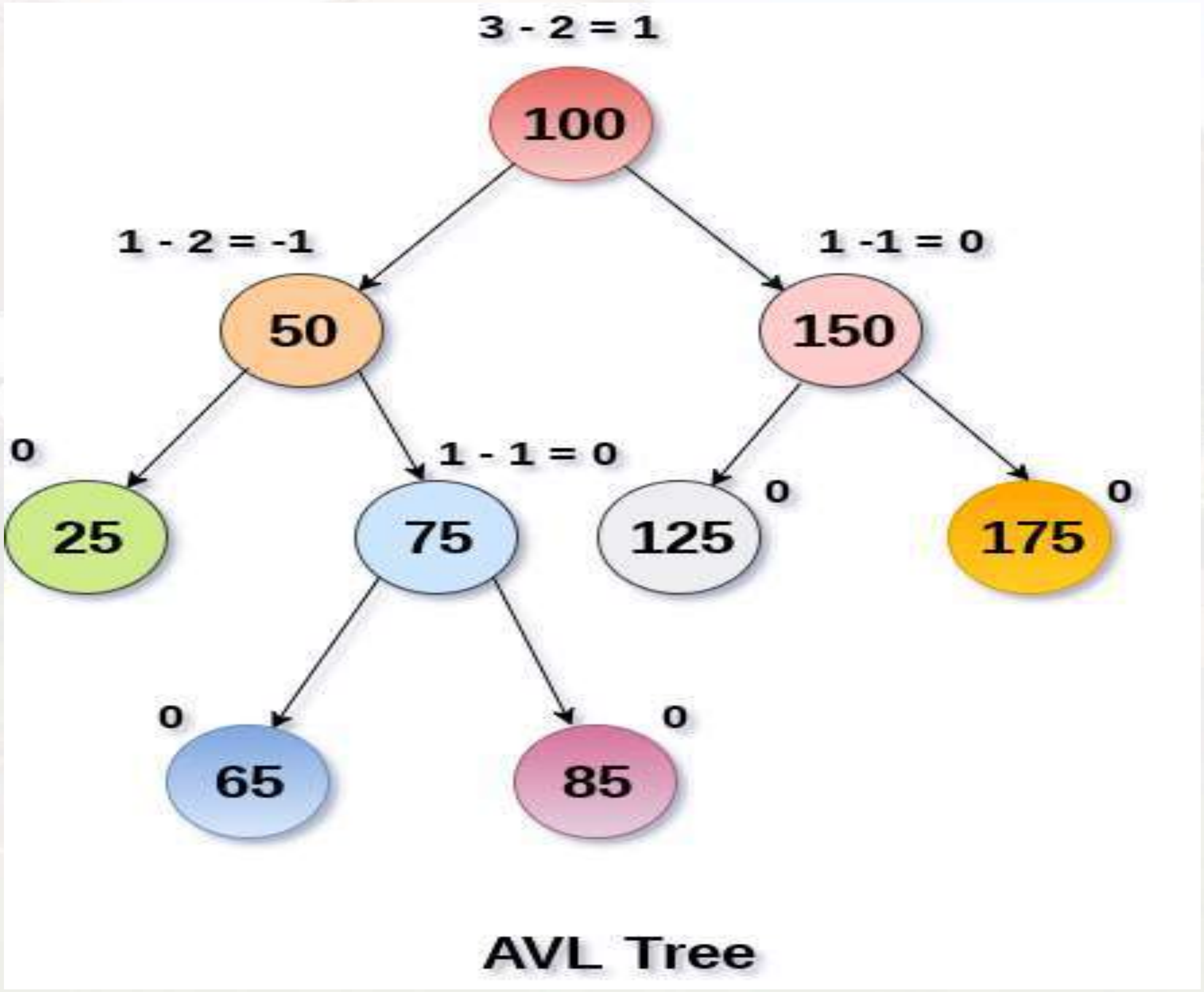
AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.

Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

Balance Factor (k) = height (left(k)) - height (right(k))

AVL Tree

Balance Factor (k) = height (left(k)) - height (right(k))



Operations of AVL Tree

SN	Operation	Description
1	Insertion	Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing. The tree can be balanced by applying rotations.
2	Deletion	Deletion can also be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree therefore, various types of rotations are used to rebalance the tree.

AVL Rotation

We perform rotation in AVL tree only in case if Balance Factor is other than **-1, 0, and 1**. There are basically four types of rotations which are as follows:

1. **LL rotation:** Inserted node is in the left subtree of left subtree of A
2. **RR rotation :** Inserted node is in the right subtree of right subtree of A
3. **LR rotation :** Inserted node is in the right subtree of left subtree of A
4. **RL rotation :** Inserted node is in the left subtree of right subtree of A

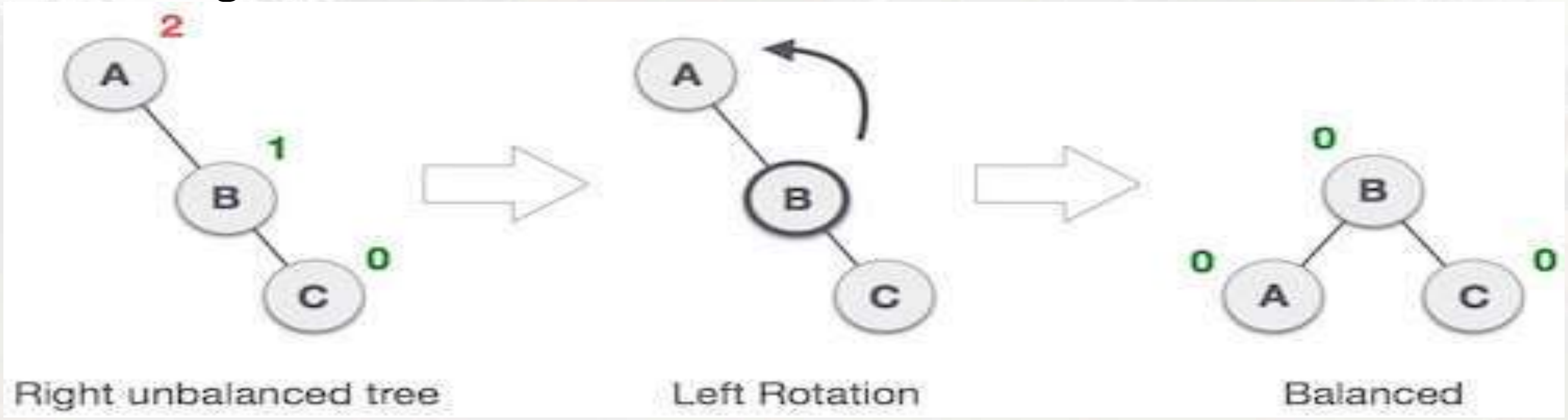
Where node A is the node whose balance Factor is other than -1, 0, 1.

The first two rotations LL and RR are single rotations and the next two rotations LR and RL are double rotations. For a tree to be unbalanced, minimum height must be at least 2. Let us understand each rotation

AVL Rotation

1. RR Rotation :

When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2

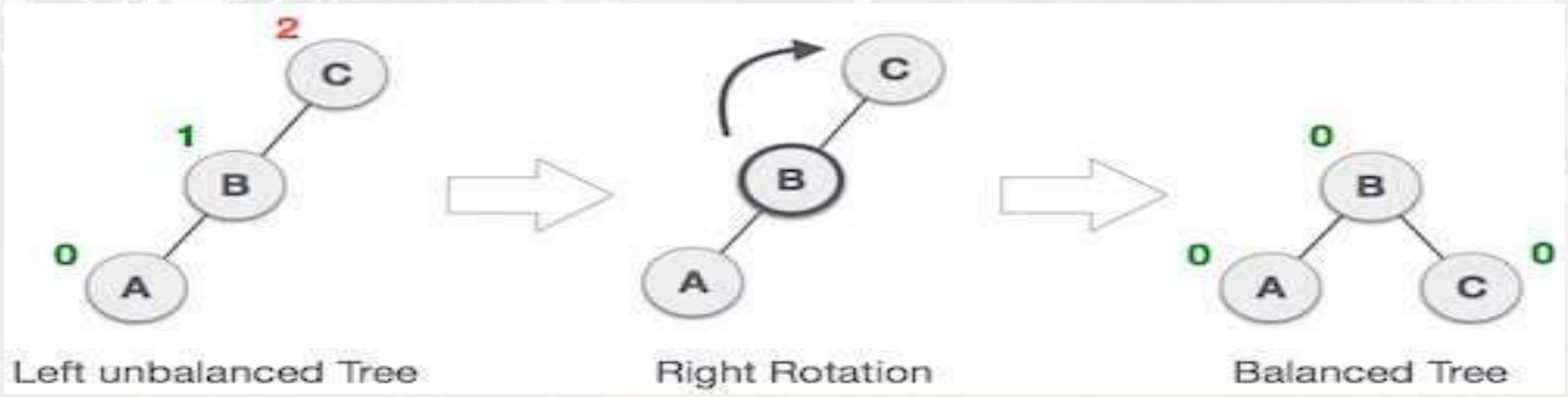


In above example, node A has balance factor -2 because a node C is inserted in the right subtree of A right subtree. We perform the RR rotation on the edge below A.

AVL Rotation

2. LL Rotation :

When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation, LL rotation is clockwise rotation, which is applied on the edge below a node having balance factor 2



In above example, node C has balance factor 2 because a node A is inserted in the left subtree of C left subtree. We perform the LL rotation on the edge below A.

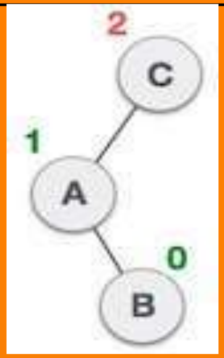
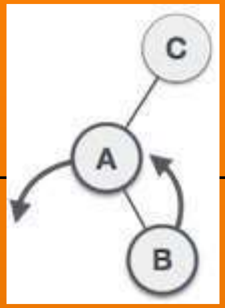
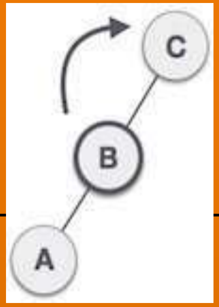
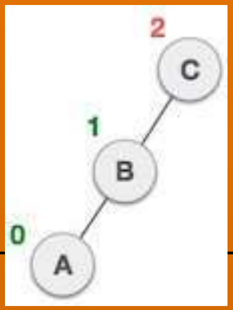
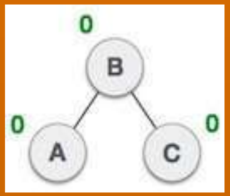
AVL Rotation

3. LR Rotation :

Double rotations are bit tougher than single rotation which has already explained above. LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

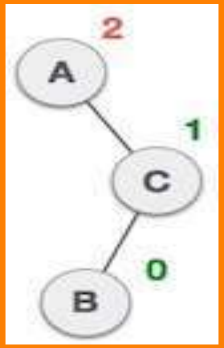
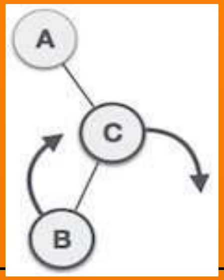
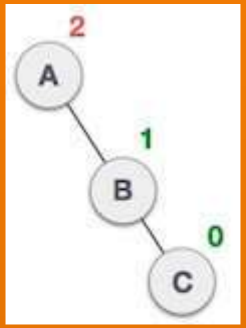
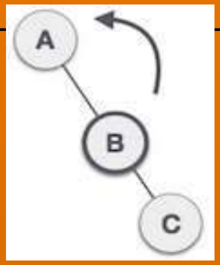
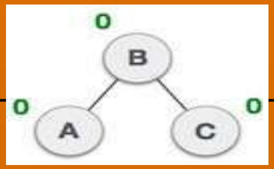
AVL Rotation

3. LR Rotation :

State	Action
 	A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of C
	As LR rotation = RR + LL rotation, hence RR (anticlockwise) on subtree rooted at A is performed first. By doing RR rotation, node A, has become the left subtree of B.
	After performing RR rotation, node C is still unbalanced, i.e., having balance factor 2, as inserted node A is in the left of left of C
	Now we perform LL clockwise rotation on full tree, i.e. on node C. node C has now become the right subtree of node B, A is left subtree of B

AVL Rotation

4. RL Rotation :

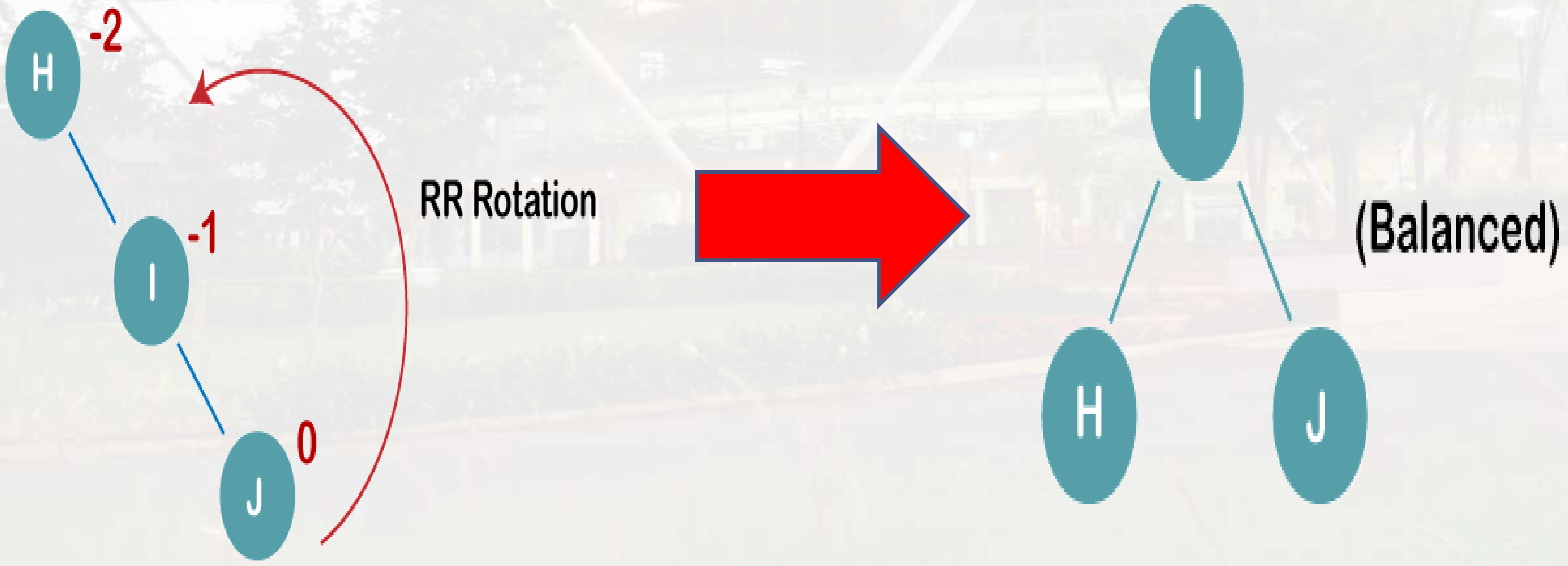
State	Action
 	A node B has been inserted into the left subtree of C the right subtree of A, because of which A has become an unbalanced node having balance factor - 2. This case is RL rotation where: Inserted node is in the left subtree of right subtree of A
	As RL rotation = LL rotation + RR rotation, hence, LL (clockwise) on subtree rooted at C is performed first. By doing RR rotation, node C has become the right subtree of B.
	After performing LL rotation, node A is still unbalanced, i.e. having balance factor -2, which is because of the right-subtree of the right-subtree node A.
	Now we perform RR rotation (anticlockwise rotation) on full tree,

AVL Tree Example

Q: Construct an AVL tree having the following elements

H, I, J, B, A, E, C, F, D, G, K, L

1. Insert H, I, J

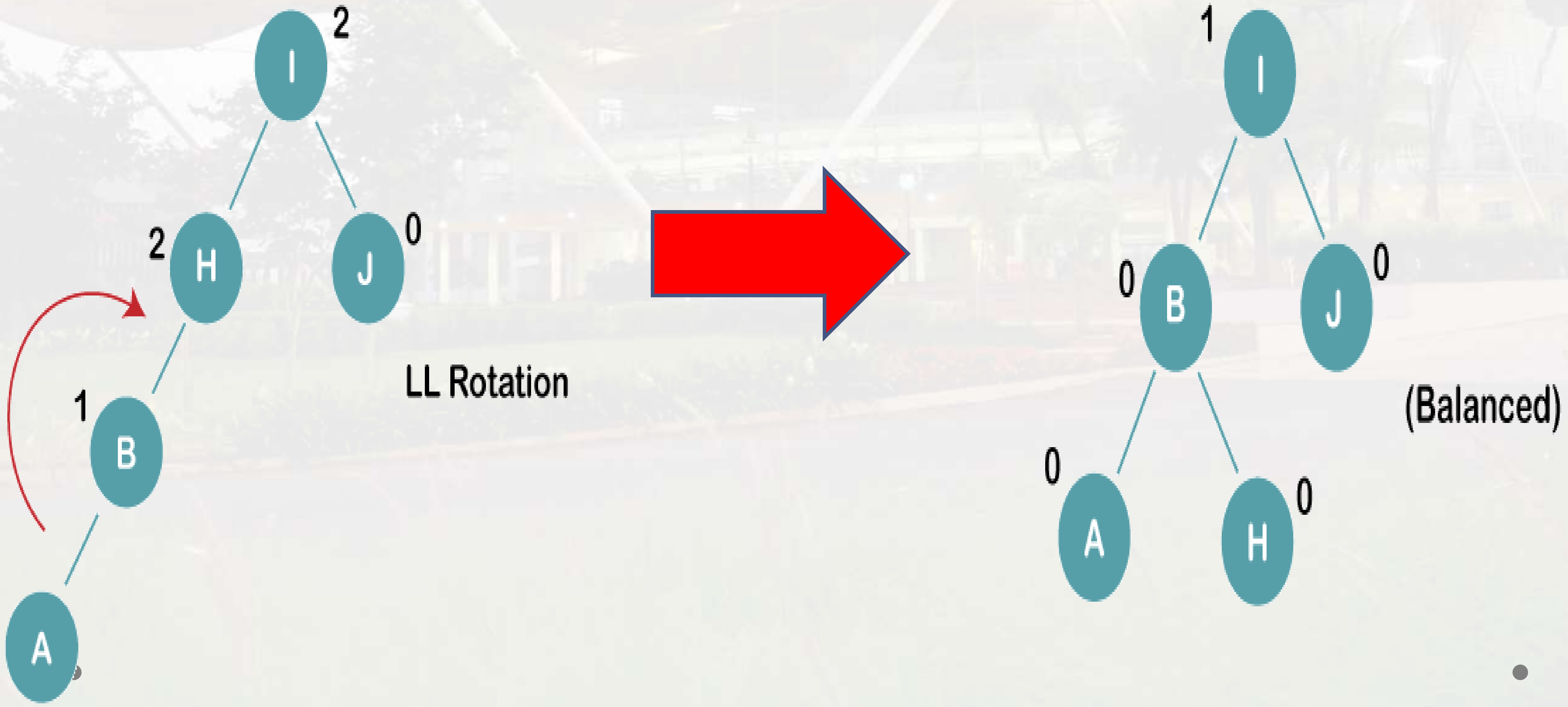


AVL Tree Example

Q: Construct an AVL tree having the following elements

H, I, J, B, A, E, C, F, D, G, K, L

2. Insert B, A



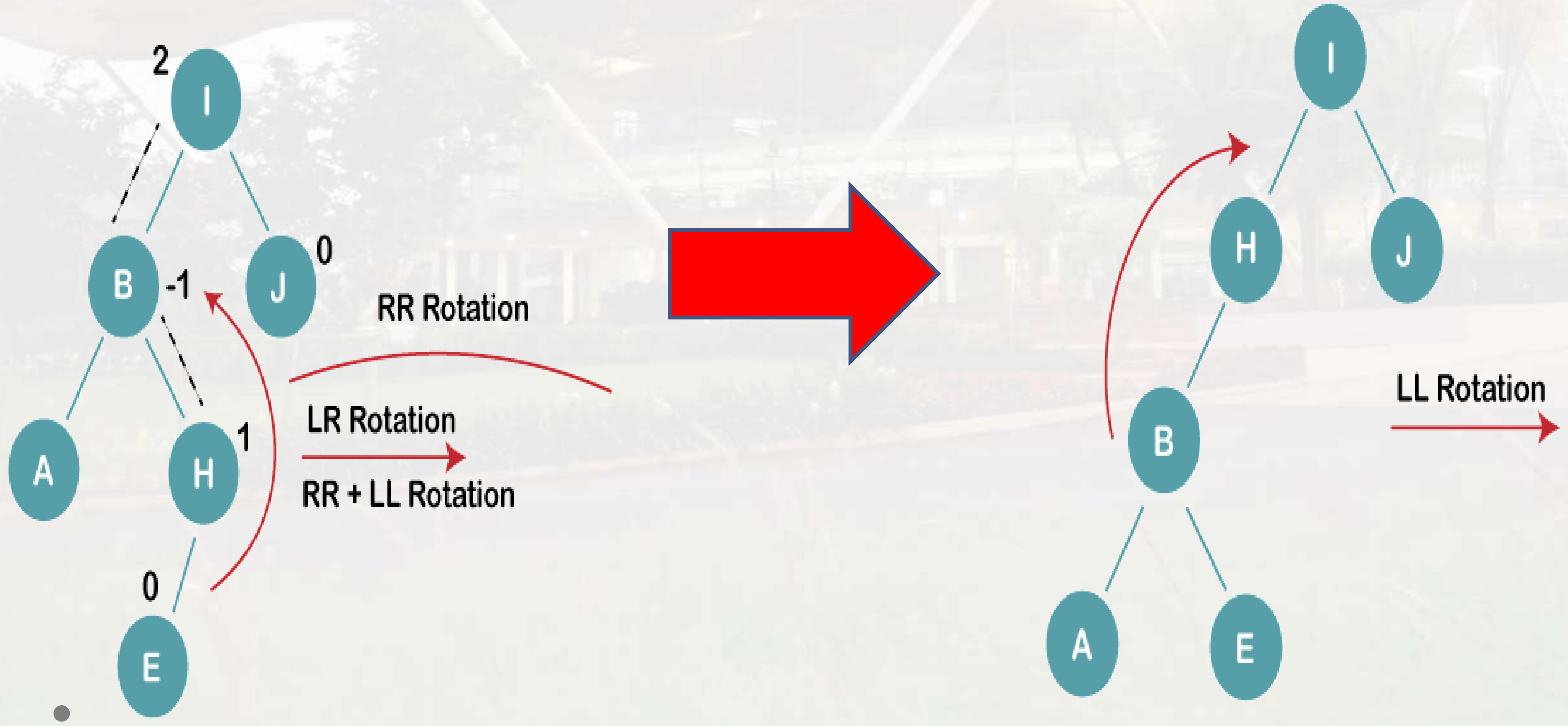
AVL Tree Example

Q: Construct an AVL tree having the following elements

H, I, J, B, A, E, C, F, D, G, K, L

3. Insert E

3 a) We first perform RR rotation on node B



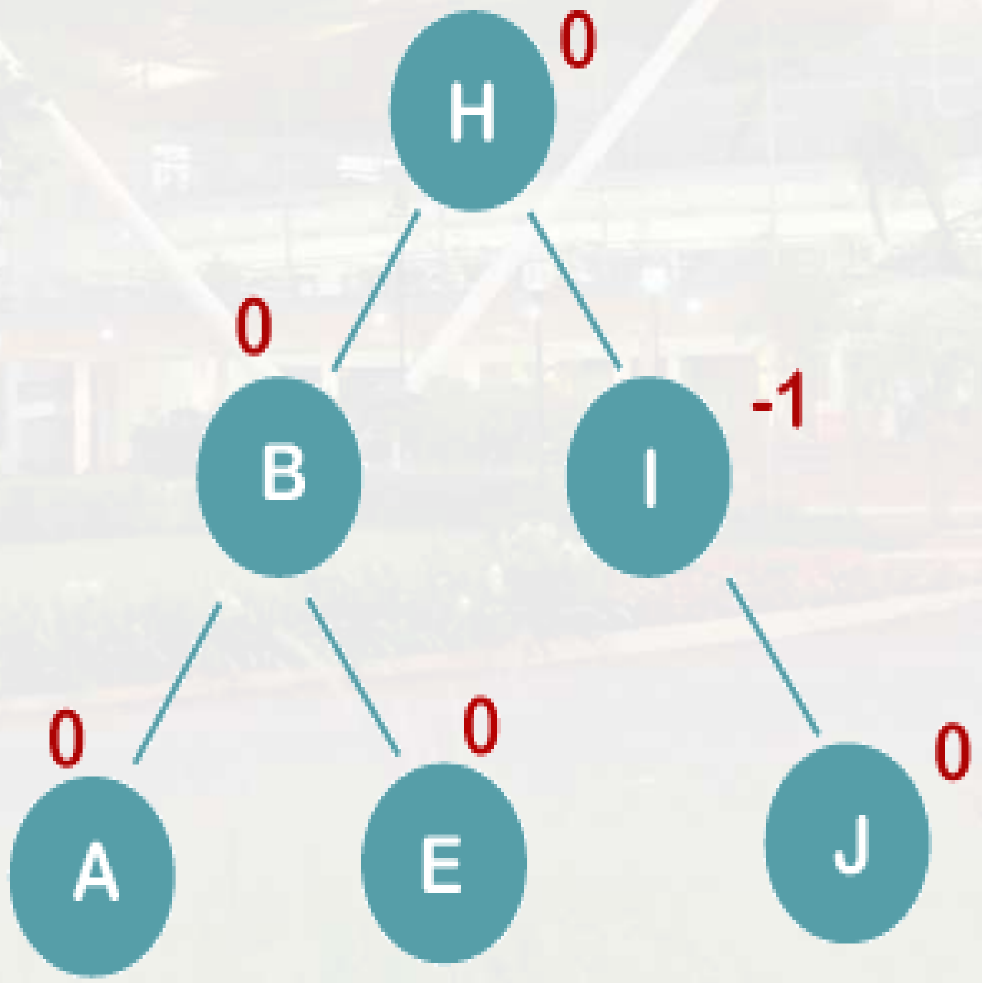
AVL Tree Example

Q: Construct an AVL tree having the following elements

H, I, J, B, A, E, C, F, D, G, K, L

3b) We first perform LL rotation on the node I

The resultant balanced tree after LL rotation is:



(Balanced)

AVL Tree Example

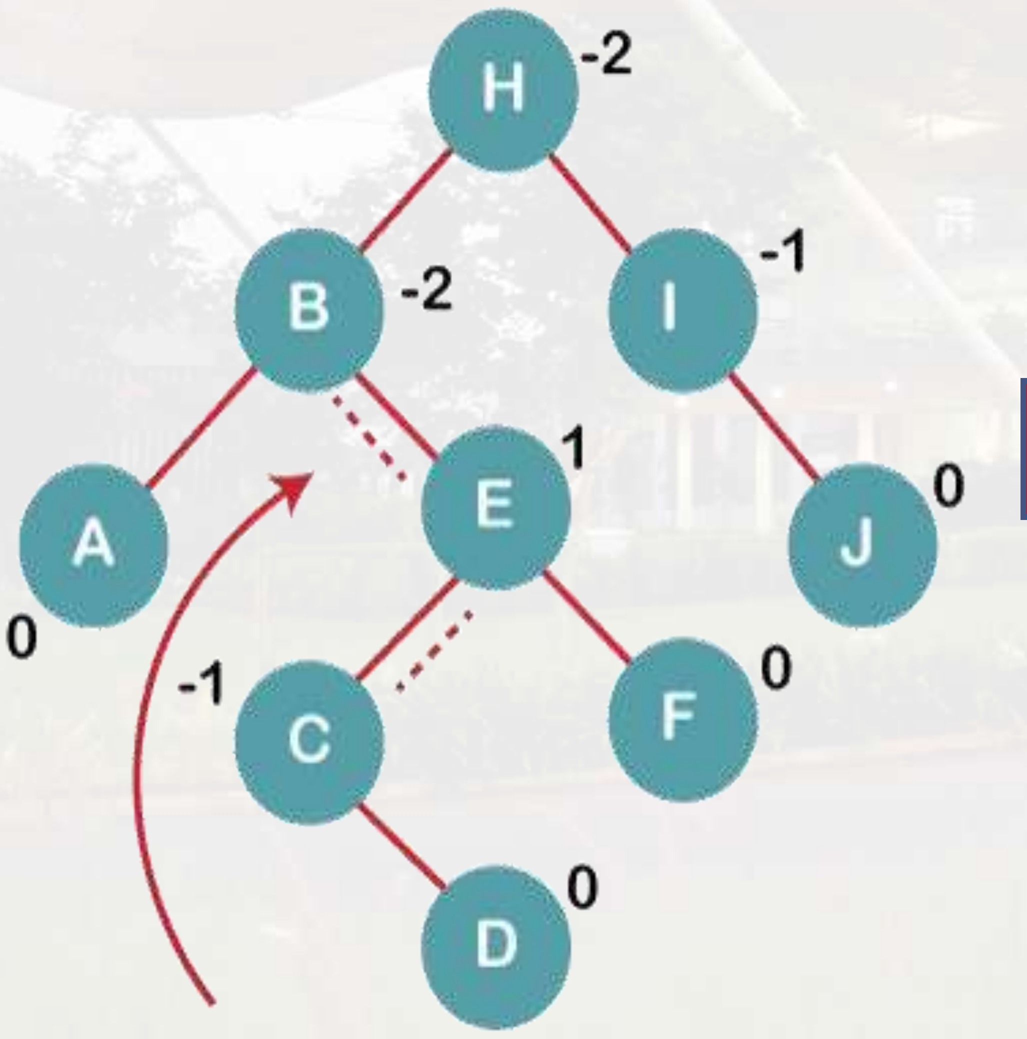
Q: Construct an AVL tree having the following elements

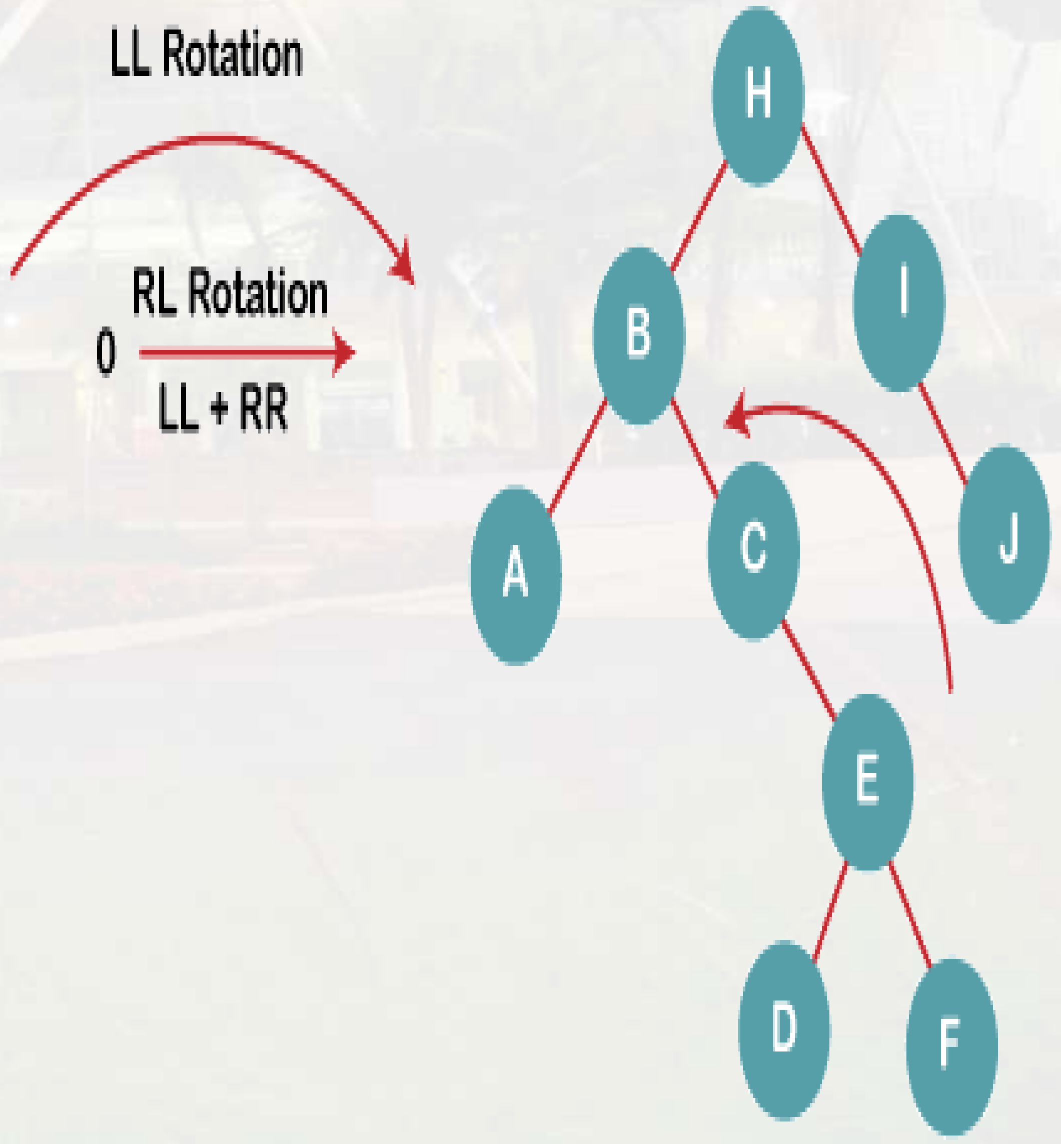
H, I, J, B, A, E, C, F, D, G, K, L

4. Insert C, F, D

4a) We first perform LL rotation on node E

The resultant tree after LL rotation is:





AVL Tree Example

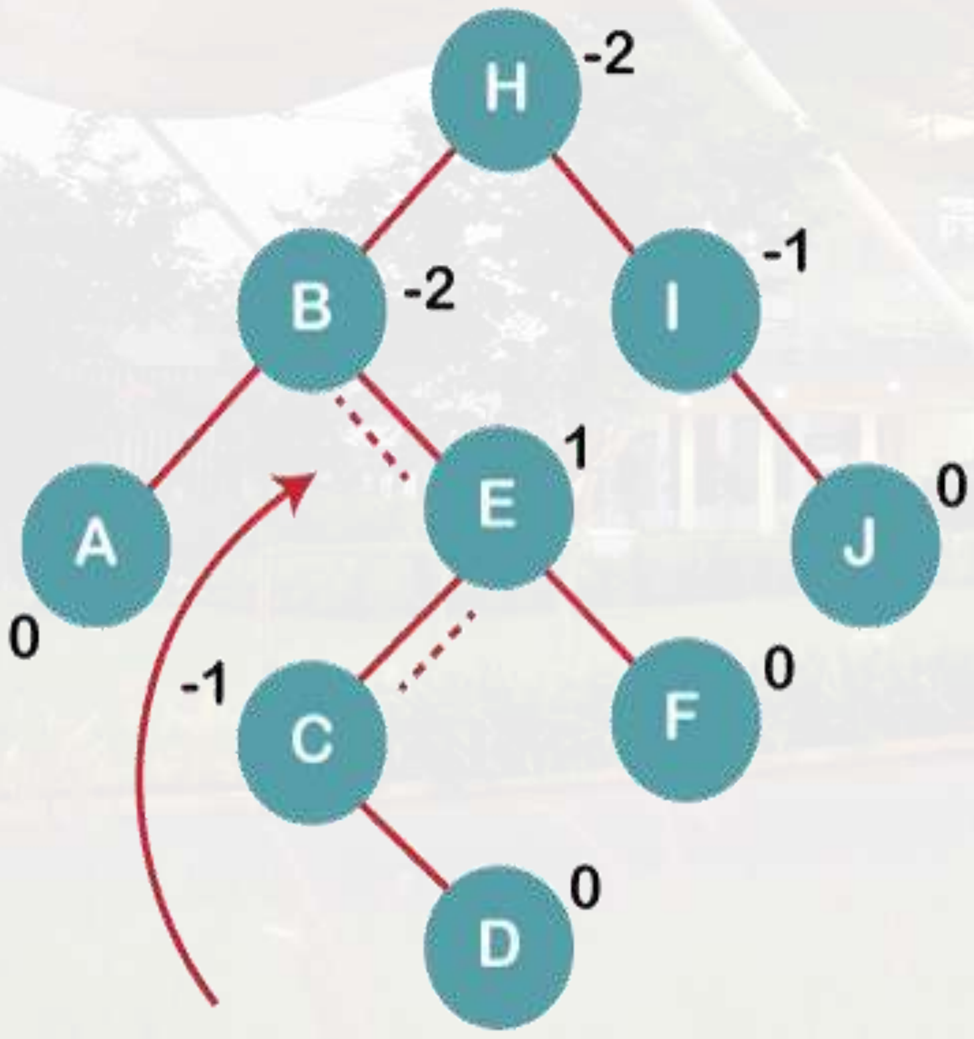
Q: Construct an AVL tree having the following elements

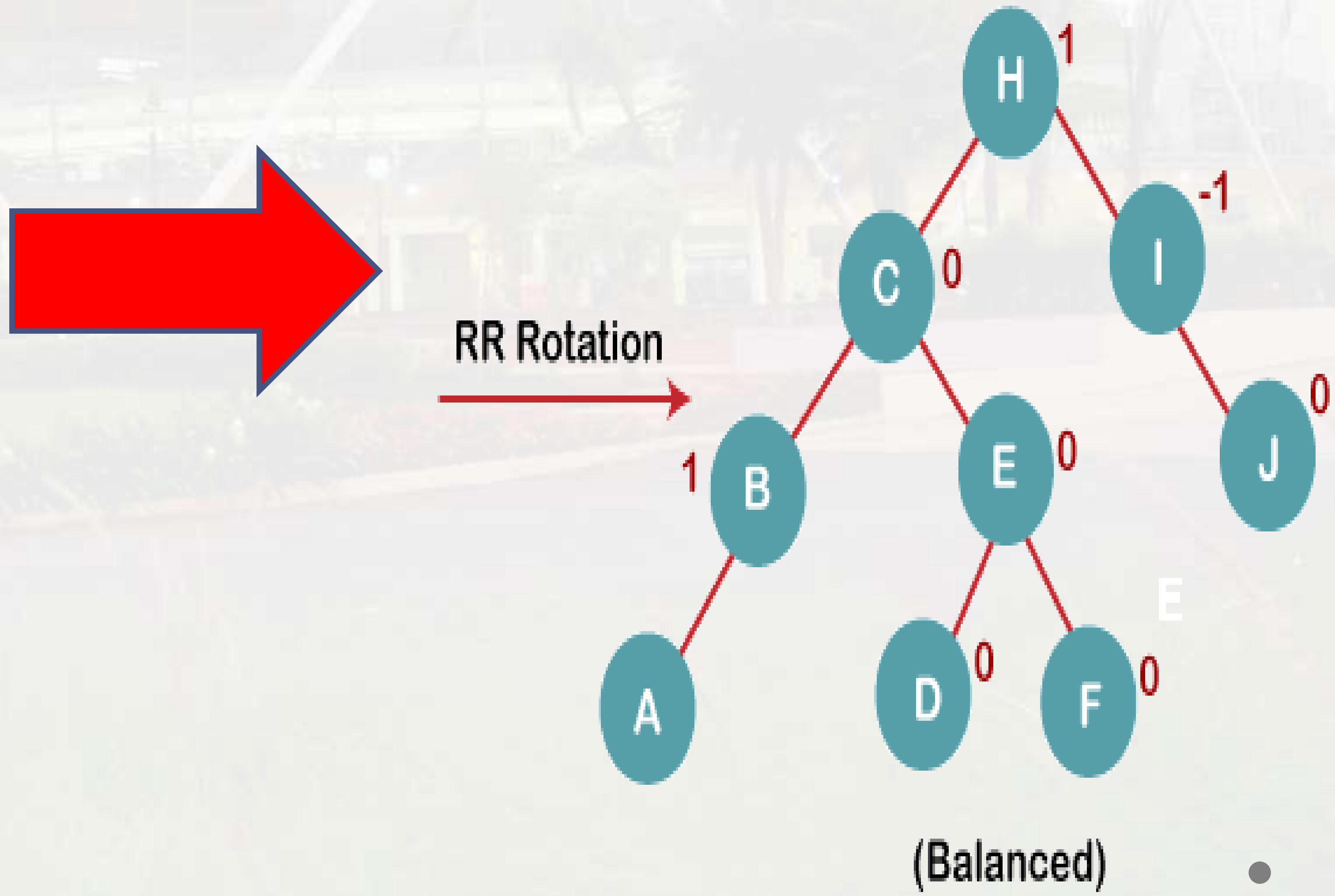
H, I, J, B, A, E, C, F, D, G, K, L

4. Insert C, F, D

4b) We then perform RR rotation on node B

The resultant balanced tree after RR rotation is:





AVL Tree Example

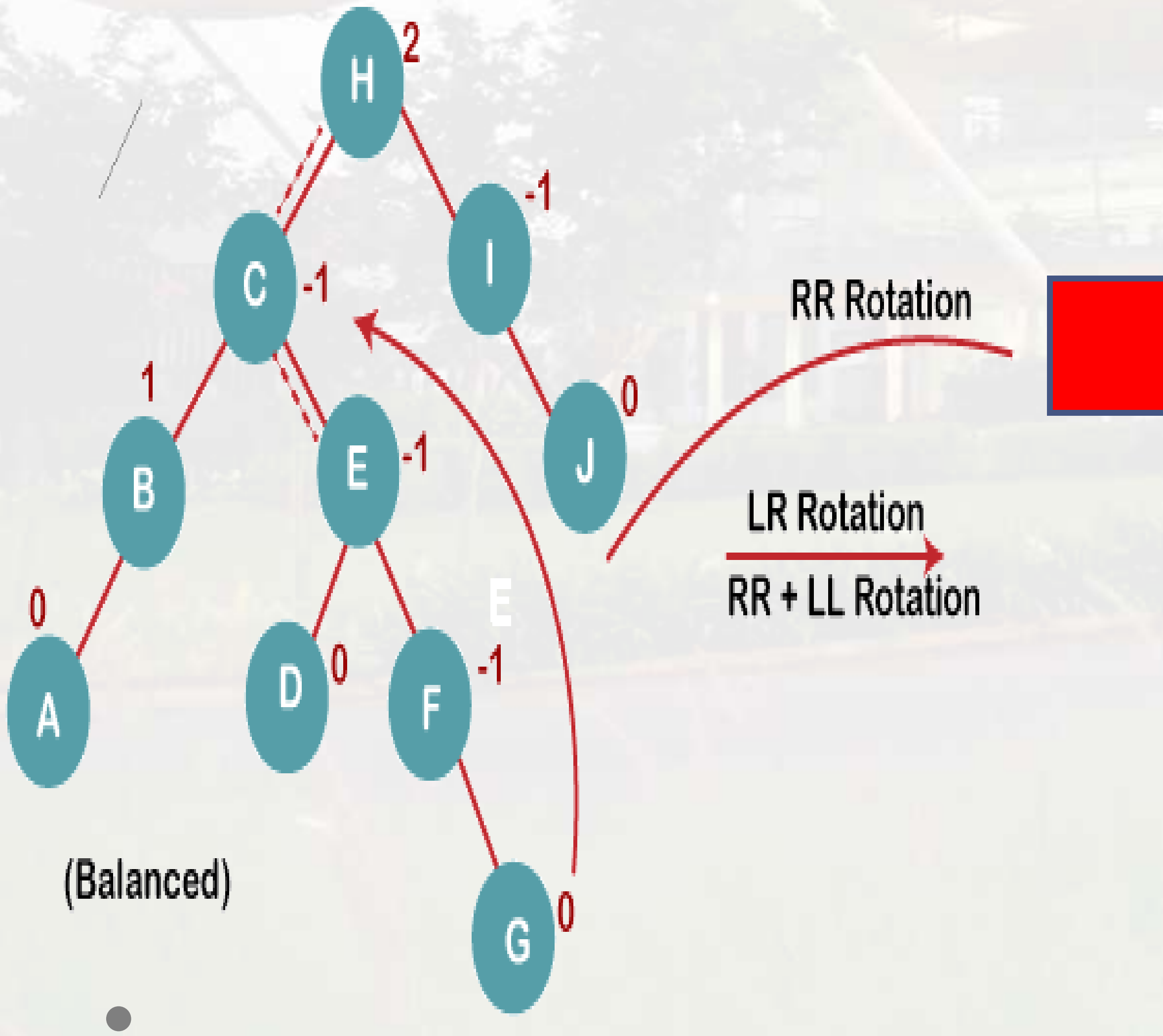
Q: Construct an AVL tree having the following elements

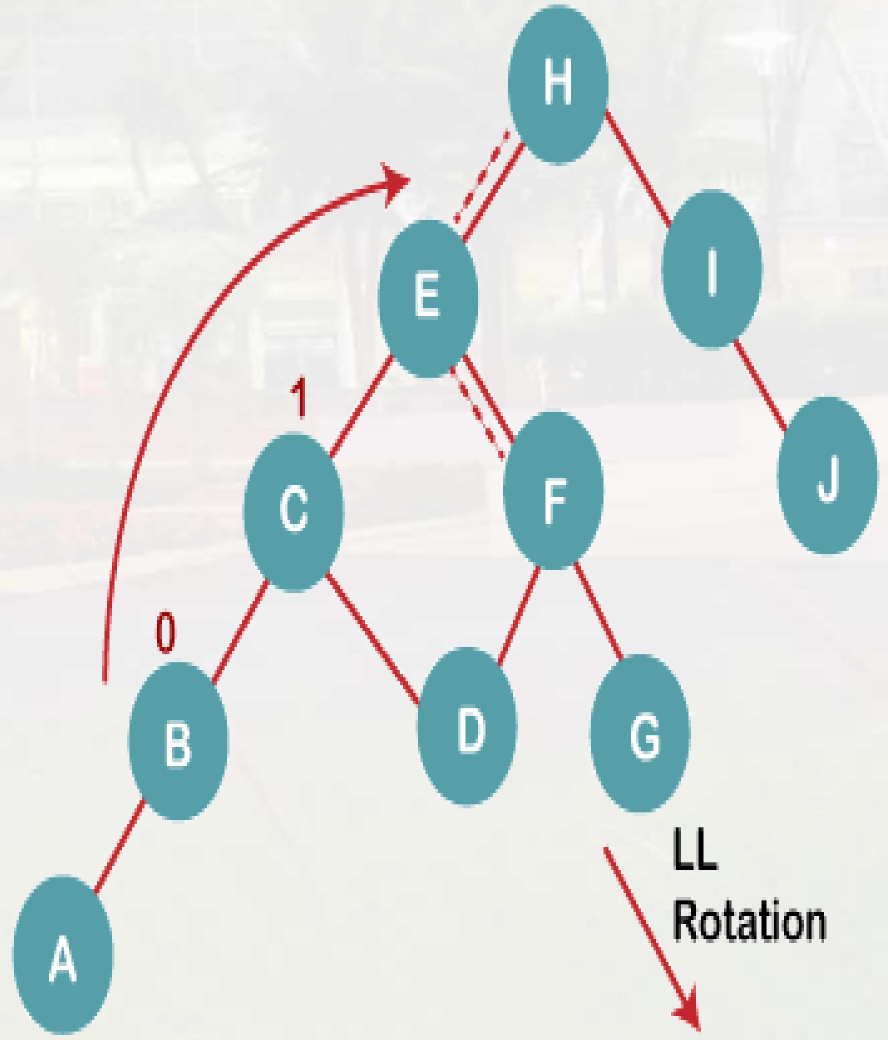
H, I, J, B, A, E, C, F, D, G, K, L

5. Insert G

5 a) We first perform RR rotation on node C

The resultant tree after RR rotation is:





AVL Tree Example

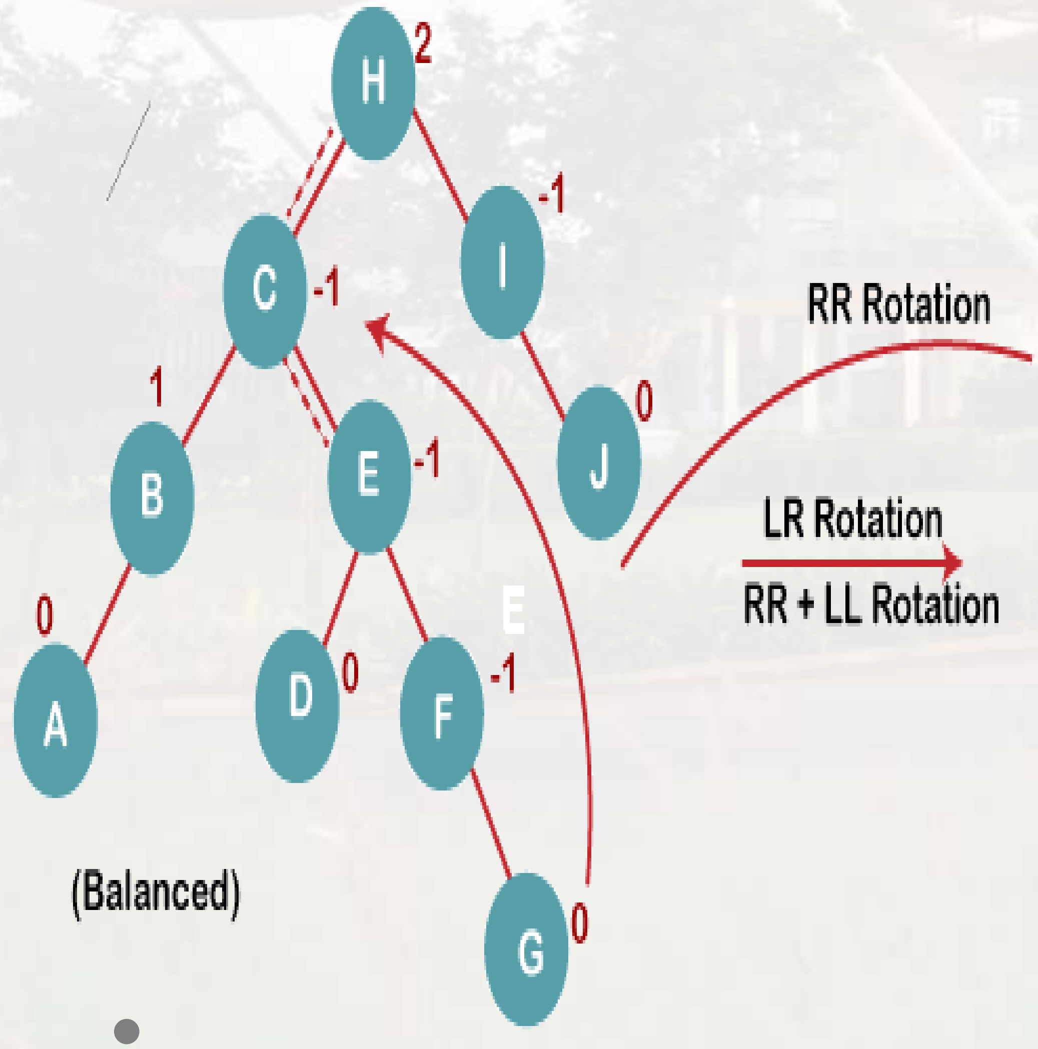
Q: Construct an AVL tree having the following elements

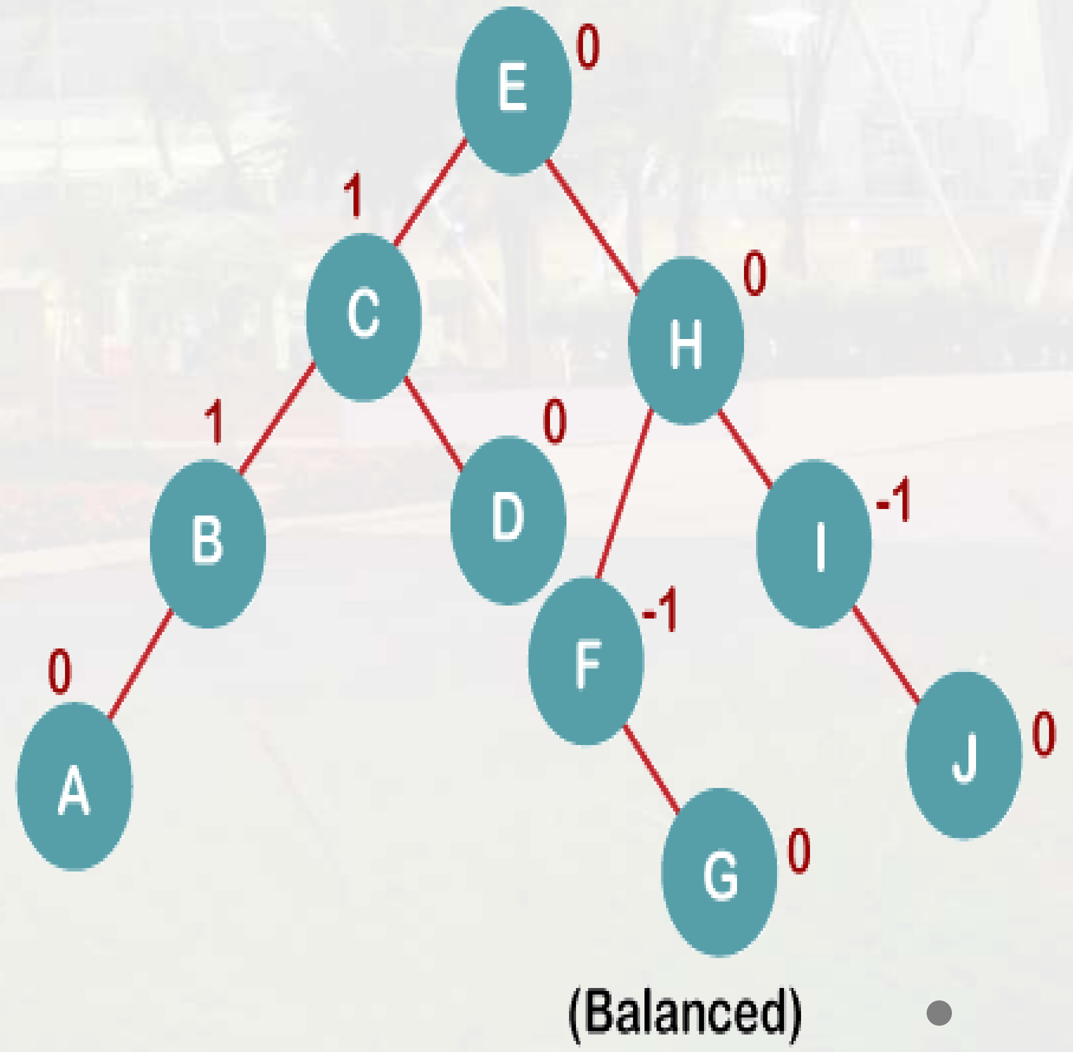
H, I, J, B, A, E, C, F, D, G, K, L

5. Insert G

5 b) We then perform LL rotation on node H

The resultant balanced tree after LL rotation is:



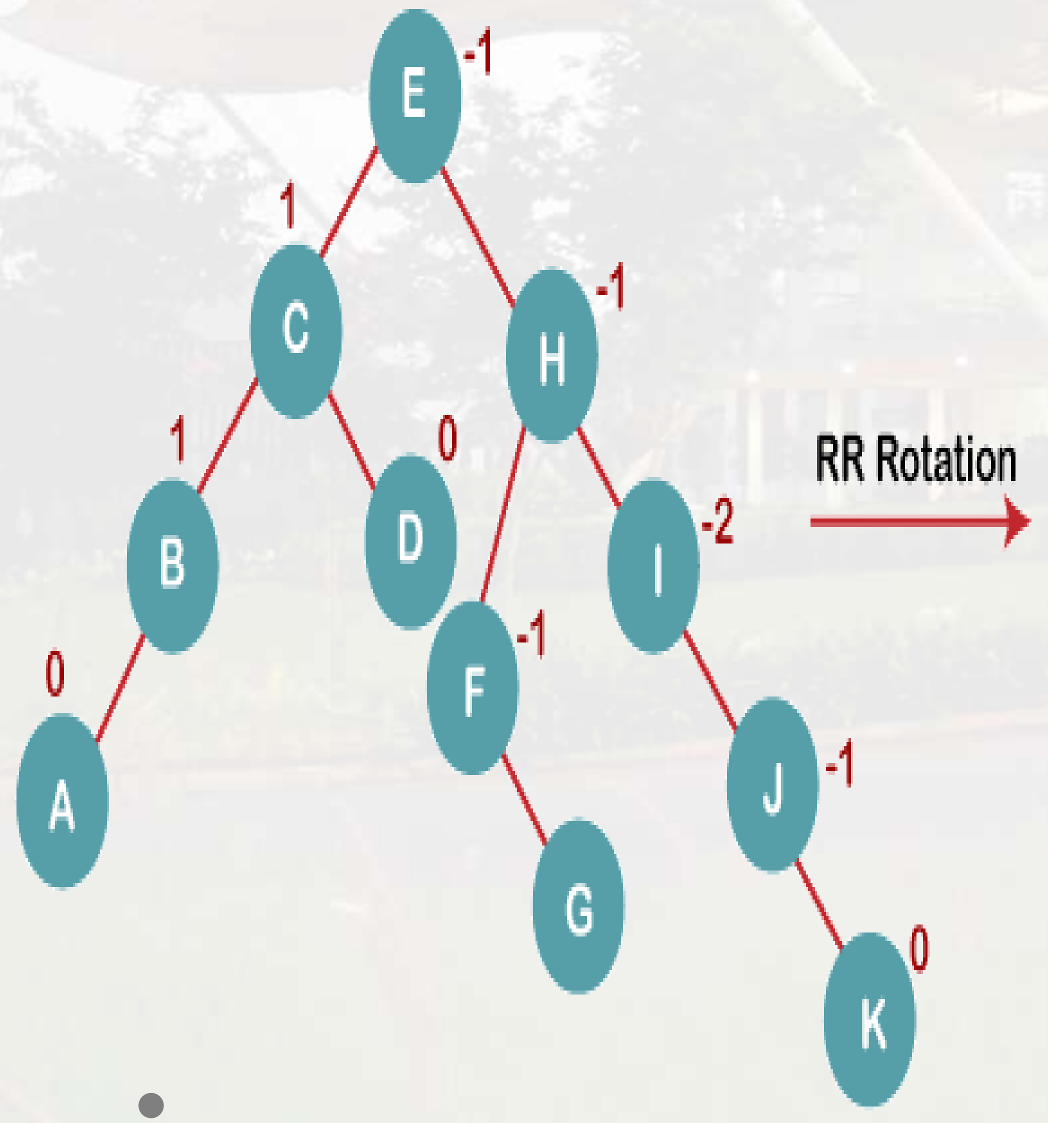


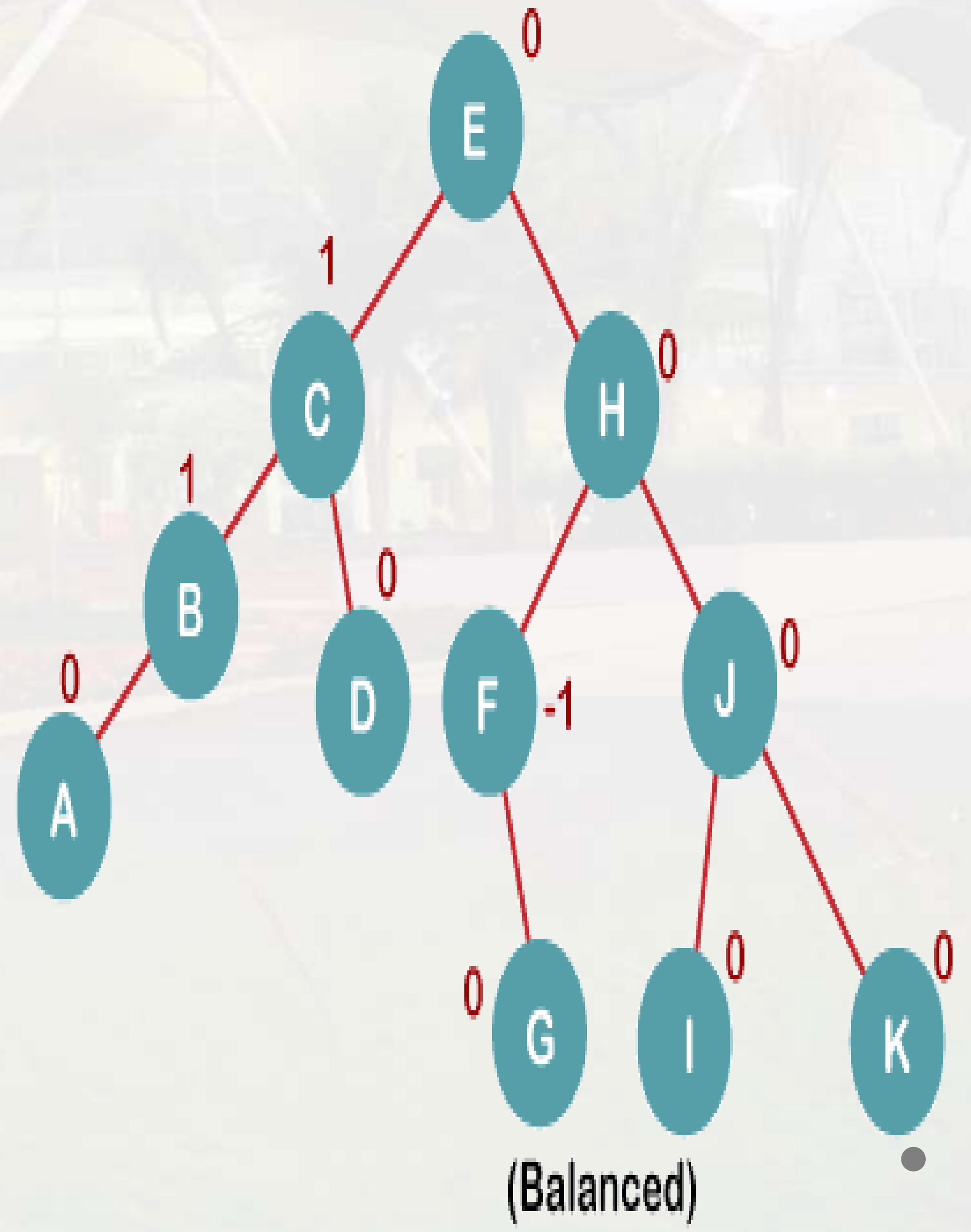
AVL Tree Example

Q: Construct an AVL tree having the following elements

H, I, J, B, A, E, C, F, D, G, K, L

6. Insert K



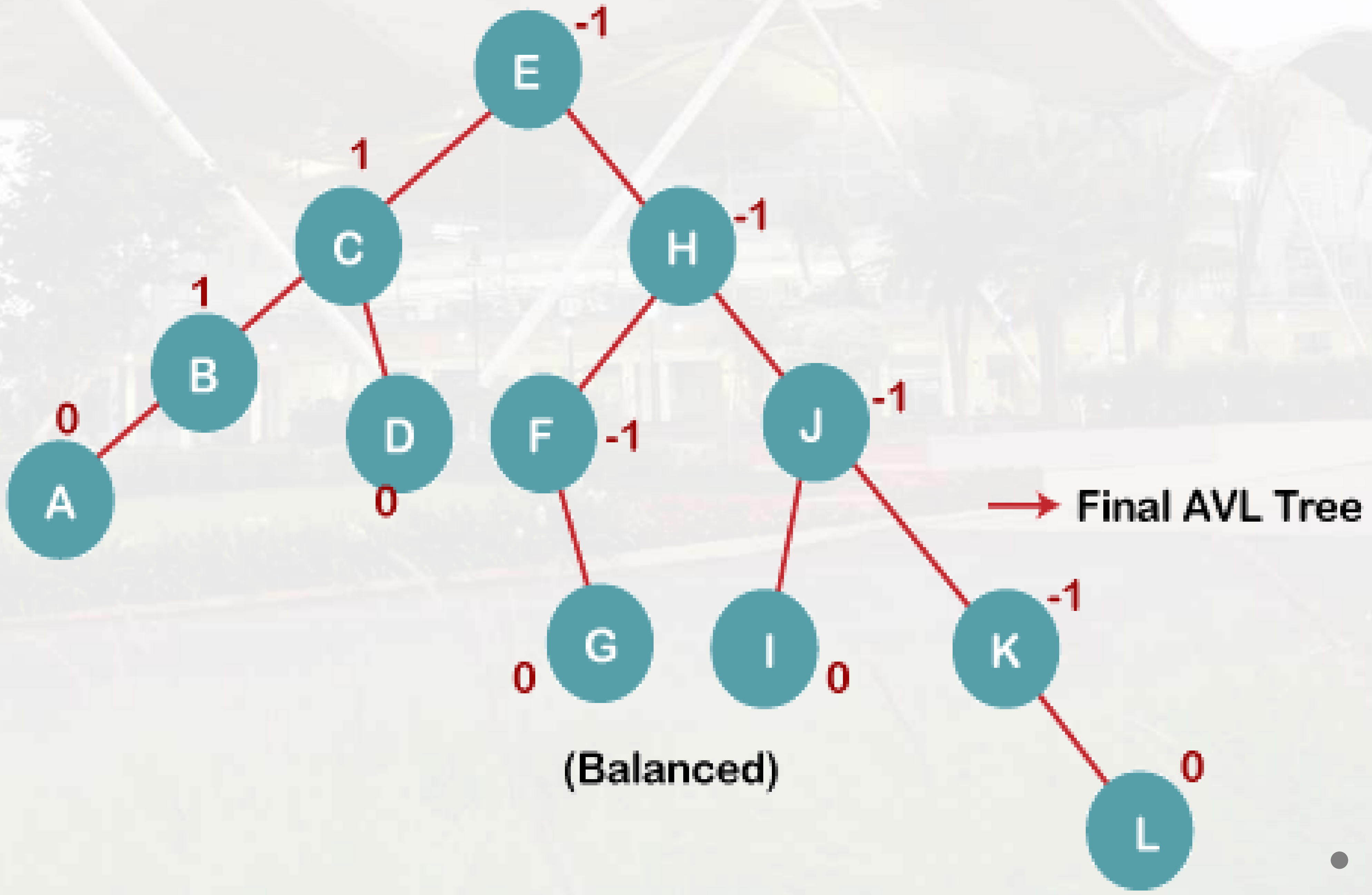


AVL Tree Example

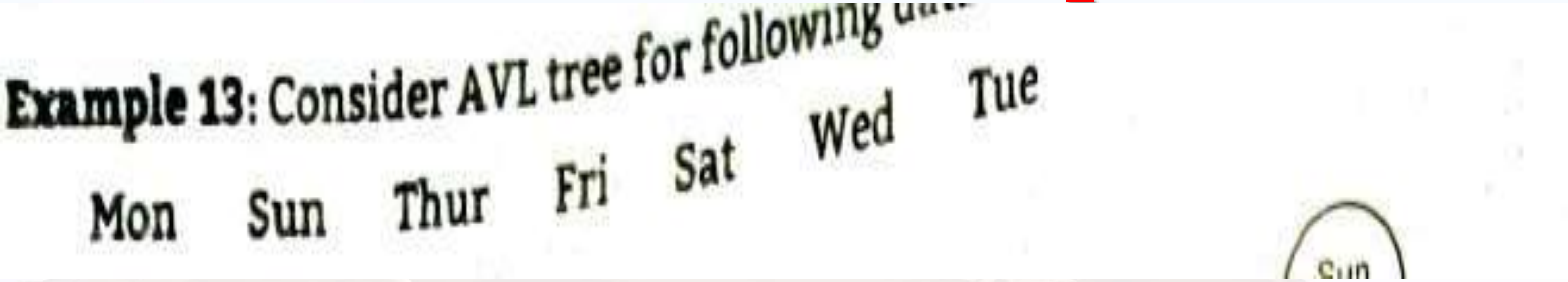
Q: Construct an AVL tree having the following elements

H, I, J, B, A, E, C, F, D, G, K, L

7. Insert L



AVL Tree Example



Example 13: Consider AVL tree for following

Mon Sun Thur Fri Sat Wed Tue

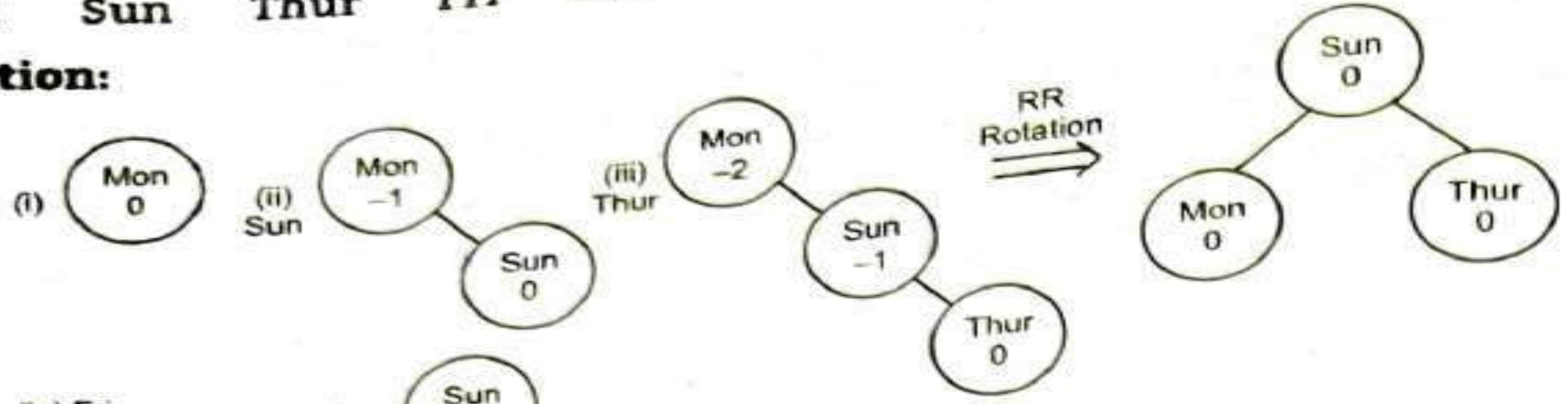
Sun

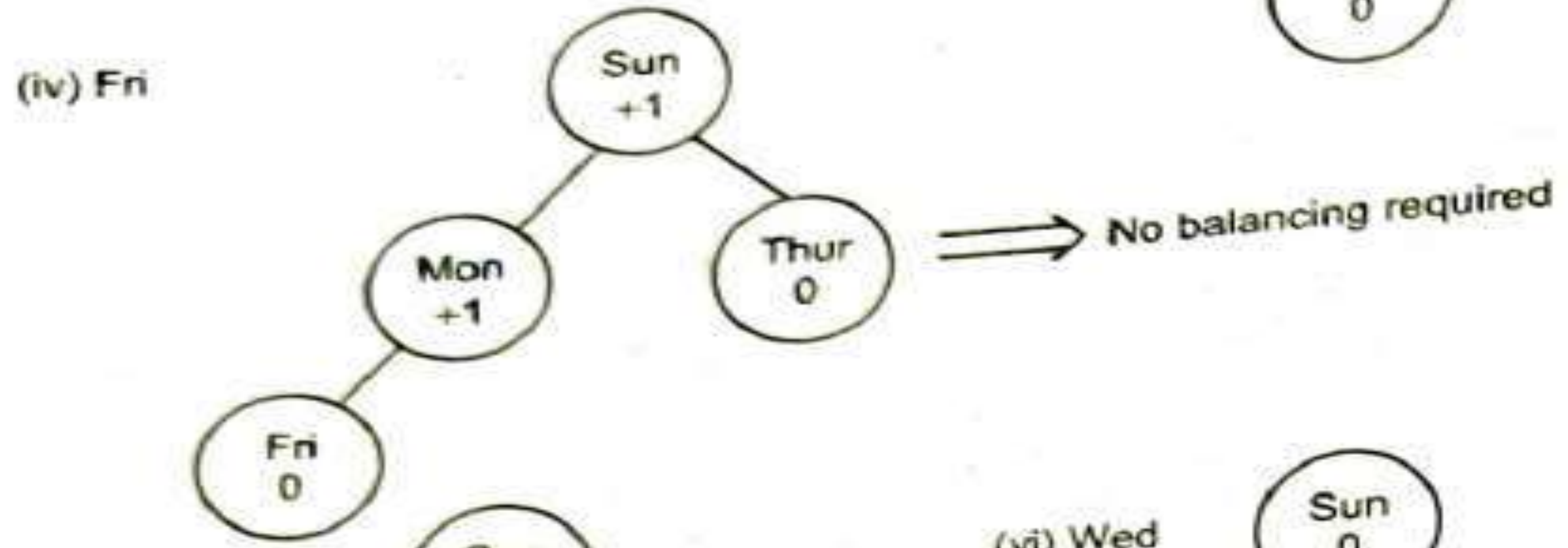
AVL Tree Example

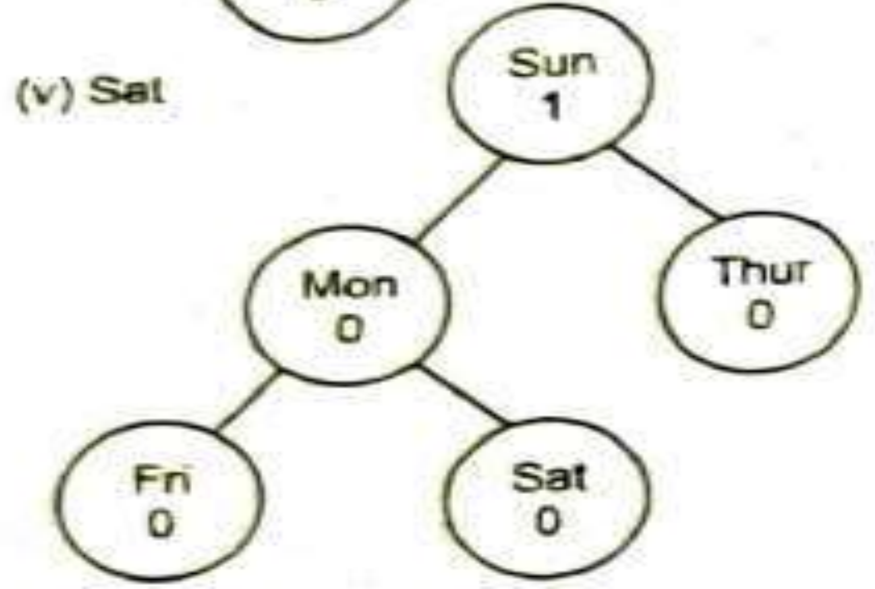
Example 13: Consider AVL tree for following

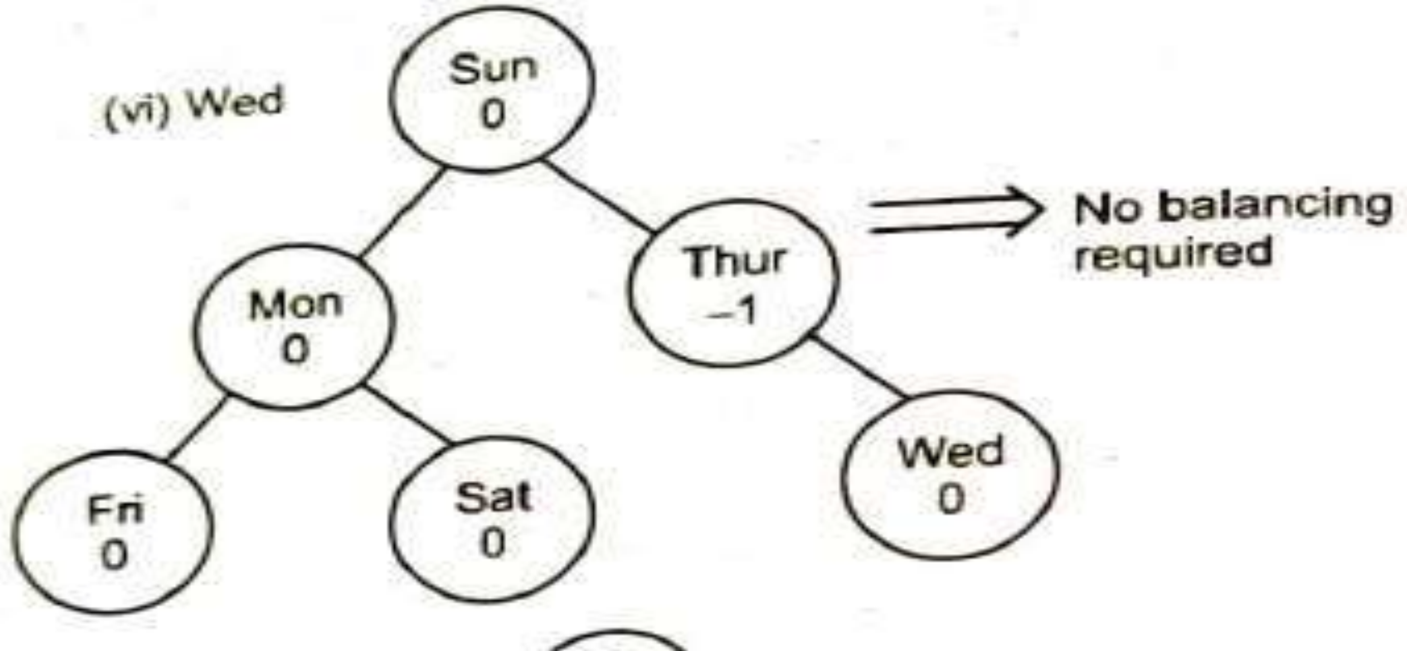
Mon Sun Thur Fri Sat Wed Tue

Solution:

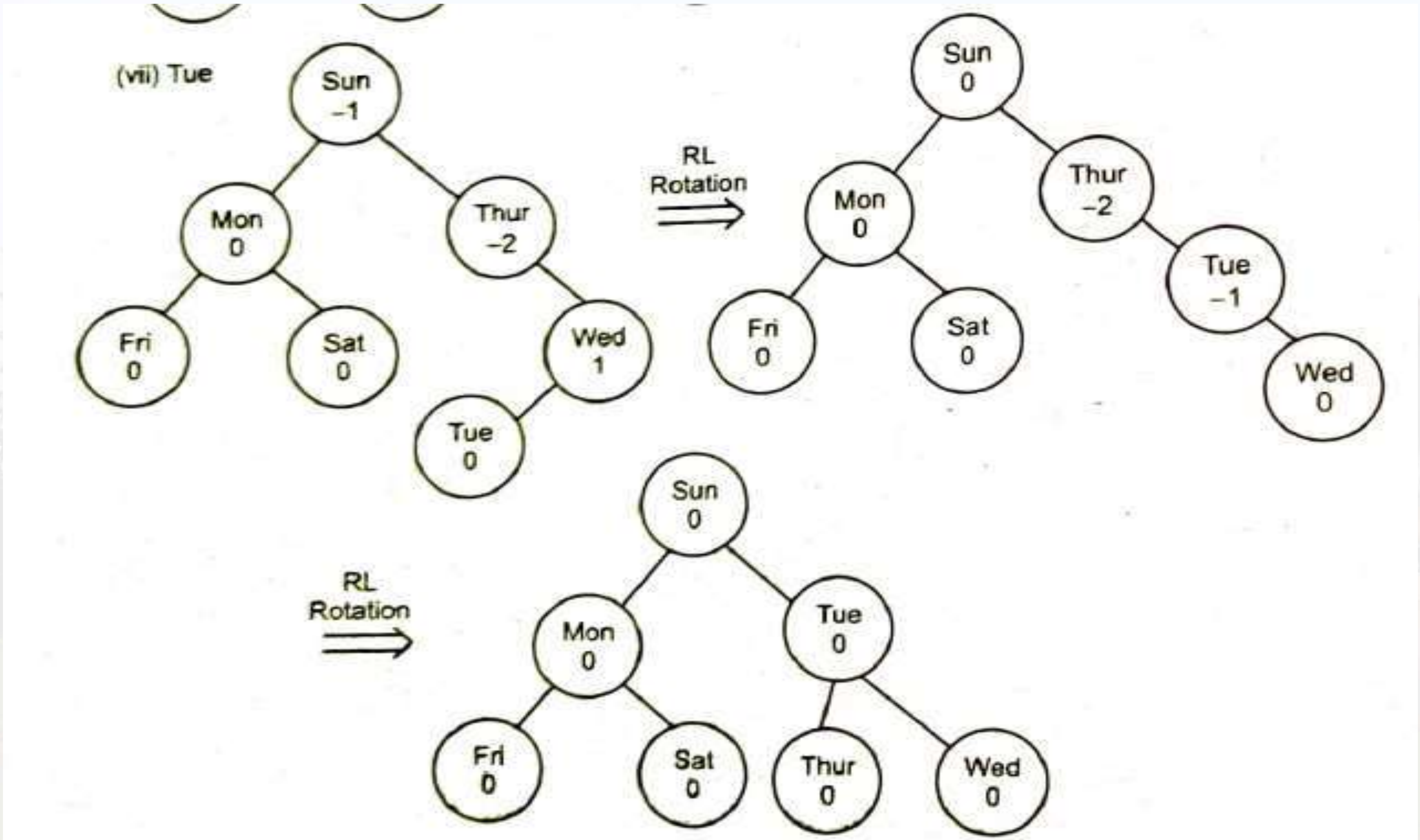






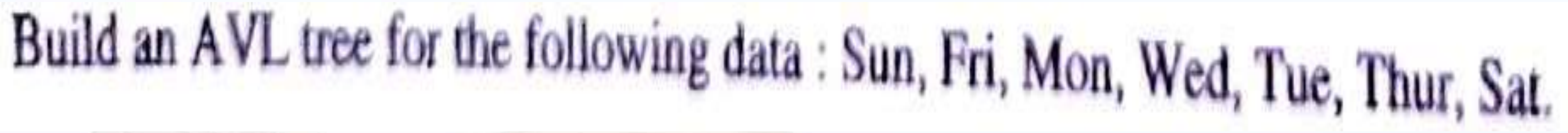


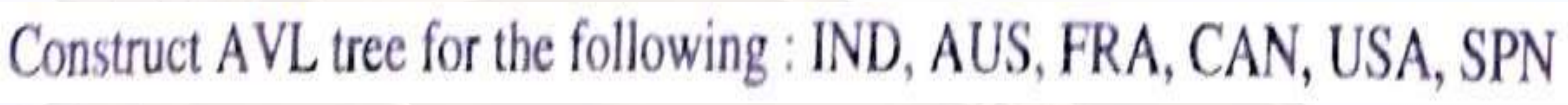
AVL Tree Example

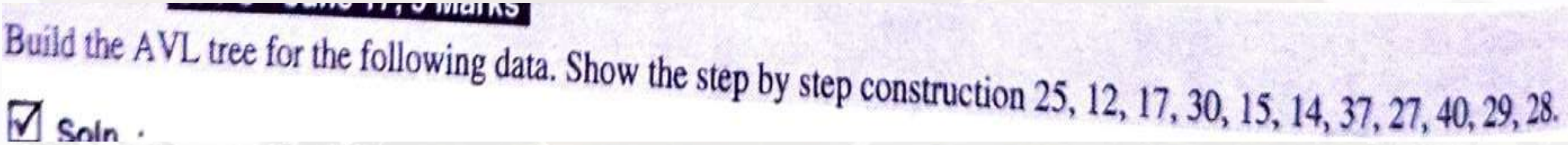


This is now balance tree.

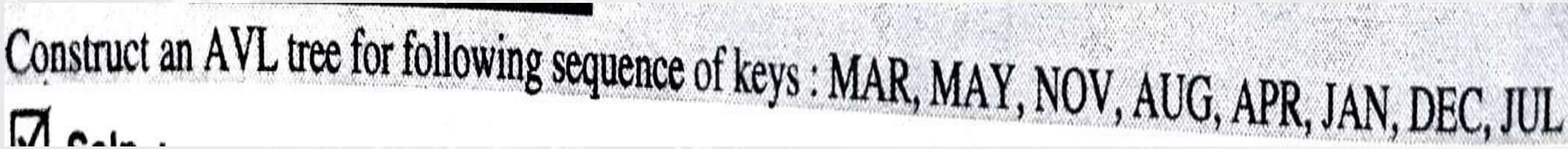
AVL Tree Example

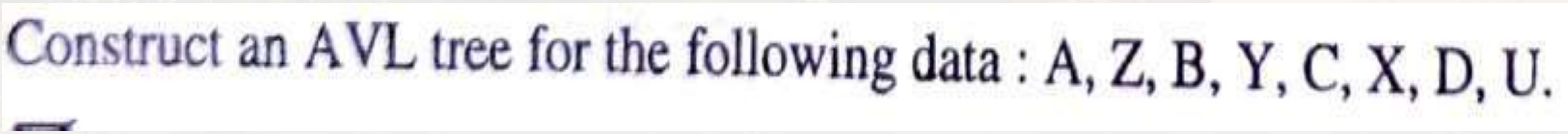

Build an AVL tree for the following data : Sun, Fri, Mon, Wed, Tue, Thur, Sat.

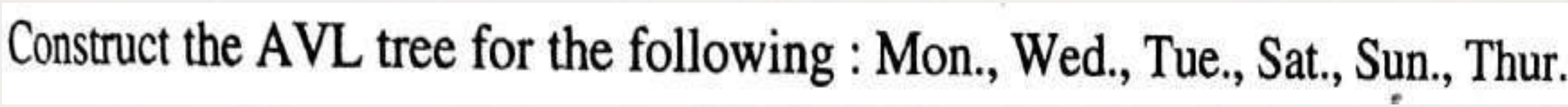

Construct AVL tree for the following : IND, AUS, FRA, CAN, USA, SPN

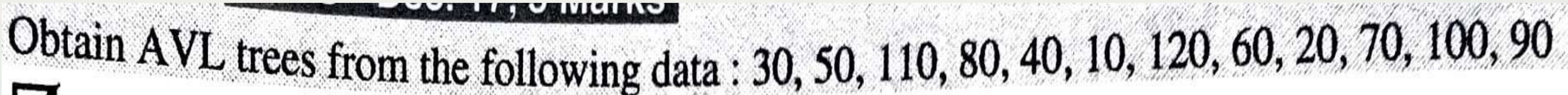

Build the AVL tree for the following data. Show the step by step construction 25, 12, 17, 30, 15, 14, 37, 27, 40, 29, 28.

☑ Soln. :


Construct an AVL tree for following sequence of keys : MAR, MAY, NOV, AUG, APR, JAN, DEC, JUL


Construct an AVL tree for the following data : A, Z, B, Y, C, X, D, U.


Construct the AVL tree for the following : Mon., Wed., Tue., Sat., Sun., Thur.


Obtain AVL trees from the following data : 30, 50, 110, 80, 40, 10, 120, 60, 20, 70, 100, 90

Red Black Tree

Red-Black tree is a self-balancing binary search tree in which each node contains an extra bit for denoting the color of the node, either red or black.

A red-black tree satisfies the following properties:

1. **Red/Black Property:** Every node is colored, either red or black.
2. **Root Property:** The root is black.
3. **Leaf Property:** Every leaf (NIL) is black.

Red Black Tree

4. **Red Property:** If a red node has children then, the children are always black.
5. **Depth Property:** For each node, any simple path from this node to any of its descendant leaf has the same black-depth (the number of black nodes).

Red Black Tree

Rules That Every Red-Black Tree Follows:

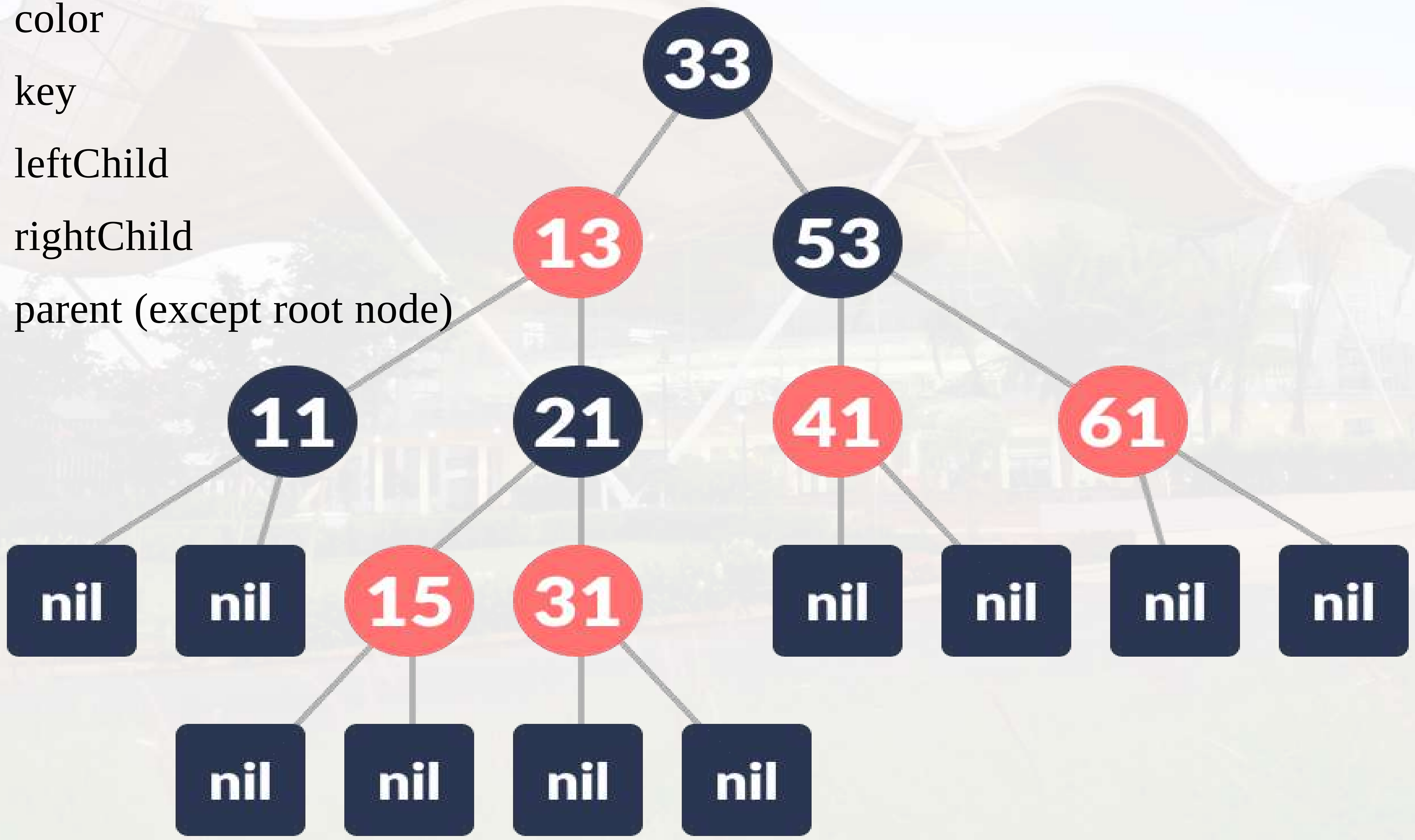
1. Every node has a colour either red or black.
2. The root of the tree is always black.
3. There are no two adjacent red nodes (A red node cannot have a red parent or red child).
4. Every path from a node (including root) to any of its descendants NULL nodes has the same number of black nodes.
5. All leaf nodes are black nodes.

Red Black Tree

An example of a red-black tree is:

Each node has the following attributes:

1. color
2. key
3. leftChild
4. rightChild
5. parent (except root node)



Red Black Tree Operations

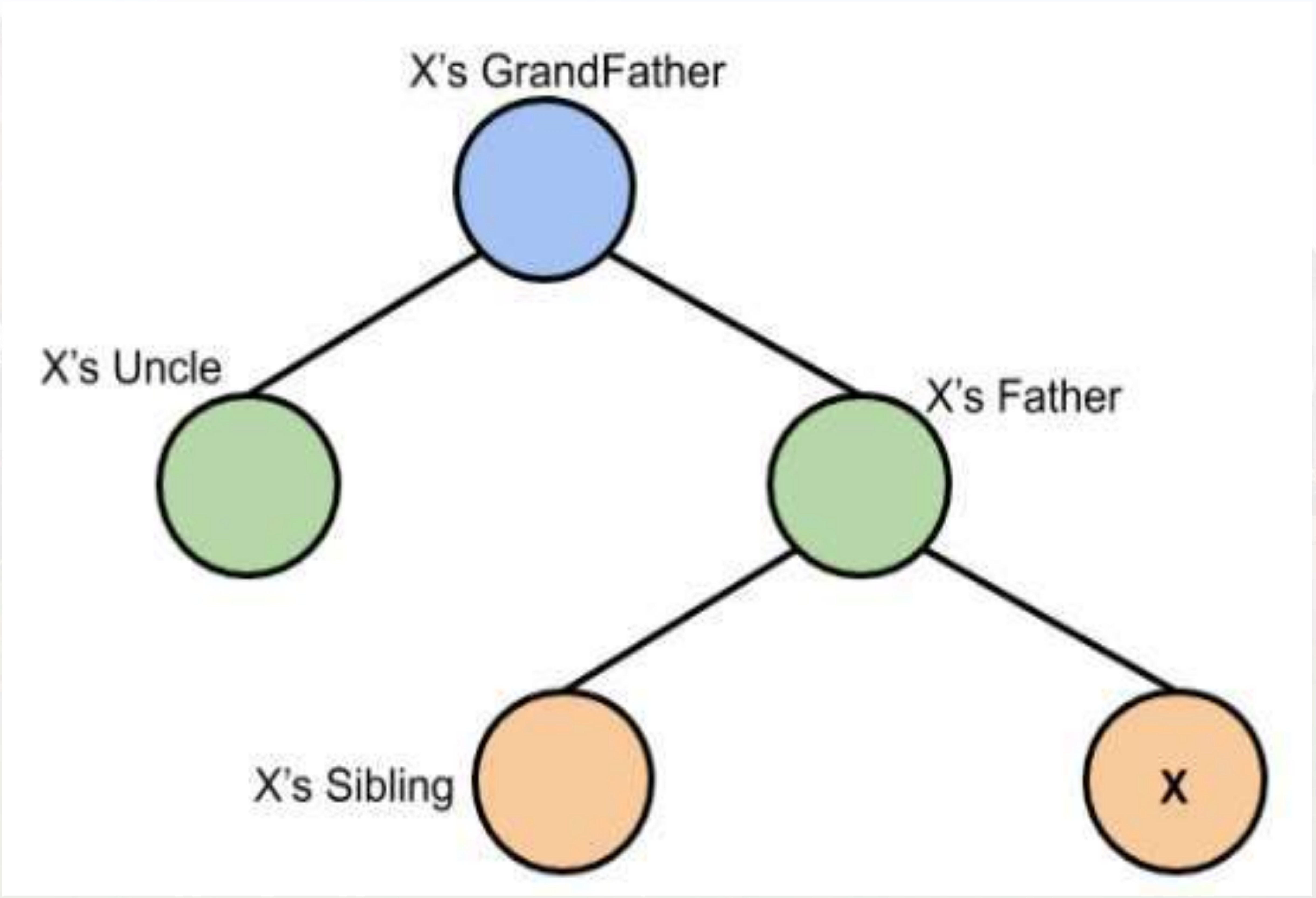
In AVL tree insertion, we used rotation as a tool to do balancing after insertion. In the Red-Black tree, we use two tools to do the balancing.

1. **Recoloring**
2. **Rotation**

Recolouring is the change in colour of the node i.e. if it is red then change it to black and vice versa. It must be noted that the colour of the NULL node is always black. Moreover, we always try recolouring first, if recolouring doesn't work, then we go for rotation. Following is a detailed algorithm. The algorithms have mainly two cases depending upon the colour of the uncle. If the uncle is red, we do recolour. If the uncle is black, we do rotations and/or recolouring.

Red Black Tree Operations

The representation we will be working with is :

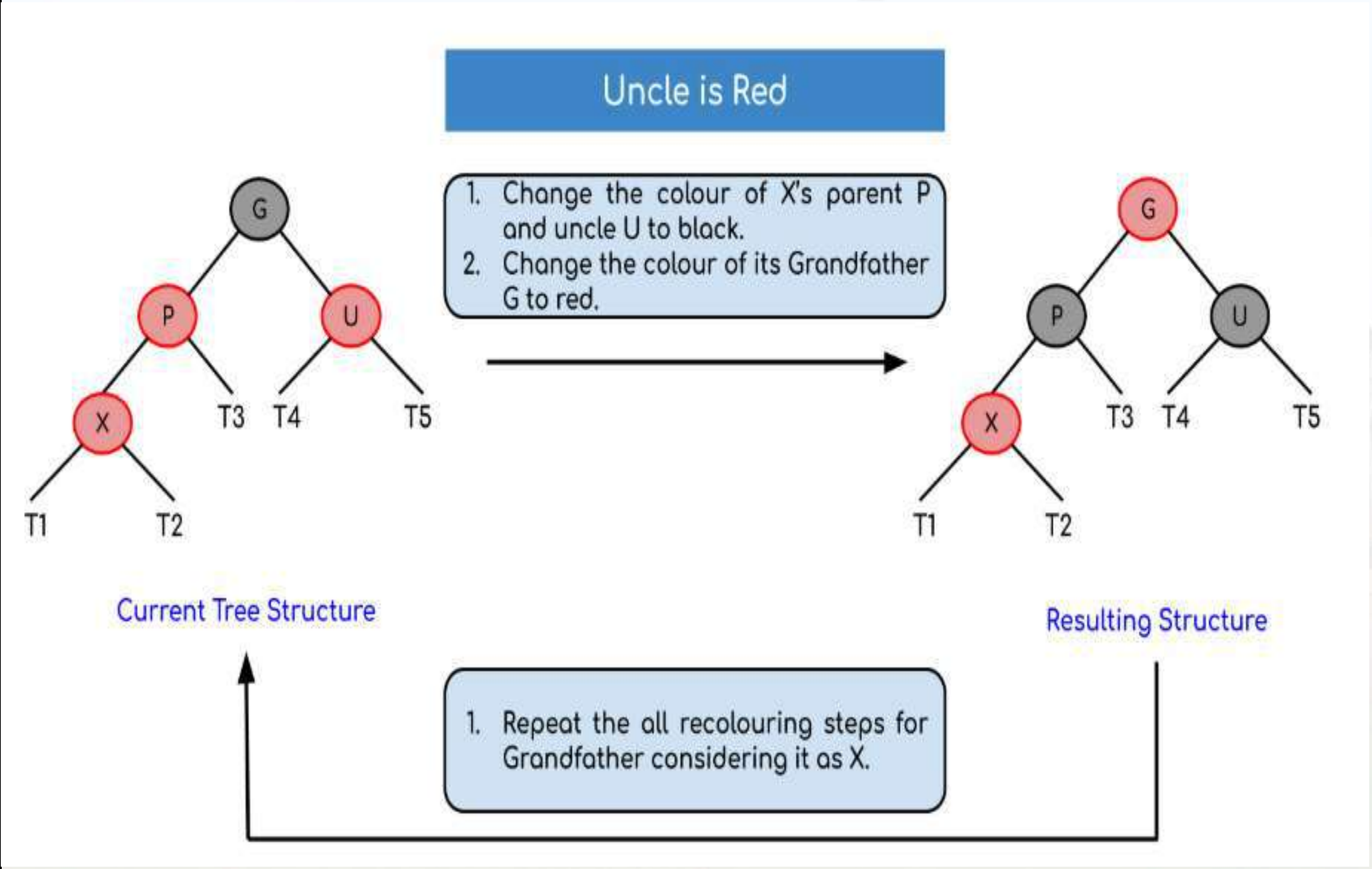


Red Black Tree Operations

Logic:

First, you have to insert the node similarly to that in a binary tree and assign a red colour to it. Now, if the node is a root node then change its colour to black, but if it is not then check the colour of the parent node. If its colour is black then don’t change the colour but if it is not i.e. it is red then check the colour of the node’s uncle. If the node’s uncle has a red colour then change the colour of the node’s parent and uncle to black and that of grandfather to red colour and repeat the same process for him (i.e. grandfather).

Red Black Tree Operations



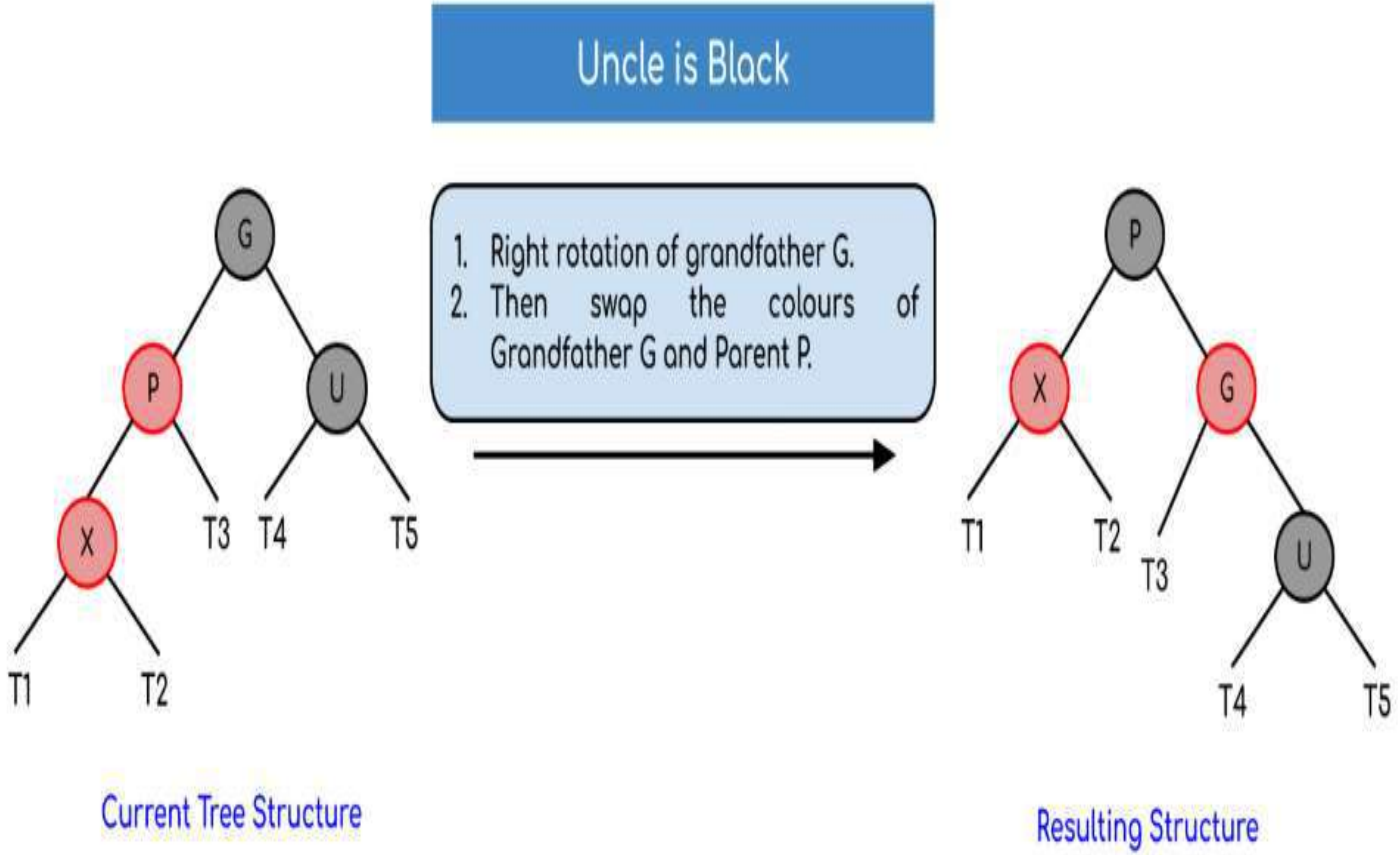
1. Change the colour of X's parent P and uncle U to black.
2. Change the colour of its Grandfather G to red.

1. Repeat the all recolouring steps for Grandfather considering it as X.

But, if the node's uncle has black colour then there are 4 possible cases:

Red Black Tree Operations

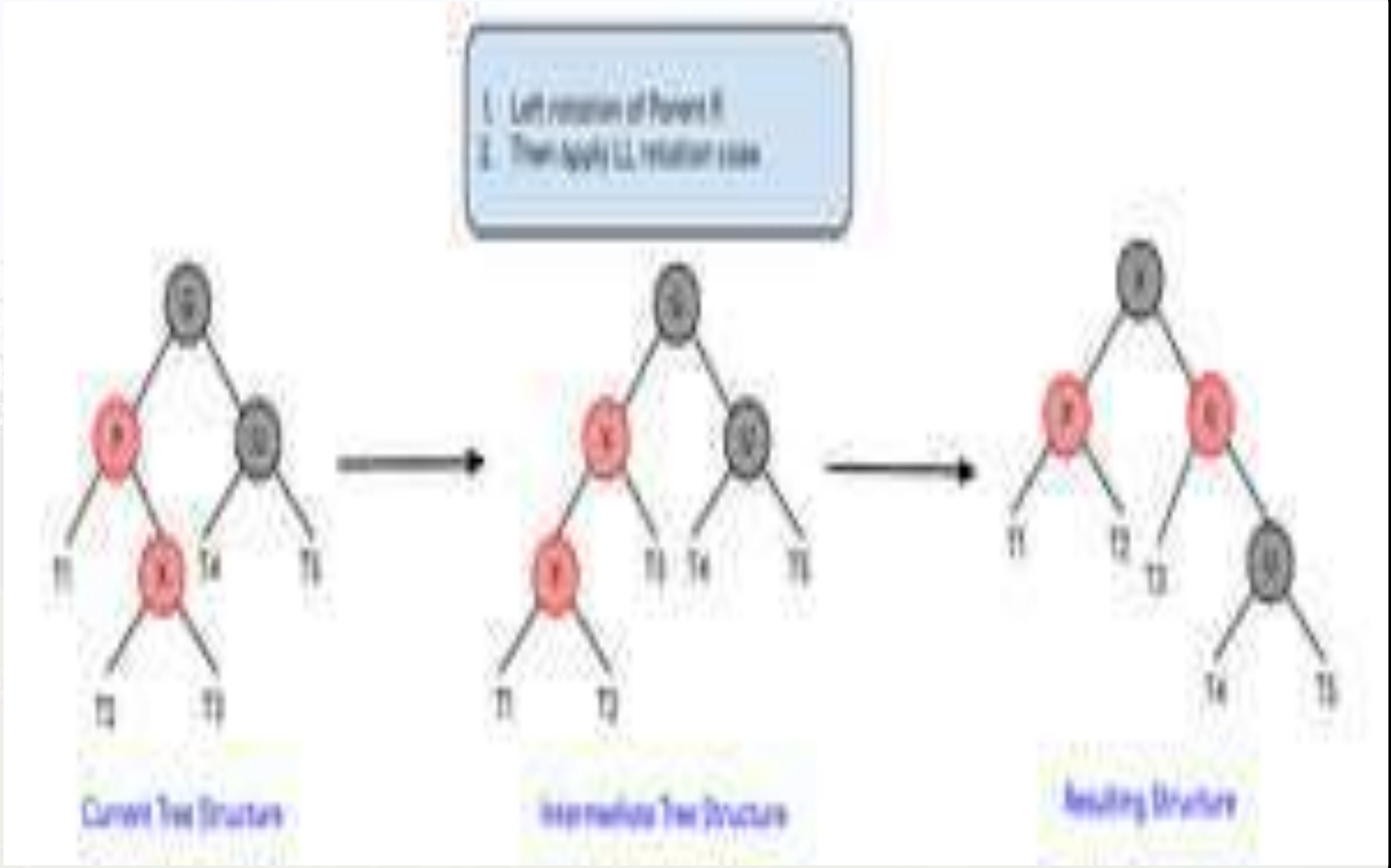
Left Left Case (LL rotation):



1. Right rotation of grandfather G.
2. Then swap the colours of Grandfather G and Parent P.

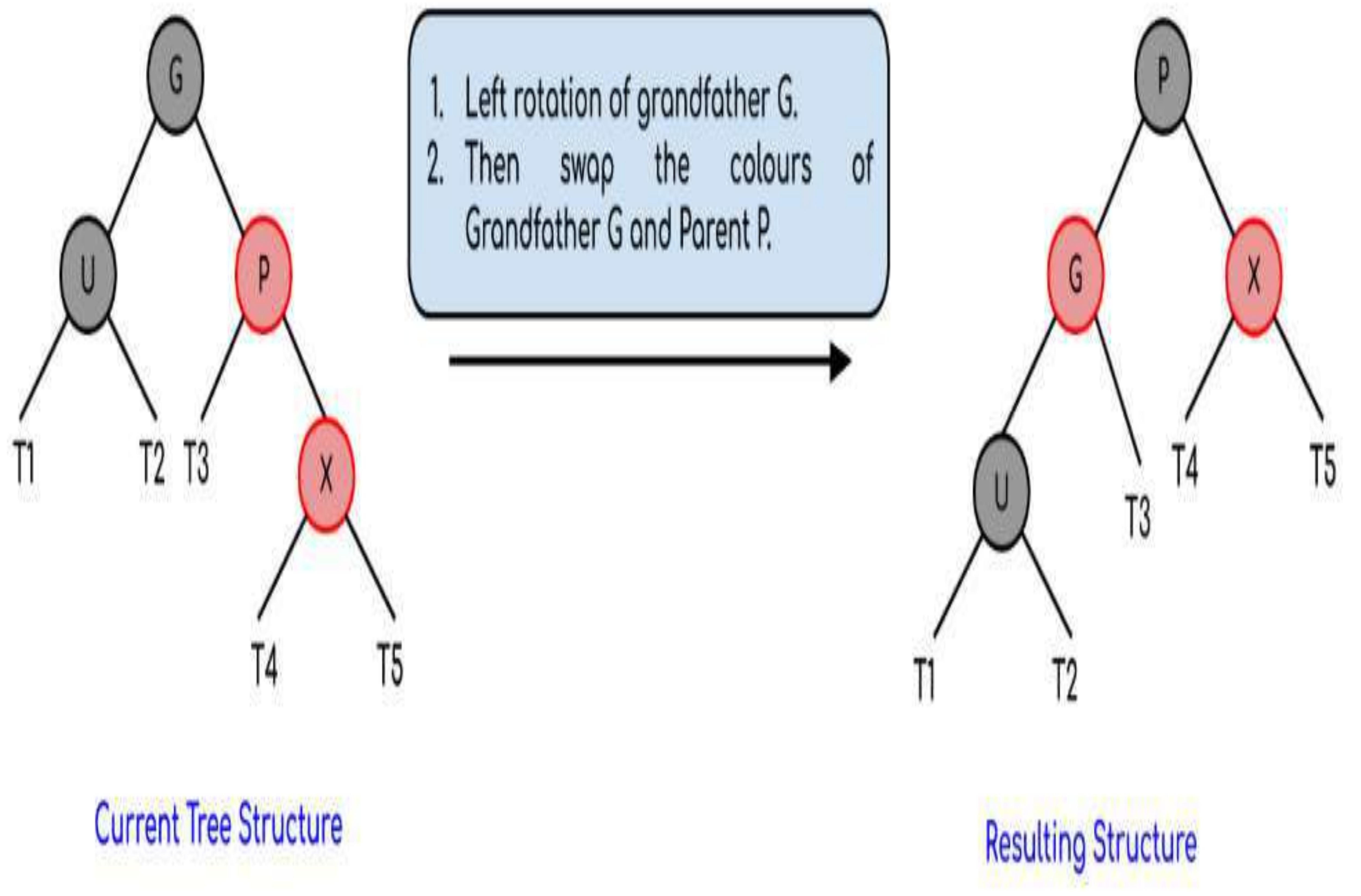
Red Black Tree Operations

Left Right Case (LR rotation):


1. Left rotation of Parent P
2. Then Apply LL rotation case
T1
T2
T3
T4
T5
Current Tree Structure
Intermediate Tree Structure
Resulting Structure

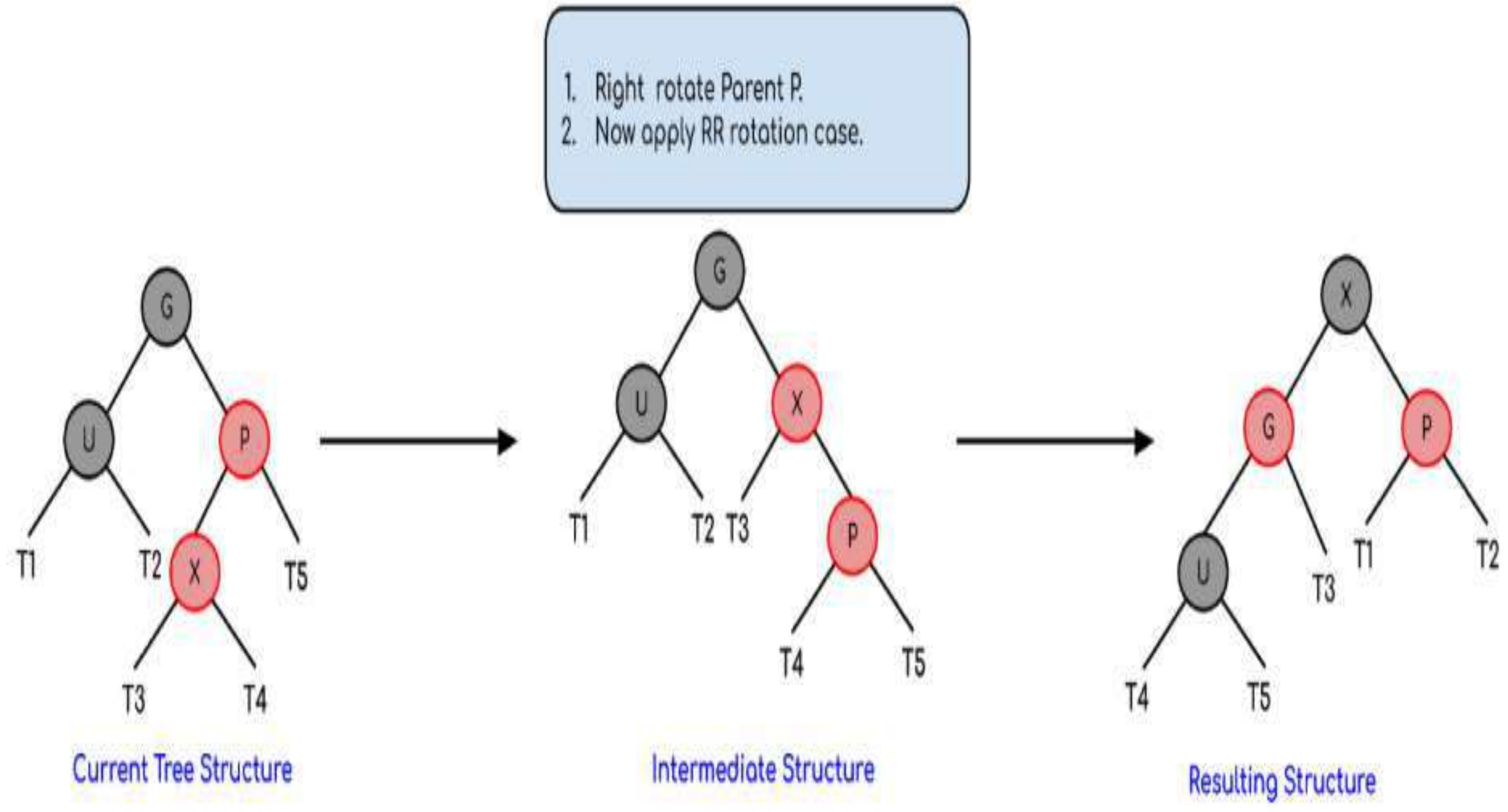
Red Black Tree Operations

Right to Right Case (RR rotation):


G
U
P
X
T1
T2
T3
T4
T5
1. Left rotation of grandfather G.
2. Then swap the colours of Grandfather G and Parent P.
P
G
X
U
T3
T4
T5
T1
T2
Current Tree Structure
Resulting Structure

Red Black Tree Operations

Right Left Case (RL rotation):



Now, after these rotations, if the colours of the nodes are miss matching then recolour them.

ALGORITHMS OF RBT

Algorithm:

Let x be the newly inserted node.

Perform standard BST insertion and make the colour of newly inserted nodes as RED.

If x is the root, change the colour of x as BLACK (Black height of complete tree increases by 1).

Do the following if the color of x's parent is not BLACK and x is not the root.

a) If x's uncle is RED (Grandparent must have been black from property 4)

(i) Change the colour of parent and uncle as BLACK.

(ii) Colour of a grandparent as RED.

(iii) Change x = x's grandparent, repeat steps 2 and 3 for new x.

b) If x's uncle is BLACK, then there can be four configurations for x, x's parent (p) and x's grandparent (g) (This is similar to AVL Tree)

(i) Left Left Case (p is left child of g and x is left child of p)

(ii) Left Right Case (p is left child of g and x is the right child of p)

(iii) Right Right Case (Mirror of case i)

(iv) Right Left Case (Mirror of case ii)

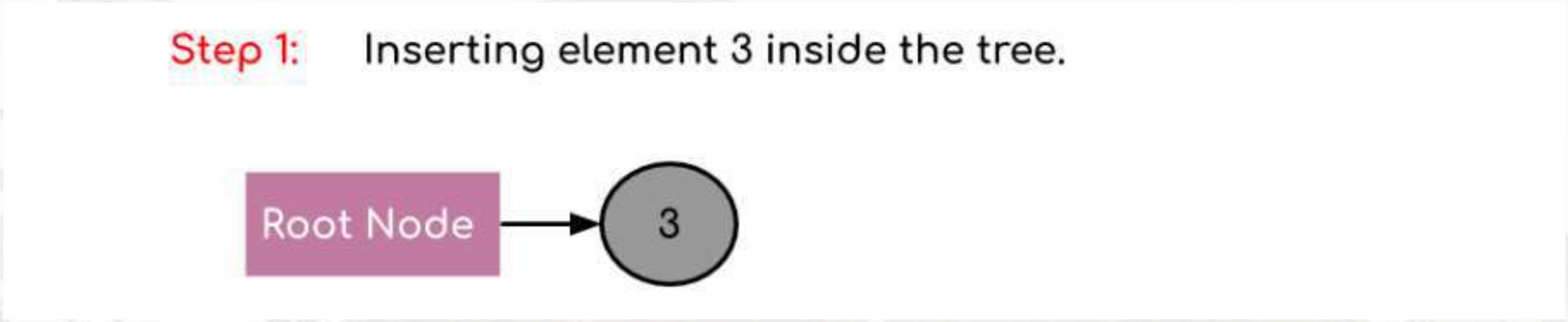
APPLICATION OF RBT

1. Most of the self-balancing BST library functions like map and set in C++ (OR TreeSet and TreeMap in Java) use Red-Black Tree.
2. It is used to implement CPU Scheduling Linux. Completely Fair Scheduler uses it.
3. Besides they are used in the K-mean clustering algorithm for reducing time complexity.
4. Moreover, MySQL also uses the Red-Black tree for indexes on tables.

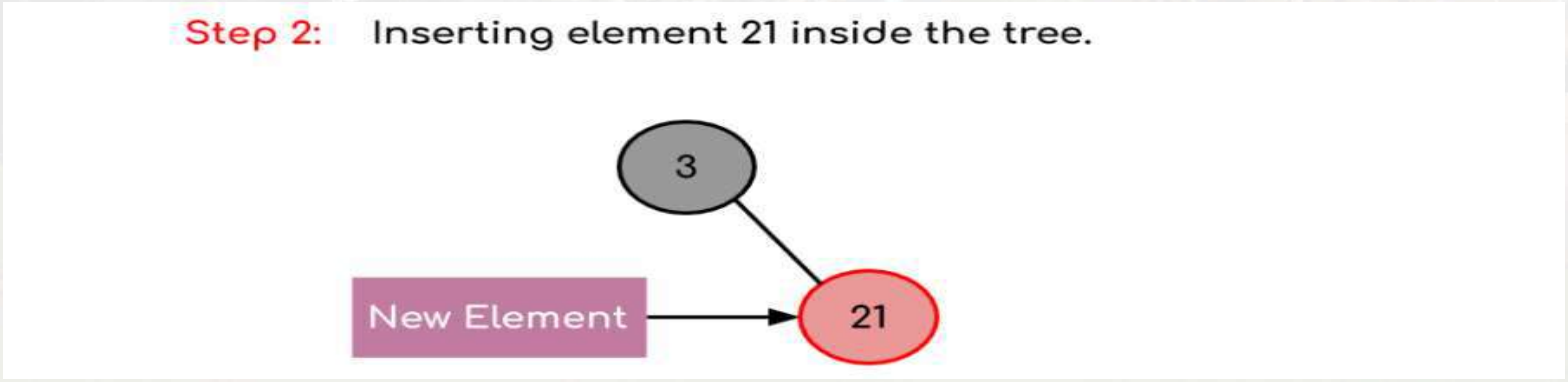
EXAMPLE OF RBT

Example: Creating a red-black tree with elements 3, 21, 32 and 15 in an empty tree.

:



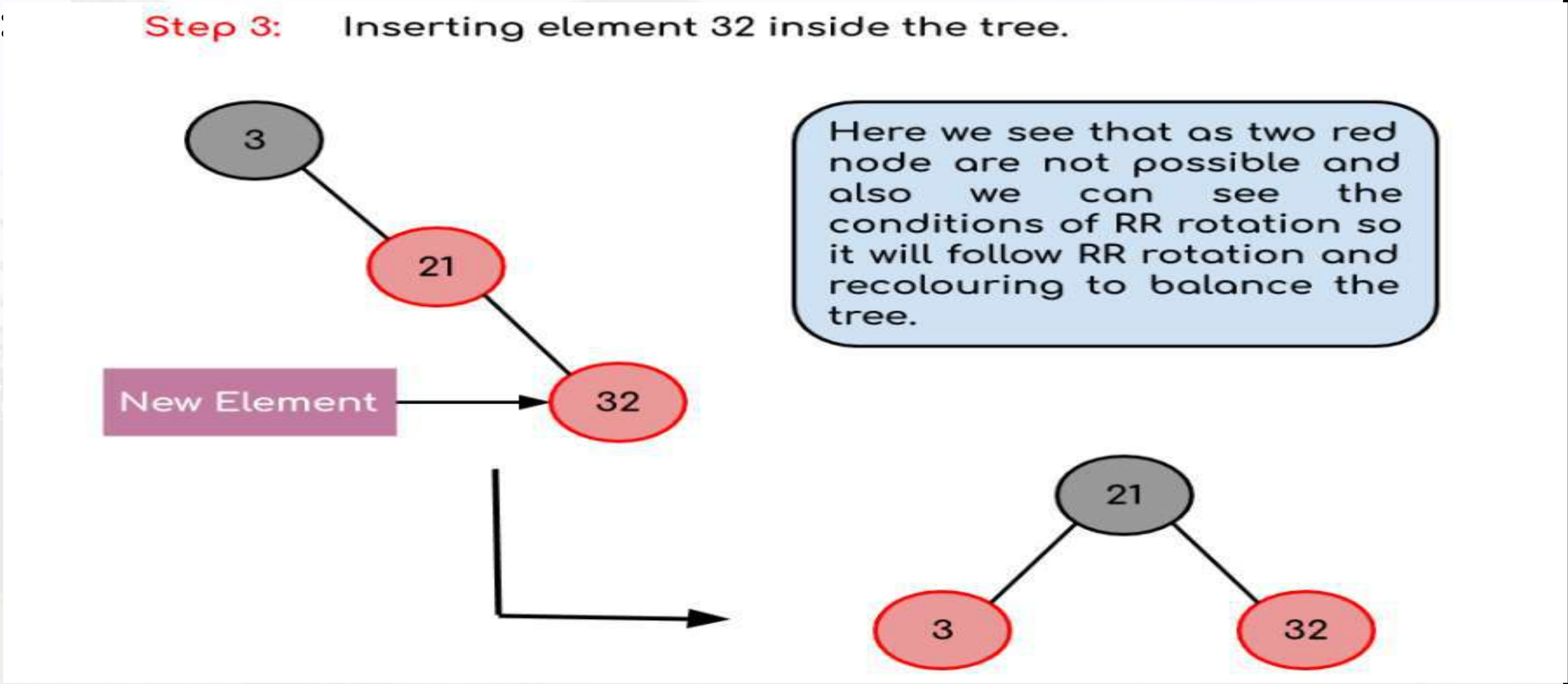
When the first element is inserted it is inserted as a root node and as root node has black colour so it acquires the colour black.



The new element is always inserted with a red colour and as 21 > 3 so it becomes the part of the right subtree of the root node.

EXAMPLE OF RBT

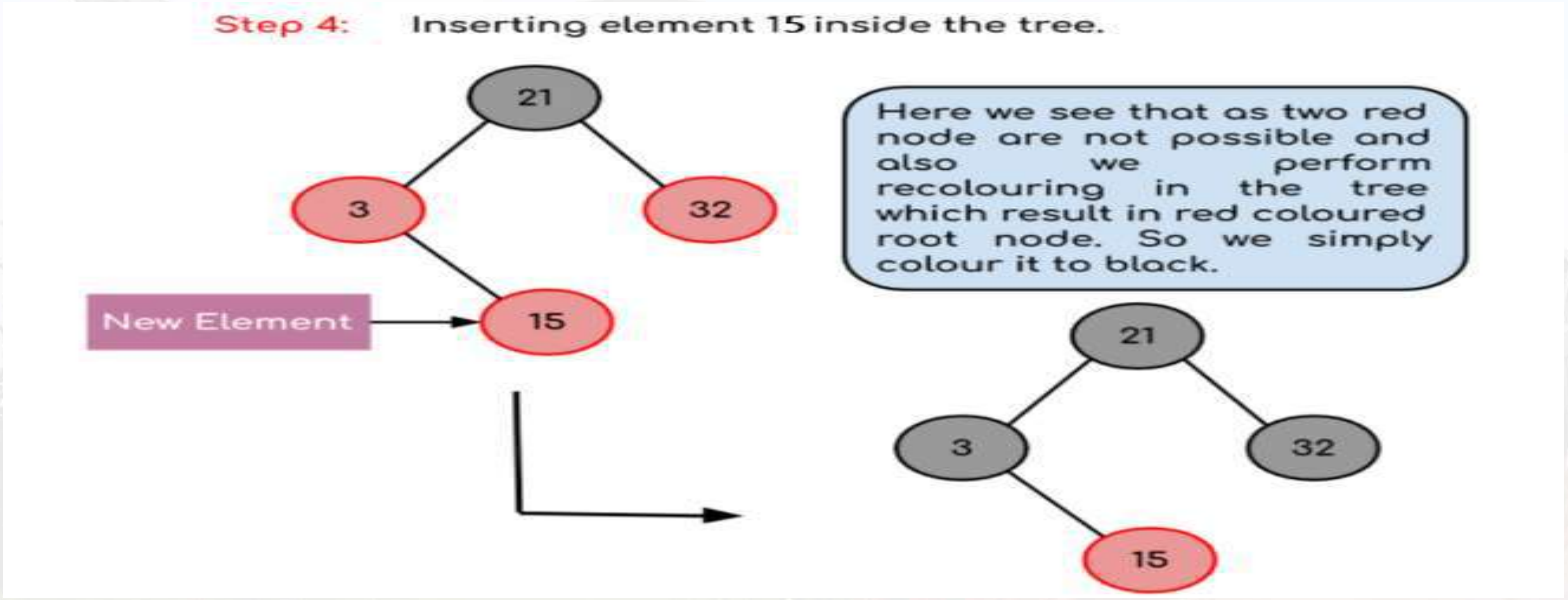
Example: Creating a red-black tree with elements 3, 21, 32 and 15 in an empty tree.


Step 3:
Inserting element 32 inside the tree.
3
21
32
New Element
Here we see that as two red node are not possible and also we can see the conditions of RR rotation so it will follow RR rotation and recolouring to balance the tree.
21
3
32

Now, as we insert 32 we see there is a red father-child pair which violates the Red-Black tree rule so we have to rotate it. Moreover, we see the conditions of RR rotation (considering the null node of the root node as black) so after rotation as the root node can't be Red so we have to perform recolouring in the tree resulting in the tree shown above.

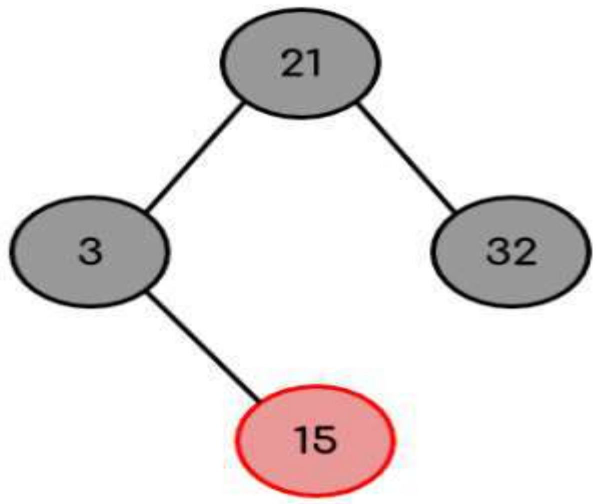
EXAMPLE OF RBT

Example: Creating a red-black tree with elements 3, 21, 32 and 15 in an empty tree.



Here we see that as two red node are not possible and also we perform recolouring in the tree which result in red coloured root node. So we simply colour it to black.

FINAL TREE :



EXAMPLE OF RBT

Insertion in Red Black tree :

The following are some rules used to create the Red-Black tree:

1. If the tree is empty, then we create a new node as a root node with the color black.
2. If the tree is not empty, then we create a new node as a leaf node with a color red.
3. If the parent of a new node is black, then exit.
4. If the parent of a new node is Red, then we have to check the color of the parent's sibling of a new node.

4a) If the color is Black, then we perform rotations and recoloring.

4b) If the color is Red then we recolor the node. We will also check whether the parents' parent of a new node is the root node or not; if it is not a root node, we will recolor and recheck the node

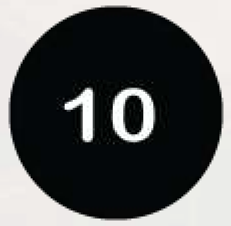
http://www.btechsmartclass.com/data_structures/red-black-trees.html

EXAMPLE OF RBT

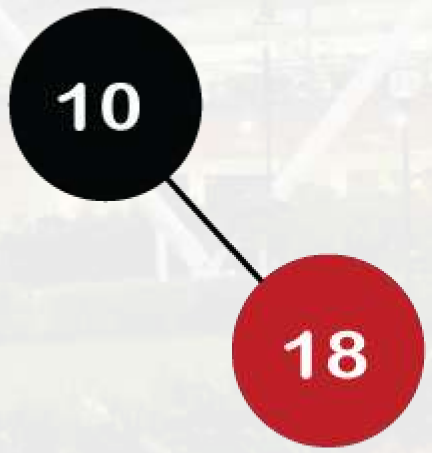
Let's understand the insertion in the Red-Black tree.

10, 18, 7, 15, 16, 30, 25, 40, 60 [**https://www.javatpoint.com/red-black-tree**](https://www.javatpoint.com/red-black-tree)

Step 1: Initially, the tree is empty, so we create a new node having value 10. This is the first node of the tree, so it would be the root node of the tree. As we already discussed, that root node must be black in color, which is shown below:



Step 2: The next node is 18. As 18 is greater than 10 so it will come at the right of 10 as shown below.



We know the second rule of the Red Black tree that if the tree is not empty then the newly created node will have the **Red** color. Therefore, node 18 has a Red color, as shown in the below figure:

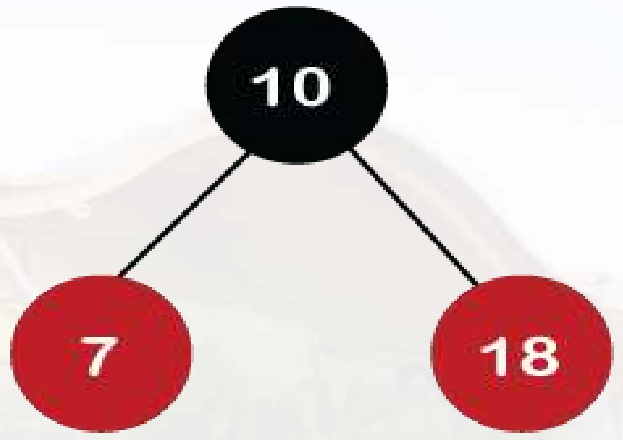
Now we verify the third rule of the Red-Black tree, i.e., the parent of the new node is black or not. In the above figure, the parent of the node is black in color; therefore, it is a Red-Black tree.

EXAMPLE OF RBT

Let's understand the insertion in the Red-Black tree.

10, 18, 7, 15, 16, 30, 25, 40, 60 https://www.javatpoint.com/red-black-tree

Step 3: Now, we create the new node having value 7 with Red color. As 7 is less than 10, so it will come at the left of 10 as shown below



Now we verify the third rule of the Red-Black tree, i.e., the parent of the new node is black or not. As we can observe, the parent of the node 7 is black in color, and it obeys the Red-Black tree's properties..

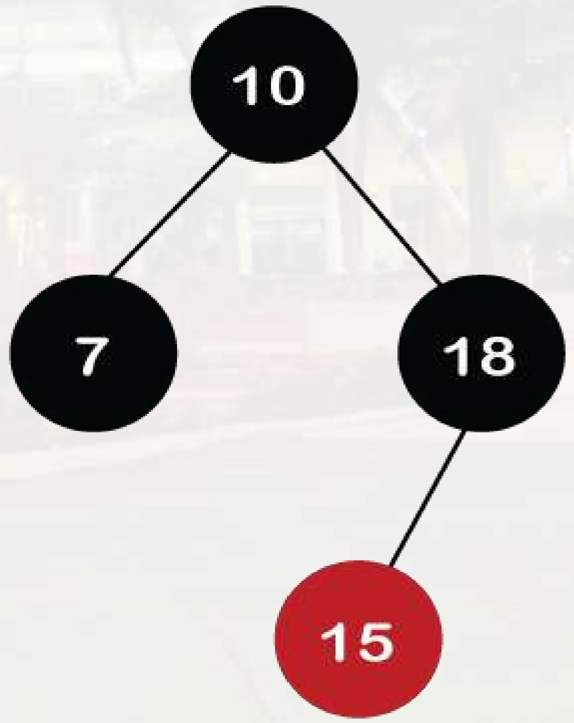
Step 4: The next element is 15, and 15 is greater than 10, but less than 18, so the new node will be created at the left of node 18. The node 15 would be Red in color as the tree is not empty.

EXAMPLE OF RBT

Let's understand the insertion in the Red-Black tree.

10, 18, 7, 15, 16, 30, 25, 40, 60 https://www.javatpoint.com/red-black-tree

The above tree violates the property of the Red-Black tree as it has Red-red parent-child relationship. Now we have to apply some rule to make a Red-Black tree. The rule 4 says that if the new node's parent is Red, then we have to check the color of the parent's sibling of a new node. The new node is node 15; the parent of the new node is node 18 and the sibling of the parent node is node 7. As the color of the parent's sibling is Red in color, so we apply the rule 4a. The rule 4a says that we have to recolor both the parent and parent's sibling node. So, both the nodes, i.e., 7 and 18, would be recolored as shown in the below figure.



We also have to check whether the parent's parent of the new node is the root node or not. As we can observe in the above figure, the parent's parent of a new node is the root node, so we do not need to recolor it.

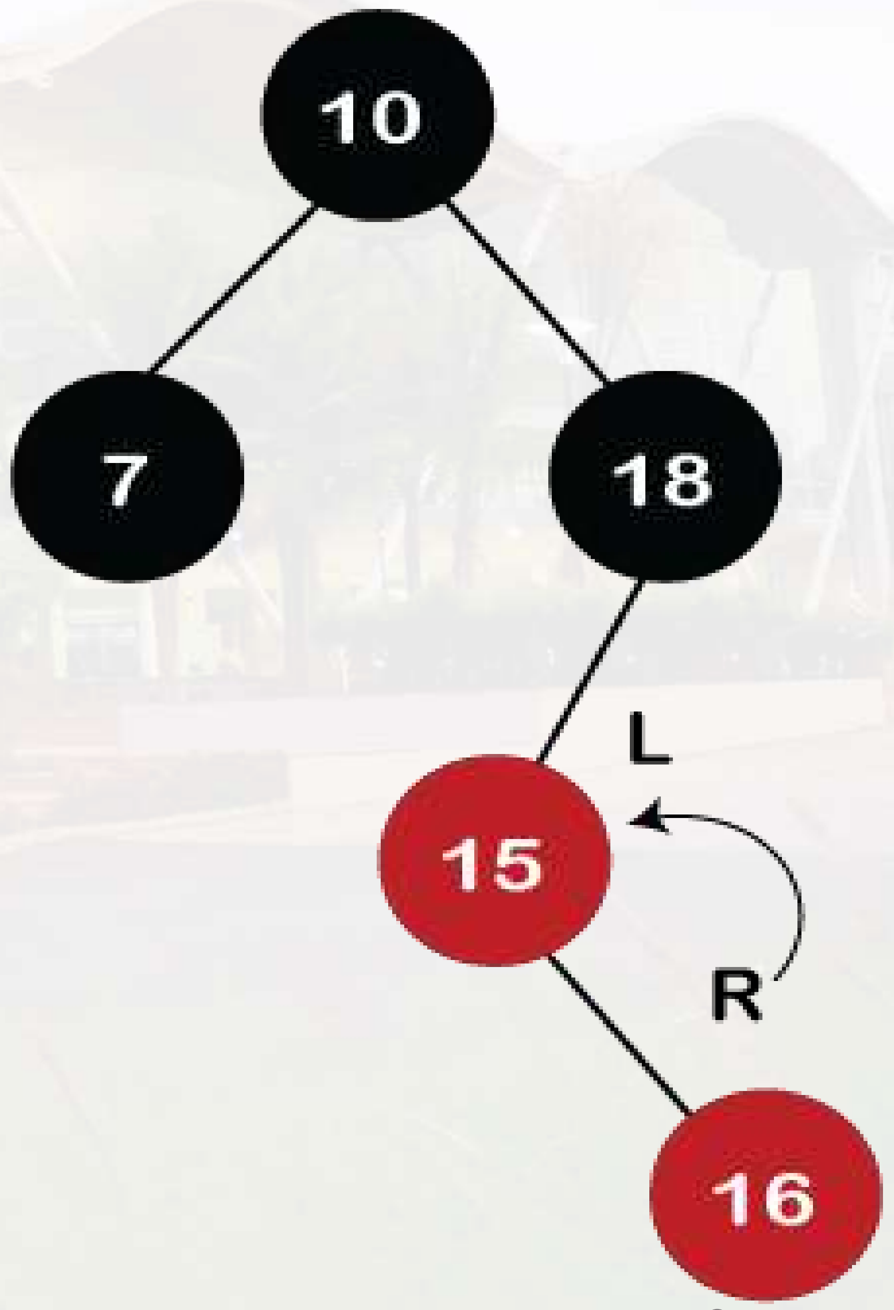
EXAMPLE OF RBT

Let's understand the insertion in the Red-Black tree.

10, 18, 7, 15, 16, 30, 25, 40, 60 **https://www.javatpoint.com/red-black-tree**

Step 5: The next element is 16. As 16 is greater than 10 but less than 18 and greater than 15, so node 16 will come at the right of node 15. The tree is not empty; node 16 would be Red in color, as shown in the below figure:

In this, it violates the property of the parent-child relationship as it has a red-red parent-child relationship. We have to apply some rules to make a Red-Black tree. Since the new node's parent is Red color, and the parent of the new node has no sibling, so rule 4a will be applied. The rule 4a says that some rotations and recoloring would be performed on the tree.



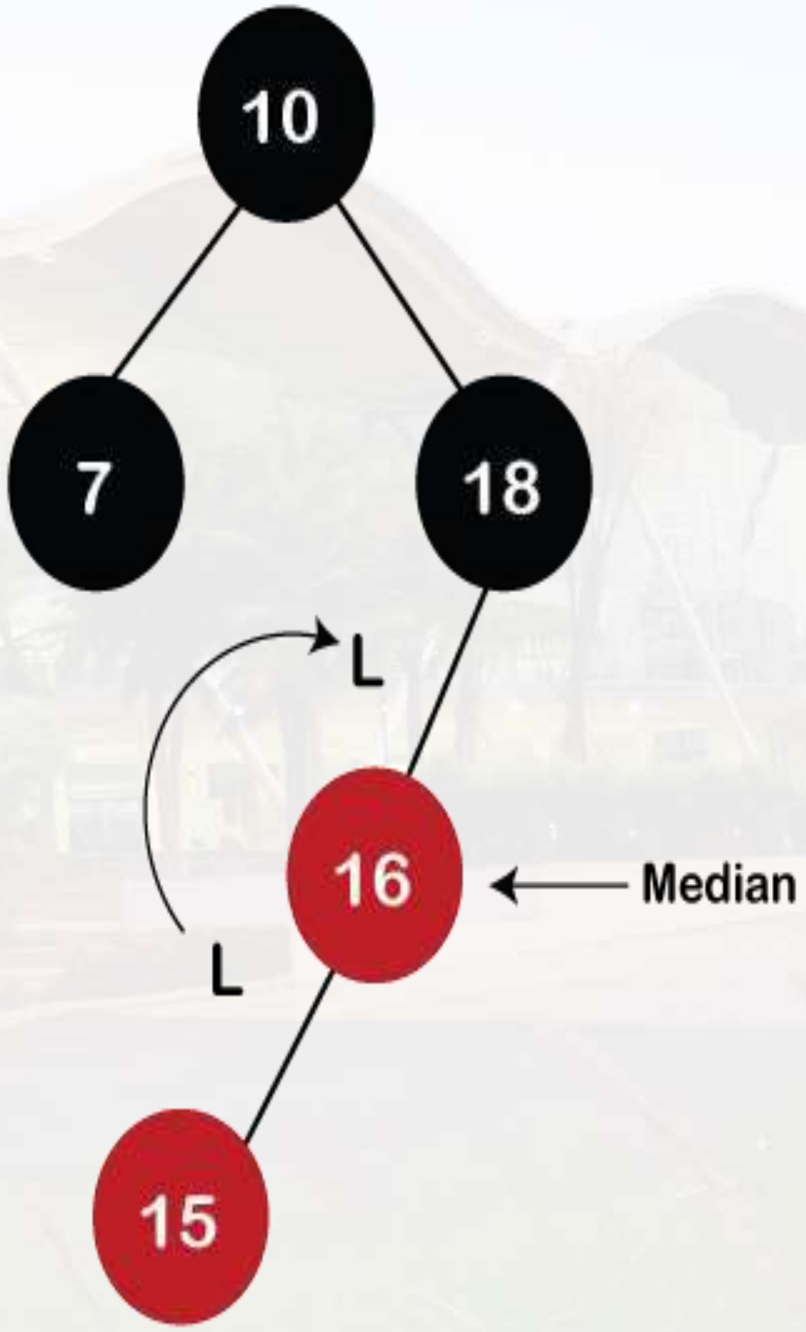
Since node 16 is right of node 15 and the parent of node 15 is node 18. Node 15 is the left of node 18. Here we have an LR relationship, so we require to perform two rotations. First, we will perform left, and then we will perform the right rotation. The left rotation would be performed on nodes 15 and 16, where node 16 will move upward, and node 15 will move downward. Once the left rotation is performed, the tree looks like as shown in the below figure:

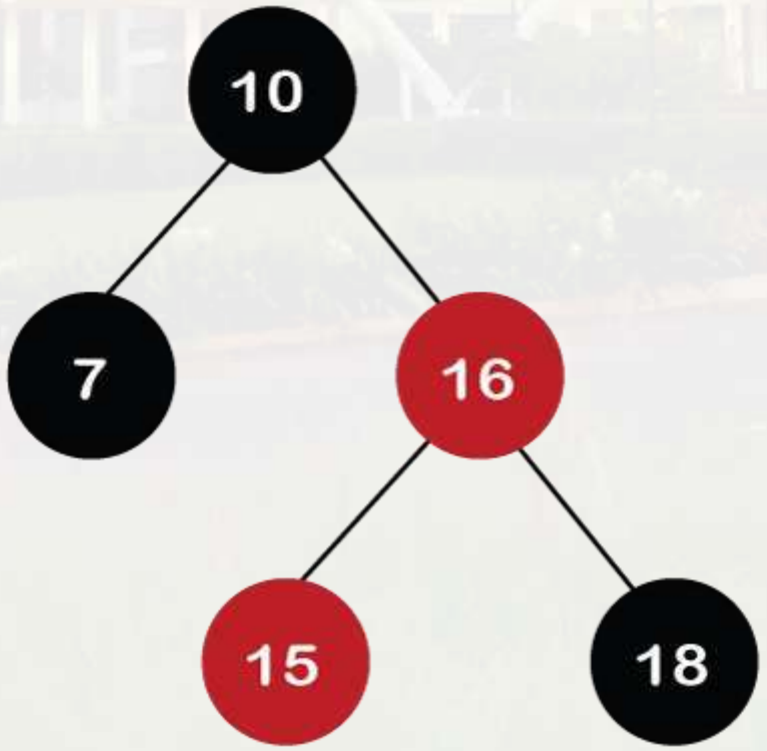
EXAMPLE OF RBT

Let's understand the insertion in the Red-Black tree.

10, 18, 7, 15, 16, 30, 25, 40, 60 https://www.javatpoint.com/red-black-tree

In the above figure, we can observe that there is an LL relationship. The above tree has a Red-red conflict, so we perform the right rotation. When we perform the right rotation, the median element would be the root node. Once the right rotation is performed, node 16 would become the root node, and nodes 15 and 18 would be the left child and right child, respectively, as shown in the below figure.



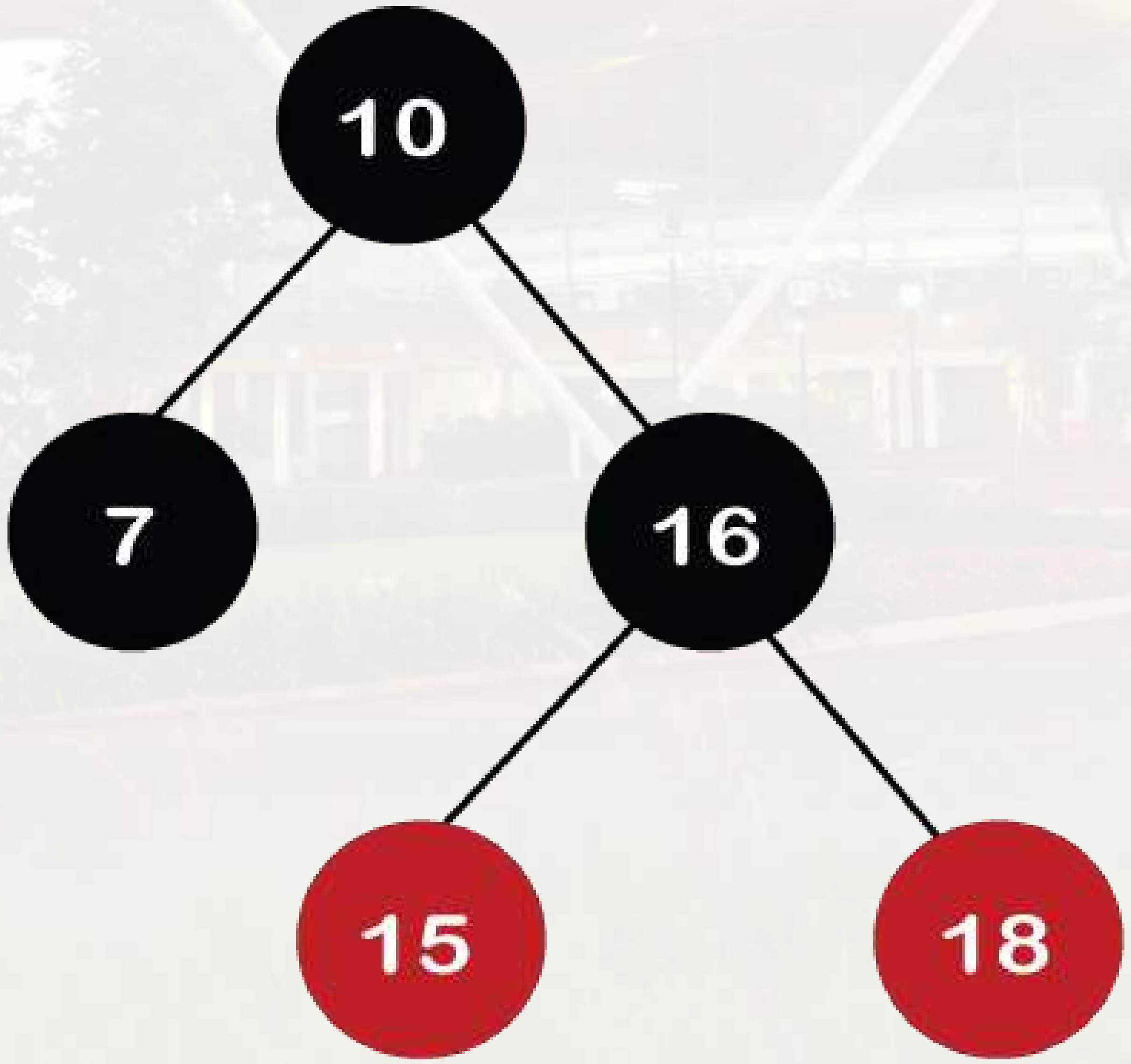


EXAMPLE OF RBT

Let's understand the insertion in the Red-Black tree.

10, 18, 7, 15, 16, 30, 25, 40, 60 https://www.javatpoint.com/red-black-tree

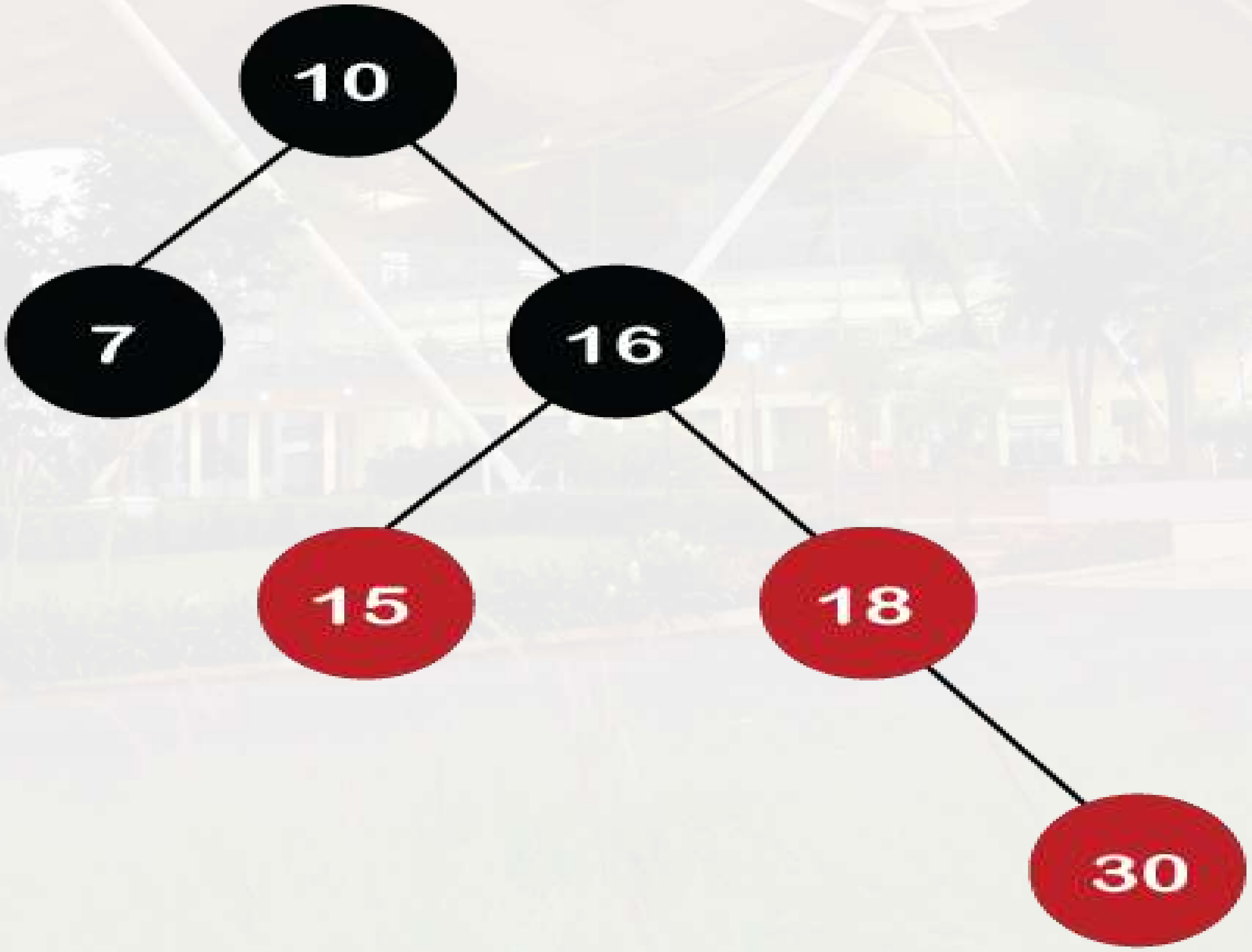
After rotation, node 16 and node 18 would be recolored; the color of node 16 is red, so it will change to black, and the color of node 18 is black, so it will change to a red color as shown in the below figure:



EXAMPLE OF RBT

Let's understand the insertion in the Red-Black tree.
10, 18, 7, 15, 16, 30, 25, 40, 60 https://www.javatpoint.com/red-black-tree

Step 6: The next element is 30. Node 30 is inserted at the right of node 18. As the tree is not empty, so the color of node 30 would be red.

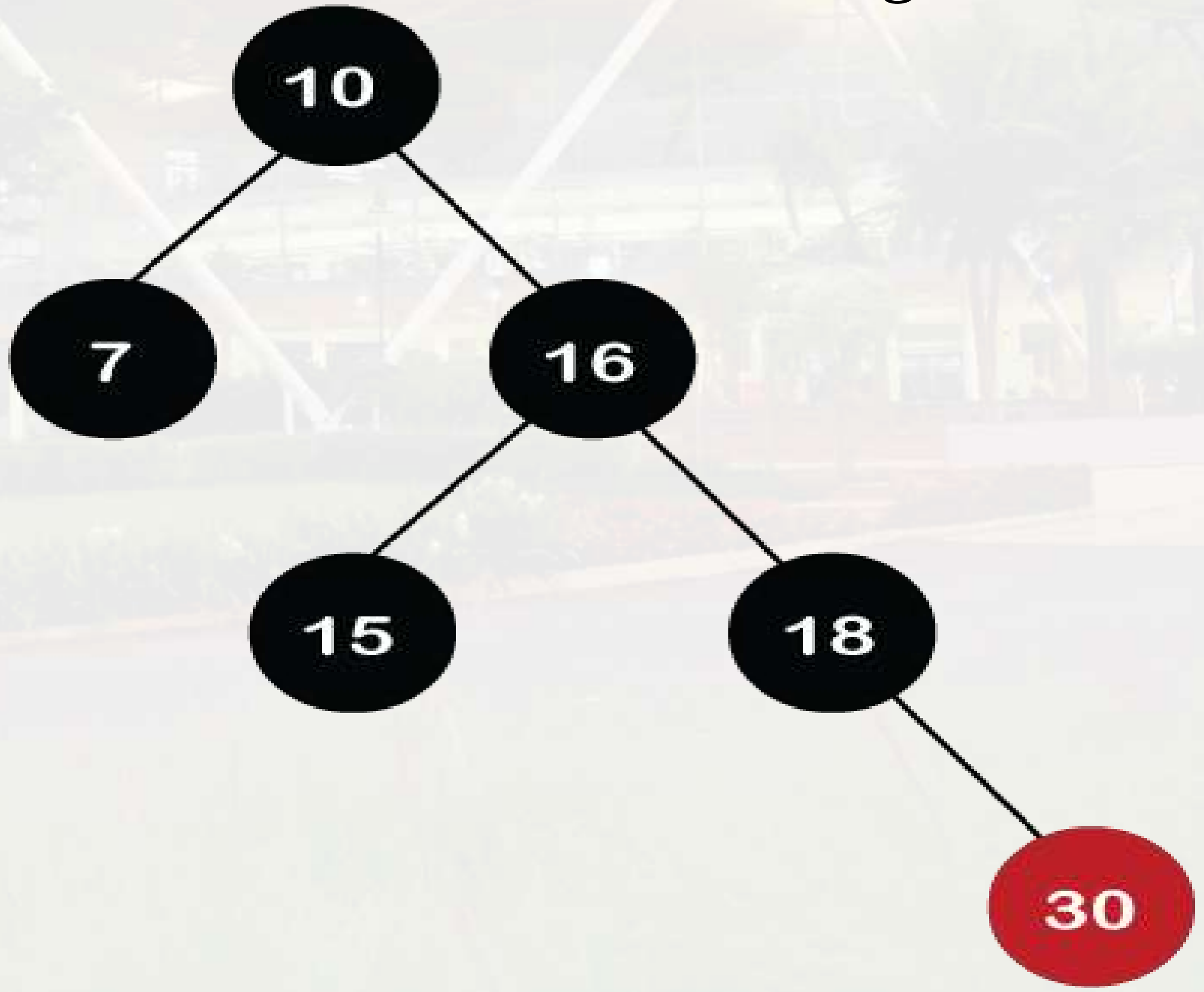


EXAMPLE OF RBT

Let's understand the insertion in the Red-Black tree.

10, 18, 7, 15, 16, 30, 25, 40, 60 **https://www.javatpoint.com/red-black-tree**

The color of the parent and parent's sibling of a new node is Red, so rule 4b is applied. In rule 4b, we have to do only recoloring, i.e., no rotations are required. The color of both the parent (node 18) and parent's sibling (node 15) would become black, as shown in the below image..

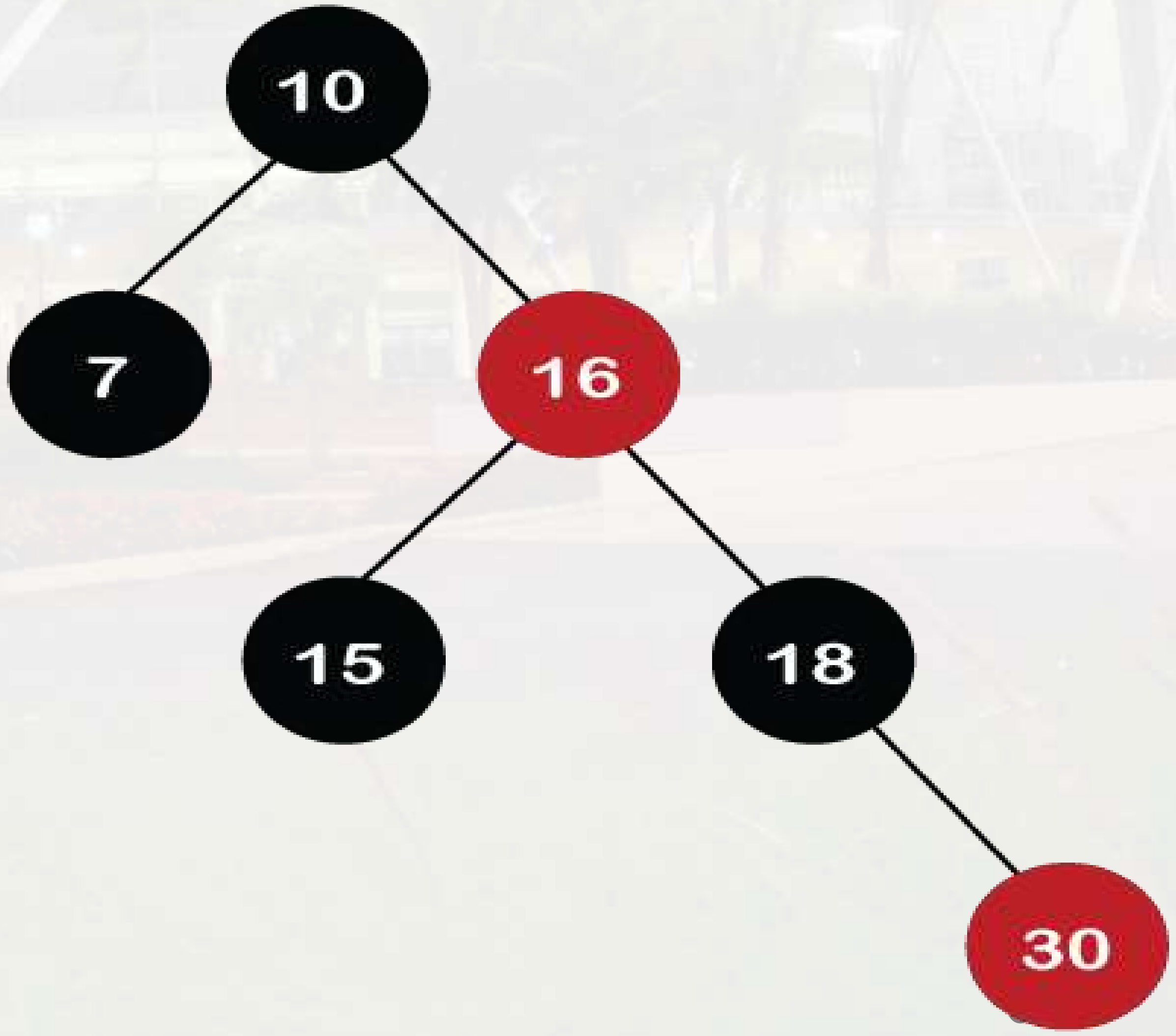


EXAMPLE OF RBT

Let's understand the insertion in the Red-Black tree.

10, 18, 7, 15, 16, 30, 25, 40, 60 [**https://www.javatpoint.com/red-black-tree**](https://www.javatpoint.com/red-black-tree)

We also have to check the parent's parent of the new node, whether it is a root node or not. The parent's parent of the new node, i.e., node 30 is node 16 and node 16 is not a root node, so we will recolor the node 16 and changes to the Red color. The parent of node 16 is node 10, and it is not in Red color, so there is no Red-red conflict.



EXAMPLE OF RBT

Let's understand the insertion in the Red-Black tree.
10, 18, 7, 15, 16, 30, 25, 40, 60 https://www.javatpoint.com/red-black-tree

Step 7: The next element is 25, which we have to insert in a tree. Since 25 is greater than 10, 16, 18 but less than 30; so, it will come at the left of node 30. As the tree is not empty, node 25 would be in Red color. Here Red-red conflict occurs as the parent of the newly created is Red color.

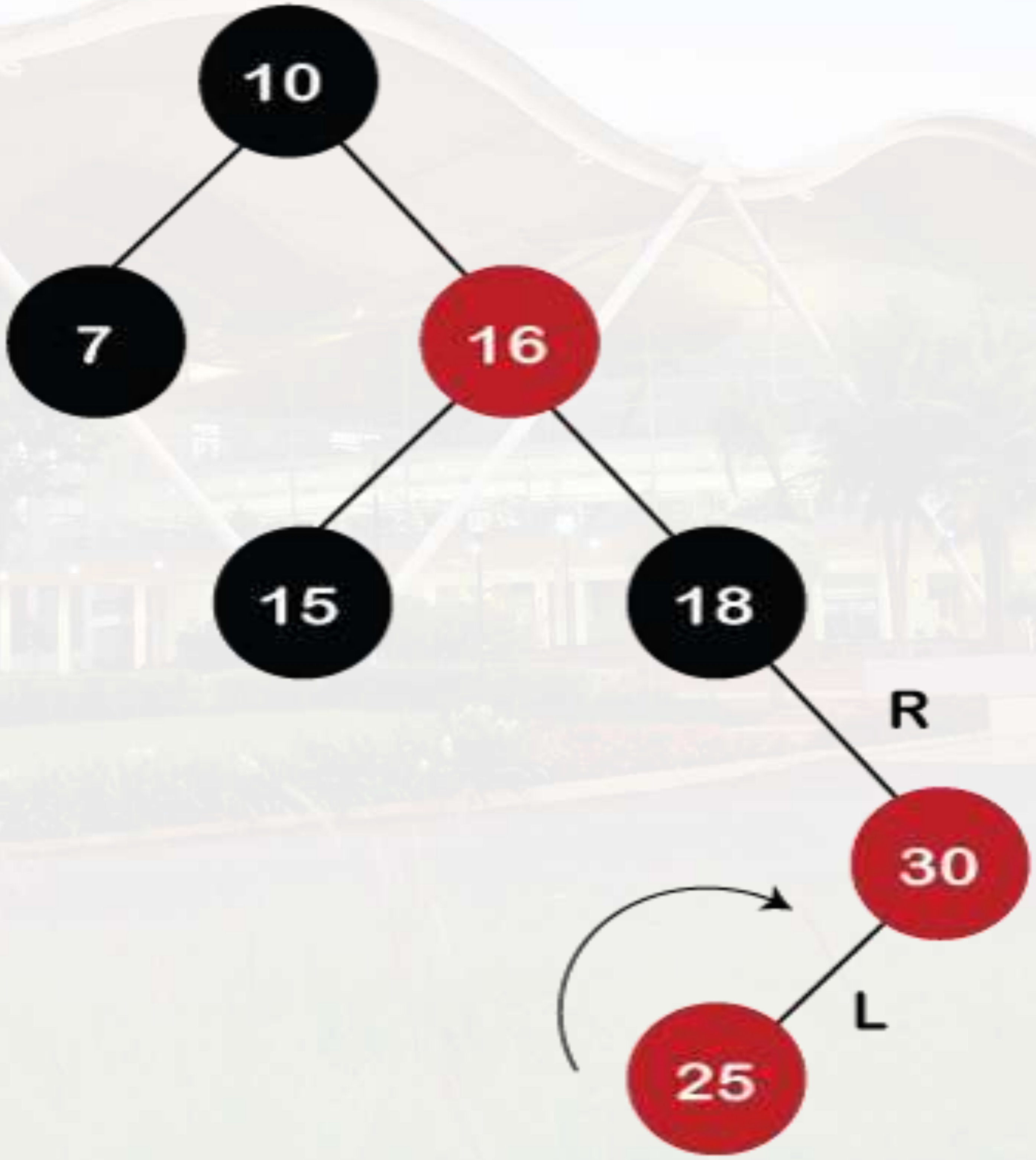
Since there is no parent's sibling, so rule 4a is applied in which rotation, as well as recoloring, are performed. First, we will perform rotations. As the newly created node is at the left of its parent and the parent node is at the right of its parent, so the RL relationship is formed. Firstly, the right rotation is performed in which node 25 goes upwards, whereas node 30 goes downwards, as shown in the below figure.

EXAMPLE OF RBT

Let's understand the insertion in the Red-Black tree.

10, 18, 7, 15, 16, 30, 25, 40, 60 https://www.javatpoint.com/red-black-tree

sads

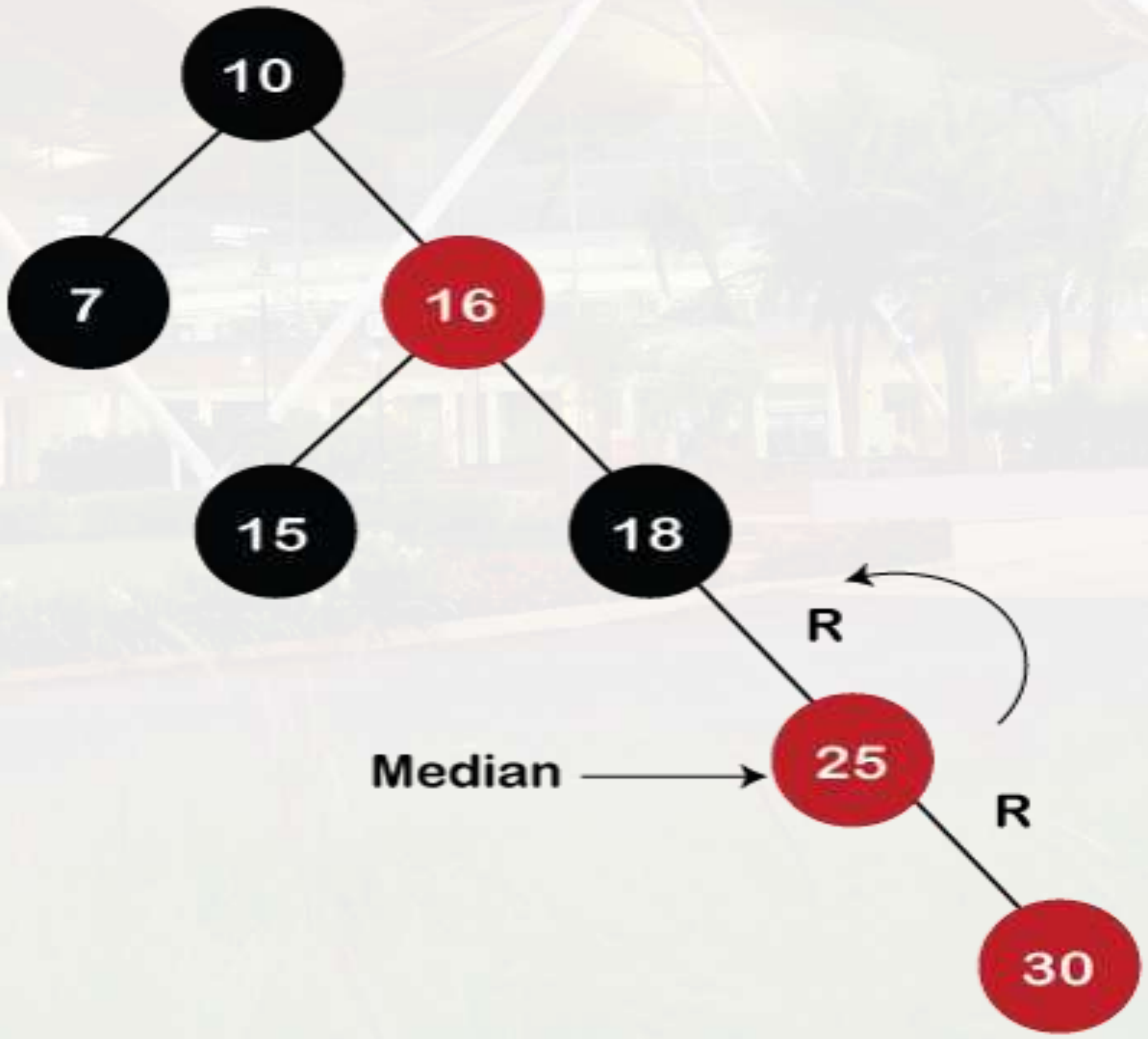


EXAMPLE OF RBT

Let's understand the insertion in the Red-Black tree.

10, 18, 7, 15, 16, 30, 25, 40, 60 https://www.javatpoint.com/red-black-tree

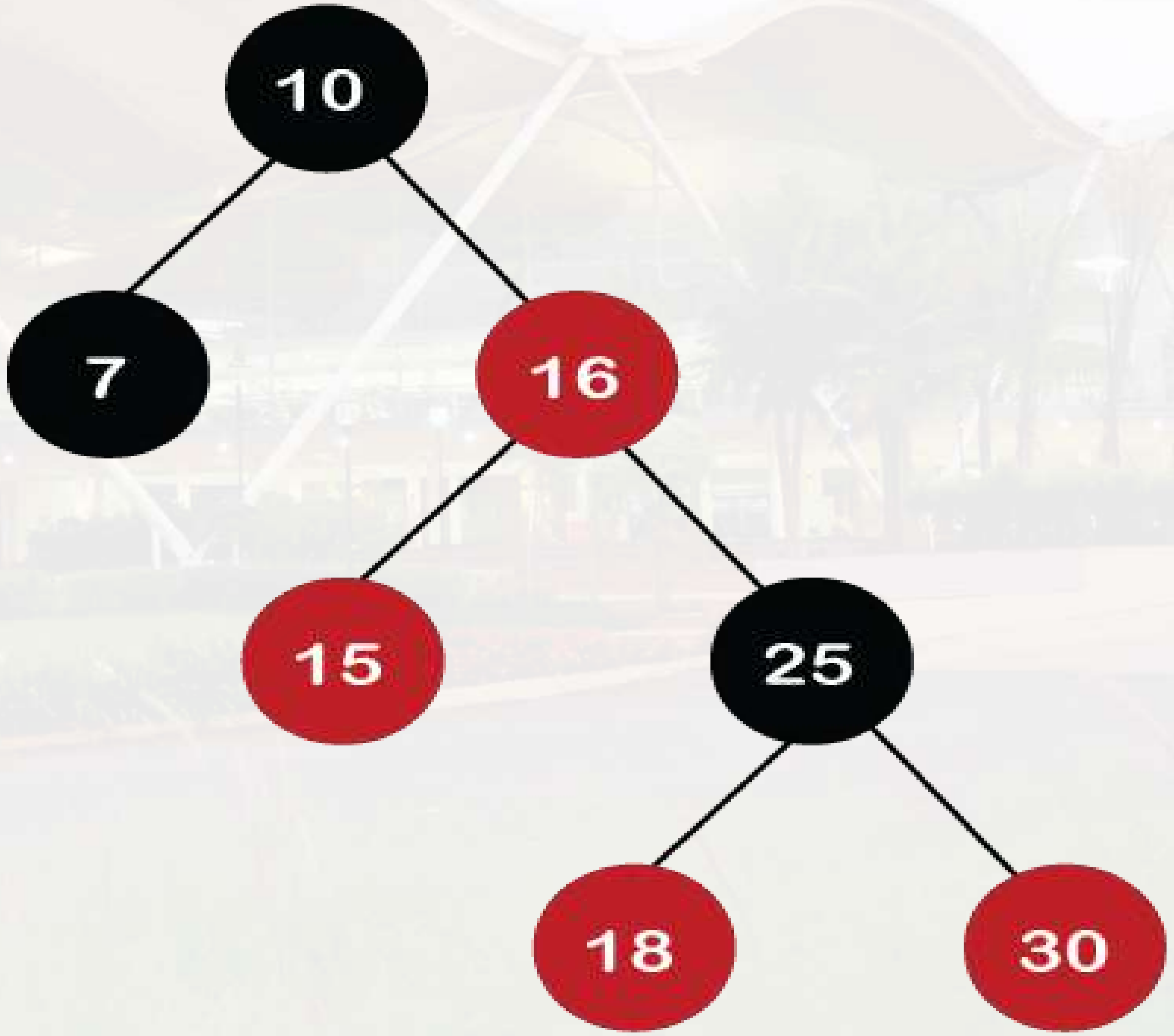
After the first rotation, there is an RR relationship, so left rotation is performed. After right rotation, the median element, i.e., 25 would be the root node; node 30 would be at the right of 25 and node 18 would be at the left of node 25.



EXAMPLE OF RBT

Let's understand the insertion in the Red-Black tree.
10, 18, 7, 15, 16, 30, 25, 40, 60 https://www.javatpoint.com/red-black-tree

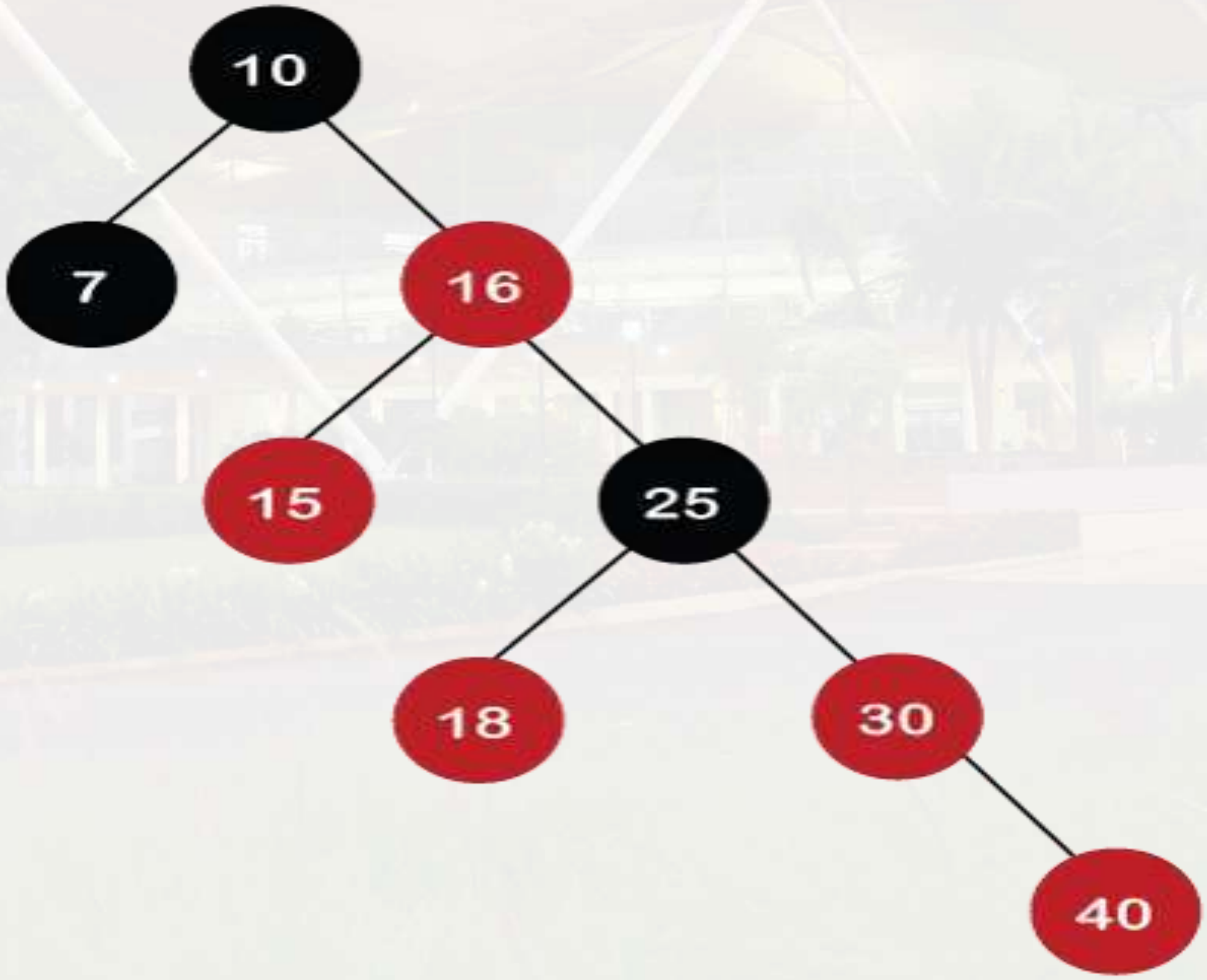
Now recoloring would be performed on nodes 25 and 18; node 25 becomes black in color, and node 18 becomes red in color.



EXAMPLE OF RBT

Let's understand the insertion in the Red-Black tree.
10, 18, 7, 15, 16, 30, 25, 40, 60 https://www.javatpoint.com/red-black-tree

Step 8: The next element is 40. Since 40 is greater than 10, 16, 18, 25, and 30, so node 40 will come at the right of node 30. As the tree is not empty, node 40 would be Red in color. There is a Red-red conflict between nodes 40 and 30, so rule 4b will be applied.



EXAMPLE OF RBT

Let's understand the insertion in the Red-Black tree.
10, 18, 7, 15, 16, 30, 25, 40, 60 https://www.javatpoint.com/red-black-tree

As the color of parent and parent's sibling node of a new node is Red so recoloring would be performed. The color of both the nodes would become black, as shown in the below image.

After recoloring, we also have to check the parent's parent of a new node, i.e., 25, which is not a root node, so recoloring would be performed, and the color of node 25 changes to Red.

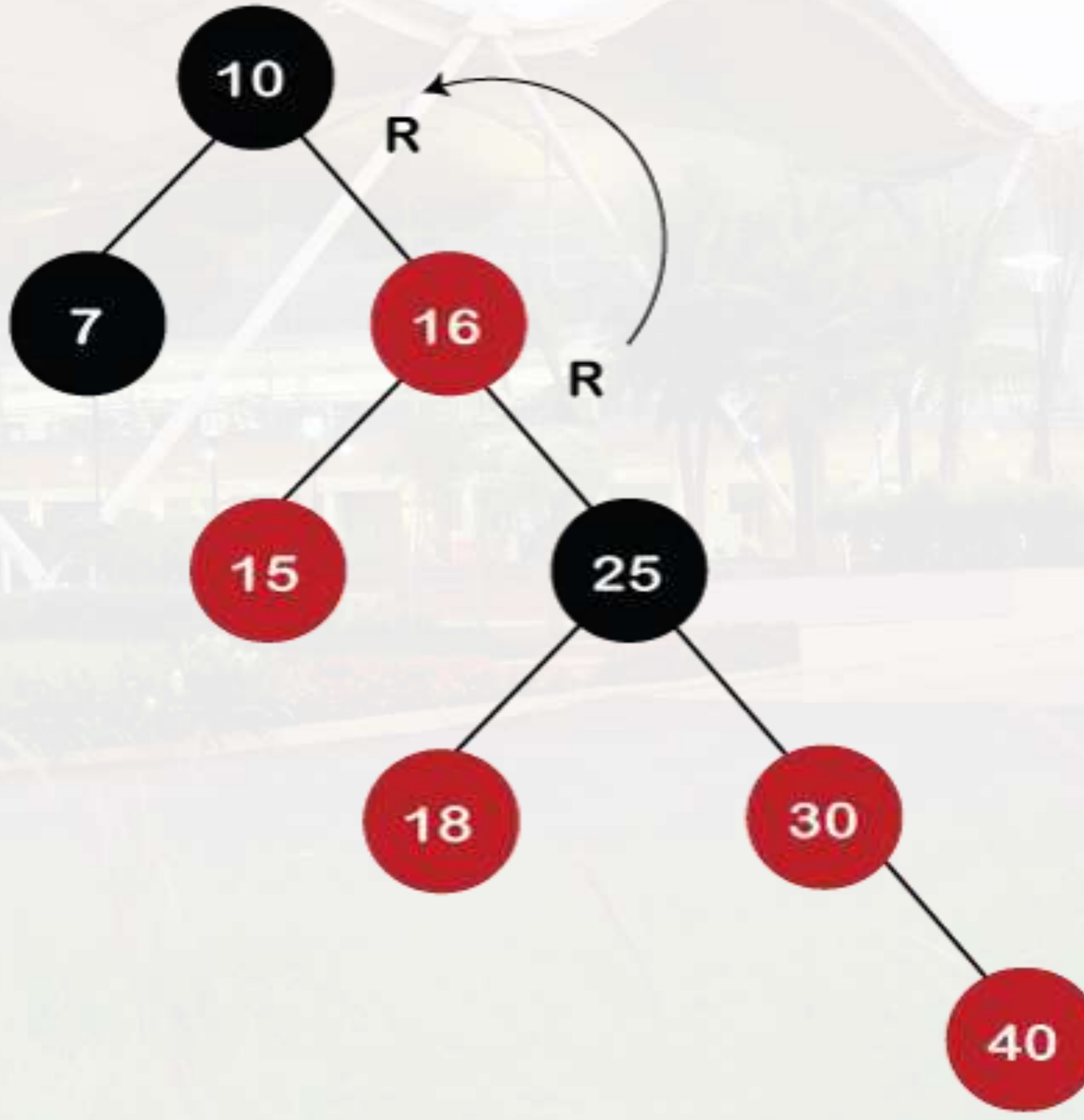
After recoloring, red-red conflict occurs between nodes 25 and 16. Now node 25 would be considered as the new node. Since the parent of node 25 is red in color, and the parent's sibling is black in color, rule 4a would be applied. Since 25 is at the right of the node 16 and 16 is at the right of its parent, so there is an RR relationship. In the RR relationship, left rotation is performed. After left rotation, the median element 16 would be the root node, as shown in the below figure

EXAMPLE OF RBT

Let's understand the insertion in the Red-Black tree.

10, 18, 7, 15, 16, 30, 25, 40, 60 [**https://www.javatpoint.com/red-black-tree**](https://www.javatpoint.com/red-black-tree)

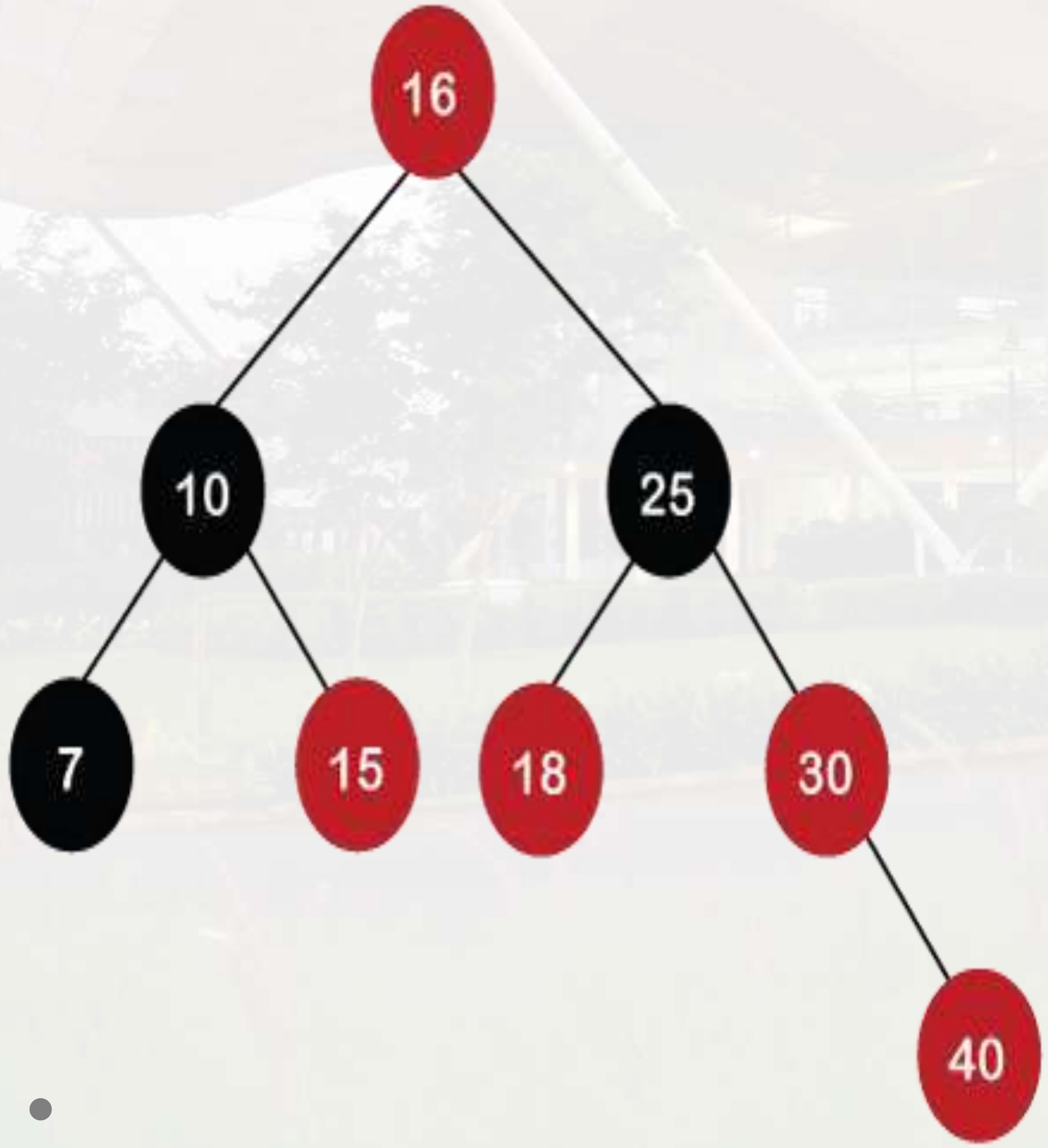
As the color of parent and parent's sibling node of a new node is Red so recoloring would be performed. The color of both the nodes would become black, as shown in the below image.

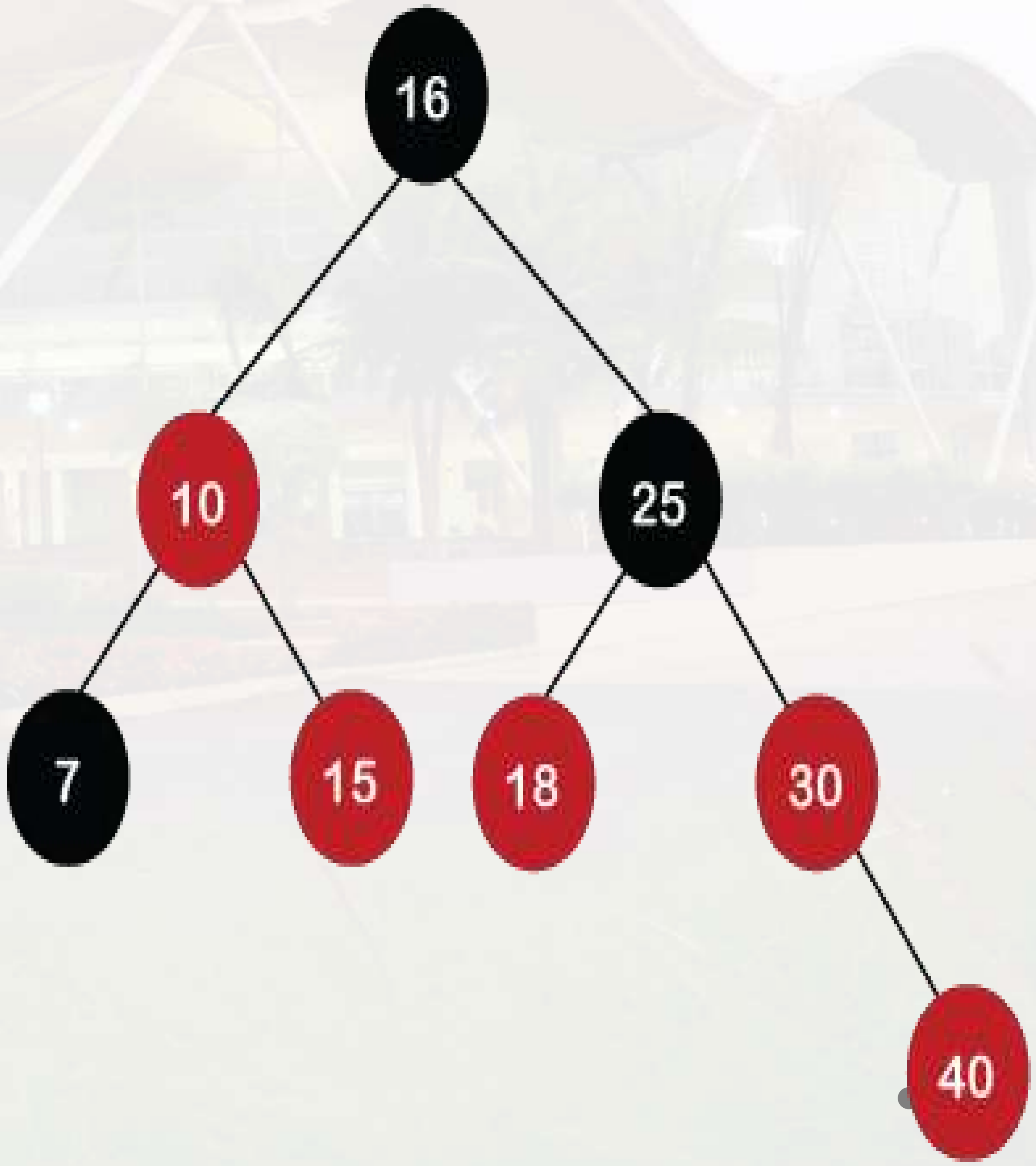


EXAMPLE OF RBT

Let's understand the insertion in the Red-Black tree.
10, 18, 7, 15, 16, 30, 25, 40, 60 **https://www.javatpoint.com/red-black-tree**

After rotation, recoloring is performed on nodes 16 and 10. The color of node 10 and node 16 changes to Red and Black, respectively as shown in the below figure



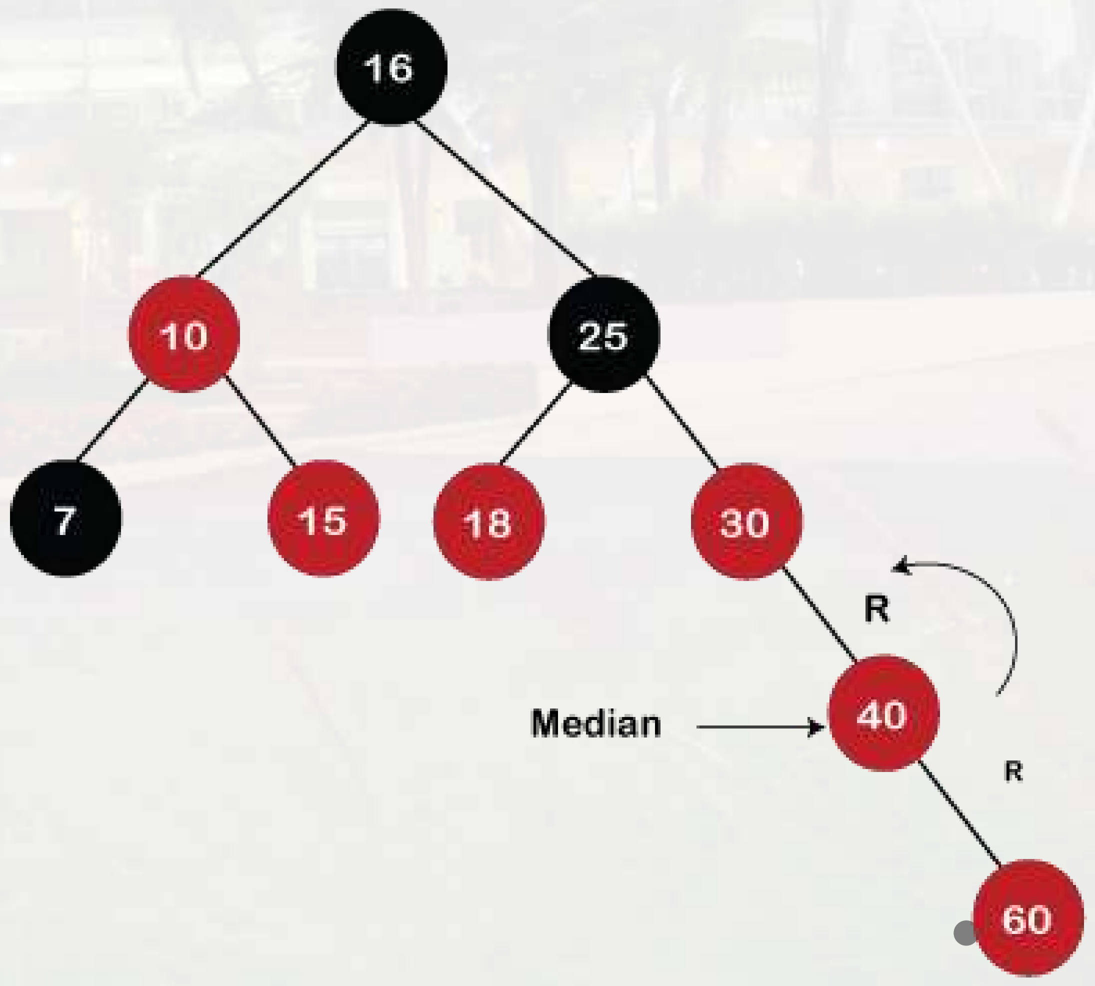


EXAMPLE OF RBT

Let's understand the insertion in the Red-Black tree.
10, 18, 7, 15, 16, 30, 25, 40, 60 https://www.javatpoint.com/red-black-tree

Step 9: The next element is 60. Since 60 is greater than 16, 25, 30, and 40, so node 60 will come at the right of node 40. As the tree is not empty, the color of node 60 would be Red.

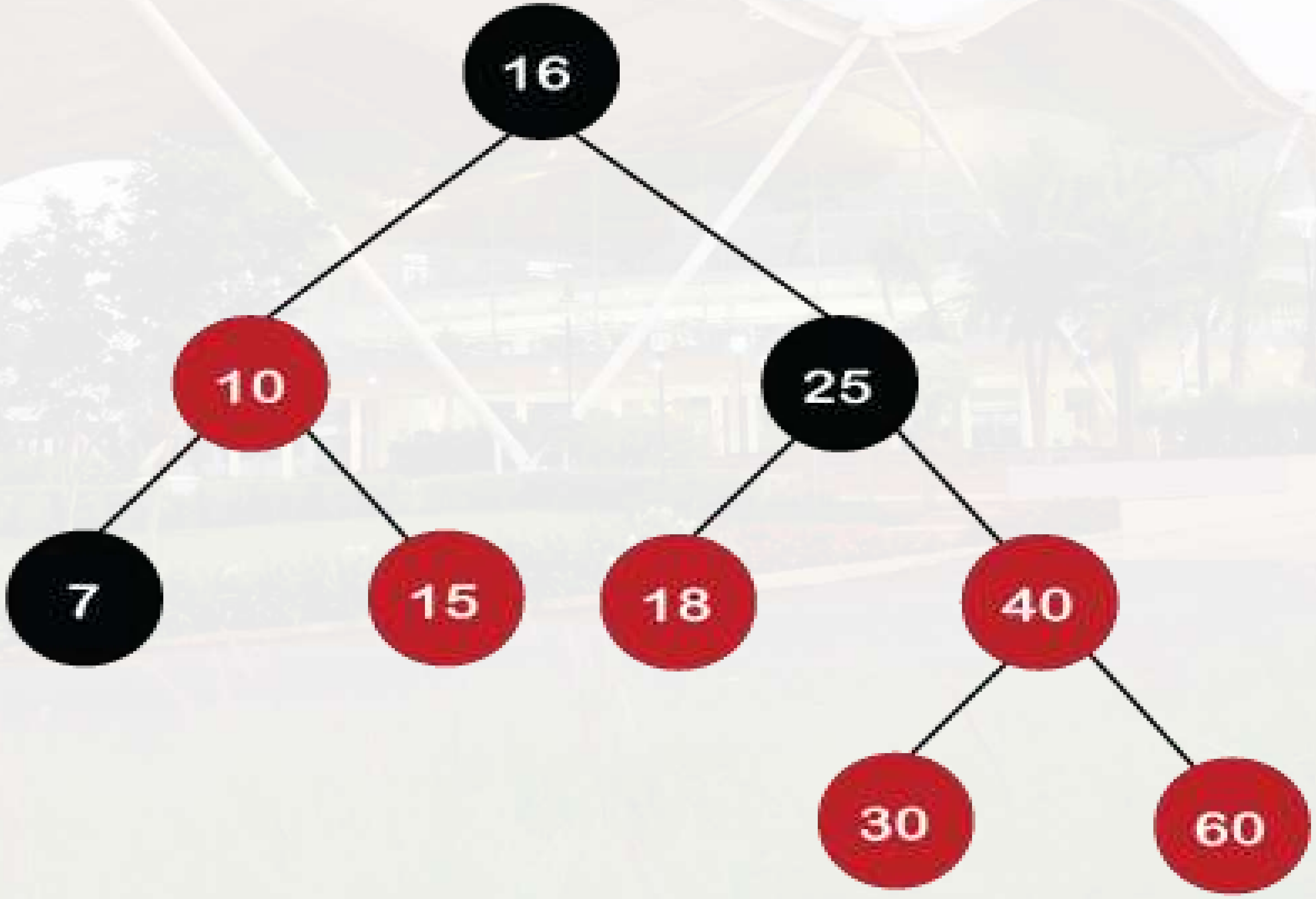
As we can observe in the above tree that there is a Red-red conflict occurs. The parent node is Red in color, and there is no parent's sibling exists in the tree, so rule 4a would be applied. The first rotation would be performed. The RR relationship exists between the nodes, so left rotation would be performed.



EXAMPLE OF RBT

Let's understand the insertion in the Red-Black tree.
10, 18, 7, 15, 16, 30, 25, 40, 60 https://www.javatpoint.com/red-black-tree

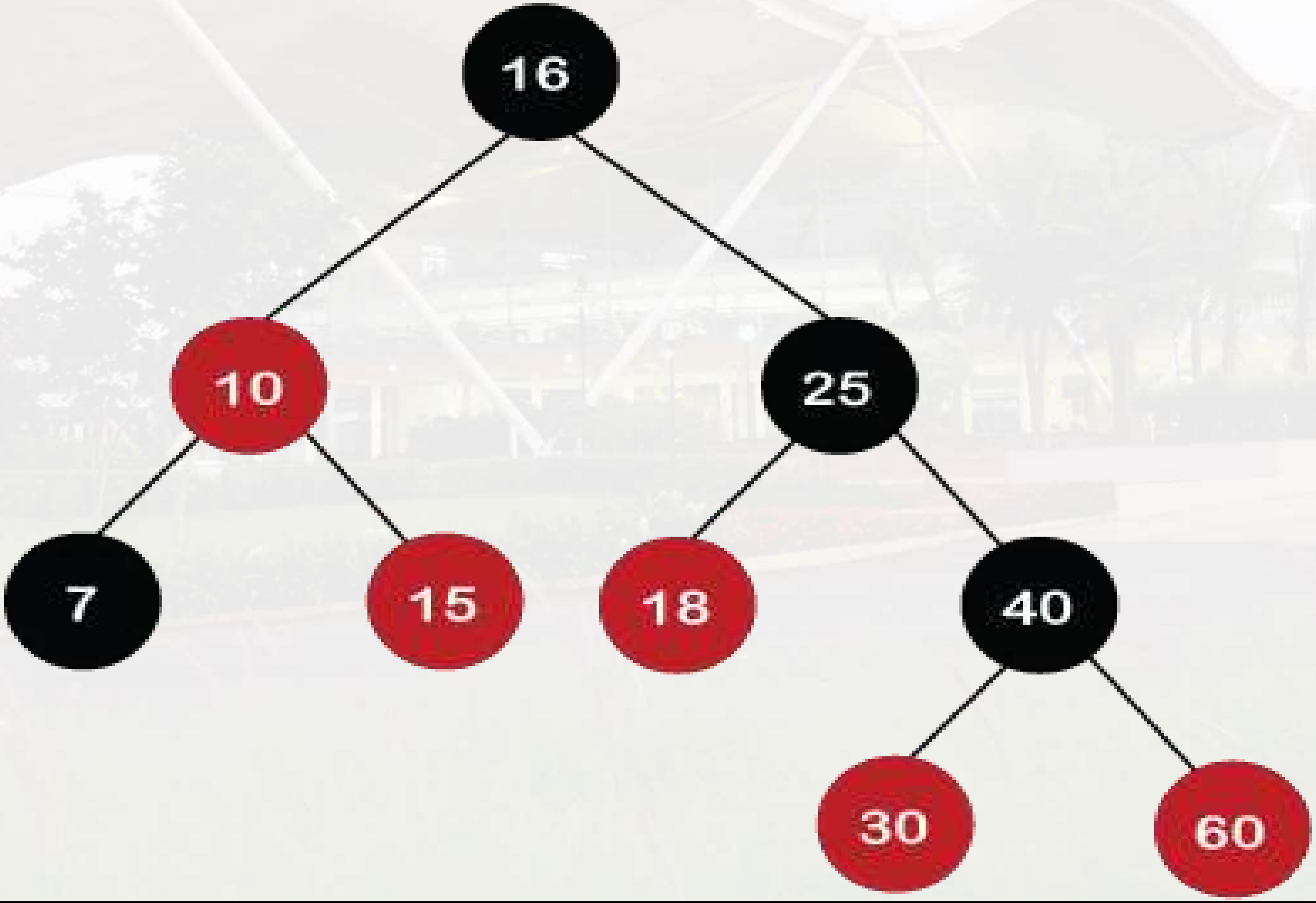
When left rotation is performed, node 40 will come upwards, and node 30 will come downwards, as shown in the below figure:



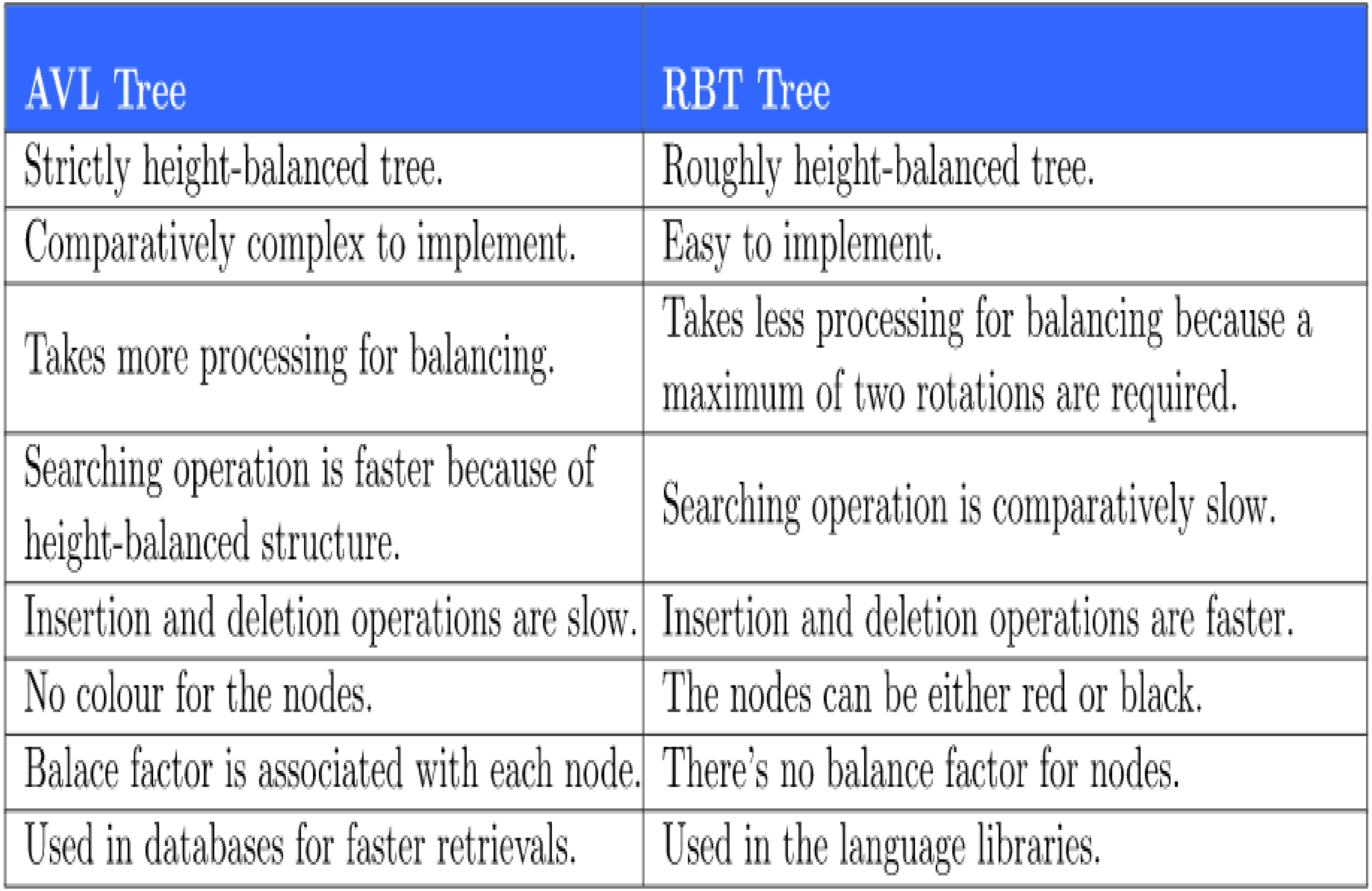
EXAMPLE OF RBT

Let's understand the insertion in the Red-Black tree.
10, 18, 7, 15, 16, 30, 25, 40, 60 https://www.javatpoint.com/red-black-tree

After rotation, the recoloring is performed on nodes 30 and 40. The color of node 30 would become Red, while the color of node 40 would become black.



AVL Vs Red Black Tree



AVL Tree	RBT Tree
Strictly height-balanced tree.	Roughly height-balanced tree.
Comparatively complex to implement.	Easy to implement.
Takes more processing for balancing.	Takes less processing for balancing because a maximum of two rotations are required.
Searching operation is faster because of height-balanced structure.	Searching operation is comparatively slow.
Insertion and deletion operations are slow.	Insertion and deletion operations are faster.
No colour for the nodes.	The nodes can be either red or black.
Balace factor is associated with each node.	There's no balance factor for nodes.
Used in databases for faster retrievals.	Used in the language libraries.

AA TREE

AA trees were introduced by Arne Andersson in 1993 and hence the name AA. They are a type of balanced binary search trees. It was developed as a simpler alternative to red black trees.

It eliminates many of the conditions that need to be considered to maintain a red-black tree.

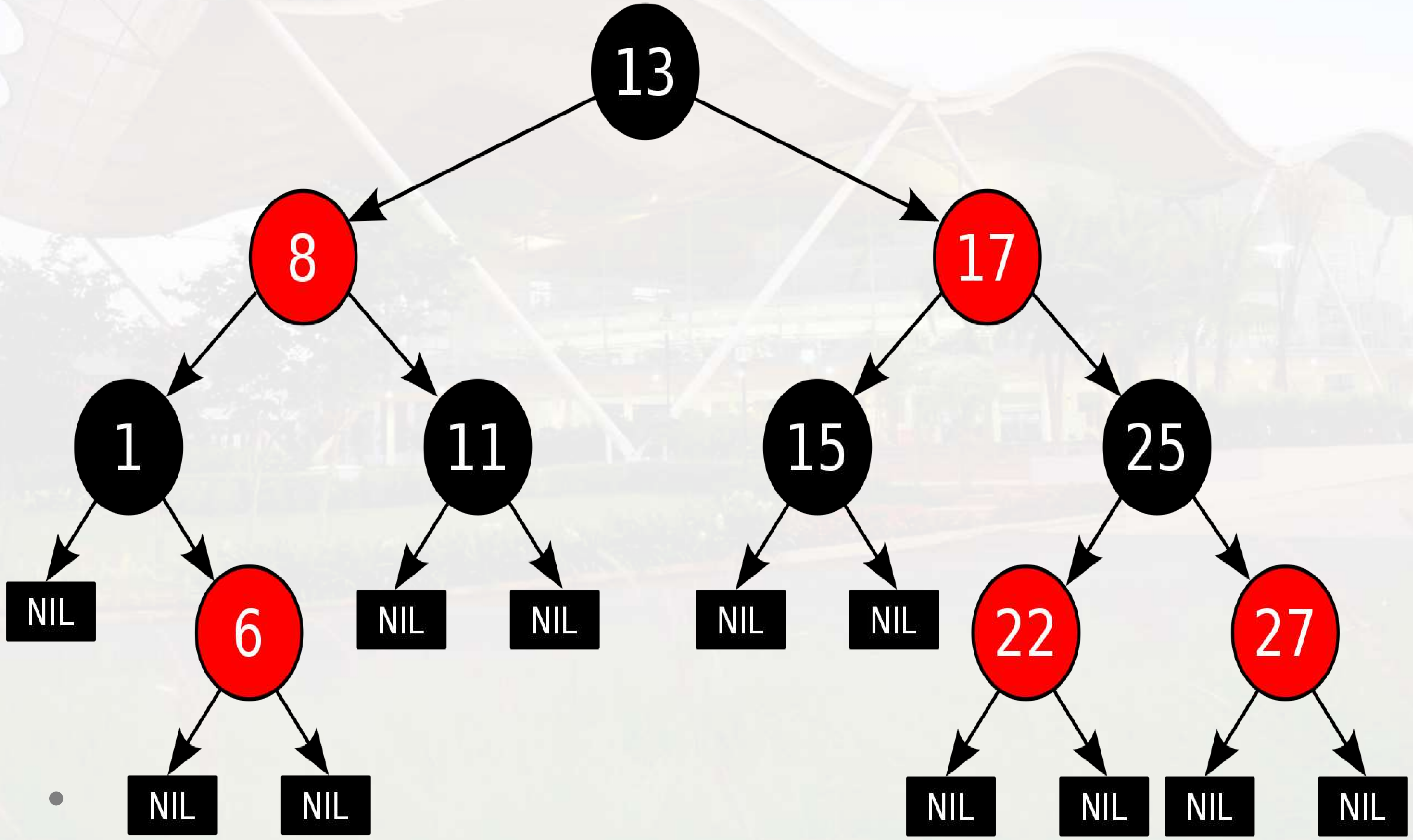
To understand AA trees, it is important to have a clear understanding of the concepts of a red black tree.

The tree below is an example of a red-black tree:

AA TREE

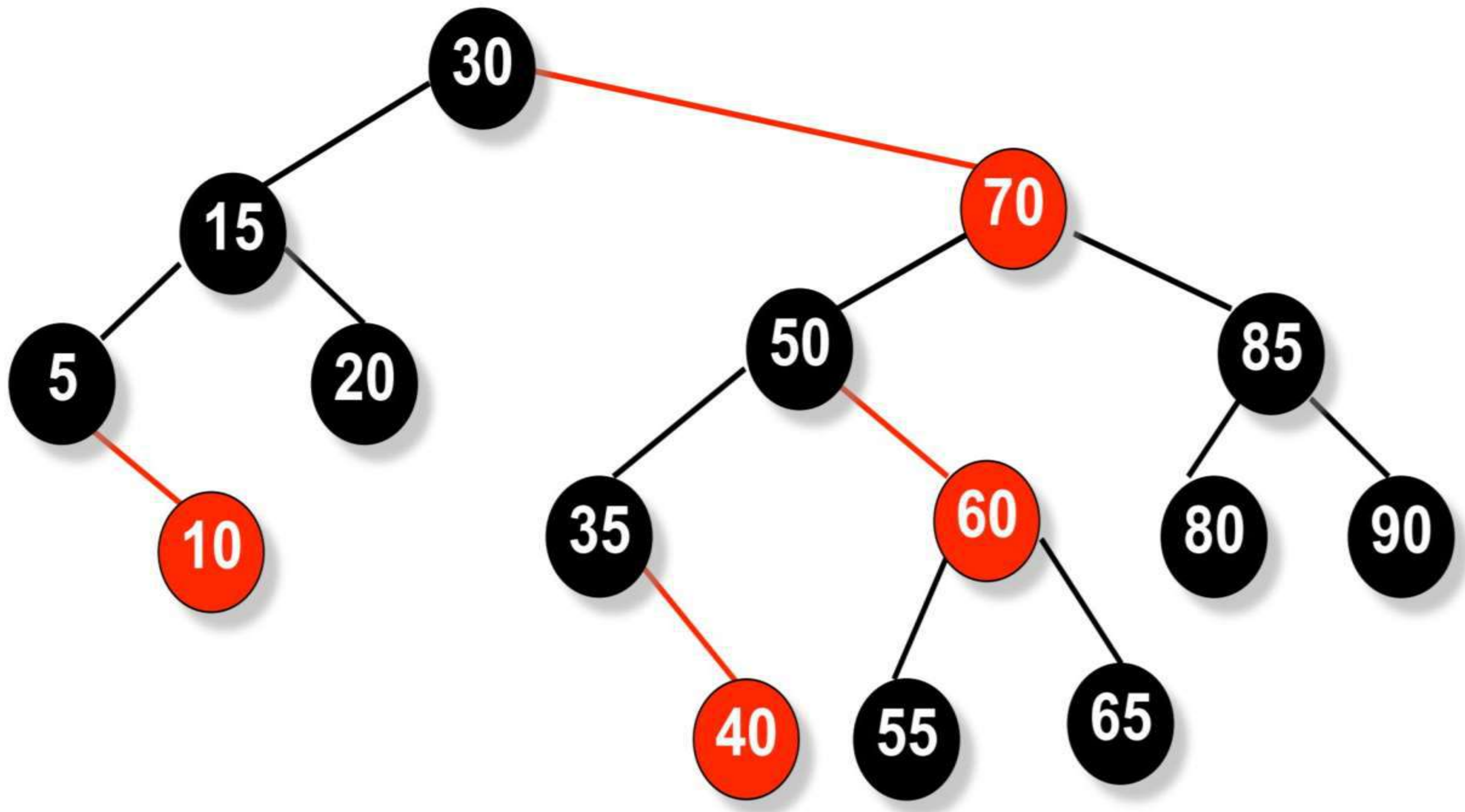
The tree below is an example of a red-black tree:

https://iq.opengenus.org/aa-trees/



AA TREE

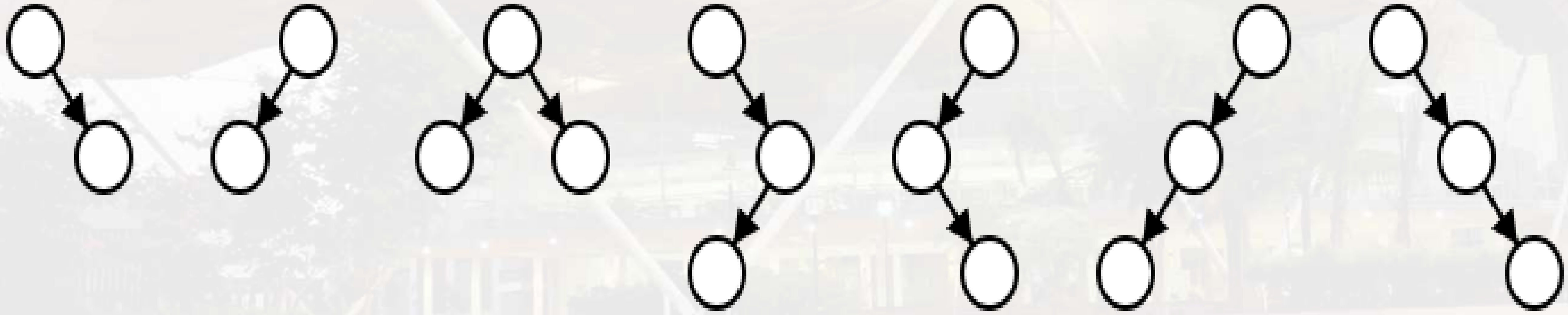
Unlike in red-black trees, red nodes on an AA tree can only be added as a right sub-child i.e. no red node can be a left sub-child. The tree below is an AA tree.



AA TREE

We can see from the above example that there are no left red children.

For the maintenance of a red-black tree, we need to consider seven different shapes to properly balance the tree:



A level of a node is the number of left links to a NULL reference.

AA trees make use of the concept of levels to aid in the balancing of trees. The level of a node is used for the balancing of the tree instead of using the color.

KD TREE

A K-Dimensional Tree (also known as K-D Tree) is a space-partitioning data structure for organizing points in a K-Dimensional space. This data structure acts similar to a binary search tree with each node representing data in the multi dimensional space.

KD TREE

Why use KD Tree?

The K-Dimensional Tree was first developed in 1975 by Jon Bentley. The purpose of the tree was to store spatial data with the goal of accomplishing:

Nearest neighbor search.

Range queries.

Fast look-up.

K-D Trees are capable of guaranteeing a Log2(n) depth, where n is the number of points in the set.

KD TREE

Why use KD Tree?

Since this data structure takes place in a multi-dimensional space, this data structure is incredibly useful right now. Some modern applications of a K-D Tree could range from astrophysical simulation to computer graphics to even data compression. Thanks to being similar in performance to a Binary Search Tree, this data structure also works exceedingly fast.

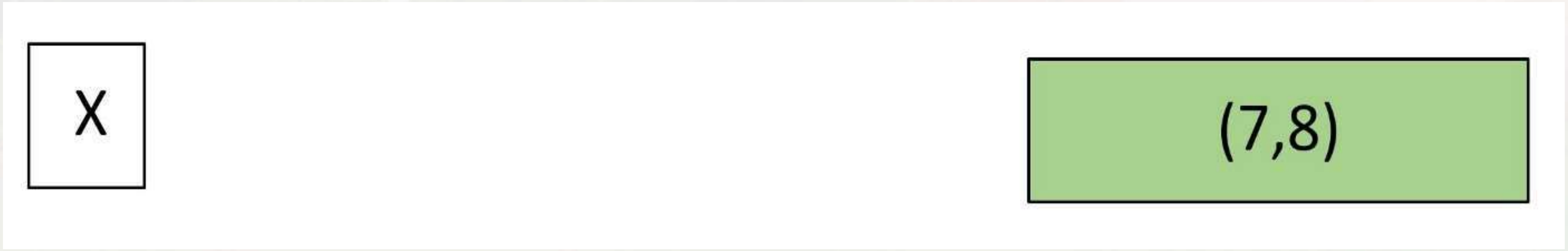
https://youtu.be/2Gul_-cbWM0

KD TREE

How it works for insertion?

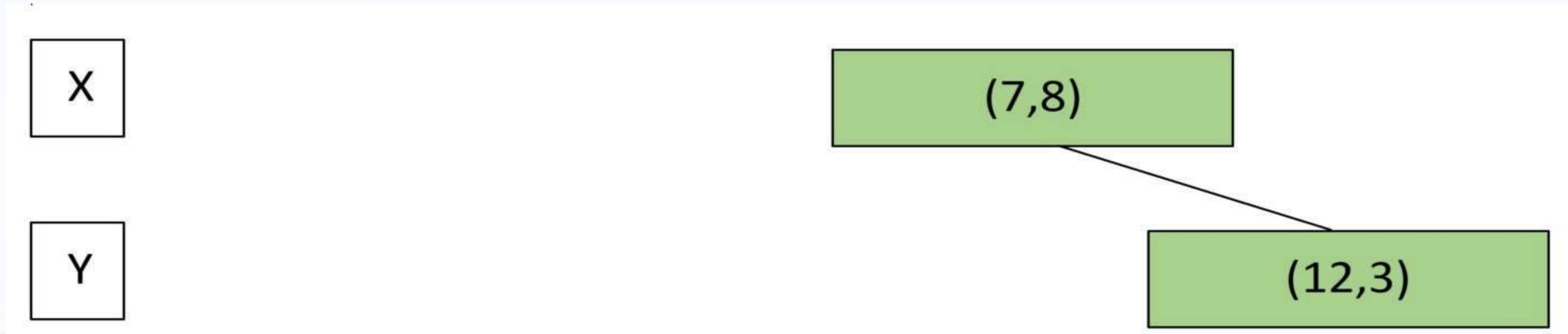
A simple example to showcase the insertion into a K-Dimensional Tree, we will use a k = 2.

The points we will be adding are: (7,8), (12,3), (14,1), (4,12), (9,1), (2,7), and (10,19).

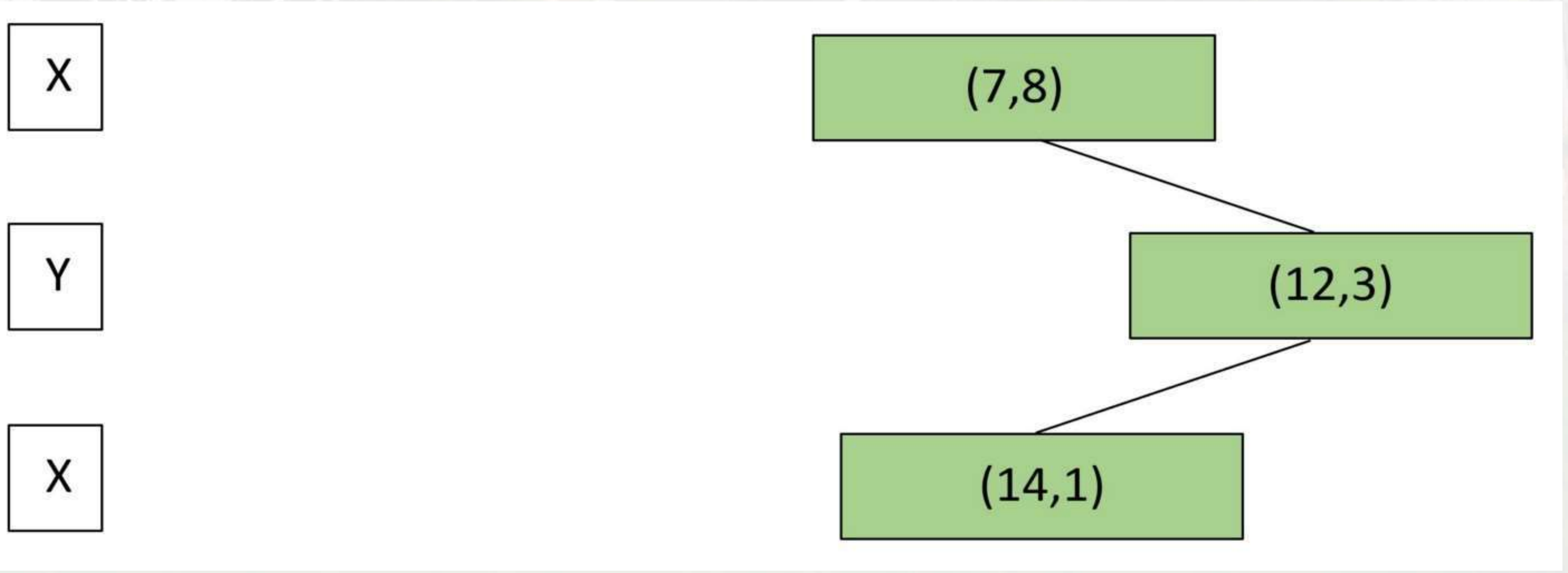


The first element inserted is (7,8). It will serve as the base node in the following K-D Tree example.

KD TREE

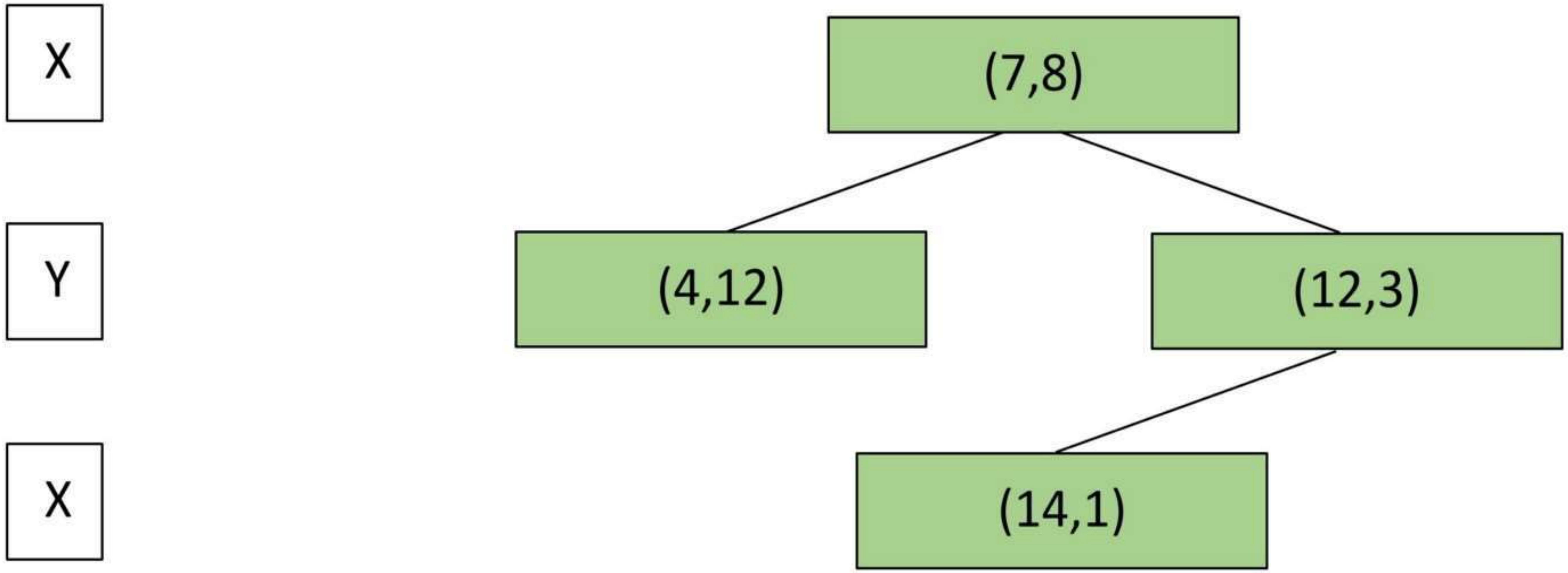


The second element inserted is (12,3). It is placed to the right leaf node of (7,8) because the X value, 12, is greater than the X value of base, 7.



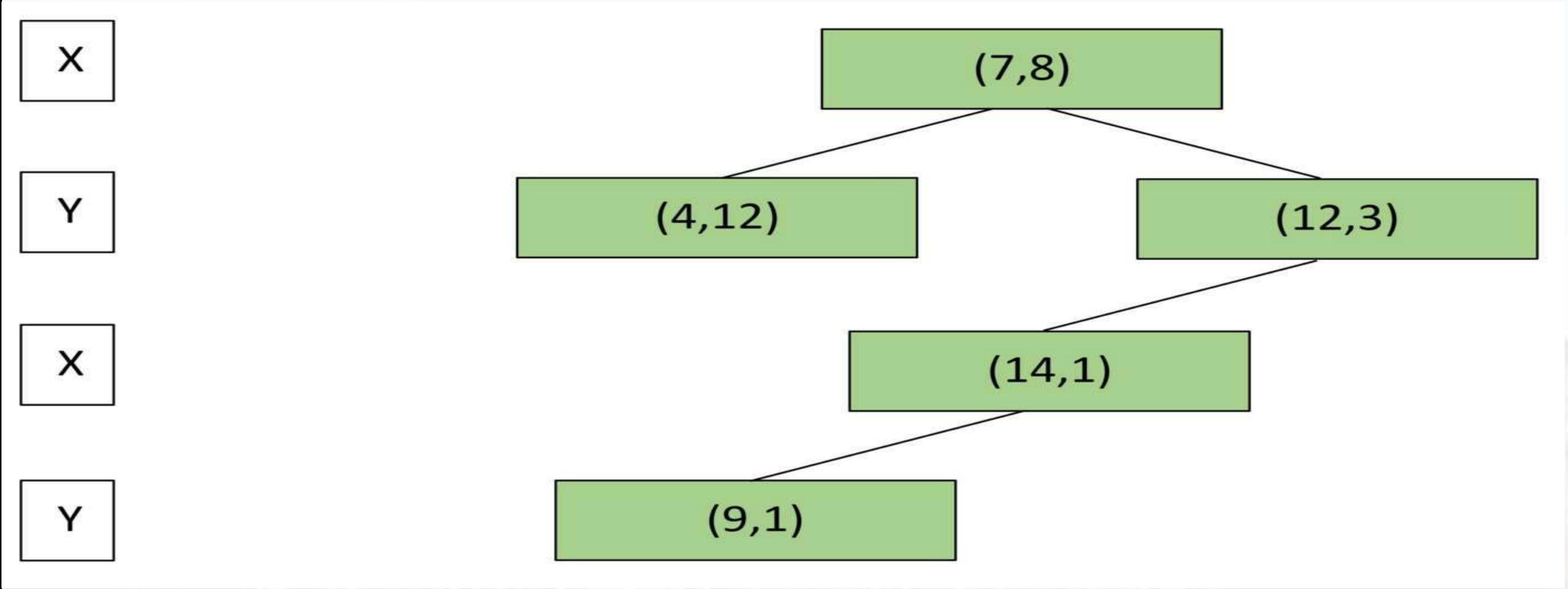
KD TREE

The next element inserted into the K-D Tree is (14,1). At the first level, we compare the X values of the set and since 14 is greater than 7 it moves to the right of the tree. Next is the comparison between (12,3) and (14,1). Each level the comparison operator changes, which means that we will be comparing the Y value of each set. Since the Y of the inserted set is 1 which is less than 3, it is inserted to the left leaf node of (12,3).



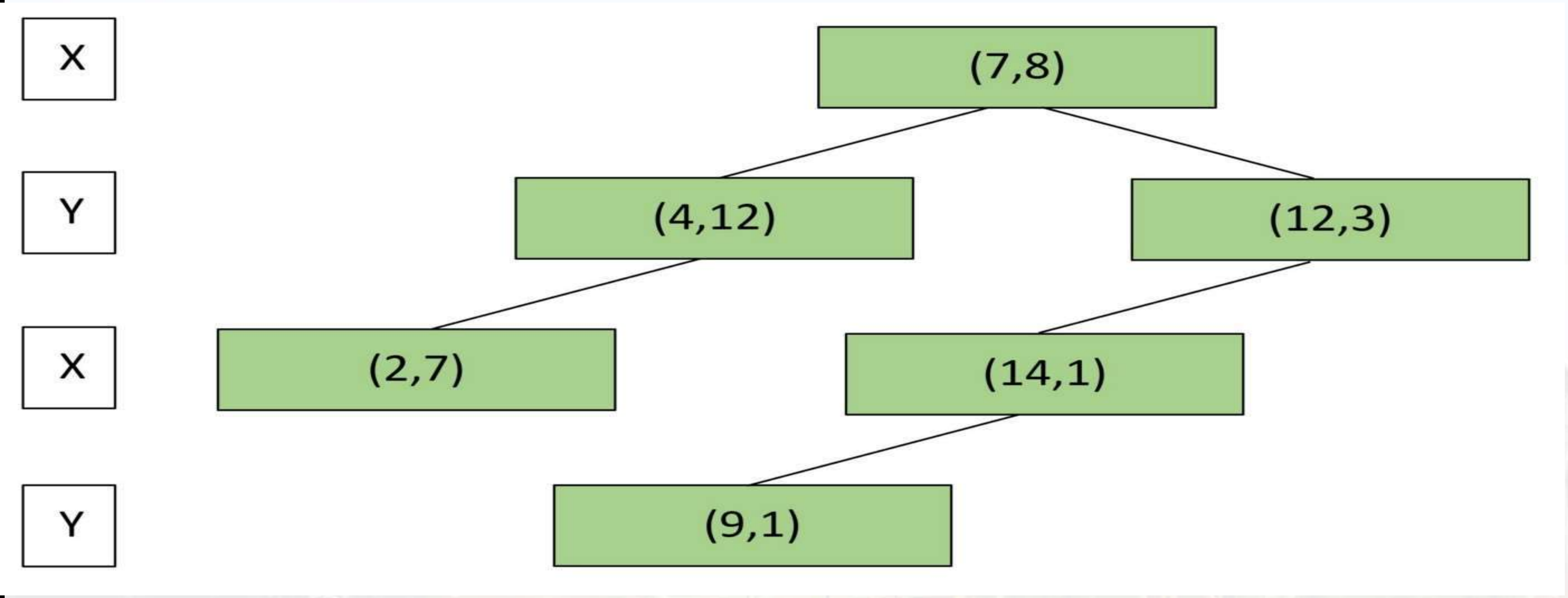
The next set inserted is (4,12). Because the X, 4, is less than the base nodes X, which is 7, the set is inserted to the left leaf node.

KD TREE



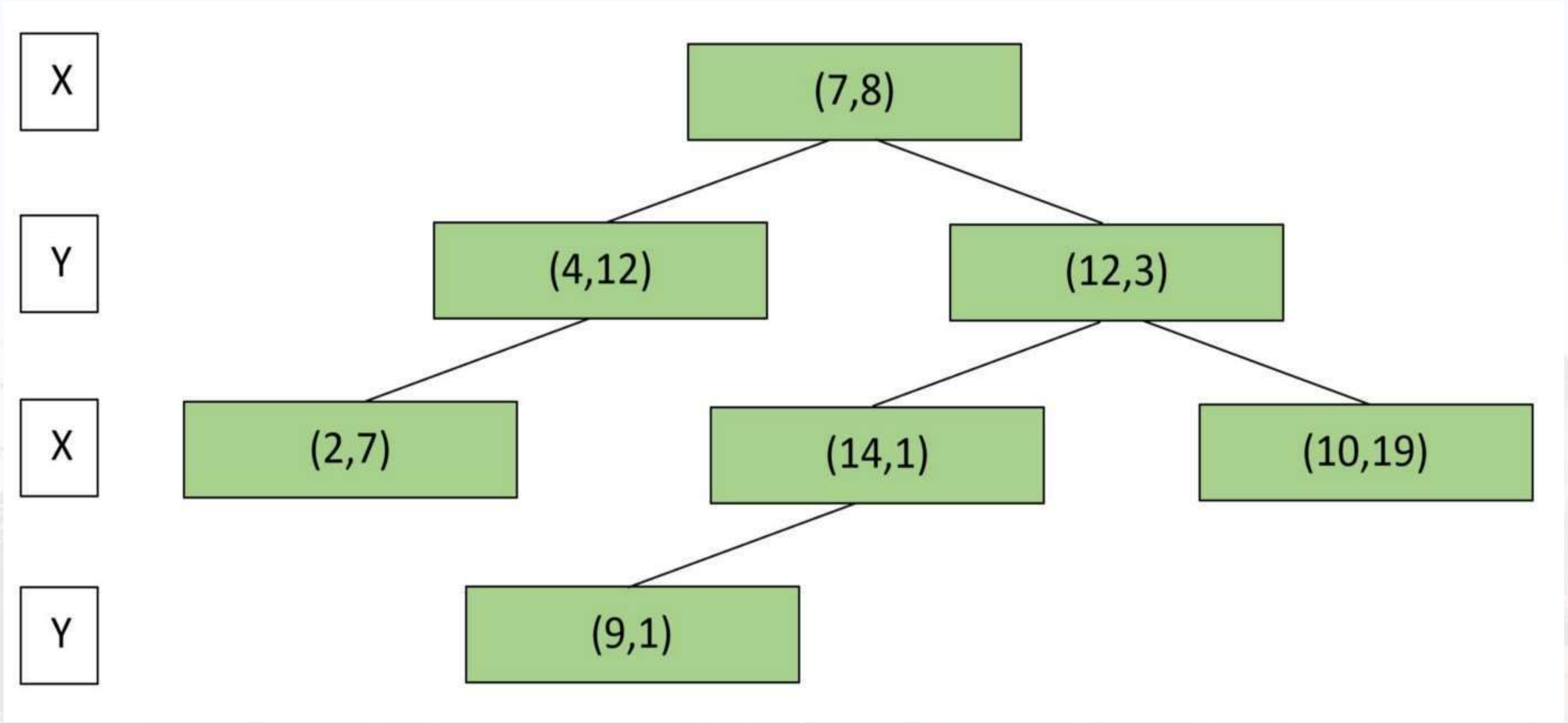
The next set is (9,1). Since the X is greater than the base, it moves to the right leaf. At level 2 we compare the Y values, which is less than the current leaf, so we move to the left leaf of this node. On level 3 we restart and compare by the X value again, but if we had multiple dimensions it would moved to the next dimension. Since 9 is less than 14, we insert it to the left of this leaf node.

KD TREE



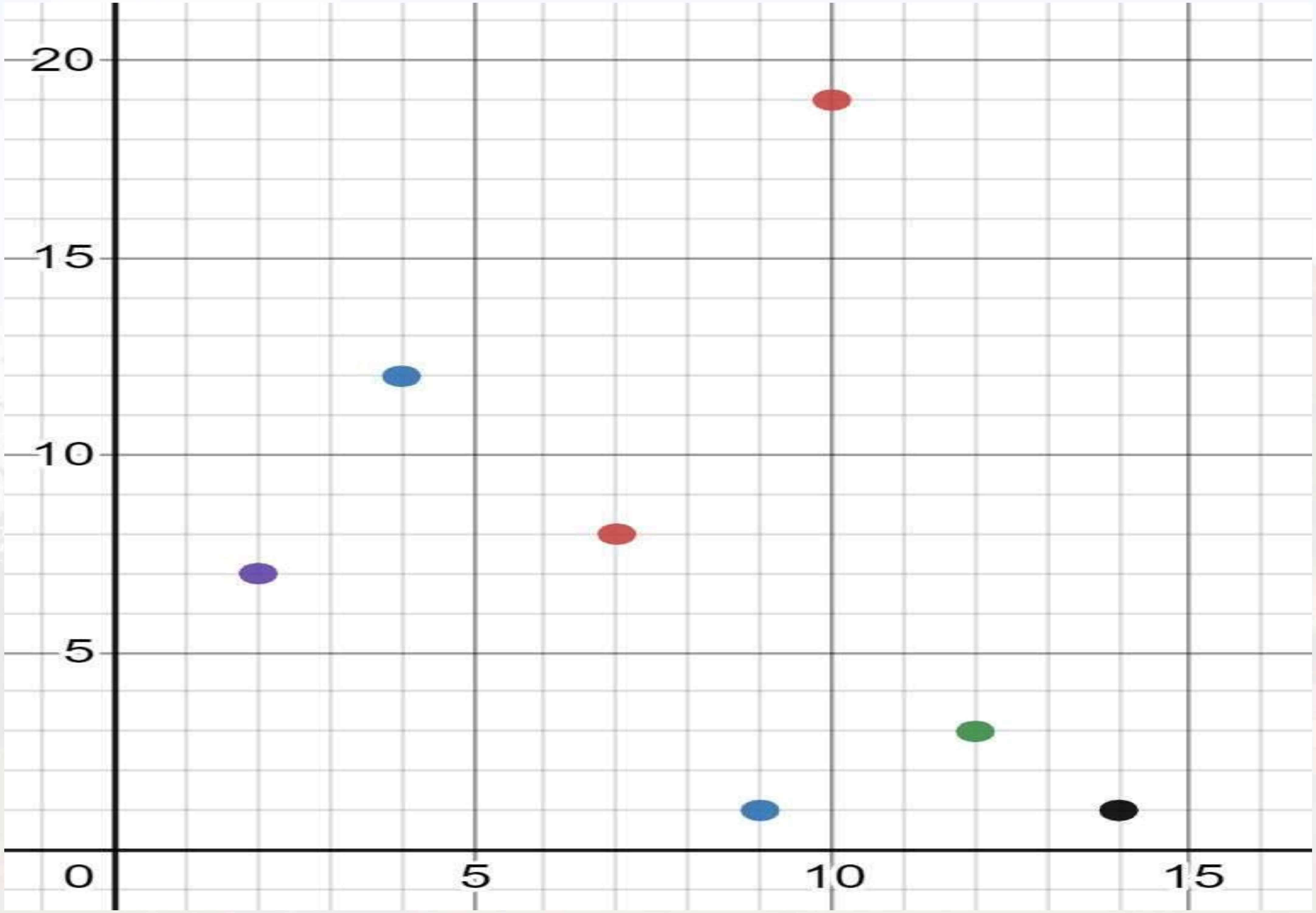
The next set inserted is (2,7). Since the X is less than 7 it compares to the left leaf node. As the Y value is less than 12, the second levels Y value, this set is inserted to the left of the (4,12) leaf node.

KD TREE



The next set inserted is (10,19). 10 is greater than 7 so it moves to the right leaf node. Additionally, 19 is greater than the second levels Y, so it it inserted to the right leaf node.

KD TREE



In the case we are given a duplicate value for the level (such as the X of the current leaf node is 10 and the insertion data X is also 10) we will insert move to the right leaf node of the current node.

SPLAY TREE

Splay trees are the self-balancing or self-adjusted binary search trees. In other words, we can say that the splay trees are the variants of the binary search trees. The prerequisite for the splay trees that we should know about the binary search trees.

- A splay tree is a self-balancing tree, but AVL and Red-Black trees are also self-balancing trees then. What makes the splay tree unique two trees. It has one extra property that makes it unique is splaying.

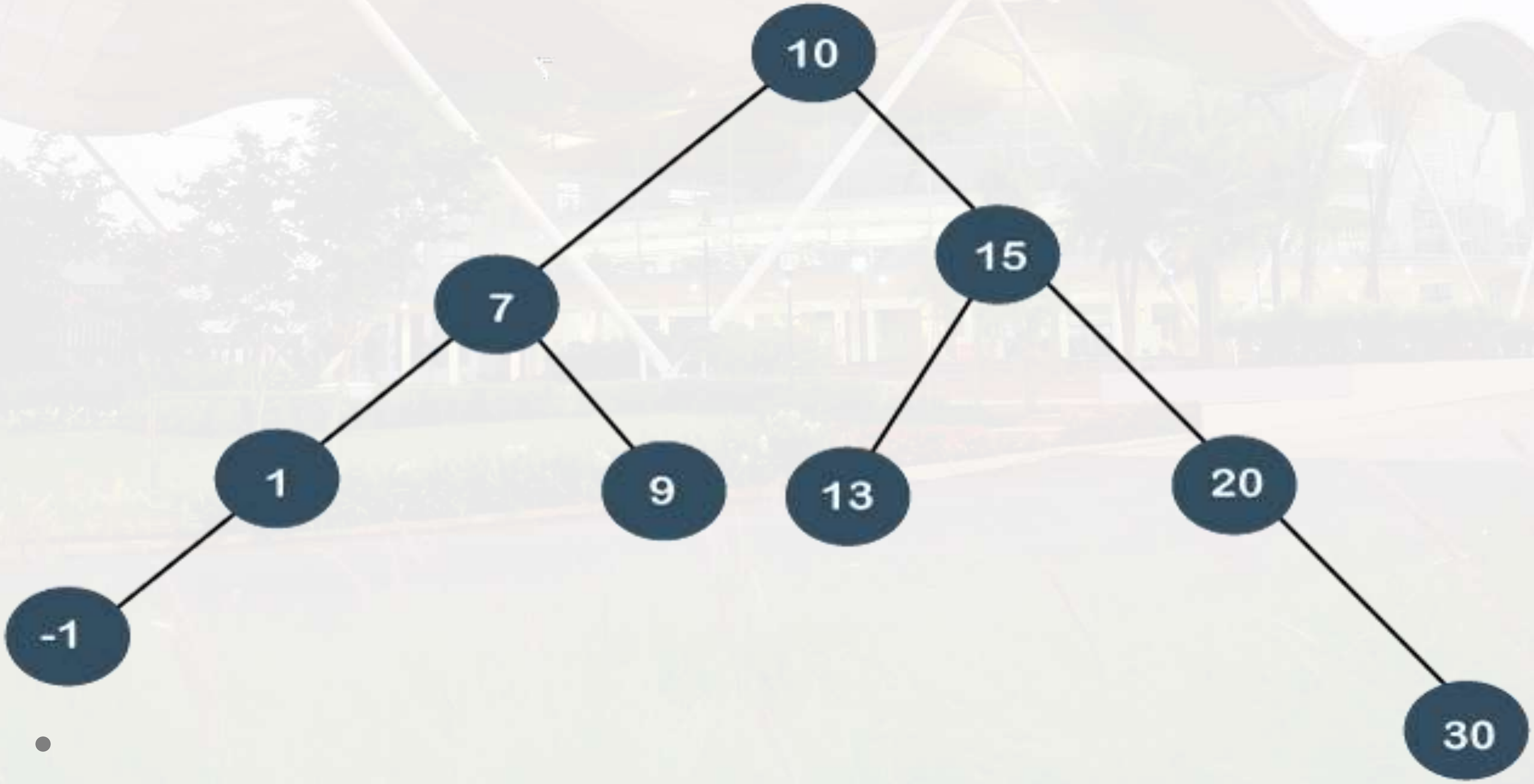
- A splay tree contains the same operations as a Binary search tree, i.e., Insertion, deletion and searching, but it also contains one more operation, i.e., splaying. So. all the operations in the splay tree are followed by splaying.

SPLAY TREE

Splay trees are not strictly balanced trees, but they are roughly balanced trees.

Let's understand the search operation in the splay-tree.

Suppose we want to search 7 element in the tree, which is shown below:



SPLAY TREE

To search any element in the splay tree, first, we will perform the standard binary search tree operation. As 7 is less than 10 so we will come to the left of the root node. After performing the search operation, we need to perform splaying. Here splaying means that the operation that we are performing on any element should become the root node after performing some rearrangements. The rearrangement of the tree will be done through the rotations.

SPLAY TREE

Rotations

There are six types of rotations used for splaying:

1. Zig rotation (Right rotation)
2. Zag rotation (Left rotation)
3. Zig zag (Zig followed by zag)
4. Zag zig (Zag followed by zig)
5. Zig zig (two right rotations)
6. Zag zag (two left rotations)

SPLAY TREE

Rotations

Factors required for selecting a type of rotation

The following are the factors used for selecting a type of rotation:

1. Does the node which we are trying to rotate have a grandparent?
2. Is the node left or right child of the parent?
3. Is the node left or right child of the grandparent?

SPLAY TREE

Cases for the Rotations

Case 1: If the node does not have a grand-parent, and if it is the right child of the parent, then we carry out the left rotation; otherwise, the right rotation is performed.

Case 2: If the node has a grandparent, then based on the following scenarios; the rotation would be performed:

SPLAY TREE

Scenario 1: If the node is the right of the parent and the parent is also right of its parent, then zig zig right right rotation is performed**.**

Scenario 2: If the node is left of a parent, but the parent is right of its parent, then zig zag right left rotation is performed.

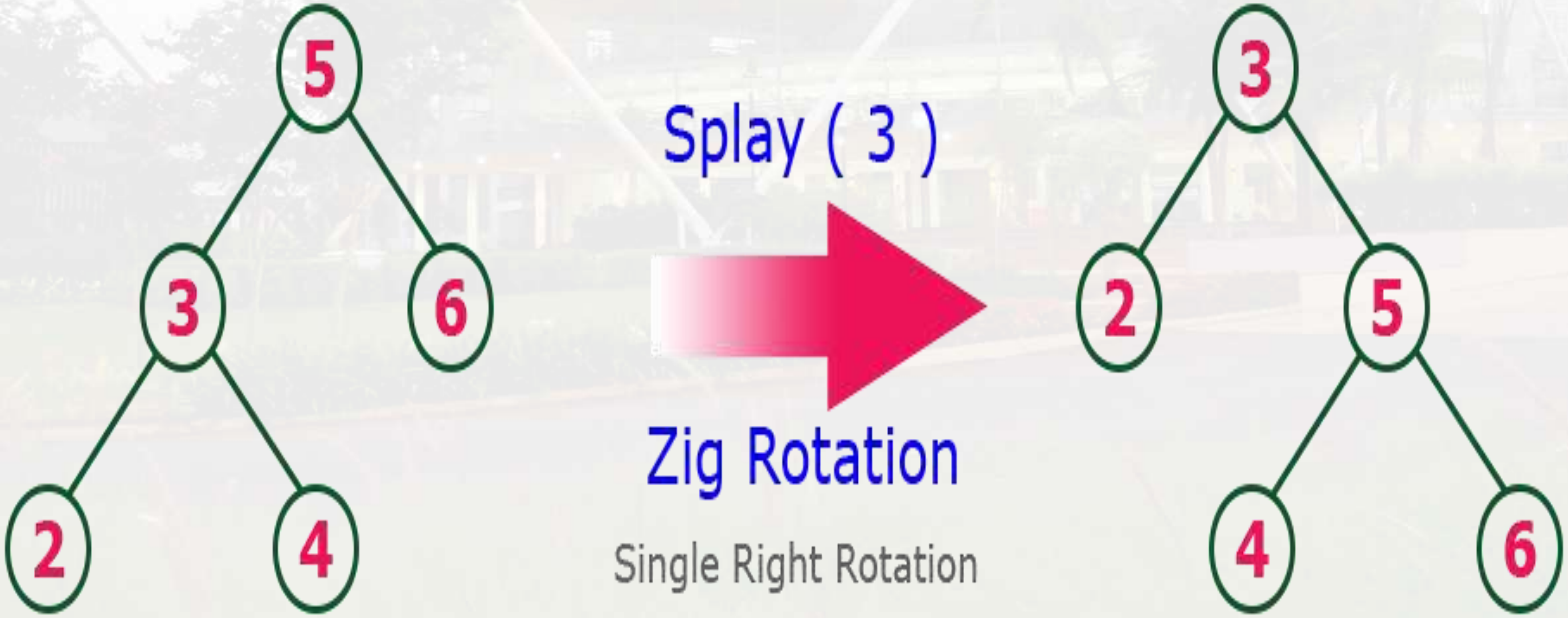
Scenario 3: If the node is right of the parent and the parent is right of its parent, then zig zig left left rotation is performed**.**

Scenario 4: If the node is right of a parent, but the parent is left of its parent, then zig zag right-left rotation is performed.

SPLAY TREE

1. Zig Rotation

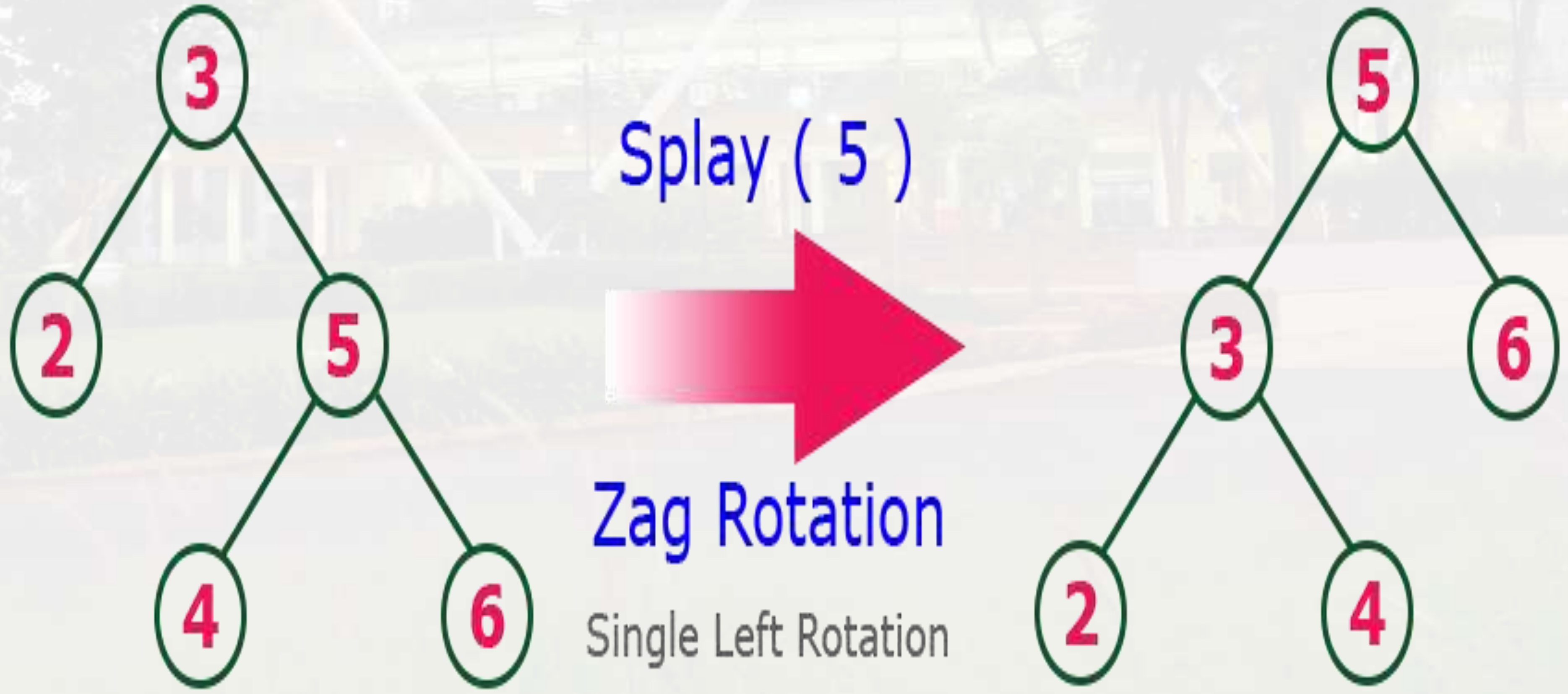
The Zig Rotation in splay tree is similar to the single right rotation in AVL Tree rotations. In zig rotation, every node moves one position to the right from its current position. Consider the following example...



SPLAY TREE

2. Zag Rotation :

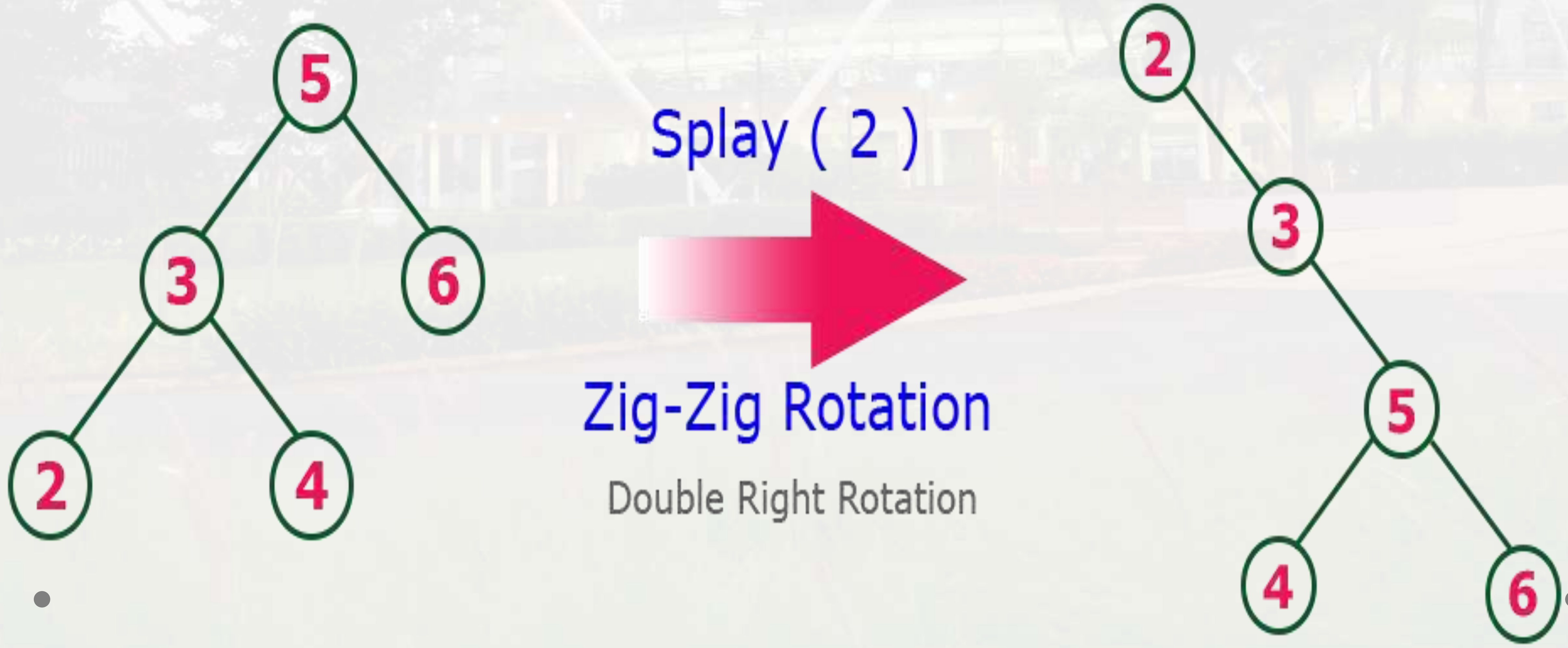
The Zag Rotation in splay tree is similar to the single left rotation in AVL Tree rotations. In zag rotation, every node moves one position to the left from its current position. Consider the following example...



SPLAY TREE

3. Zig-Zig Rotation

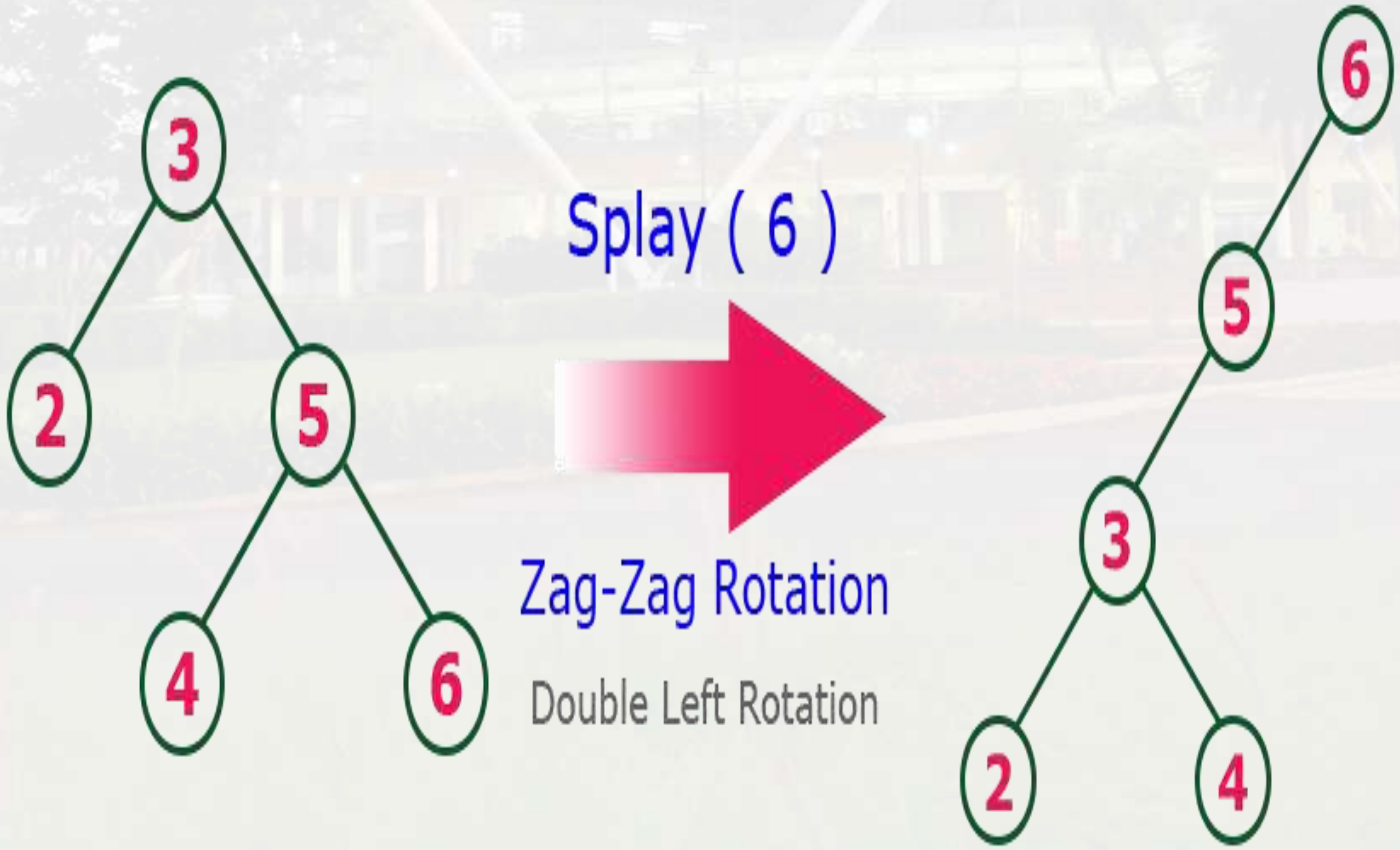
The Zig-Zig Rotation in splay tree is a double zig rotation. In zig-zig rotation, every node moves two positions to the right from its current position. Consider the following example...



SPLAY TREE

4. Zag-Zag Rotation :

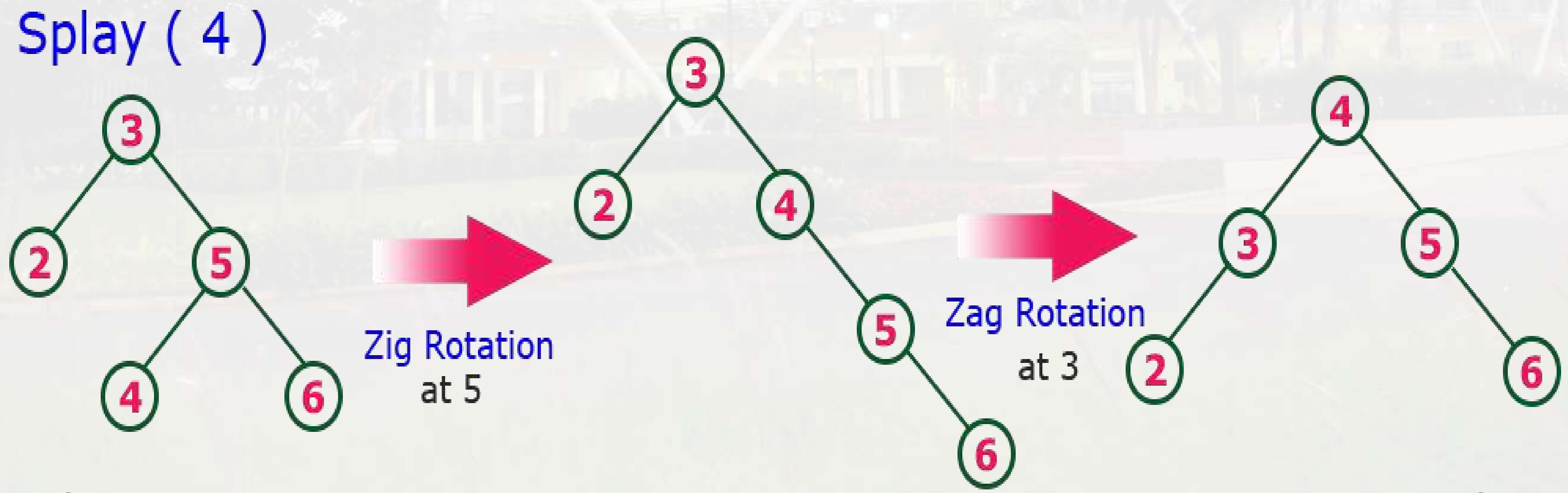
The Zag-Zag Rotation in splay tree is a double zag rotation. In zag-zag rotation, every node moves two positions to the left from its current position. Consider the following example...



SPLAY TREE

5. Zig-Zag Rotation

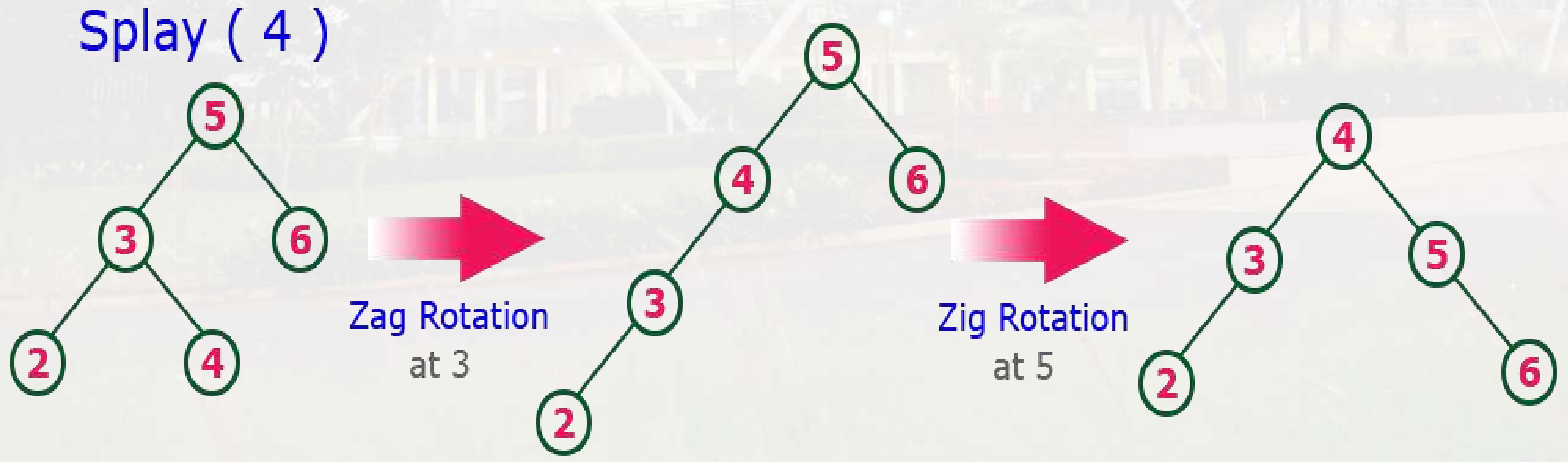
The Zig-Zag Rotation in splay tree is a sequence of zig rotation followed by zag rotation. In zig-zag rotation, every node moves one position to the right followed by one position to the left from its current position. Consider the following example...



SPLAY TREE

6. Zag-Zig Rotation

The Zag-Zig Rotation in splay tree is a sequence of zag rotation followed by zig rotation. In zag-zig rotation, every node moves one position to the left followed by one position to the right from its current position. Consider the following example...



SPLAY TREE

Advantages of Splay tree

1. In the splay tree, we do not need to store the extra information. In contrast, in AVL trees, we need to store the balance factor of each node that requires extra space, and Red-Black trees also require to store one extra bit of information that denotes the color of the node, either Red or Black.
2. It is the fastest type of Binary Search tree for various practical applications. It is used in Windows NT and GCC compilers.
3. It provides better performance as the frequently accessed nodes will move nearer to the root node, due to which the elements can be accessed quickly in splay trees. It is used in the cache implementation as the recently accessed data is stored in the cache so that we do not need to go to the memory for accessing the data, and it takes less time.

SPLAY TREE

Drawback of Splay tree

The major drawback of the splay tree would be that trees are not strictly balanced, i.e., they are roughly balanced. Sometimes the splay trees are linear, so it will take O(n) time complexity.

THANK YOU!!!

My Blog : https://anandgharu.wordpress.com/

Email : gharu.anand@gmail.com