

# **UNIT I**

## **ITERATIVE AND RECURSIVE ALGORITHMS**

Iterative Algorithms: Measures of Progress and Loop Invariants-Paradigm Shift: Sequence of Actions versus Sequence of Assertions- Steps to Develop an Iterative Algorithm-Different Types of Iterative Algorithms-Typical Errors-Recursion-Forward versus Backward- Towers of Hanoi-Checklist for Recursive Algorithms-The Stack Frame-Proving Correctness with Strong Induction- Examples of Recursive Algorithms-Sorting and Selecting Algorithms- Operations on Integers Ackermann's Function- Recursion on Trees-Tree Traversals- Examples- Generalizing the Problem -Heap Sort and Priority Queues-Representing Expressions.

# Iterative Algorithms: Measures of Progress and Loop Invariants

- An *iterative algorithm* to solve a computational problem is a bit like following a road, possibly long and difficult, from your start location to your destination.
- With each iteration, you have a method that takes you a single step closer.
- To ensure that you move forward, you need to have a *measure of progress* telling you how far you are either from your starting location or from your destination.

- You cannot expect to know exactly where the algorithm will go, so you need to expect some weaving and winding.
- On the other hand, you do not want to have to know how to handle every ditch and dead end in the world.
- A compromise between these two is to have a *loop invariant*, which defines a road (or region) that you may not leave.
- As you travel, worry about one step at a time.

- You must know how to get onto the road from any start location.
- From every place along the road, you must know what actions you will take in order to step forward while not leaving the road.
- Finally, when sufficient progress has been made along the road, you must know how to exit and reach your destination in a reasonable amount of time.

# A Paradigm Shift: A Sequence of Actions vs a Sequence of Assertions

- Iterative algorithms requires understanding the difference between a *loop invariant*, which is an *assertion* or picture of the computation at a particular point in time, and the actions that are required to maintain such a loop invariant.
- One of the first important paradigm shifts that programmers struggle to make is from viewing an algorithm as a sequence of actions to viewing it as a sequence of snapshots of the state of the computer.

- Programmers tend to fixate on the first view, because code is a sequence of instructions for action and a computation is a sequence of actions.
- Imagine stopping time at key points during the computation and taking still pictures of the state of the computer.
- Then a computation can equally be viewed as a sequence of such snapshots.
- Having two ways of viewing the same thing gives one both more tools to handle it and a deeper understanding of it.

# The Challenge of the Sequence-of-Actions View

- Suppose one is designing a new algorithm or explaining an algorithm to a friend.
- If one is thinking of it as sequence of actions, then one will likely start at the beginning: Do this. Do that. Do this.
- Shortly one can get lost and not know where one is.
- To handle this, one simultaneously needs to keep track of how the state of the computer changes with each new action.

- In order to know what action to take next, one needs to have a global plan of where the computation is to go.
- To make it worse, the computation has many *IFs* and *LOOPS* so one has to consider all the various paths that the computation may take.

### Advantages of the Sequence of Snapshots View:

- This new paradigm is useful one from which one can think about, explain, or develop an algorithm.



# Steps to Develop an Iterative Algorithm

## Iterative Algorithms:

- A good way to structure many computer programs is to store the key information you currently know in some data structure
- And then have each iteration of the main loop take a step towards your destination by making a simple change to this data.

## Loop Invariant:

- A *loop invariant* expresses important relationships among the variables that must be true at the start of every iteration and when the loop terminates.
- If it is true, then the computation is still on the road.
- If it is false, then the algorithm has failed.

## The Code Structure:

- The basic structure of the code is as follows.

```
begin routine
   $\langle pre-cond \rangle$ 
   $code_{pre-loop}$  % Establish loop invariant
  loop
     $\langle loop-invariant \rangle$ 
    exit when  $\langle exit-cond \rangle$ 
     $code_{loop}$  % Make progress while maintaining the loop invariant
  end loop
   $code_{post-loop}$  % Clean up loose ends
   $\langle post-cond \rangle$ 
end routine
```

## **Proof of Correctness:**

- Naturally, you want to be sure your algorithm will work on all specified inputs and give the correct answer.

## **Running Time:**

- You also want to be sure that your algorithm completes in a reasonable amount of time.

## **The Most Important Steps:**

- If you need to design an algorithm, do not start by typing in code without really knowing how or why the algorithm works.
- Instead, first accomplishing the following tasks.

## **Steps to Develop an Iterative Algorithm:**

- 1.Specifications
- 2.Basic Steps
- 3.Measure of Progress
- 4.The Loop Invariant
- 5.Main Steps
- 6.Make Progress
- 7.Maintain Loop Invariant
- 8.Establishing the Loop Invariant
- 9.Exit Condition
- 10.Ending

- 11. Termination and Running Time
- 12. Special Cases
- 13. Coding and Implementation Details
- 14. Formal Proof

# Different Types of Iterative Algorithms

- More of the Output—Selection Sort

## 1) Specifications:

The goal is to rearrange a list of  $n$  values in no decreasing order.

## 2) Basic Steps:

We will repeatedly select the smallest unselected element.

## 3) Measure of Progress:

The measure of progress is the number  $k$  of elements selected.

#### **4)The Loop Invariant:**

The loop invariant states that the selected elements are the  $k$  smallest of the elements and that these have been sorted. The larger elements are in a set on the side.

#### **5) Main Steps:**

The main step is to find the smallest element from among those in the remaining set of larger elements and to add this newly selected element to the end of the sorted list of elements.

#### **6) Make Progress:**

Progress is made because  $k$  increases.



## 7) Maintain Loop Invariant:

- We must prove that *loop-invariant* & *not exit-cond* & *code loop*  $\Rightarrow$  *loop-invariant* .
- By the previous loop invariant, the newly selected element is at least the size of the previously selected elements.
- By the step, it is no bigger than the elements on the side.
- It follows that it must be the  $k + 1$ st element in the list.
- Hence, moving this element from the set on the side to the end of the sorted list ensures that the selected elements in the new list are the  $k + 1$  smallest and are sorted.

**8) Establishing the Loop Invariant:** We must prove that *pre-cond* & *codepre-loop*  $\Rightarrow$  *loop-invariant* Initially,  $k = 0$  are sorted and all the elements are set aside.

**9) Exit Condition:**

Stop when  $k = n$ .

**10) Ending:** We must prove *loop-invariant* & *exit-cond* & *codepost-loop*  $\Rightarrow$  *post-cond* . By the exit condition, all the elements have been selected, and by the loop invariant these selected elements have been sorted.

**11) Termination and Running Time:** We have not considered how long it takes to find the next smallest element or to handle the data structures.

## **Typical Errors:**

### **Be Clear:**

- The code specifies the current subinterval  $A[i..j]$  with two integers  $i$  and  $j$ .
- Clearly document whether the sublist includes the end points  $i$  and  $j$  or not. It does not matter which, but you must be consistent. Confusion in details like this is the cause of many bugs.

## Math Details:

- Small math operations like computing the index of the middle element of the subinterval  $A(i..j)$  are prone to bugs.
- Check for yourself that the answer is  $mid = \lfloor i+j/2 \rfloor$

# Recursion-Forward versus Backward

## **Recursion**

- Recursion is more than just a programming technique. It has two other uses in computer science and software engineering, namely:
- a way of describing, defining, or specifying things.
- a way of designing solutions to problems (divide and conquer).

## **Stack of Stack Frames:**

- Recursive algorithms are executed using a stack of stackframes.

## **Tree of Stack Frames:**

- This is a useful way of viewing the entire computation at once.
- It is particularly useful when computing the running time of the algorithm.
- However, the structure of the computation tree may be very complex and difficult to understand all at once.

## Friends, on Strong Induction:

- You construct for each friend an instance of the same computational problem that is *smaller* than your own. This is referred as *subinstances*.
- Your friends magically provide you with the solutions to these.
- You then combine these *subsolutions* into a solution for your original instance.
- I refer to this as the *friends* level of abstraction.
- If you prefer, you can call it the *strong induction* level of abstraction and use the word “recursion” instead of “friend.”

# Forward vs. Backward

- Recursion involves designing an algorithm by using it as if it already exists. At first this looks paradoxical.
- Suppose, for example, the key to the
- house that you want to get into is in that same house. If you could get in, you could get the key.
- Then you could open the door, so that you could get in. This is a circular argument.
- It is not a legal recursive program because the subinstance is not smaller.



# Working Forward vs. Backward

- An iterative algorithm works forward.
- It knows about house  $i - 1$ . It uses a loop invariant to show that this house has been opened.
- It searches this house and learns that the key within it is that for house  $i$ .
- Because of this, it decides that house  $i$  would be a good one to go to next.

- A recursion algorithm works backward. It knows about house  $i$ . It wants to get it open.
- It determines that the key for house  $i$  is contained in house  $i - 1$ .
- Hence, opening house  $i - 1$  is a subtask that needs to be accomplished.

Advantages of recursive algorithms over iterative ones:

- The first is that sometimes it is easier to work backward than forward.
- The second is that a recursive algorithm is allowed to have more than one subtask to be solved.
- This forms a tree of houses to open instead of a row of houses.

# Towers of Hanoi

- The towers of Hanoi is a classic puzzle for which the only possible way of solving it is to think recursively.
- **Specification:** The puzzle consists of three poles and a stack of  $N$  disks of different sizes.
- **Precondition:** All the disks are on the first of the three poles.
- **Postcondition:** The goal is to move the stack over to the last pole. See the first and the last parts of figure.



- You are only allowed to take one disk from the top of the stack on one pole and place it on the top of the stack on another pole.
- Another rule is that no disk can be placed on top of a smaller disk.

**Code:**

algorithm *TowersOfHanoi*( $n$ , *source*, *destination*, *spare*)

⟨ *pre-cond* ⟩: The  $n$  smallest disks are on  $pole_{source}$ .

⟨ *post-cond* ⟩: They are moved to  $pole_{destination}$ .

begin

  if( $n \leq 0$ )

    Nothing to do

  else

*TowersOfHanoi*( $n - 1$ , *source*, *spare*, *destination*)

    Move the  $n$ th disk from  $pole_{source}$  to  $pole_{destination}$ .

*TowersOfHanoi*( $n - 1$ , *spare*, *destination*, *source*)

  end if

end algorithm

# Checklist for Recursive Algorithms

- Writing a recursive algorithm is surprisingly hard when you are first starting out and surprisingly easy when you get it.
- This section contains a list of things to think about to make sure that you do not make any of the common mistakes.

- 1) Specifications
- 2) Variables
  - 2.1) Your Input
  - 2.2) Your Output
    - 2.2.1) Every Path
    - 2.2.2) Type of Output
  - 2.3) Your Friend's Input
  - 2.4) Your Friend's Output
  - 2.5) Rarely Need New Inputs or Outputs
  - 2.6) No Global Variables or Global Effects
  - 2.7) Few Local Variables

### 3)Tasks to Complete

- 3.1) Accept Your Mission
- 3.2) Construct Subinstances
- 3.3) Trust Your Friend
- 3.4) Construct Your Solution
- 3.5) Base Cases

### **The Stack Frame**

- **Tree of Stack Frames:**
- Tracing out the entire computation of a recursive algorithm, one line of code at a time, can get incredibly complex.



- This is why the friends level of abstraction, which considers one stack frame at a time, is the best way to understand, explain, and design a recursive algorithm
- However, it is also useful to have some picture of the entire computation.
- For this, the tree-of-stack-frames level of abstraction is best.
- The key thing to understand is the difference between a particular routine and a particular execution of a routine on a particular input instance.

- A single routine can at one moment in time have many executions going on. Each such execution is referred to as a *stack frame*.

### **Stack of Stack Frames:**

- The algorithm is actually implemented on a computer by a stack of stack frames. What is stored in the computer memory at any given point in time is only a single path down the tree.
- The tree represents what occurs throughout time.

## Using a Stack Frame:

- Recall that a stack is a data structure in which either a new element is *pushed* onto the top or the last element to have been added is *popped* off
- Let us denote the top stack frame by  $A$ . When the execution of  $A$  makes a subroutine call to a routine with some input values, a stack frame is created for this new instance.
- This frame denoted  $B$  is pushed onto the stack after that for  $A$ .

- In addition to a separate copy of the local variables for the routine, it contains a pointer to the next line of code that *A* must execute when *B* returns.
- When *B* returns, its stack frame is popped, and *A* continues to execute at the line of code that had been indicated within *B*.
- When *A* completes, it too is popped off the stack.

# Proving Correctness with Strong Induction

## **Strong Induction:**

- Strong induction is similar to induction, except that instead of assuming only  $S(n - 1)$  to prove  $S(n)$ , you must assume all of  $S(0), S(1), S(2), \dots, S(n - 1)$ .

## **A Statement for Each $n$ :**

For each value of  $n \geq 0$ , let  $S(n)$  represent a Boolean statement. For some values of  $n$  this statement may be true, and for others it may be false.

## Goal:

Our goal is to prove that it is true for every value of  $n$ , namely that  $\forall n \geq 0, S(n)$ .

## Proof Outline:

Proof by strong induction on  $n$ .

**Induction Hypothesis:** For each  $n \geq 0$ , let  $S(n)$  be the statement that . . . . (It is important to state this clearly.)

## Base Case:

Prove that the statement  $S(0)$  is true.

**Induction Step:** For each  $n \geq 0$ , prove  $S(0), S(1), S(2), \dots, S(n-1) \Rightarrow S(n)$ .

## Conclusion:

By way of induction, we can conclude that  $\forall n \geq 0, S(n)$ .

# Examples of Recursive Algorithms

## Sorting and Selecting Algorithms:

- The classic divide-and-conquer algorithms are merge sort and quick sort. They both have the following basic structure.

## General Recursive Sorting Algorithm:

- Take the given list of objects to be sorted (numbers, strings, student records, etc.).
- Split the list into two sublists.

- Recursively have friends sort each of the two sublists.
- Combine the two sorted sublists into one entirely sorted list.
- This process leads to four different algorithms, depending on the following factors,
  - Sizes
  - Work

## **Operations on Integers:**

- Raising an integer to a power  $bN$ , multiplying  $x \times y$ , and matrix multiplication each have surprising divide-and-conquer algorithms.



# Ackermann's Function

- If you are wondering just how slowly a program can run, consider the algorithm below.
- Assume the input parameters  $n$  and  $k$  are natural numbers.

## Recurrence Relation:

- Let  $T_k(n)$  denote the value returned by  $A(k, n)$ . This gives  $T_0(n) = 2 + n$ ,  $T_1(0) = 0$ ,  $T_k(0) = 1$  for  $k \geq 2$ , and  $T_k(n) = T_{k-1}(T_k(n-1))$  for  $k > 0$  and  $n > 0$ .

**Algorithm:**

```
algorithm  $A(k, n)$ 
  if(  $k = 0$ ) then
    return(  $n + 1 + 1$  )
  else
    if(  $n = 0$ ) then
      if(  $k = 1$ ) then
        return( 0 )
      else
        return( 1 )
    else
      return(  $A(k - 1, A(k, n - 1))$ )
    end if
  end if
end algorithm
```

## Running Time:

- The only way that the program builds up a big number is by continually incrementing it by one.
- Hence, the number of times one is added is at least as huge as the value  $Tk(n)$  returned.

## Recursion on Trees:

- One key application of recursive algorithms is to perform actions on trees, because trees themselves have a recursive definition

**Recursive Definition of Tree:** A tree is either:

- an empty tree (zero nodes) or
- a root node with some subtrees as children.
- A *binary tree* is a special kind of tree where each node has a right and a left subtree

### **Tree Traversals:**

A task one needs to be able to perform on a binary tree is to traverse it, visiting each node once, in one of three defined orders.

- Before one becomes familiar with recursive programs, one tends to think about computation iteratively, “I visit this node first, then this one, then this one, and so on.”
- Each iteration, the program says “I just visited this node, so now let me find the next node to visit.” Surprisingly, such a computation is hard to code.
- The reason is that binary trees by their very nature have a recursive structure.

# Examples

- Here is a list of problems involving binary trees.
- 1. Return the maximum of data fields of nodes.
- 2. Return the height of the tree.
- 3. Return the number of leaves in the tree. (A harder one.)
- 4. Copy the tree.

# Generalizing the Problem

- When writing a recursive algorithm for a problem it is easier to solve a more general version of the problem, providing more information about the original instance or asking for more information about subinstances.

## **Subinstance:**

- It is better to combine the *IsBSTtree* and the *Min* and *Max* routines into one routine so that the tree only needs to be traversed once.

**algorithm** *IsBSTtree* (*tree*)

⟨ *pre-cond* ⟩: *tree* is a binary tree.

⟨ *post-cond* ⟩: The output indicates whether it is a BST. It also gives the minimum and the maximum values in the tree.

**begin**

**if** (*tree* = *emptyTree*) **then**

**return** ⟨ *Yes*,  $\infty$ ,  $-\infty$  ⟩

**else**

        ⟨ *leftIs*, *leftMin*, *leftMax* ⟩ = *IsBSTtree*(*leftSub*(*tree*) )

        ⟨ *rightIs*, *rightMin*, *rightMax* ⟩ = *IsBSTtree*( *rightSub*(*tree*) )

*min* = *min*(*leftMin*, *rightMin*, *rootKey*(*tree*))

*max* = *max*(*leftMax*, *rightMax*, *rootKey*(*tree*))

**if** ( *leftIs* and *rightIs* and *leftMax* ≤ *rootKey*(*tree*) ≤ *rightMin* ) **then**

*isBST* = *Yes*

**else**

*isBST* = *No*

**Recursion**

**end if**

**return** ⟨ *isBST*, *min*, *max* ⟩

**end if**

**end algorithm**



## Original Instance:

- Another elegant algorithm for the *IsBST* problem generalizes the problem in order to provide your friend more information about your subinstance.
- Here the more general problem, in addition to the tree, will provide a range of values  $[min, max]$  and ask whether the tree is a BST with values within this range.
- The original problem is solved using
- *IsBSTtree*(tree,  $[-\infty, \infty]$ ).

algorithm *IsBSTtree*(*tree*, [*min*,*max*])

⟨ *pre-cond* ⟩: *tree* is a binary tree. In addition, [*min*, *max*] is a range of values.

⟨ *post-cond* ⟩: The output indicates whether it is a BST with values within this range.

begin

    if(*tree* = *emptyTree*) then

        return *Yes*

    else if( *rootKey*(*tree*)  $\in$  [*min*, *max*] and

*IsBSTtree*( *leftSub*(*tree*), [*min*,*rootKey*(*tree*)])) and

*IsBSTtree*( *rightSub*(*tree*), [*rootKey*(*tree*),*max*]) then

        return *Yes*

    else

        return *No*

    end if

end algorithm

# Heap Sort and Priority Queues

- Heap sort is a fast sorting algorithm that is easy to implement.
- Like quick sort, it has the advantage of being done in place in memory, whereas merge and radix–counting sorts require an auxiliary array of memory to transfer the data to.

## **Completely Balanced Binary Tree:**

- The values being sorted are stored in a binary tree that is completely balanced, i.e., every level of the tree is completely full except for the bottom level, which is filled in from the left.

## **Definition of a Heap:**

- A heap imposes a partial order on the set of values, requiring that the value of each node be greater than or equal to that of each of the node's children.
- There are no rules about whether the left or the right child is larger.

## **Maximum at Root:**

- An implication of the heap rules is that the root contains the maximum value.
- The maximum may appear repeatedly in other places as well.

# The Heapify Problem:

## Specifications:

- ***Precondition:*** The input is a balanced binary tree such that its left and right subtrees are heaps. (That is, it is a heap except that its root might not be larger than that of its children)
- ***Postcondition:*** Its values are rearranged in place to make it complete heap.

## Iterative Algorithm:

- A good loop invariant would be “The entire tree is a heap except that node  $i$  might not be greater or equal to both of its children.
- As well, the value of  $i$ 's parent is at least the value of  $i$  and of  $i$ 's children.”
- When  $i$  is the root, this is the precondition.
- The algorithm proceeds as in the recursive algorithm.
- Node  $i$  follows one path down the tree to a leaf.
- When  $i$  is a leaf, the whole tree is a heap.

## The MakeHeap Problem:

### Specifications:

- ***Precondition:*** The input is an array of numbers, which can be viewed as a balanced binary tree of numbers.
- ***Postcondition:*** Its values are rearranged in place to make it heap.

## The Heap Sort Problem:

### Specifications:

- ***Precondition:*** The input is an array of numbers.
- ***Postcondition:*** Its values are rearranged in place to be in sorted order.

Code:

algorithm *HeapSort*( )

⟨ *pre-cond* ⟩: The input is an array of numbers.

⟨ *post-cond* ⟩: Its values are rearranged in place to be in sorted order.

begin

*MakeHeap*( )

$i = n$

    loop

        ⟨ *loop-invariant* ⟩: The  $n - i$  largest elements have been removed and are sorted in  $A[i + 1, n]$ , and the remaining  $i$  elements form a heap in  $A[1, i]$ .

        exit when  $i = 1$

        swap( $A[root]$ ,  $A[i]$ )

$i = i - 1$

*Heapify*( $root$ )      % On a heap of size  $i$ .

    end loop

end algorithm



# Priority Queues

- Like stacks and queues, priority queues are an important ADT.
- **Definition:** A *priority queue* consists of:
  - **Data:** A set of elements, each of which is associated with an integer that is referred to as the *priority* of the element.

## **Operations:**

### ***Insert an Element:***

An element, along with its priority, is added to the queue.

### ***Change Priority:***

The priority of an element already in the queue is changed. The routine is passed a pointer to the element within the priority queue and its new priority. ***Remove an Element:*** Removes and returns an element of the highest priority from the queue.

# Representing Expressions with Trees

- consider how to represent multivariate expressions using binary trees.
- We will develop the algorithms to evaluate, copy, differentiate, simplify, and print such an expression.
- Though these are seemingly complex problems, they have simplerecursive solutions.

## Recursive Definition of an Expression:

- Single variables  $x$ ,  $y$ , and  $z$  and single real values are themselves expressions.
- If  $f$  and  $g$  are expressions, then  $f + g$ ,  $f - g$ ,  $f * g$ , and  $f/g$  are also expressions.

**Tree Data Structure:** The recursive definition of an expression directly mirrors that of a binary tree. Because of this, a binary tree is a natural data structure for storing an expression. (Conversely, you can use an expression to represent a binary tree.)

# Non-Linear Data Structure

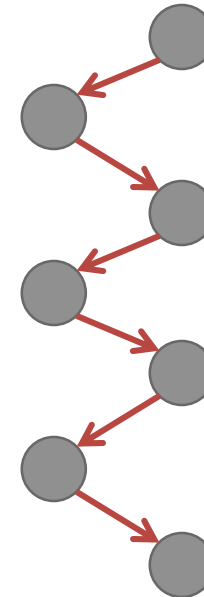
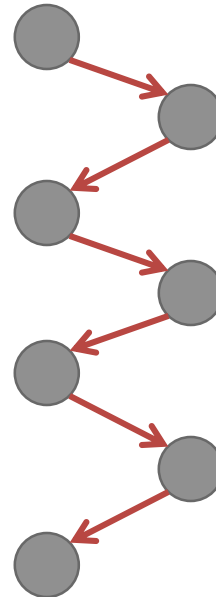
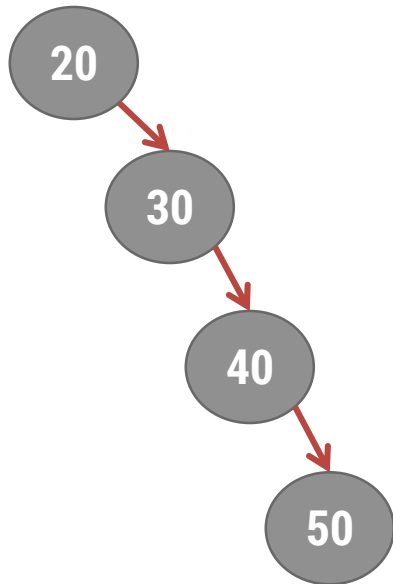
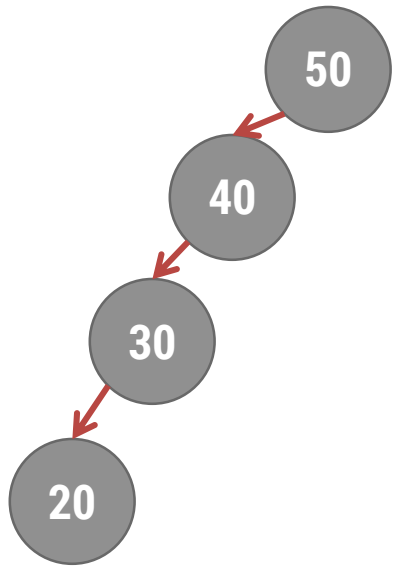
## Tree Part-3

- ◆ Height/Weight – Balanced Tree
- ◆ Multiway Search Tree (B-Tree)

# Balanced Tree

- ▶ Binary Search Tree gives advantage of Fast Search, but sometimes in few cases we are not able to get this advantage. E.g. look into worst case BST
- ▶ Balanced binary trees are classified into two categories
  - ➔ Height Balanced Tree (AVL Tree)
  - ➔ Weight Balanced Tree

## Worst search time cases for Binary Search Tree



# Height Balanced Tree (AVL Tree)

- ▶ AVL tree is a height-balanced binary search tree.
- ▶ AVL tree is also a binary search tree but it is a balanced tree. A binary tree is said to be balanced if, the difference between the heights of left and right subtrees of every node in the tree is either -1, 0 or +1.
- ▶ In other words, a binary tree is said to be balanced if the height of left and right children of every node differ by either -1, 0 or +1.
- ▶ In an AVL tree, every node maintains an extra information known as **balance factor**.
- ▶ The AVL tree was introduced in the year 1962 by G.M. Adelson-Velsky and E.M. Landis.
- ▶ An AVL tree is defined as follows...

*An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1.*



# Balance Factor

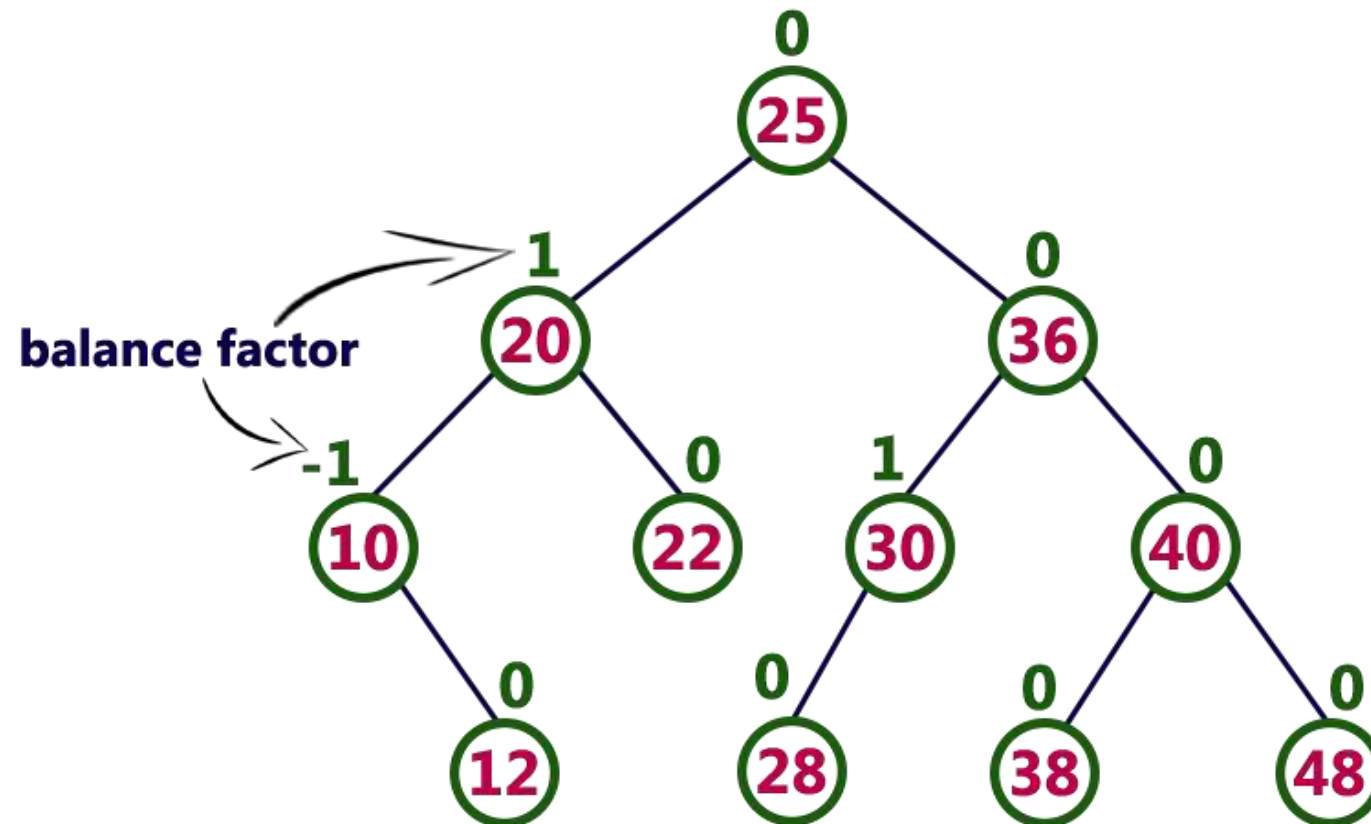
- ▶ Balance factor of a node is the difference between the heights of the left and right subtrees of that node.
- ▶ The balance factor of a node is calculated either **height of left subtree - height of right subtree** (OR) **height of right subtree - height of left subtree**.

In the following explanation, we calculate as follows...

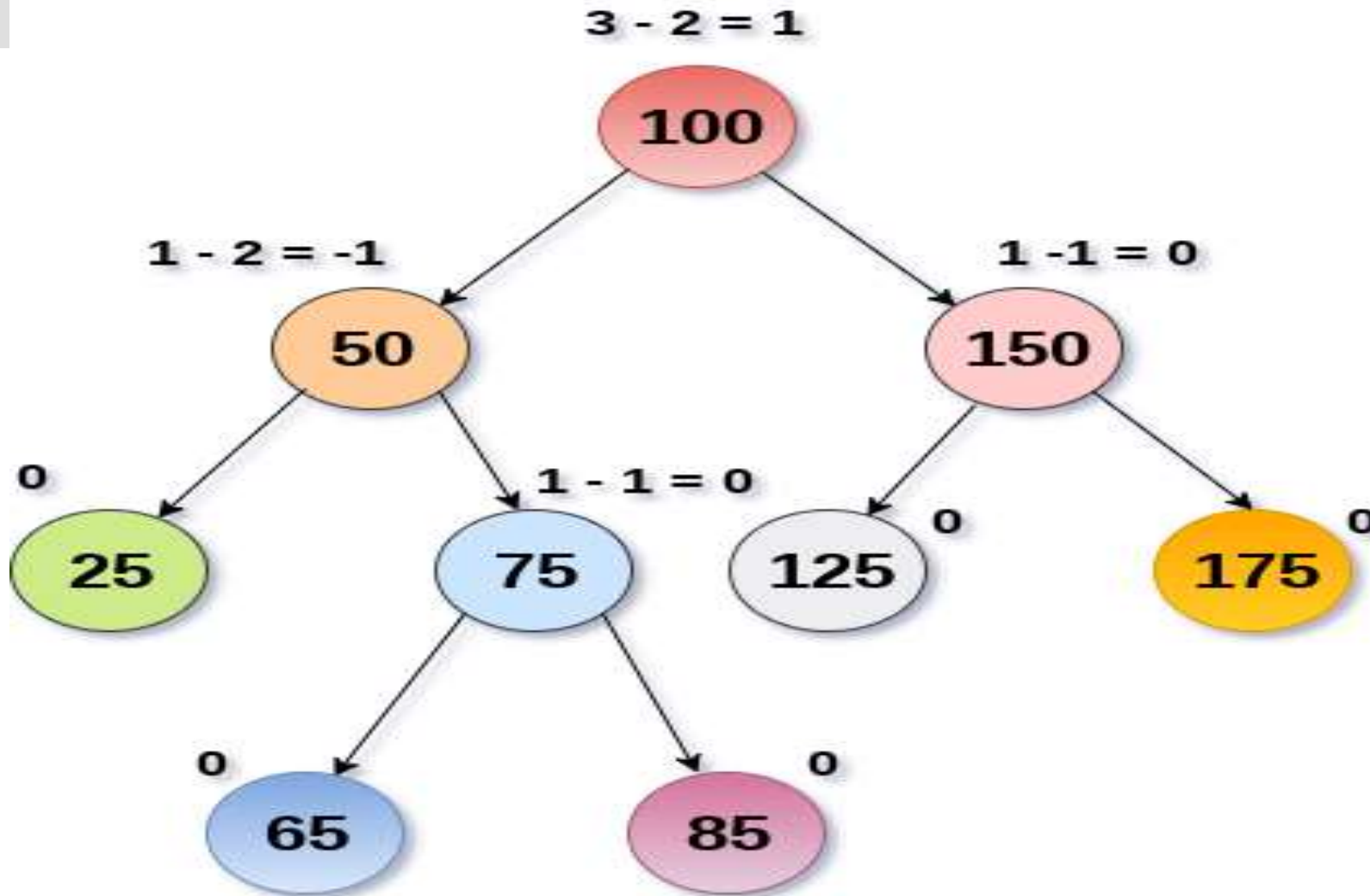
**Balance factor = heightOfLeftSubtree – heightOfRightSubtree**



# Example of AVL Tree

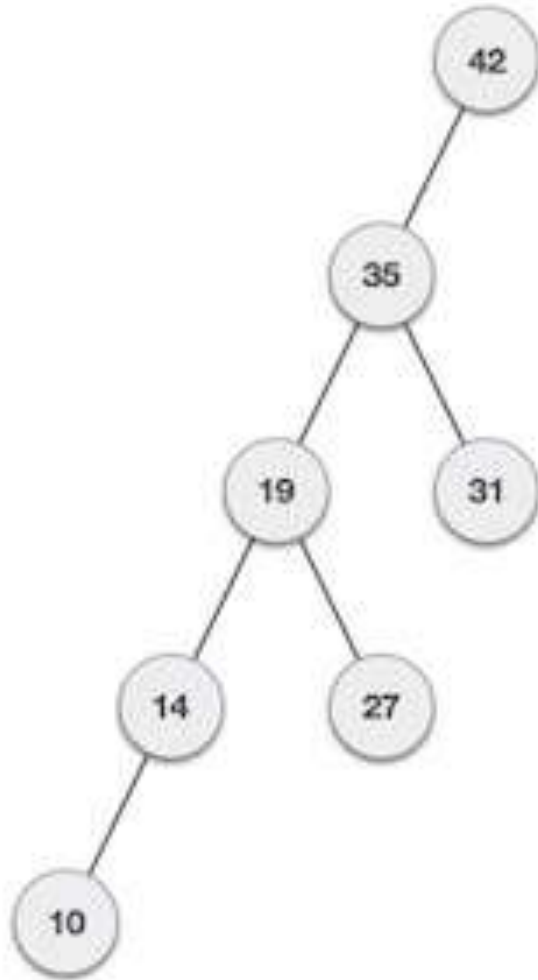


# Example of AVL Tree

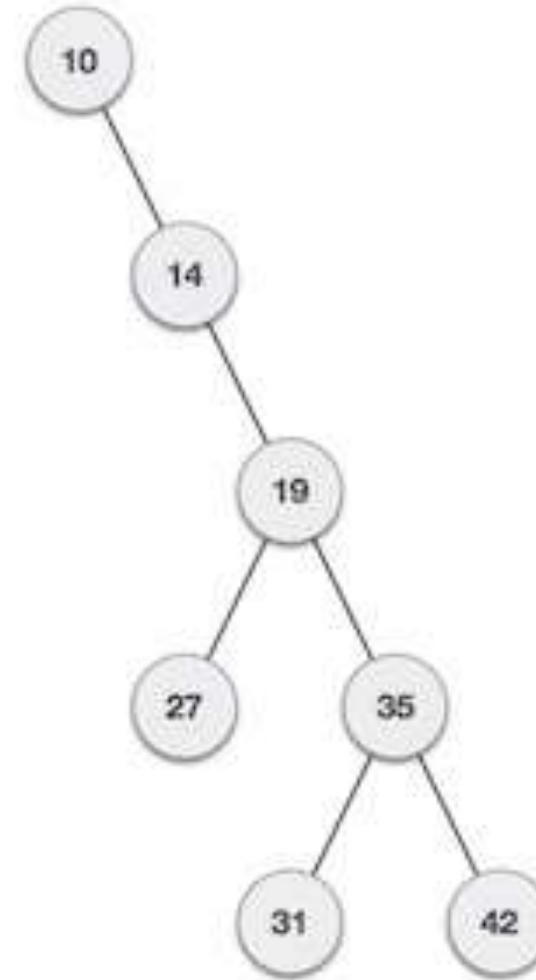


**AVL Tree**

# Example of AVL Tree - Unbalanced



If input 'appears' non-increasing manner



If input 'appears' in non-decreasing manner

# AVL Tree Rotations

- ▶ In AVL tree, after performing operations like insertion and deletion
- ▶ we need to check the **balance factor** of every node in the tree.
- ▶ If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced.
- ▶ Whenever the tree becomes imbalanced due to any operation we use **rotation** operations to make the tree balanced.

Rotation operations are used to make the tree balanced.

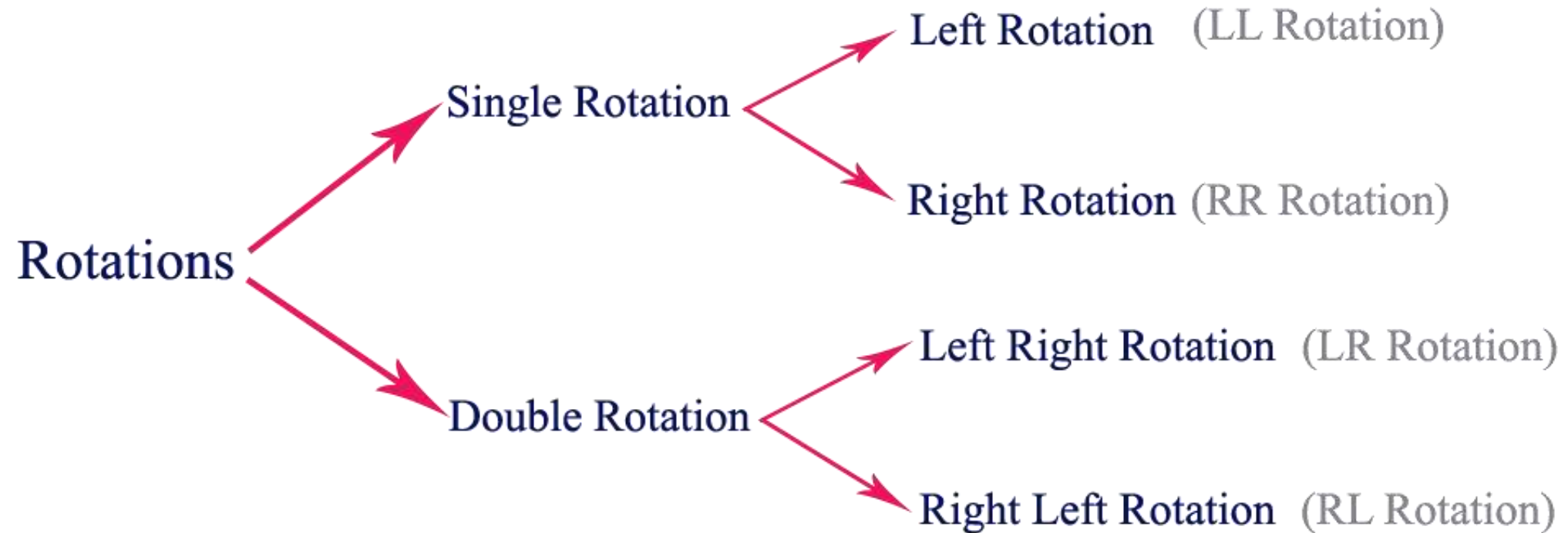
- ▶ **Rotation is the process of moving nodes either to left or to right to make the tree balanced.**

# AVL Tree Rotations

There are basically four types of rotations which are as follows:

1. L L rotation: Inserted node is in the left subtree of left subtree of A
2. R R rotation : Inserted node is in the right subtree of right subtree of A
3. L R rotation : Inserted node is in the right subtree of left subtree of A
4. R L rotation : Inserted node is in the left subtree of right subtree of A

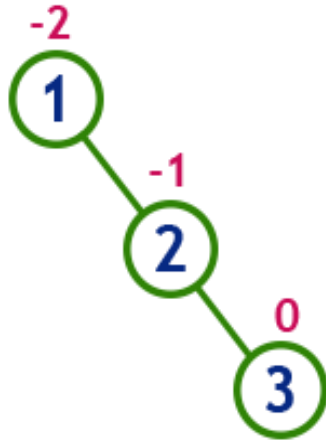
# AVL Tree Rotations



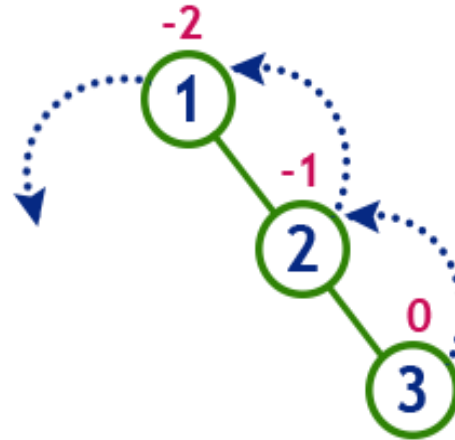
# Single Left Rotation (LL Rotation)

In LL Rotation, every node moves one position to left from the current position. To understand LL Rotation, let us consider the following insertion operation in AVL Tree...

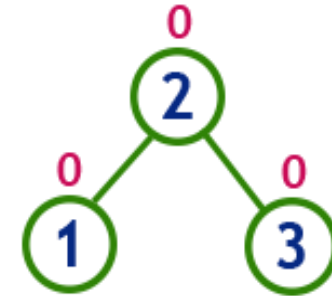
insert 1, 2 and 3



Tree is imbalanced



To make balanced we use LL Rotation which moves nodes one position to left

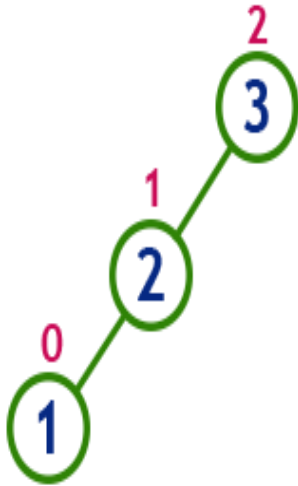


After LL Rotation Tree is Balanced

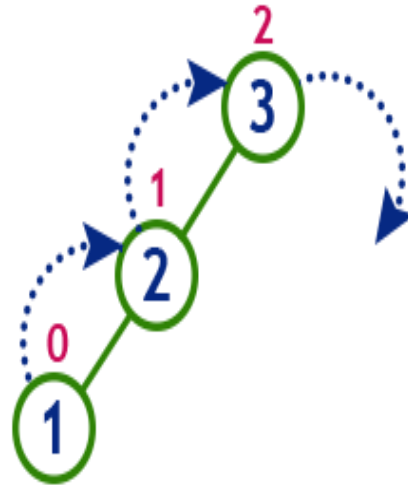
# Single Right Rotation (RR Rotation)

In RR Rotation, every node moves one position to right from the current position. To understand RR Rotation, let us consider the following insertion operation in AVL Tree...

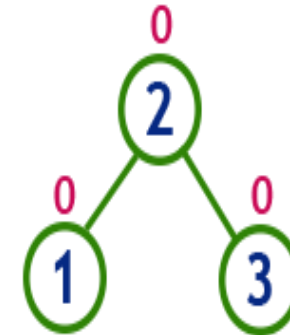
insert 3, 2 and 1



Tree is imbalanced  
because node 3 has balance factor 2



To make balanced we use  
RR Rotation which moves  
nodes one position to right

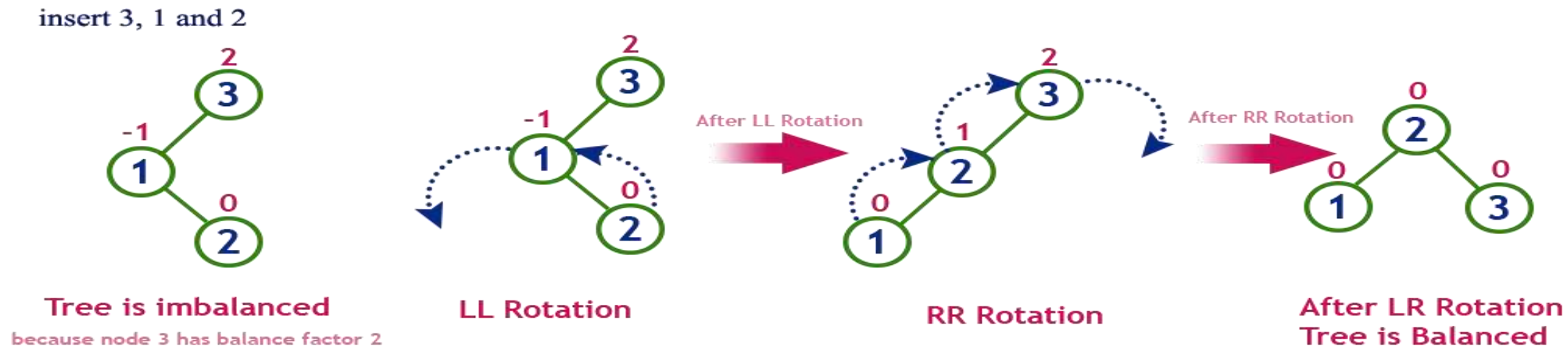


After RR Rotation  
Tree is Balanced



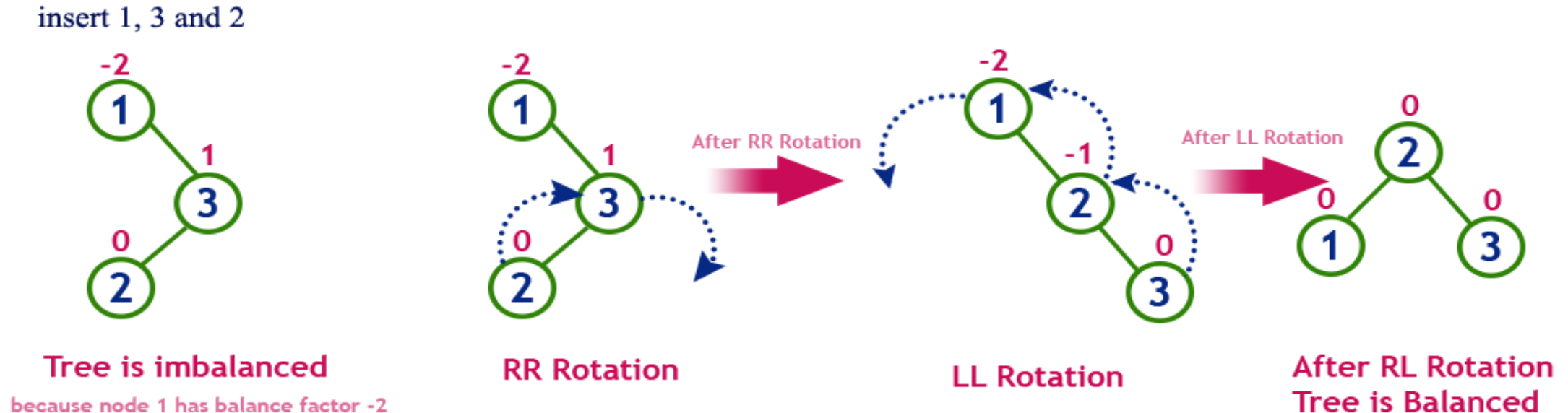
# Left Right Rotation (LR Rotation)

- The LR Rotation is a sequence of single left rotation followed by a single right rotation.
- In LR Rotation, at first, every node moves one position to the left and one position to right from the current position.
- To understand LR Rotation, let us consider the following insertion operation in AVL Tree...



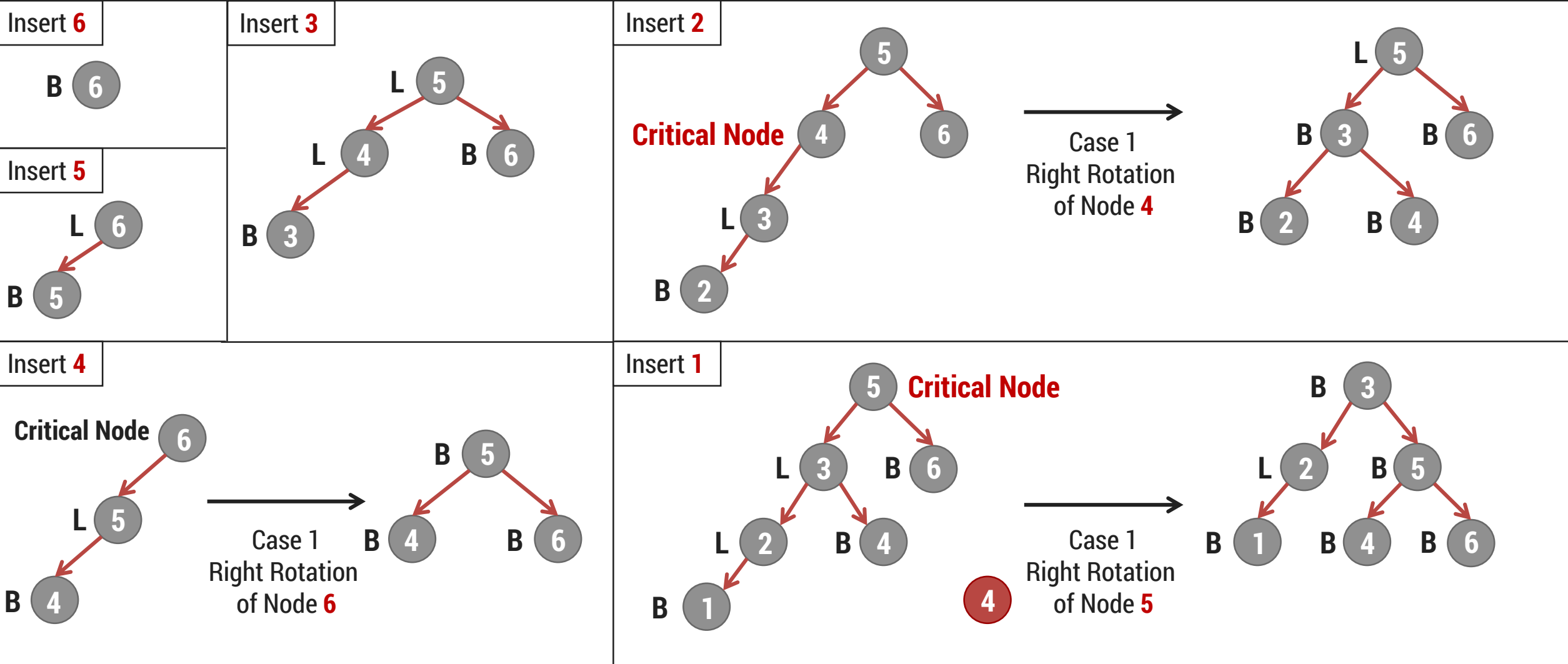
# Right Left Rotation (RL Rotation)

- The RL Rotation is sequence of single right rotation followed by single left rotation.
- In RL Rotation, at first every node moves one position to right and one position to left from the current position.
- To understand RL Rotation, let us consider the following insertion operation in AVL Tree...



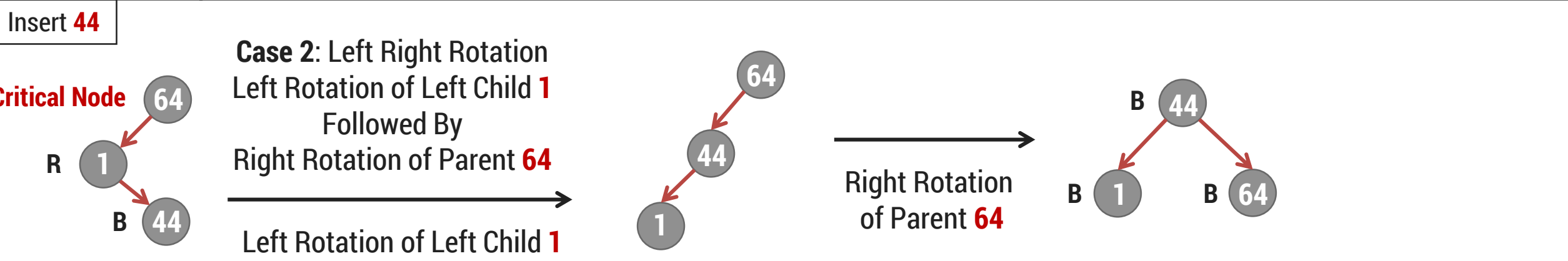
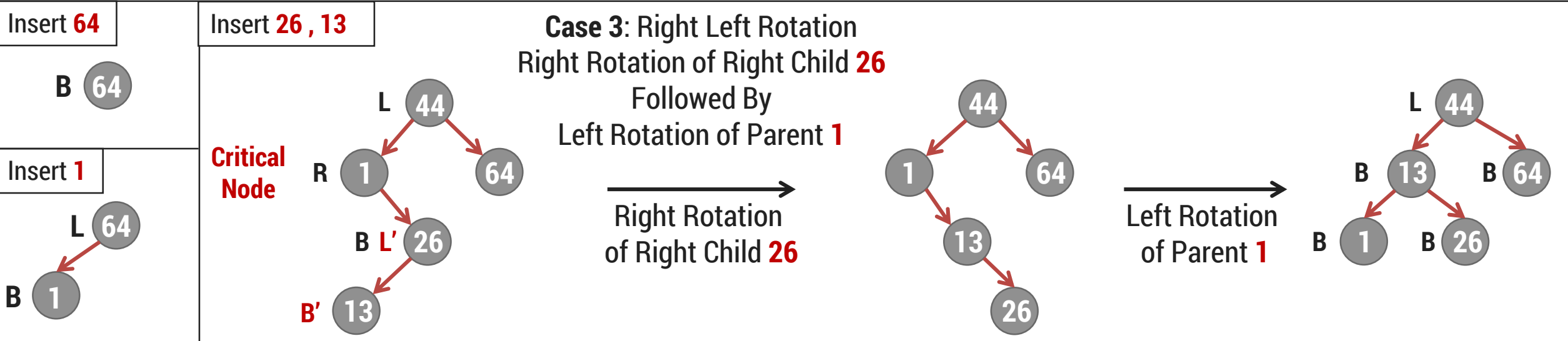
# Construct AVL Search Tree

Construct AVL Search tree by inserting following elements in order of their occurrence **6, 5, 4, 3, 2, 1**



# Construct AVL Search Tree

Construct AVL Search tree by inserting following elements in order of their occurrence **64, 1, 44, 26, 13, 110, 98, 85**

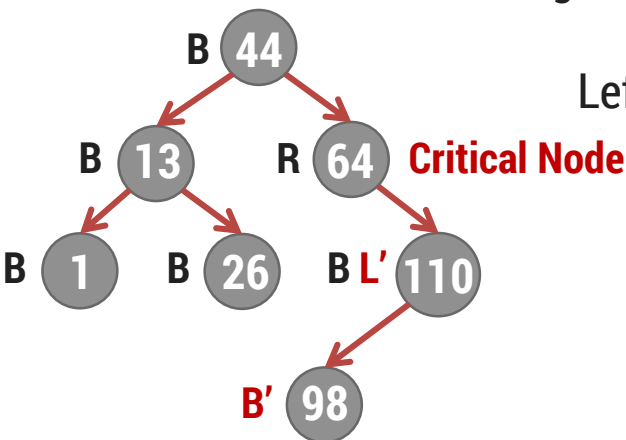


# Construct AVL Search Tree

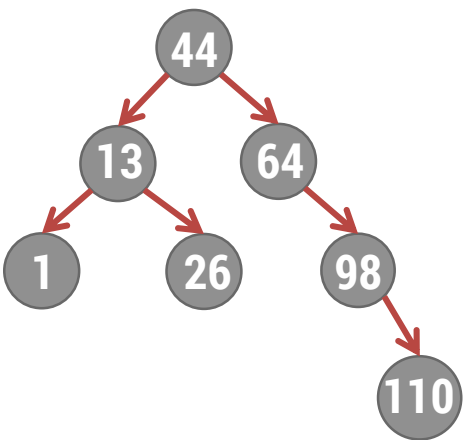
Construct AVL Search tree by inserting following elements in order of their occurrence **64, 1, 44, 26, 13, 110, 98, 85**

Insert **110, 98**

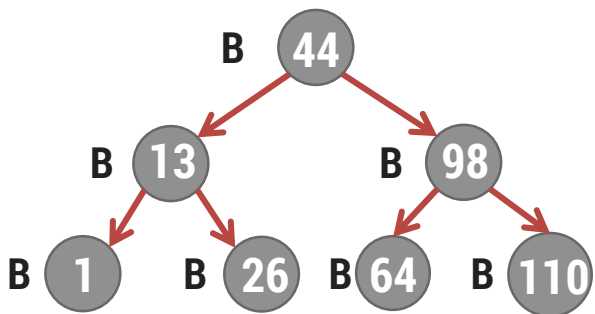
**Case 3: Right Left Rotation**  
Right Rotation of Right Child 110  
Followed By  
Left Rotation of Parent 64



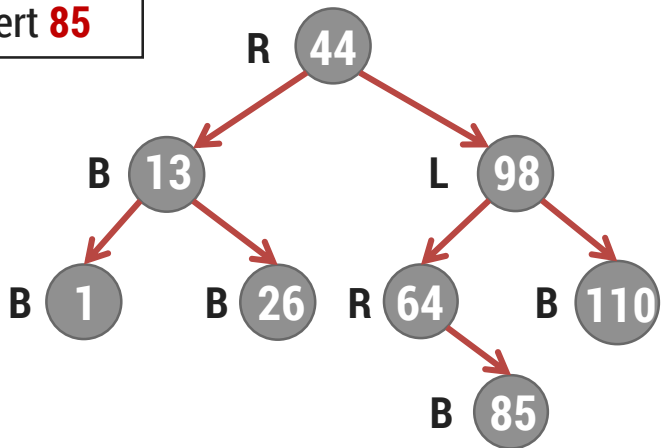
Right Rotation  
of Right Child **110**



Left Rotation  
of Parent **64**

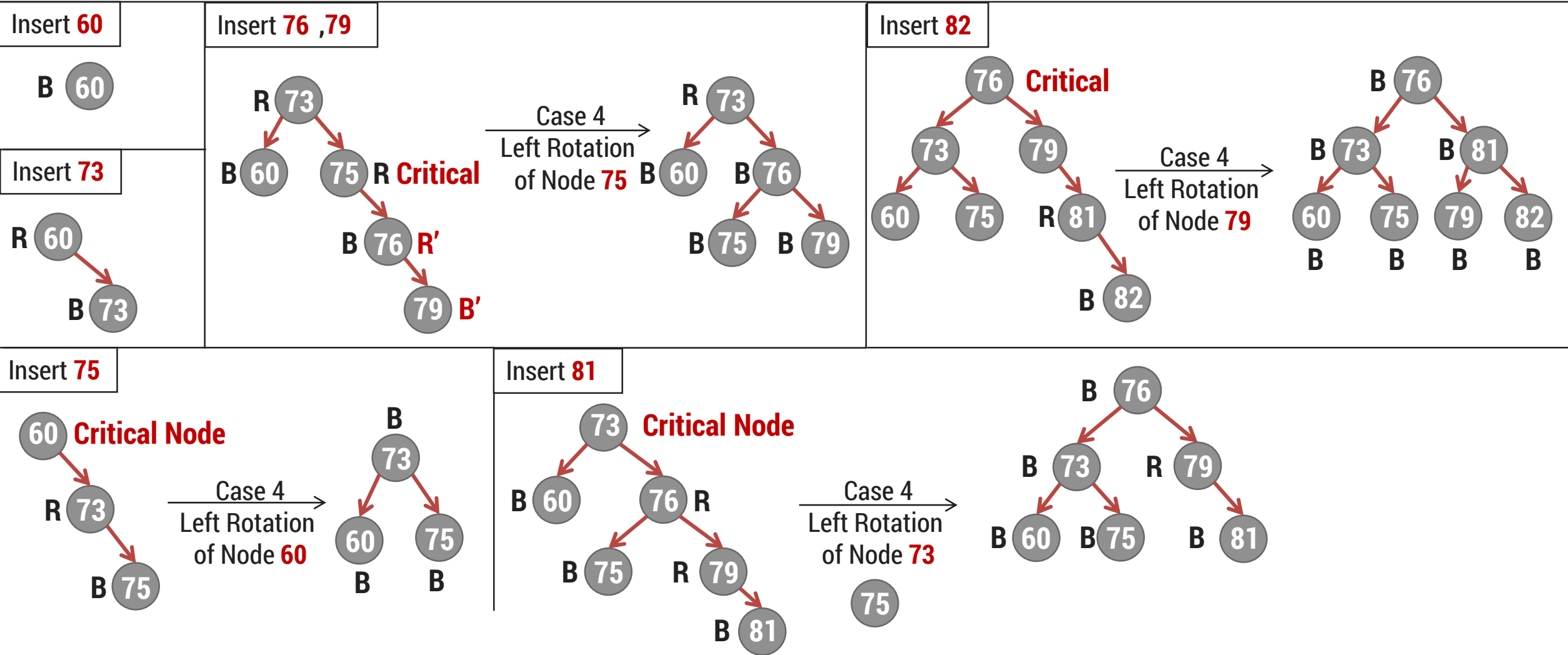


Insert **85**



# Construct AVL Search Tree

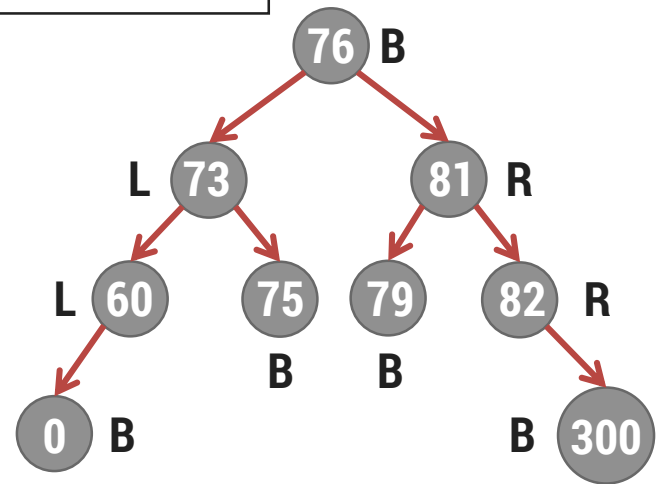
Construct AVL Search tree by inserting following elements in order of their occurrence **60,73,75,76,79,81,82,300,0,5,73**



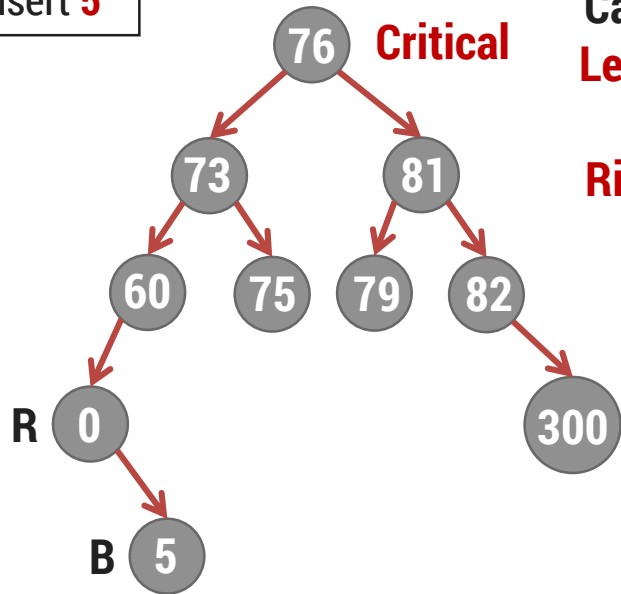
# Construct AVL Search Tree

Construct AVL Search tree by inserting following elements in order of their occurrence **60,73,75,76,79,81,82,300,0,5,73**

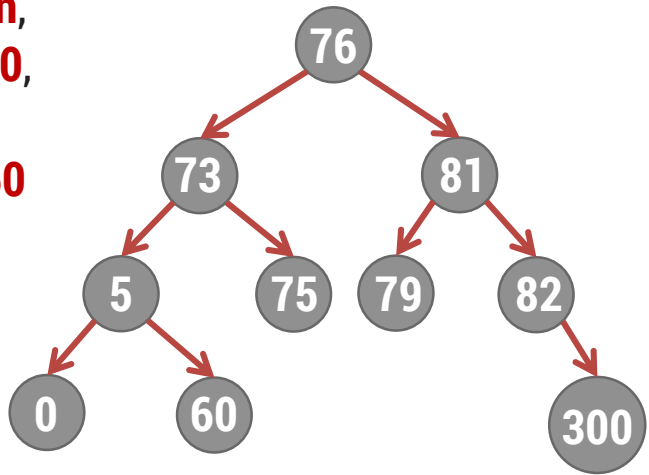
Insert **300** , **0**



Insert **5**



**Case 2: Left Right Rotation,**  
**Left Rotation** of Left Child **0**,  
Followed By  
**Right Rotation** of Parent **60**

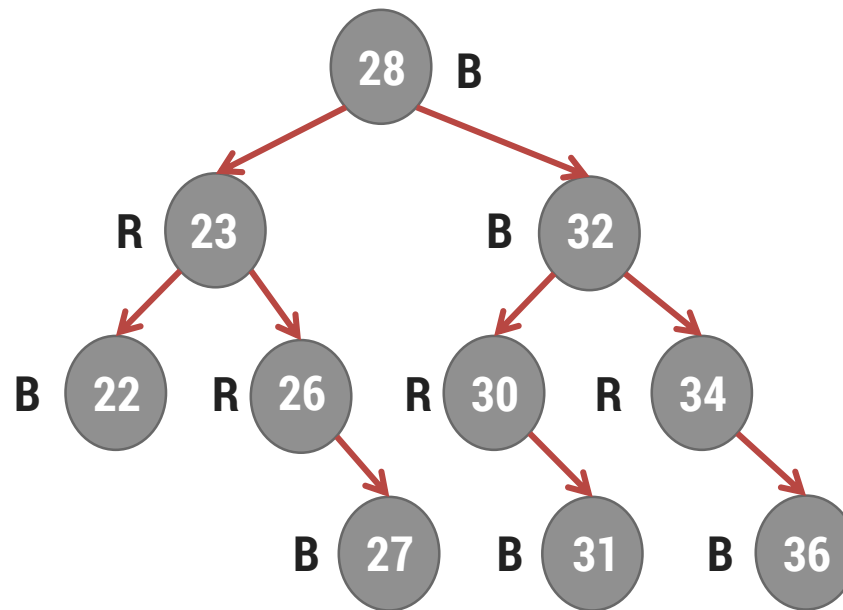


Insert **73**

Can not Insert **73** as duplicate key found

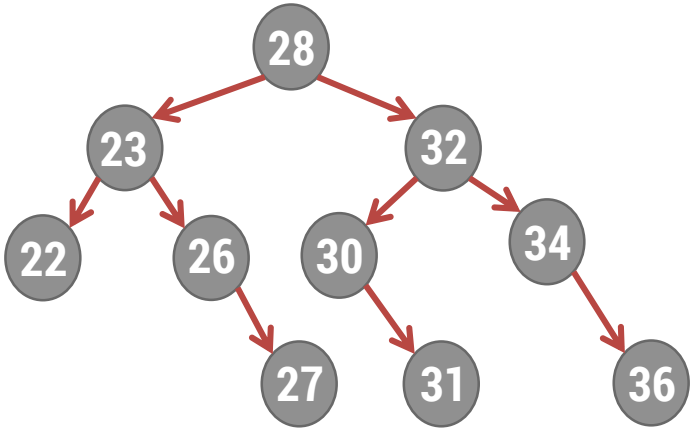
# Deleting node from AVL Tree

- ▶ If element to be deleted **does not have empty right sub-tree**, then **element** is **replaced** with its **In-Order successor** and its **In-Order successor** is **deleted** instead
- ▶ During **winding up phase**, we need to **revisit every node** on the **path** from the **point of deletion** up to the **root**, rebalance the tree if require



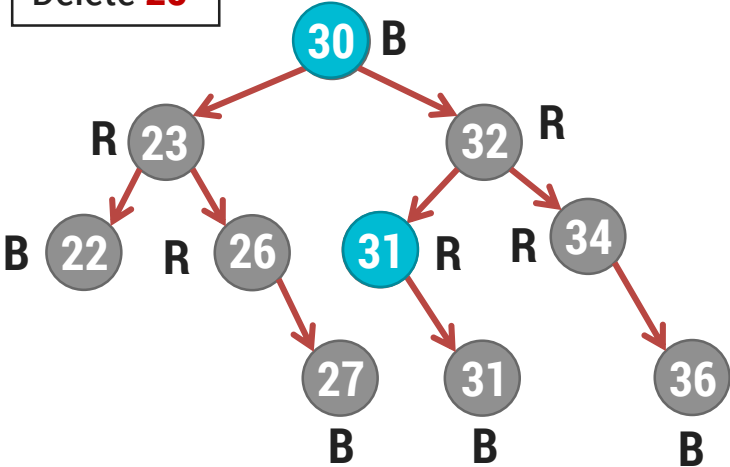


# Deleting node from AVL Tree

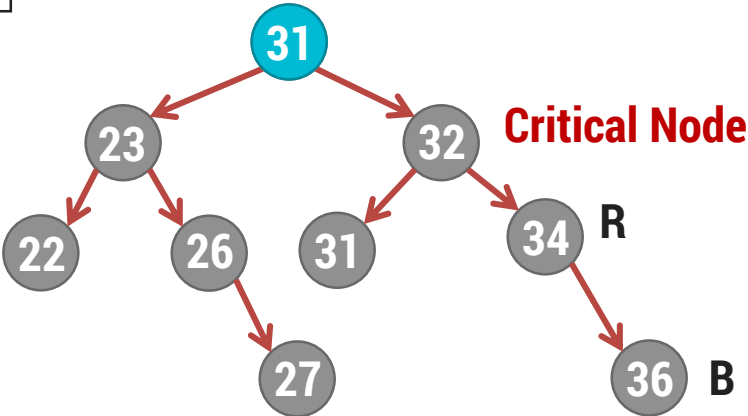


In-Order Traversal  
22, 23, 26, 27, 28, 30, 31, 32, 34, 36

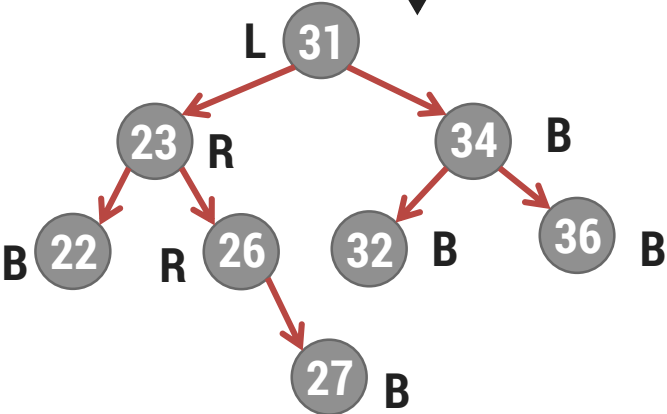
Delete 28



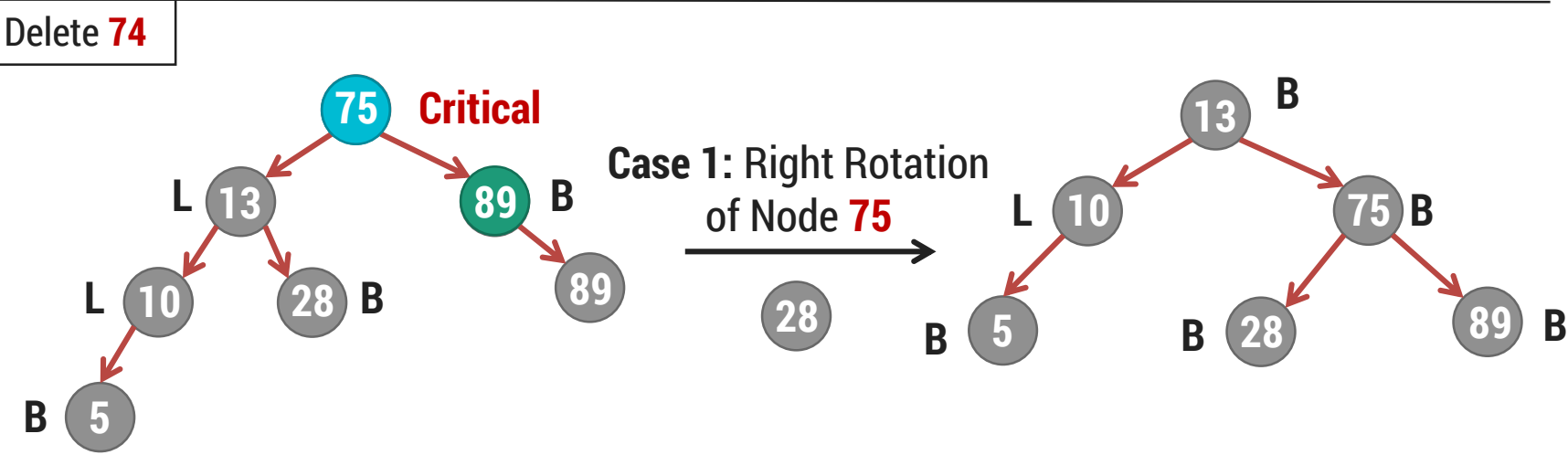
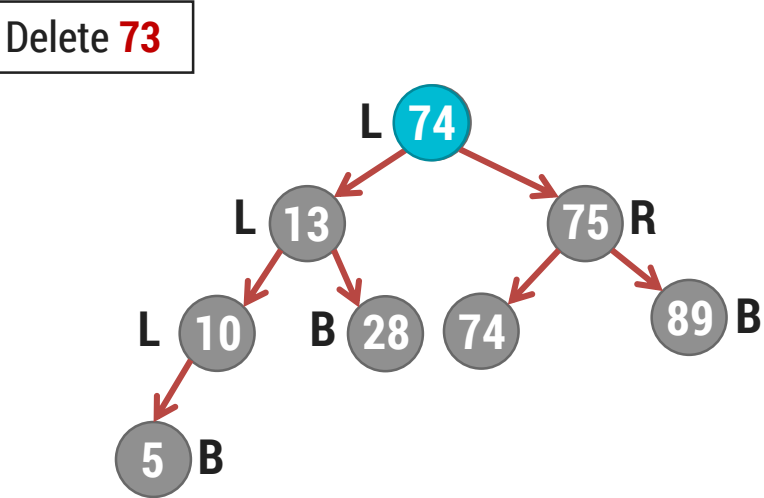
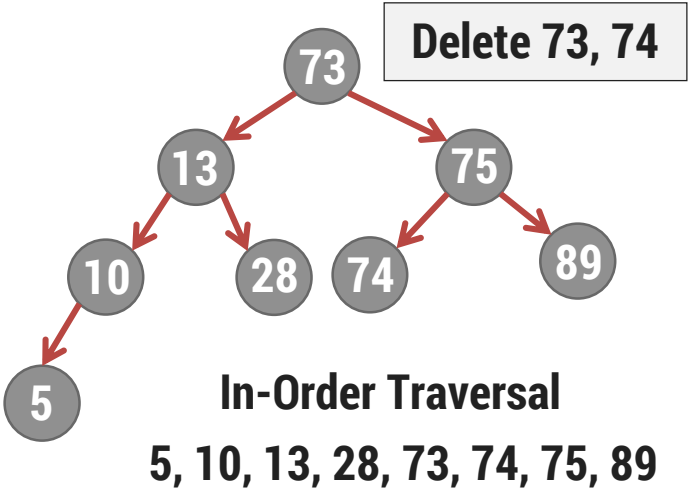
Delete 30



Case 4:  
Left Rotation of  
Node 32



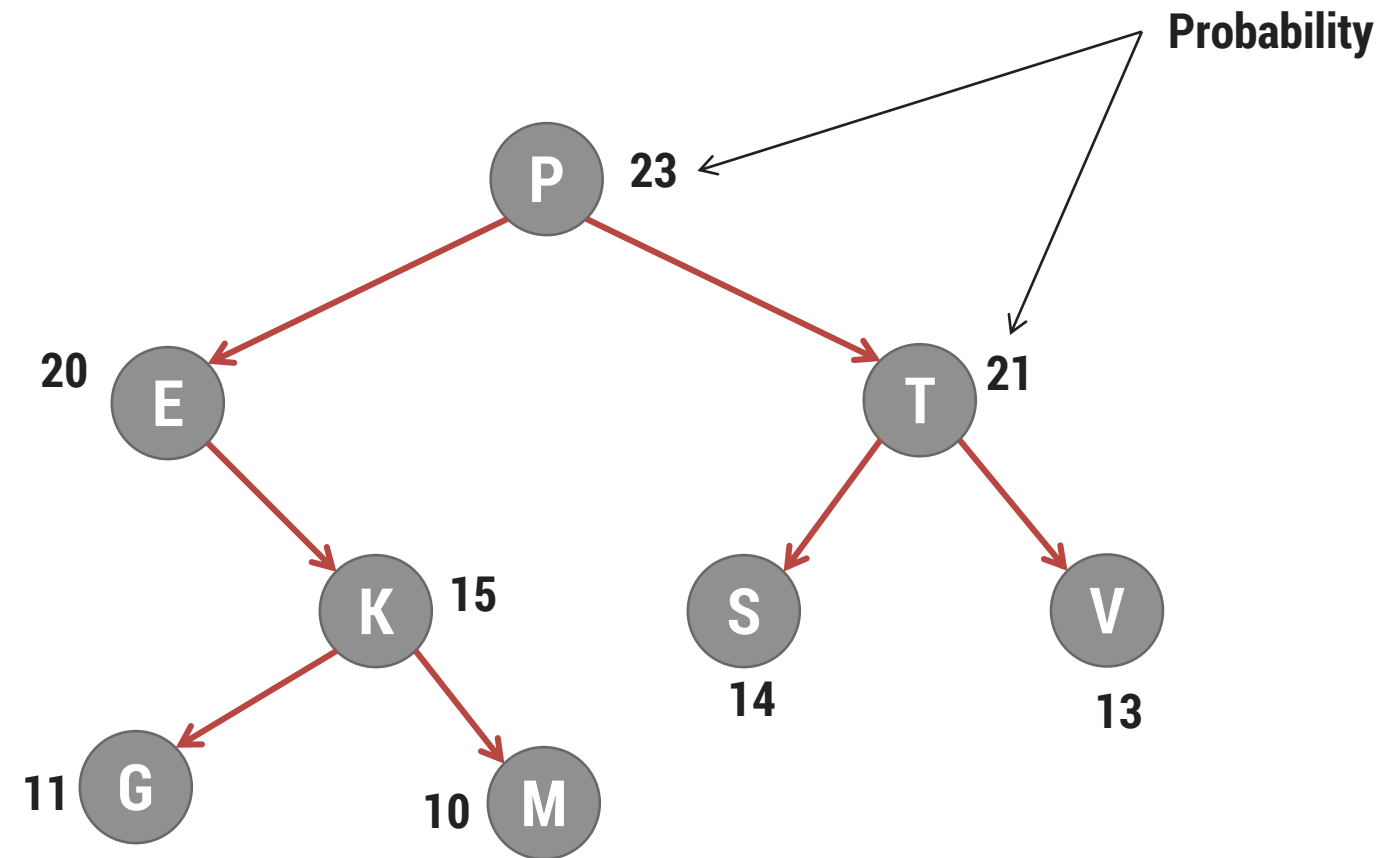
# Deleting node from AVL Tree



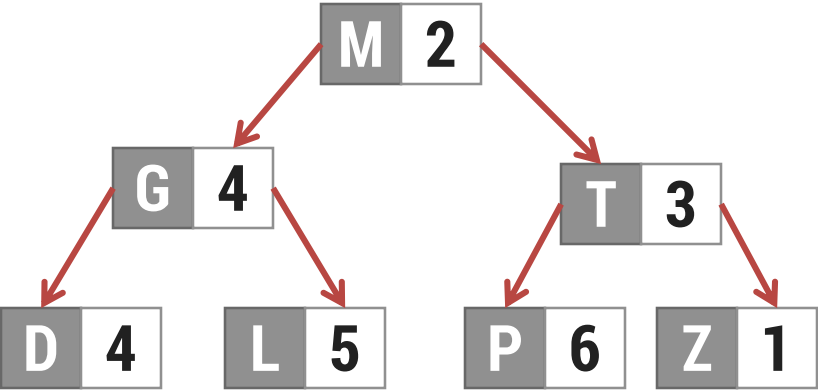
# Weight Balanced Tree

- ▶ In a **weight balanced tree**, the **nodes are arranged** on the **basis of** the knowledge available on the **probability for searching** each node
- ▶ The node with **highest probability** is placed at the **root** of the tree
- ▶ The **nodes** in the **left sub-tree** are **less in ranking** as well as **less in probability** then the root node
- ▶ The **nodes** in the **right sub-tree** are **higher in ranking** but **less in probability** then the root node
- ▶ Each node of such a Tree has an information field contains the value of the node and count number of times node has been visited

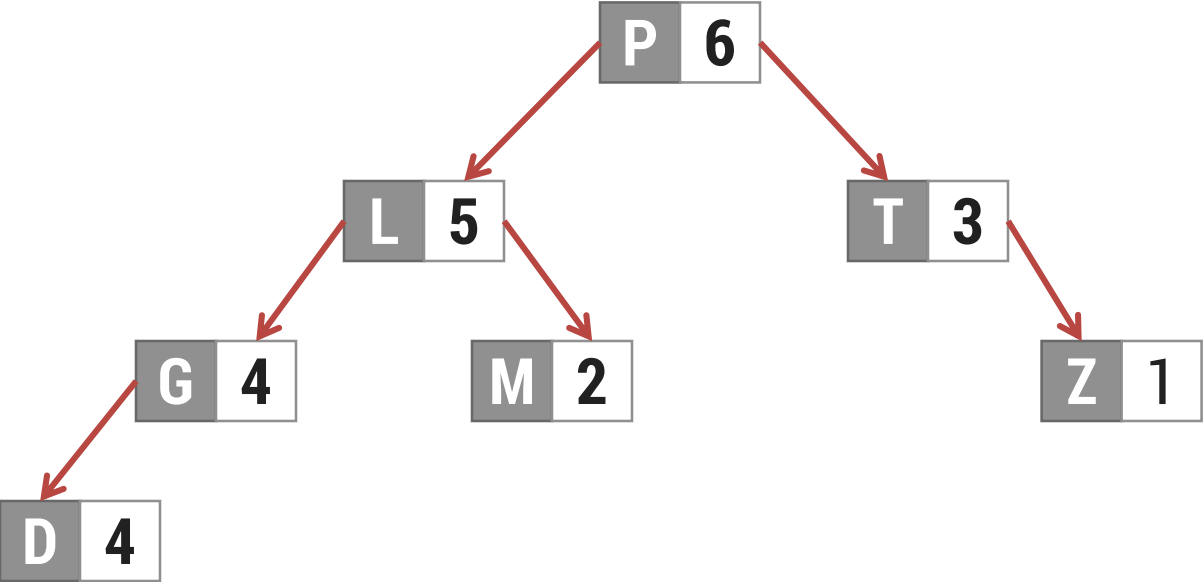
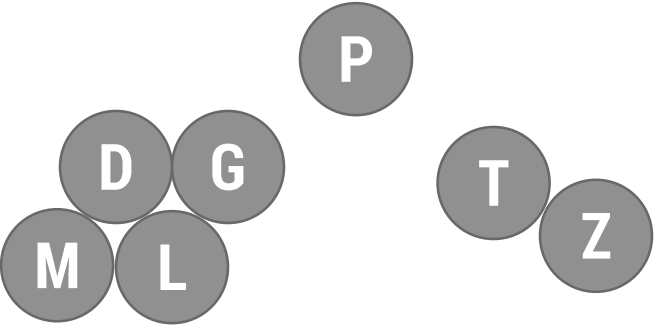
# Weight Balanced Tree



# Weight Balanced Tree

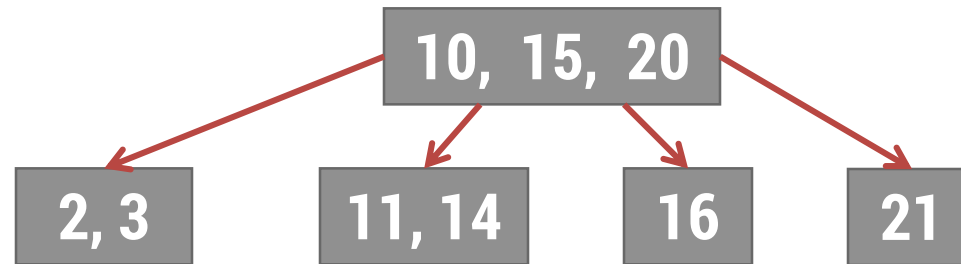


Ordered Tree



# Multiway Search Tree (B - Tree)

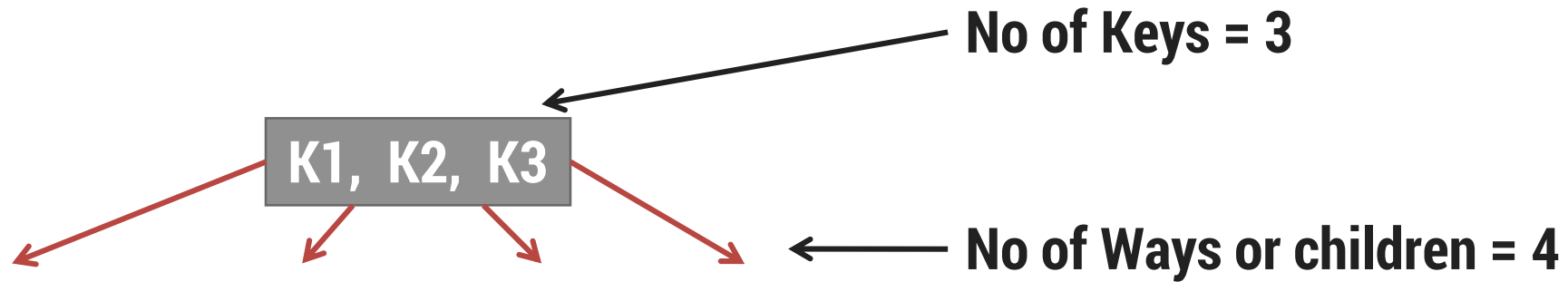
- ▶ The **nodes** in a **binary tree** like AVL tree **contains only one record**
- ▶ **AVL tree** is commonly **stored in primary memory**
- ▶ In **database applications** where **huge volume** of **data** is handled, the **search tree** can **not be accommodated** in **primary memory**
- ▶ **B-Trees** are primarily meant for **secondary storage**
- ▶ **B-Tree** is a **M-way tree** which can have **maximum of M Children**



4 – way Tree

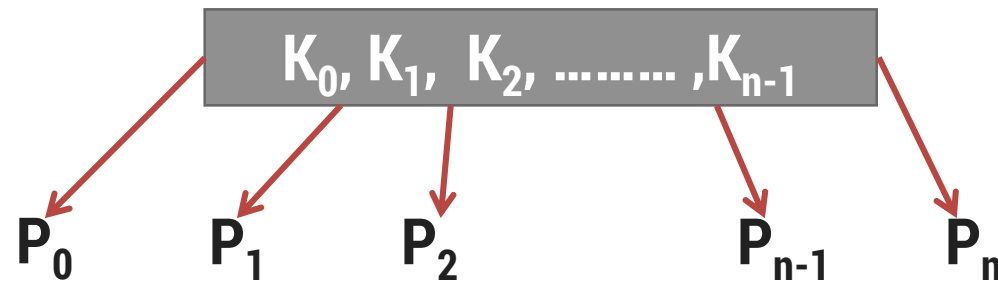
# Multiway Search Tree (B - Tree)

- ▶ An **M- way** tree contains **multiple keys** in a node
- ▶ This leads to **reduction** in overall **height** of the tree
- ▶ If a **node** of M-way tree **holds K keys** then it will have **k+1 children**



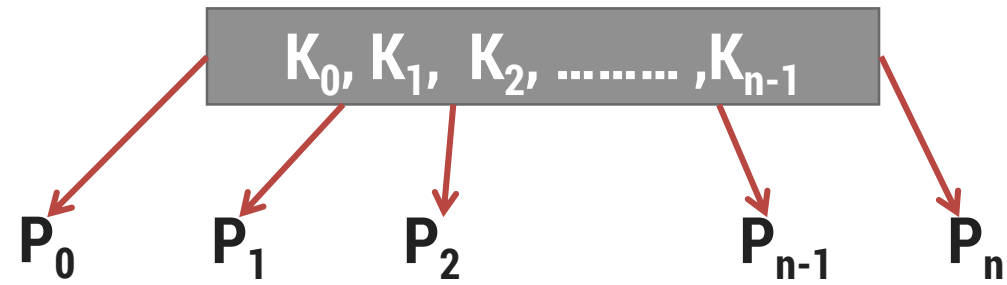
# Multiway Search Tree (B - Tree)

- A **tree** of **order M** is a **M-way** search tree with the following properties
1. The **Root** can have **1 to M-1 keys**
  2. All **nodes** (**except Root**) have  **$(M-1)/2$  to  $(M-1)$  keys**
  3. All **leaves** are at the **same level**
  4. If a node has '**t**' number of **children**, then it must have '**t-1**' keys
  5. **Keys** of the nodes are stored in **ascending order**



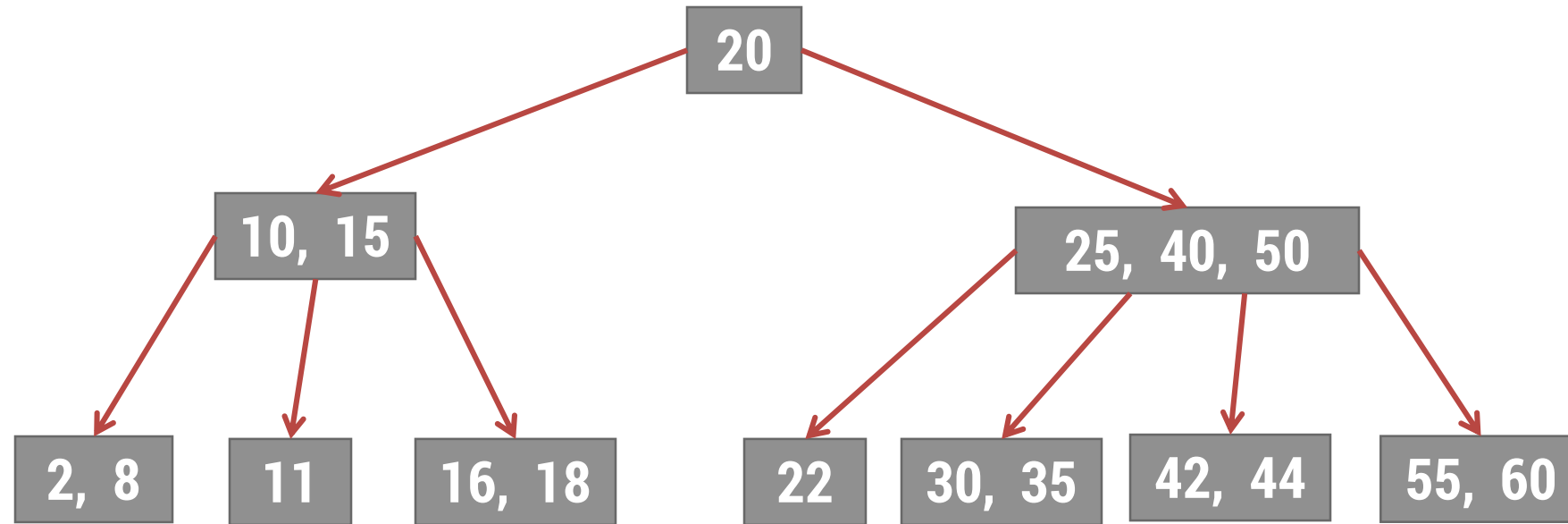


# Multiway Search Tree (B - Tree)



- ▶  $K_0, K_1, K_2, \dots, K_{n-1}$  are **keys** stored in the node
- ▶ **Sub-Trees** are pointed by  **$P_0, P_1, P_2, \dots, P_n$**  then
  - ➔  $K_0 \geq$  all keys of sub-tree  $P_0$
  - ➔  $K_1 \geq$  all keys of sub-tree  $P_1$
  - ➔ .....
  - ➔ .....
  - ➔  $K_{n-1} \geq$  all keys of sub-tree  $P_{n-1}$
  - ➔  $K_{n-1} <$  all keys of sub-tree  $P_n$

# Multiway Search Tree (B - Tree)

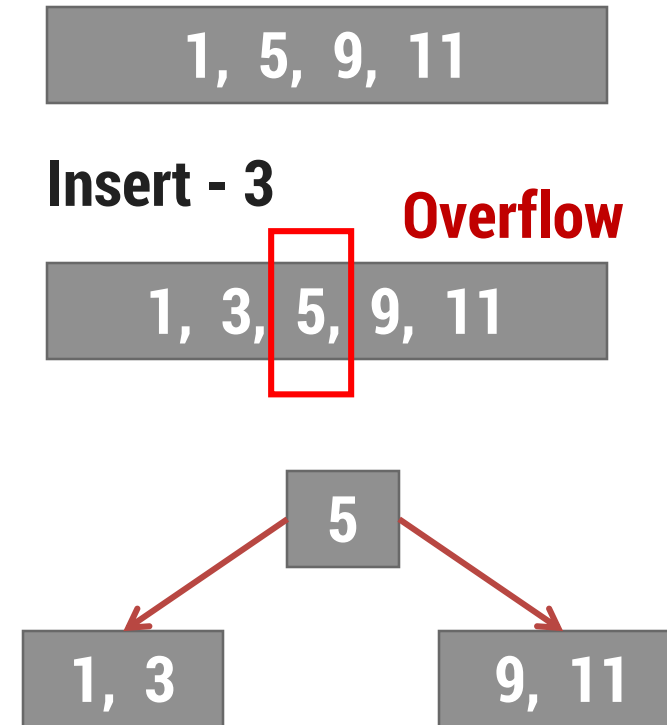
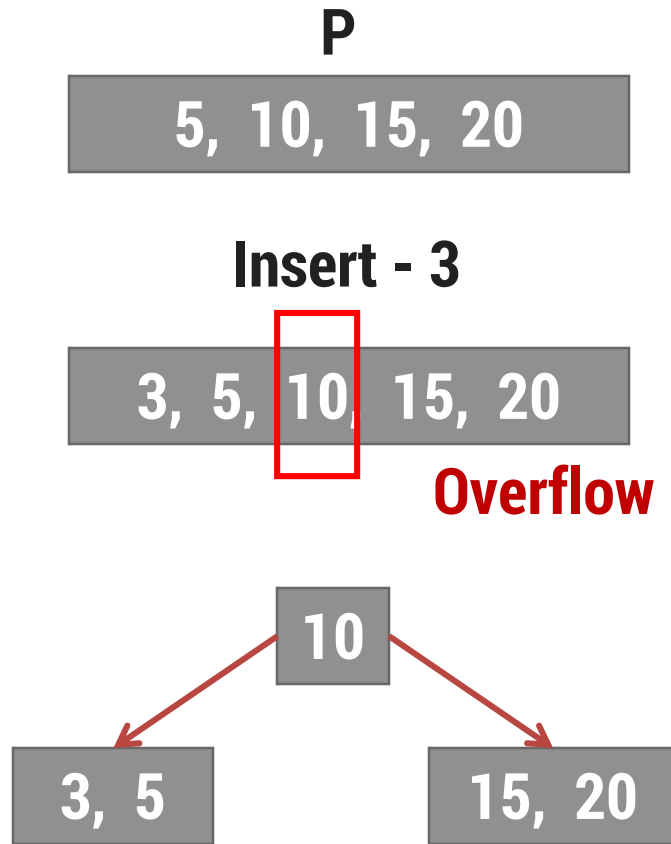


**B-Tree of Order 4 (4 way Tree)**

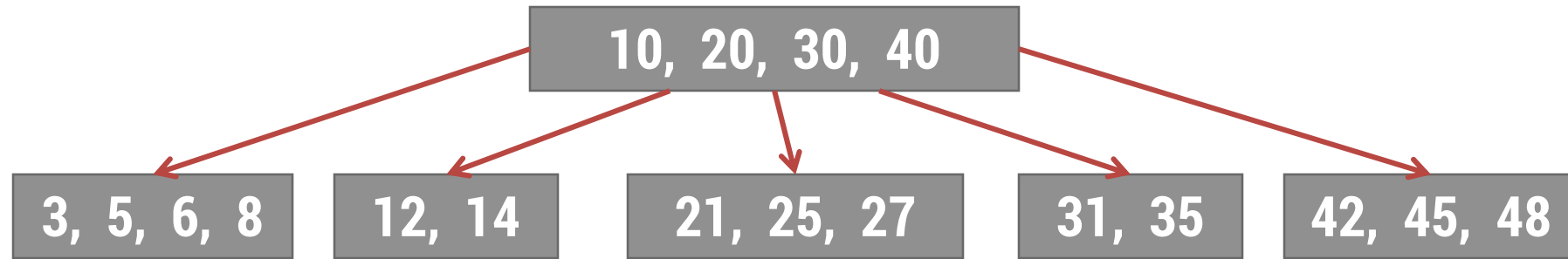
# Insertion of Key in B-Tree

1. If **Root** is **NULL**, **construct** a node and **insert key**
2. If **Root** is **NOT NULL**
  - I. Find the **correct leaf** node to which key should be added
  - II. If **leaf node has space** to accommodate key, it is **inserted** and **sorted**
  - III. If **leaf node does not have space** to accommodate key, we **split node** into two parts

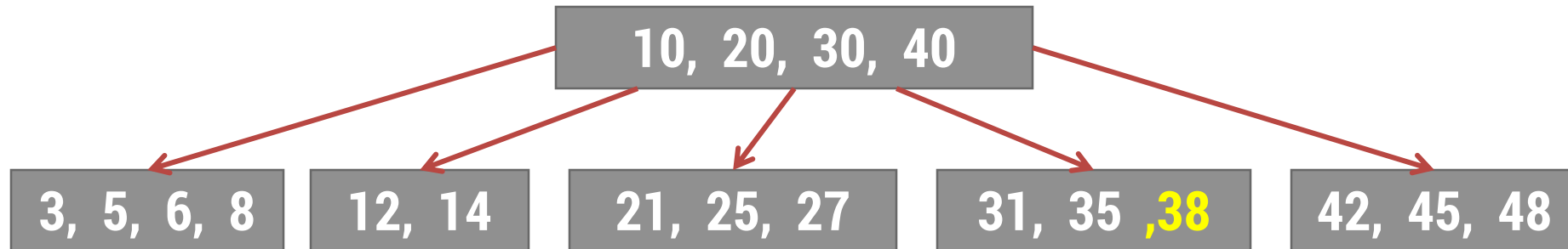
# Split Node (5 way Tree, max 4 Keys)



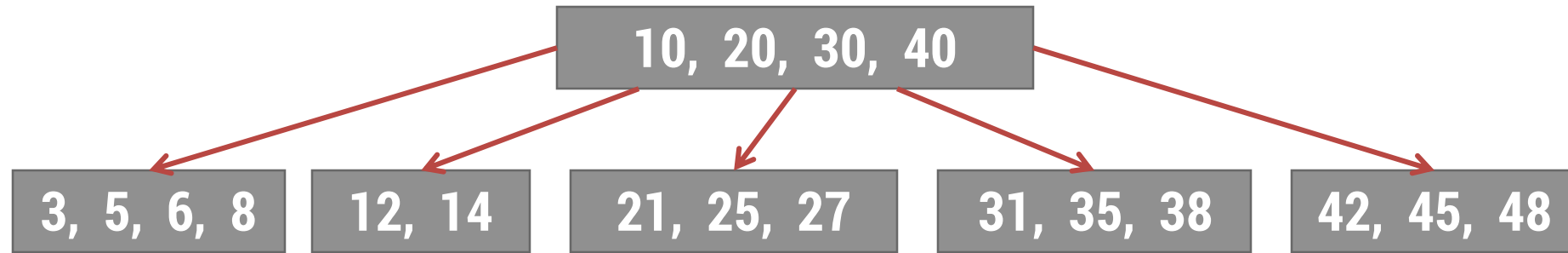
# Split Node (5 way Tree, max 4 Keys)



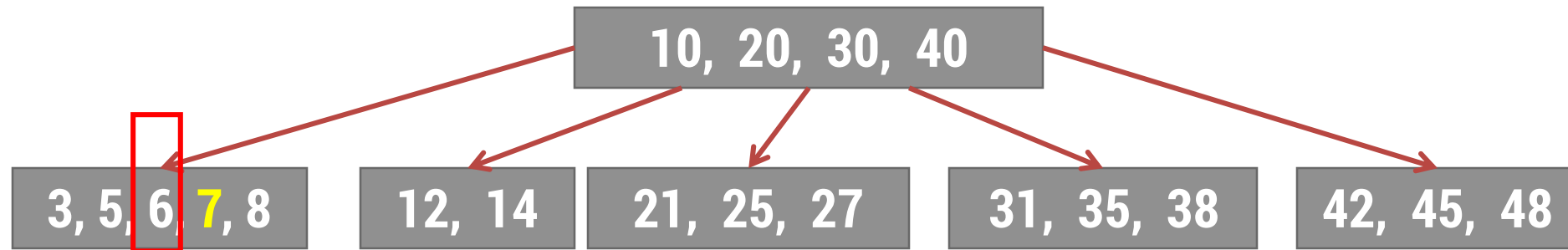
**Insert - 38**



# Split Node (5 way Tree, max 4 Keys)

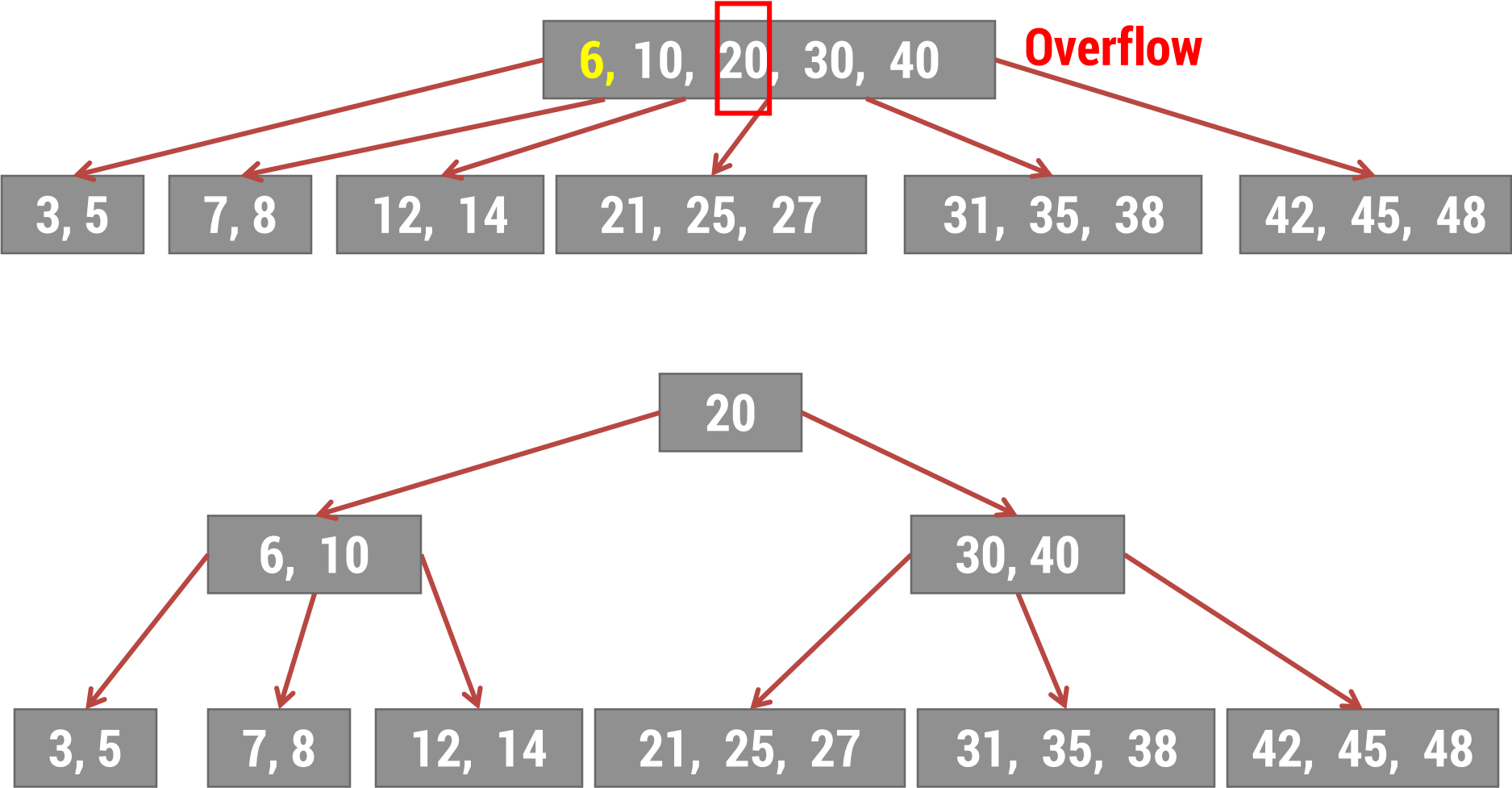


**Insert 7**



**Overflow**

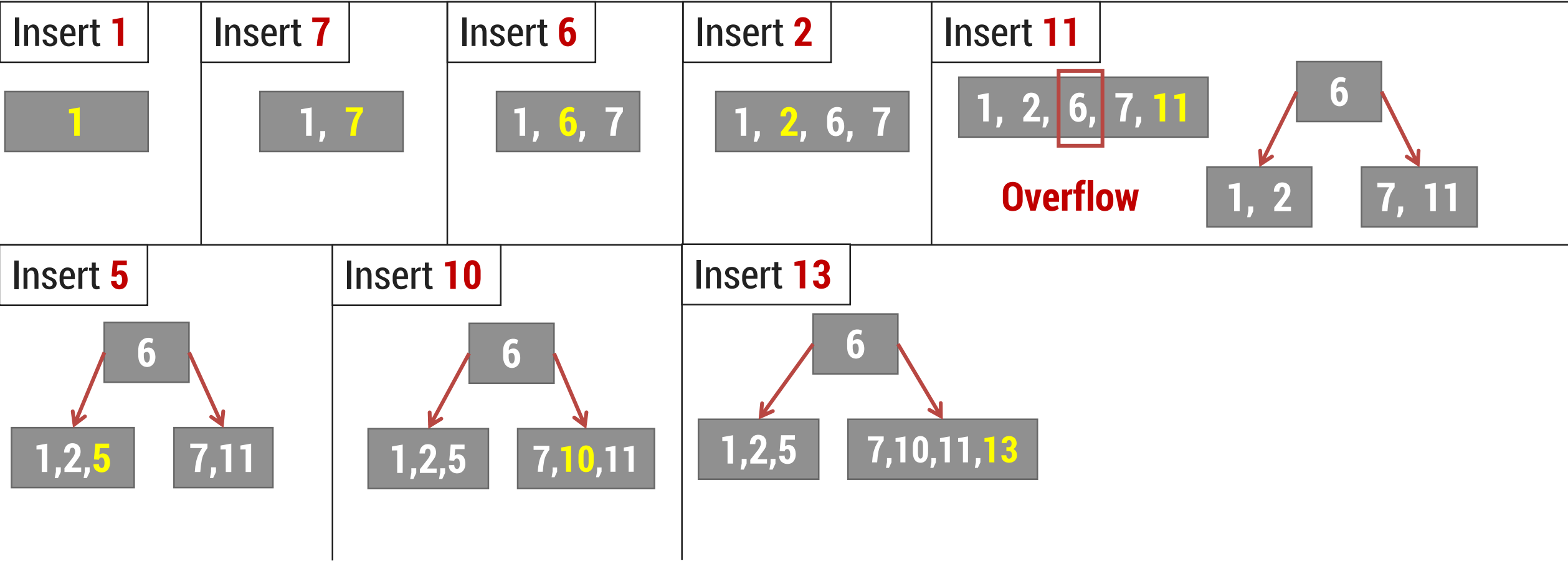
# Split Node (5 way Tree, max 4 Keys)



# Construct M-Way Tree

Construct **5 Order (5 Way)** Tree from following data  
1, 7, 6, 2, 11, 5, 10, 13, 12, 20, 16, 24, 3, 4, 18, 19, 14, 25

We are asked to **create 5 Order Tree (5 Way Tree) maximum 4 records** can be accommodated in a node

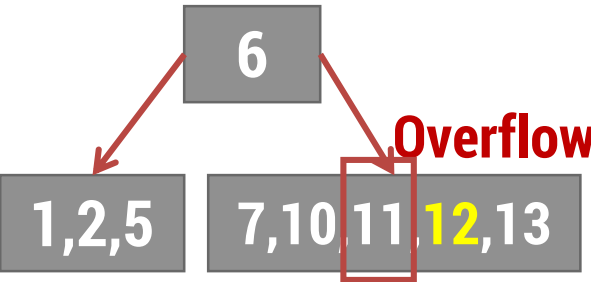




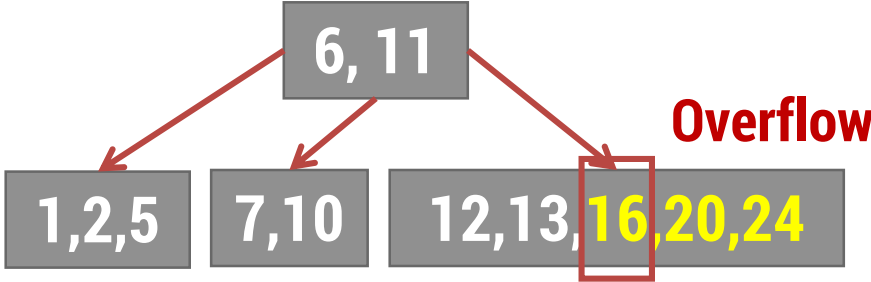
# Construct M-Way Tree

Construct **5 Order (5 Way)** Tree from following data  
1, 7, 6, 2, 11, 5, 10, 13, 12, 20, 16, 24, 3, 4, 18, 19, 14, 25

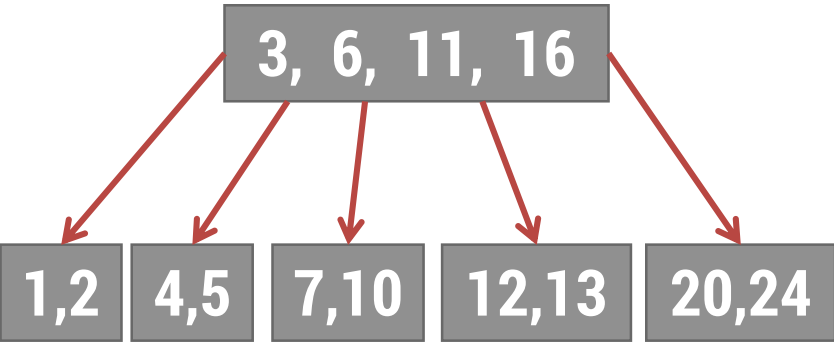
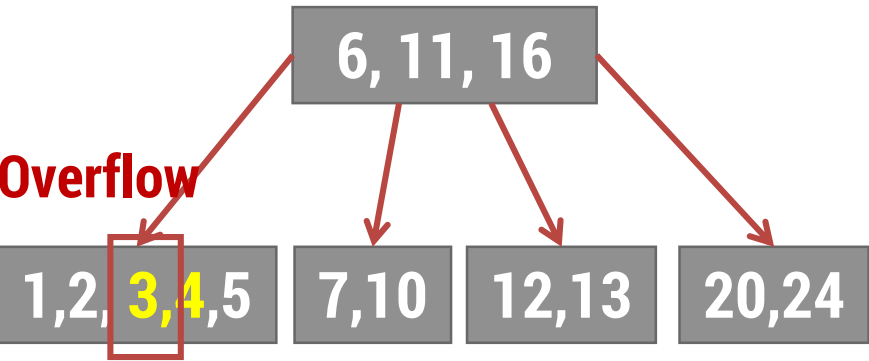
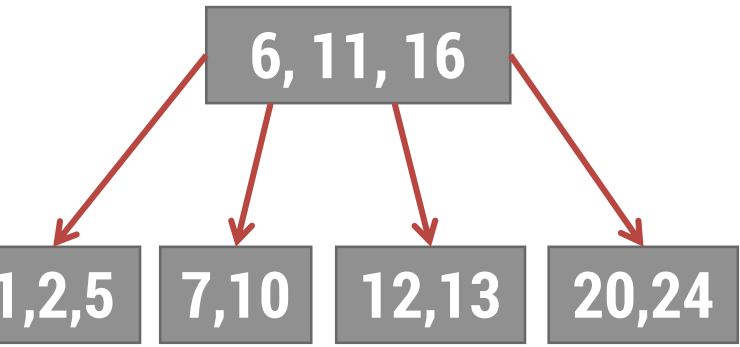
Insert **12**



Insert **20, 16, 24**



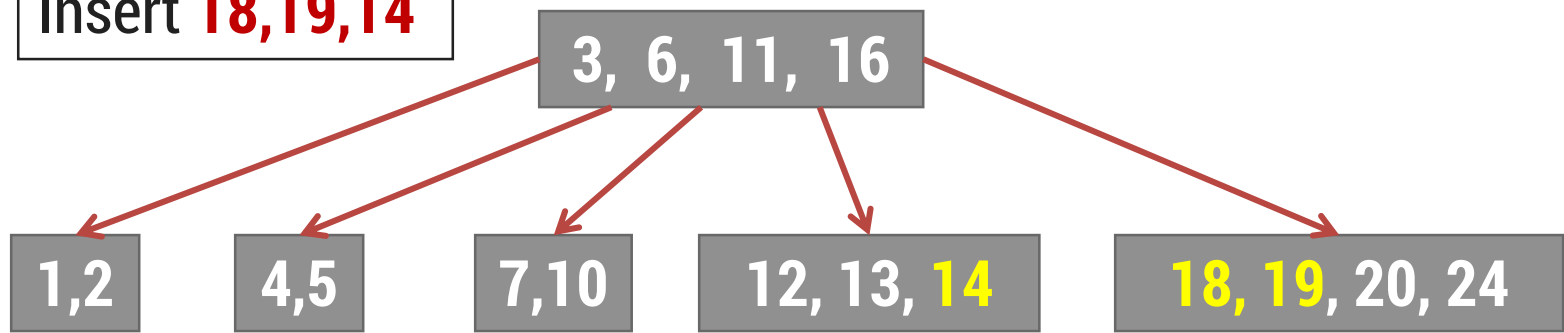
Insert **3,4**



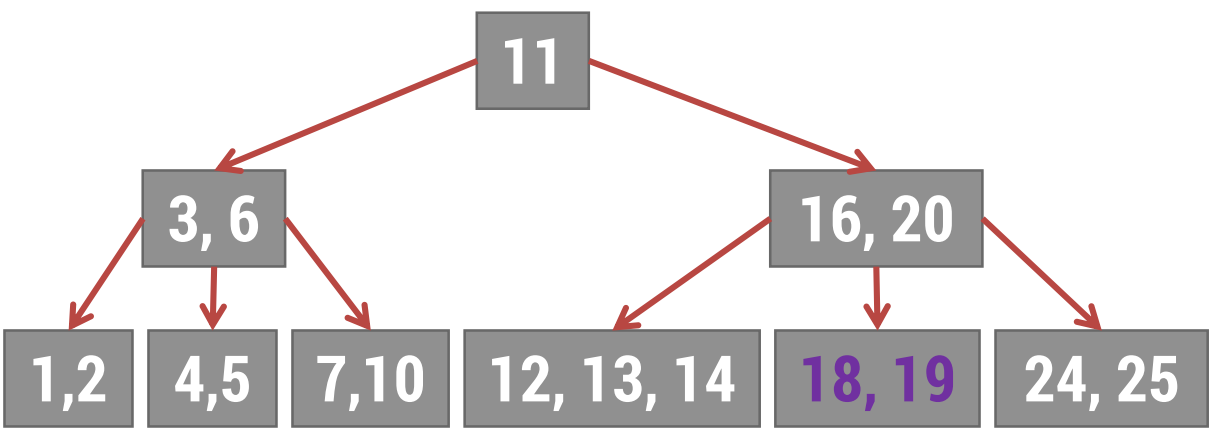
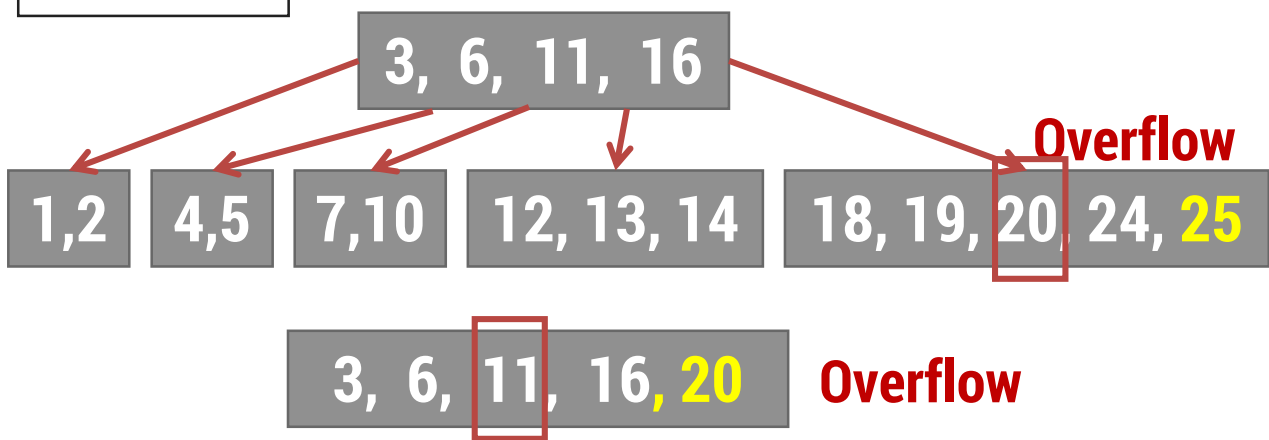
# Construct M-Way Tree

Construct **5 Order (5 Way)** Tree from following data  
1, 7, 6, 2, 11, 5, 10, 13, 12, 20, 16, 24, 3, 4, 18, 19, 14, 25

Insert **18,19,14**



Insert **25**



# Deletion - M-Way Tree

Deletion is also performed at the leaf nodes.

The node which is to be deleted can either be a leaf node or an internal node.

Following algorithm needs to be followed in order to delete a node from a B tree.

1. Locate the leaf node.
2. If there are more than  $m/2$  keys in the leaf node then delete the desired key from the node.
3. If the leaf node doesn't contain  $m/2$  keys then complete the keys by taking the element from right or left sibling.
  - 3.1 If the left sibling contains more than  $m/2$  elements then push its largest element up to its parent and move the intervening element down to the node where the key is deleted.
  - 3.2 If the right sibling contains more than  $m/2$  elements then push its smallest element up to the parent and move intervening element down to the node where the key is deleted.

# Deletion - M-Way Tree

4. If neither of the sibling contain more than  $m/2$  elements then create a new leaf node by joining two leaf nodes and the intervening element of the parent node.
5. If parent is left with less than  $m/2$  nodes then, apply the above process on the parent too.

If the the node which is to be deleted is an internal node, then replace the node with its in-order successor or predecessor. Since, successor or predecessor will always be on the leaf node hence, the process will be similar as the node is being deleted from the leaf node

# Deletion in B-Tree

For deletion in b tree we wish to remove from a leaf. T

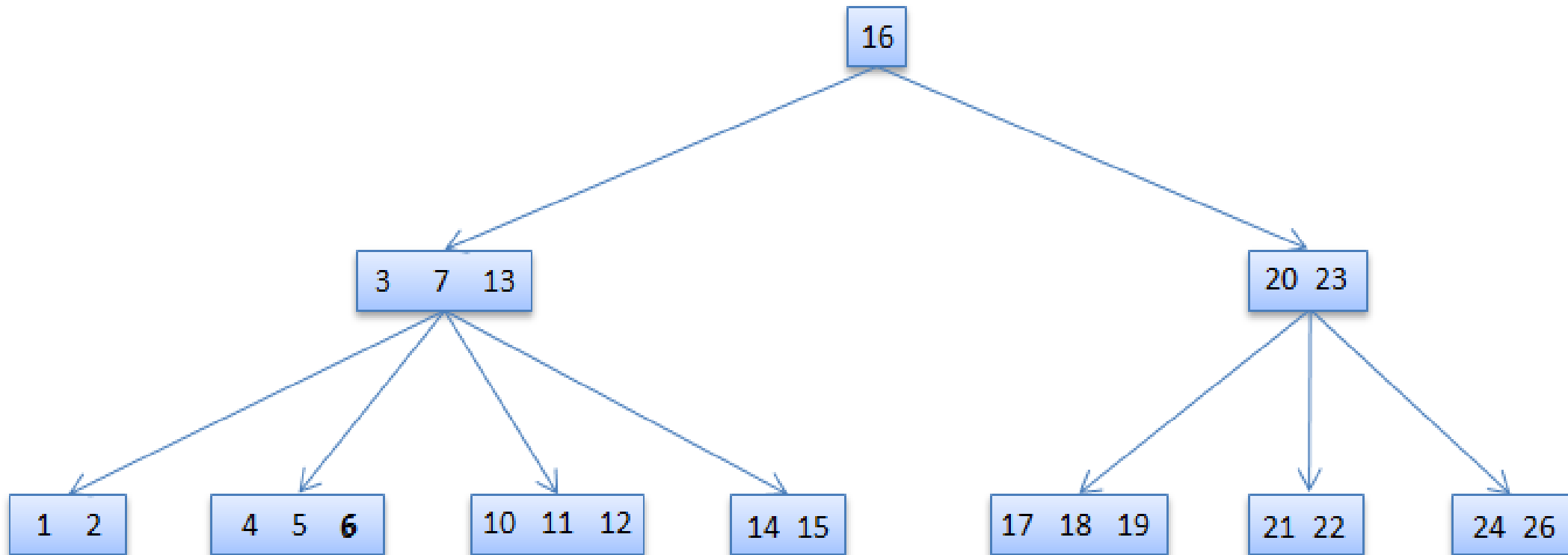
There are three possible case for deletion in b tree.

Let  $k$  be the key to be deleted,  $x$  the node containing the key.

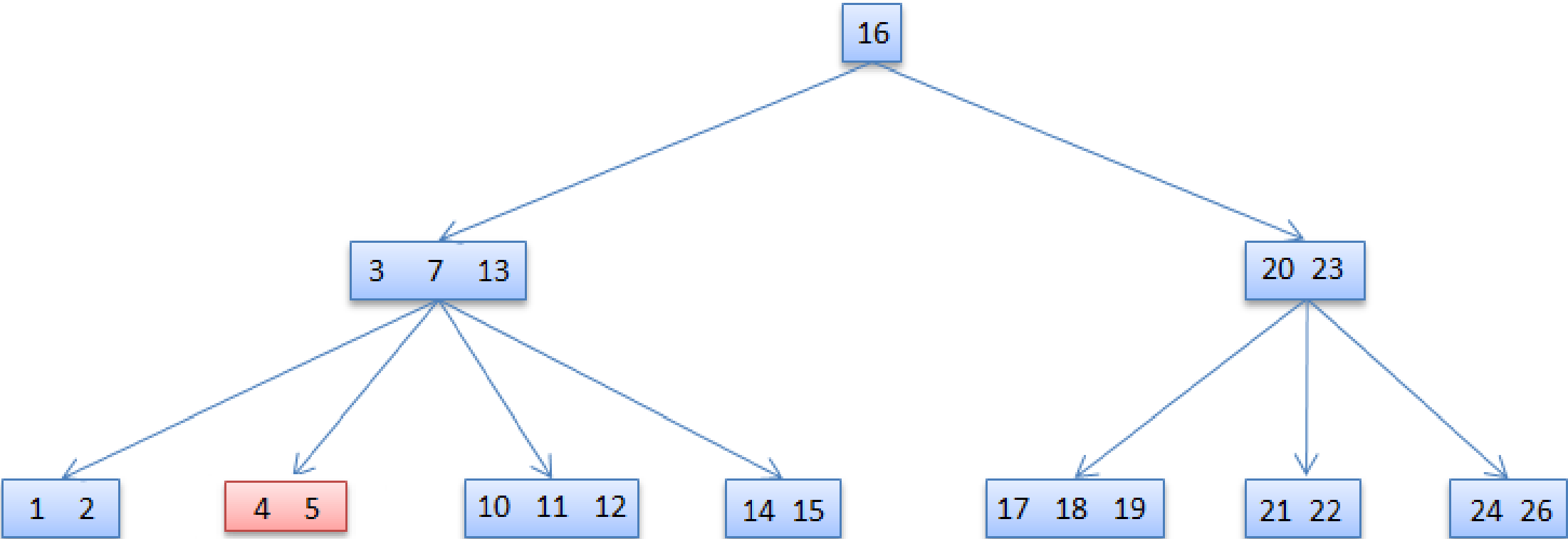
Then the cases are:

# Case-I

If the key is already in a leaf node, and removing it doesn't cause that leaf node to have too few keys, then simply remove the key to be deleted. key  $k$  is in node  $x$  and  $x$  is a leaf, simply delete  $k$  from  $x$ .



6 deleted



## Case-II

If key  $k$  is in node  $x$  and  $x$  is an internal node, there are three cases to consider:

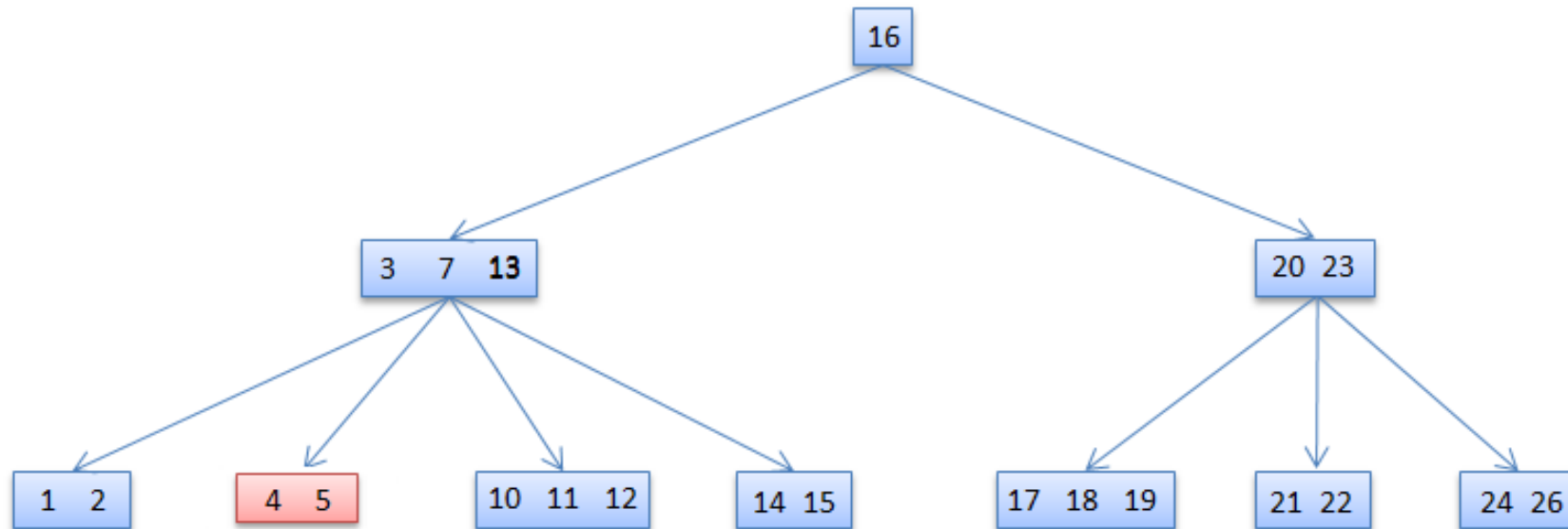
### Case-II-a

If the child  $y$  that precedes  $k$  in node  $x$  has at least  $t$  keys (more than the minimum), then find the predecessor key  $k'$  in the subtree rooted at  $y$ .  
Recursively delete  $k'$  and replace  $k$  with  $k'$  in  $x$

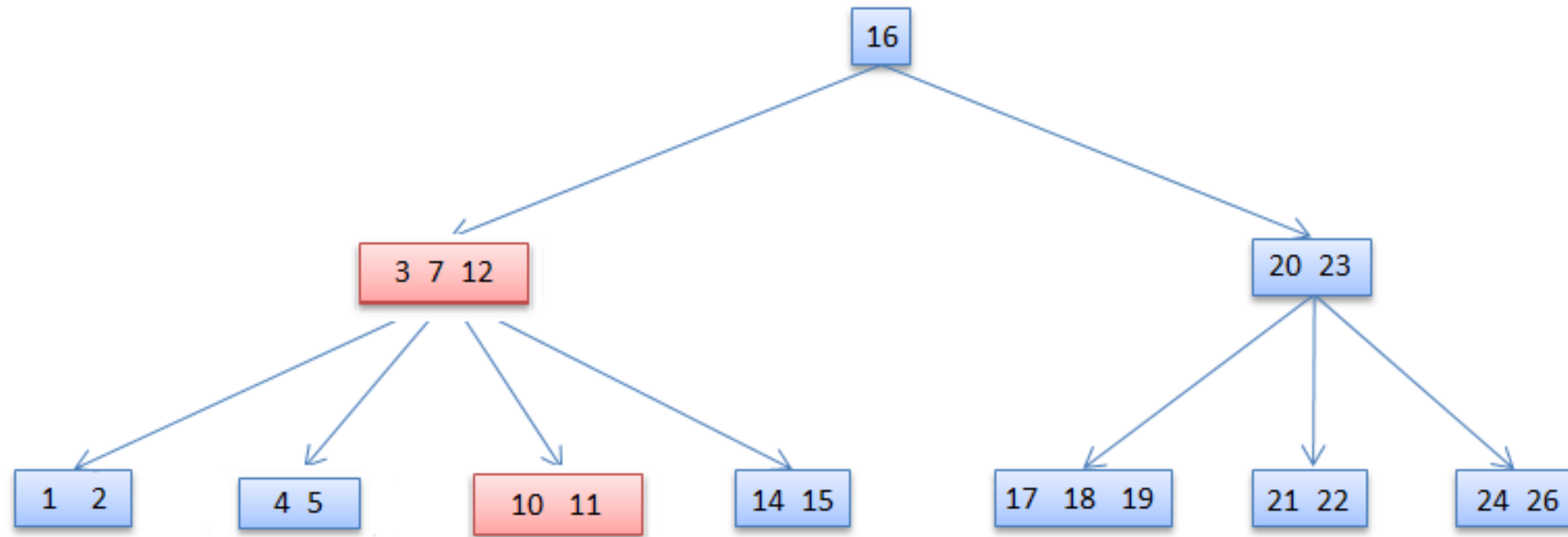


## Case-II-b

Symmetrically, if the child  $z$  that follows  $k$  in node  $x$  has at least  $t$  keys, find the successor  $k'$  and delete and replace as before. Note that finding  $k'$  and deleting it can be performed in a single downward pass.



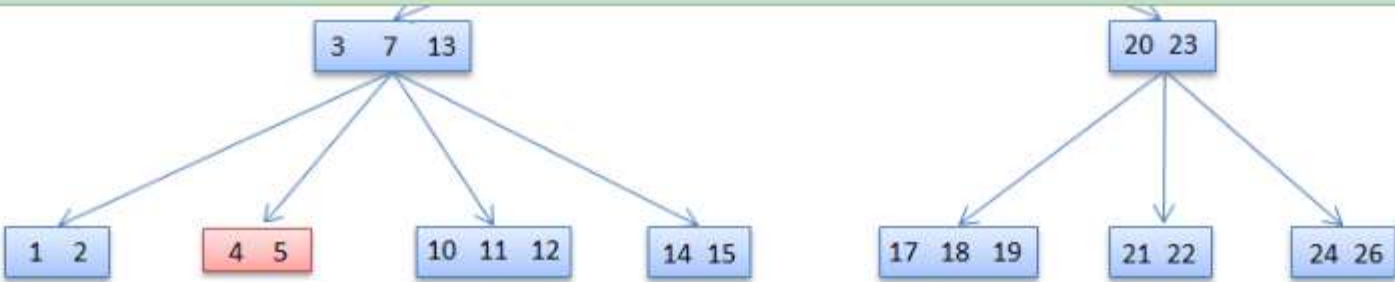
13 deleted



Removal Services,  
Find Best Deals

TempoSearch®

Open >



### Case-II

If key  $k$  is in node  $x$  and  $x$  is an internal node, there are three cases to consider:

#### Case-II-a

If the child  $y$  that precedes  $k$  in node  $x$  has at least  $t$  keys (more than the minimum), then find the predecessor key  $k'$  in the subtree rooted at  $y$ . Recursively delete  $k'$  and replace  $k$  with  $k'$  in  $x$

#### Case-II-b



Find Tree  
Removers in  
My Area



Browse All Tree  
Removers in My Area  
Find Best Deals



## Time and Space complexity of Binary Search Tree (BST)

OPERATION	WORST CASE	AVERAGE CASE	BEST CASE	SPACE
Search	$O(N)$	$O(\log N)$	$O(1)$	$O(N)$
Insert	$O(N)$	$O(\log N)$	$O(1)$	$O(N)$
Delete	$O(N)$	$O(\log N)$	$O(N)$	$O(N)$

# Time & Space Complexity of AVL Tree operations

OPERATION	BEST CASE	AVERAGE CASE	WORST CASE
Insert	$O(\log n)$	$O(\log n)$	$O(\log n)$
Delete	$O(\log n)$	$O(\log n)$	$O(\log n)$
Search	$O(1)$	$O(\log n)$	$O(\log n)$
Traversal	$O(\log n)$	$O(\log n)$	$O(\log n)$

# B-Tree : Searching and Insertion

## Complexity

- Worst case search time complexity:  $\Theta(\log n)$
- Average case search time complexity:  $\Theta(\log n)$
- Best case search time complexity:  $\Theta(\log n)$
- Average case Space complexity:  $\Theta(n)$
- Worst case Space complexity:  $\Theta(n)$









# **Non-Linear Data Structure**

## **Tree Part-2**

# Red - Black Tree

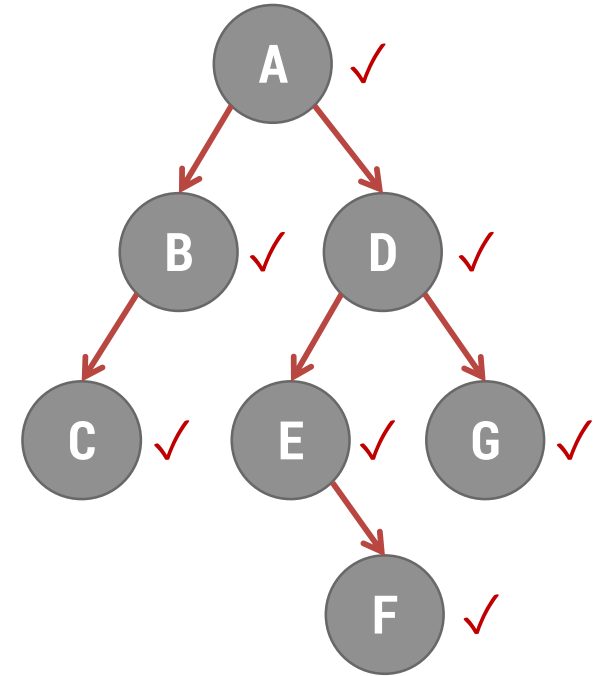
- ▶ We can d
- ▶ **Red Black Tree is a Binary Search Tree in which every node is colored either RED or BLACK.**

Define a Red Black Tree as follows...

Red Black Tree is a Binary Search Tree in which every node is colored either RED or BLACK.

# Preorder Traversal

- ▶ Preorder traversal of a binary tree is defined as follow
  1. **Process** the **root node**
  2. **Traverse** the **left subtree** in preorder
  3. **Traverse** the **right subtree** in preorder
- ▶ If particular **subtree is empty** (i.e., node has no left or right descendant) the traversal is performed by **doing nothing**.
- ▶ In other words, a **null subtree** is **considered to be fully traversed** when it is encountered.



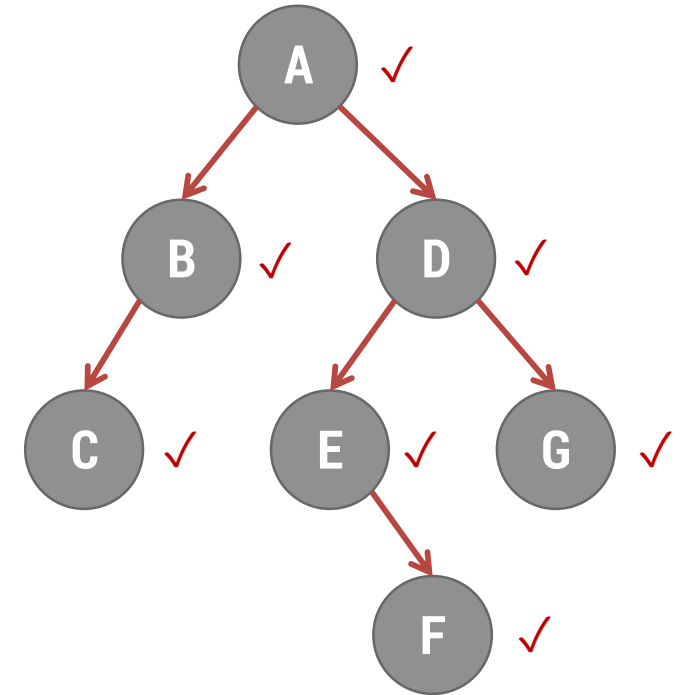
Preorder traversal of a given tree as

**A B C D E F G**

# Inorder Traversal

► Inorder traversal of a binary tree is defined as follow

1. **Traverse** the **left subtree** in Inorder
2. **Process** the **root node**
3. **Traverse** the **right subtree** in Inorder



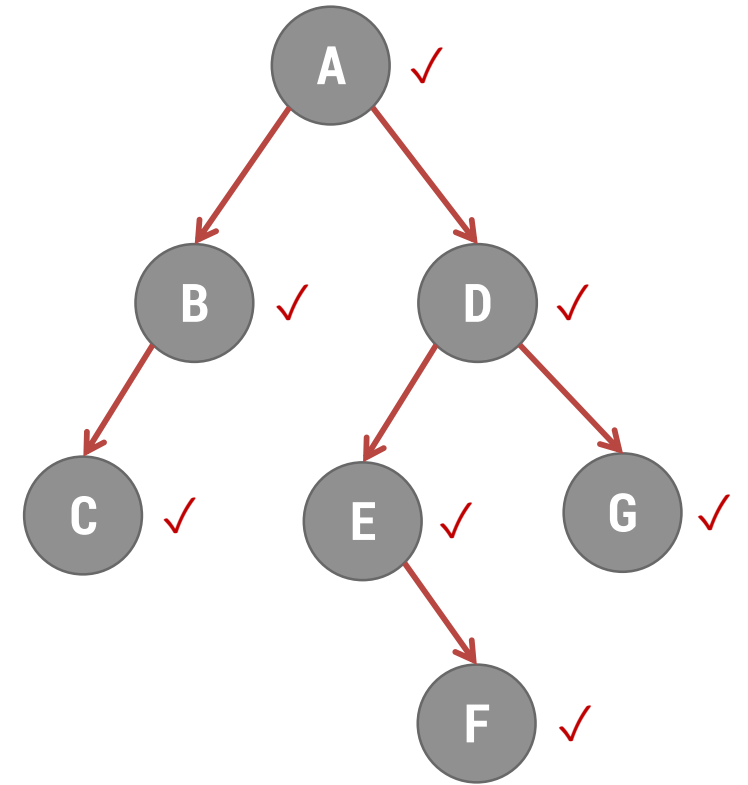
Inorder traversal of a given tree as

**C B A E F D G**

# Postorder Traversal

► Postorder traversal of a binary tree is defined as follow

1. **Traverse** the **left subtree** in Postorder
2. **Traverse** the **right subtree** in Postorder
3. **Process** the **root node**



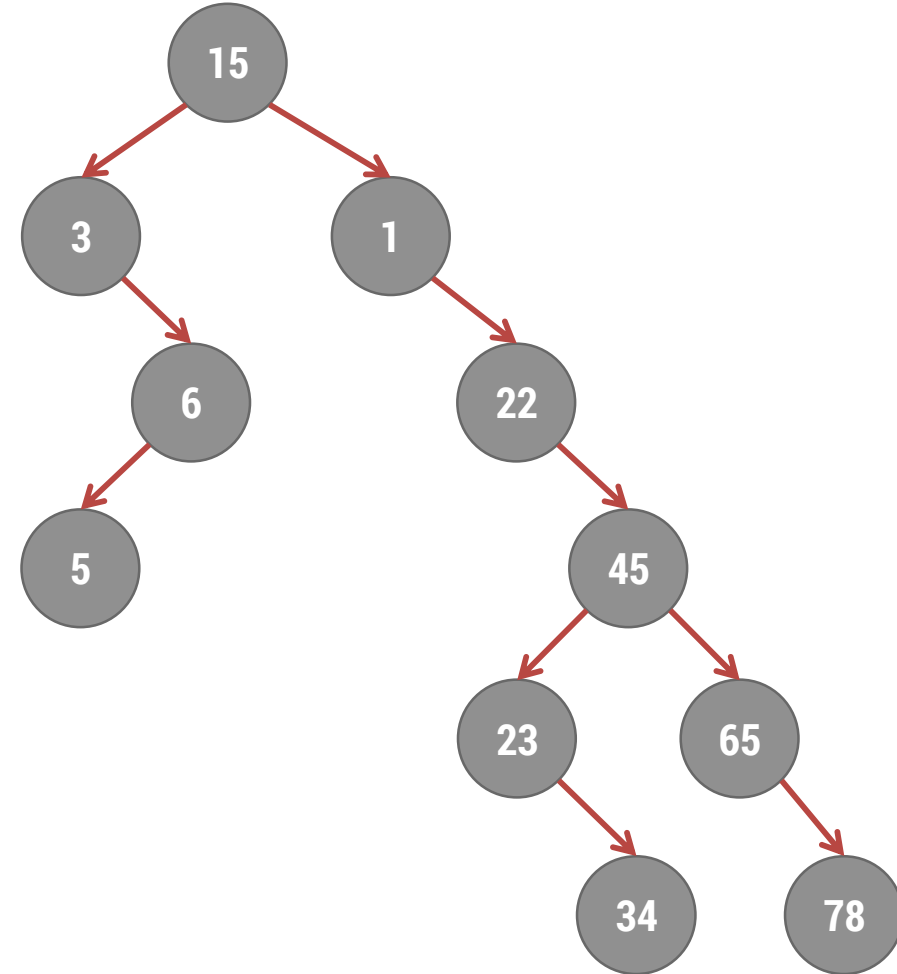
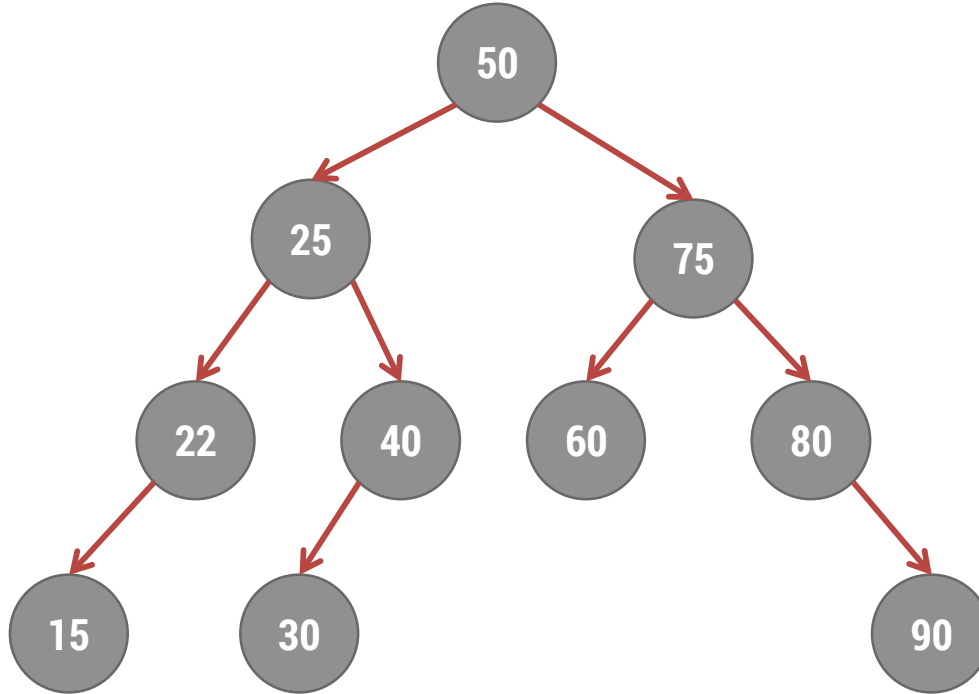
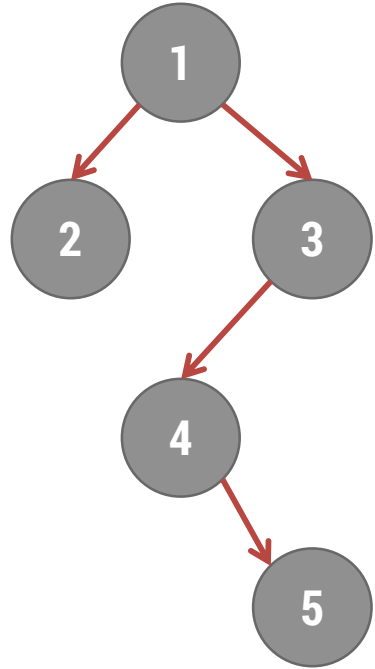
Postorder traversal of a given tree as

**C B F E G D A**

# Converse Traversal

- ▶ If we *interchange left and right words in the preceding definitions*, we obtain three new traversal orders which are called
  - ↳ **Converse Preorder** Traversal: A D G E F B C
  - ↳ **Converse Inorder** Traversal: G D F E A B C
  - ↳ **Converse Postorder** Traversal: G F E D C B A

# Write Pre/In/Post Order Traversal

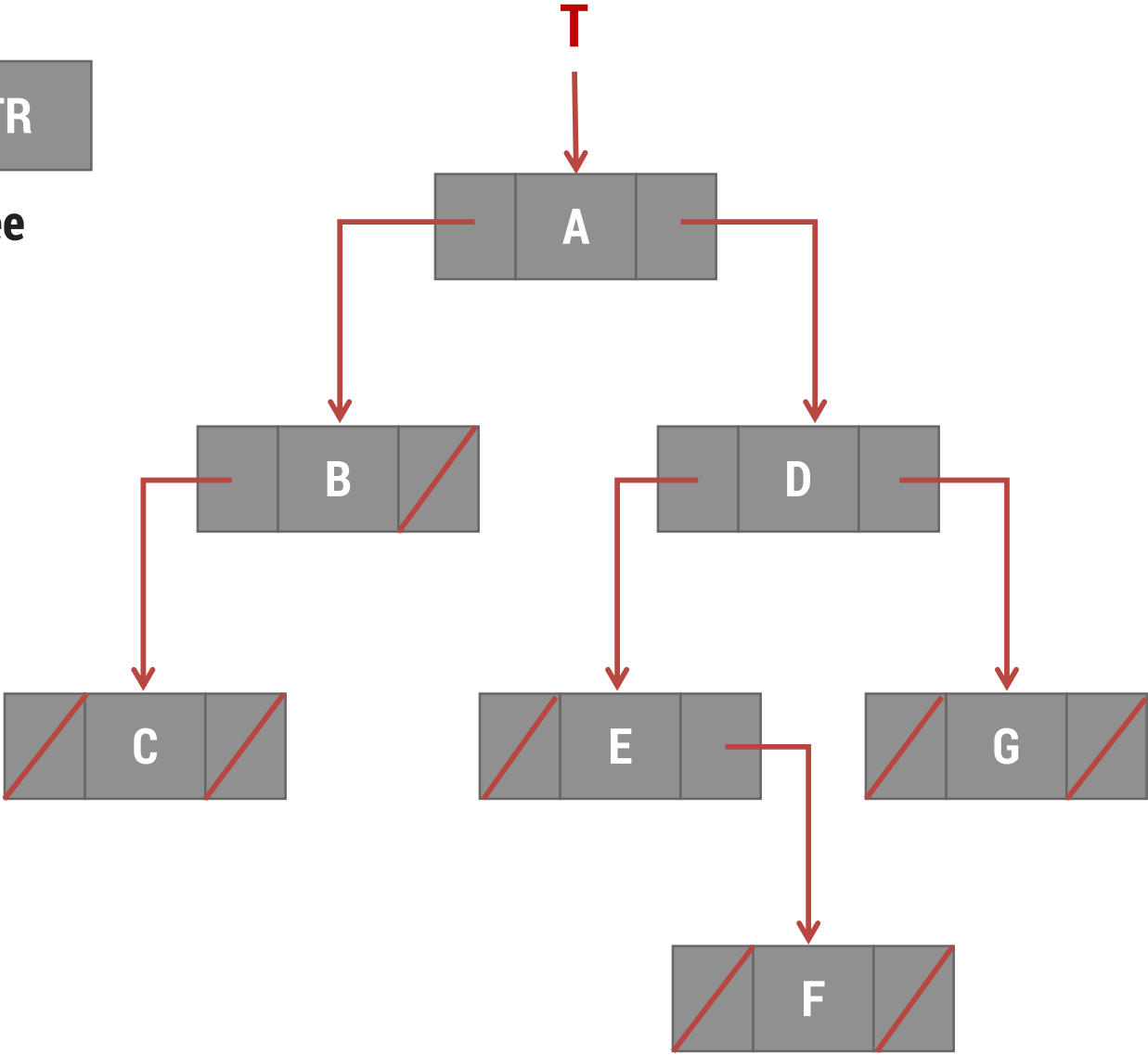
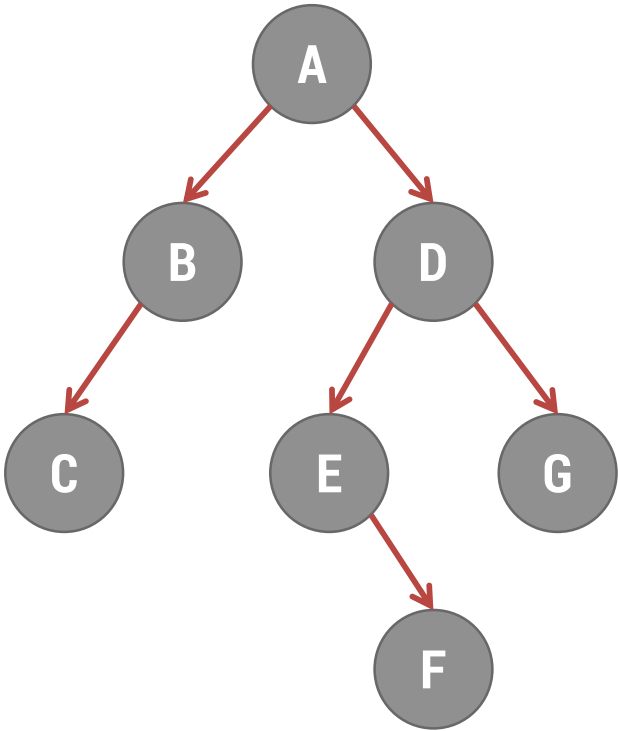




# Linked Representation of Binary Tree



Typical node of Binary Tree



# Algorithm of Binary Tree Traversal

- ▶ Preorder Traversal - Procedure: RPREORDER(T)
- ▶ Inorder Traversal - Procedure: RINORDER(T)
- ▶ Postorder Traversal - Procedure: RPOSTORDER(T)

# Procedure: RPREORDER(T)

- ▶ This procedure **traverses the tree** in **preorder**, in a recursive manner.
- ▶ **T is root node address** of given binary tree
- ▶ Node structure of binary tree is described as below

LPTR	DATA	RPTR
------	------	------

Typical node of Binary Tree

## 1. [Check for Empty Tree]

```
IF      T = NULL
THEN   write ('Empty Tree')
       return
```

```
ELSE   write (DATA(T))
```

## 2. [Process the Left Sub Tree]

```
IF      LPTR (T) ≠ NULL
THEN   RPREORDER (LPTR (T))
```

## 3. [Process the Right Sub Tree]

```
IF      RPTR (T) ≠ NULL
THEN   RPREORDER (RPTR (T))
```

## 4. [Finished]

```
Return
```

# Procedure: RINORDER(T)

- ▶ This procedure **traverses the tree** in **InOrder**, in a recursive manner.
- ▶ **T is root node address** of given binary tree.
- ▶ Node structure of binary tree is described as below.

LPTR	DATA	RPTR
------	------	------

Typical node of Binary Tree

## 1. [Check for Empty Tree]

```
IF      T = NULL
THEN   write ('Empty Tree')
       return
```

## 2. [Process the Left Sub Tree]

```
IF      LPTR (T) ≠ NULL
THEN   RINORDER (LPTR (T))
```

## 3. [Process the Root Node]

```
write (DATA(T))
```

## 4. [Process the Right Sub Tree]

```
IF      RPTR (T) ≠ NULL
THEN   RINORDER (RPTR (T))
```

## 5. [Finished]

```
Return
```

# Procedure: RPOSTORDER(T)

- ▶ This procedure **traverses the tree** in **PostOrder**, in a recursive manner.
- ▶ **T is root node address** of given binary tree.
- ▶ Node structure of binary tree is described as below.

LPTR	DATA	RPTR
------	------	------

Typical node of Binary Tree

```
1. [Check for Empty Tree]
   IF      T = NULL
   THEN write ('Empty Tree')
        return

2. [Process the Left Sub Tree]
   IF      LPTR (T) ≠ NULL
   THEN RPOSTORDER (LPTR (T))

3. [Process the Right Sub Tree]
   IF      RPTR (T) ≠ NULL
   THEN RPOSTORDER (RPTR (T))
```

```
4. [Process the Root Node]
   write (DATA(T))

5. [Finished]
   Return
```

# Construct Binary Tree from Traversal

Construct a Binary tree from the given **Inorder** and **Postorder** traversals

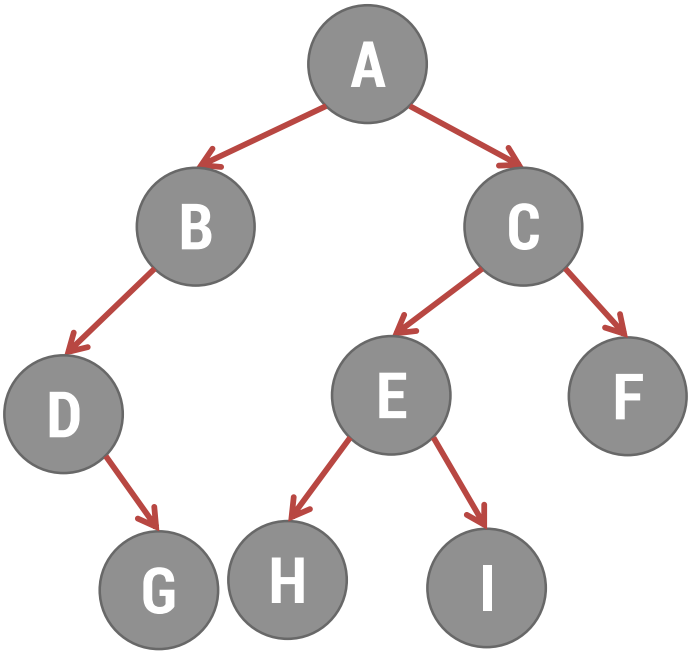
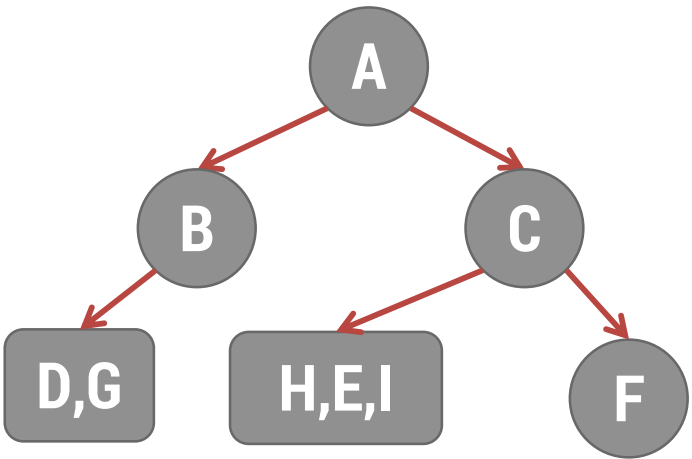
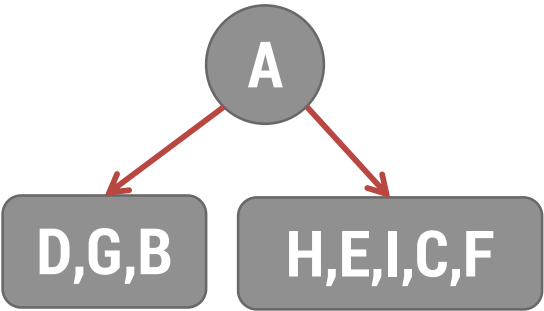
**Inorder** : D G B A H E I C F

**Postorder** : G D B H I E F C A

- Step 1: Find the root node
  - Preoder Traversal – first node is root node
  - Postoder Traversal last node is root node
- Step 2: Find Left & Right Sub Tree
  - Inorder traversal gives Left and right sub tree

**Postorder** : G D B H I E F C **A**

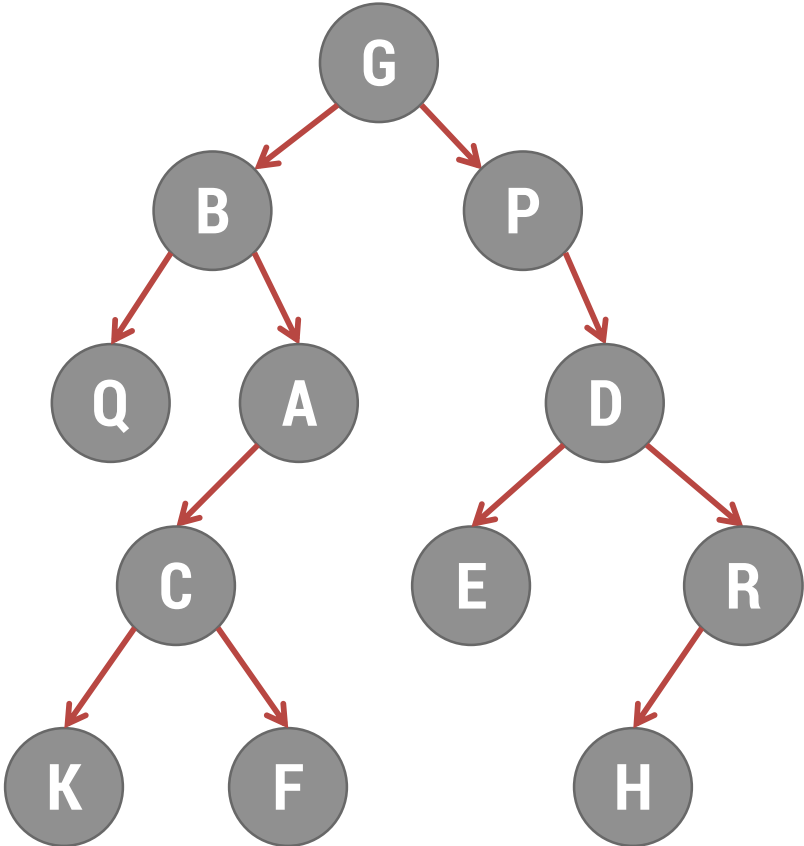
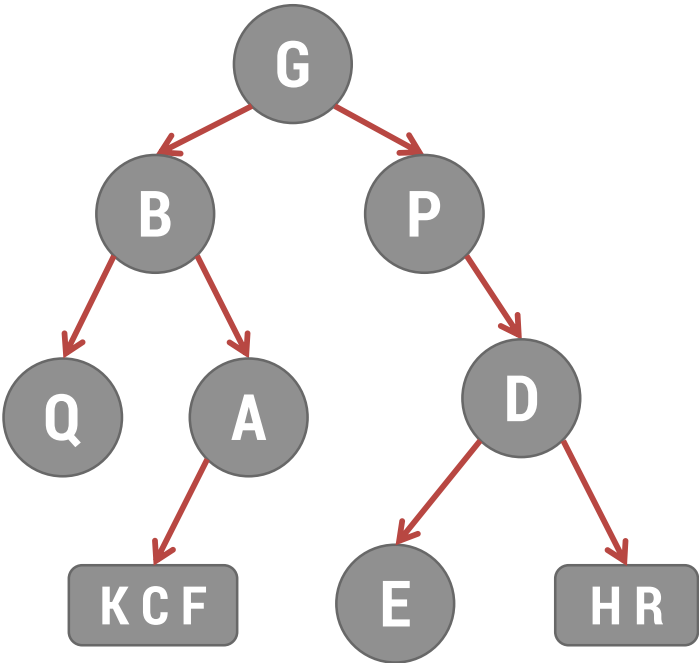
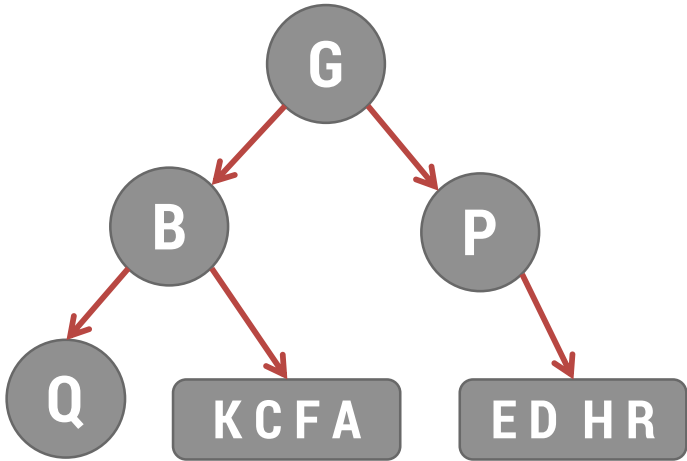
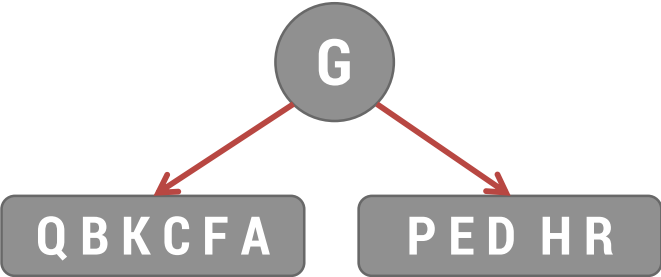
**Inorder** : D G B **A** H E I C F



# Construct Binary Tree from Traversal

Preorder : **G** B Q A C K F P D E R H

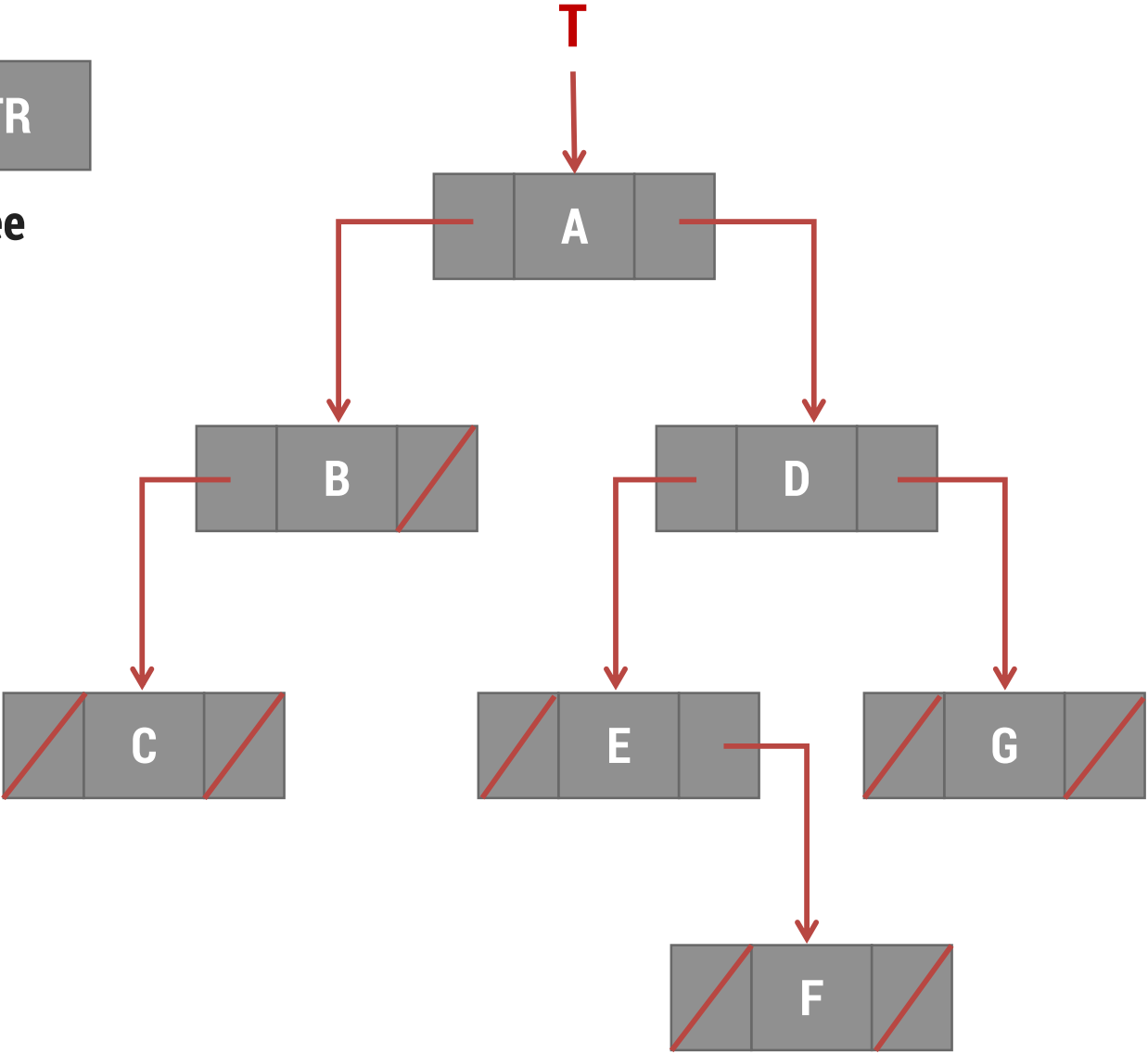
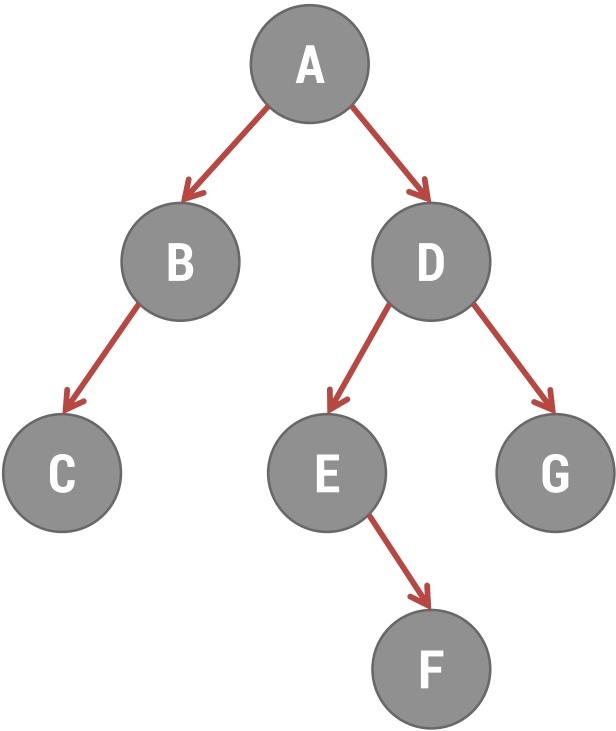
Inorder : Q B K C F A **G** P E D H R



# Linked Representation of Binary Tree



Typical node of Binary Tree





# Threaded Binary Tree

- ▶ The **wasted NULL** links in the binary tree storage representation can be **replaced by threads**
- ▶ A binary **tree** is **threaded according** to particular **traversal order**. e.g.: Threads for the inorder traversals of tree are pointers to its higher nodes, for this traversal order
- ▶ **In-Threaded Binary Tree**
  - If left link of **node P is null**, then this link is **replaced by** the **address of its predecessor**
  - If right link of **node P is null**, then this link is **replaced by** the **address of its successor**
- ▶ Because the left or right **link** of a **node** can denote **either structural link** or **a thread**, we must somehow be able to distinguish them

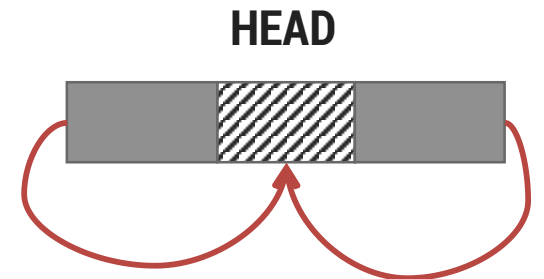
# Threaded Binary Tree

- ▶ **Method 1:-** Represent **thread a Negative address**
- ▶ **Method 2:-** To have a **separate Boolean flag** for each of left and right pointers, node structure for this is given below

LPTR	LTHREAD	DATA	RTHREAD	RPTR
------	---------	------	---------	------

Typical node of Threaded Binary Tree

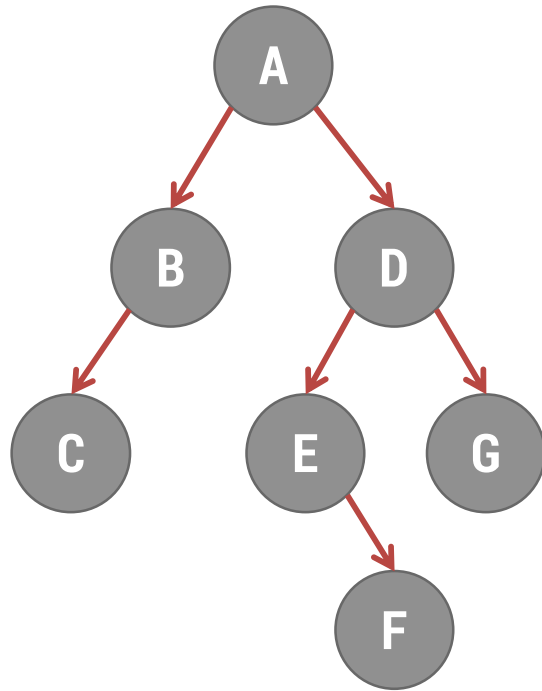
- **LTHREAD = true** = Denotes leaf thread link
- **LTHREAD = false** = Denotes leaf structural link
- **RTHREAD = true** = Denotes right threaded link
- **RTHREAD = false** = Denotes right structural link



Head node is simply another node which serves as the predecessor and successor of first and last tree nodes.

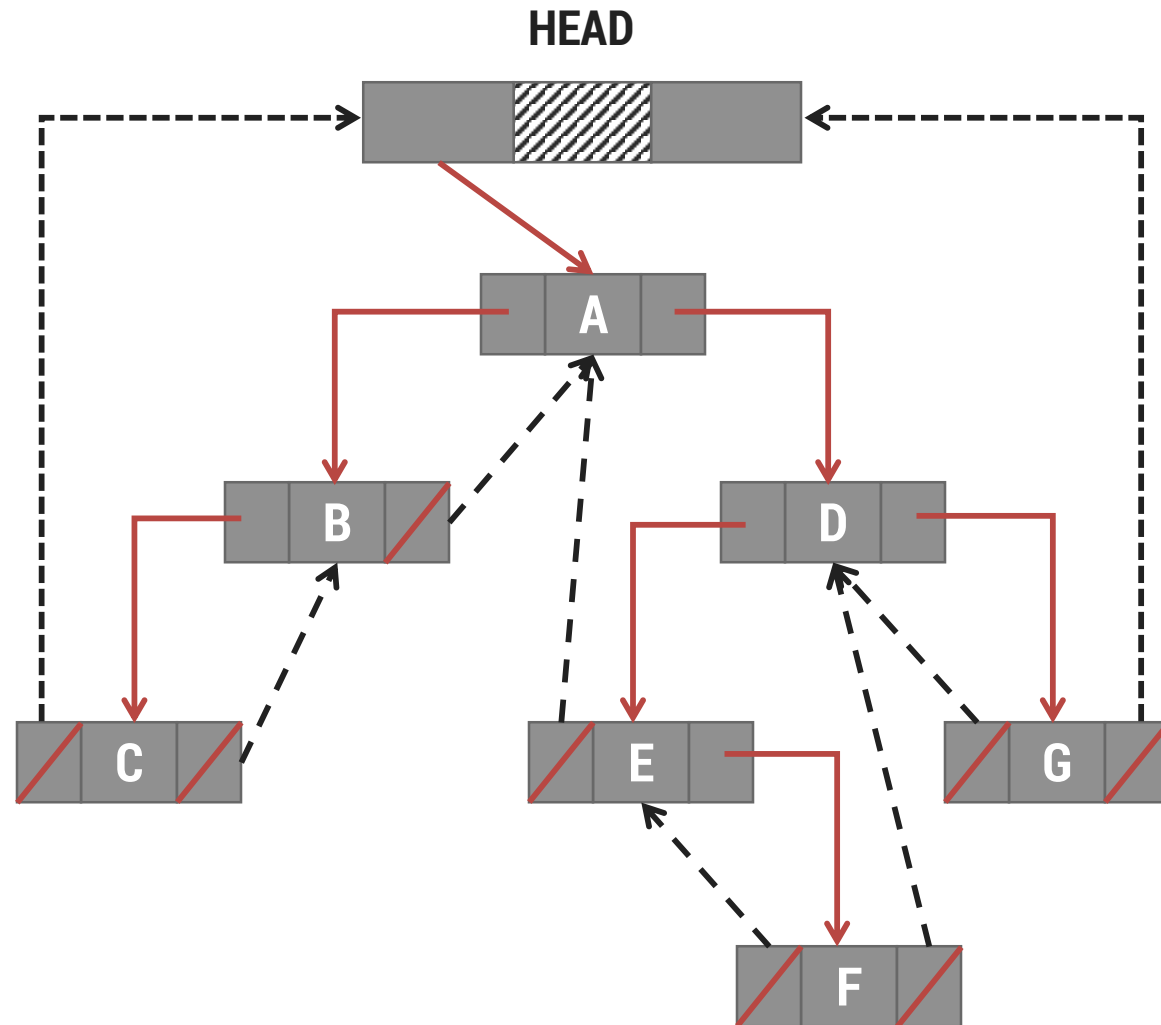
Tree is attached to the left branch of the head node.

# Threaded Binary Tree



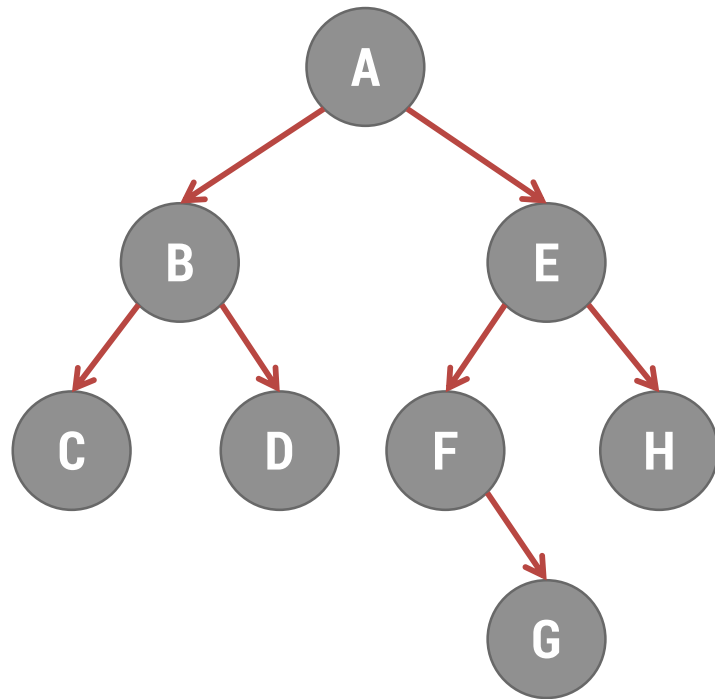
Inorder Traversal

**C B A E F D G**



Fully In-Threaded Binary Tree

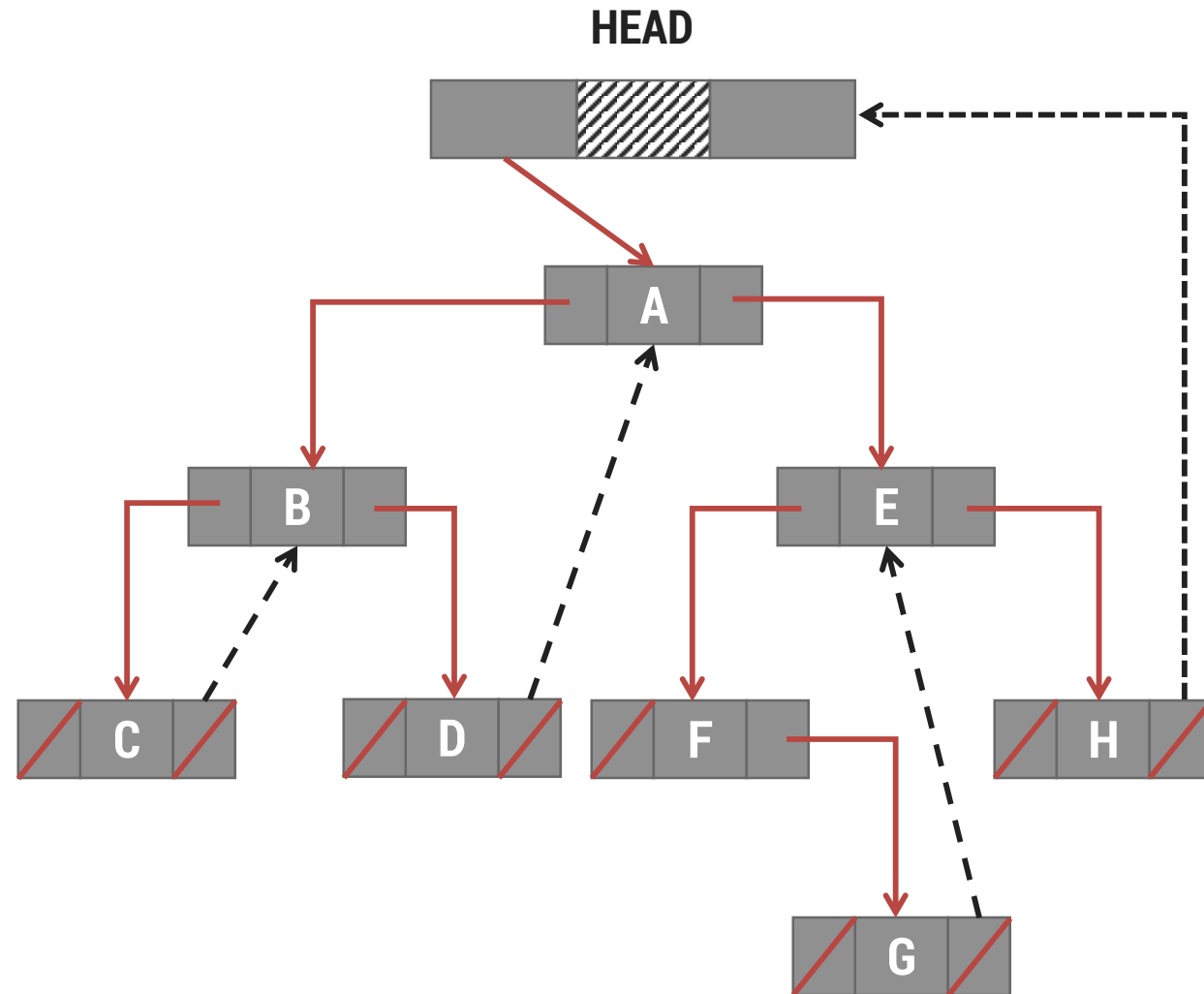
# Threaded Binary Tree



Inorder Traversal

**C B D A F G E H**

Construct Right In-Threaded Binary Tree of given Tree



# Advantages of Threaded Binary Tree

- ▶ **Inorder traversal is faster** than unthreaded version as stack is not required.
- ▶ **Effectively determines** the **predecessor and successor** for inorder traversal, for unthreaded tree this task is more difficult.
- ▶ **A stack is required** to provide upward pointing information **in binary tree** which **threading provides without stack**.
- ▶ It is possible to **generate successor or predecessor** of any node **without** having over head of **stack** with the help of threading.

# Disadvantages of Threaded Binary Tree

- ▶ Threaded trees are **unable to share common sub trees**.
- ▶ If **Negative addressing is not permitted** in programming language, **two additional fields are required**.
- ▶ **Insertion** into and **deletion** from threaded binary tree are **more time consuming** because both thread and structural link must be maintained.

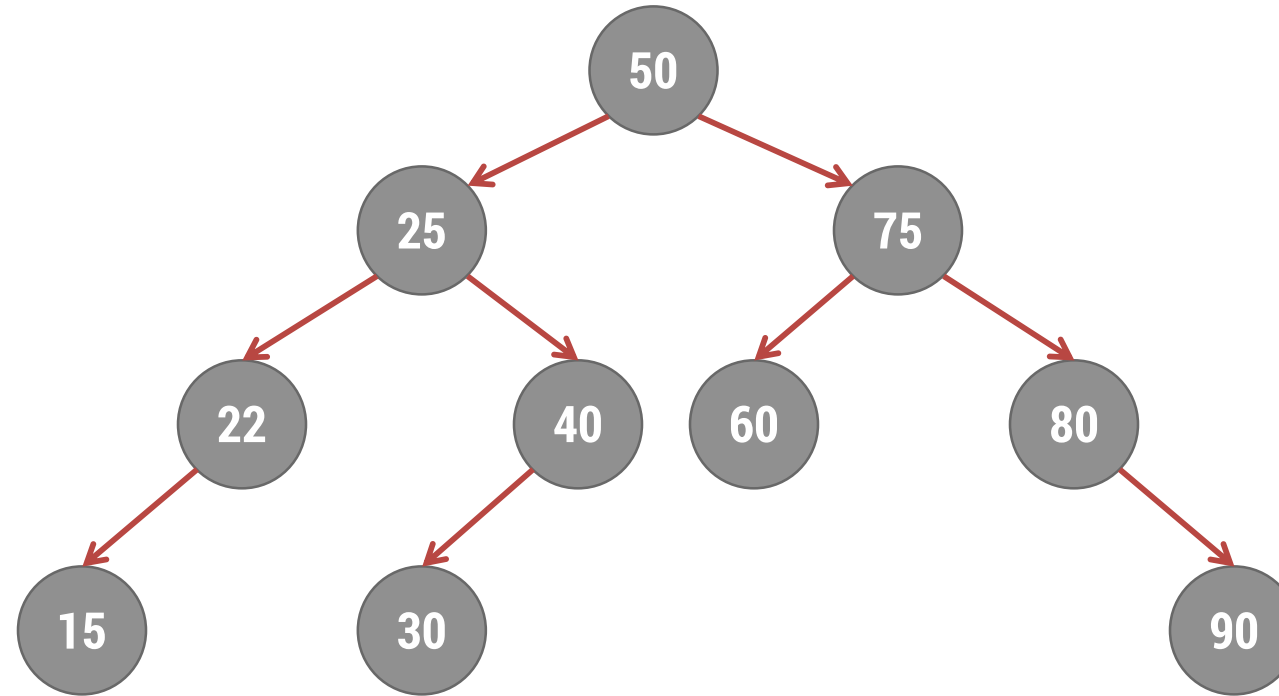
# Binary Search Tree (BST)

► A **binary search tree** is a **binary tree** in which **each node** possessed a key that **satisfy** the **following conditions**

1. All **key** (if any) in **the left sub tree** of the root **precedes the key** in the **root**
2. The **key in the root precedes** all **key** (if any) in the **right sub tree**
3. The **left and right sub trees** of the root are again **search trees**

# Construct Binary Search Tree (BST)

Construct binary search tree for the following data  
50 , 25 , 75 , 22 , 40 , 60 , 80 , 90 , 15 , 30



---

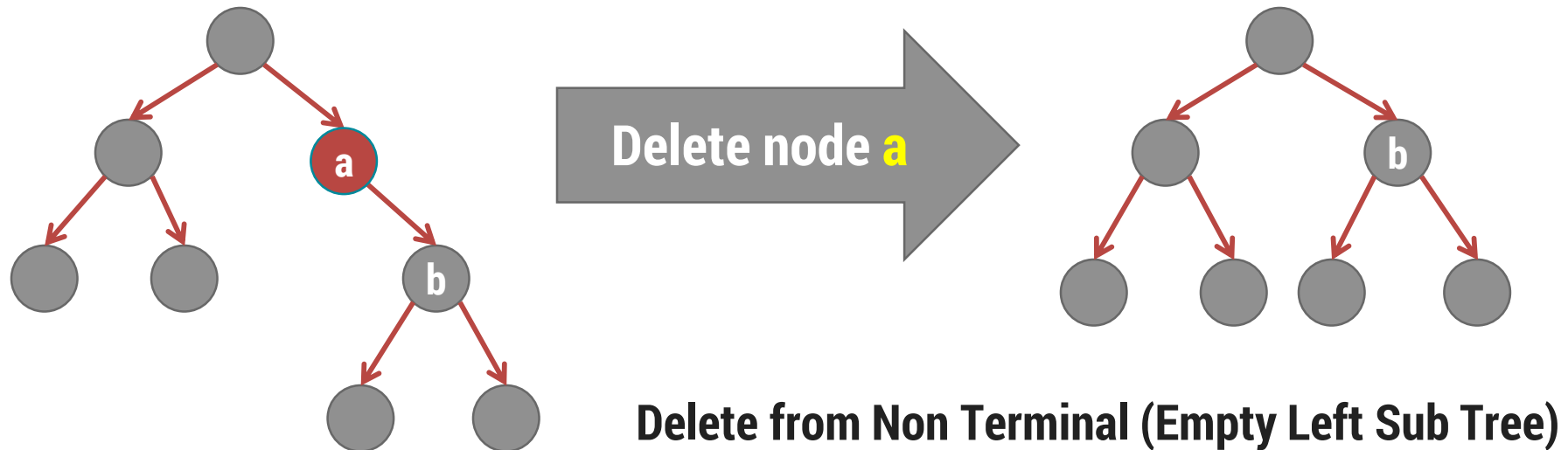
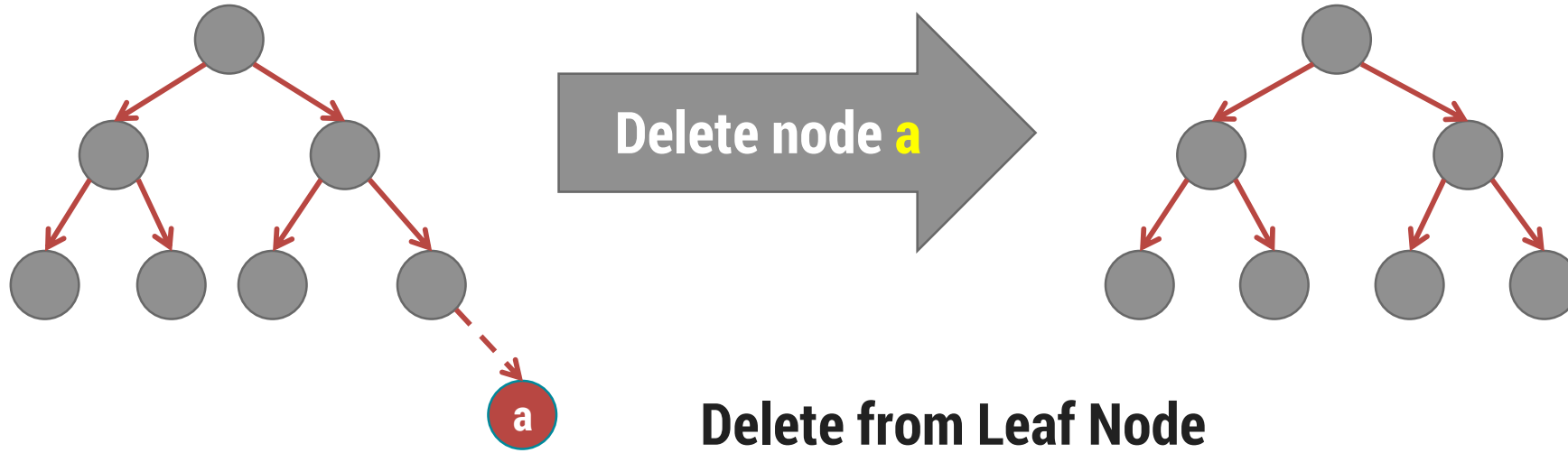
Construct binary search tree for the following data  
10, 3, 15, 22, 6, 45, 65, 23, 78, 34, 5



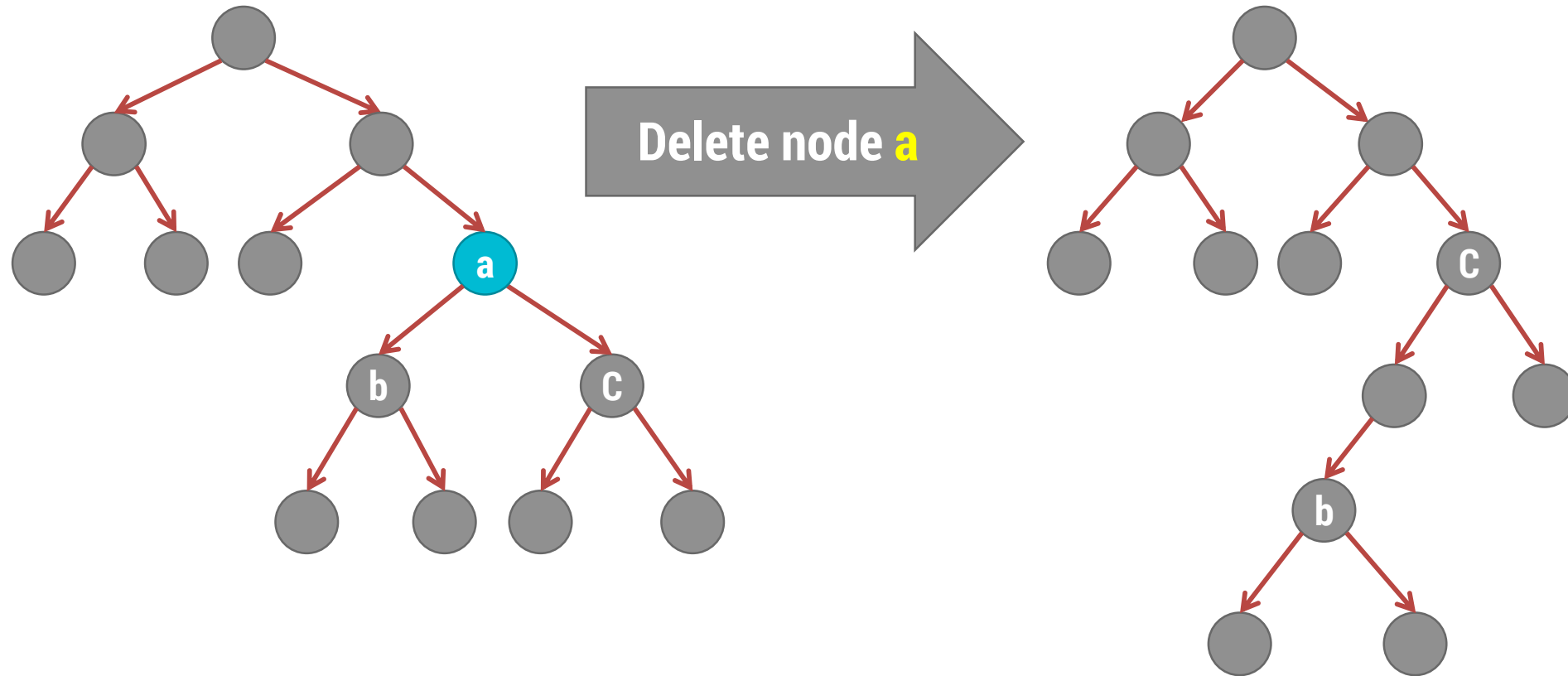
# Search a node in Binary Search Tree

- ▶ To search for target value.
- ▶ We first compare it with the key at root of the tree.
- ▶ If it is not same, we go to either Left sub tree or Right sub tree as appropriate and repeat the search in sub tree.
- ▶ If we have **In-Order List** & we want to search for specific node it requires  **$O(n)$  time**.
- ▶ In case of **Binary tree** it requires  **$O(\log_2 n)$**  time to search a node.

# Delete node from Binary Search Tree



# Delete node from BST



**Delete from Non Terminal (Neither Sub Tree is Empty)**

THANK YOU

# **Non-Linear Data Structure Tree Part-2**

# Red - Black Tree

- ▶ **Red Black Tree is a Binary Search Tree in which every node is colored either RED or BLACK.**

We can define a Red Black Tree as follows...

Red Black Tree is a Binary Search Tree in which every node is colored either RED or BLACK.

- ▶ Every Red Black Tree has the following properties.

## Properties of Red Black Tree

- ▶ **Property #1:** Red - Black Tree must be a Binary Search Tree.
- ▶ **Property #2:** The ROOT node must be colored BLACK.
- ▶ **Property #3:** The children of Red colored node must be colored BLACK. (There should not be two consecutive RED nodes).
- ▶ **Property #4:** In all the paths of the tree, there should be same number of BLACK colored nodes.
- ▶ **Property #5:** Every new node must be inserted with RED color.
- ▶ **Property #6:** Every leaf (e.i. NULL node) must be colored BLACK.

































# Unit-3:

## Divide and Conquer

# **Algorithms**



## Outline

- Introduction to Recurrence Equation
- Different methods to solve recurrence
- Divide and Conquer Technique
- Multiplying large Integers Problem
- Problem Solving using divide and conquer algorithm –
  - ✓ Binary Search
  - ✓ Sorting (Merge Sort, Quick Sort)
  - ✓ Matrix Multiplication
  - ✓ Exponential

# Recurrence Equation

# Introduction

- ▶ Many algorithms (divide and conquer) are **recursive** in nature.
- ▶ When we analyze them, we get a **recurrence relation** for time complexity.
- ▶ We get running time as a function of  $n$  (input size) and we get the running time **on inputs of smaller sizes**.
- ▶ A recurrence is a **recursive description of a function**, or a description of a function in terms of itself.
- ▶ A recurrence relation **recursively defines a sequence** where the next term is a function of the previous terms.



# Methods to Solve Recurrence

- ▶ Substitution
- ▶ Homogeneous (characteristic equation)
- ▶ Inhomogeneous
- ▶ Master method
- ▶ Recurrence tree
- ▶ Intelligent guess work
- ▶ Change of variable
- ▶ Range transformations

# Substitution Method – Example 1

- ▶ We make a guess for the solution and then we use mathematical induction to prove the guess is correct or incorrect.

## Example 1:

Time to solve the  
instance of size  $n - 1$

$$T(n) = \underline{T(n-1)} + n$$

1

- ▶ Replacing  $n$  by  $n-1$  and  $n-2$ , we can write following equations.

Time to solve the  
instance of size  $n$

$$\underline{T(n-1)} = \underline{T(n-2)} + n - 1$$

2

$$\underline{T(n-2)} = T(n-3) + n - 2$$

3

- ▶ Substituting equation 3 in 2 and equation 2 in 1 we have now,

$$T(n) = T(n-3) + n - 2 + n - 1 + n$$

4

# Substitution Method – Example 1

$$T(n) = T(n - 3) + n - 2 + n - 1 + n \dots \textcircled{4}$$

- From above, we can write the general form as,

$$T(n) = T(n - k) + (n - k + 1) + (n - k + 2) + \dots + n$$

- Suppose, if we take  $k = n$  then,

$$T(n) = T(n - n) + (n - n + 1) + (n - n + 2) + \dots + n$$

$$T(n) = 0 + 1 + 2 + \dots + n$$

$$T(n) = \frac{n(n + 1)}{2} = O(n^2)$$

# Substitution Method – Example 2

$$t(n) = \begin{cases} c1 & \text{if } n = 0 \\ c2 + t(n - 1) & \text{o/w} \end{cases}$$

---

► Rewrite the equation,  $t(n) = c2 + t(n - 1)$

► Now, replace  $n$  by  $n - 1$  and  $n - 2$

$$\begin{aligned} t(n - 1) &= c2 + \underline{t(n - 2)} & \therefore \underline{t(n - 1)} &= c2 + c2 + t(n - 3) \\ \underline{t(n - 2)} &= c2 + t(n - 3) \end{aligned}$$

► Substitute the values of  $n - 1$  and  $n - 2$

$$t(n) = c2 + c2 + c2 + t(n - 3)$$

► In general,

$$t(n) = kc2 + t(n - k)$$

► Suppose if we take  $k = n$  then,

$$t(n) = nc2 + t(n - n) = nc2 + t(0)$$

$$\boxed{t(n) = nc2 + c1 = O(n)}$$

# Substitution Method Exercises

► Solve the following recurrences using substitution method.

1.  $T(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } 1 \\ T(n-1) + n - 1 & \text{o/w} \end{cases}$

2.  $T(n) = T(n-1) + 1$  and  $T(1) = \theta(1)$ .

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= T(n-2) + (n-1) + n \\ &= T(n-3) + (n-2) + (n-1) + n \\ &\vdots \\ &= T(0) + 1 + 2 + \dots + (n-2) + (n-1) + n \\ &= T(0) + \frac{n(n+1)}{2} = O(n^2) \end{aligned}$$

# Homogeneous Recurrence

- ▶ Recurrence equation

$$a_0 t_n + a_1 t_{n-1} + a_2 t_{n-2} + \cdots + a_k t_{n-k} = 0$$

- ▶ The equation of degree  $k$  in  $x$  is called the **characteristic equation** of the recurrence,

$$p(x) = a_0 x^k + a_1 x^{k-1} + \cdots + a_k x^0$$

- ▶ Which can be factorized as,

$$p(x) = \prod_{i=1}^k (x - r_i)$$

- ▶ The solution of recurrence is given as,

$$t_n = \sum_{i=1}^k c_i r_i^n$$

# Homogeneous Recurrence – Example 1 : Fibonacci Series

## ► Fibonacci series Iterative Algorithm

```
Function fibiter(n)
    i ← 1; j ← 0;
    for k ← 1 to n do
        j ← i + j;
        i ← j - i;
    return j
```

- **Analysis of Iterative Algorithm:** If we count all arithmetic operations at unit cost; the instructions inside *for* loop take constant time  $c$ . The time taken by the *for* loop is bounded above by  $n$ , *i.e.*,  $nc = \Theta(n)$

Case 1

# Homogeneous Recurrence – Example 1 : Fibonacci Series

- ▶ If the value of  $n$  is large, then time needed to execute addition operation increases linearly with the length of operand.
- ▶ At the end of  $k^{th}$  iteration, the value of  $i$  and  $j$  will be  $f_{k-1}$  and  $f_k$ .
- ▶ As per De Moivre's formula the size of  $f_k$  is in  $\theta(k)$ .
- ▶ So,  $k^{th}$  iteration takes time in  $\theta(k)$ . let  $c$  be some constant such that this time is bounded above by  $ck$  for all  $k \geq 1$ .
- ▶ The time taken by *fibiter* algorithm is bounded above by,

$$\sum_{k=1}^n c \cdot k = c \cdot \sum_{k=1}^n k = c \cdot \frac{n(n+1)}{2}$$

$$T(n) = \theta(n^2)$$

Case 2



# Homogeneous Recurrence – Example 1 : Fibonacci Series

- ▶ Recursive Algorithm for Fibonacci series,

```
Function fibrec(n)
    if n < 2 then return n
    else return fibrec (n - 1) + fibrec (n - 2)
```

- ▶ The recurrence equation of above algorithm is given as,

$$T(n) = \begin{cases} n & \text{if } n = 0 \text{ or } 1 \\ T(n-1) + T(n-2) & \text{o/w} \end{cases}$$

- ▶ The recurrence can be re-written as,

$$T(n) - T(n-1) - T(n-2) = 0$$

- ▶ The characteristic polynomial is,

$$x^2 - x - 1 = 0$$

# Homogeneous Recurrence – Example 1 : Fibonacci Series

- ▶ Find the roots of characteristic polynomial,

$$x^2 - x - 1 = 0$$

- ▶ The roots are,

$$r_1 = \frac{1+\sqrt{5}}{2} \quad \text{and} \quad r_2 = \frac{1-\sqrt{5}}{2}$$

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Here,  $a = 1$ ,  $b = 1$  and  $c = 1$

- ▶ The general solution is therefore of the form,

$$T_n = c_1 r_1^n + c_2 r_2^n$$

$$T_n = \sum_{i=1}^k c_i r_i^n$$

- ▶ Substituting initial values  $n = 0$  and  $n = 1$

$$T_0 = c_1 + c_2 = 0 \quad (1)$$

$$T_1 = c_1 r_1 + c_2 r_2 = 1 \quad (2)$$

- ▶ Solving these equations, we obtain

$$c_1 = \frac{1}{\sqrt{5}} \quad \text{and} \quad c_2 = -\frac{1}{\sqrt{5}}$$

# Homogeneous Recurrence – Example 1 : Fibonacci Series

- ▶ Substituting the values of roots and constants in general solution,

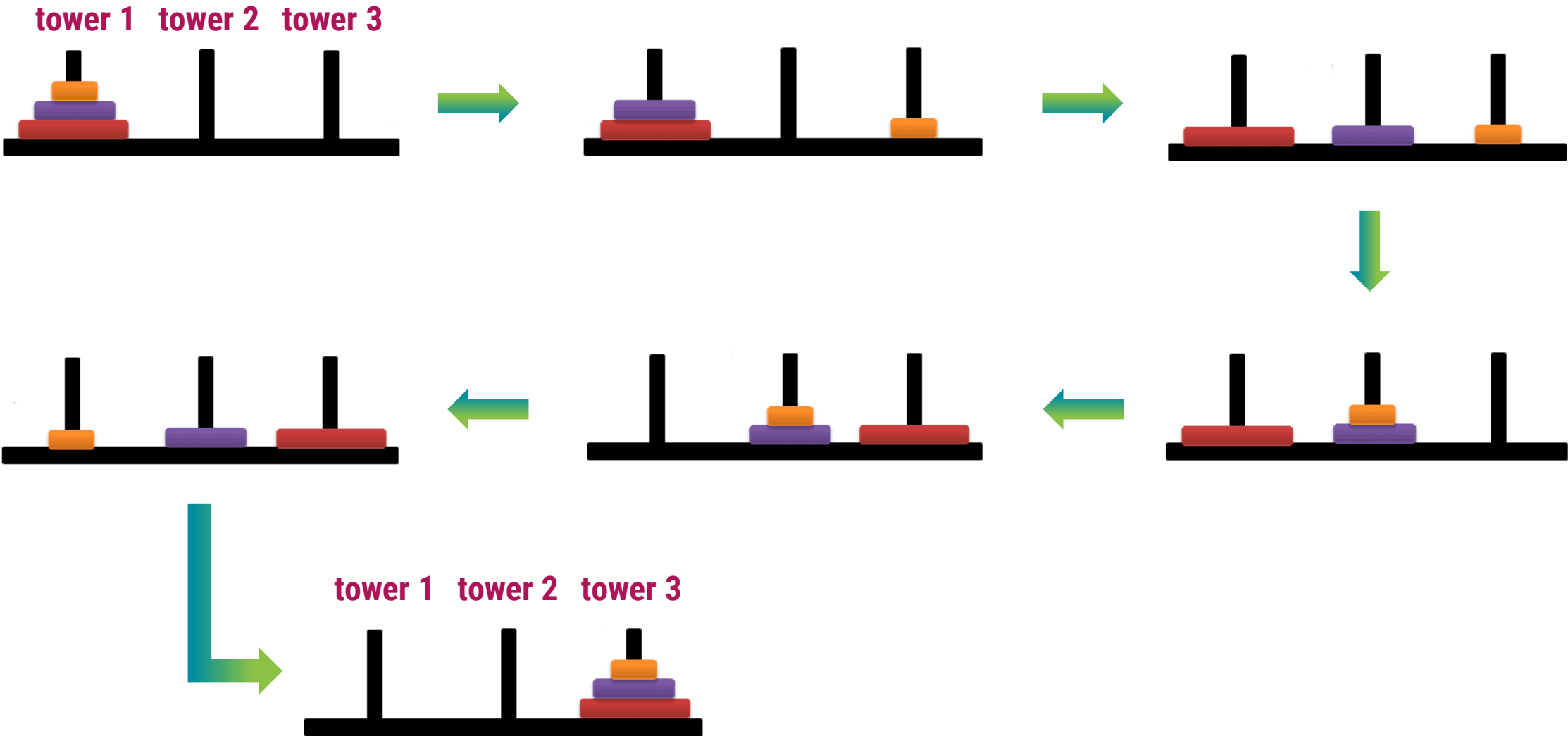
$$T_n = c_1 r_1^n + c_2 r_2^n$$

$$T_n = \frac{1}{\sqrt{5}} \left[ \left( \frac{1 + \sqrt{5}}{2} \right)^n - \left( \frac{1 - \sqrt{5}}{2} \right)^n \right] \dots \dots \dots \text{de Moivre's formula}$$

$$T_n \in O(\phi)^n$$

- ▶ Time taken for recursive Fibonacci algorithm grows **Exponentially**.

# Example 2 : Tower of Hanoi



## Example 2 : Tower of Hanoi

- ▶ The number of movements of a ring required in the tower of Hanoi problem is given by,

$$t(m) = \begin{cases} 0 & \text{if } m = 0 \\ 2t(m-1) + 1 & \text{o/w} \end{cases}$$

- ▶ The equation can be written as,

$$t(m) - 2t(m-1) = 1 \quad (1)$$

Inhomogeneous equation

- ▶ To convert it into a homogeneous equation, multiply with  $-1$  and replace  $m$  by  $m-1$ ,

$$-t(m-1) + 2t(m-2) = -1 \quad (2)$$

- ▶ Solving equations (1) and (2), we have now

$$t(m) - 3t(m-1) + 2t(m-2) = 0$$

## Example 2 : Tower of Hanoi

- ▶ The characteristic polynomial is,

$$x^2 - 3x + 2 = 0$$

$$t(m) - 3t(m-1) + 2t(m-2) = 0$$

- ▶ Whose roots are,

$$r_1 = 2 \text{ and } r_2 = 1$$

- ▶ The general solution is therefore of the form,

$$t_m = c_1 1^m + c_2 2^m$$

- ▶ Substituting initial values  $m = 0$  and  $m = 1$

$$t_0 = c_1 + c_2 = 0 \quad (1)$$

$$t_1 = c_1 + 2c_2 = 1 \quad (2)$$

- ▶ Solving these linear equations we get  $c_1 = -1$  and  $c_2 = 1$ .
- ▶ Therefore, time complexity of tower of Hanoi problem is given as,

$$t(m) = 2^m - 1 = O(2^m)$$

# Homogeneous Recurrence Exercises

► Solve the following recurrences

$$1. \quad t_n = \begin{cases} n & \text{if } n = 0 \text{ or } 1 \\ 5t_{n-1} - 6t_{n-2} & \text{o/w} \end{cases}$$

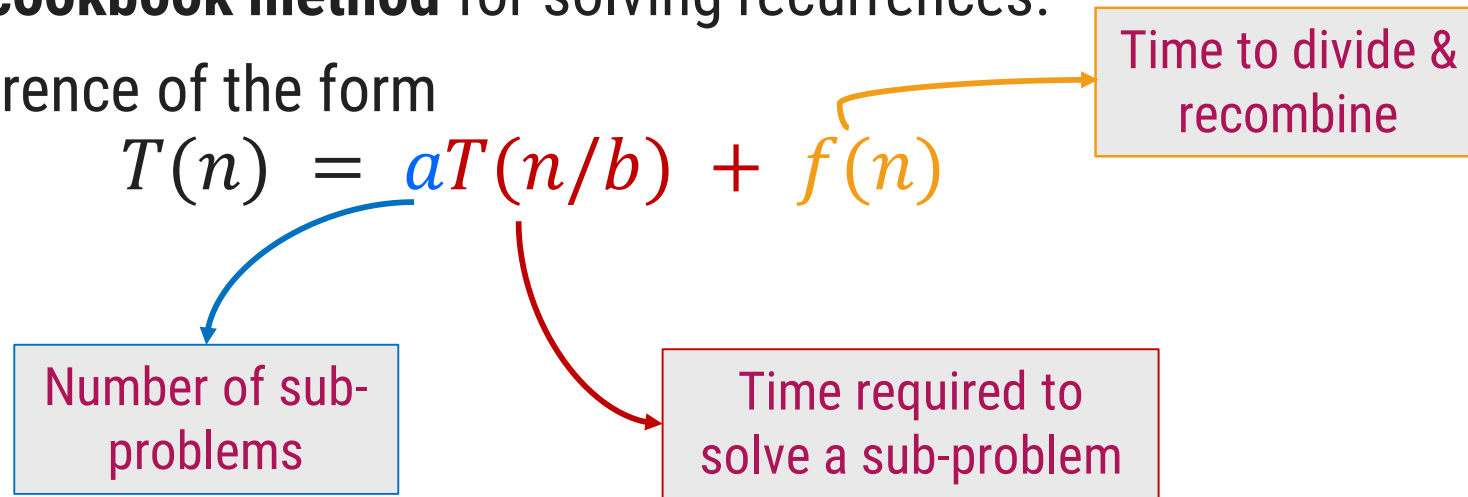
$$2. \quad t_n = \begin{cases} n & \text{if } n = 0, 1 \text{ or } 2 \\ 5t_{n-1} - 8t_{n-2} + 4t_{n-3} & \text{o/w} \end{cases}$$

# Master Theorem

- ▶ The master theorem is a **cookbook method** for solving recurrences.

- ▶ Suppose you have a recurrence of the form

$$T(n) = aT(n/b) + f(n)$$



- ▶ **This recurrence would arise in the analysis of a recursive algorithm.**
- ▶ When input size  $n$  is large, the problem is divided up into  $a$  sub-problems each of size  $n/b$ . Sub-problems are solved recursively and results are recombined.
- ▶ The work to split the problem into sub-problems and recombine the results is  $f(n)$ .



# Master Theorem – Example 1

$$T(n) = aT(n/b) + f(n)$$

► There are three cases:

1. case 1: if  $f(n)$  is in  $\mathcal{O}(n^{\log_b a})$  [ $f(n) \leq n^{\log_b a}$ ] then  $T(n) = \theta(n^{\log_b a})$
2. case 2:  $f(n)$  is in  $\theta(n^{\log_b a})$  [ $f(n) = n^{\log_b a}$ ] then  $T(n) = \theta(n^{\log_b a} \lg n)$
3. case 3:  $f(n)$  is in  $\Omega(n^{\log_b a})$  [ $f(n) \geq n^{\log_b a}$ ] then  $T(n) = \theta(f(n))$

► Example 1:  $T(n) = 2T(n/2) + \theta(n)$

Merge sort

► Here  $a = 2, b = 2$ . So,  $n^{\log_b a} = n$

► Also,  $f(n) = \theta(n) = cn$

► Case 2 applies:  $T(n) = \theta(n \lg n)$

# Master Theorem – Example 2

$$T(n) = aT(n/b) + f(n)$$

► There are three cases:

1. case 1: if  $f(n)$  is in  $\mathcal{O}(n^{\log_b a})$  [ $f(n) \leq n^{\log_b a}$ ] then  $T(n) = \theta(n^{\log_b a})$
  2. case 2:  $f(n)$  is in  $\theta(n^{\log_b a})$  [ $f(n) = n^{\log_b a}$ ] then  $T(n) = \theta(n^{\log_b a} \lg n)$
  3. case 3:  $f(n)$  is in  $\Omega(n^{\log_b a})$  [ $f(n) \geq n^{\log_b a}$ ] then  $T(n) = \theta(f(n))$
- 

► Example 2:  $T(n) = T(n/2) + \theta(1)$  **Binary Search**

► Here  $a = 1, b = 2$ . So,  $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$

►  $f(n) = \theta(1) = 1$

► **Case 2 applies: the solution is  $\theta(n^{\log_b a} \log n)$**

►  $T(n) = \theta(\log n)$

# Master Theorem – Example 3

$$T(n) = aT(n/b) + f(n)$$

► There are three cases:

1. case 1: if  $f(n)$  is in  $\mathcal{O}(n^{\log_b a})$  [ $f(n) \leq n^{\log_b a}$ ] then  $T(n) = \theta(n^{\log_b a})$
  2. case 2:  $f(n)$  is in  $\Theta(n^{\log_b a})$  [ $f(n) = n^{\log_b a}$ ] then  $T(n) = \theta(n^{\log_b a} \lg n)$
  3. case 3:  $f(n)$  is in  $\Omega(n^{\log_b a})$  [ $f(n) \geq n^{\log_b a}$ ] then  $T(n) = \theta(f(n))$
- 

► Example 3:  $T(n) = 4T(n/2) + n$

► Here  $a = 4, b = 2$ . So,  $\log_b a = 2$  and  $n^{\log_b a} = n^2$

►  $f(n) = n$ ,

► So,  $f(n) \leq n^2 \Rightarrow f(n)$  is in  $\mathcal{O}(n^{\log_b a})$

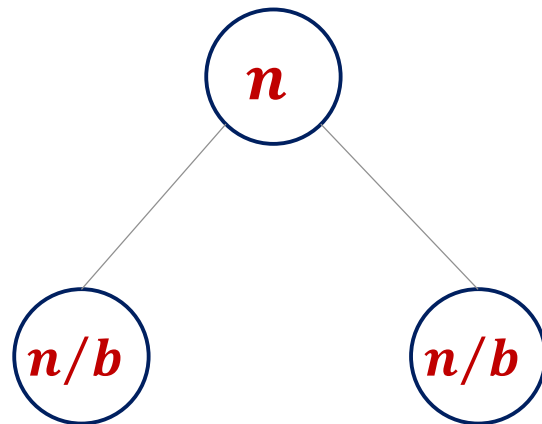
► **Case 1 applies:**  $T(n) = \theta(n^2)$

# Master Theorem Exercises

- ▶ Example 4:  $T(n) = 4T(n/2) + n^2$
- ▶ Example 5:  $T(n) = 4T(n/2) + n^3$
- ▶ Example 6:  $T(n) = 9T(n/3) + n$  (Summer 17, Summer 19)
- ▶ Example 7:  $T(n) = T(2n/3) + 1$  (Summer 17)
- ▶ Example 8:  $T(n) = 7T(n/2) + n^3$  (Winter 18)
- ▶ Example 9:  $T(n) = 27T(n^2) + 16n$  (Winter 19)

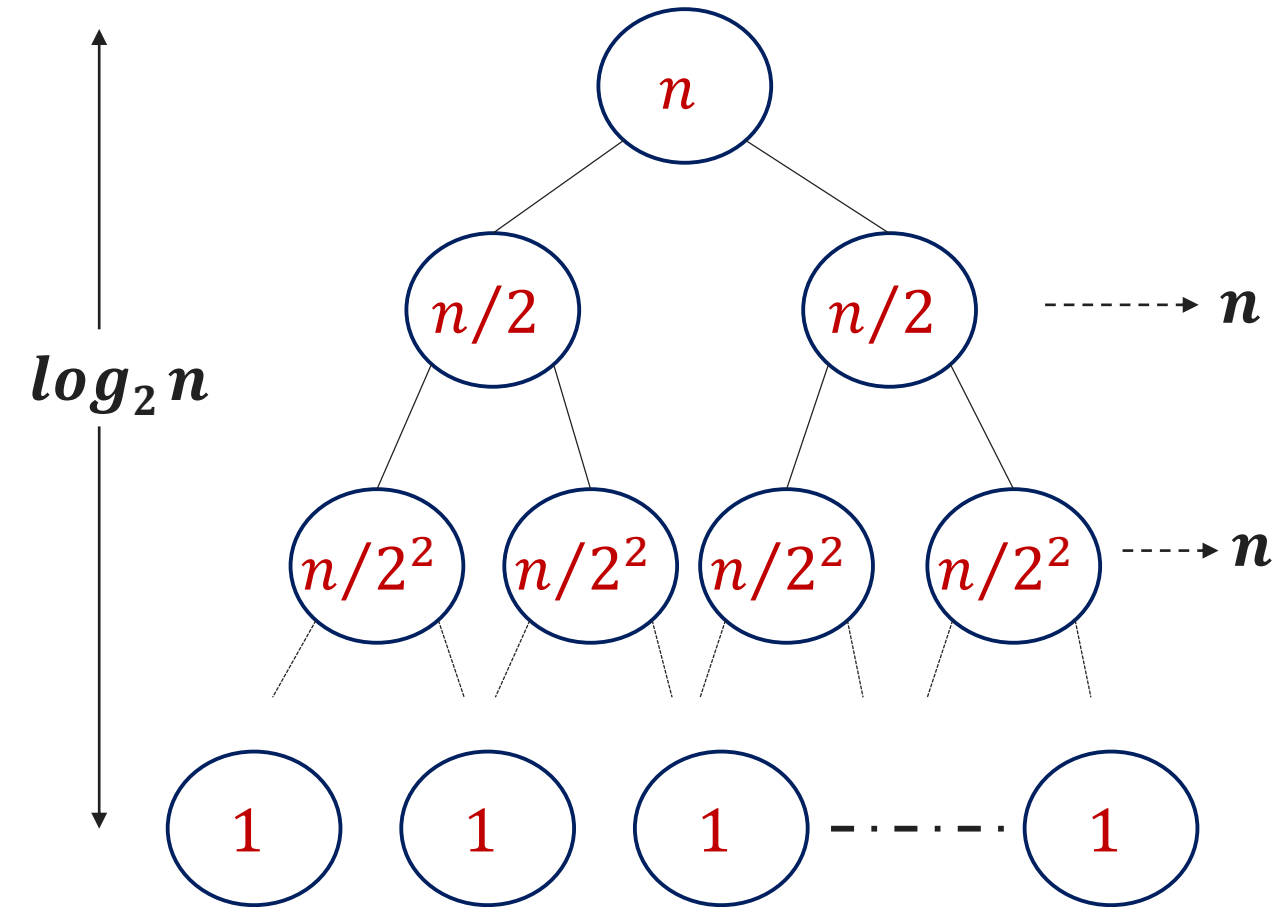
# Recurrence Tree Method

- ▶ In recurrence tree, each node represents the **cost of a single sub-problem** in the set of recursive function invocations.
- ▶ We sum the **costs within each level** of the tree to obtain a set of per level costs.
- ▶ Then we sum the all the **per level costs** to determine the total cost of all levels of the recursion.
- ▶ Here while solving recurrences, we **divide the problem** into sub-problems of equal size.
- ▶ E.g.,  $T(n) = a T(n/b) + f(n)$  where  $a > 1$ ,  $b > 1$  and  $f(n)$  is a given function.
- ▶  $F(n)$  is the cost of **splitting or combining** the sub problems.



# Recurrence Tree Method

The recursion tree for this recurrence is



## Example 1: $T(n) = 2T(n/2) + n$

- When we add the values across the levels of the recursion tree, we get a value of  $n$  for every level.
- The bottom level has  $2^{\log n}$  nodes, each contributing the cost  $T(1)$ .

- We have  $n + n + n + \dots$   $\log n$  times

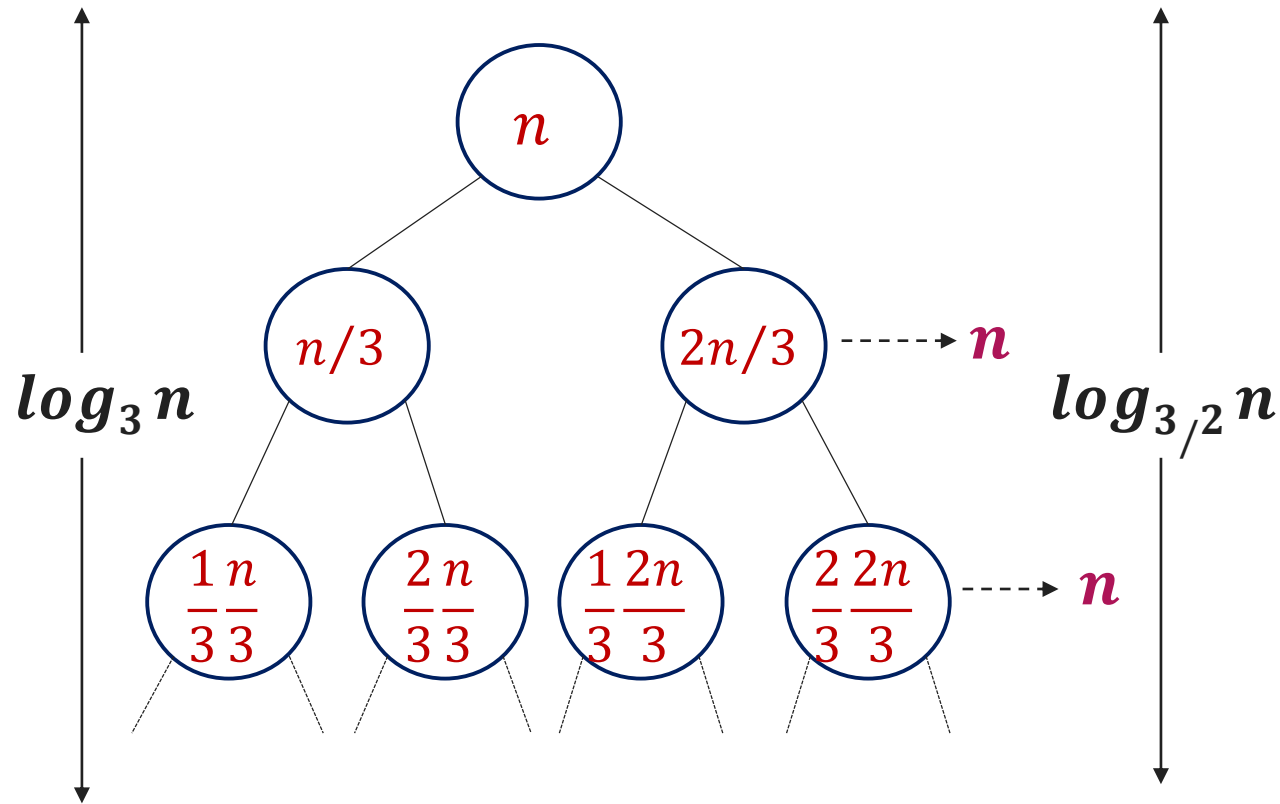
$$T(n) = \sum_{i=0}^{\log_2 n - 1} n + 2^{\log n} T(1)$$

$$T(n) = n \log n + n$$

$$T(n) = O(n \log n)$$

# Recurrence Tree Method

The recursion tree for this recurrence is



**Example 2:**  $T(n) = \underline{T(n/3)} + \underline{T(2n/3)} + n$

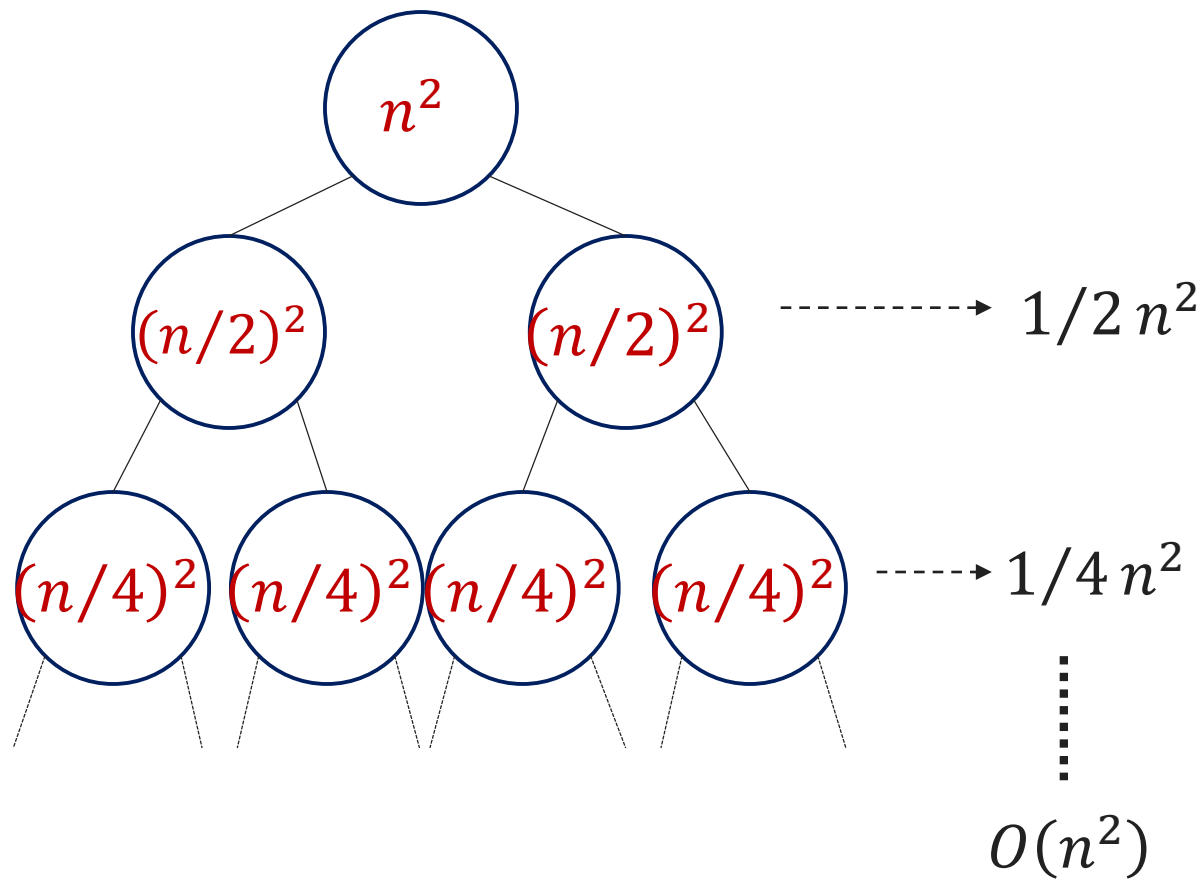
- When we add the values across the levels of the recursion tree, we get a value of  $n$  for every level.

$$T(n) = \sum_{i=0}^{\log_{3/2} n - 1} n + n^{\log_{3/2} 2} T(1)$$

$$T(n) \in n \log_{3/2} n$$

# Recurrence Tree Method

The recursion tree for this recurrence is



**Example 3:**  $T(n) = 2T(n/2) + c.n^2$

- Sub-problem size at level  $i$  is  $n/2^i$
- Cost of problem at level  $i$  is  $(n/2^i)^2$
- Total cost,

$$T(n) \leq n^2 \sum_{i=0}^{\log_2 n - 1} \left(\frac{1}{2}\right)^i$$

$$T(n) \leq n^2 \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i$$

$$T(n) \leq 2n^2$$

$$T(n) = O(n^2)$$



# Recurrence Tree Method - Exercises

- ▶ Example 1:  $T(n) = T(n/4) + T(3n/4) + c \cdot n$
- ▶ Example 2:  $T(n) = 3T(n/4) + c \cdot n^2$
- ▶ Example 3:  $T(n) = T(n/4) + T(n/2) + n^2$
- ▶ Example 4:  $T(n) = T(n/3) + T(2n/3) + n$

# **Divide & Conquer (D&C) Technique**

# Introduction

- ▶ Many useful algorithms are **recursive in structure**: to solve a given problem, they call themselves recursively one or more times.
- ▶ These algorithms typically follow a **divide-and-conquer** approach:
- ▶ The divide-and-conquer approach involves **three steps** at each level of the recursion:
  1. **Divide**: Break the problem into several sub problems that are similar to the original problem but smaller in size.
  2. **Conquer**: Solve the sub problems recursively. If the sub problem sizes are small enough, just solve the sub problems in a straightforward manner.
  3. **Combine**: Combine these solutions to create a solution to the original problem.

# D&C Running Time Analysis

- ▶ The **running-time analysis** of such divide-and-conquer (D&C) algorithms is almost automatic.
- ▶ Let  $g(n)$  be the **time required by D&C** on instances of size  $n$ .
- ▶ The **total time**  $t(n)$  taken by this divide-and-conquer algorithm is given by recurrence equation,

$$t(n) = lt(n/b) + g(n)$$

$$\mathbf{T(n) = aT(n/b) + f(n)}$$

- ▶ The solution of equation is given as,

$$t(n) = \begin{cases} \theta(n^k) & \text{if } l < b^k \\ \theta(n^k \log n) & \text{if } l = b^k \\ \theta(n^{\log_b l}) & \text{if } l > b^k \end{cases}$$

where  $k$  is the power of  $n$  in  $g(n)$

# Binary Search

# Introduction

- ▶ Binary Search is an extremely well-known instance of **divide-and-conquer** approach.
- ▶ Let  $T[1 \dots n]$  be an array of **increasing sorted order**; that is  $T[i] \leq T[j]$  whenever  $1 \leq i \leq j \leq n$ .
- ▶ Let  $x$  be some number. The problem consists of **finding**  $x$  in the array  $T$  if it is there.
- ▶ If  $x$  is not in the array, then we want to find **the position** where it might be inserted.

# Binary Search Example

Input: sorted array of integer values.  $x = 7$

1	3	7	9	11	32	52	74	90
---	---	---	---	----	----	----	----	----

Step 1:

1	2	3	4	5	6	7	8	9
1	3	7	9	11	32	52	74	90



Find approximate midpoint

# Binary Search Example

Step 2:

$x = 7$

1	2	3	4	5	6	7	8	9
1	3	7	9	11	32	52	74	90



Is  $7$  = midpoint value?

Step 3:

1	2	3	4	5	6	7	8	9
1	3	7	9	11	32	52	74	90



Search for the target in the area before midpoint.



# Binary Search Example

Step 4:

$x = 7$

1	2	3	4	5	6	7	8	9
1	3	7	9	11	32	52	74	90



Find approximate midpoint

Step 5:

1	2	3	4	5	6	7	8	9
1	3	7	9	11	32	52	74	90




$7 > \text{value of midpoint?}$  **Y**

# Binary Search Example

Step 6:

$x = 7$


1	2	3	4	5	6	7	8	9
1	3	7	9	11	32	52	74	90



Search for the  $x$  in the area after midpoint.

Step 7:

1	2	3	4	5	6	7	8	9
1	3	7	9	11	32	52	74	90



Find approximate midpoint.  
Is  $x$  = midpoint value?

# Binary Search – Iterative Algorithm

Algorithm: Function `biniter(T[1,...,n], x)`

if  $x > T[n]$  then return  $n+1$

$i \leftarrow 1$ ;

$j \leftarrow n$ ;

while  $i < j$  do

$k \leftarrow (i + j) \div 2$

    if  $x \leq T[k]$  then  $j \leftarrow k$

    else  $i \leftarrow k + 1$

return  $i$

$n = 7$

$x = 33$

i	3
	6
	7
k	11
	32
j	33
	53

# Binary Search – Recursive Algorithm

```
Algorithm: Function binsearch(T[1,...,n], x)
    if n = 0 or x > T[n] then return n + 1
    else return binrec(T[1,...,n], x)
Function binrec(T[i,...,j], x)
    if i = j then return i
    k ← (i + j) ÷ 2
    if x ≤ T[k] then
        return binrec(T[i,...,k], x)
    else return binrec(T[k + 1,...,j], x)
```

# Binary Search - Analysis

- ▶ Let  $t(n)$  be the time required for a call on  $\text{binrec}(T[i, \dots, j], x)$ , where  $n = j - i + 1$  is the number of elements **still under consideration** in the search.

- ▶ The recurrence equation is given as,

$$t(n) = t(n/2) + \theta(1)$$

$$T(n) = aT(n/b) + f(n)$$

- ▶ Comparing this to the general template for divide and conquer algorithm,  $a = 1, b = 2$  and  $f(n) = \theta(1)$ .

$$\therefore t(n) \in \theta(\log n)$$

- ▶ The complexity of binary search is  $\theta(\log n)$

- ▶ Example 2:  $T(n) = T(n/2) + \theta(1)$

- ▶ Here  $a = 1, b = 2$ . So,  $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$

- ▶  $f(n) = \theta(1) = 1$

- ▶ Case 2 applies: the solution is  $\theta(n^{\log_b a} \log n)$

- ▶  $T(n) = \theta(\log n)$

# Binary Search – Examples

1. Demonstrate binary search algorithm and find the element  $x = 12$  in the following array. [3 / 4]

2, 5, 8, 12, 16, 23, 38, 56, 72, 91

2. Explain binary search algorithm and find the element  $x = 31$  in the following array. [7]

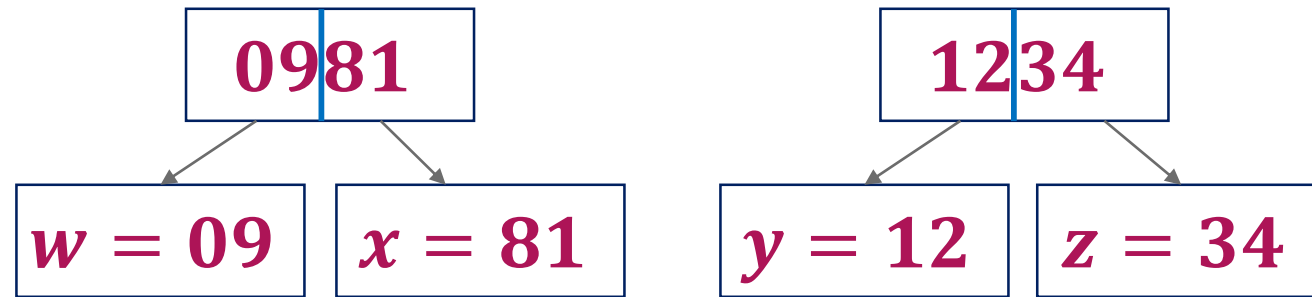
10, 15, 18, 26, 27, 31, 38, 45, 59

3. Let  $T[1..n]$  be a sorted array of distinct integers. Give an algorithm that can find an index  $i$  such that  $1 \leq i \leq n$  and  $T[i] = i$ , provided such an index exists. Prove that your algorithm takes time in  $O(\log n)$  in the worst case.

# Multiplying Large Integers

# Multiplying Large Integers – Introduction

- ▶ Multiplying two  $n$  digit large integers using **divide and conquer method**.
- ▶ Example: Multiplication of **981** by **1234**.
  1. Convert both the numbers into same length nos. and split each operand into two parts:



2. We can write as,

$$\begin{aligned} &10^2w + x \\ &= 10^2(09) + 81 \\ &= 900 + 81 \\ &= 981 \end{aligned}$$

$$0981 = 10^2w + x$$

$$1234 = 10^2y + z$$



# Multiplying Large Integers – Example 1

- ▶ Now, the required product can be computed as,

$$\begin{aligned} 0981 \times 1234 &= (10^2 w + x) \times (10^2 y + z) \\ &= 10^4 \underline{w \cdot y} + 10^2 (\underline{w \cdot z} + \underline{x \cdot y}) + \underline{x \cdot z} \\ &= 1080000 + 127800 + 2754 \\ &= 1210554 \end{aligned}$$

$$\begin{aligned} w &= 09 \\ x &= 81 \\ y &= 12 \\ z &= 34 \end{aligned}$$

- ▶ The above procedure still needs **four half-size multiplications**:

$$(i) w \cdot y \quad (ii) w \cdot z \quad (iii) x \cdot y \quad (iv) x \cdot z$$

- ▶ The computation of  $(\underline{w \cdot z} + \underline{x \cdot y})$  can be done as,

$$r = (w + x) \otimes (y + z) = \boxed{w \cdot y} + \boxed{w \cdot z + x \cdot y} + \boxed{x \cdot z}$$

- ▶ Only **one** multiplication is required instead of two.

Additional terms

# Multiplying Large Integers – Example 1

$$10^4 w \cdot y + 10^2 (w \cdot z + x \cdot y) + x \cdot z$$

$$\begin{aligned} w &= 09 \\ x &= 81 \\ y &= 12 \\ z &= 34 \end{aligned}$$

► Now we can compute the required product as follows:

$$p = w \cdot y = 09 \cdot 12 = 108$$

$$q = x \cdot z = 81 \cdot 34 = 2754$$

$$r = (w + x) \times (y + z) = 90 \cdot 46 = 4140$$

$$r = (w + x) \times (y + z) = w \cdot y + (w \cdot z + x \cdot y) + x \cdot z$$

$$981 \times 1234 = 10^4 p + 10^2 (r - p - q) + q$$

$$= 1080000 + 127800 + 2754$$

$$= 1210554.$$

# Multiplying Large Integers – Analysis

- ▶  $981 \times 1234$  can be reduced to **three multiplications** of two-figure numbers ( $09 \cdot 12$ ,  $81 \cdot 34$  and  $90 \cdot 46$ ) together with a certain number of shifts, additions and subtractions.
- ▶ Reducing four multiplications to three will enable us to cut 25% of the computing time required for large multiplications.
- ▶ We obtain an algorithm that can multiply two  $n$ -figure numbers in a time,

$$T(n) = 3t(n/2) + g(n),$$

$$T(n) = aT(n/b) + f(n)$$

- ▶ Solving it gives,

$$T(n) \in \theta(n^{\lg 3} \mid n \text{ is a power of } 2)$$

# Multiplying Large Integers – Example 2

- ▶ Example: Multiply **8114** with **7622** using divide & conquer method.
- ▶ Solution using D&C

Step 1:

$$w = 81$$

$$x = 14$$

$$y = 76$$

$$z = 22$$

Step 2:

Calculate  $p$ ,  $q$  and  $r$

$$p = w \cdot y = 81 \cdot 76 = 6156$$

$$q = x \cdot z = 14 \cdot 22 = 308$$

$$r = (w + x) \cdot (y + z) = 95 \cdot 98 = 9310$$

$$\begin{aligned} 8114 \times 7622 &= 10^4 p + 10^2 (r - p - q) + q \\ &= 61560000 + 284600 + 308 \\ &= 61844908 \end{aligned}$$

# Merge Sort

# Introduction

- ▶ Merge Sort is an example of **divide and conquer algorithm**.
- ▶ It is based on the **idea of breaking down a list into several sub-lists** until each sub list consists of a **single element**.
- ▶ **Merging those sub lists** in a manner that results into a sorted list.
- ▶ **Procedure**
  - ↪ Divide the unsorted list into  $N$  sub lists, each containing 1 element
  - ↪ Take adjacent pairs of two singleton lists and merge them to form a list of 2 elements.  $N$  will now convert into  $N/2$  lists of size 2
  - ↪ Repeat the process till a single sorted list of all the elements is obtained

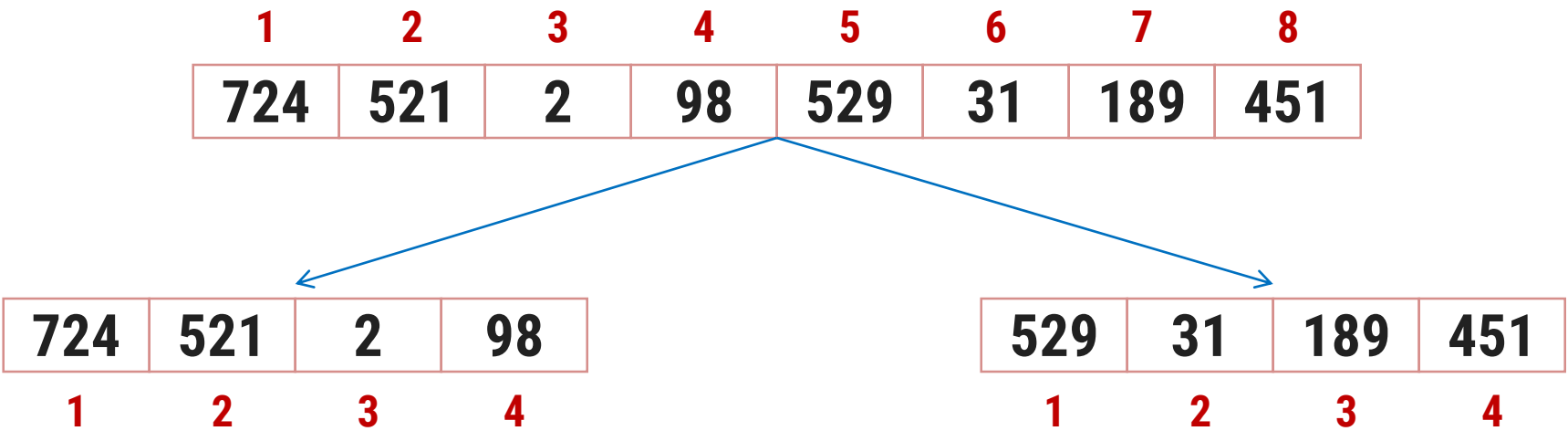
# Merge Sort – Example

## Unsorted Array

724	521	2	98	529	31	189	451
1	2	3	4	5	6	7	8

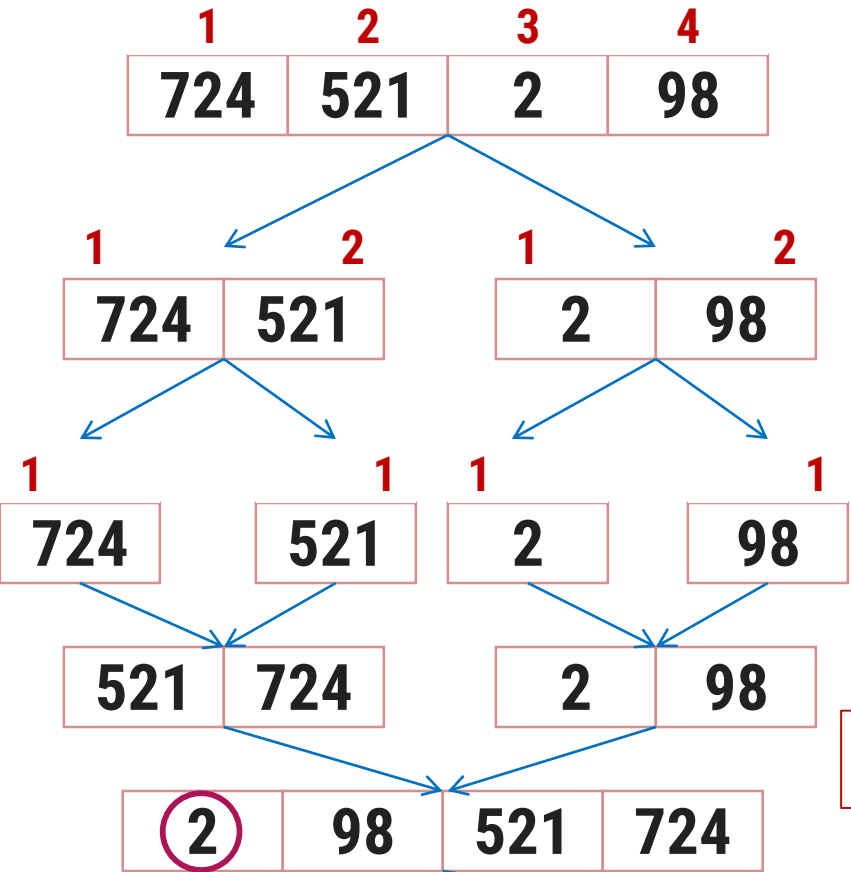
---

## Step 1: Split the selected array

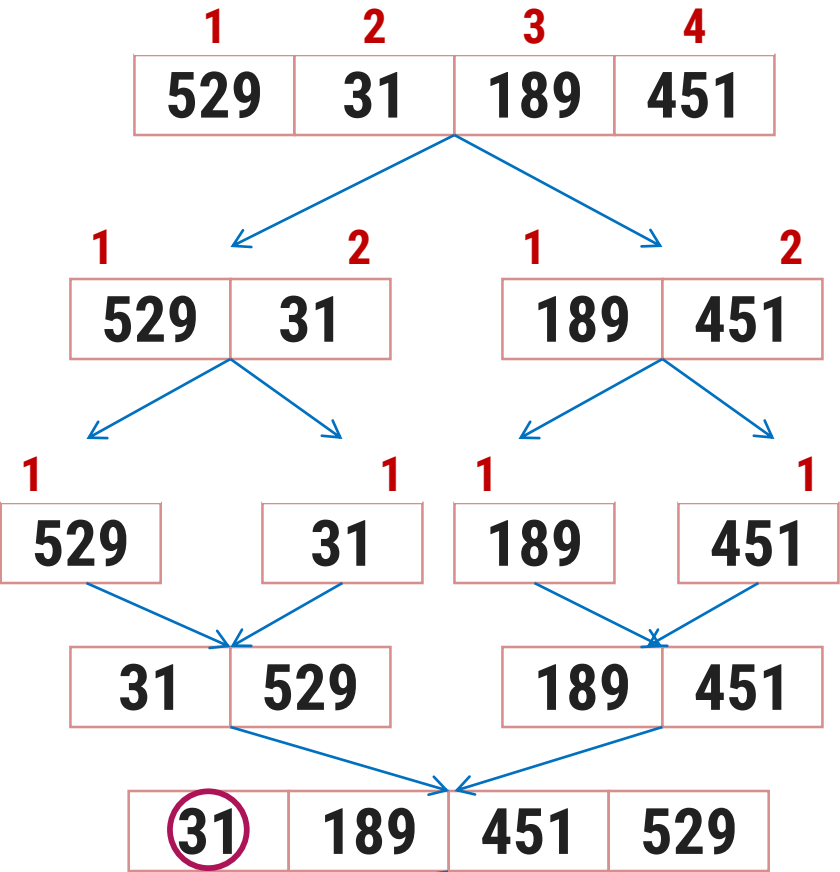


# Merge Sort – Example

Select the left subarray and Split



Select the right subarray and Split





# Merge Sort – Algorithm

```
Procedure: mergesort( $T[1, \dots, n]$ )
if  $n$  is sufficiently small then
  insert( $T$ )
else
  array  $U[1, \dots, 1+n/2], V[1, \dots, 1+n/2]$ 
     $U[1, \dots, n/2] \leftarrow T[1, \dots, n/2]$ 
     $V[1, \dots, n/2] \leftarrow T[n/2+1, \dots, n]$ 
    mergesort( $U[1, \dots, n/2]$ )
    mergesort( $V[1, \dots, n/2]$ )
  merge( $U, V, T$ )
```

```
Procedure:
merge( $U[1, \dots, m+1], V[1, \dots, n+1], T[1, \dots, m+n]$ )
 $i \leftarrow 1;$ 
 $j \leftarrow 1;$ 
 $U[m+1], V[n+1] \leftarrow \infty;$ 
for  $k \leftarrow 1$  to  $m + n$  do
  if  $U[i] < V[j]$ 
    then  $T[k] \leftarrow U[i];$ 
     $i \leftarrow i + 1;$ 
  else  $T[k] \leftarrow V[j];$ 
     $j \leftarrow j + 1;$ 
```

# Merge Sort - Analysis

- ▶ Let  $T(n)$  be the time taken by this algorithm to sort an array of  $n$  elements.
- ▶ Separating  $T$  into  $U$  &  $V$  takes **linear time**;  $merge(U, V, T)$  also takes **linear time**.

$$T(n) = T(n/2) + T(n/2) + g(n) \quad \text{where } g(n) \in \theta(n).$$

$$T(n) = 2t(n/2) + \theta(n)$$

$$t(n) = lt(n/b) + g(n)$$

- ▶ Applying the general case,  $l = 2, b = 2, k = 1$
- ▶ Since  $l = b^k$  the **second case** applies so,  $t(n) \in \theta(n \log n)$ .
- ▶ Time complexity of merge sort is  $\theta(n \log n)$ .

$$t(n) = \begin{cases} \theta(n^k) & \text{if } l < b^k \\ \theta(n^k \log n) & \text{if } l = b^k \\ \theta(n^{\log_b l}) & \text{if } l > b^k \end{cases}$$

# Strassen's Algorithm for Matrix Multiplication

# Matrix Multiplication

- ▶ Multiply following two matrices. Count how many scalar multiplications are required.

$$\begin{bmatrix} 1 & 3 \\ 7 & 5 \end{bmatrix} \cdot \begin{bmatrix} 6 & 8 \\ 4 & 2 \end{bmatrix}$$

$$answer = \begin{bmatrix} 1 \times 6 + 3 \times 4 & 1 \times 8 + 3 \times 2 \\ 7 \times 6 + 5 \times 4 & 7 \times 8 + 5 \times 2 \end{bmatrix}$$

- ▶ To multiply  $2 \times 2$  matrices, total 8 ( $2^3$ ) scalar multiplications are required.

# Matrix Multiplication

- ▶ In general,  $A$  and  $B$  are two  $2 \times 2$  matrices to be multiplied.

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \text{ and } B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$$

$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$$

$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$$

$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$$

- ▶ Computing each entry in the product takes  **$n$  multiplications** and there are  **$n^2$  entries** for a total of  **$O(n^3)$** .

# Strassen's Algorithm for Matrix Multiplication

- ▶ Consider the problem of **multiplying** two  $n \times n$  matrices.
- ▶ Strassen's devised a better method which has the **same basic method** as the multiplication of long integers.
- ▶ The main idea is **to save one multiplication** on a small problem and then use recursion.

# Strassen's Algorithm for Matrix Multiplication

## Step 1

$$\begin{aligned}S_1 &= B_{12} - B_{22} \\S_2 &= A_{11} + A_{12} \\S_3 &= A_{21} + A_{22} \\S_4 &= B_{21} - B_{11} \\S_5 &= A_{11} + A_{22} \\S_6 &= B_{11} + B_{22} \\S_7 &= A_{12} - A_{22} \\S_8 &= B_{21} + B_{22} \\S_9 &= A_{11} - A_{21} \\S_{10} &= B_{11} + B_{12}\end{aligned}$$

## Step 2

$$\begin{aligned}P_1 &= A_{11} \odot S_1 \\P_2 &= S_2 \odot B_{22} \\P_3 &= S_3 \odot B_{11} \\P_4 &= A_{22} \odot S_4 \\P_5 &= S_5 \odot S_6 \\P_6 &= S_7 \odot S_8 \\P_7 &= S_9 \odot S_{10}\end{aligned}$$

All above  
operations  
involve only **one**  
**multiplication.**

## Step 3

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{21} \end{bmatrix} \text{ and } B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Final Answer:

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

Where,

$$C_{11} = P_5 + P_4 - P_2 + P_6$$

$$C_{12} = P_1 + P_2$$

$$C_{21} = P_3 + P_4$$

$$C_{22} = P_5 + P_1 - P_3 - P_7$$

No multiplication is  
required here.

# Strassen's Algorithm - Analysis

- ▶ It is therefore possible to multiply two  $2 \times 2$  matrices using only **seven scalar multiplications**.
- ▶ Let  $t(n)$  be the time needed to multiply two  $n \times n$  matrices by **recursive use of equations**.

$$t(n) = 7t(n/2) + g(n)$$

$$t(n) = lt(n/b) + g(n)$$

Where  $g(n) \in O(n^2)$ .

- ▶ The general equation applies with  $l = 7, b = 2$  and  $k = 2$ .
- ▶ Since  $l > b^k$ , the **third case** applies and  $t(n) \in O(n^{lg7})$ .
- ▶ Since  $lg7 > 2.81$ , it is possible to multiply two  $n \times n$  matrices in a time  $O(n^{2.81})$ .

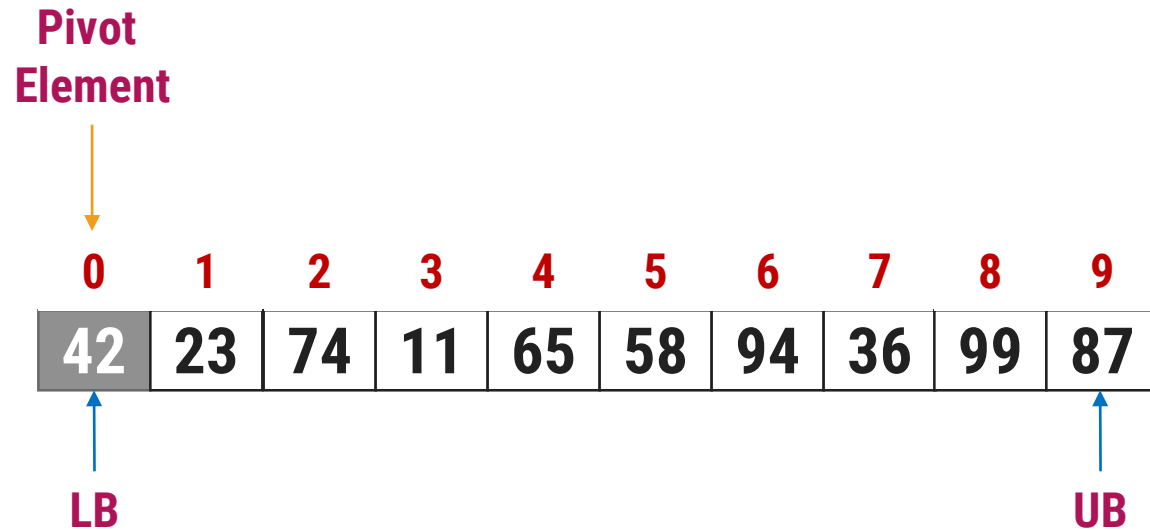
$$t(n) = \begin{cases} \theta(n^k) & \text{if } l < b^k \\ \theta(n^k \log n) & \text{if } l = b^k \\ \theta(n^{\log_b l}) & \text{if } l > b^k \end{cases}$$



# Quick Sort

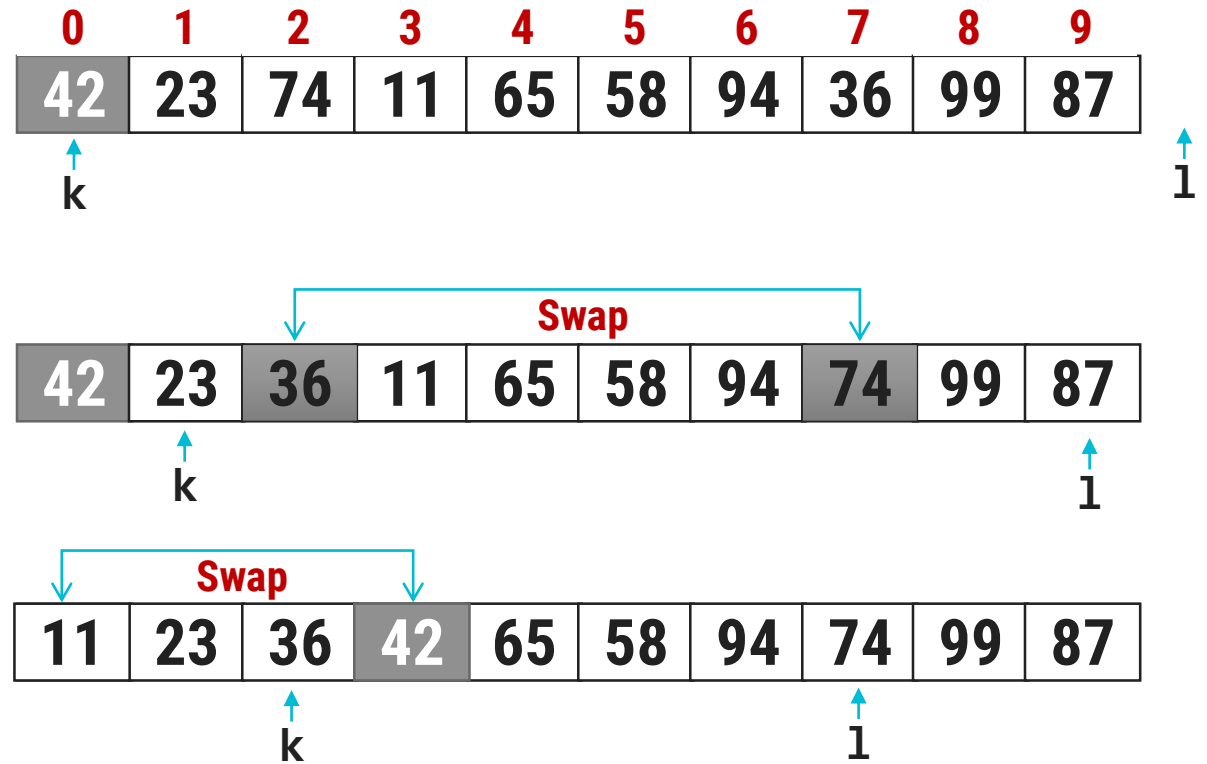
# Introduction

- ▶ Quick sort chooses the first element as a **pivot element**, a **lower bound is the first index** and an **upper bound is the last index**.
- ▶ The array is then **partitioned** on either side of the **pivot**.
- ▶ Elements are moved so that, those **greater** than the **pivot** are shifted to its **right** whereas the others are shifted to its **left**.
- ▶ Each Partition is **internally sorted recursively**.



# Quick Sort - Example

```
Procedure pivot(T[i,...,j]; var l)
p ← T[i]
k ← i; l ← j+1
Repeat
  k ← k+1 until T[k] > p or k ≥ j
Repeat
  l ← l-1 until T[l] ≤ p
While k < l do
  Swap T[k] and T[l]
  Repeat k ← k+1 until T[k] > p
  Repeat l ← l-1 until T[l] ≤ p
Swap T[i] and T[l]
```



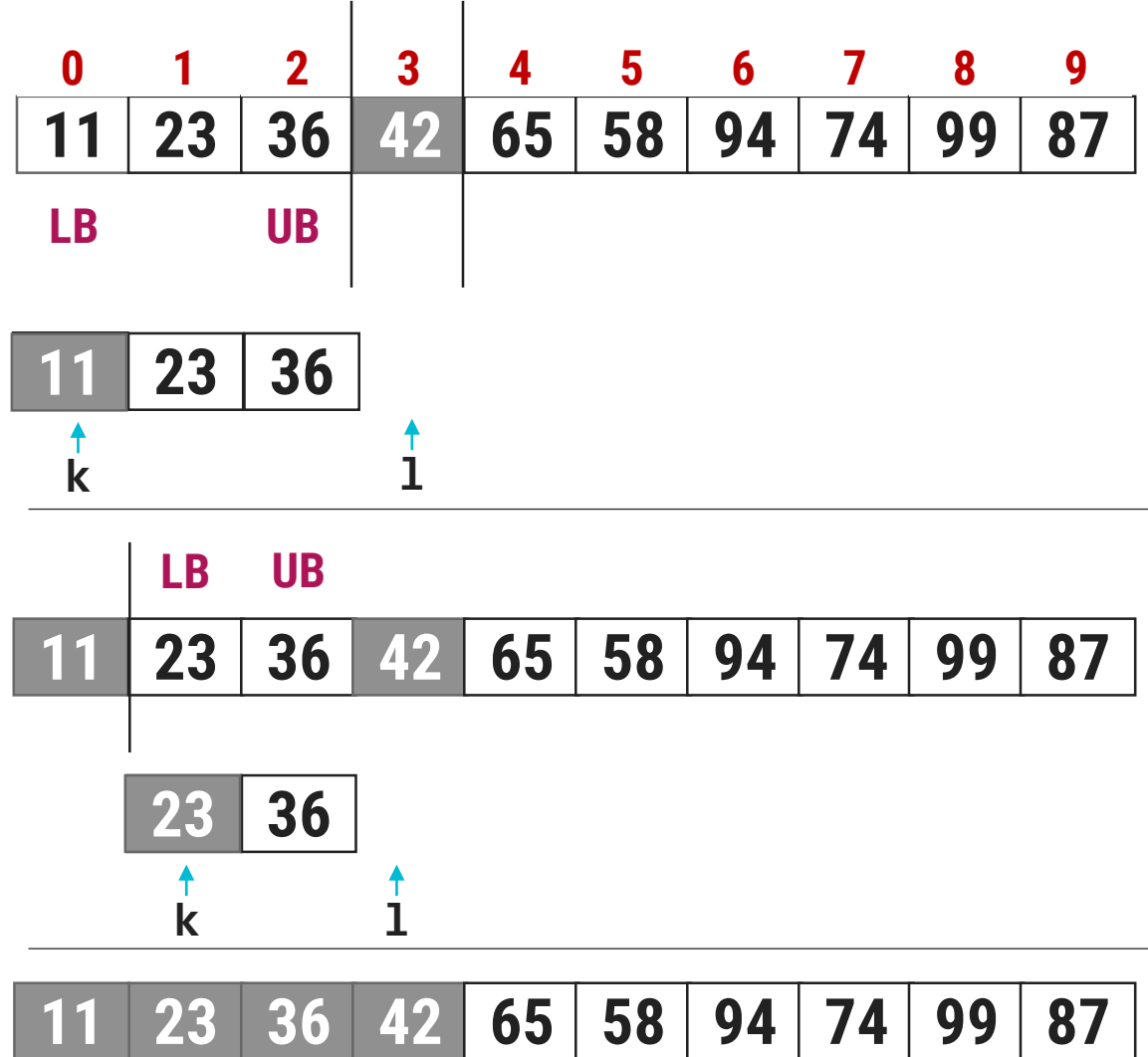
LB = 0, UB = 9

p = 42

k = 0, l = 10

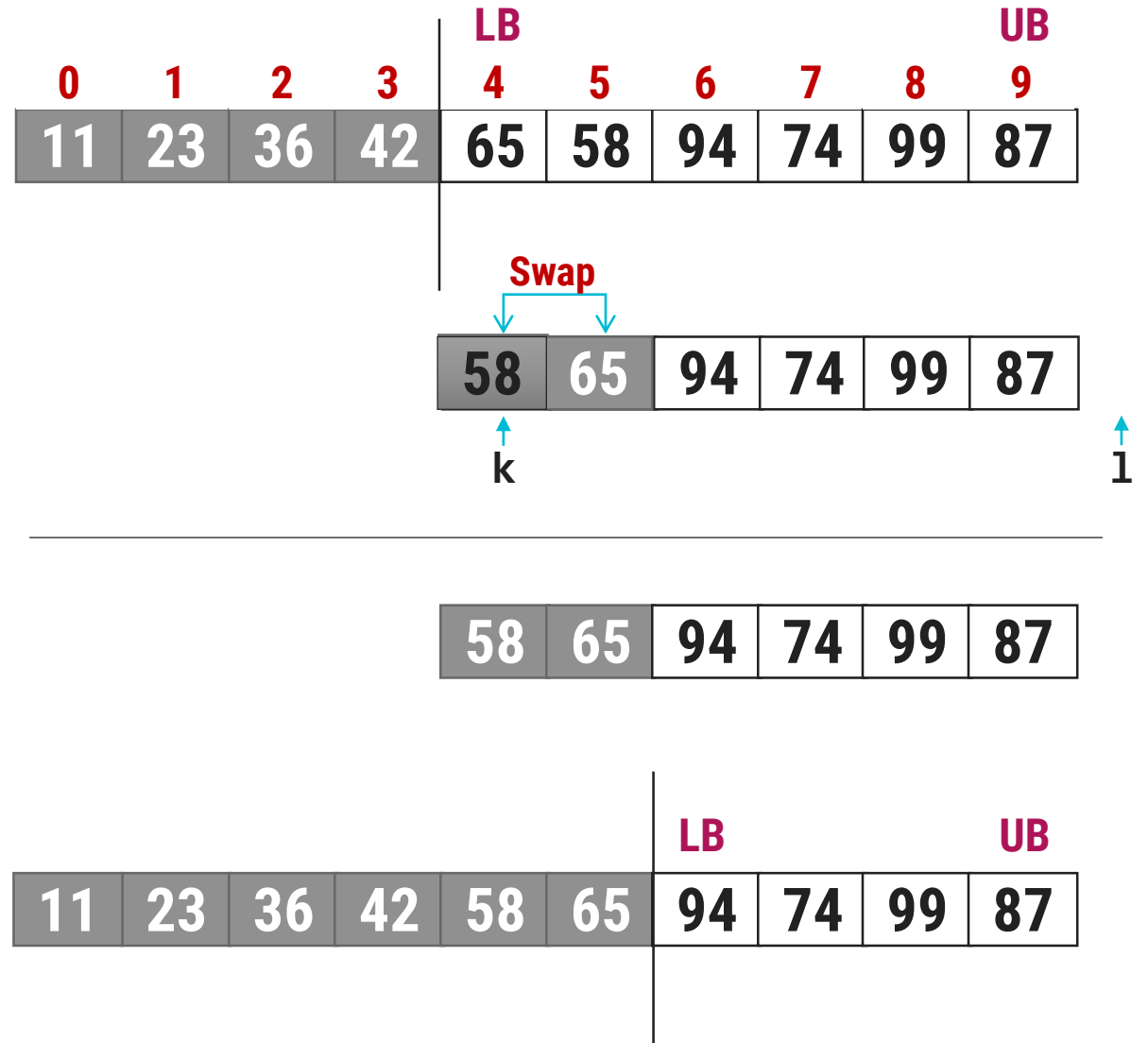
# Quick Sort - Example

```
Procedure pivot(T[i,...,j]; var l)
p ← T[i]
k ← i; l ← j+1
Repeat
  k ← k+1 until T[k] > p or k ≥ j
Repeat
  l ← l-1 until T[l] ≤ p
While k < l do
  Swap T[k] and T[l]
  Repeat k ← k+1 until
    T[k] > p
  Repeat l ← l-1 until
    T[l] ≤ p
Swap T[i] and T[l]
```



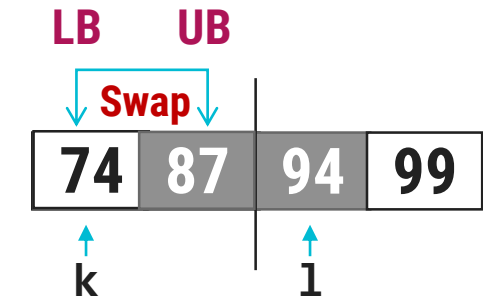
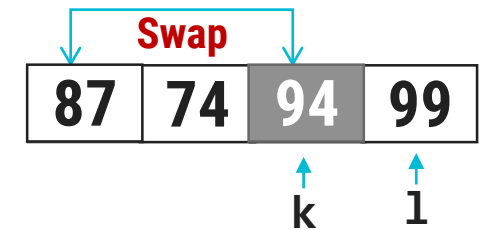
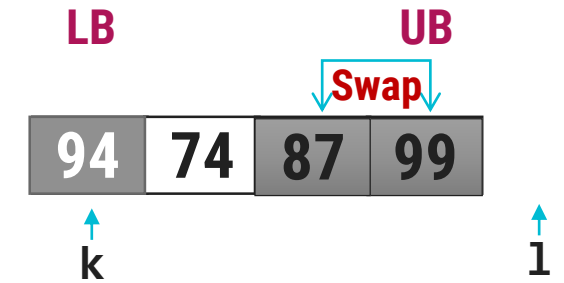
# Quick Sort - Example

```
Procedure pivot(T[i,...,j]; var l)
p ← T[i]
k ← i; l ← j+1
Repeat
  k ← k+1 until T[k] > p or k ≥ j
Repeat
  l ← l-1 until T[l] ≤ p
While k < l do
  Swap T[k] and T[l]
  Repeat k ← k+1 until
    T[k] > p
  Repeat l ← l-1 until
    T[l] ≤ p
Swap T[i] and T[l]
```



# Quick Sort - Example

```
Procedure pivot(T[i,...,j]; var l)
p ← T[i]
k ← i; l ← j+1
Repeat
  k ← k+1 until T[k] > p or k ≥ j
Repeat
  l ← l-1 until T[l] ≤ p
While k < l do
  Swap T[k] and T[l]
  Repeat k ← k+1 until
    T[k] > p
  Repeat l ← l-1 until
    T[l] ≤ p
Swap T[i] and T[l]
```



# Quick Sort - Algorithm

```
Procedure: quicksort(T[i,...,j])
{Sorts subarray T[i,...,j] into
ascending order}
if j - i is sufficiently small
then insert (T[i,...,j])
else
    pivot(T[i,...,j],l)
    quicksort(T[i,...,l - 1])
    quicksort(T[l+1,...,j])
```

```
Procedure: pivot(T[i,...,j]; var l)
p ← T[i]
k ← i
l ← j + 1
repeat k ← k+1 until T[k] > p or k ≥ j
repeat l ← l-1 until T[l] ≤ p
while k < l do
    Swap T[k] and T[l]
    Repeat k ← k+1 until T[k] > p
    Repeat l ← l-1 until T[l] ≤ p
Swap T[i] and T[l]
```

# Quick Sort Algorithm – Analysis

## 1. Worst Case

- Running time depends on **which element is chosen as key or pivot** element.
- The worst case behavior for quick sort occurs when the array is partitioned into one sub-array with  $n - 1$  **elements and the other with 0 element**.
- In this case, the recurrence will be,

$$T(n) = T(n - 1) + T(0) + \theta(n)$$

$$T(n) = T(n - 1) + \theta(n)$$

$$T(n) = \theta(n^2)$$

## 2. Best Case

- **Occurs when partition produces sub-problems each of size  $n/2$ .**
- Recurrence equation:

$$T(n) = 2T(n/2) + \theta(n)$$

$$l = 2, b = 2, k = 1, \text{ so } l = b^k$$

$$T(n) = \theta(n \log n)$$



# Quick Sort Algorithm – Analysis

## 3. Average Case

- Average case running time is much closer to the best case.
- If suppose the partitioning algorithm produces a **9:1 proportional** split the recurrence will be,

$$T(n) = T(9n/10) + T(n/10) + \theta(n)$$

$$T(n) = \theta(n \log n)$$

# Quick Sort - Examples

► Sort the following array in ascending order using quick sort algorithm.

1. 5, 3, 8, 9, 1, 7, 0, 2, 6, 4

2. 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9

3. 9, 7, 5, 11, 12, 2, 14, 3, 10, 6

# Exponentiation

# Exponentiation - Sequential

- ▶ Let  $a$  and  $n$  be two integers. We wish to compute the **exponentiation**  $x = a^n$ .
- ▶ Algorithm using **Sequential Approach**:

```
function exposeq(a, n)
    r ← a
    for i ← 1 to n - 1 do
        r ← a * r
    return r
```

- ▶ This algorithm takes a time in  $\theta(n)$  since the instruction  $r = a * r$  is executed exactly  $n - 1$  **times**, provided the multiplications are counted as elementary operations.

# Exponentiation - Sequential

- ▶ But **to handle larger operands**, we must consider the time required for each multiplication.
- ▶ Let  $m$  is the size of operand  $a$ .
- ▶ Therefore, the multiplication performed the  $i^{th}$  time round the loop concerns **an integer of size  $m$  and an integer whose size is between  $im - i + 1$  and  $im$** , which takes a time between

$$M(m, im - i + 1) \text{ and } M(m, im)$$

$a = 5$  so  $m = 1$  and  $n = 25$  and suppose  $i = 10$

The body of loop executes  $10^{th}$  time as,

$$r = a * r$$

here 9 times multiplication is already done so  $r = 5^9 = 1953125$

The size of  $r$  in the  $10^{th}$  iteration will be between  $im - i + 1$  to  $im$ , i.e.,  
between **1 to 10**

10-10+1

10

# Exponentiation - Sequential

- ▶ The total time  $T(m, n)$  spent multiplying when computing  $a^n$  with **exposeq** is therefore,

$$\sum_{i=1}^{n-1} M(m, im - 1 + 1) \leq T(m, n) \leq \sum_{i=1}^{n-1} M(m, im)$$

$$T(m, n) \leq \sum_{i=1}^{n-1} M(m, im) \leq \sum_{i=1}^{n-1} cm \cdot im$$

$$cm^2 \sum_{i=1}^{n-1} i \leq cm^2 n^2 = \theta(m^2 n^2)$$

- ▶ If we use the **divide-and-conquer** multiplication algorithm,

$$T(m, n) \in \theta(m^{\lg 3} n^2)$$

# Exponentiation – D & C

► Suppose, we want to compute  $a^{10}$

► We can write as,

$$a^{10} = (a^5)^2 = (a \cdot a^4)^2 = (a \cdot (a^2)^2)^2$$

► In general,

$$a^n = \begin{cases} a & \text{if } n = 1 \\ (a^{n/2})^2 & \text{if } n \text{ is even} \\ a \times a^{n-1} & \text{otherwise} \end{cases}$$

► Algorithm using **Divide & Conquer Approach**:

```
function expoDC(a, n)
```

```
    if n = 1 then return a
```

```
    if n is even then return [expoDC(a, n/2)]2
```

```
    return a * expoDC(a, n - 1)
```

# Exponentiation – D & C

Number of operations performed by the algorithm is given by,

$$N(n) = \begin{cases} 0 & \text{if } n = 1 \\ N(n/2) + 1 & \text{if } n \text{ is even} \\ N(n-1) + 1 & \text{otherwise} \end{cases}$$

Time taken by the algorithm is given by,

$$T(m, n) = \begin{cases} 0 & \text{if } n = 1 \\ T(m, n/2) + M(mn/2, mn/2) & \text{if } n \text{ is even} \\ T(m, n-1) + M(m, (n-1)m) & \text{otherwise} \end{cases}$$

Solving it gives,  $T(m, n) \in \theta(m^{\lg^3 n} n^{\lg^3})$

```
function expoDC(a, n)
```

```
    if n = 1 then return a
```

```
    if n is even then return [expoDC(a, n/2)]2
```

```
    return a * expoDC(a, n - 1)
```



# Exponentiation – Summary

	Multiplication	
	Classic	D&C
exposeq	$\theta(m^2 n^2)$	$\theta(m^{lg^3} n^2)$
expoDC	$\theta(m^2 n^2)$	$\theta(m^{lg^3} n^{lg^3})$

**Thank You**