



Dr. Vishwanath Karad

**MIT WORLD PEACE  
UNIVERSITY** | PUNE

TECHNOLOGY, RESEARCH, SOCIAL INNOVATION & PARTNERSHIPS

## CET2001B    **Advanced Data Structure**

---

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**



# CET2013A      Data Structures-II with C++ Programming

---

**Examination scheme: Marks-50 [Continuous Assessment]**

## **Course Objectives:**

1. To study algorithms related to trees and graphs.
2. To understand the concept of symbol table.
3. To realize appropriate data structures to solve problems in various domains.

## **Course Outcomes:**

1. To select appropriate data structures in problem solving
2. To implement various algorithms related to trees and graphs
3. To demonstrate the use of trees for symbol table implementation.

# Syllabus cont...

---

## Unit 3

**Heap-** Heap as a priority queue, Heap sort.

**Symbol Table-** Introduction to Symbol Tables, Static tree table- Optimal Binary Search Tree (OBST), Dynamic tree table-AVL tree, Multi way search tree- B-Tree.

# List of Assignments

---

1. Implement binary tree using C++ and perform following operations: Creation of binary tree and traversal (recursive and non- recursive)
2. Implement dictionary using binary search tree where dictionary stores keywords & its meanings. Perform following operations:
  - Insert a keyword
  - Delete a keyword
  - Create mirror image and display level wise
  - Copy
3. Implement threaded binary tree and perform inorder traversal.

## List of Assignments contd...

---

4. Consider a friend's network on Facebook social web site. Model it as a graph to represent each node as a user and a link to represent the friend relationship between them using adjacency list representation and perform DFS & BFS traversals.
5. A business house has several offices in different countries; they want to lease phone lines to connect them with each other and the phone company charges different rent to connect different pairs of cities. Business house wants to connect all its offices with a minimum total cost. Solve the problem using Prim's algorithm.
6. Store data of students with roll no, name and grade. Implement linear probing with and without replacement.
7. Read the marks obtained by students of second year in an online examination of particular subject. Find out maximum and minimum marks obtained in that subject. Use heap data structure.

# Learning Resources

---

## **Text Books:**

Horowitz, Sahani, Dinesh Mehta, “Fundamentals of Data Structures in C++”, Galgotia Publisher, ISBN: 8175152788, 9788175152786.

Peter Brass, “Advanced Data Structures”, Cambridge University Press, ISBN: 978-1-107-43982.

## **Reference Books:**

Sartaj Sahani, “Data Structures, Algorithms and Applications in C++”, Second Edition, University Press, ISBN: 81-7371522 X.

Augenstein ,Tenenbaum & Langsam,“Data Structure Using C & C++”,PHI Publication.

## **Supplementary Reading:**

Yashwant Kanitkar,” Data Structures through C++”, BPB Publication.

# Learning Resources contd...

---

## Web Resources:

<https://www.khanacademy.org/computing/computer-science/algorithms>

<https://www.hackerrank.com/contests/basic-ds-quiz-1/>

## Web links:

[https://www.tutorialspoint.com/data\\_structures\\_algorithms/](https://www.tutorialspoint.com/data_structures_algorithms/)

## MOOCs:

<http://nptel.ac.in/courses/106102064/1>

<https://nptel.ac.in/courses/106103069/>

# Heap

---

- Heap as a Data Structure
- Types of heap -Min heap and Max heap
- Operations on Heap
- Heap Sort
- Applications of Heap



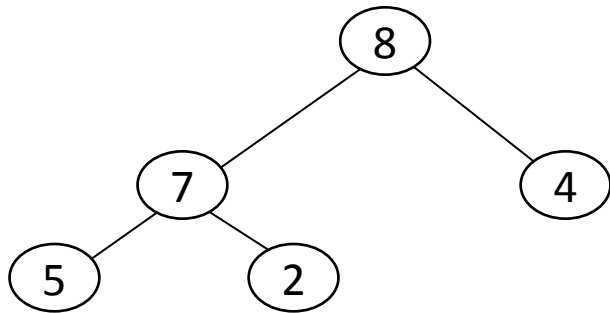
# The Heap Data Structure

*Def:* A **max** (min) heap is a tree in which the **key value** in each node is **no smaller** (larger) than the key values in its **children** (if any).

- A max heap is a complete binary tree that is also a max tree.
- A min heap is a complete binary tree that is also a min tree.

**Order (heap) property: for any node x (for Max heap)**

$$\text{Parent}(x) \geq x$$



Heap

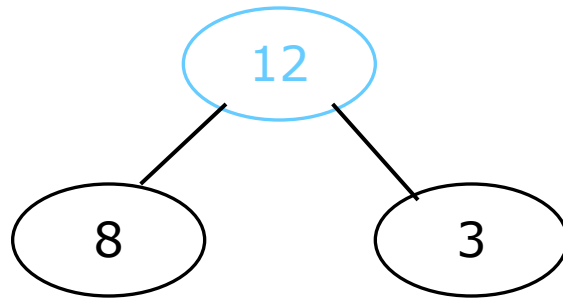
Example of Max heap

**“The root is the maximum element of the heap!”**

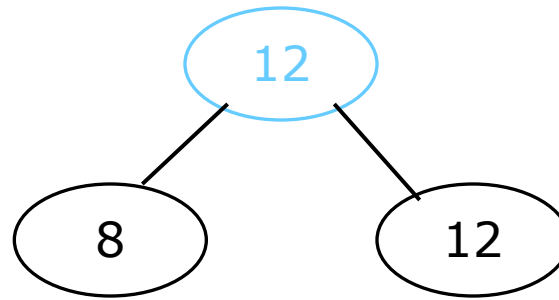
**A heap is a binary tree that is filled in order**

# The heap property (for Max heap)

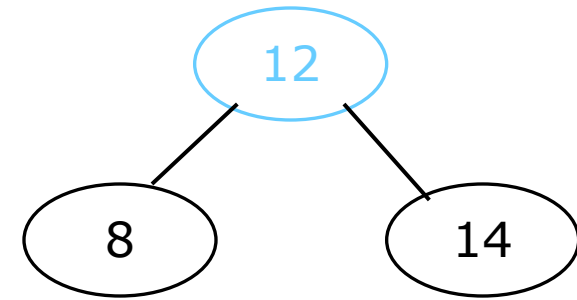
A node has the **heap property** if the value in the node is as large as or larger than the values in its children



Blue node has heap property



Blue node has heap property



Blue node does not have heap property

All leaf nodes automatically have the heap property

A binary tree is a **heap** if *all* nodes in it have the heap property

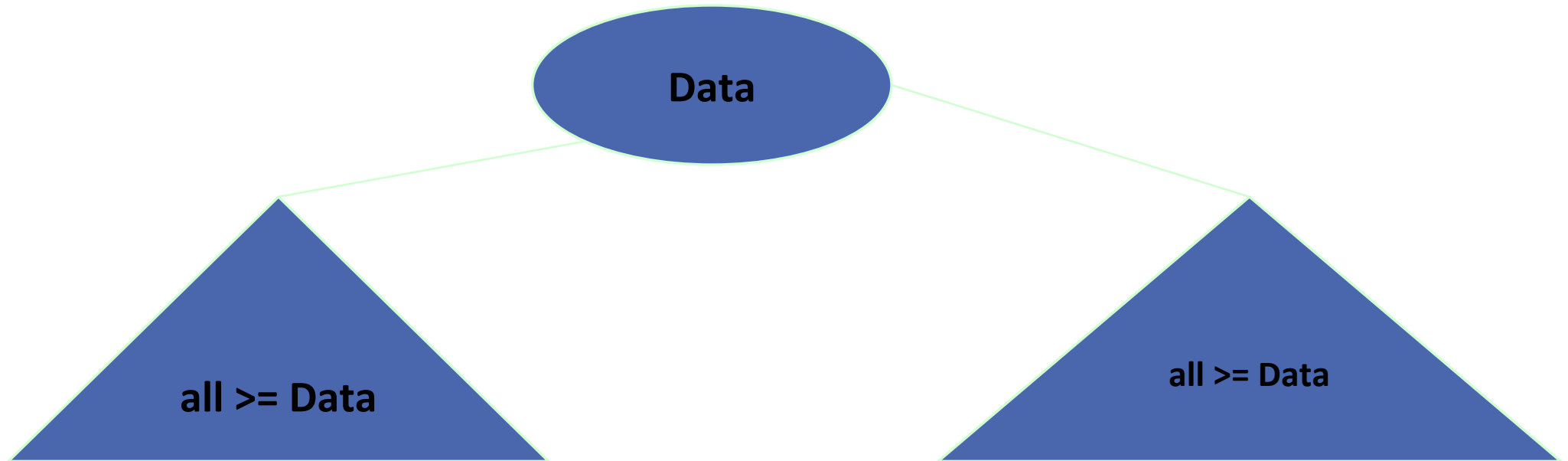
# Types of Heap

---

◆ **Min-heap**

◆ **Max-heap**

# Min Heap



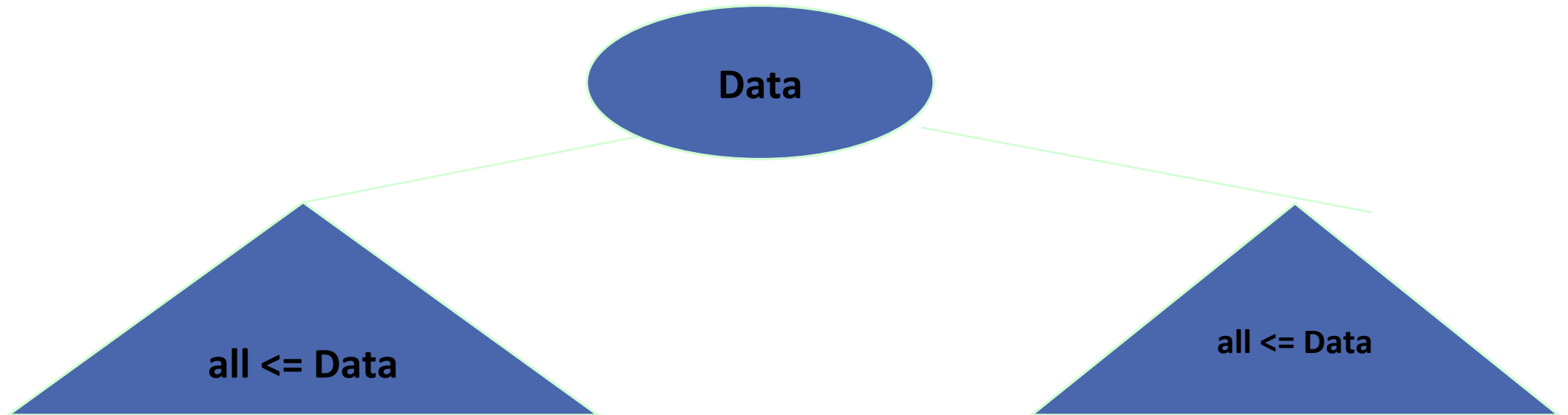
# Min-Heap

---

- ◆ In min-heap, the key value of each node is lesser than or equal to the key value of its children
- ◆ In addition, every path from root to leaf should be sorted in ascending order

# Max Heap

---



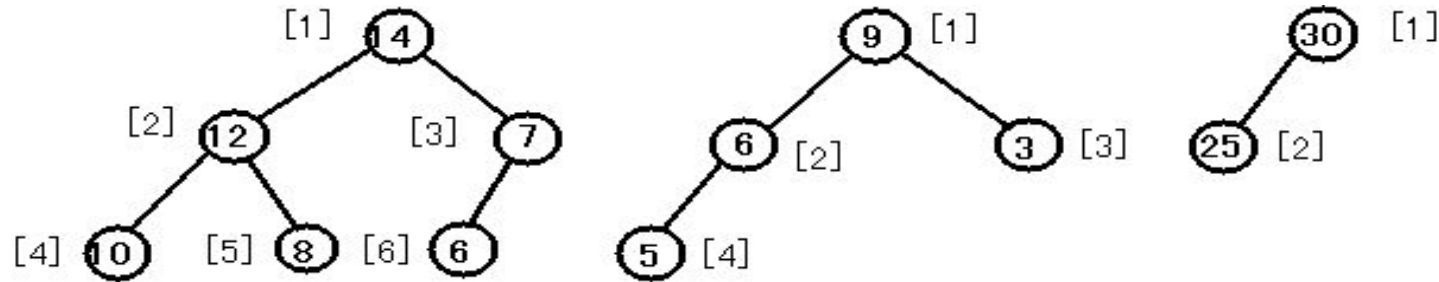
# Max-heap

---

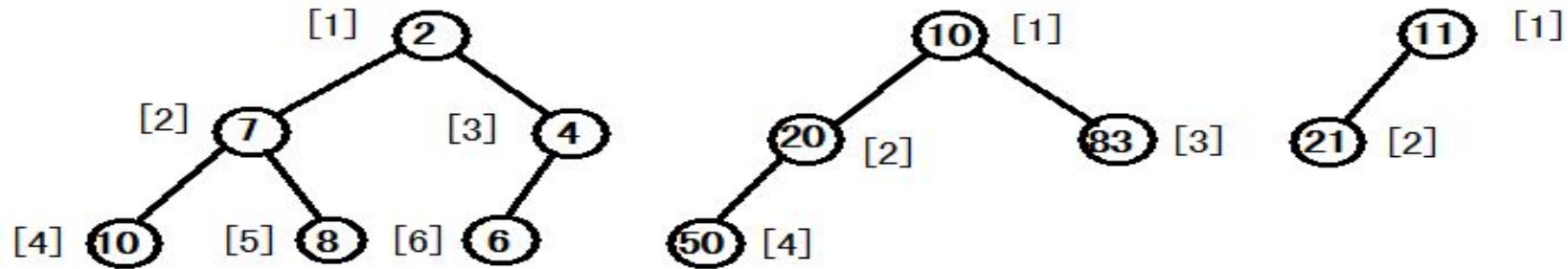
- ◆ A max-heap is where the key value of a node is greater than the key values in of its children

# Example of Heap

**Max-Heap**

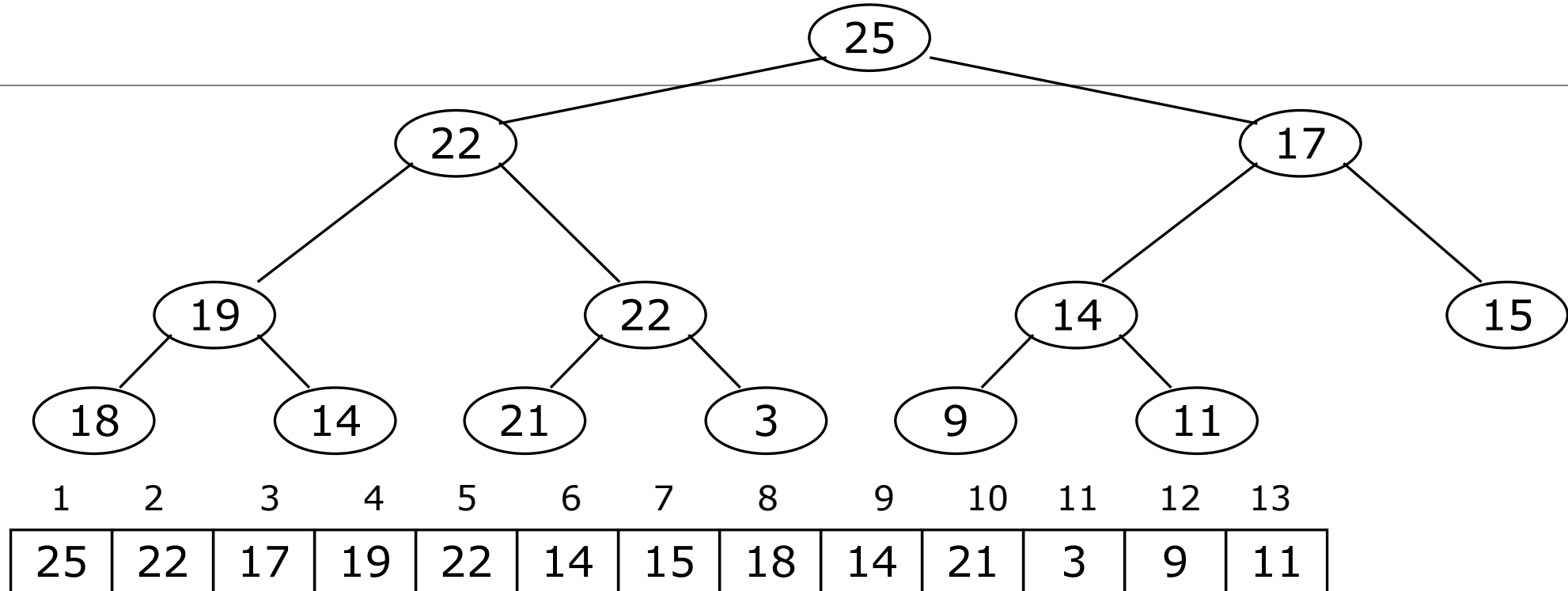


**Min-Heap**





# Mapping into an array



Notice:

- The left child of index  $i$  is at index  $2*i$  (index starting from 1)
- The right child of index  $i$  is at index  $2*i+1$
- Example: the children of node value (19) [index 4] are at [index 8] (18) and [index 9] (14)

# Operations on Heaps

---

- ◆ **Create**—To create an empty heap to which ‘root’ points
- ◆ **Insert**—To insert an element into the heap
- ◆ **Delete**—To delete max (or min) element from the heap
- ◆ **ReheapUp**—To rebuild heap when we use the insert() function
- ◆ **ReheapDown**—To build heap when we use the delete() function

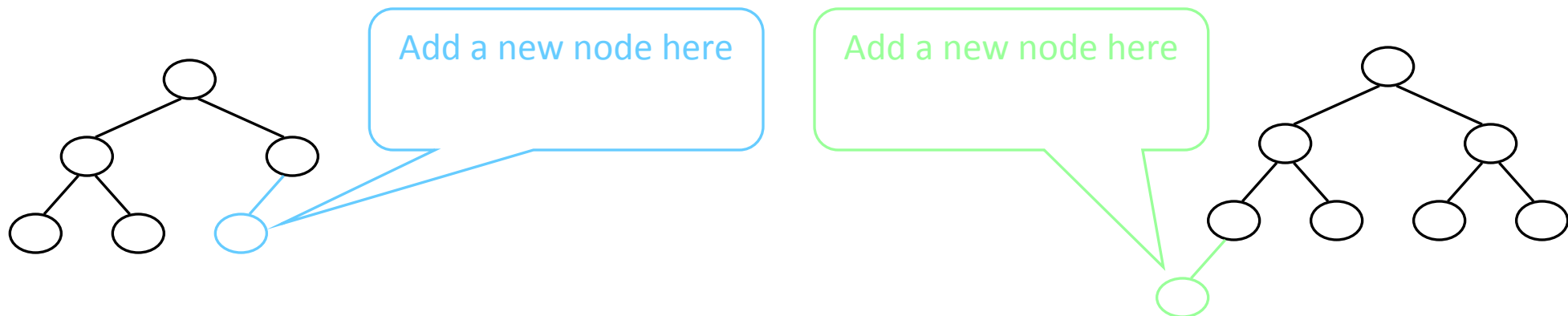
# Constructing a (max) heap

A tree consisting of a single node is automatically a heap

We construct a heap by adding nodes one at a time:

- Add the node just to the right of the rightmost node in the deepest level
- If the deepest level is full, start a new level

Examples:



# Constructing a (max)heap

contd...

---

Each time we add a node, we may destroy the heap property of its parent node

To fix this, we shift up

But each time we shift up, the value of the topmost node in the shift may increase, and this may destroy the heap property of *its* parent node

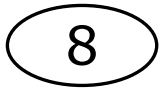
We repeat the shifting up process, moving up in the tree, until either

We reach nodes whose values don't need to be swapped (because the parent is *still* larger than both children), or

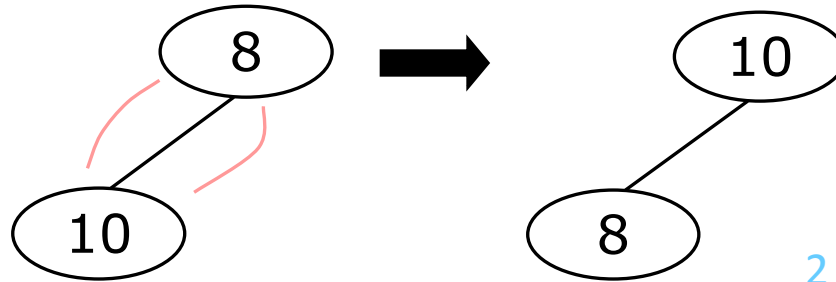
- We reach the root

# Constructing a heap

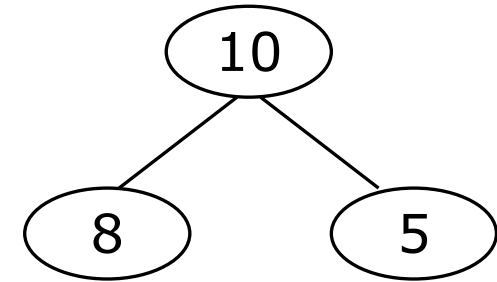
contd...



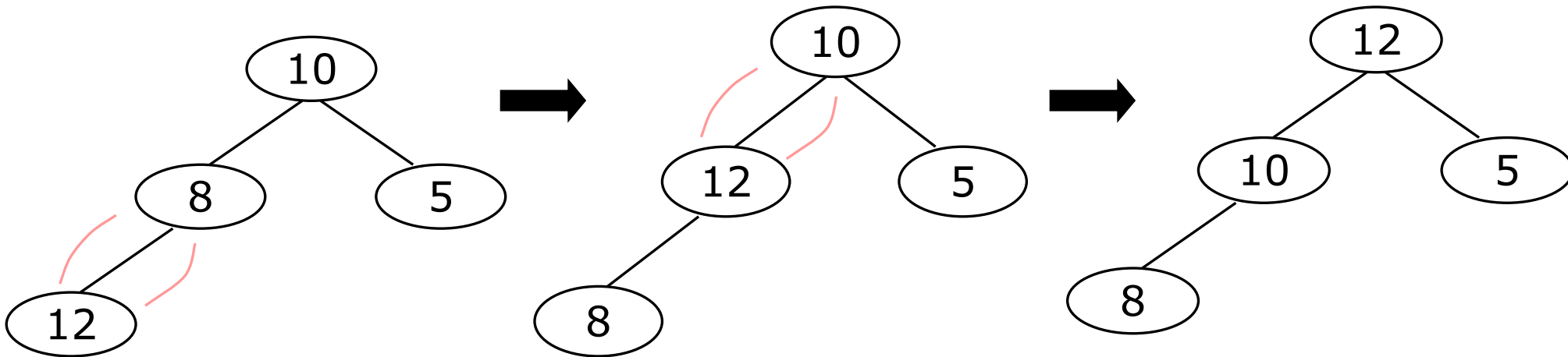
1



2

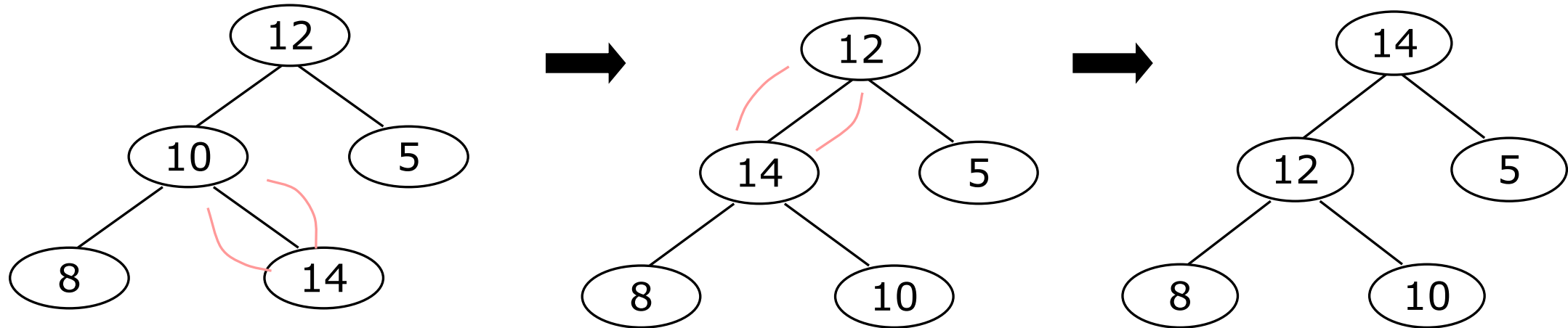


3



4

# Other children are not affected



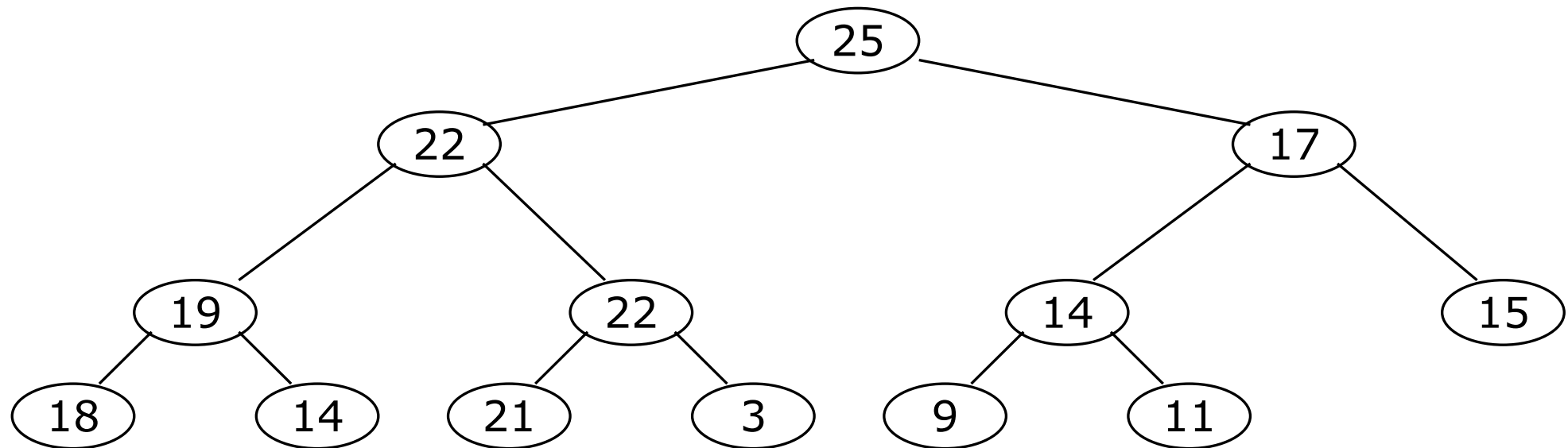
The node containing 8 is not affected because its parent gets larger, not smaller

The node containing 5 is not affected because its parent gets larger, not smaller

The node containing 8 is still not affected because, although its parent got smaller, its parent is still greater than it was originally

# A sample heap

Here's a sample binary tree after it has been heapified

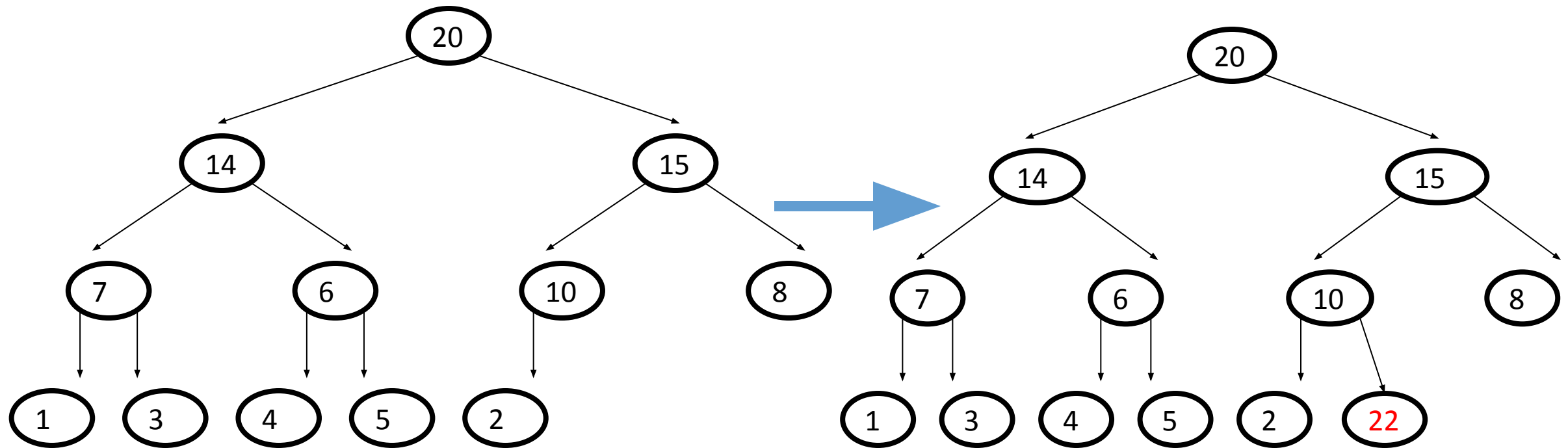


Notice that heapified does *not* mean sorted

Heapifying does *not* change the shape of the binary tree; this binary tree is balanced and left-justified because it started out that way

# Insert (max heap)

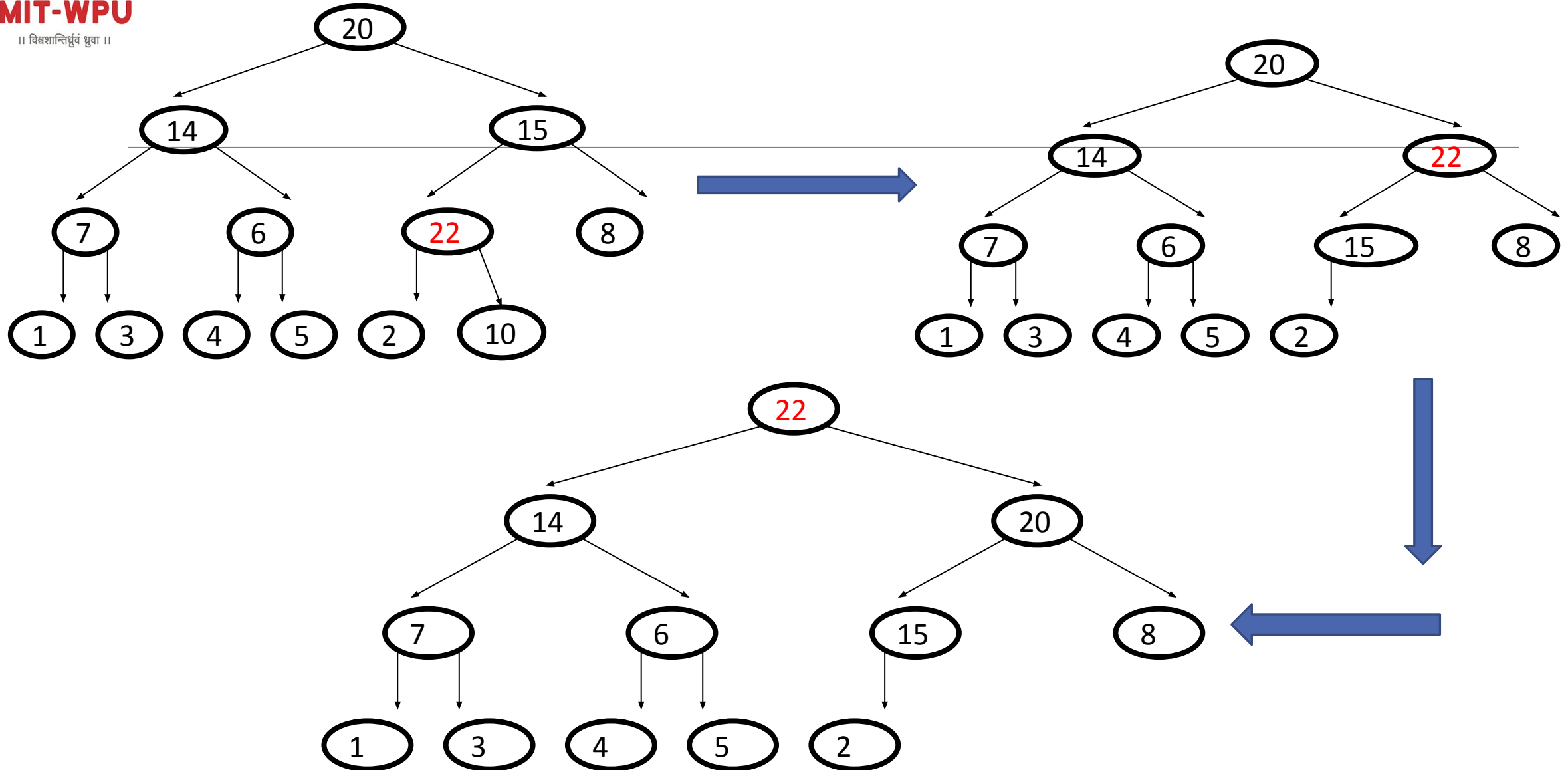
`insert(22)`





# Insert (max heap)

contd...



### Algorithm add()

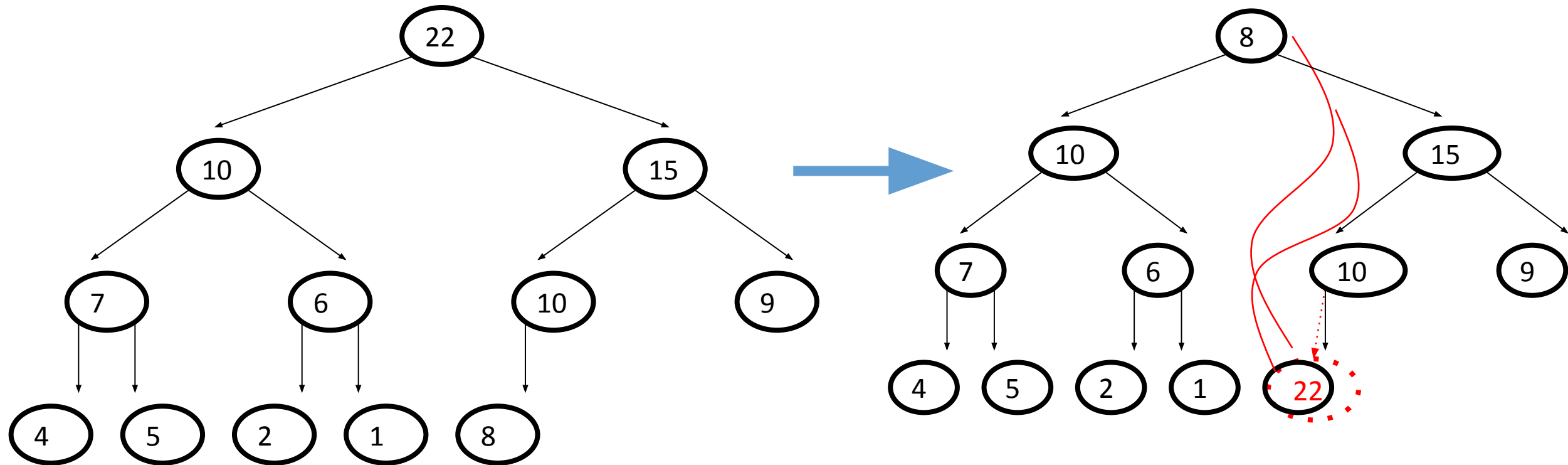
```
{ //index of a starting from 1
  j=1 ; a[20];
  Get Choice
  Repeat
  {
    a[j]=elem;
    insert(a,j);
    j++;
  }
  until(choice=='n');
}
```

### Algorithm Insert(a,n)

```
{
  i=n; elem=a[n];
  while((i>1)&&(a[i/2]<elem)do
  {
    a[i]=a[i/2];
    i=(i/2);
  }
  a[i]=elem;
  return true;
}
```

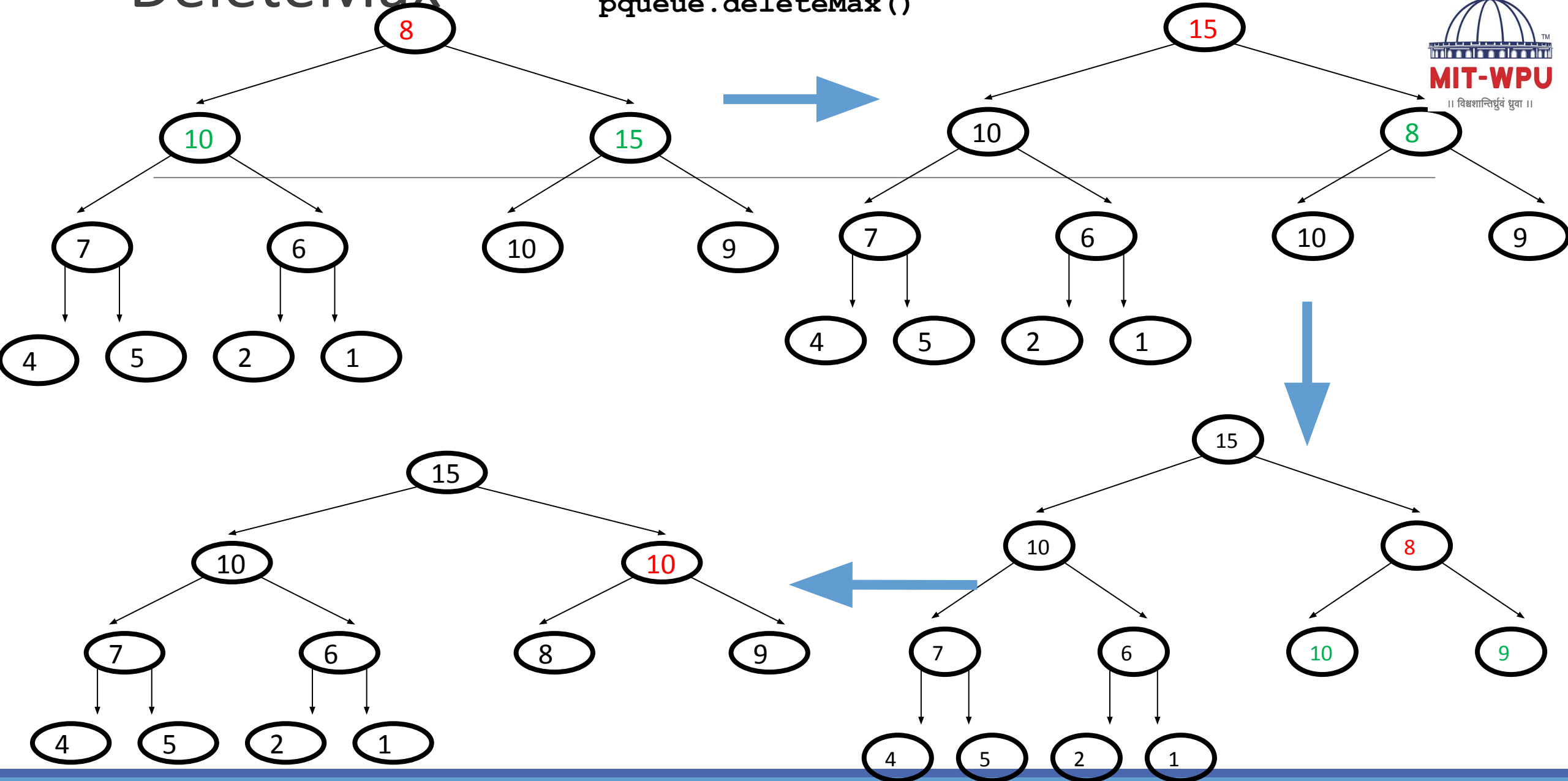
# DeleteMax

`pqueue.deleteMax()`



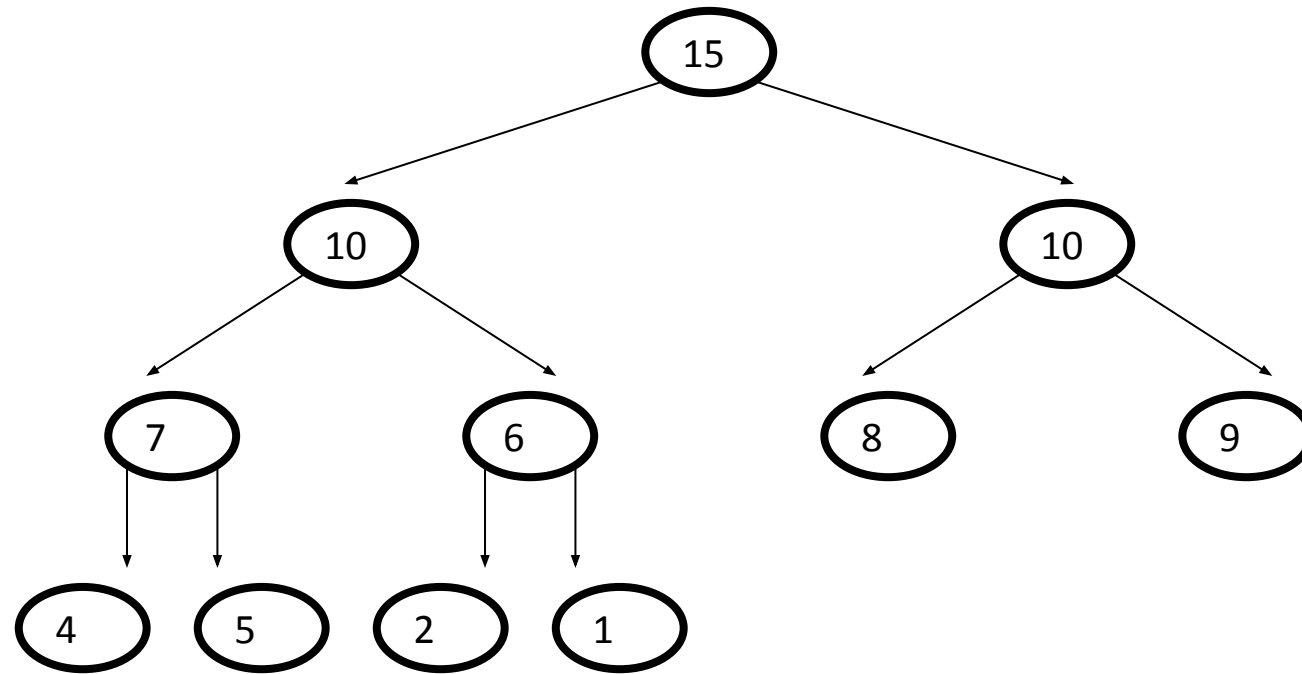
# DeleteMax

`pqueue.deleteMax()`



# DeleteMax (Final)

`pqueue.deleteMax()`



## Delete (Max) element Algorithm(index starting from 1)

Algorithm Deleteheap(a,n) //n are the no. of elements in array

```
{  
    // Interchange the maximum with the element at the end of  
    //array  
    repeat{  
        t=a[1];  
        a[1]=a[n] ;  
        a[n]=t;  
        n --;  
        Adjust(a, n,1);  
        accept choice  
    }until(choice='y');  
}
```

Algorithm Adjust(a,n,i)

```
{  
    while(2*i<=n) do  
    {  
        j=2*i;      //index of left child  
        // compare left and right child and let j be the larger child  
        if((j < n) and (a[j+1] > a[j]))  
            j=j+1  
        If a[i] >= a[j]  
            then break; //if parent > children then  
        break  
        else  
        {  
            //swap a[i] with a[j]  
            temp=a[i]; a[i]=a[j]; a[j]=temp;  
            i=j;  
        }  
    } //end of while  
}
```

# Sorting

---

- ❖ Other than as a priority queue, the heap has one other important usage: heap sort
- ❖ Heap sort is one of the fastest sorting algorithms, achieving speed as that of the quicksort and merge sort algorithms
- ❖ The advantages of heap sort are that it does not use recursion, and it is efficient for any data order
- ❖ There is no worse-case scenario in case of heap sort

# Heap Sort

---

- ❖ Steps for heap sort (ascending order) are as follows:
  1. Build the heap tree (for given array as it may not be in heap tree form)
  2. Start Delete Heap operations, storing each deleted element at the end of the heap array
  3. After performing step 2, again adjust the heap tree (ReHeapDown)
  4. Repeat step 2 and 3 for  $n-1$  times to sort the complete array



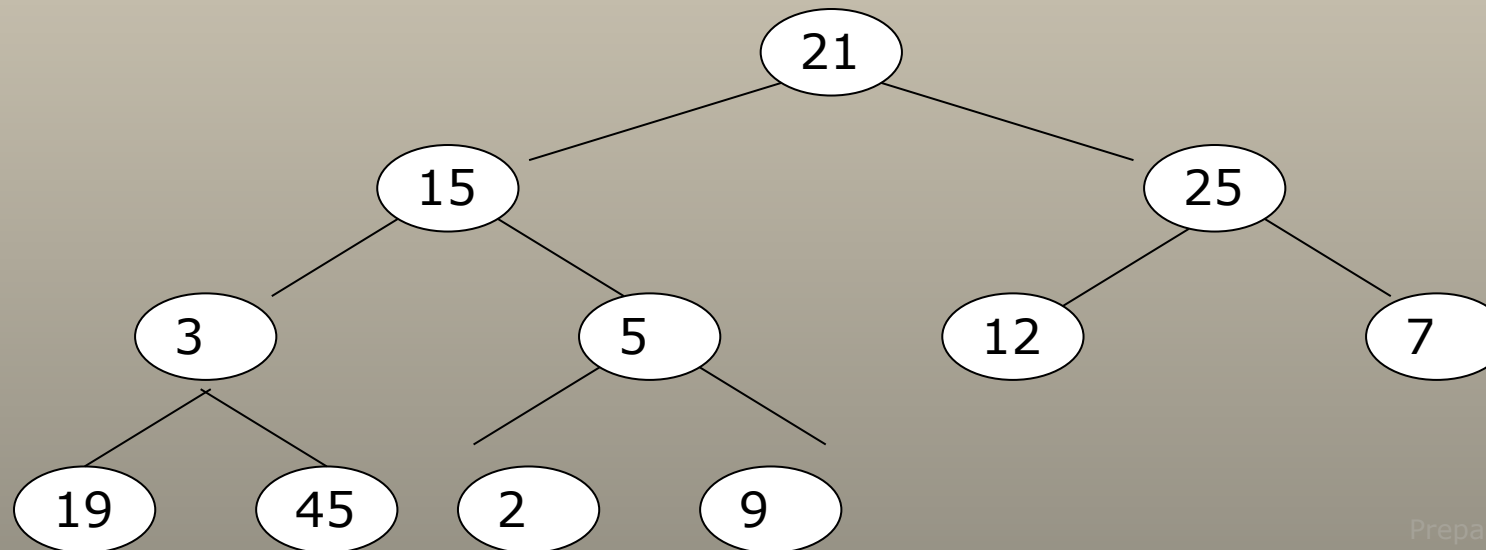
# Sample Run

- Start with unordered array of data

Array representation:

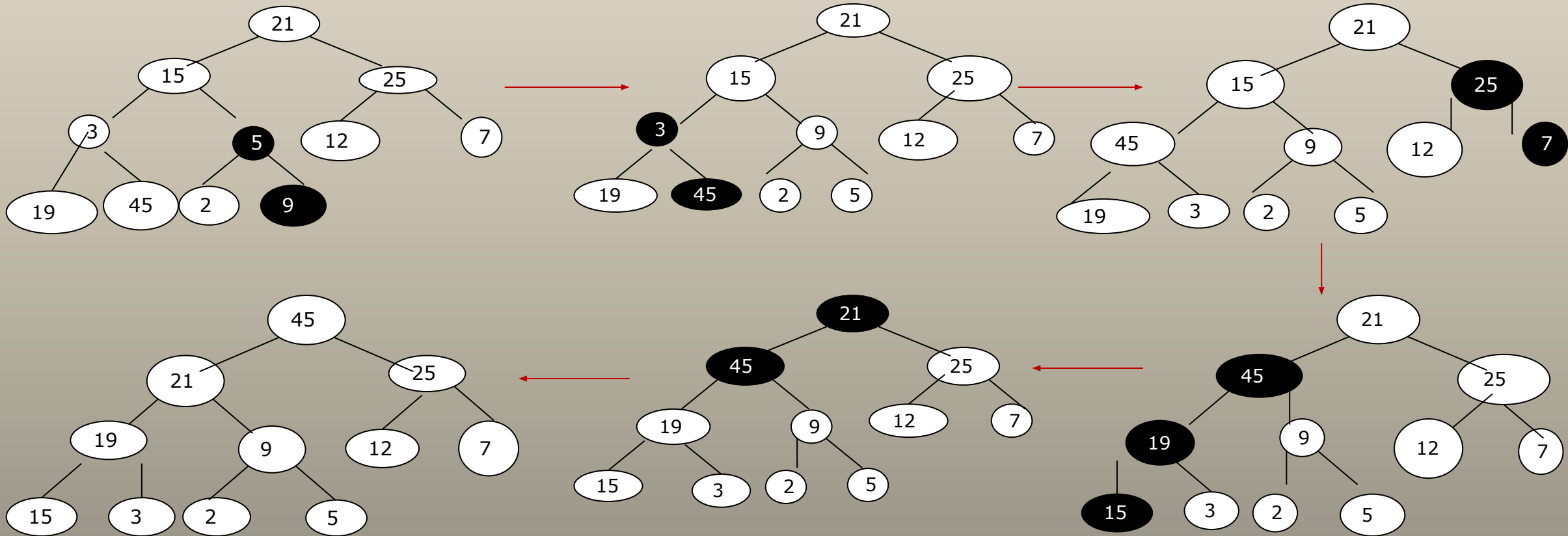
21	15	25	3	5	12	7	19	45	2	9
----	----	----	---	---	----	---	----	----	---	---

Binary tree representation:

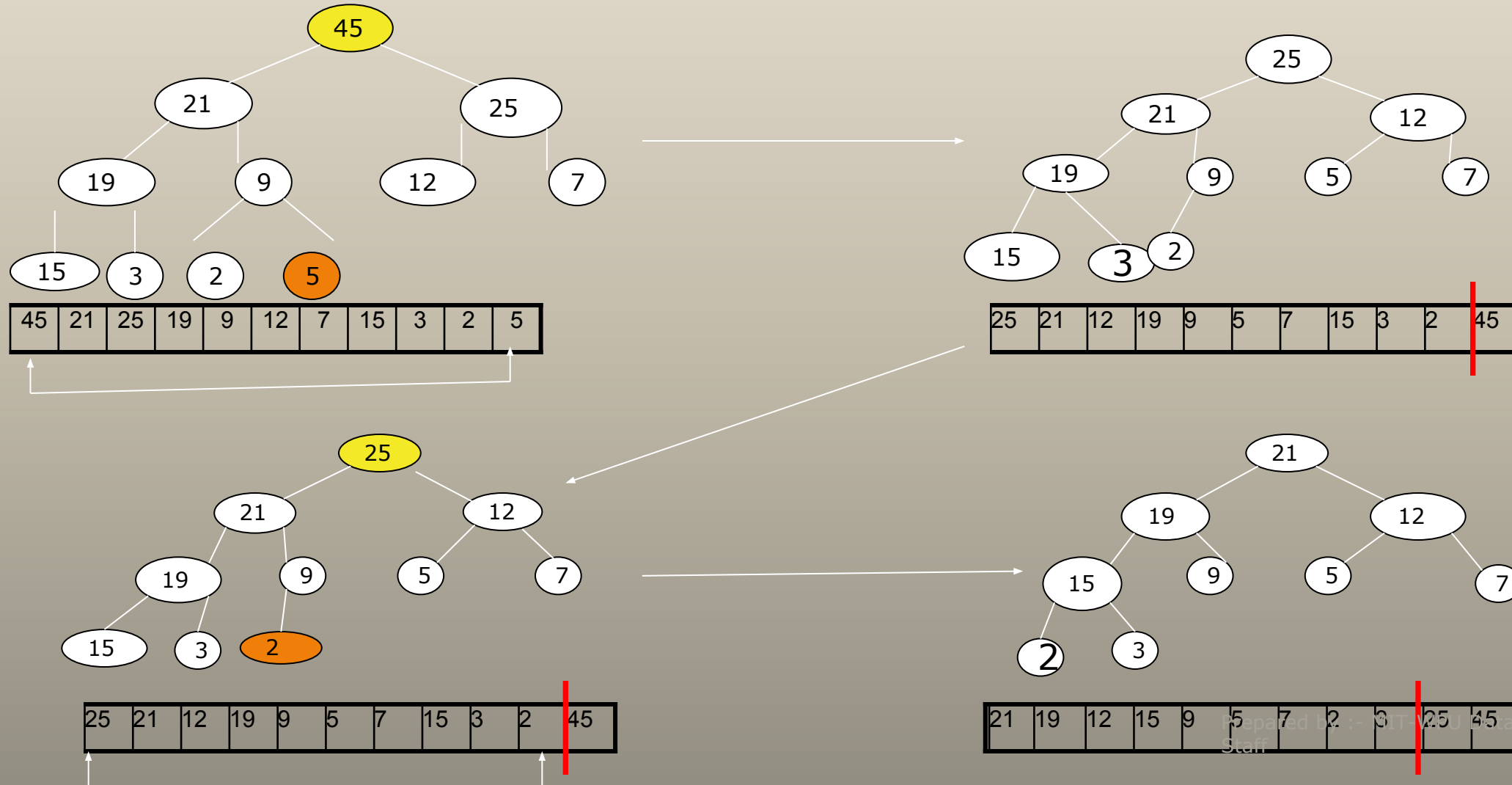


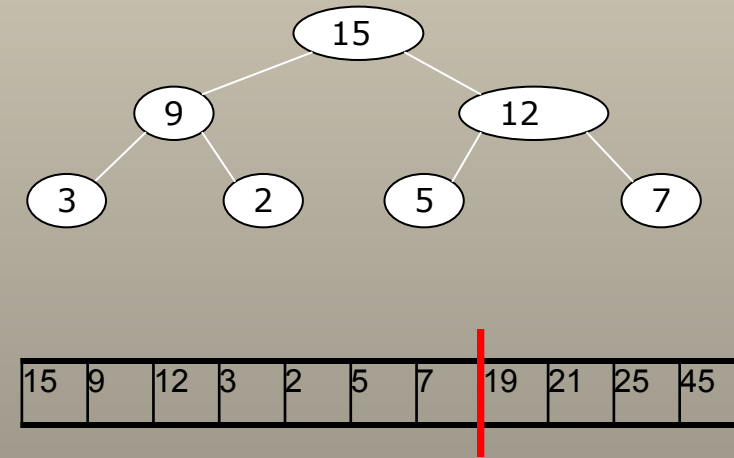
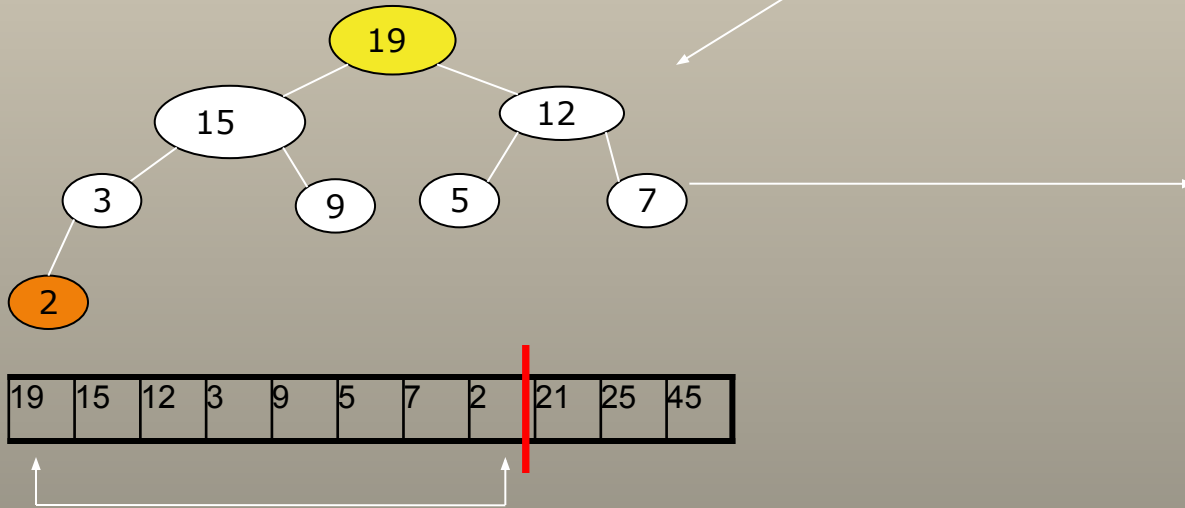
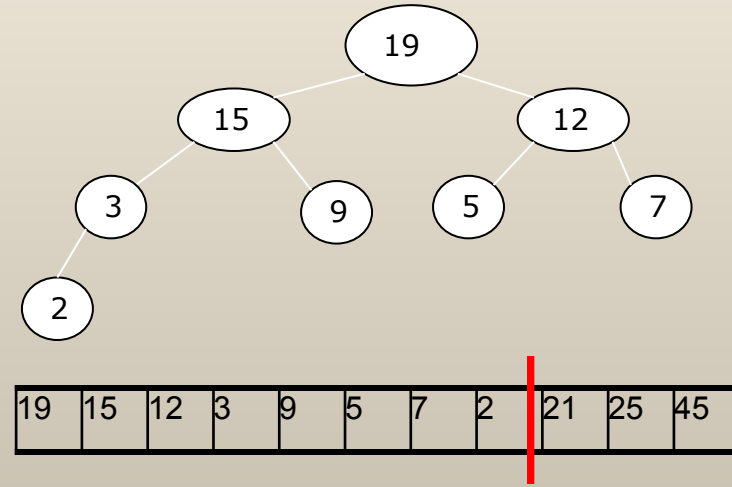
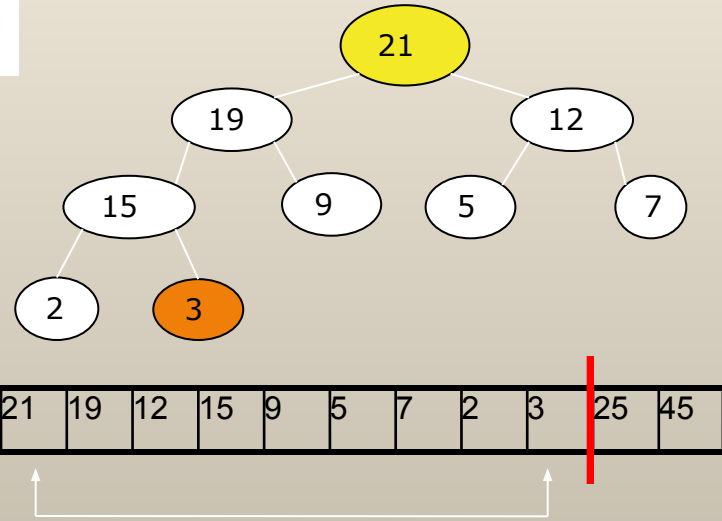
# Sample Run

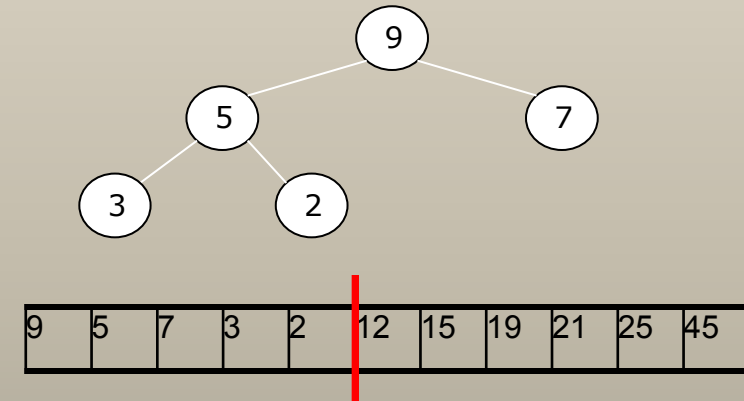
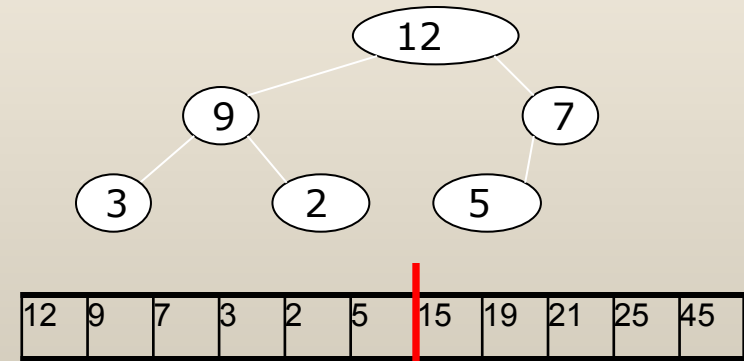
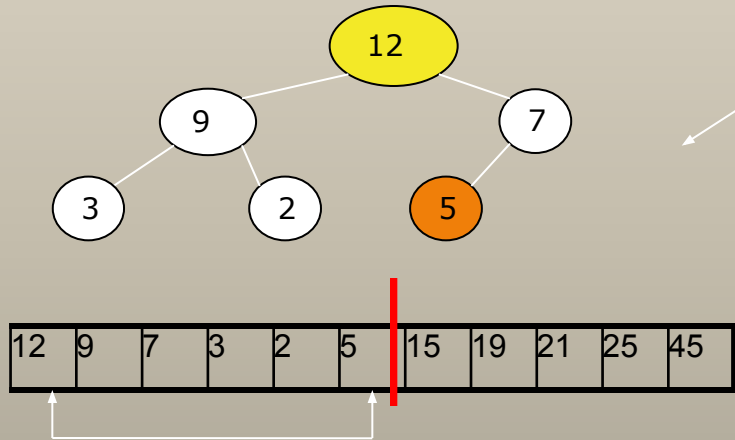
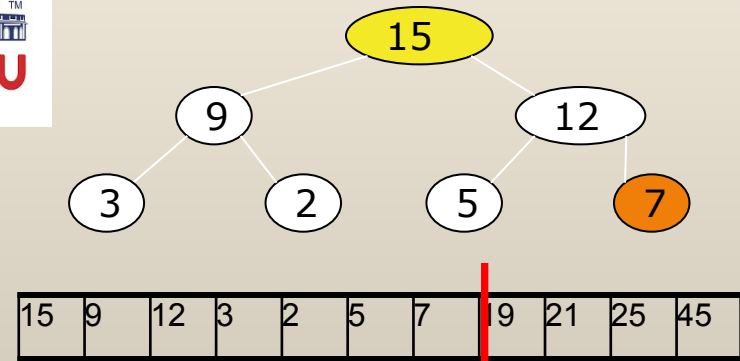
- Heapify the binary tree –(from  $n/2$  to 1)(ReheapDown)



**Step 2** – perform  $n - 1$  deleteMax(es), and replace last element in heap with first, then re-heapify. Place deleted element in the last nodes position.

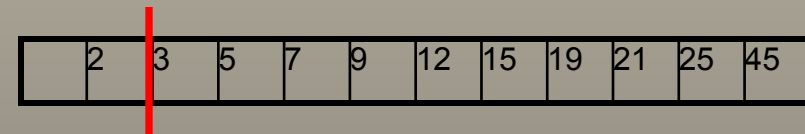






This continues till only one is left

...and finally



# Heap sort Algorithm(index starting from 1)



Algorithm HeapSort(a,n) //here n is the total no. of elements in the array

```
{
```

```
    for i= (n/2) to 1 step -1 do
```

```
        Adjust(a,n,i)
```

```
    // Interchange the new maximum with the element at the end of array
```

```
    while(n>1)
```

```
    {
```

```
        t=a[1];
```

```
        a[1]=a[n] ;
```

```
        a[n]=t;
```

```
        n --;
```

```
        Adjust(a, n, 1);
```

```
    }
```

```
}
```

Algorithm Adjust(a,n,i)

{

while( $2*i < n$ ) do

{

$j = 2*i$ ;     //index of left child

if( $(j \leq n)$  and  $(a[j+1] > a[j])$ )

$j = j+1$ ;     // compare left and right child and let j be the larger child

If  $a[i] \geq a[j]$

then break;     //if root > children then break

else

{

    //swap  $a[i]$  with  $a[j]$

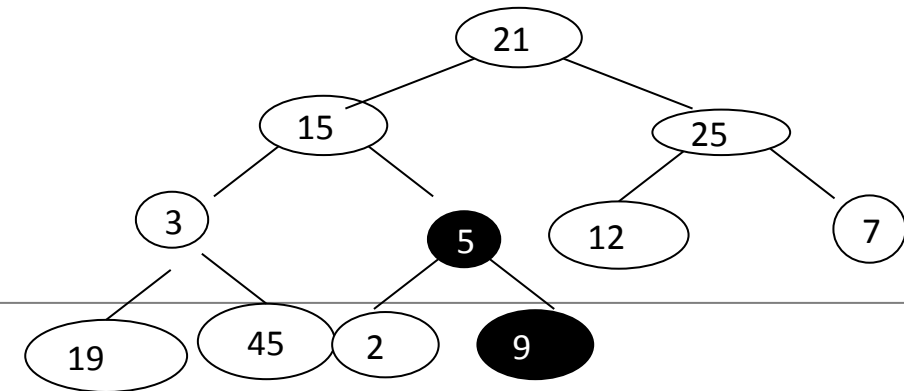
    temp= $a[i]$ ;  $a[i]=a[j]$ ;  $a[j]=temp$ ;

$i=j$ ;

}

} //end of while

}



# Heap Applications

---

Heaps are used commonly in the following operations:

- ❖ Selection algorithm
- ❖ Scheduling and prioritizing (priority queue)
- ❖ Sorting



# Selection Problem

---

- ❖ For the solution to the problem of determining the  $k$ th element, we can create the heap and delete  $k - 1$  elements from it, leaving the desired element at the root.
- ❖ So the selection of the  $k^{\text{th}}$  element will be very easy as it is the root of the heap
- ❖ For this, we can easily implement the algorithm of the selection problem using heap creation and heap deletion operations
- ❖ This problem can also be solved in  $O(n \log n)$  time using priority queues

# Scheduling and prioritizing (priority queue)

---

- ❖ The heap is usually defined so that only the largest element (that is, the root) is removed at a time.
- ❖ This makes the heap useful for scheduling and prioritizing
- ❖ In fact, one of the two main uses of the heap is as a priority queue, which helps systems decide what to do next

# Applications

---

- ❖ Applications of priority queues where heaps are implemented are as follows:
  - ❖ CPU scheduling
  - ❖ I/O scheduling
  - ❖ Process scheduling.

---

# Symbol Table

# Symbol Table

---

- Introduction to Symbol Tables
- Static tree table- Optimal Binary Search Tree (OBST)
- Dynamic tree table-AVL tree
- Multiway search tree- B-Tree

# Symbol Table

---

- Symbol table is defined as a name-value pair
- Symbol tables are data structures that are used by compilers to hold information about source-program constructs.

# Symbol Tables

---

Associates **attributes with identifiers used in a program**

- For instance, a **type attribute is usually associated with each identifier**
- A symbol table is a necessary component
  - Definition (declaration) of identifiers appears once in a program
  - Use of identifiers may appear in many places of the program text
- Identifiers and attributes are entered by the analysis phases
  - When processing a definition (declaration) of an identifier
  - In simple languages with only global variables and implicit declarations: **The scanner can enter an identifier into a symbol table if it is not already there**
  - In block-structured languages with scopes and explicit declarations: **The parser and/or semantic analyzer enter identifiers and corresponding attributes**

# Symbol Tables

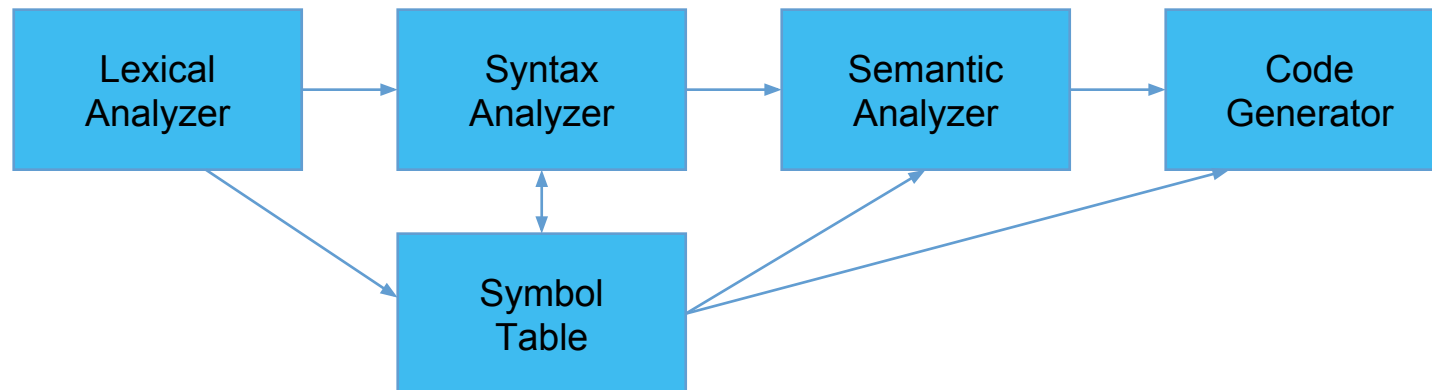
---

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

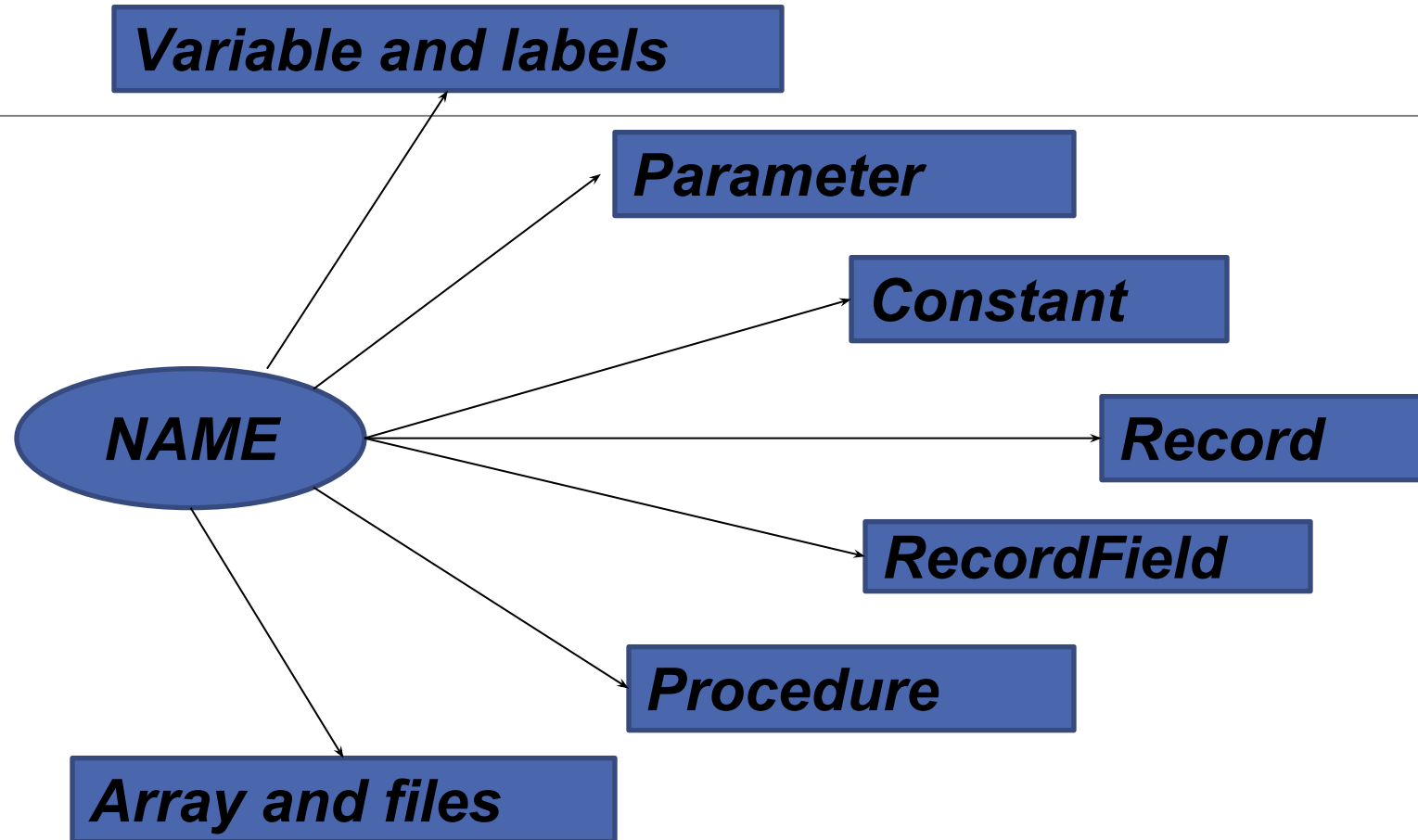


# The Symbol Table

- When identifiers are found, they will be entered into a symbol table, which will hold all relevant information about identifiers.
- This information will be used later by the semantic analyzer and the code generator.



# SYMBOL TABLE - NAMES



# SYMBOL TABLE-ATTRIBUTES

- Each piece of information associated with a name is called an ***attribute***.
- Attributes are language dependent.
- Different classes of Symbols have different Attributes

Variable, Constants	Procedure or function	Array
<ul style="list-style-type: none"><li>•• Type , Line number where declared , Lines where referenced , Scope</li></ul>	<ul style="list-style-type: none"><li>•• Number of parameters, parameters themselves, result type.</li></ul>	<ul style="list-style-type: none"><li>•• # of Dimensions, Array bounds.</li></ul>

# Symbol Table

---

- While compilers and assemblers are scanning a program, each identifier must be examined to determine if it is a keyword
- This information concerning the keywords in a programming language is stored in a symbol table
- Symbol table is a kind of 'keyed table'
- The keyed table stores <key, information> pairs with no additional logical structure

# Symbol Table(Contd..)

---

- The operations performed on symbol tables are the following:
- Insert the pairs <key, information> into the collection
- Remove the pairs <key, information> by specifying the key
- Search for a particular key
- Retrieve the information associated with a key

# Possible implementations:

---

- Unordered list: for a very small set of variables.
  - Simplest to implement
  - Implemented as an array or a linked list
  - Linked list can grow dynamically – alleviates problem of a fixed size array
  - Insertion is fast  $O(1)$ , but lookup is slow for large tables –  $O(n)$  on average.
- Ordered linear list:
  - If an array is sorted, it can be searched using binary search –  $O(\log_2 n)$
  - Insertion into a sorted array is expensive –  $O(n)$  on average
  - Useful when set of names is known in advance – table of reserved words

# Possible implementations:

---

- Binary search tree:
  - Can grow dynamically
  - Insertion and lookup are  $O(\log_2 n)$  on average

# Representation of Symbol Table

---

There are two different techniques for implementing a keyed table:

- Static Tree Tables
- Dynamic Tree Tables



# Static Tree Tables

---

- When symbols are known in advance and no insertion and deletion is allowed, it is called a static tree table
- An example of this type of table is a reserved word table in a compiler

# Dynamic Tree Table

---

- A dynamic tree table is used when symbols are not known in advance but are inserted as they come and deleted if not required
- Dynamic keyed tables are those that are built on-the-fly
- The keys have no history associated with their use

# Optimal Binary Search Trees

---

- To optimize a table knowing what keys are in the table and what the probable distribution is of those that are not in the table, we build an optimal binary search tree (OBST)
- Optimal binary search tree is a binary search tree having an average search time of all keys optimal
- An OBST is a BST with the minimum cost

# Optimal binary search trees

A full binary tree may not be an optimal binary search tree if the identifiers are searched for **with different frequency**

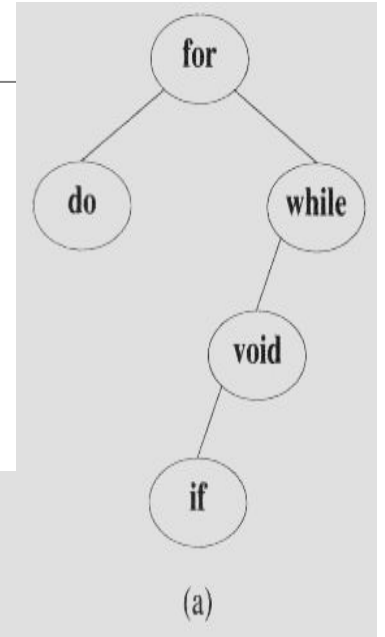
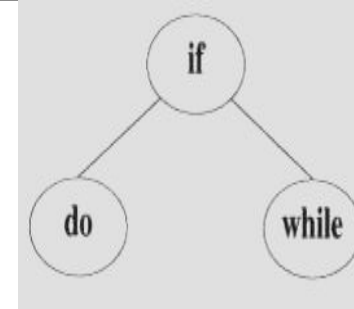
Consider these two search trees,  
If we search for each identifier with **equal Probability**

In first tree (a)

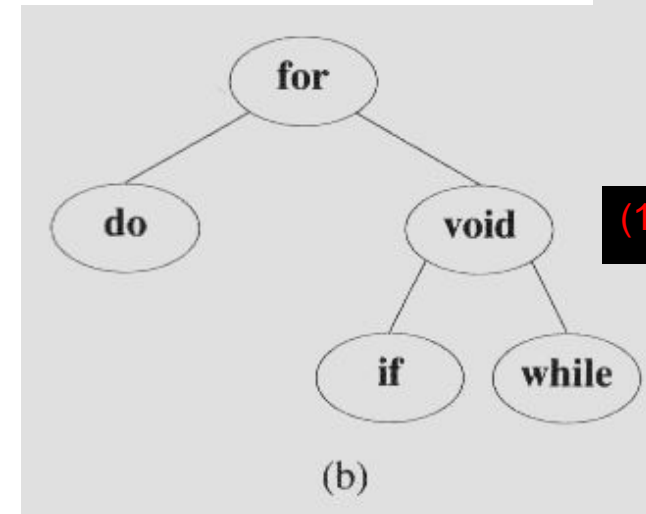
- the average number of comparisons for successful search is 2.4.
- Comparisons for second tree is 2.2.

The second tree (b) has

- a better worst case search time than the first tree.
- a better average behavior.



$$(1+2+2+3+4)/5 = 2.4$$

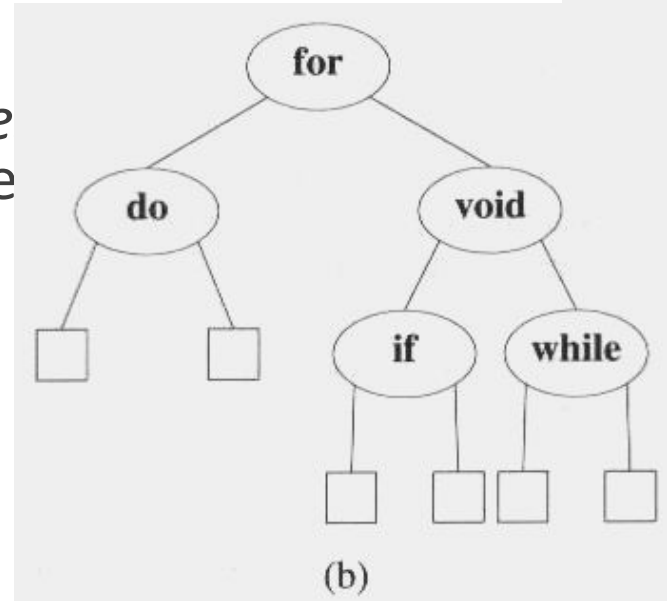
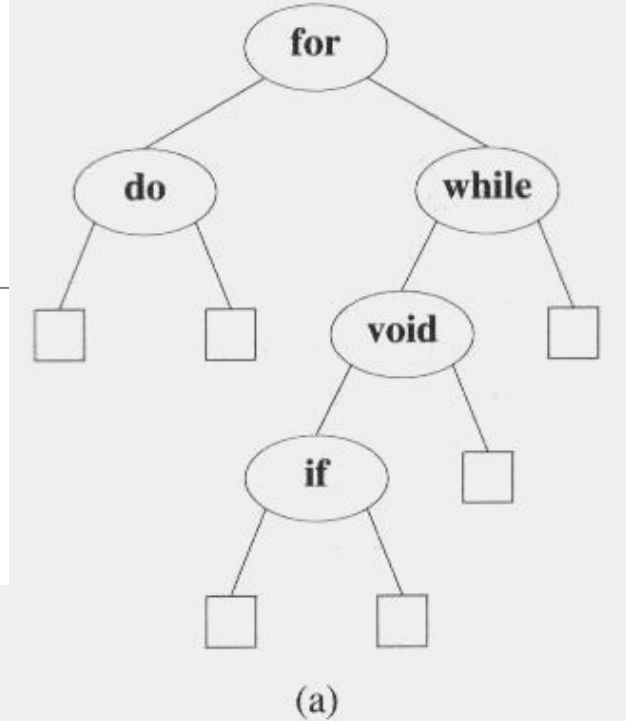


$$(1+2+2+3+3)/5 = 2.2$$

# Optimal binary search trees

In evaluating binary search trees, it is useful to add a special square node at every place there is a null links.

- We call these nodes *external node*
- We also refer to the external node as *failure nodes*.
- The remaining nodes are *internal nodes*.
- A binary tree with external nodes added is an *extended binary tree*



# Optimal binary search trees

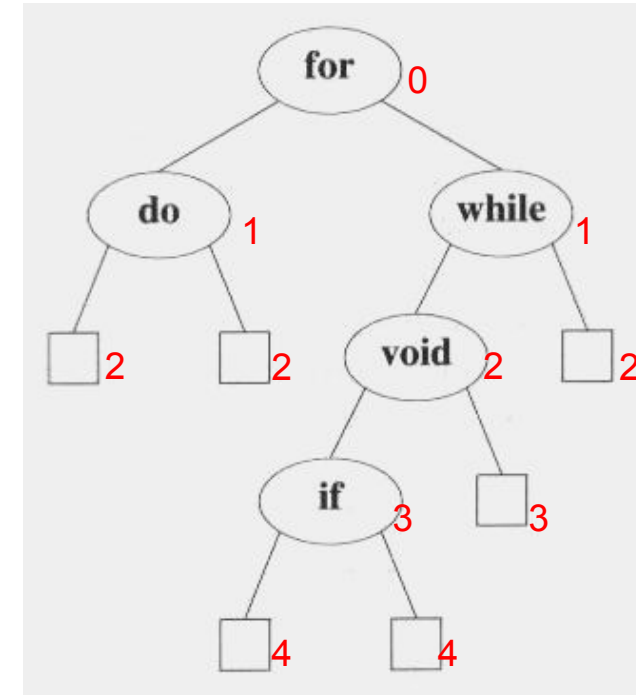
## *External / internal path length*

- The sum of all external / internal nodes' levels.

For example

- Internal path length,  $I$ , is:  
$$I = 0 + 1 + 1 + 2 + 3 = 7$$
- External path length,  $E$ , is :  
$$E = 2 + 2 + 4 + 4 + 3 + 2 = 17$$

A binary tree with  $n$  internal nodes are related by the formula :  $E = I + 2n$



# Optimal binary search trees

---

In the binary search tree:

- The identifiers  $a_1, a_2, \dots, a_n$  with  $a_1 < a_2 < \dots < a_n$
- The probability of searching for each  $a_i$  is  $p_i$
- The total cost (when only successful searches are made) is:

$$\sum_{i=1}^n p_i \cdot \text{level}(a_i)$$

# Optimal binary search trees

---

If we replace the null subtree by a failure node, we may partition the identifiers that are not in the binary search tree into  $n+1$  classes  $E_i$ ,  $0 \leq i \leq n$

- $E_i$  contains all identifiers  $x$  such that  $a_i < x < a_{i+1}$
- For all identifiers in a particular class,  $E_i$ , the search terminates at the same failure node



# Optimal binary search trees

**We number the failure nodes from 0 to  $n$  with  $i$  being for class  $E_i$ ,  $0 \leq i \leq n$ .**

- If  $q_i$  is the probability that the identifier we are searching for is in  $E_i$ , then the cost of the failure node is:

$$\sum_{i=0}^n q_i \cdot (\text{level}(\text{failure node } i) - 1)$$

Therefore, the total cost of a binary search tree is:

$$\sum_{i=1}^n p_i \cdot \text{level}(a_i) + \sum_{i=0}^n q_i \cdot (\text{level}(\text{failure node } i) - 1)$$

- An optimal binary search tree for the identifier set  $a_1, \dots, a_n$  is one that minimizes
- Since all searches must terminate either successfully or unsuccessfully, we have

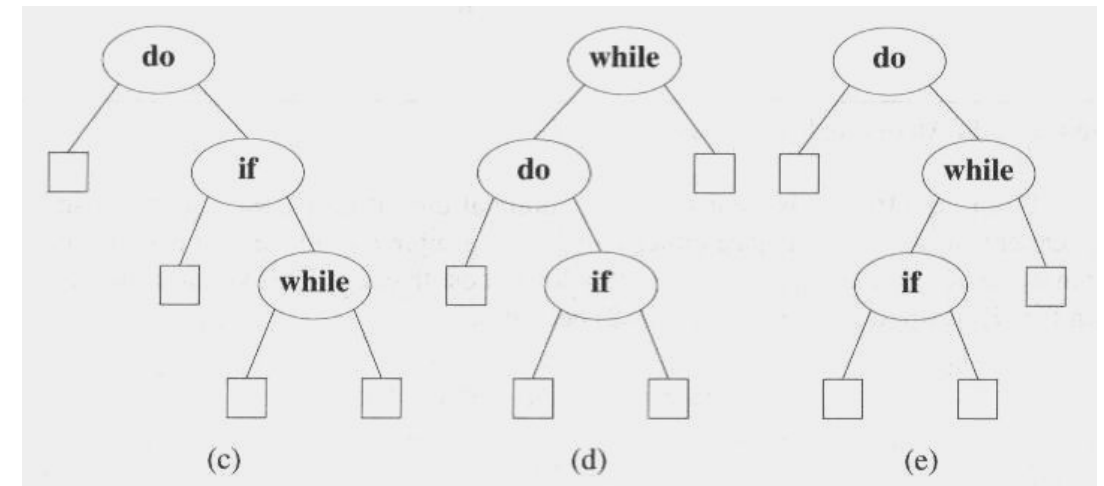
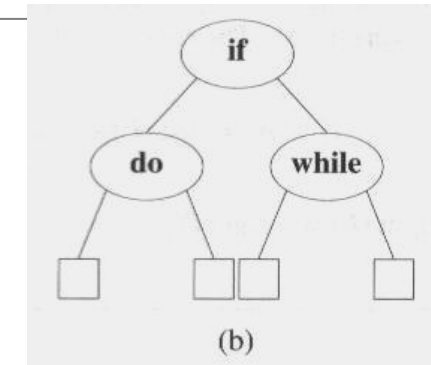
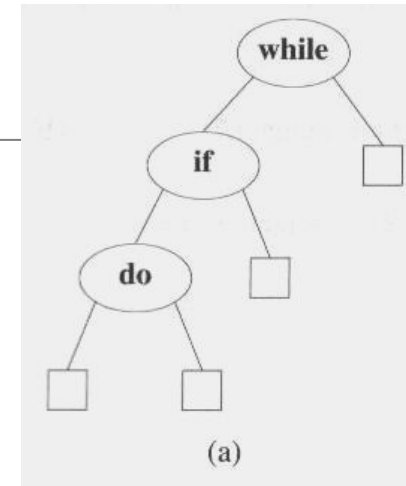
$$\sum_{i=1}^n p_i + \sum_{i=0}^n q_i = 1$$

# Optimal binary search trees

The possible binary search trees for the identifier set  $(a_1, a_2, a_3) = (\mathbf{do}, \mathbf{if}, \mathbf{while})$

- The identifiers with equal probabilities,  $p_i = a_j = 1/7$  for all  $i, j$ ,
  - $\text{cost}(\text{tree } a) = 15/7$ ;
  - $\text{cost}(\text{tree } b) = 13/7$ ; (optimal)
  - $\text{cost}(\text{tree } c) = 15/7$ ;
  - $\text{cost}(\text{tree } d) = 15/7$ ;
  - $\text{cost}(\text{tree } e) = 15/7$ ;

- $p_1 = 0.5, p_2 = 0.1, p_3 = 0.05$ ,  
 $q_0 = 0.15, q_1 = 0.1, q_2 = 0.05, q_3 = 0.05$ 
  - $\text{cost}(\text{tree } a) = 2.65$ ;
  - $\text{cost}(\text{tree } b) = 1.9$ ;
  - $\text{cost}(\text{tree } c) = 1.5$ ; (optimal)
  - $\text{cost}(\text{tree } d) = 2.05$ ;
  - $\text{cost}(\text{tree } e) = 1.6$ ;



# OPTIMAL BINARY SEARCH TREES (Contd..)

With equal probability  $P(i)=Q(i)=1/7$ .

Let us find an OBST out of these.

$$\text{Cost}(\text{tree a}) = \sum_{1 \leq i \leq n} P(i) * \text{level a}(i) + \sum_{0 \leq i \leq n} Q(i) * \text{level (E}_i) - 1$$

$$1 \leq i \leq n$$

$$0 \leq i \leq n$$

$$(2-1) \quad (3-1) \quad (4-1) \quad (4-1)$$

$$= 1/7 [1+2+3 + 1 + 2 + 3 + 3] = 15/7$$

$$\text{Cost (tree b)} = 1/7 [1+2+2+2+2+2+2] = 13/7$$

$$\text{Cost (tree c)} = \text{cost (tree d)} = \text{cost (tree e)} = 15/7$$

∴ tree b is optimal.

## OPTIMAL BINARY SEARCH TREES (Contd..)

---

If  $P(1) = 0.5$ ,  $P(2) = 0.1$ ,  $P(3) = 0.005$ ,  $Q(0) = 0.15$ ,  $Q(1) = 0.1$ ,  $Q(2) = 0.05$  and  $Q(3) = 0.05$  find the OBST.

$$\text{Cost (tree a)} = .5 \times 3 + .1 \times 2 + .05 \times 3 + .15 \times 3 + .1 \times 3 + .05 \times 2 + .05 \times 1 = 2.65$$

$$\text{Cost (tree b)} = 1.9, \text{Cost (tree c)} = 1.5, \text{Cost (tree d)} = 2.05,$$

$$\text{Cost (tree e)} = 1.6 \text{ Hence tree C is optimal.}$$

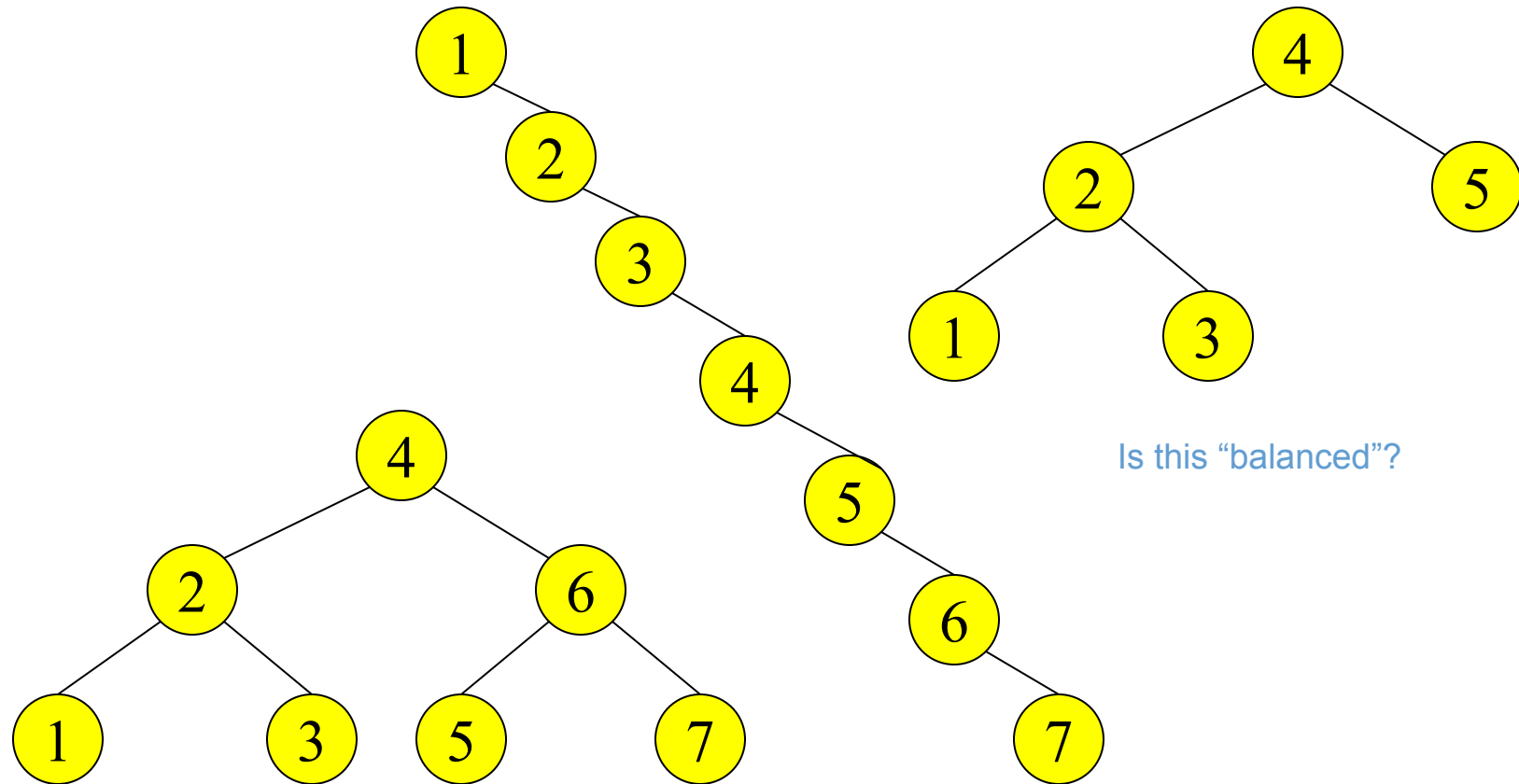
# Binary Search Tree - Worst Time

---

Worst case running time is  $O(N)$

- What happens when you Insert elements in ascending order?
  - **Insert: 2, 4, 6, 8, 10, 12 into an empty BST**
- Problem: Lack of “balance”:
  - compare depths of left and right subtree
- Unbalanced degenerate tree

# Balanced and unbalanced BST



# Balancing Binary Search Trees

---

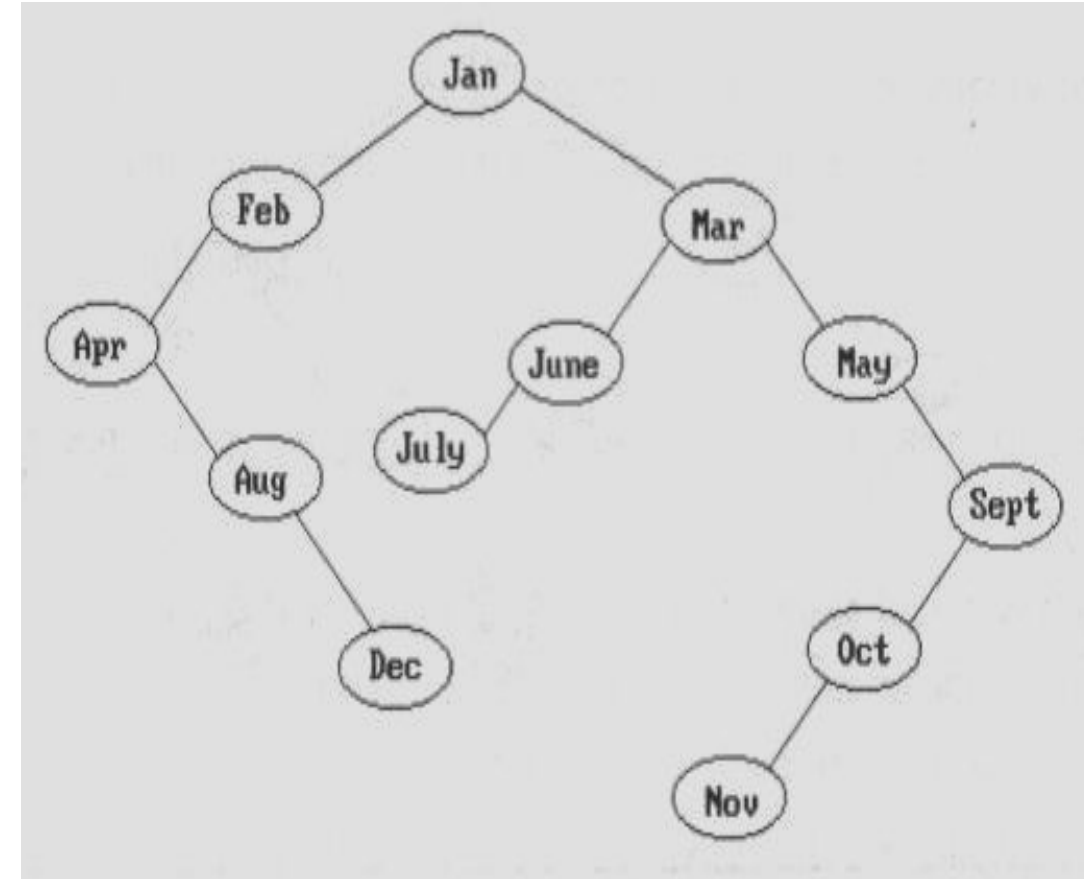
Many algorithms exist for keeping binary search trees balanced

- Adelson-Velskii and Landis (**AVL**) trees (height-balanced trees)
- **Splay trees** and other self-adjusting trees
- **B-trees** and other Multiway search trees

# AVL Trees

We also may maintain dynamic tables as binary search trees.

- The binary search tree obtained by entering the months *January* to *December*, in that order, into an initially empty binary search tree
- The maximum number of comparisons needed to search for any identifier in the tree (for *November*).
- Average number of comparisons is  $42/12 = 3.5$

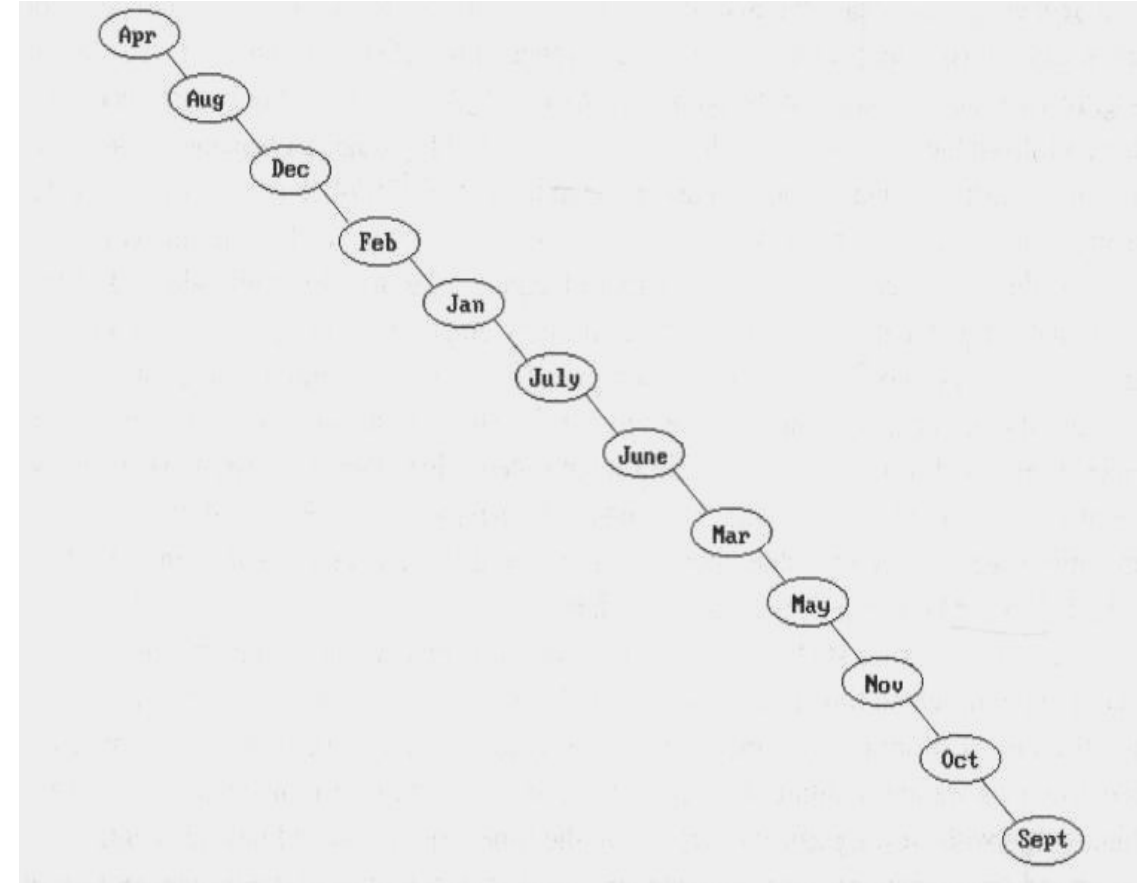




# AVL Trees

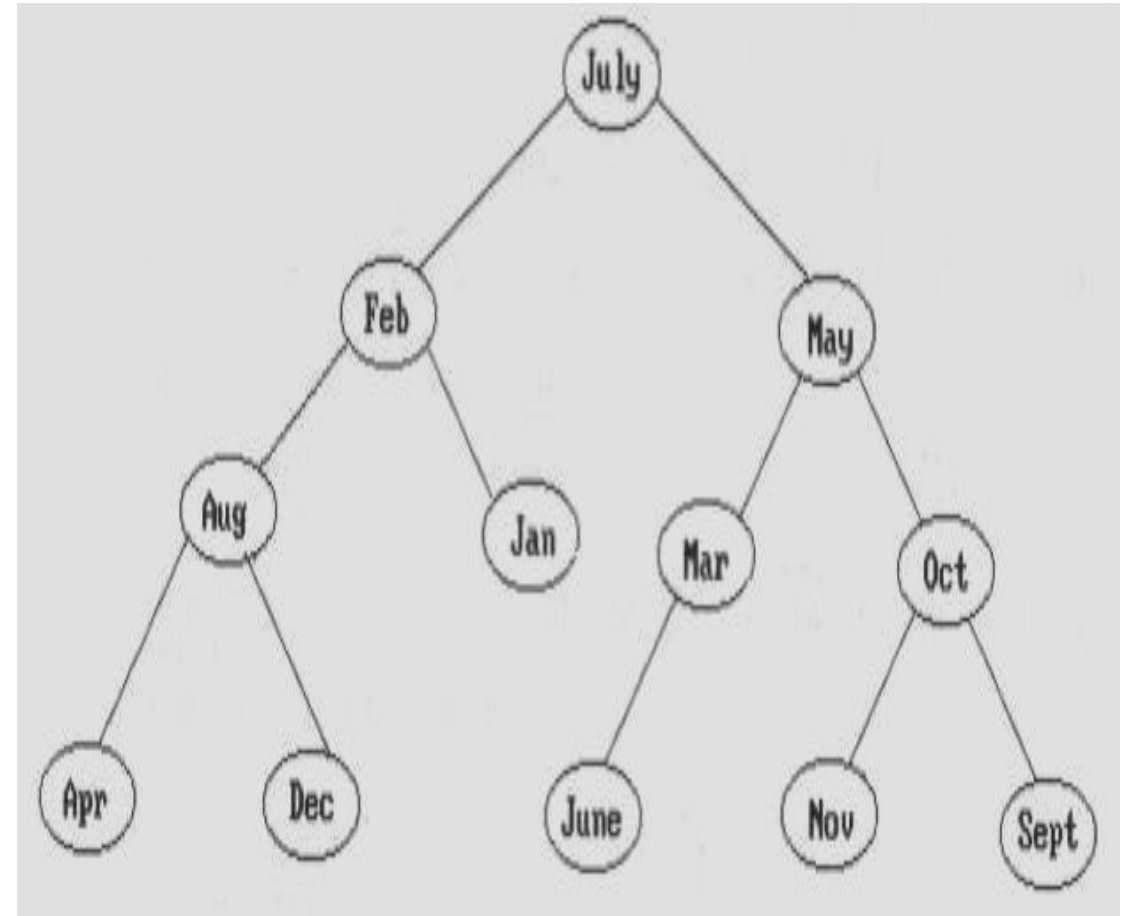
Suppose that we now enter the months into an initially empty tree in **alphabetical order**

- The tree degenerates into the chain  
number of comparisons:  
maximum: 12, and average: 6.5
- in the worst case, binary search trees correspond to sequential searching in an ordered list



## Another insert sequence

- In the order *Jul, Feb, May, Aug, Jan, Mar, Oct, Apr, Dec, Jun, Nov, and Sep*
- Well balanced and does not have any paths to leaf nodes that are much longer than others.
- Number of comparisons:  
maximum: 4, and average:  $37/12 \approx 3.1$ .
- All intermediate trees created during the construction of Figure are also well balanced



# AVL Trees

---

Adelson-Velskii and Landis introduced a binary tree structure (*AVL trees*):

- Balanced with respect to the heights of the subtrees.
- We can perform dynamic retrievals in  $O(\log n)$  time for a tree with  $n$  nodes.
- We can enter an element into the tree, or delete an element from it, in  $O(\log n)$  time. The resulting tree remain height balanced.
- As with binary trees, we may define AVL tree recursively

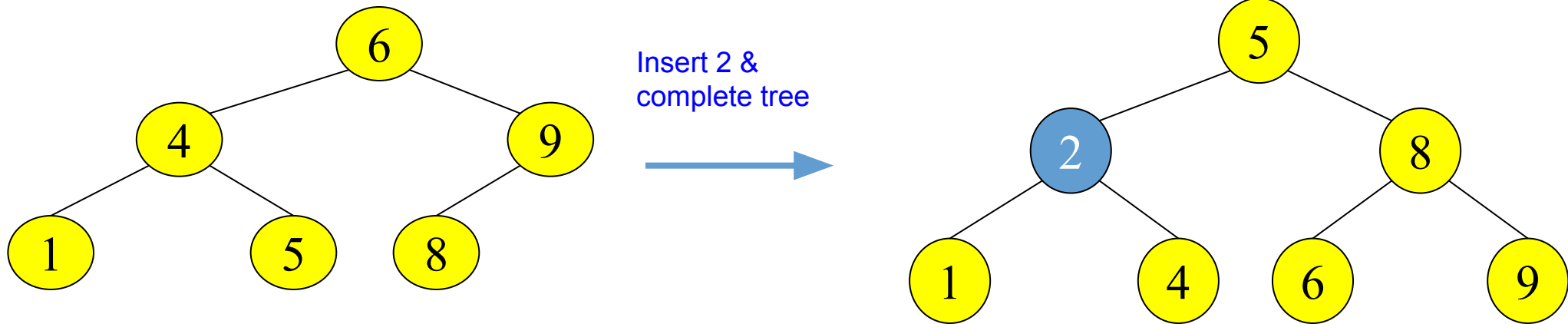
# Perfect Balance

Want a **complete tree** after every operation

- tree is full except possibly in the lower right

This is expensive

- For example, insert 2 in the tree on the left and then rebuild as a complete tree



# AVL - Good but not Perfect Balance

---

AVL trees are height-balanced binary search trees

Balance factor of a node

- $\text{height}(\text{left subtree}) - \text{height}(\text{right subtree})$

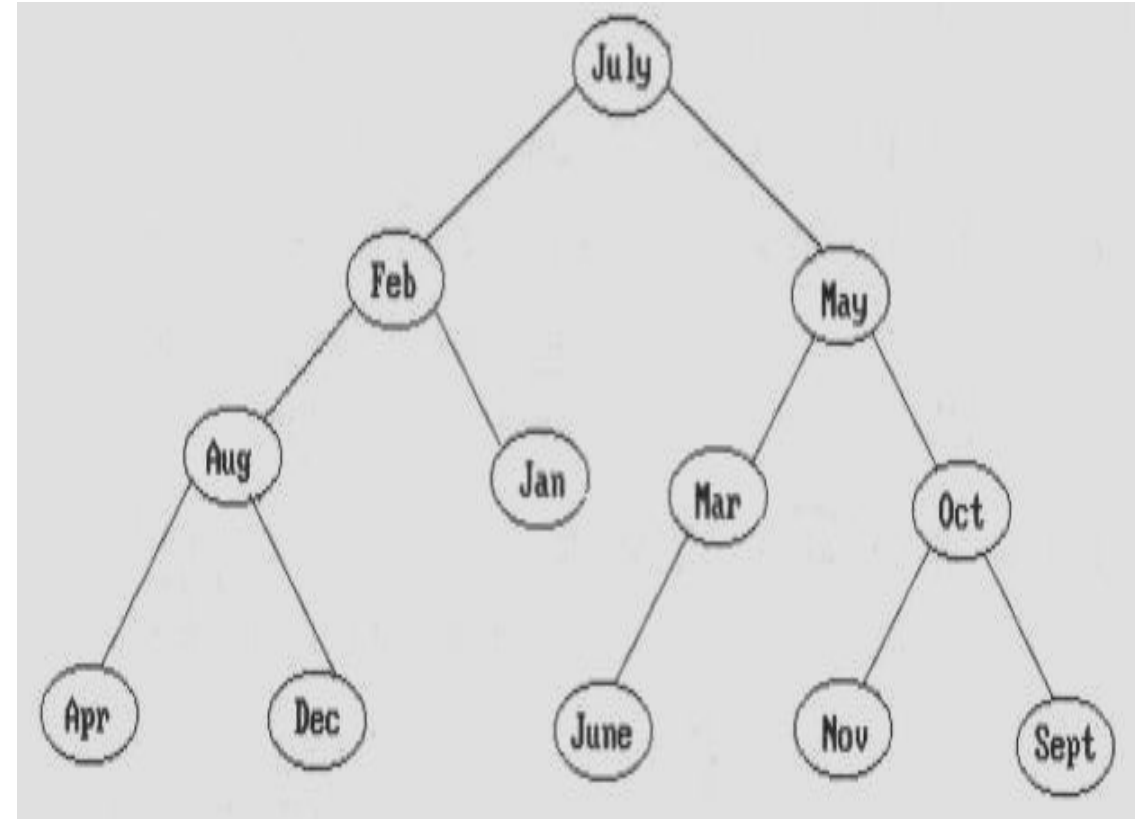
An AVL tree has balance factor calculated at every node

- For every node, heights of left and right subtree can differ by no more than 1
- Store current heights in each node

# AVL Trees Definition:

- An empty binary tree is height balanced.
- If  $T$  is a nonempty binary tree with  $T_L$  and  $T_R$  as its left and right subtrees, then  $T$  is *height balanced iff*
  - $T_L$  and  $T_R$  are height balanced, and
  - $|h_L - h_R| \leq 1$  where  $h_L$  and  $h_R$  are the heights of  $T_L$  and  $T_R$ , respectively.

The definition of a height balanced binary tree requires that every subtree also be height balanced



# AVL Trees

---

The numbers by each node represent the difference in heights between the left and right subtrees of that node

We refer to this as the balance factor of the node

## Definition:

- The balance factor,  $BF(T)$ , of a node,  $T$ , in a binary tree is defined as  $h_L - h_R$ , where  $h_L, h_R$  are the heights of the left/right subtrees of  $T$ .
- For any node  $T$  in an AVL tree  $BF(T) = -1, 0$ , or  $1$ .

# AVL Trees

---

We carried out the rebalancing using four different kinds of rotations: *LL*, *RR*, *LR*, and *RL*

- *LL* and *RR* are symmetric as are *LR* and *RL*
- These rotations are characterized by the nearest ancestor, *A*, of the inserted node, *Y*, whose balance factor becomes  $\pm 2$ .
  - *LL*: *Y* is inserted in the left subtree of the left subtree of *A*.
  - *LR*: *Y* is inserted in the right subtree of the left subtree of *A*
  - *RR*: *Y* is inserted in the right subtree of the right subtree of *A*
  - *RL*: *Y* is inserted in the left subtree of the right subtree of *A*



# Insertions in AVL Trees

Let the node that needs rebalancing be  $\alpha$ .

There are 4 cases:

**Outside Cases** (require single rotation) :

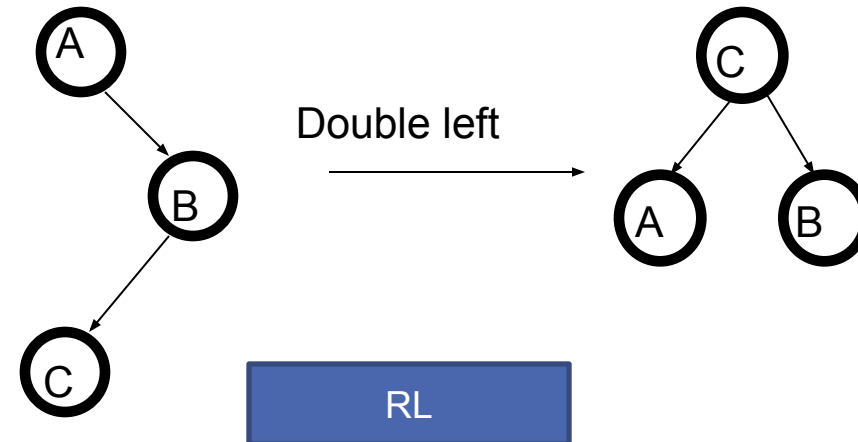
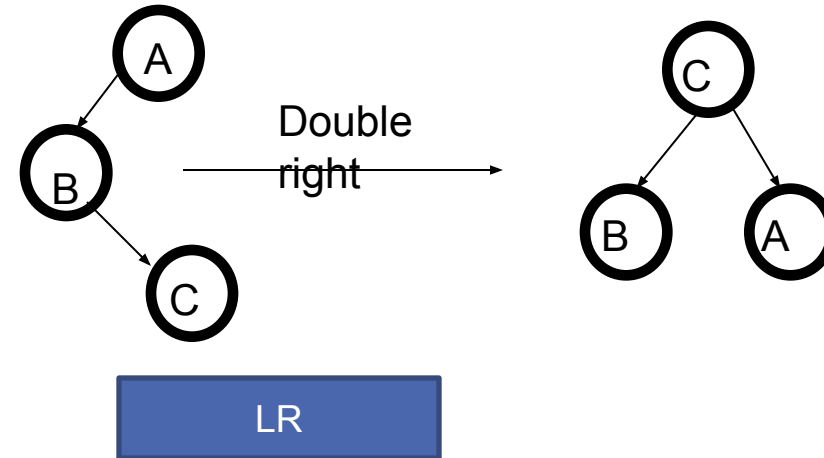
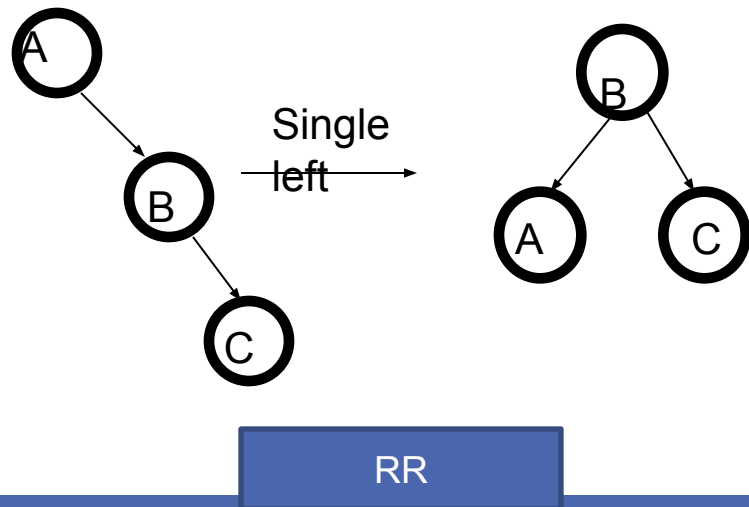
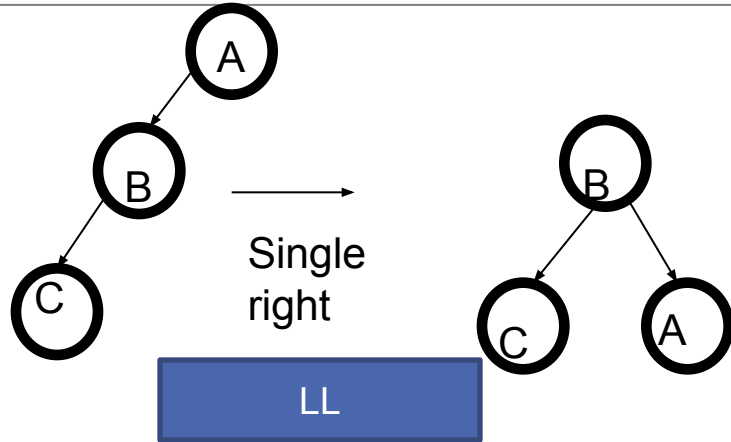
1. Insertion into **left** subtree **of left** child of  $\alpha$ .
2. Insertion into **right** subtree **of right** child of  $\alpha$ .

**Inside Cases** (require double rotation) :

3. Insertion into **right** subtree **of left** child of  $\alpha$ .
4. Insertion into **left** subtree **of right** child of  $\alpha$ .

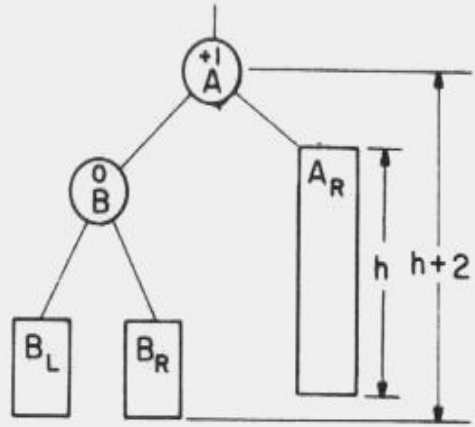
The rebalancing is performed through four separate rotation algorithms.

# AVL Balancing : Four Rotations

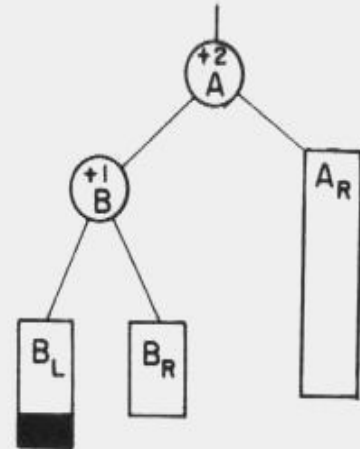


# Rebalancing rotations

Balanced subtree



Unbalanced following insertion

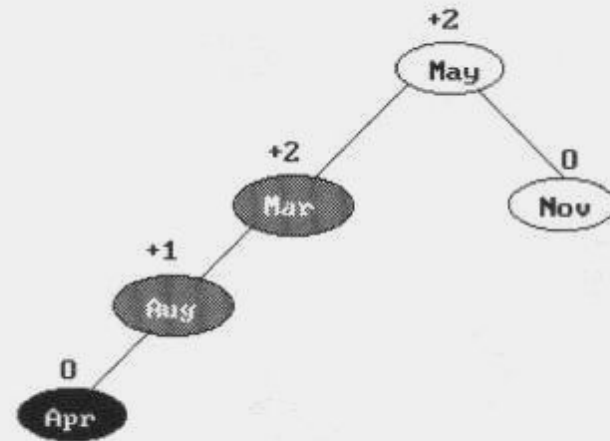
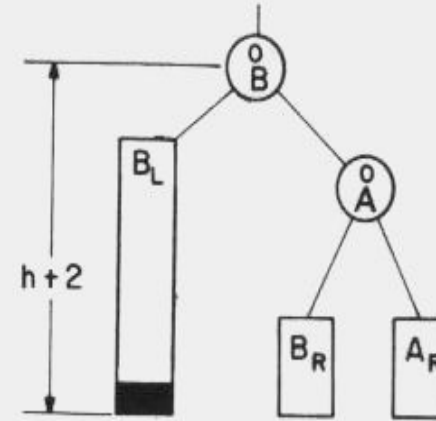


Height of  $B_L$  increases to  $h+1$

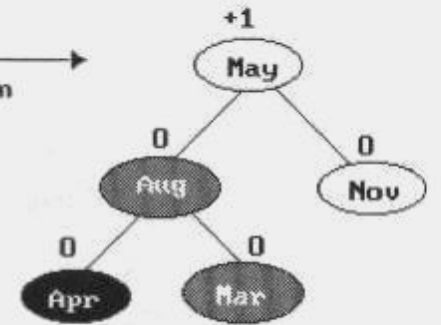
rotation type

LL

Rebalanced subtree



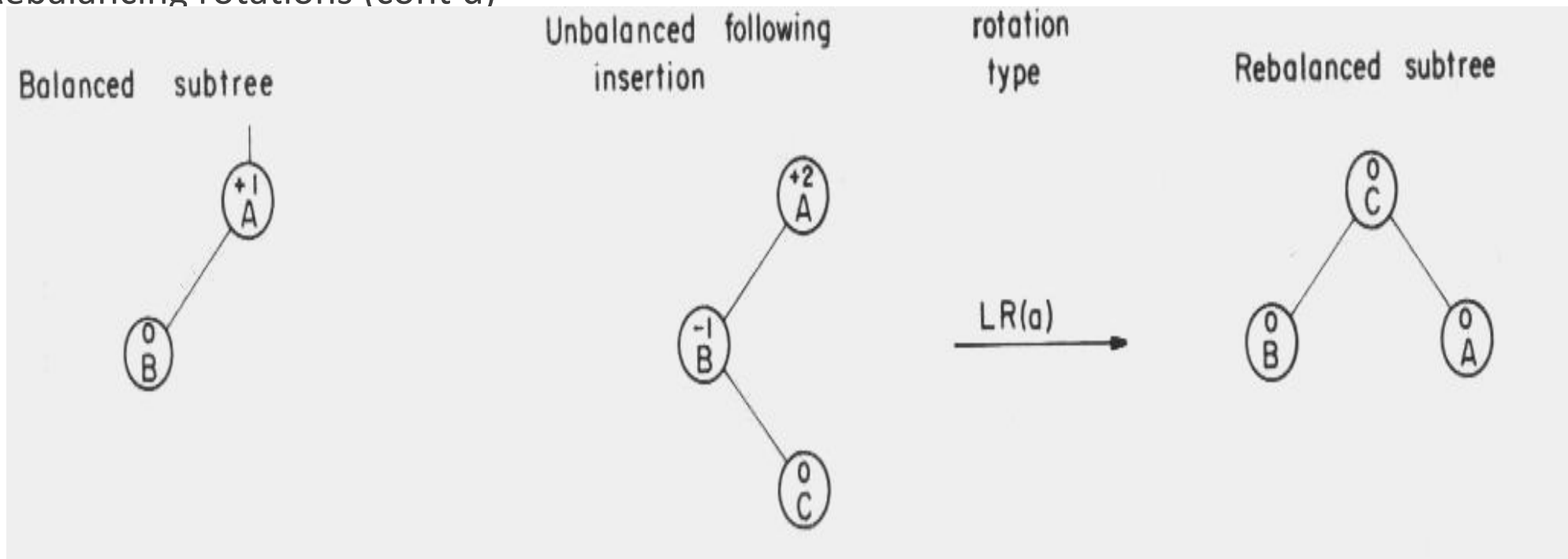
LL  
Rotation



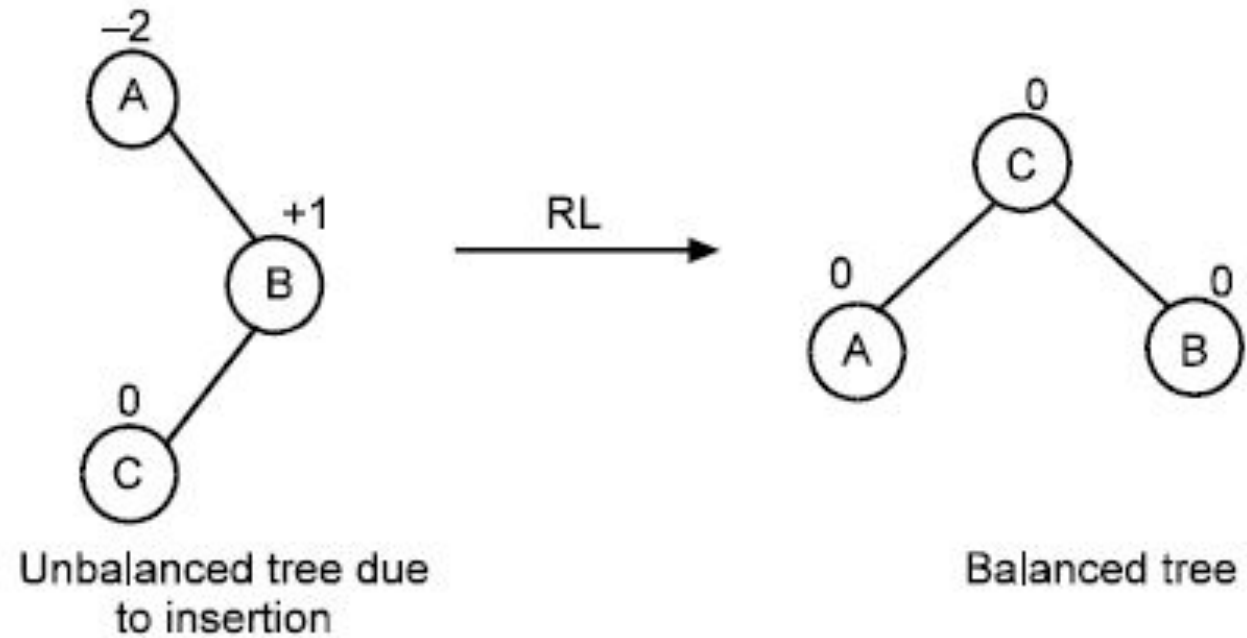
(e) Insert April

# AVL Trees

Rebalancing rotations (cont'd)

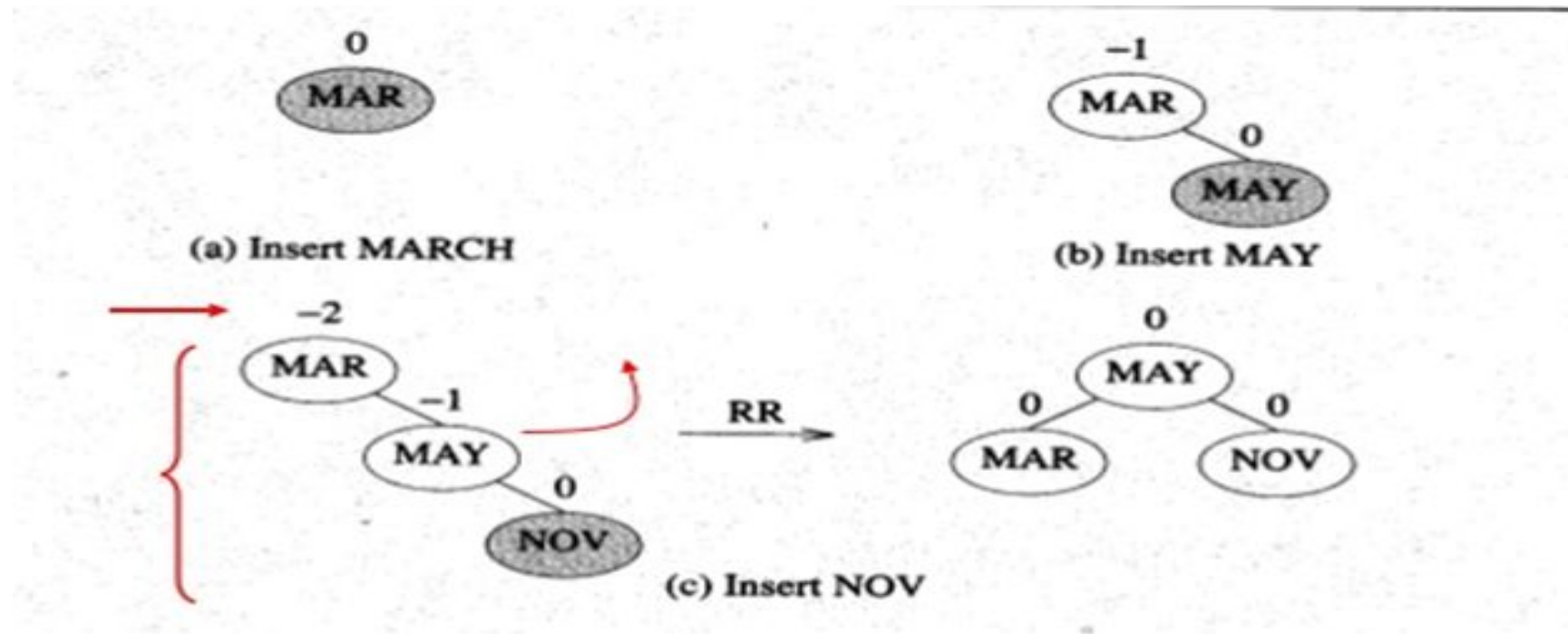


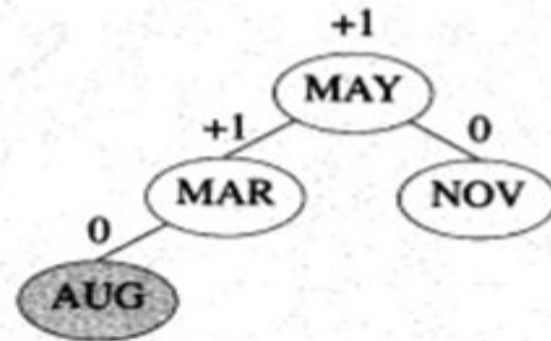
# Case 4: RL (Left of Right)



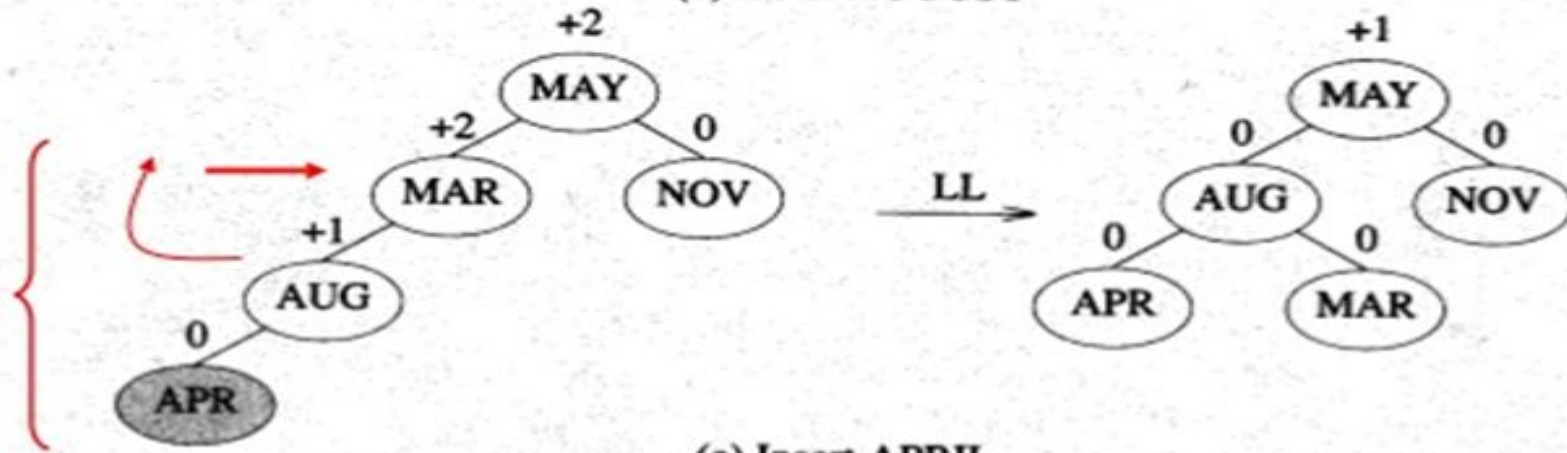
(a)

- This time we will insert the months into the tree in the order
  - *Mar, May, Nov, Aug, Apr, Jan, Dec, Jul, Feb, Jun, Oct, Sep*
  - It shows the tree as it grows, and the [restructuring involved in keeping it balanced](#).
- 



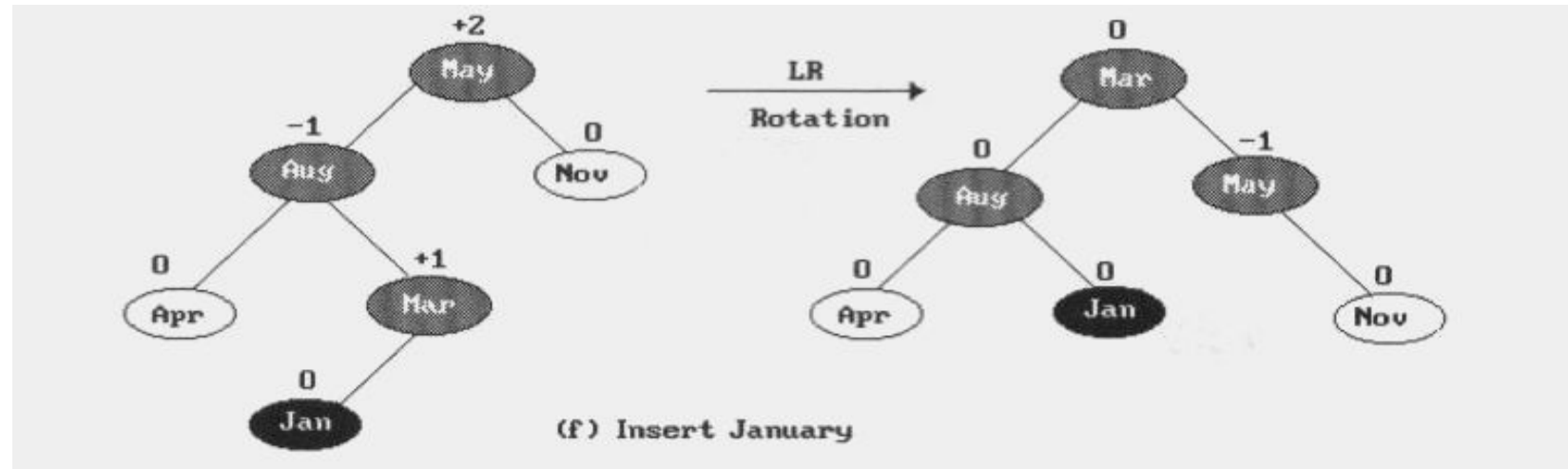
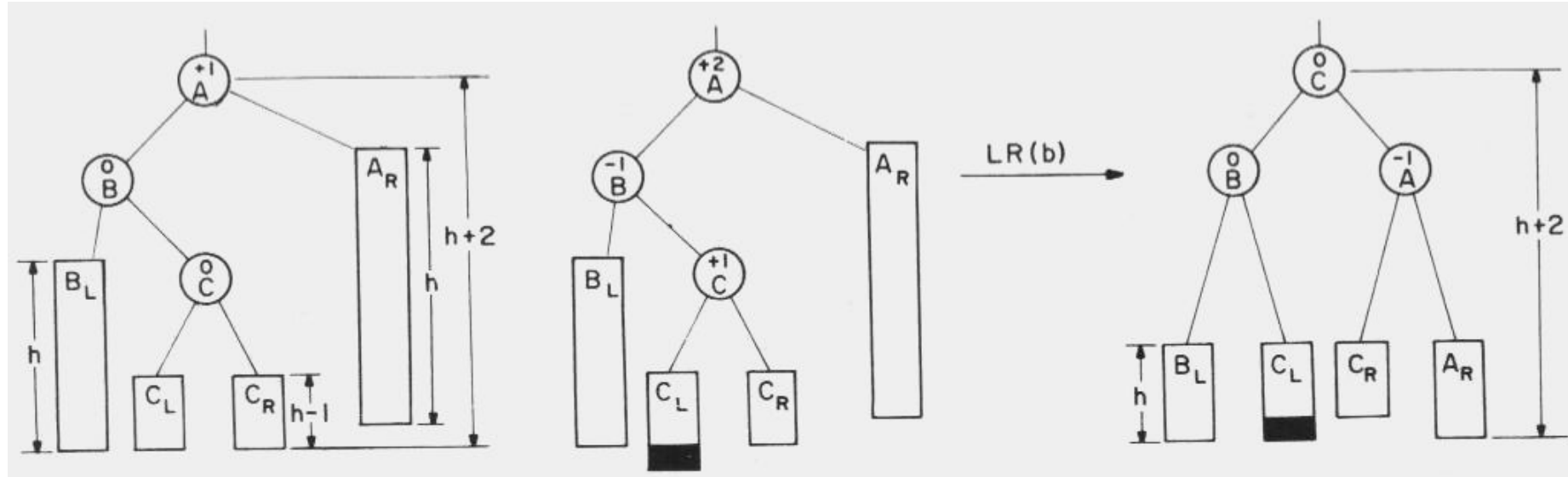


(d) Insert AUGUST

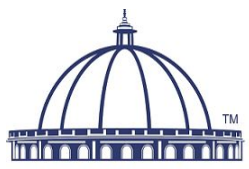


(e) Insert APRIL

# Rebalancing rotations (cont'd)

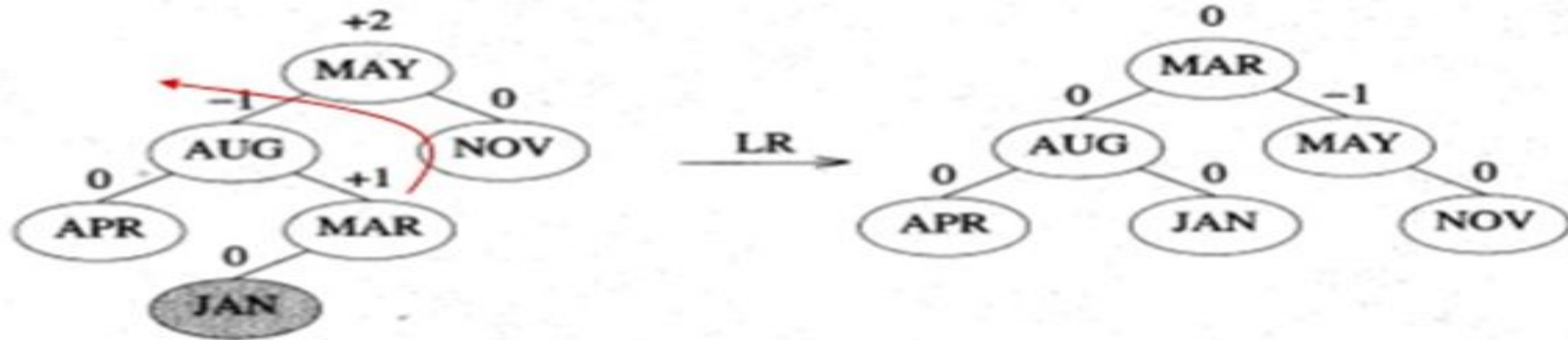






MIT-WPU

॥ विश्वशान्तिर्ध्रुवं ध्रुवा ॥



(f) Insert JANUARY

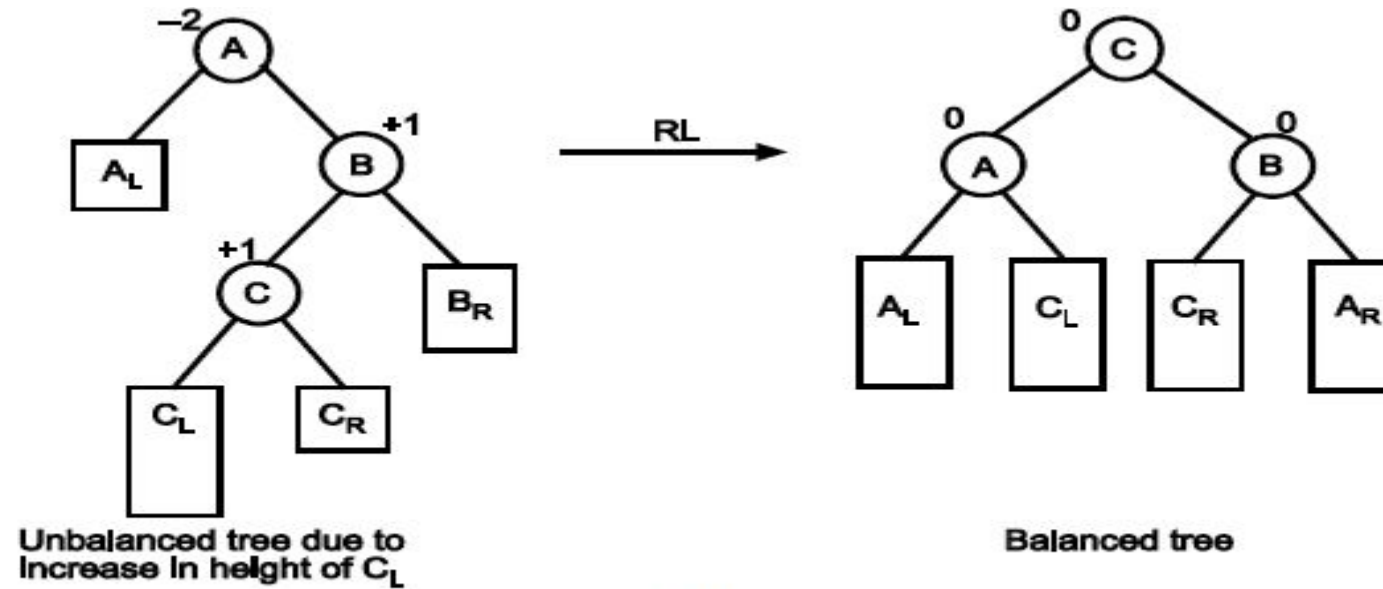


(g) Insert DECEMBER



(h) Insert JULY

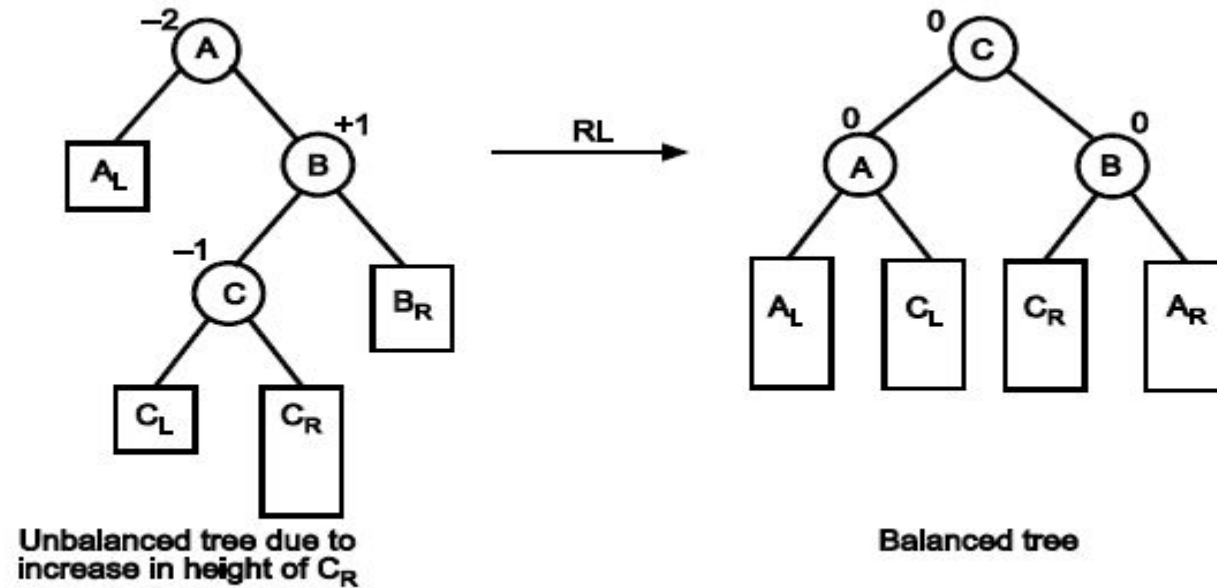
# Case 4: RL(Left of Right)



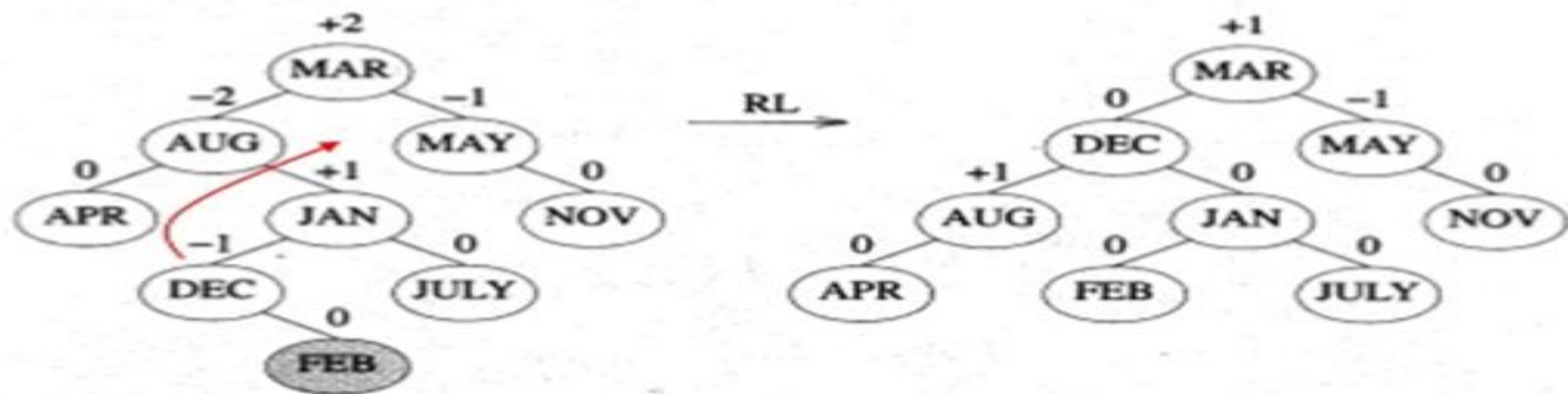
(b)

# Case 4: RL (Left of Right)

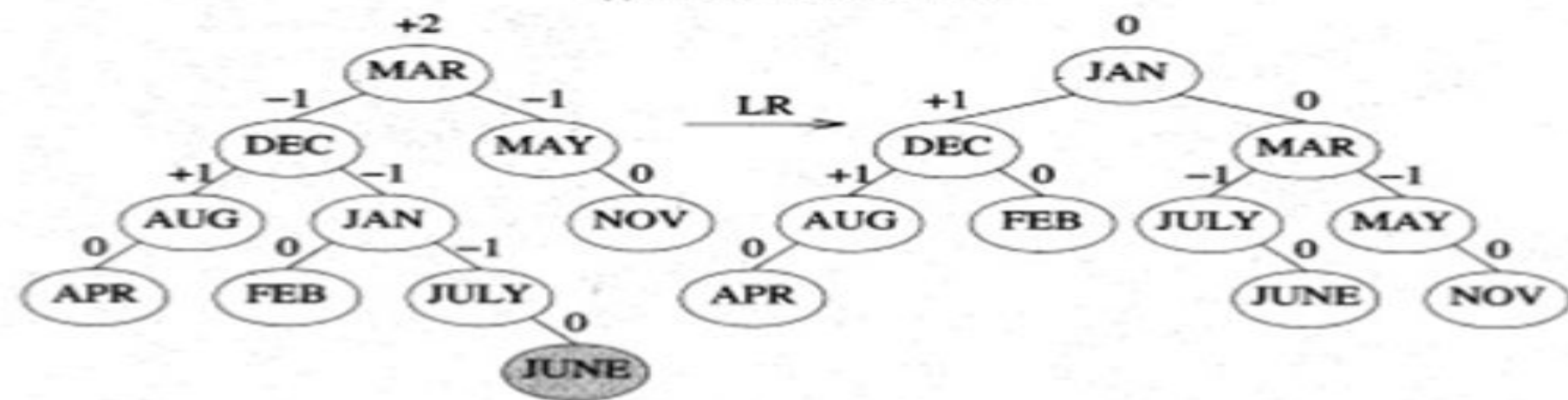
---



(c)

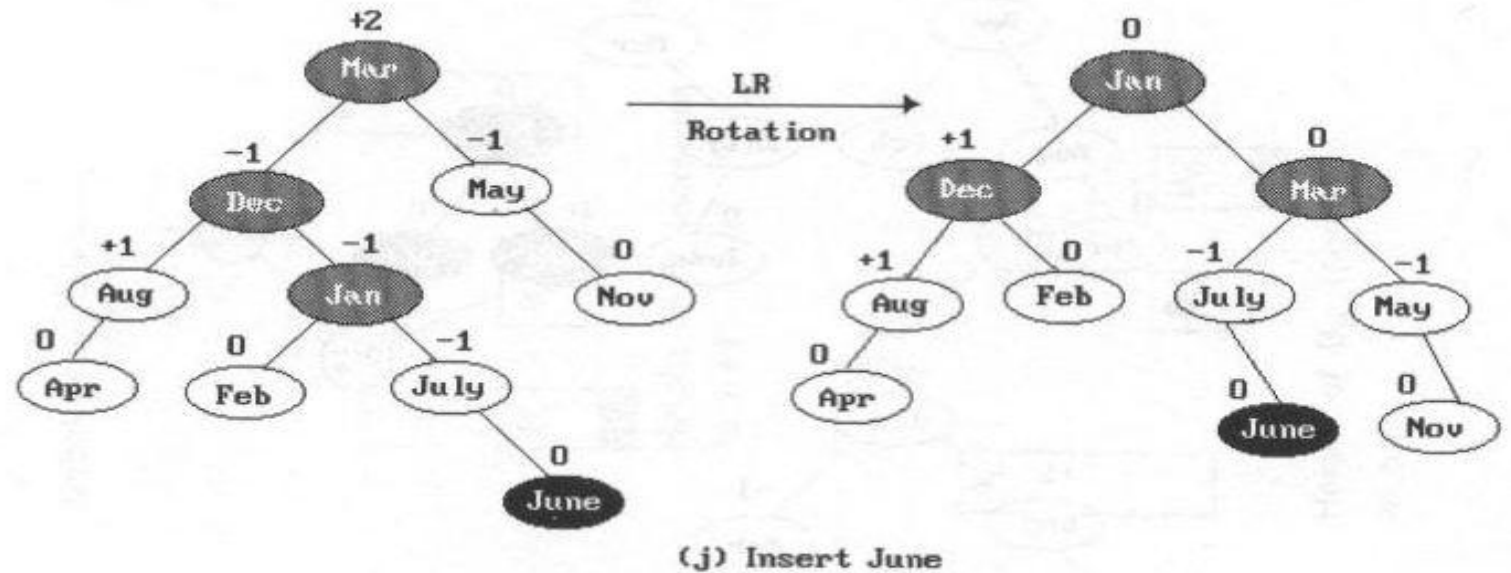
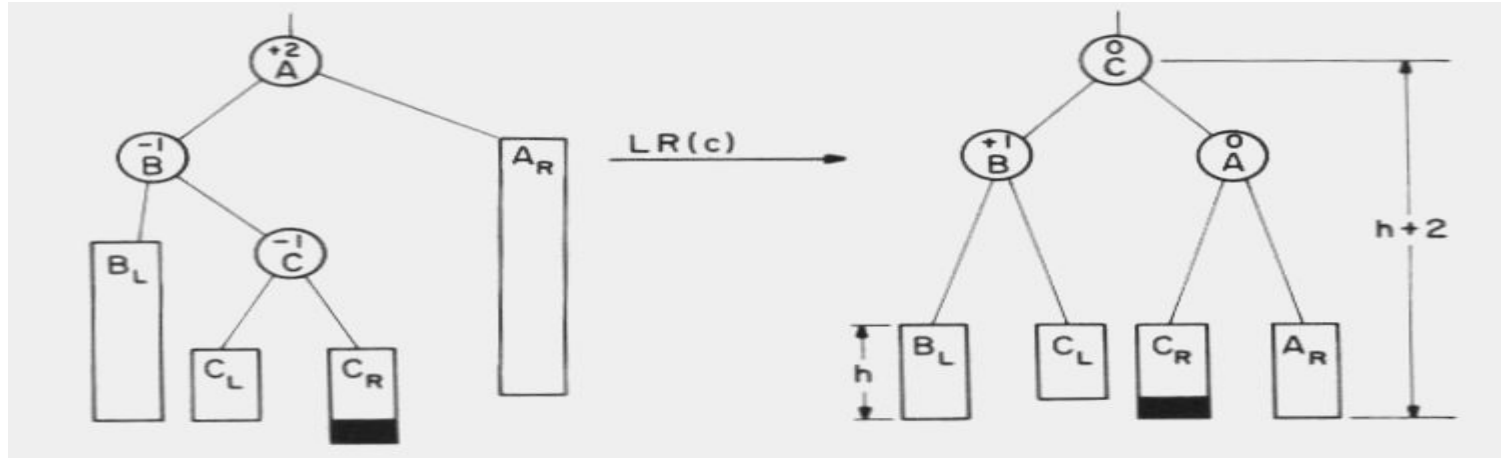


(i) Insert FEBRUARY

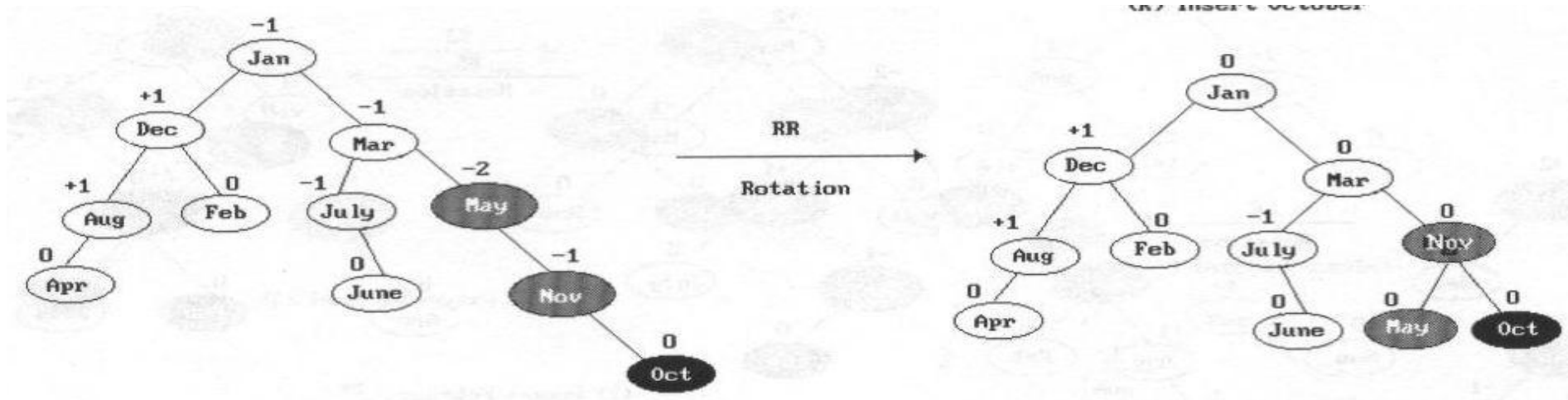
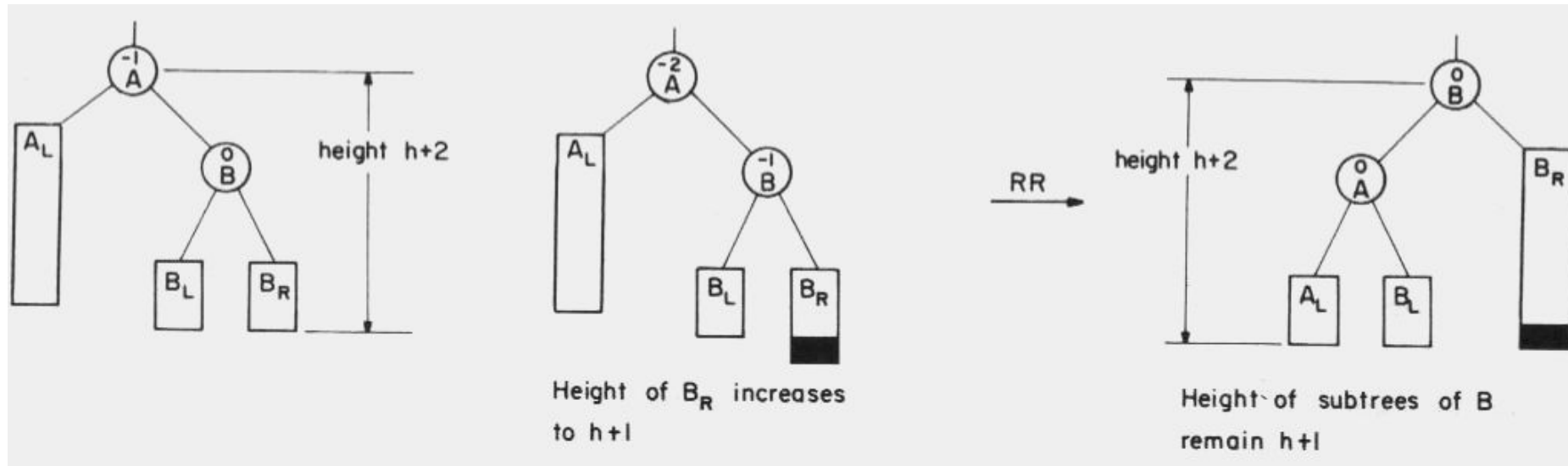


(j) Insert JUNE

# Case 3: LR (Left of Right )

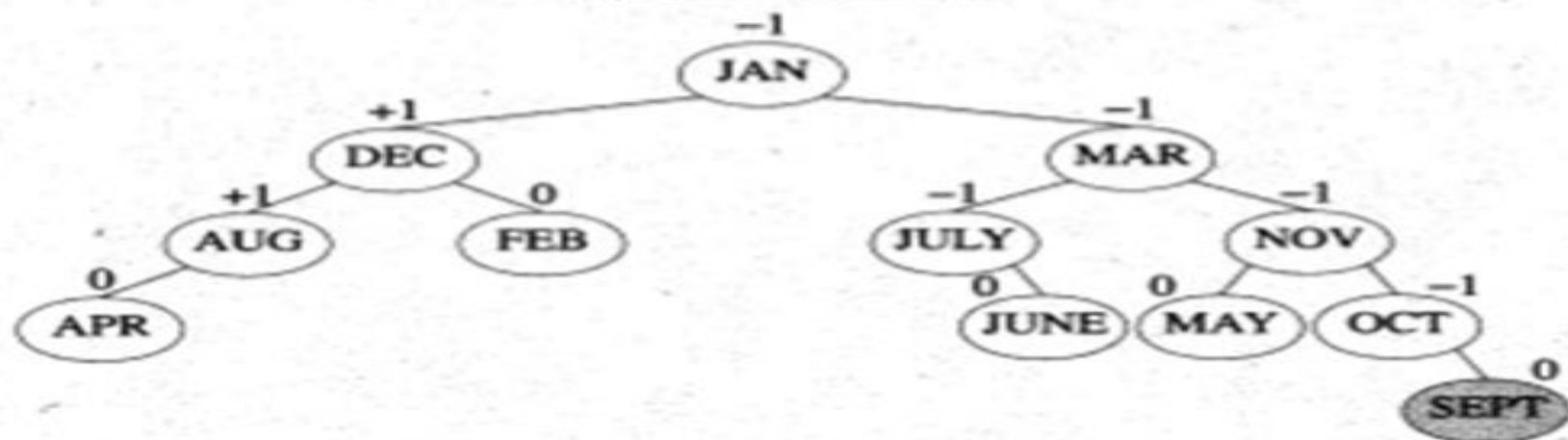


# AVL Trees





(k) Insert OCTOBER



(l) Insert SEPTEMBER

# Node Structure for AVL Tree

---

```
class node
{
    int value;

    node *left, *right;
};
```

```
class avlTree
{
    public:
        int height(avl_node *);
        int diff(avl_node *);
        avl_node *rr_rotation(avl_node *);
        avl_node *ll_rotation(avl_node *);
        avl_node *lr_rotation(avl_node *);
        avl_node *rl_rotation(avl_node *);
        avl_node* balance(avl_node *);
        avl_node* insert(avl_node *, int );
        void display(avl_node *, int);
        void inorder(avl_node *);
        void preorder(avl_node *);
        void postorder(avl_node *);
        avlTree()
        {
            root = NULL;
        }
};
```



# LL Rotation

---

```
Algorithm LL_rotation (node *parent)
{
    temp = parent->left;
    parent->left = temp->right;
    temp->right = parent;
    return temp;
}
```

# RR Rotation

---

```
Algorithm RR_rotation (node *parent)
{
    temp = parent->right;
    parent->right = temp->left;
    temp->left = parent;
    return temp;
}
```



```
Algorithm LR_rotation (node *parent)
{
    temp = parent->left;
    parent->left = RR_rotation (temp); //calling RR rotation
    return LL_rotation (parent);
    // return root after LL rotation
}
```

# RL Rotation

---

```
Algorithm RL_rotation (node *parent)
{
    temp = parent->right;
    parent->right = LL_rotation (temp); // calling LL rotation
    return RR_rotation (parent);
    // return root after RR rotation
}
```

# Insert()

---

```
insert() //workhorse to insert
{
    do
    {
        Accept key to be inserted :";
        root=insert(root, value);
        accept choice for next node
    } while (choice is Y);
}
```

Algorithm insert( node \*root, int value)

```
{  
    if (root == NULL)  
    {  
        root = new avl_node;  
        root->data = value;  
        root->left = NULL;  
        root->right = NULL;  
        return root;  
    }
```

else if (value < root->data)

```
{  
    root->left = insert(root->left, value);  
    root = balance (root);  
}  
else if (value >= root->data)  
{  
    root->right = insert(root->right,  
value);  
    root = balance (root);  
}  
return root;  
}
```

```
Algorithm balance(node *temp)
{
    int bal_factor = diff (temp);
    if (bal_factor > 1)
    {
        if (diff (temp->left) > 0)
            temp = LL_rotation (temp);
        else
            temp = LR_rotation (temp);
    }
}
```

```
else if (bal_factor < -1)
{
    if (diff (temp->right) > 0)
        temp = RL_rotation (temp);
    else
        temp = RR_rotation (temp);
}
return temp;
}
```

```
int avlTree::diff(avl_node *temp)
{
    int l_height = height (temp->left);
    int r_height = height (temp->right);
    int b_factor= l_height - r_height;
    return b_factor;
}
```

```
int avlTree::height(avl_node *temp)
{
    int h = 0;
    if (temp != NULL)
    {
        int l_height = height (temp->left);
        int r_height = height (temp->right);
        int max_height = max (l_height,
                                r_height);

        h = max_height + 1;
    }
    return h;
}
```



```
Algorithm display (node *ptr, int level) // consider level =1
{
    if (ptr!=NULL)
    {
        display(ptr->right, level + 1);
        printf("\n");
        if (ptr == root)
            cout<<"Root -> ";
        for (i = 0; i < level && ptr != root; i++)
            cout<<"    ";
        cout<<ptr->data;
        display(ptr->left, level + 1);
    }
}
```

# Multiway-Search Tree

---

## Introduction:

- ❖ A file is a collection of records, each record having one or more fields
- ❖ The fields used to distinguish among the records are known as keys
- ❖ File organization describes the way where the records are stored in a file
- ❖ File organization is concerned with representing data records on an external storage media
- ❖ Mode of Retrieval:
  - Real time : response time for any query should be minimized.
  - Batched: Response time is not very significant.
  - ❖ Real time: Reservation system
  - ❖ Batched: Bank account.

# FILE ORGANIZATION

---

## Number of keys:

- ❖ Files having only one key & files with more than one key.
- ❖ One key-records may be stored on this key and stored sequentially either on tape or disk.(batch retrieval)
- ❖ More than one keys or real time responses are needed, sequential organization is not adequate.
- ❖ Several indices have to be maintained
- ❖ File organization breaks down into two aspects:
  - ❖ Directory—for collection of indices
  - ❖ File organization—for the physical organization of records

- 
- ❖ File organization is the way records are organized on a physical storage
  - ❖ One of such organizations is sequential (ordered and unordered)
  - ❖ In this general framework, processing a query or updating a request would proceed in two steps:
    - ❖ The indices would be interrogated to determine the parts of the physical file to be searched
    - ❖ These parts of the physical file will be searched

# Indexing

---

- ❖ An index, whether it is a book or a data file index (in computer memory), is based on the basic concepts such as keys and reference fields
- ❖ The index to a book provides a way to find a topic quickly

# Indexing techniques

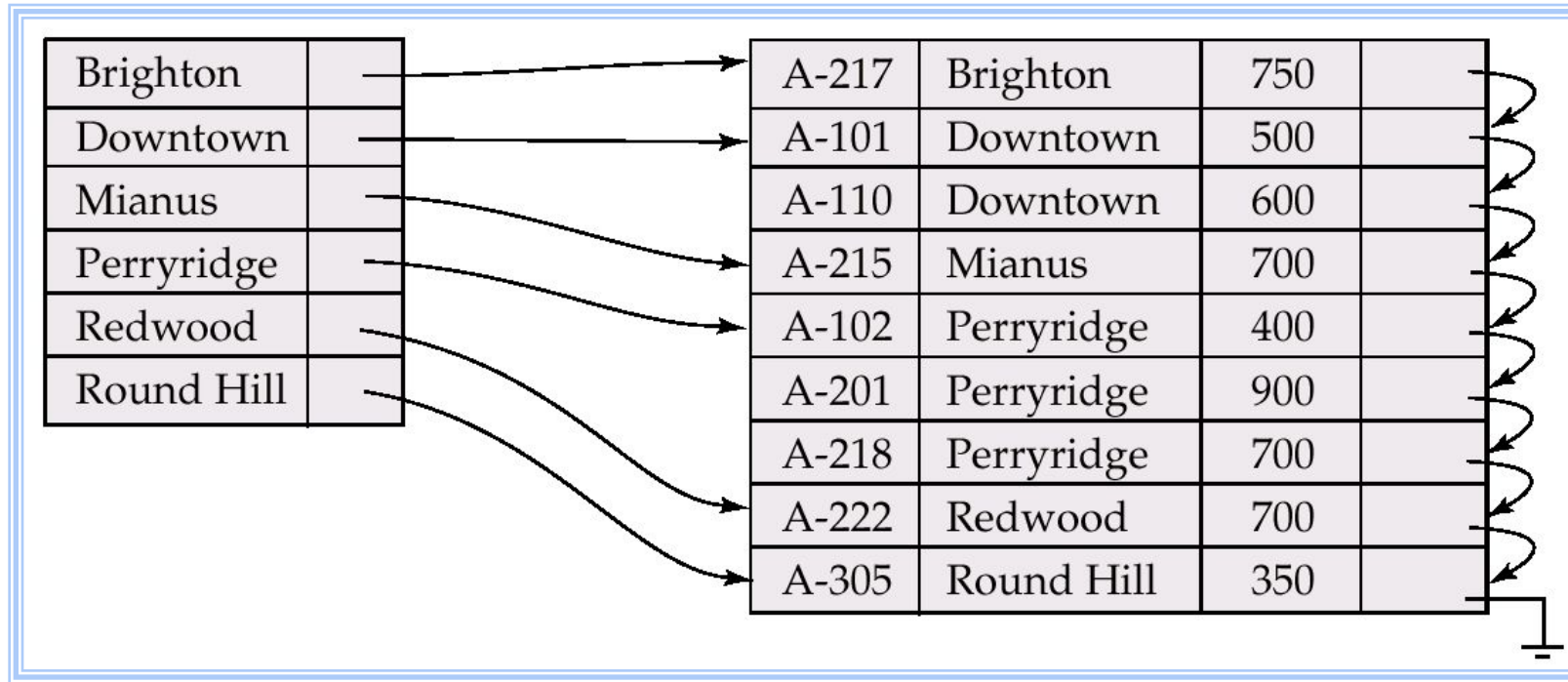
---

- ❖ A directory is a collection of indices.
- ❖ Directory contain one index for every key or only one index for some of the keys.
- ❖ If an index contains an entry for every record, then it is called as dense index.
- ❖ If an index contains an entry for only some of the records, then it is called as non-dense index.

# Dense Index Files

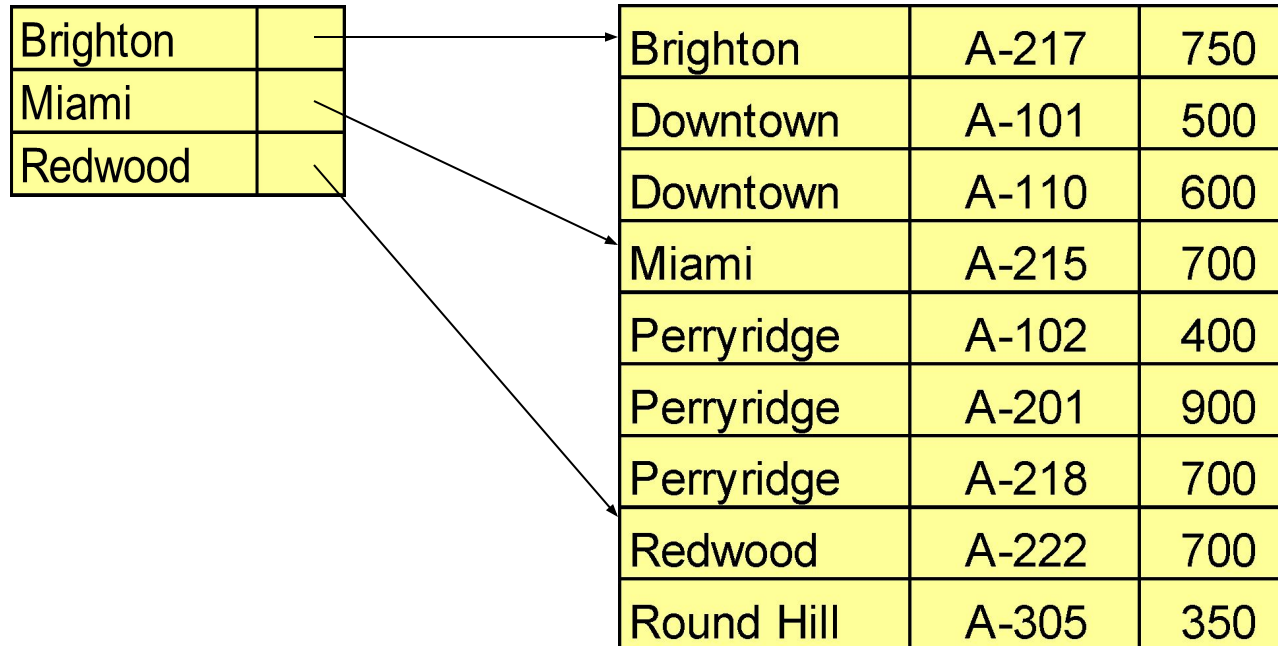
**Dense index** — Index record appears for every search-key value in the file.

---



# Example of Sparse Index Files

---





- 
- ❖ An index, whether it is a book or a data file index (in computer memory), is based on the basic concepts such as keys and reference fields
  - ❖ The index to a book provides a way to find a topic quicklyAn index, whether it is a book or a data file index (in computer memory), is based on the basic concepts such as keys and reference fields
  - ❖ The index to a book provides a way to find a topic quickly.

# Cylinder-Surface Indexing

---

- ❖ This is the simplest type of index organization. It is useful only for the primary key index of a sequentially ordered file
- ❖ In a sequentially ordered file, the physical sequence of records is ordered by the key, called the primary key
- ❖ The cylinder-surface index consists of a cylinder index and several surface indexes
- ❖ For each cylinder, there is a surface index. If the disk has  $S$  usable surfaces, then each surface index has  $s$  entries. The total number of surface index entries is  $C.S$

Emp. No.	Emp. Name	Cylinder	Surface
1	Aboleer	1	1
2	Anand	1	1
3	Amit	1	2
4	Amol	1	2
5	Rohit	2	1
6	Santosh	2	1
7	Saurabh	2	2
8	Shila	2	2

- ❖ Let there be two surfaces and two records stored per track. The file is organized sequentially on the field 'Emp. name'
- ❖ The cylinder index is shown in following table

Emp. No.	Highest Key Value
1	Amol
2	Shila

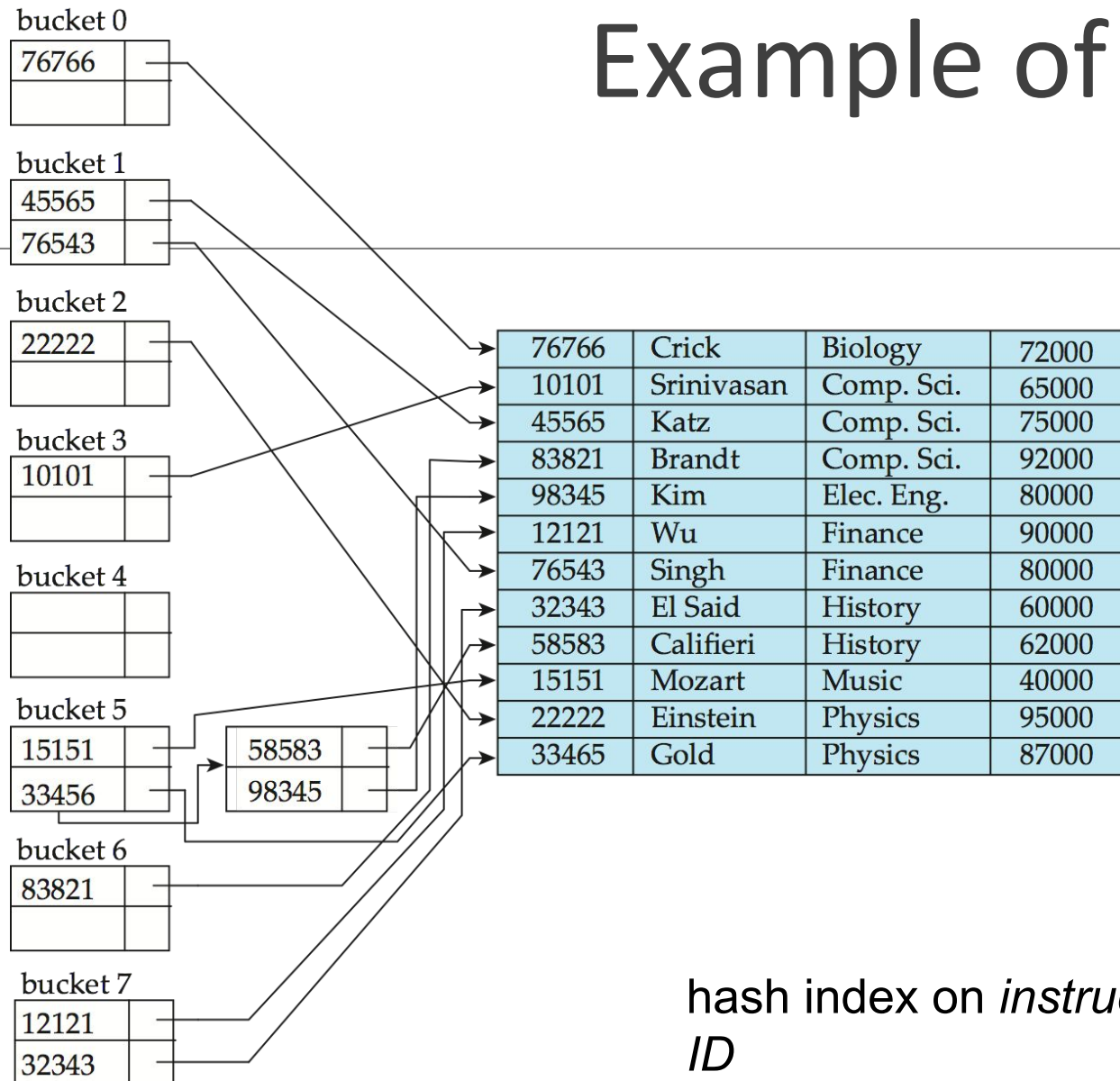
- ❖ This method of maintaining a file and index is referred to as ISAM (indexed sequential access method)
- ❖ It is the simplest file organization for single key files but not useful for multiple key files

# Hash Indices

---

- ❖ Hashing can be used not only for file organization, but also for index-structure creation.
- ❖ A **hash index** organizes the search keys, with their associated record pointers, into a hash file structure.
- ❖ Strictly speaking, hash indices are always secondary indices
  - ❖ if the file itself is organized using hashing, a separate primary hash index on it using the same search-key is unnecessary.
  - ❖ However, we use the term hash index to refer to both secondary index structures and hash organized files.

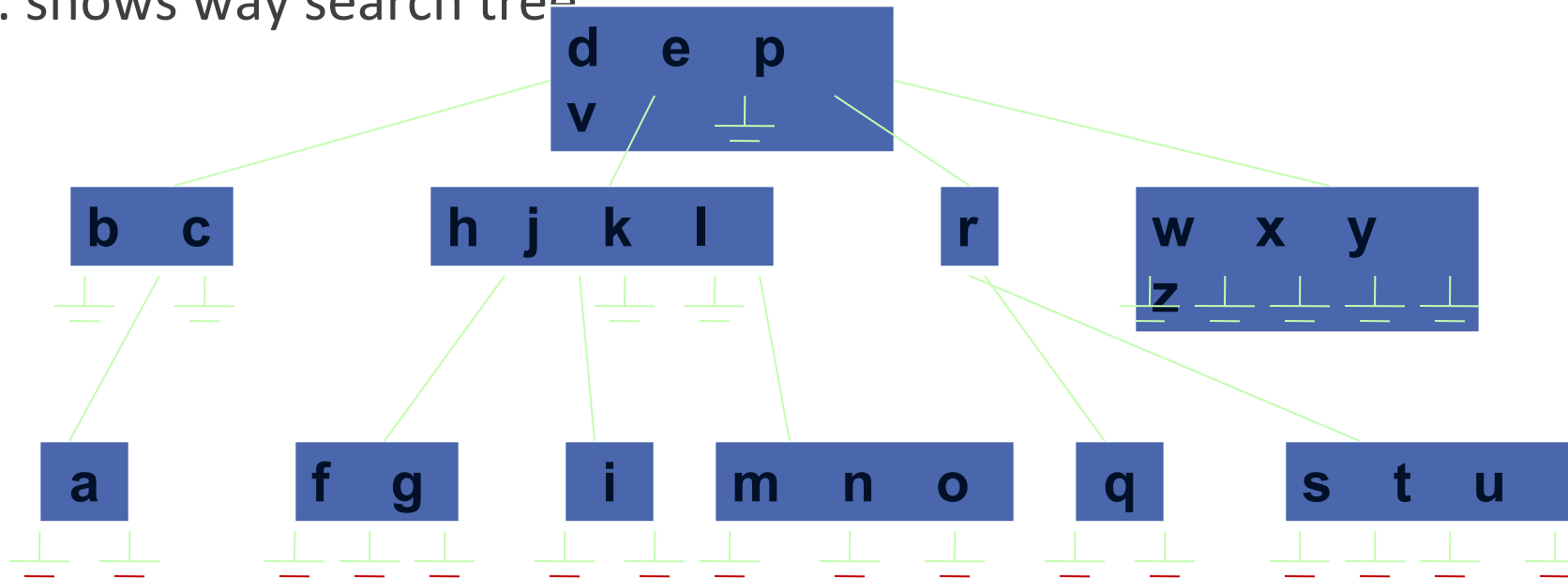
# Example of Hash Index



hash index on *instructor*, on attribute *ID*

# Multiway Search Trees

- ❖ A multiway search tree is a tree of order  $m$ , where each node has utmost  $m$  children
- ❖ Fig. shows way search tree:



# M-Way Trees

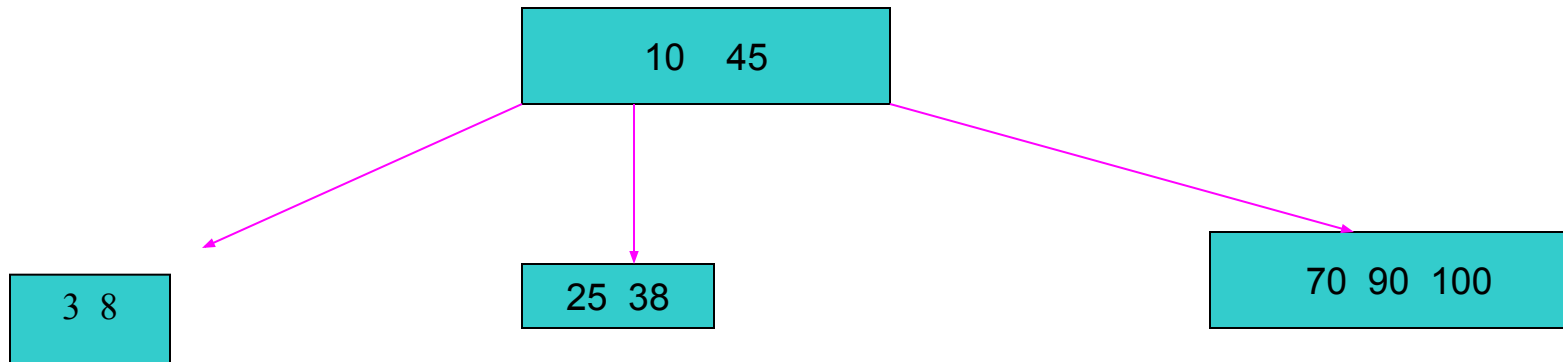
---

- ❖ M-way tree is a search tree.
- ❖ Each node of tree can have from zero to  $m$  subtrees,
- ❖  $m$  is defined as the order of the tree.

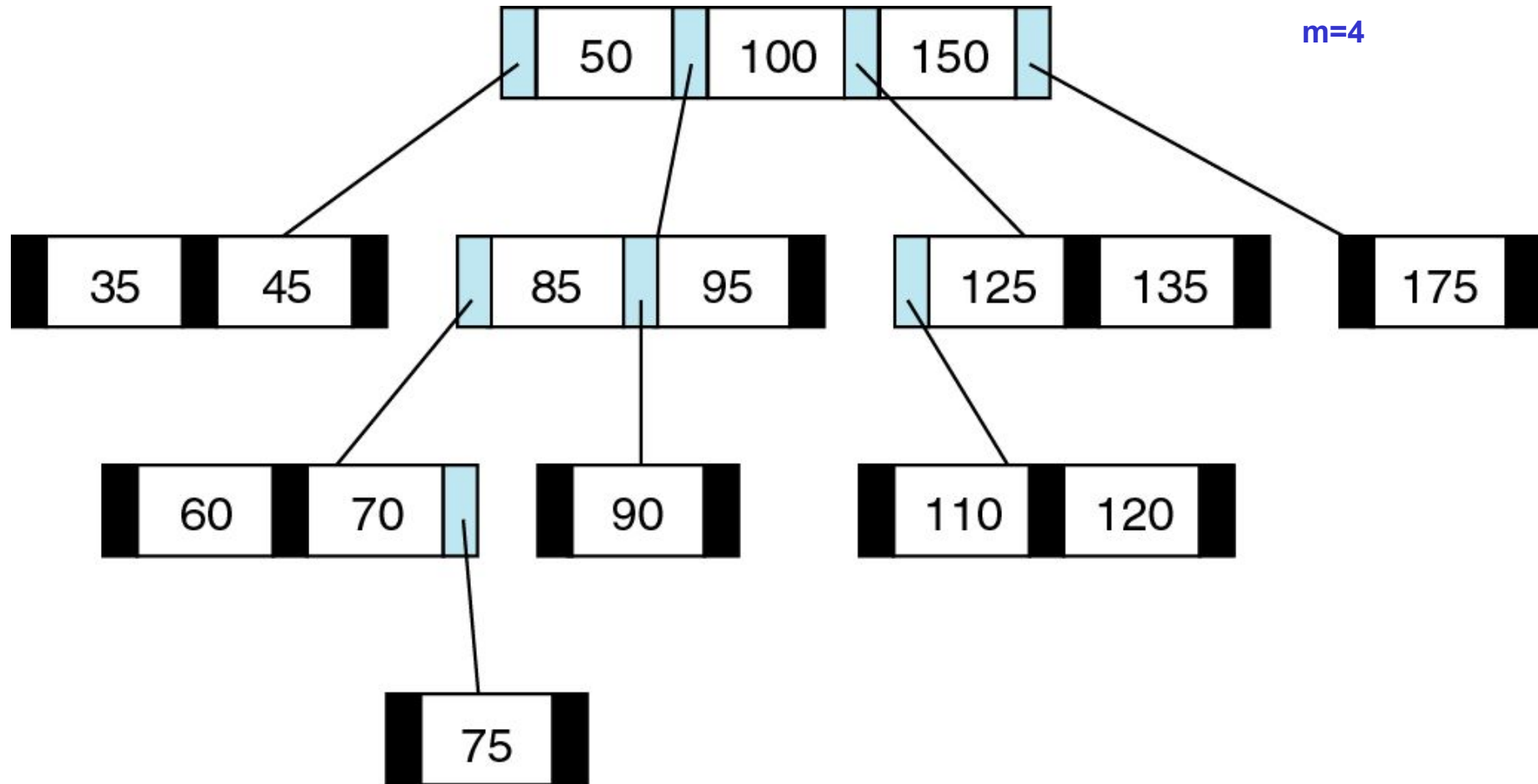


# Example

---

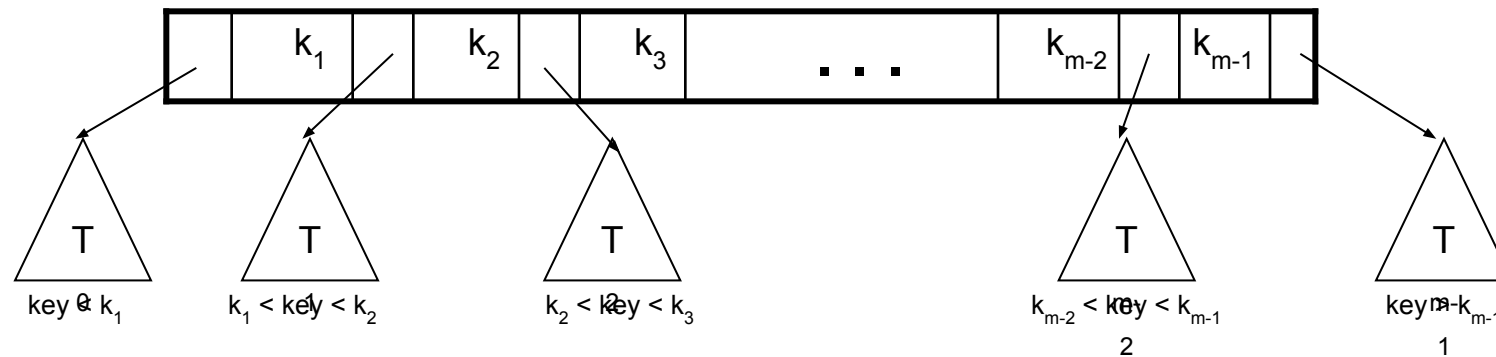


# M-Way Trees



# What is a Multi-way tree?

- ❖ A multi-way (or m-way) search tree of order m is a tree in which
  - ❖ Each node has at-most **m** subtrees, where the subtrees may be empty.
  - ❖ Each node consists of at least **1** and at most **m-1** distinct keys
- ❖ The keys in each node are sorted.



- ❖ The keys and subtrees of a non-leaf node are ordered as:
  - ❖  $T_0, k_1, T_1, k_2, T_2, \dots, k_{m-1}, T_{m-1}$  such that:
    - ❖ All keys in subtree  $T_0$  are less than  $k_1$ .
    - ❖ All keys in subtree  $T_i$ ,  $1 \leq i \leq m-2$ , are greater than  $k_i$  but less than  $k_{i+1}$ .
    - ❖ All keys in subtree  $T_{m-1}$  are greater than  $k_{m-1}$ .

# Definition of a B-tree

---

- ❖ A B-tree of order  $m$  is an  $m$ -way tree (i.e., a tree where each node may have up to  $m$  children) in which:
  - ❖ The number of keys in each non-leaf node is one less than the number of its children and these keys partition the keys in the children in the fashion of a search tree
  - ❖ All leaves are on the same level
  - ❖ All non-leaf nodes except the root have at least  $\lceil m / 2 \rceil$  children
  - ❖ The root is either a leaf node, or it has from two to  $m$  children
  - ❖ A leaf node contains no more than  $m - 1$  keys

# Use of B-tree

---

- ❖ To understand use of B-Trees, we must think of huge amount of data that cannot fit in main memory.
- ❖ When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to main memory access time.
- ❖ The main idea of using B-Trees is to reduce the number of disk accesses.
- ❖ Most of the tree operations (search, insert, delete, max, min, ..etc ) require  $O(h)$  disk accesses where  $h$  is height of the tree.

---

Create B-tree of order 3 for the data values:  
78, 21, 14, 11, 97, 85, 74, 63, 45, 42, 57,  
20, 16, 19, 52, 30, 21

# Insertion

---

Insertion:

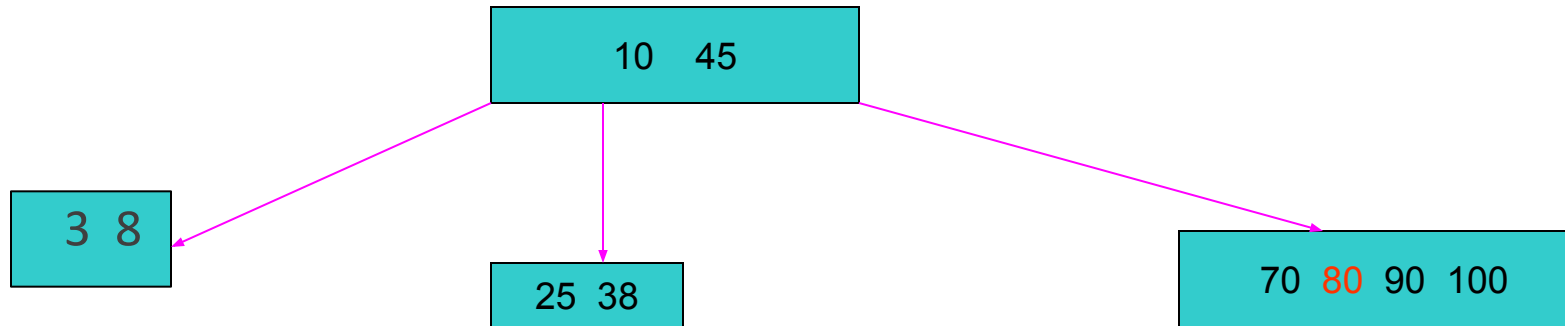
Find the appropriate leaf. If there is only one or two items, just add to leaf.

If no room, move middle item to parent and split remaining two items among two children.

# Insertion

---

insert 80



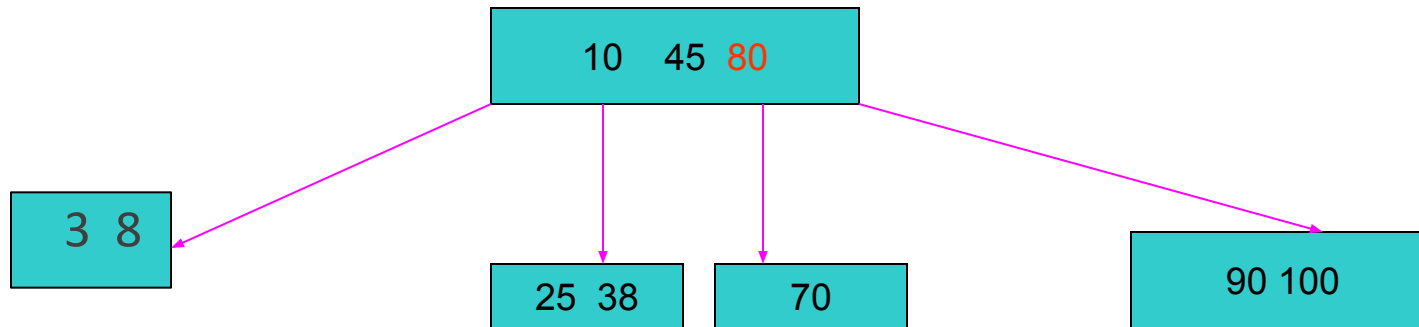
Overflow!



# Insertion

---

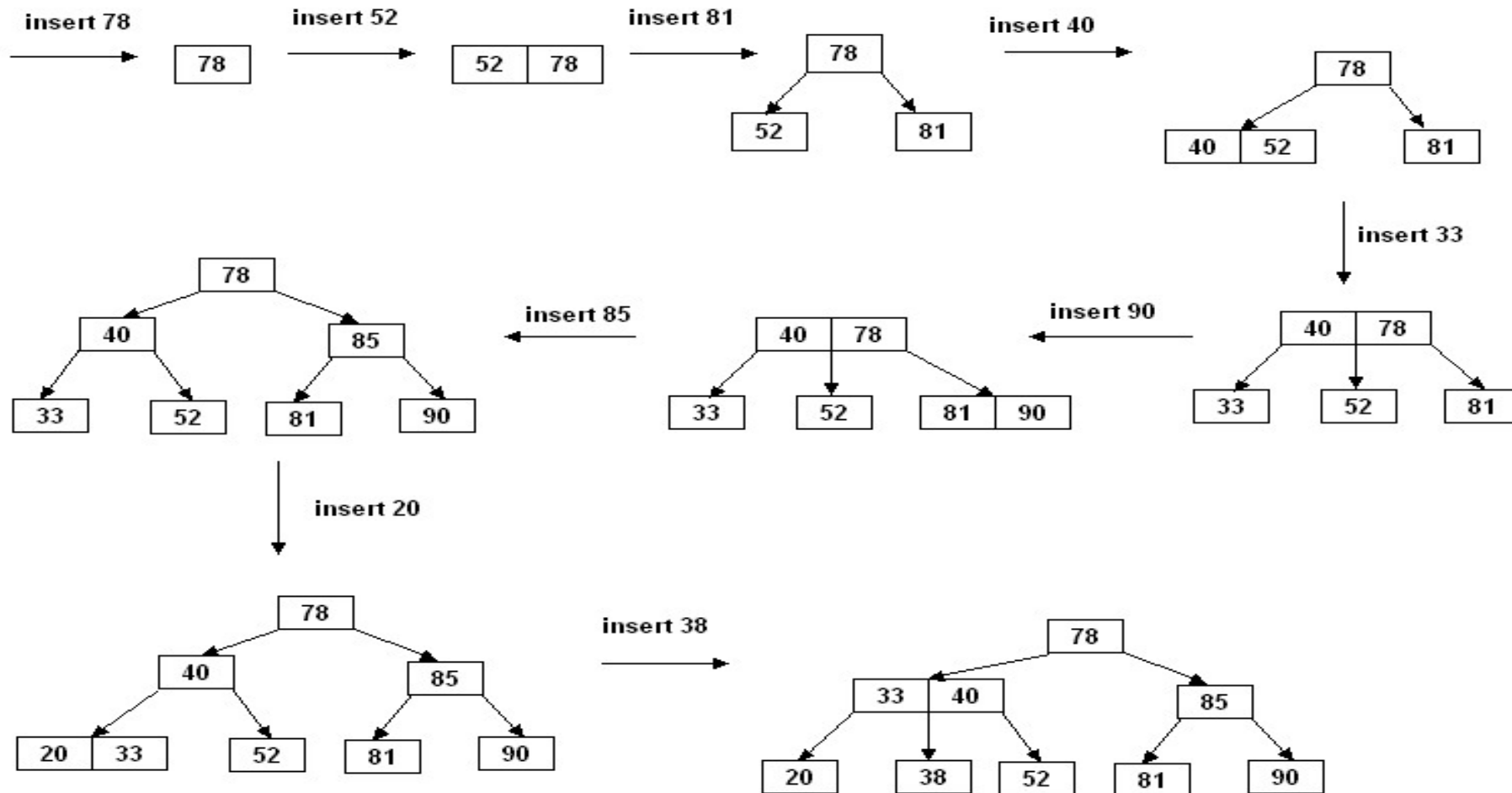
Split & move middle element to parent



# Insertion in B-Trees

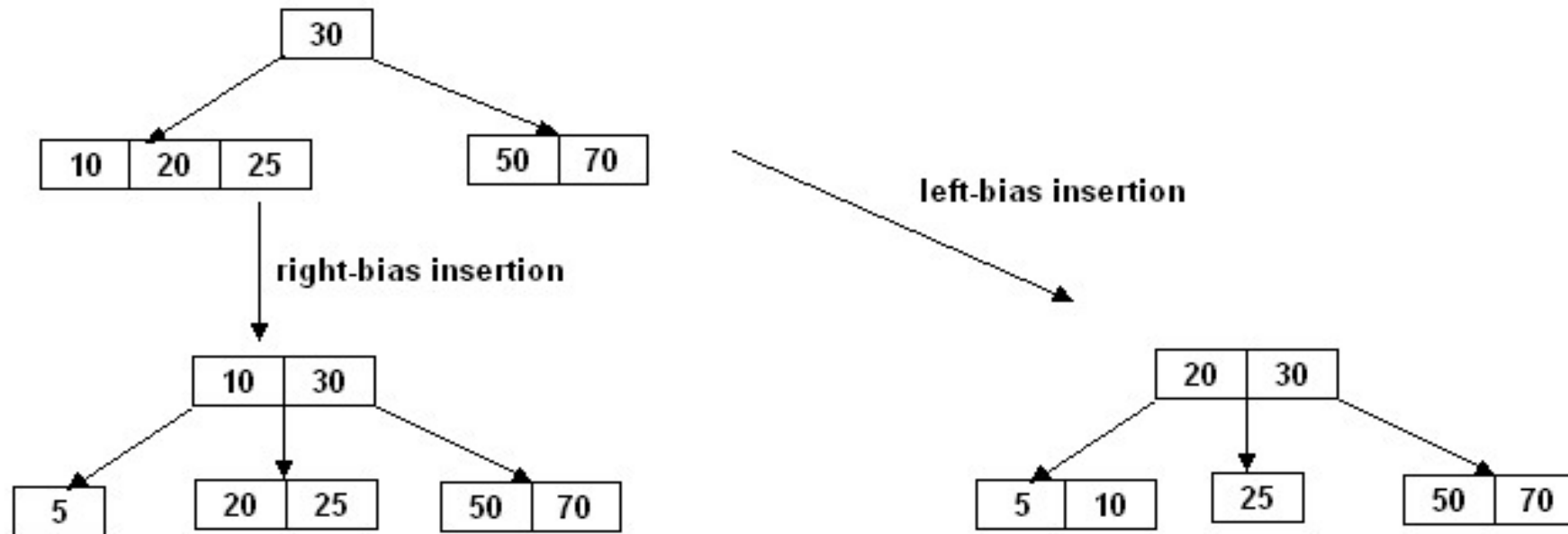
## Insertion in a B-tree of odd order

Example: Insert the keys 78, 52, 81, 40, 33, 90, 85, 20, and 38 in this order in an initially empty B-tree of order 3



# Insertion in B-Trees

- ◆ Insertion in a B-tree of even order
- ◆ At each node the insertion can be done in two different ways:
- ◆ **right-bias:** The node is split such that its right subtree has more keys than the left subtree.
- ◆ **left-bias:** The node is split such that its left subtree has more keys than the right subtree.
- ◆ **Example:** Insert the key 5 in the following B-tree of order 4:



# Insertion

---

- ❖ B-tree of order 5:
- ❖ C N G A H E K Q M F W L T Z D P R X Y S
- ❖ Order 5 means that a node can have a maximum of 5 children and 4 keys.
- ❖ All nodes other than the root must have a minimum of 2 keys.

---

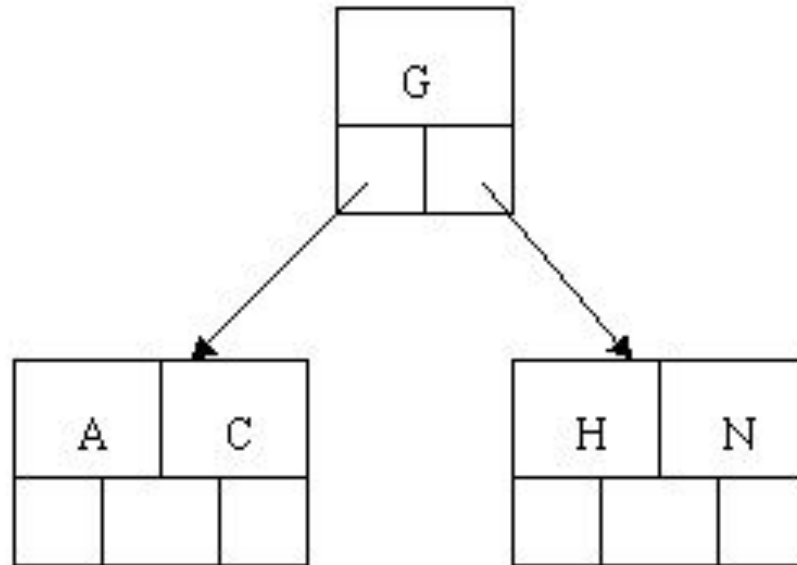
C N G A Order this ACGN

Inserting ACGN

A	C	G	N

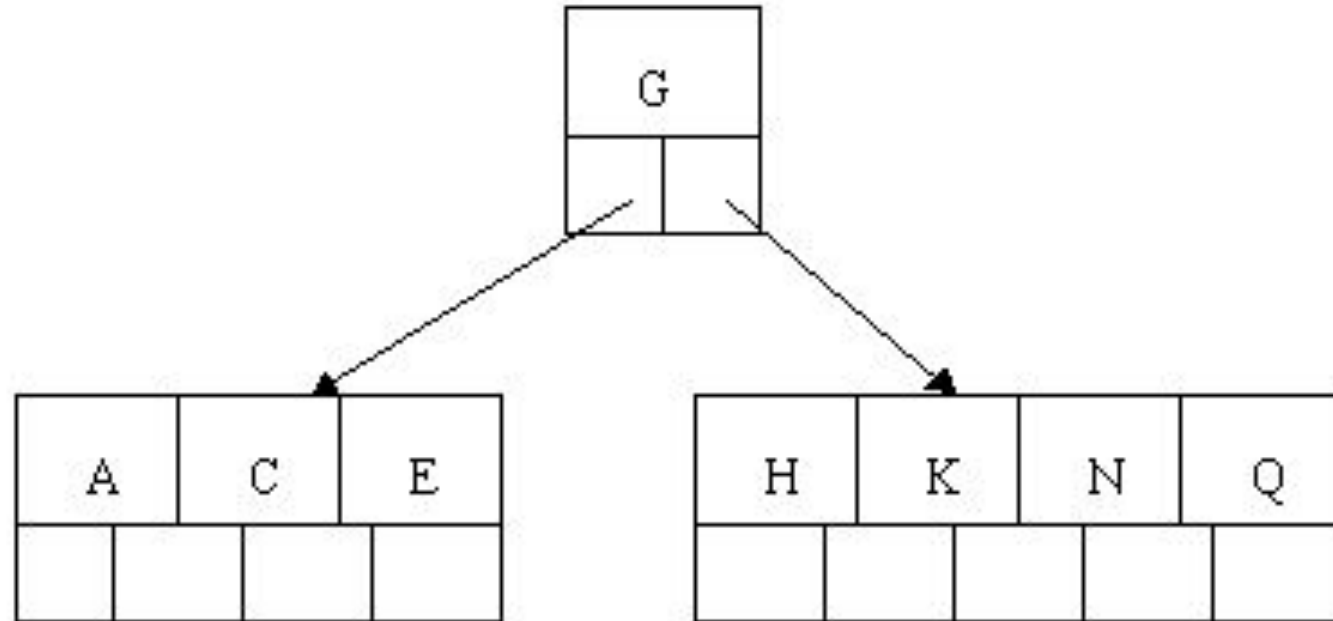
# Inserting H

---



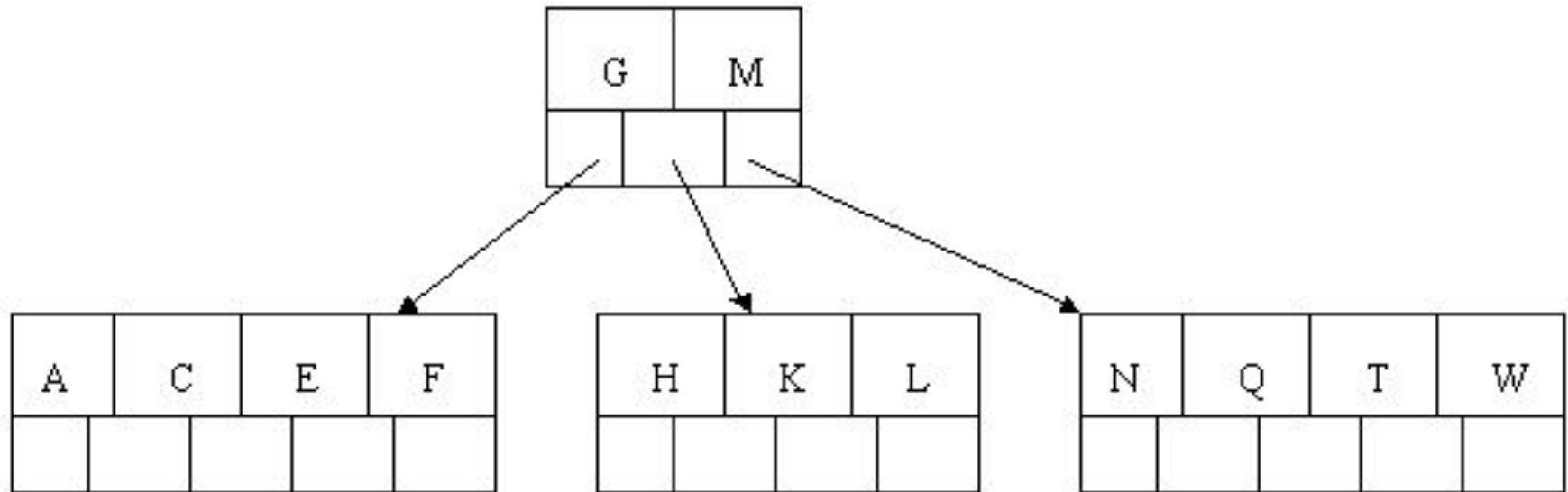
Inserting E, K, and Q proceeds without requiring any splits:

---



The letters F, W, L, and T are then added without needing any split

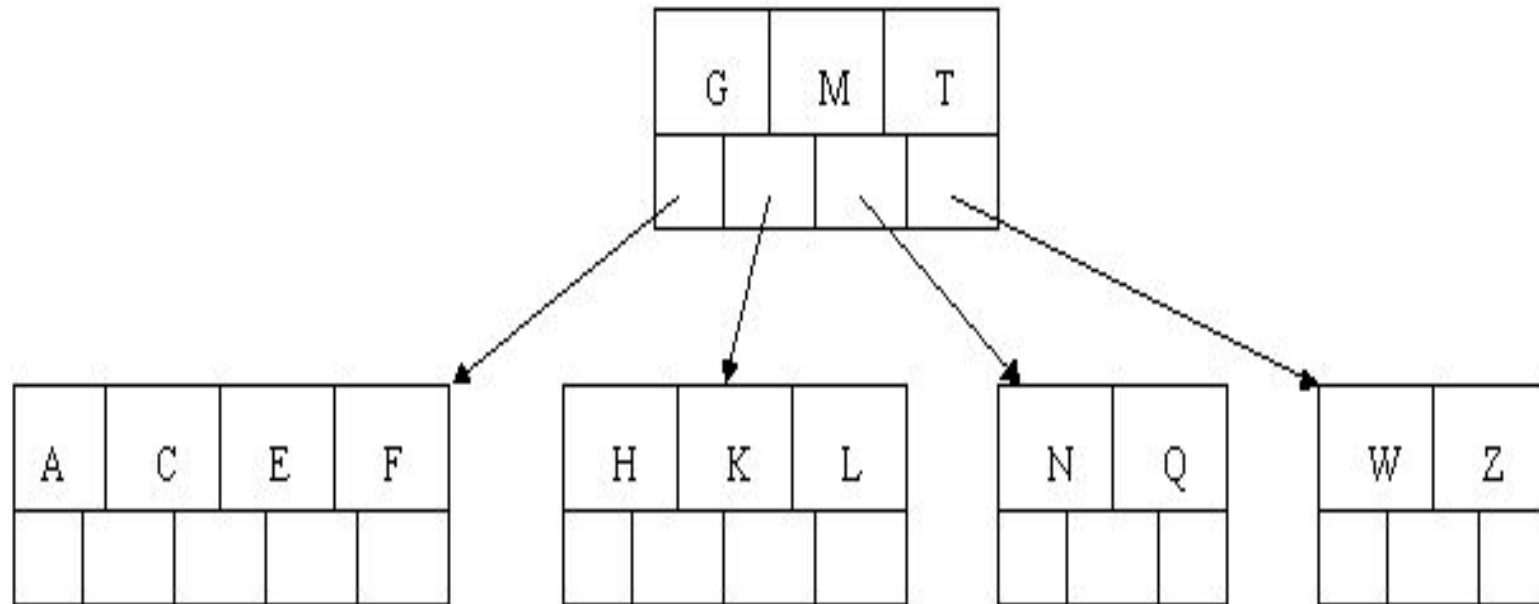
---





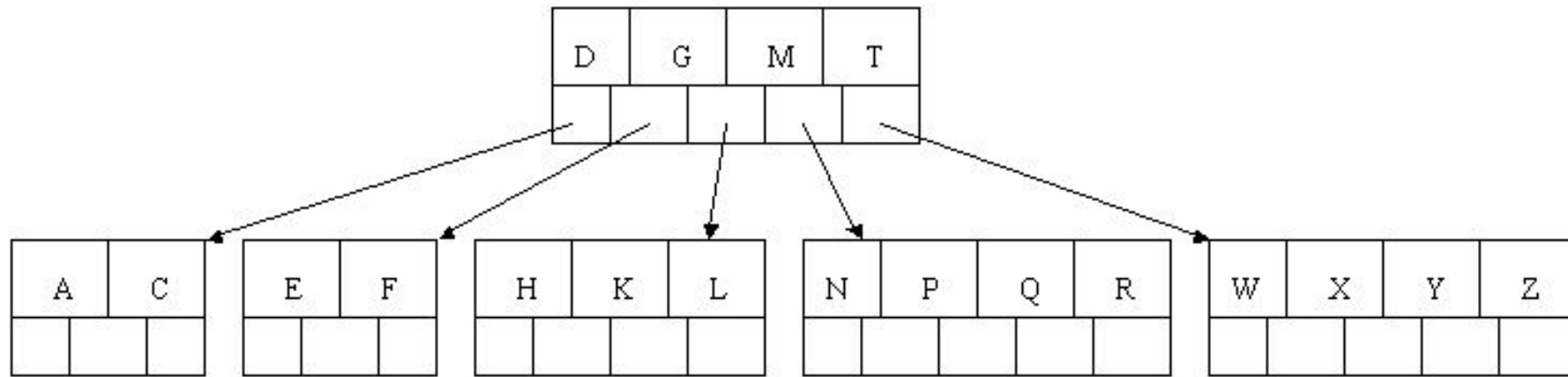
# Adding Z

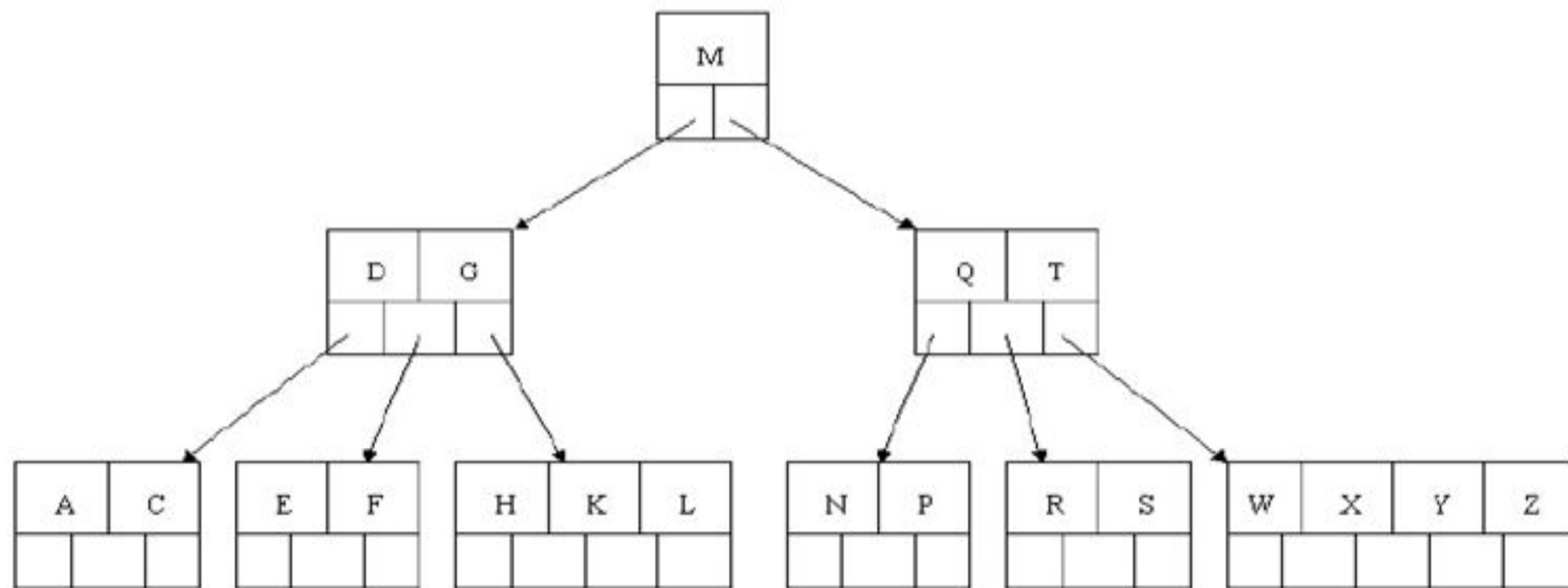
---



# Inserting D

---





# Removal from a B-tree

---

- ❖ During insertion, the key always goes *into* a *leaf*. For deletion we wish to remove *from* a leaf. There are three possible ways we can do this:
- ❖ 1 - 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.
- ❖ 2 - If the key is *not* in a leaf then it is guaranteed (by the nature of a B-tree) that its predecessor or successor will be in a leaf -- in this case can we delete the key and promote the predecessor or successor key to the non-leaf deleted key's position.

# Removal from a B-tree (2)

---

- ❖ If (1) or (2) lead to a leaf node containing less than the minimum number of keys then we have to look at the siblings immediately adjacent to the leaf in question:
  - ❖ 3: if one of them has more than the min number of keys then we can promote one of its keys to the parent and take the parent key into our lacking leaf
  - ❖ 4: if neither of them has more than the min' number of keys then the lacking leaf and one of its neighbours can be combined with their shared parent (the opposite of promoting a key) and the new leaf will have the correct number of keys; if this step leaves the parent with too few keys then we repeat the process up to the root itself, if required

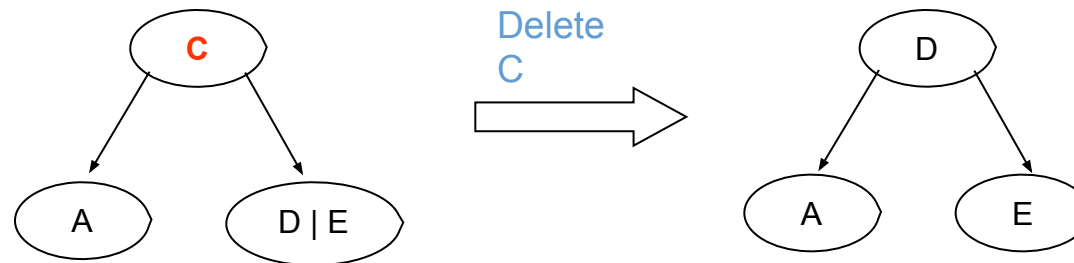
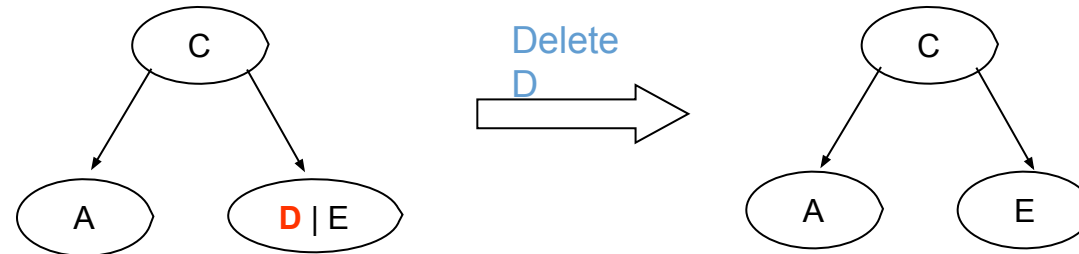
# Deletion

---

- ❖ If the entry to be deleted is not in a leaf, swap it with its successor (or predecessor) under the natural order of the keys. Then delete the entry from the leaf.
- ❖ If leaf contains more than the minimum number of entries, then one can be deleted with no further action.

# Deletion Example 1

---



Successor is promoted, Element D  
C is Deleted.

# Deletion Cont...

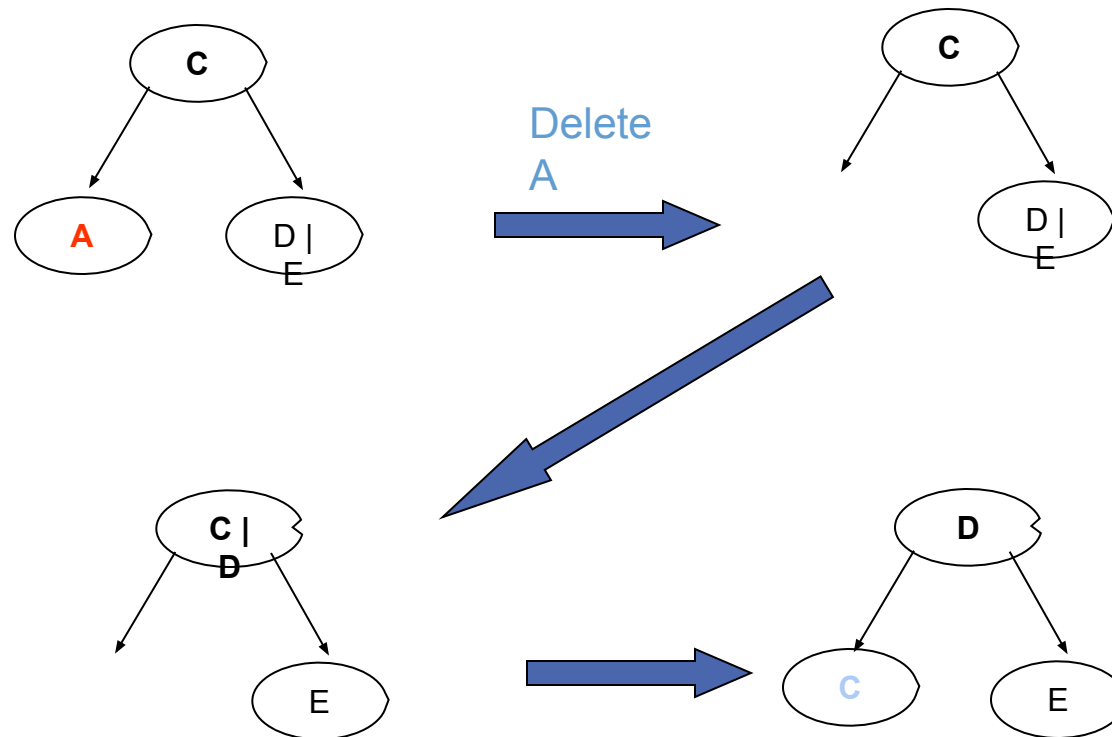
---

- ❖ If the node contains the minimum number of entries, consider the two immediate siblings of the parent node:
- ❖ If one of these siblings has more than the minimum number of entries, then redistribute one entry from this sibling to the parent node, and one entry from the parent to the deficient node.
- ❖ This is a rotation which balances the nodes
- ❖ **Note: all nodes must comply with minimum entry restriction.**



# Deletion Example 2

---

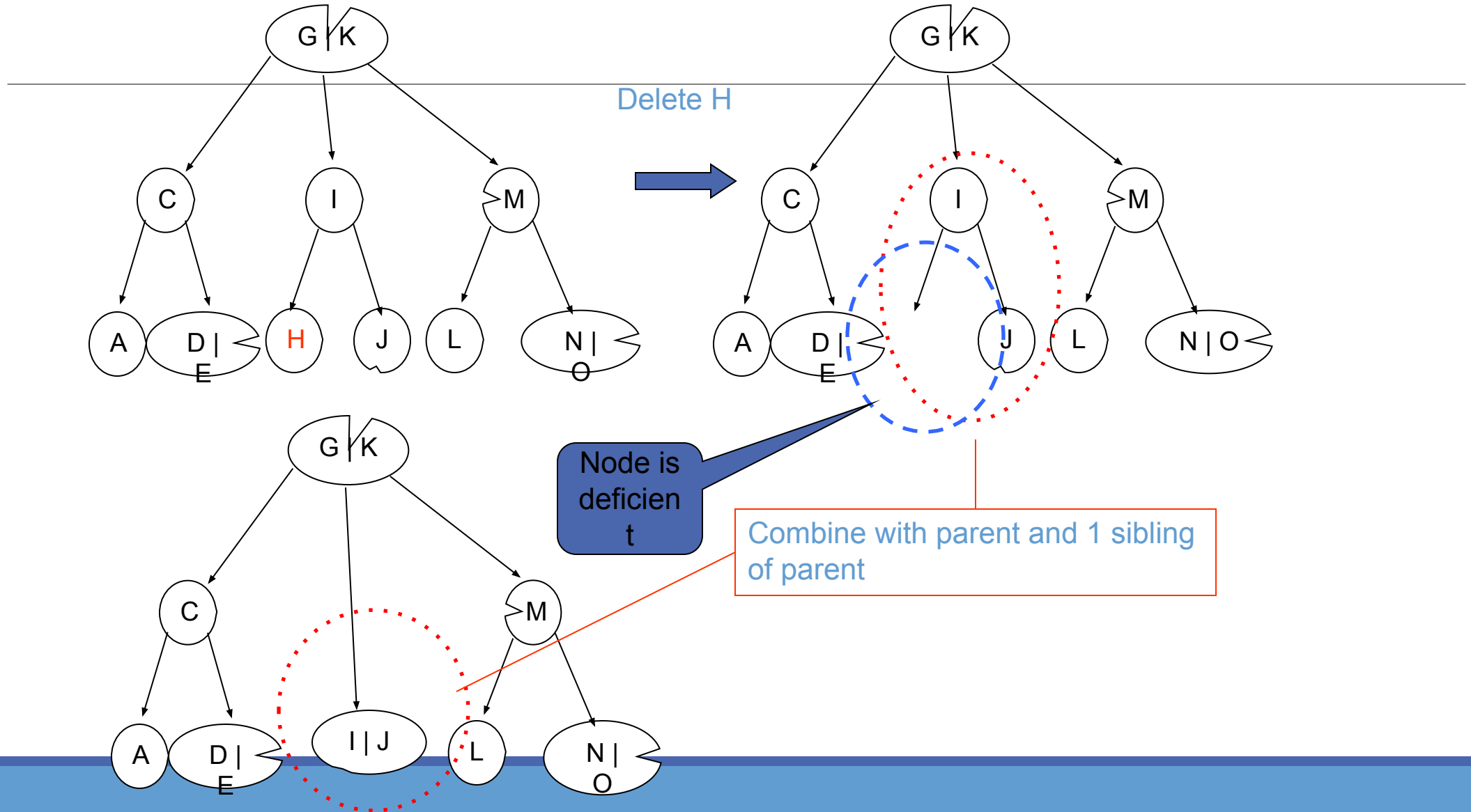


# Deletion Cont...

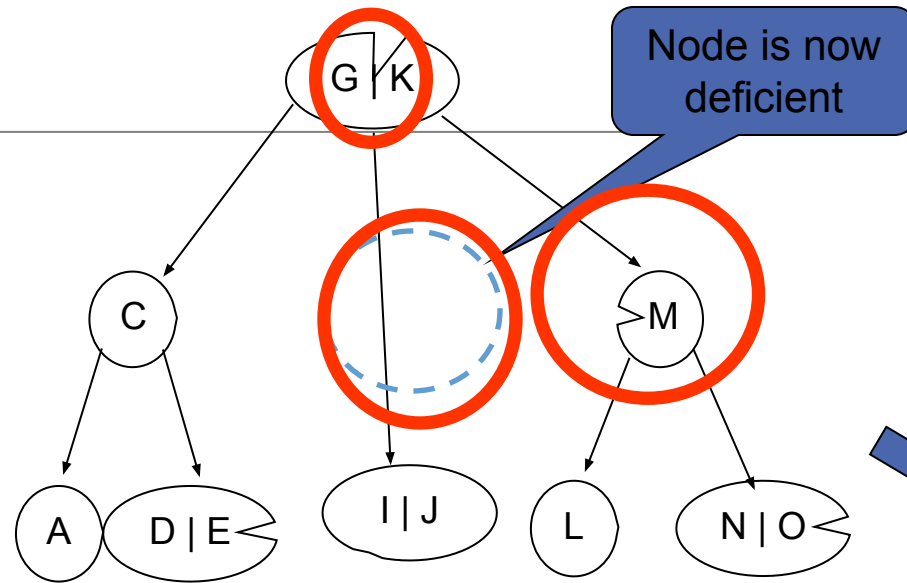
---

- ❖ If both immediate siblings have exactly the minimum number of entries, then merge the deficient node with one of the immediate sibling node and one entry from the parent node.
- ❖ If this leaves the parent node with too few entries, then the process is propagated upward.

# Deletion Example 3

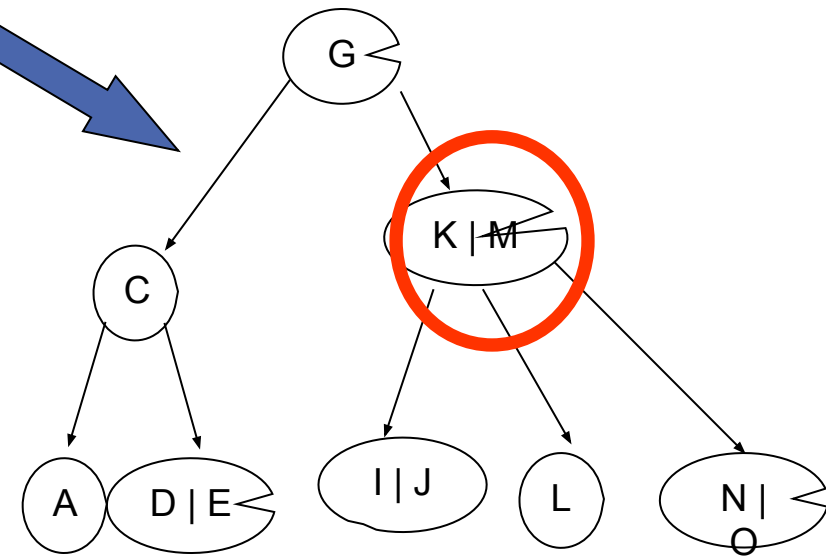


# Deletion Example 3 Cont..



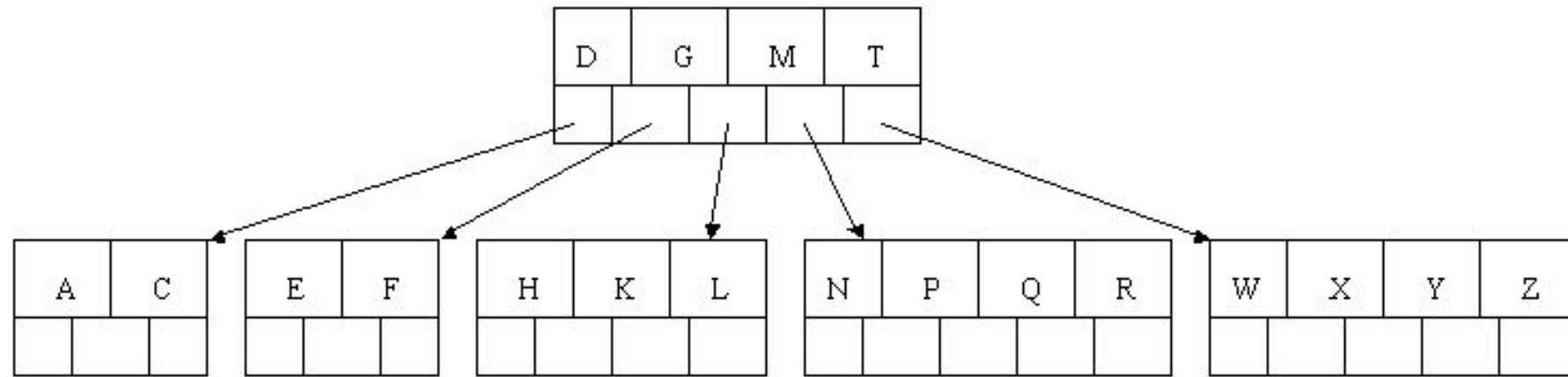
Deficient node is combined with 1 key from parent and sibling of parent

Node G is legal so propagation up the tree stops.



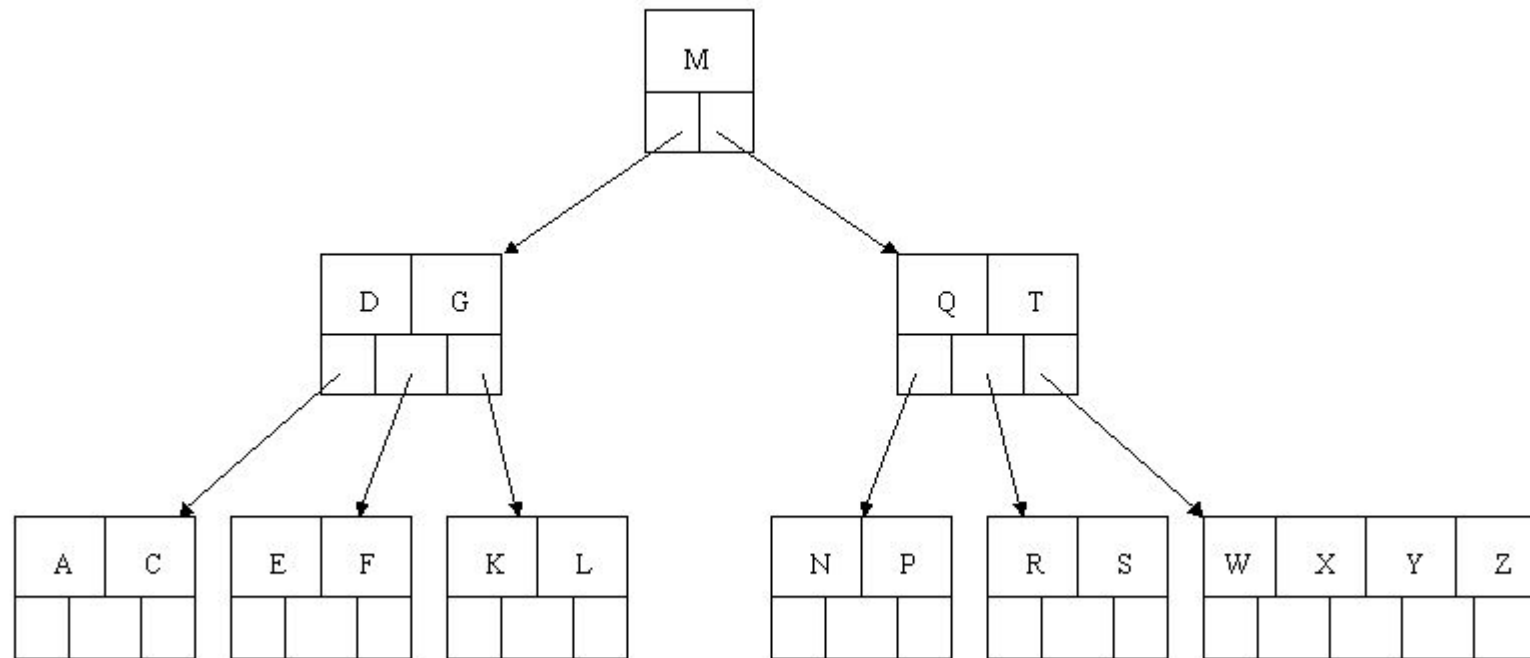
# Inserting S

---



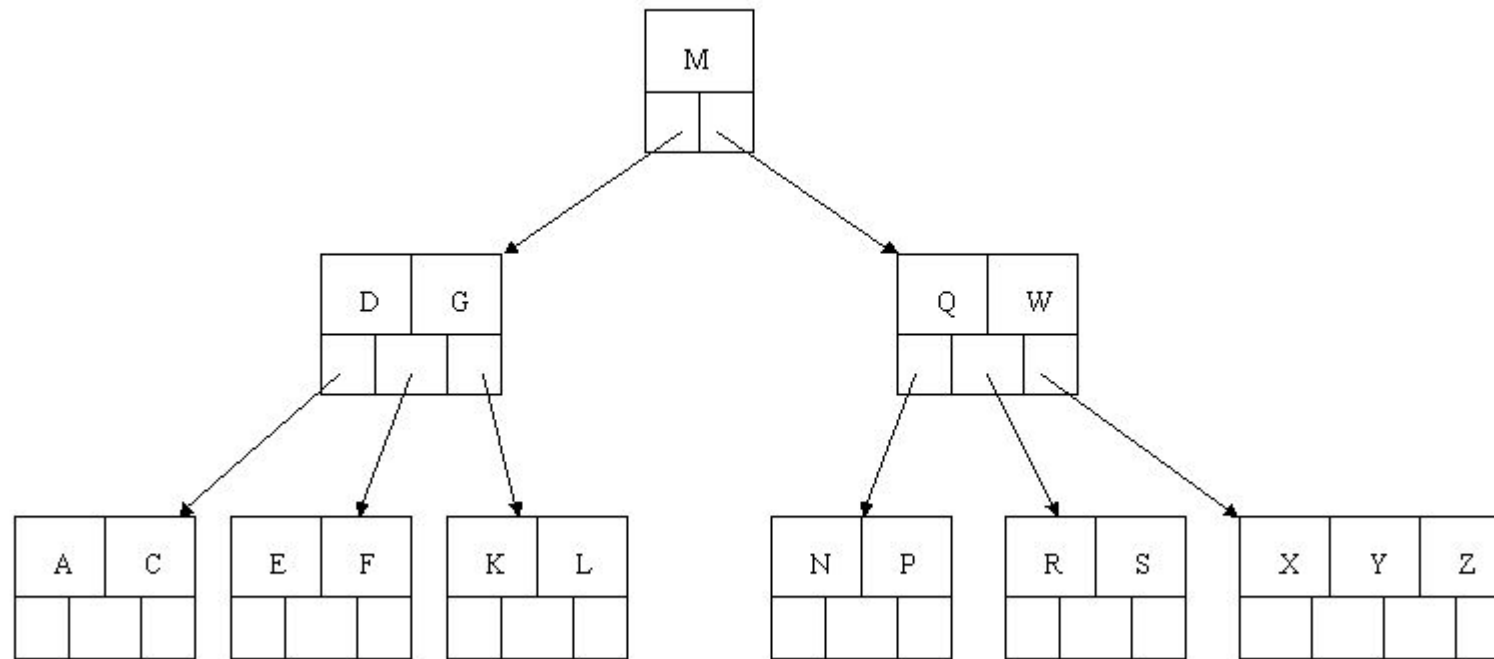
# DELETION (H)

---



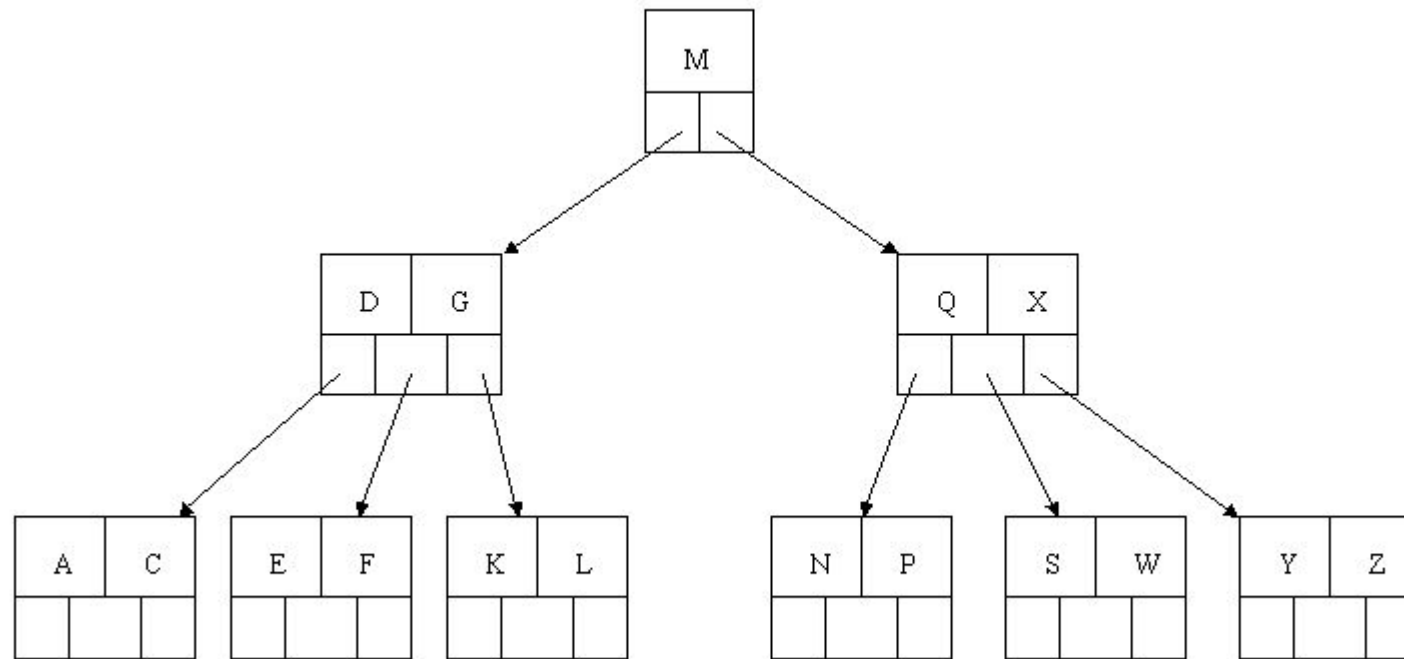
# Delete T

---



# Delete R

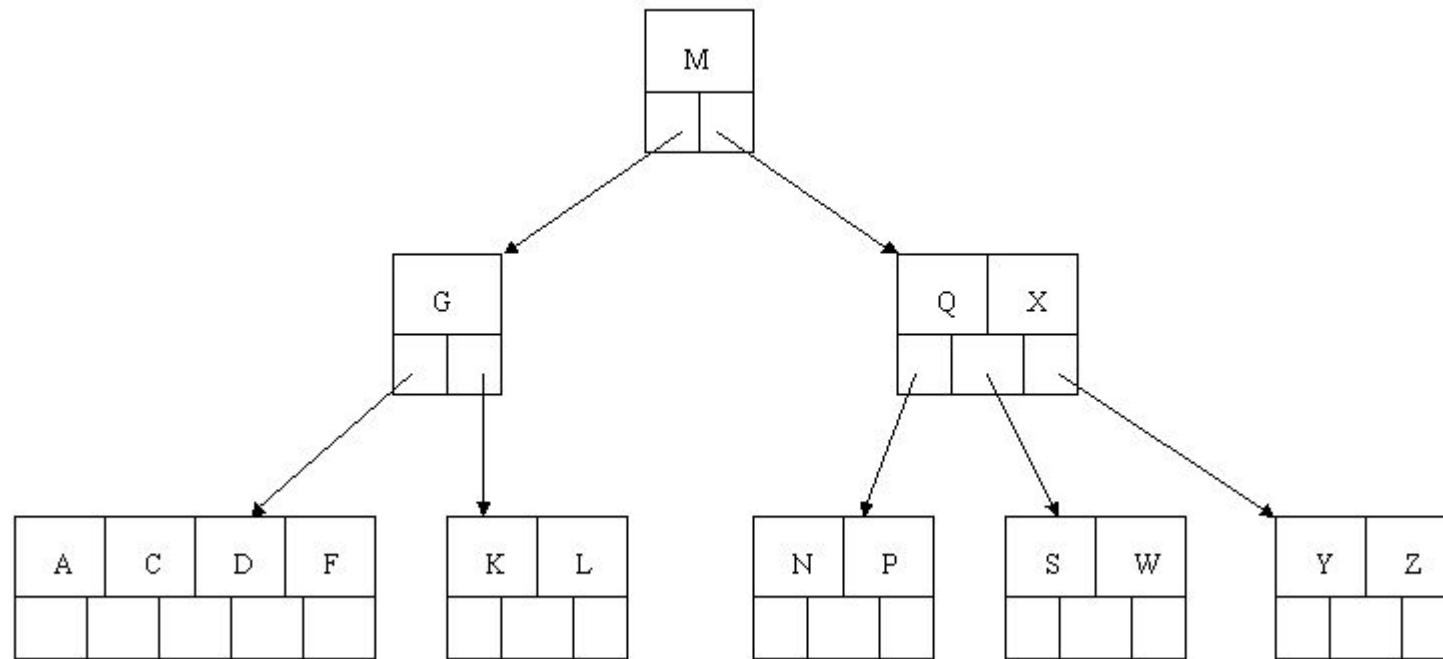
---





# Delete E

---



# Splay Tree

---

# Introduction

---

Splay tree is another variant of binary search tree. In a splay tree, the recently accessed element is placed at the root of the tree. A splay tree is defined as follows..

Splay Tree is a self - adjusted Binary Search Tree in which every operation on an element rearrange the tree so that the element is placed at the root position of the tree.

In a splay tree, every operation is performed at root of the tree. All the operations on a splay tree are involved with a common operation called "Splaying".

Splaying an element is the process of bringing it to the root position by performing suitable rotation operations.

In a splay tree, splaying an element rearrange all the elements in the tree so that splayed element is placed at root of the tree.

With the help of splaying an element we can bring most frequently used element closer to the root of the tree so that any operation on those element performed quickly. That means the splaying operation automatically brings more frequently used elements closer to the root of the tree.

# Introduction

---

Every operation on a splay tree performs the splaying operation.

For example, the insertion operation first inserts the new element as it inserted into the binary search tree, after insertion the newly inserted element is splayed so that it is placed at root of the tree.

The search operation in a splay tree is search the element using binary search process then splay the searched element so that it placed at the root of the tree.

# Rotations in Splay Tree

---

- 1. Zig Rotation
- 2. Zag Rotation
- 3. Zig - Zig Rotation
- 4. Zag - Zag Rotation
- 5. Zig - Zag Rotation
- 6. Zag - Zig Rotation

# Zig Rotation

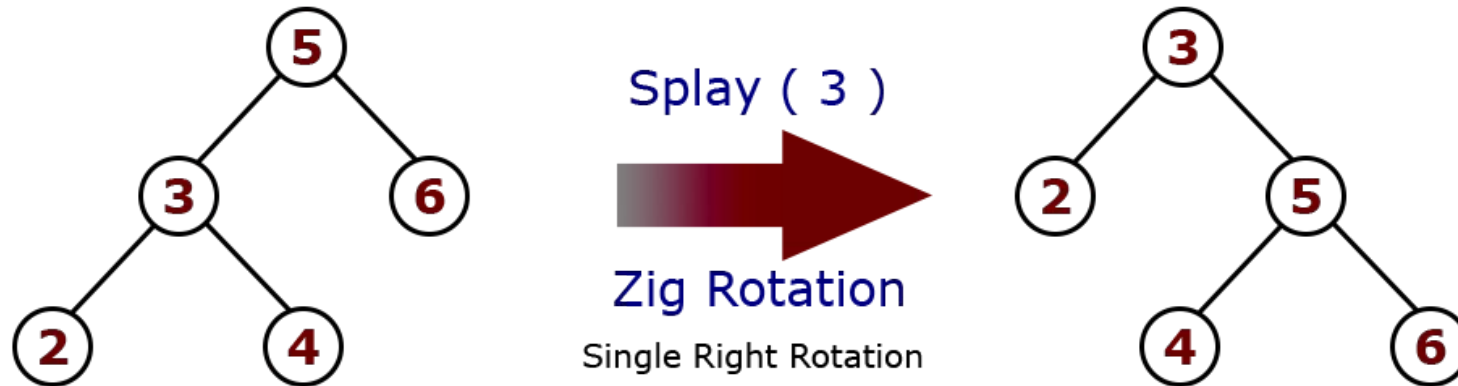
---

The Zig Rotation in a splay tree is similar to the single right rotation in AVL Tree rotations.

In zig rotation every node moves one position to the right from its current position.

# Zig Rotation

---



# Zag Rotation

---

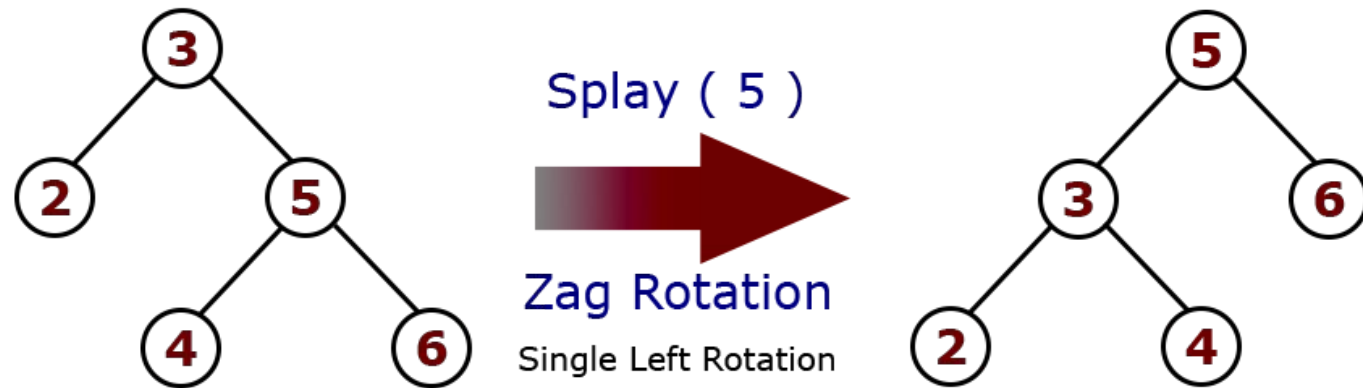
The Zag Rotation in a splay tree is similar to the single left rotation in AVL Tree rotations.

In zag rotation every node moves one position to the left from its current position



# Zag Rotation

---



# Zig-Zig Rotation

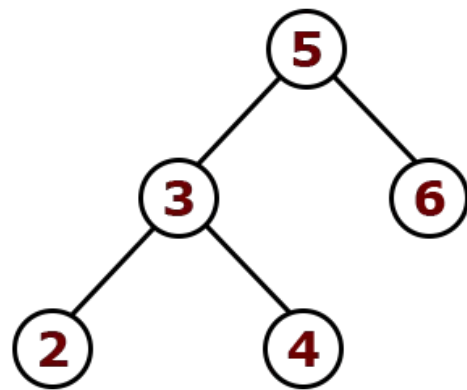
---


The **Zig-Zig Rotation** in a splay tree is a double zig rotation.

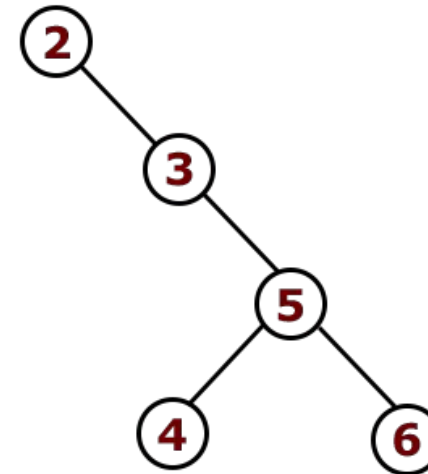
In zig-zig rotation every node moves two position to the right from its current position.

# Zig-Zig Rotation

---

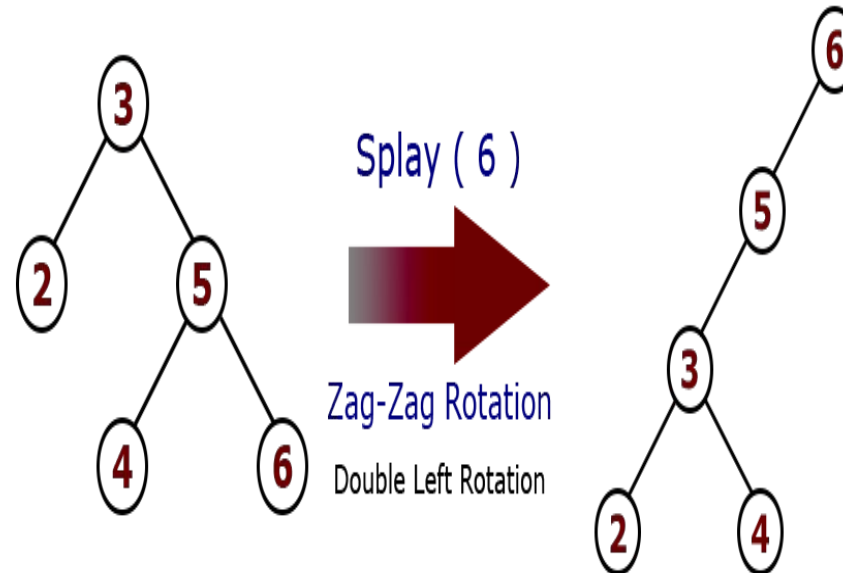


Splay ( 2 )  
  
Zig-Zig Rotation  
Double Right Rotation



# Zag-Zag Rotation

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



# Zig-Zag Rotation

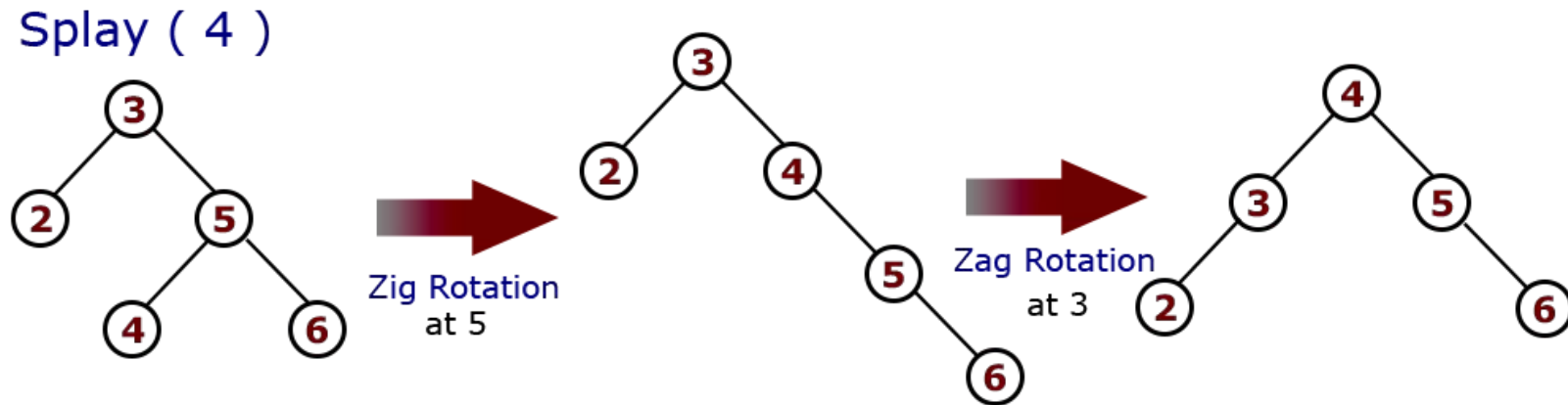
---

The **Zig-Zag Rotation** in a splay tree is a sequence of zig rotation followed by zag rotation.

In zig-zag rotation every node moves one position to the right followed by one position to the left from its current position.

# Zig-Zag Rotation

---



# Zag-Zig Rotation

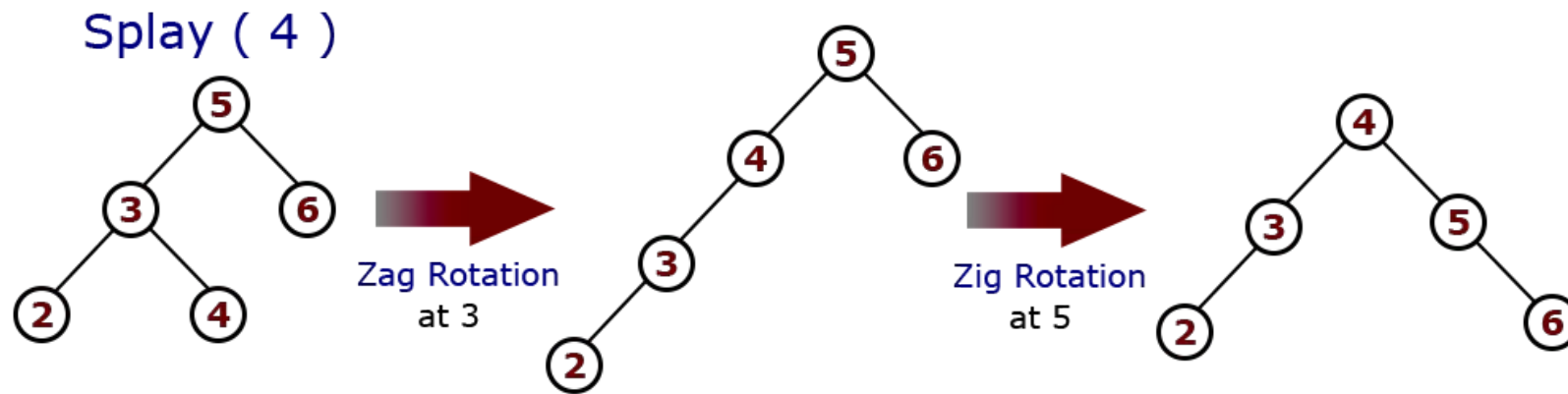
---

The **Zag-Zig Rotation** in a splay tree is a sequence of zag rotation followed by zig rotation.

In zag-zig rotation every node moves one position to the left followed by one position to the right from its current position.

# Zag-Zig Rotation

---





---

**Every Splay tree must be a binary search tree but it is need not to be balanced tree.**

# Insertion Operation in Splay Tree

---

The insertion operation in Splay tree is performed using following steps...

**Step 1:** Check whether tree is Empty.

**Step 2:** If tree is Empty then insert the **newNode** as Root node and exit from the operation.

**step 3:** If tree is not Empty then insert the newNode as a leaf node using Binary Search tree insertion logic.

**Step 4:** After insertion, **Splay** the **newNode**

# Deletion Operation in Splay Tree

---

In a Splay Tree, the deletion operation is similar to deletion operation in Binary Search Tree. But before deleting the element first we need to **splay** that node then delete it from the root position then join the remaining tree.

# Red-Black Trees

---

Red - Black Tree is another variant of 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.

In a Red Black Tree the color of a node is decided based on the Red Black Tree properties. 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 colored BLACK.

**Property #3:** The children of Red colored node must colored BLACK. (There should not be two consecutive RED nodes).

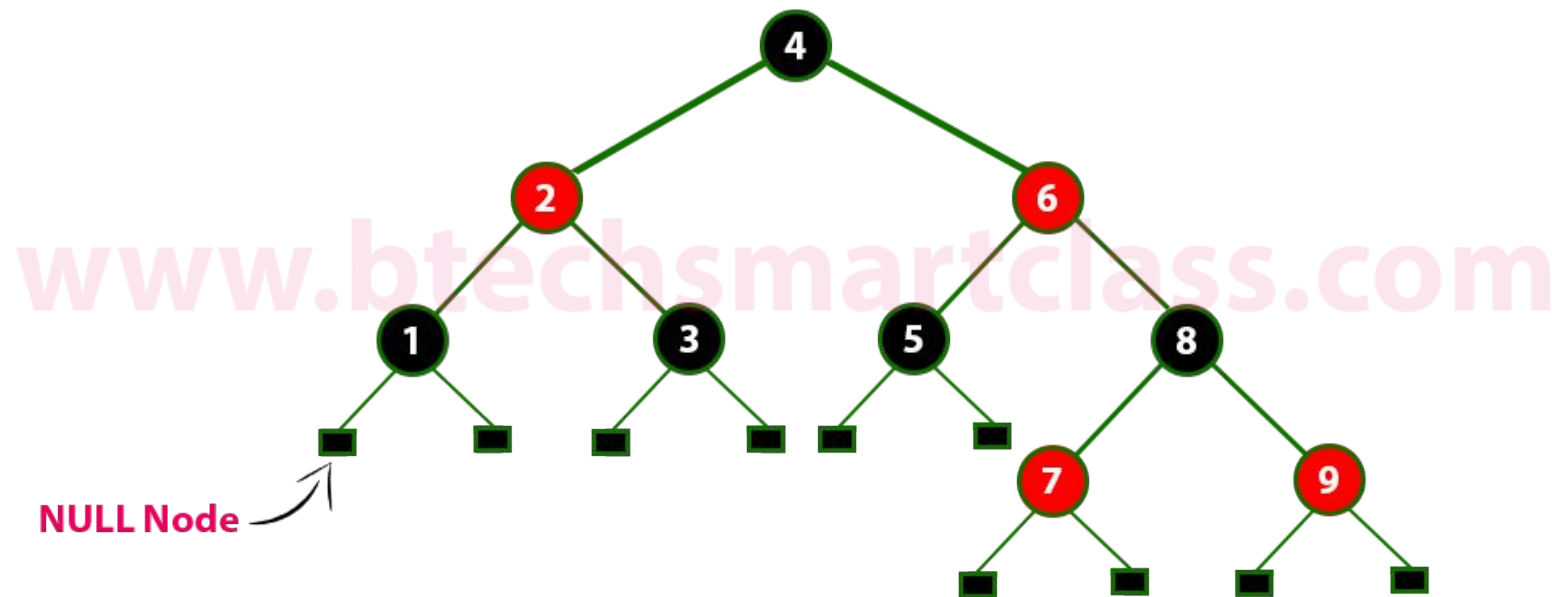
**Property #4:** In all the paths of the tree there must be same number of BLACK colored nodes.

**Property #5:** Every new node must inserted with RED color.

**Property #6:** Every leaf (e.i. NULL node) must colored BLACK.

# Example

---



---

Every Red Black Tree is a binary search tree but all the Binary Search Trees need not to be Red Black trees.

# Insertion into RED BLACK Tree:

---

In a Red Black Tree, every new node must be inserted with color RED.

The insertion operation in Red Black Tree is similar to insertion operation in Binary Search Tree.

But it is inserted with a color property.

After every insertion operation, we need to check all the properties of Red Black Tree.



# Insertion into RED BLACK Tree:

---

If all the properties are satisfied then we go to next operation otherwise we need to perform following operation to make it Red Black Tree.

1. Recolor
3. Rotation followed by Recolor

# Insertion into RED BLACK Tree:

---

**Step 1:** Check whether tree is Empty.

**Step 2:** If tree is Empty then insert the **newNode** as Root node with color **Black** and exit from the operation.

**step 3:** If tree is not Empty then insert the newNode as a leaf node with Red color.

**Step 4:** If the parent of newNode is Black then exit from the operation.

**Step 5:** If the parent of newNode is Red then check the color of parent node's sibling of newNode.

**Step 6:** If it is Black or NULL node then make a suitable Rotation and Recolor it.

**Step 7:** If it is Red colored node then perform Recolor and Recheck it. Repeat the same until tree becomes Red Black Tree.

# Example

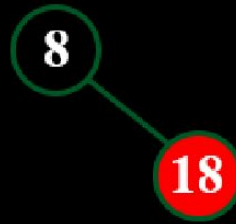
---

Create a RED BLACK Tree by inserting following sequence of number  
8, 18, 5, 15, 17, 25, 40 & 80.

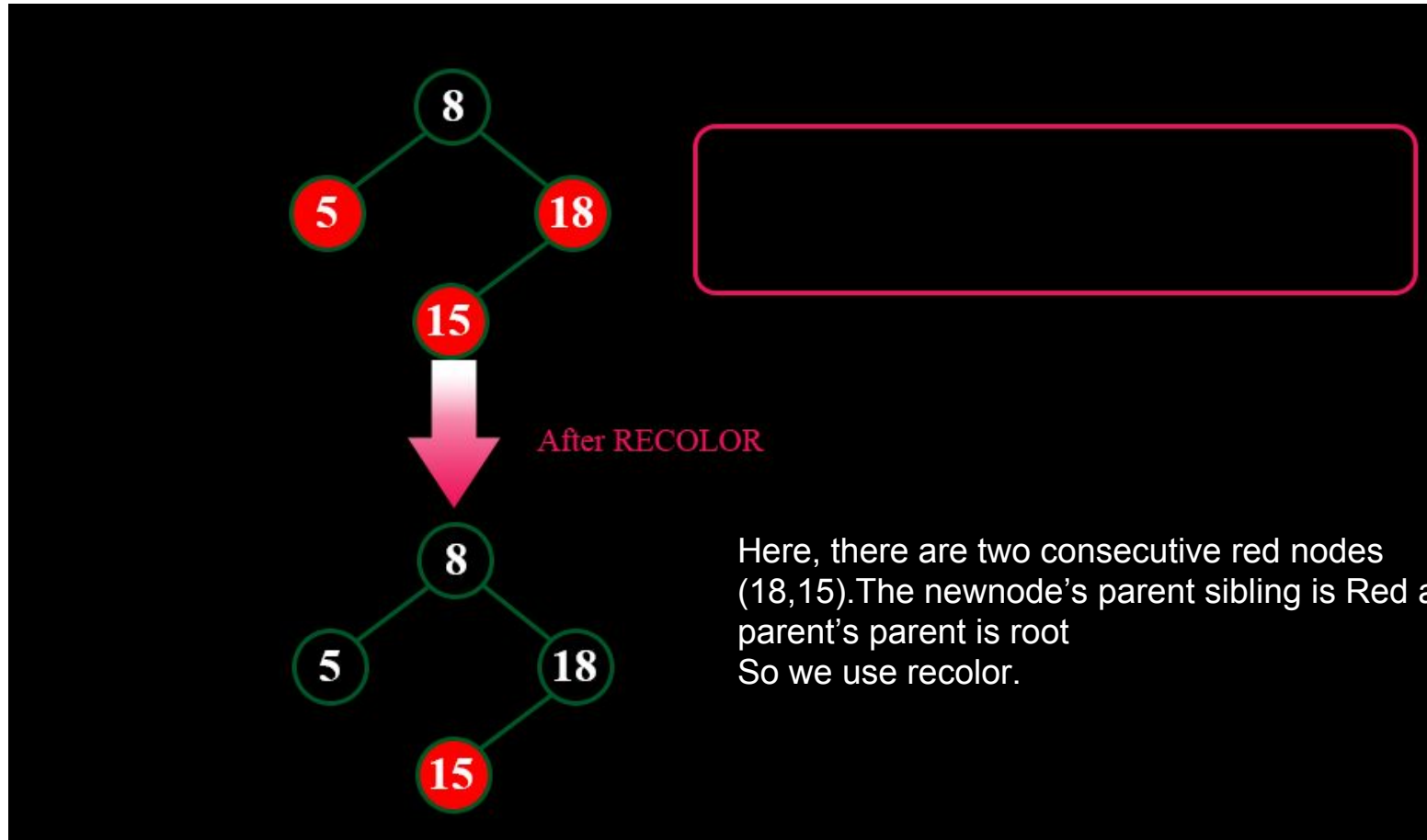
insert ( 8 )



insert ( 18 )

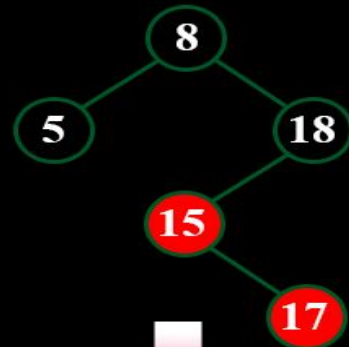


# Insert 15

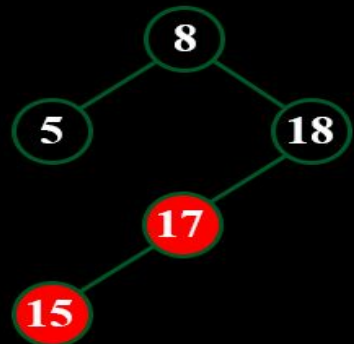


# Example-Insert 17

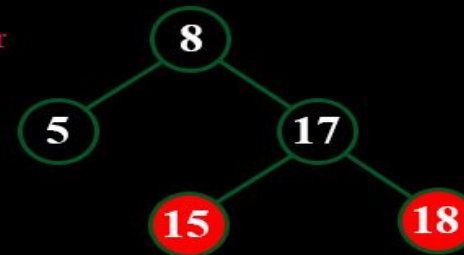
insert ( 17 )



After Left Rotation



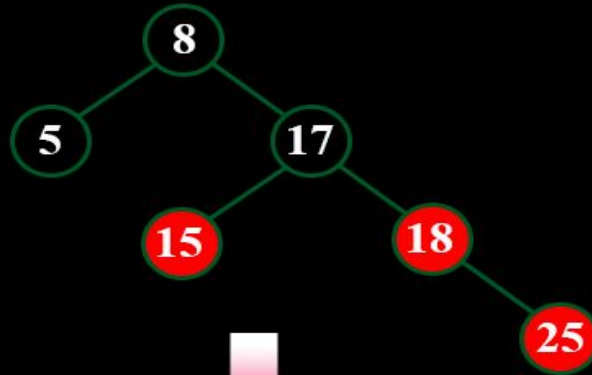
After Right Rotation & Recolor



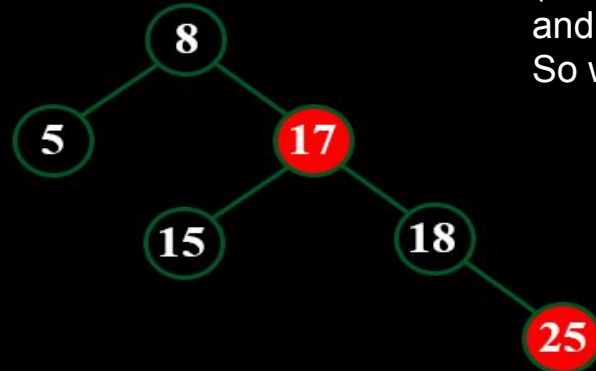
Here, there are two consecutive red nodes (15,17). The newnode's parent sibling is NULL. So we use rotation. Use LR rotation

# Insert 25

insert ( 25 )

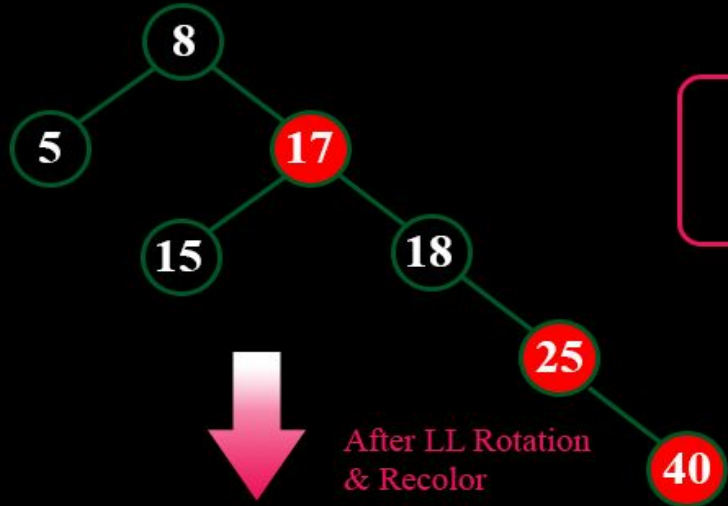


After Recolor

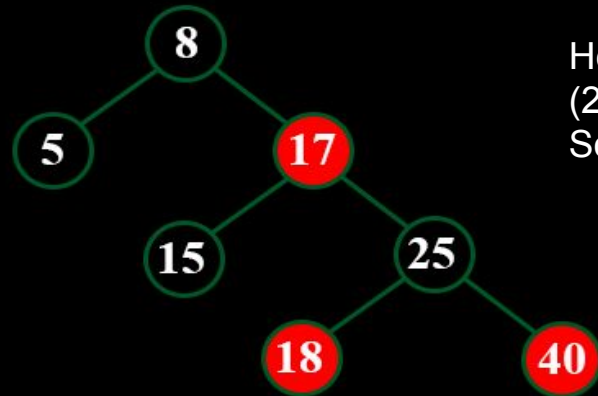


Here, there are two consecutive red nodes (18,25). The newnode's parent sibling color is red and parent's parent is not root node. So we use recolor and recheck.

# Insert 40

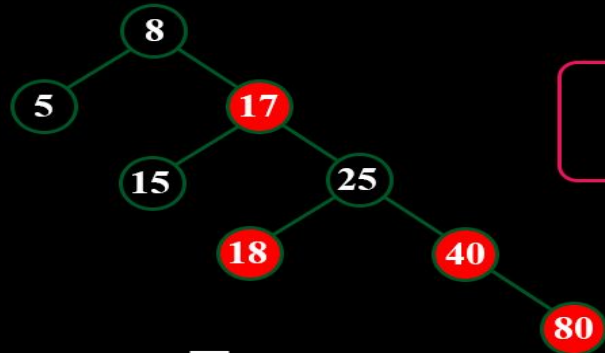


After LL Rotation  
& Recolor

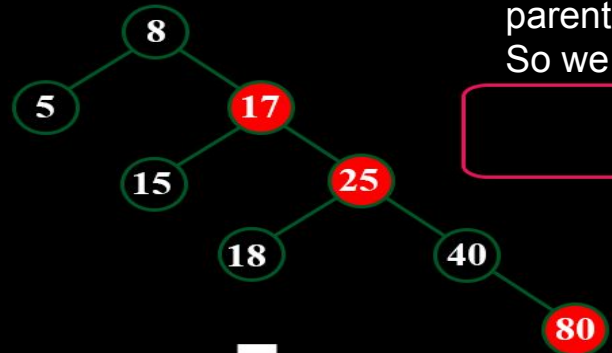


Here, there are two consecutive red nodes  
(25,40). The newnode's parent sibling is NULL  
So we use rotation. Use LL rotation

# Insert 80



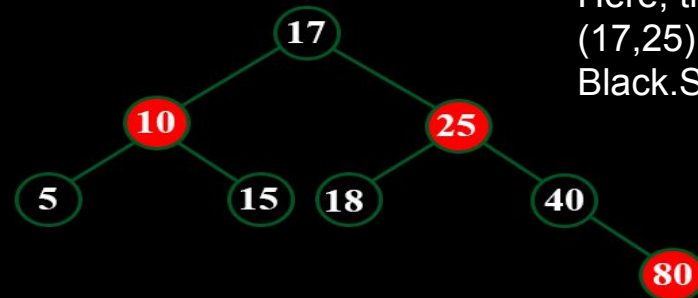
After Recolor



Here, there are two consecutive red nodes (40,80). The newnode's parent sibling is Red. parent's parent is not root. So we use recolor.



After Left Rotation & Recolor



Here, there are two consecutive red nodes (17,25). The newnode's parent sibling color is Black. So we use left rotation.



# FAQ

---

- Construct the AVL tree for
  - March, May, Nov, Aug, April, Jan, Dec, July, Feb, June, Oct & Sept.
  - Madhushri, Kanchan, Anagha, Rohit, Neeta, Snehal, Pratibha, Lalit, Mrudula, Kavita, Apurva & Harsha.
  - 30, 31, 32, 23, 22, 28, 24, 29, 26, 27, 34, 36.
  - MOV, MUL, POP, DAA, JNB, INC, JNE, JMP, LOOPZ, PUSH, RET.

# References

---

1. E. Horowitz, S. Sahani, S. Anderson-Freed, "Fundamentals of Data Structures in C", Universities Press, 2008
2. Treamblay, Sorenson, "An introduction to data structures with applications", Tata McGraw Hill, Second Edition
3. Aaron Tanenbaum, "Data Structures using C", Pearson Education
4. R. Gilberg, B. Forouzan, "Data Structures: A pseudo code approach with C", Cenage Learning, ISBN 9788131503140