Dr. Vishwanath Karad MIT WORLD PEACE UNIVERSITY | PUNE

TECHNOLOGY, RESEARCH, SOCIAL INNOVATION & PARTNERSHIPS

CET2001B Advanced Data Structure

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Heap

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

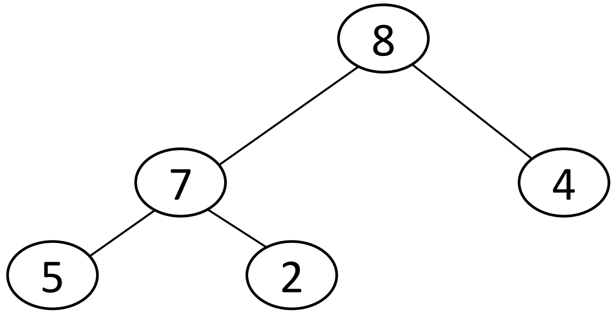
The Heap Data Structure

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)

Parent(x) ≥ x



Heap

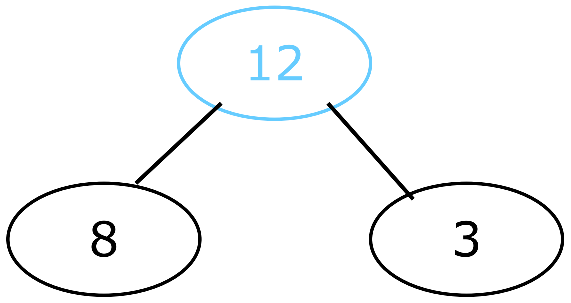
Example of Max heap

"The root is the maximum element of the heap!"

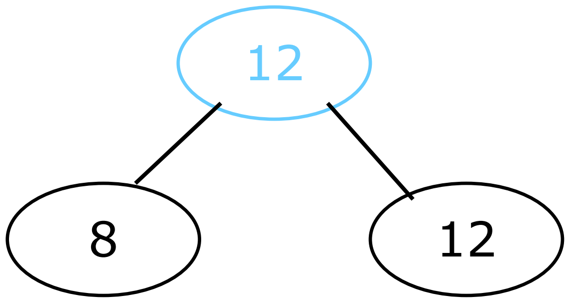
A heap is a binary tree that is filled in order

The heap property **(for Max heap)**

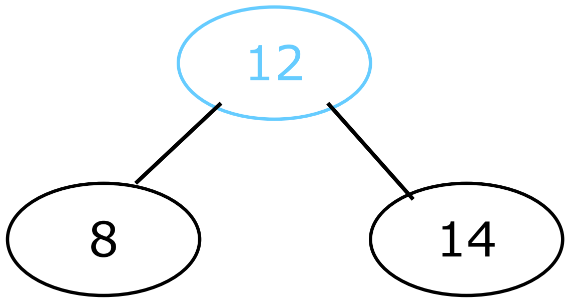
A node has the heap property if the value in the node is as large as or larger than the values in its children



Blue node has heap property



Blue node has heap property



Blue node does not have heap property

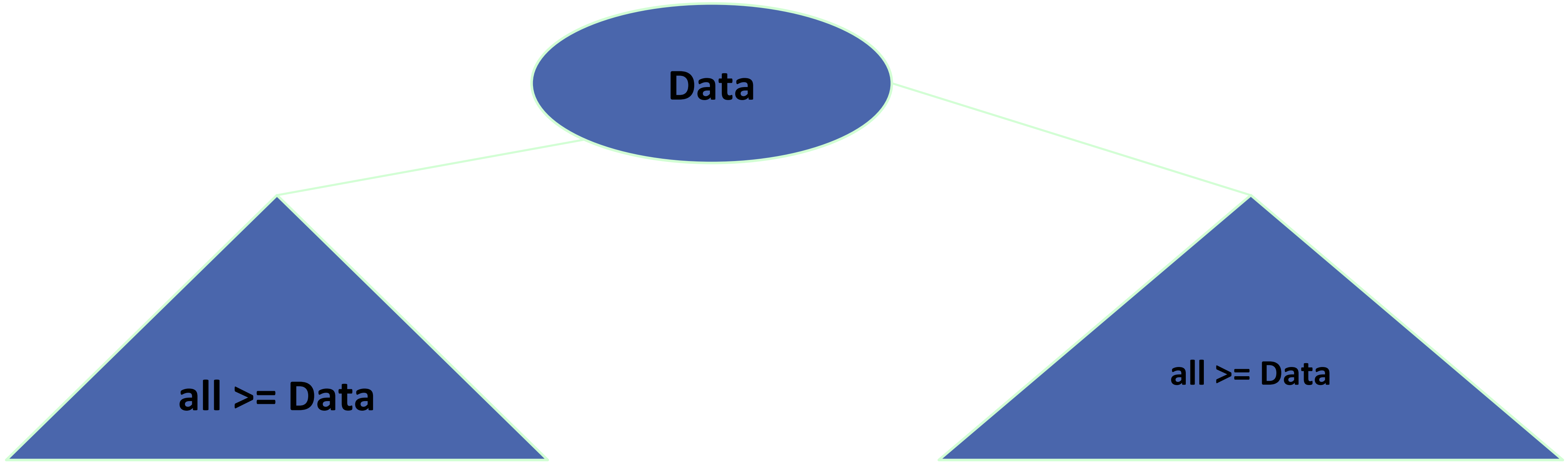
All leaf nodes automatically have the heap property

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

Types of Heap

- **Min-heap**
- **Max-heap**

Min Heap

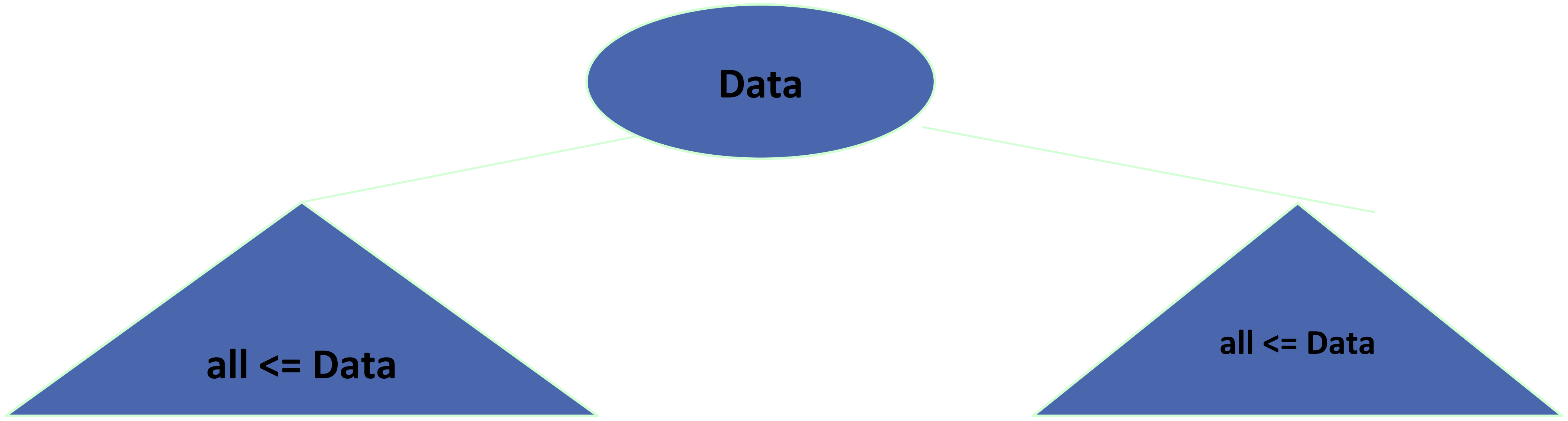


Min-Heap

- 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

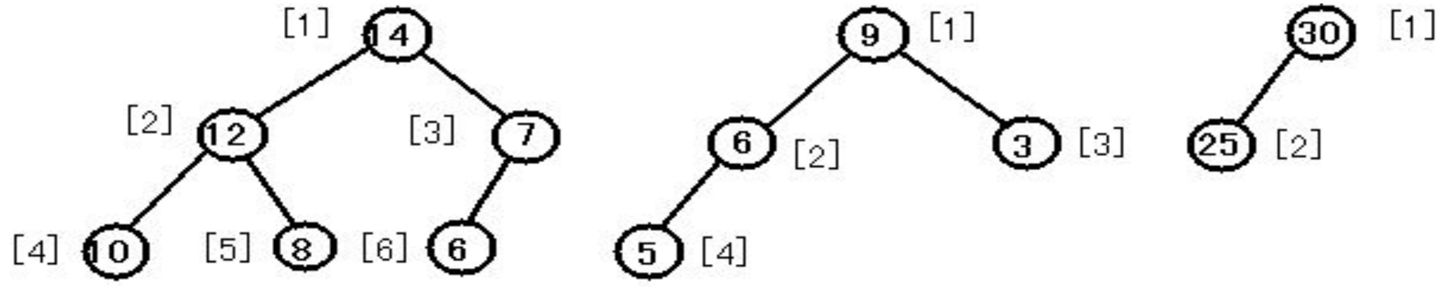


Max-heap

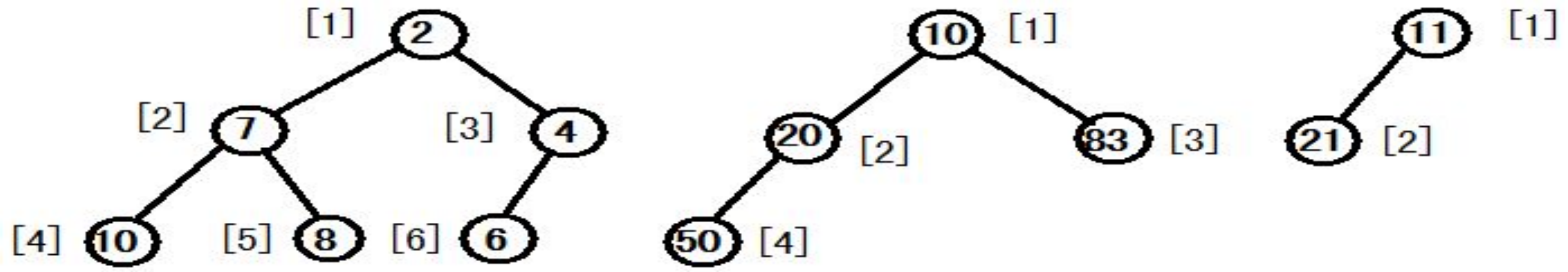
- A max-heap is where the key value of a node is greater than the key values in of its children

Example of Heap

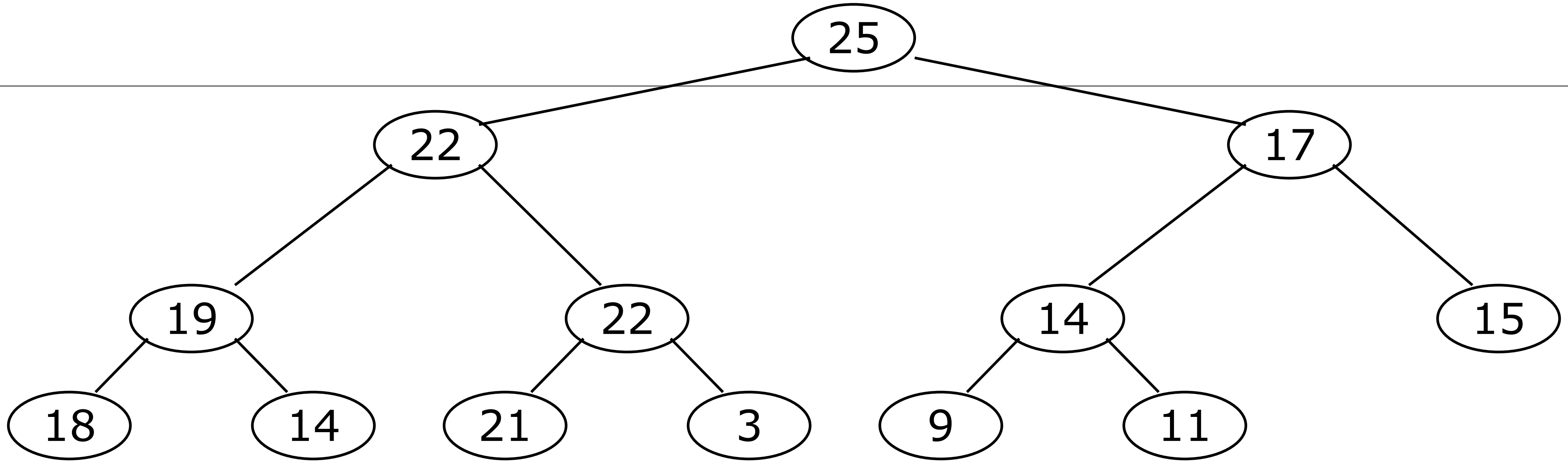
Max-Heap



Min-Heap



Mapping into an array



1	2	3	4	5	6	7	8	9	10	11	12	13
25	22	17	19	22	14	15	18	14	21	3	9	11

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**

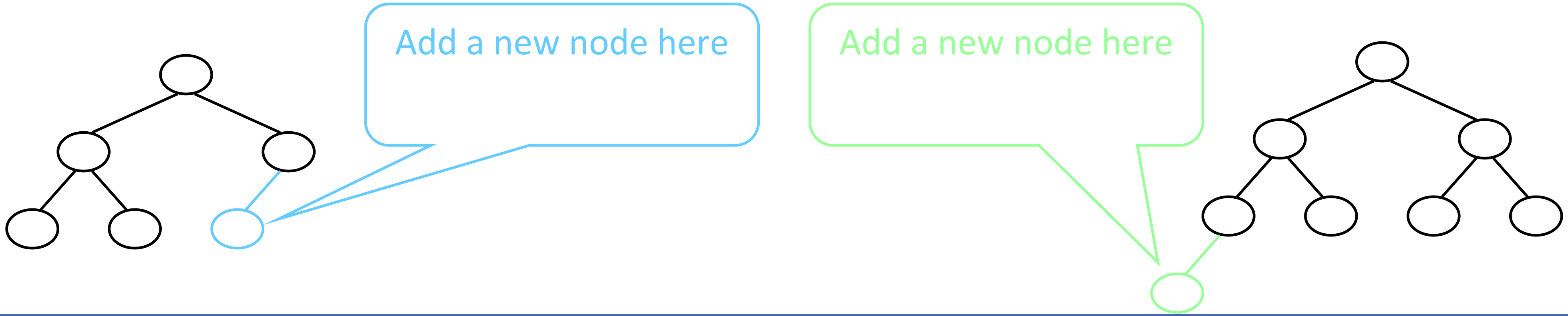
Constructing a (max) heap

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

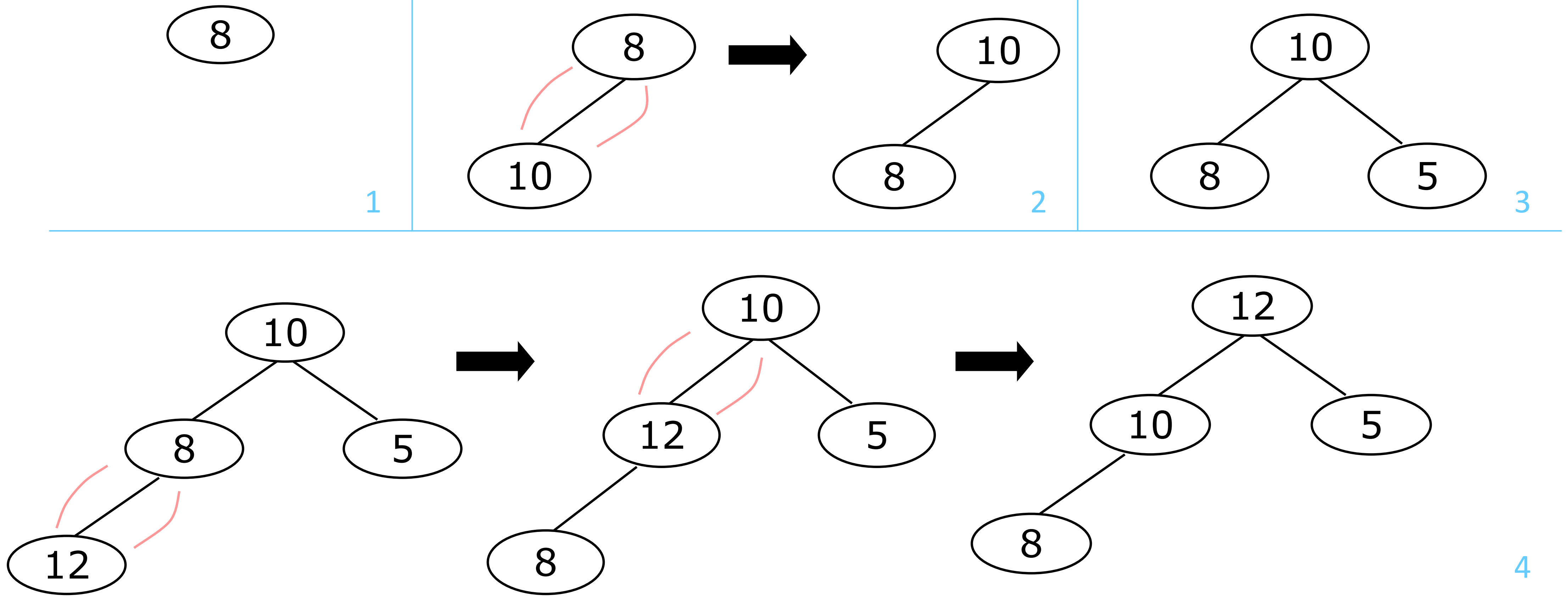
We repeat the shifting up process, moving up in the tree, until either

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

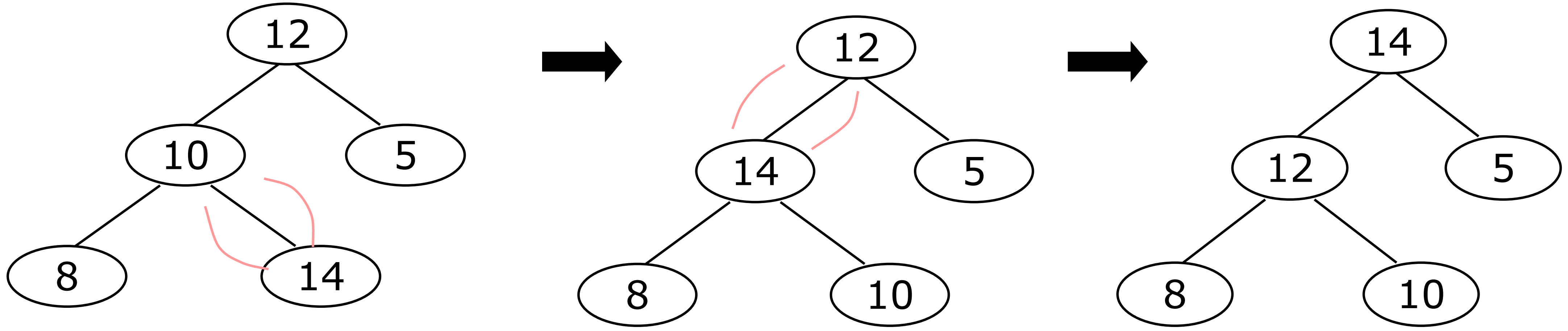
- We reach the root

Constructing a heap

contd...



Other children are not affected



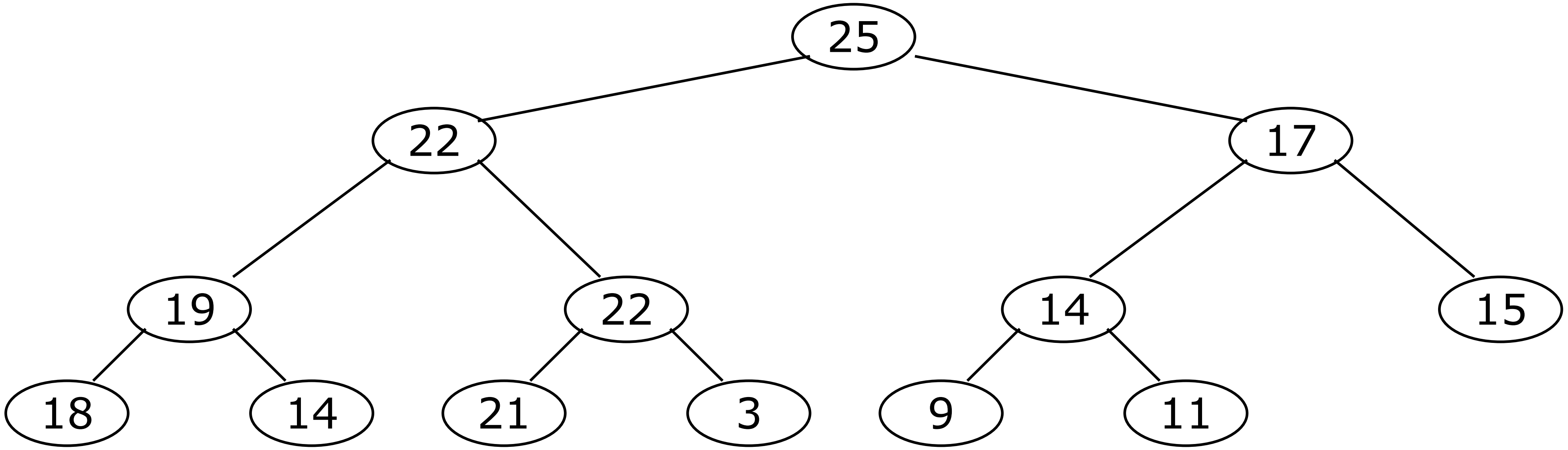
The node containing 8 is not affected because its parent gets larger, not smaller

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

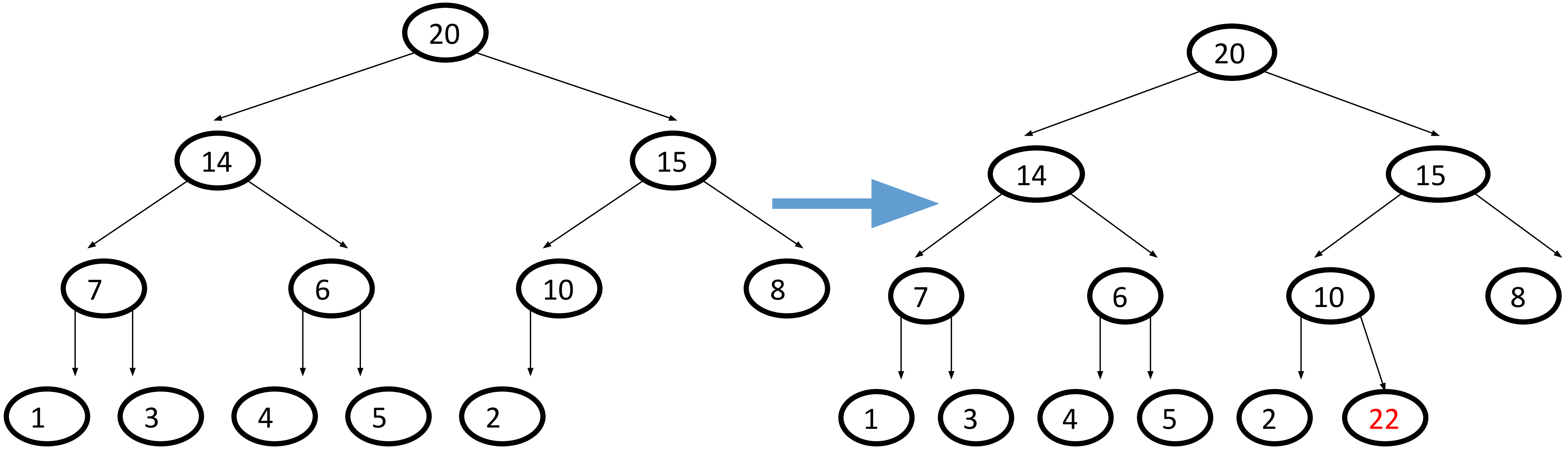


Notice that heapified does *not* mean sorted

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

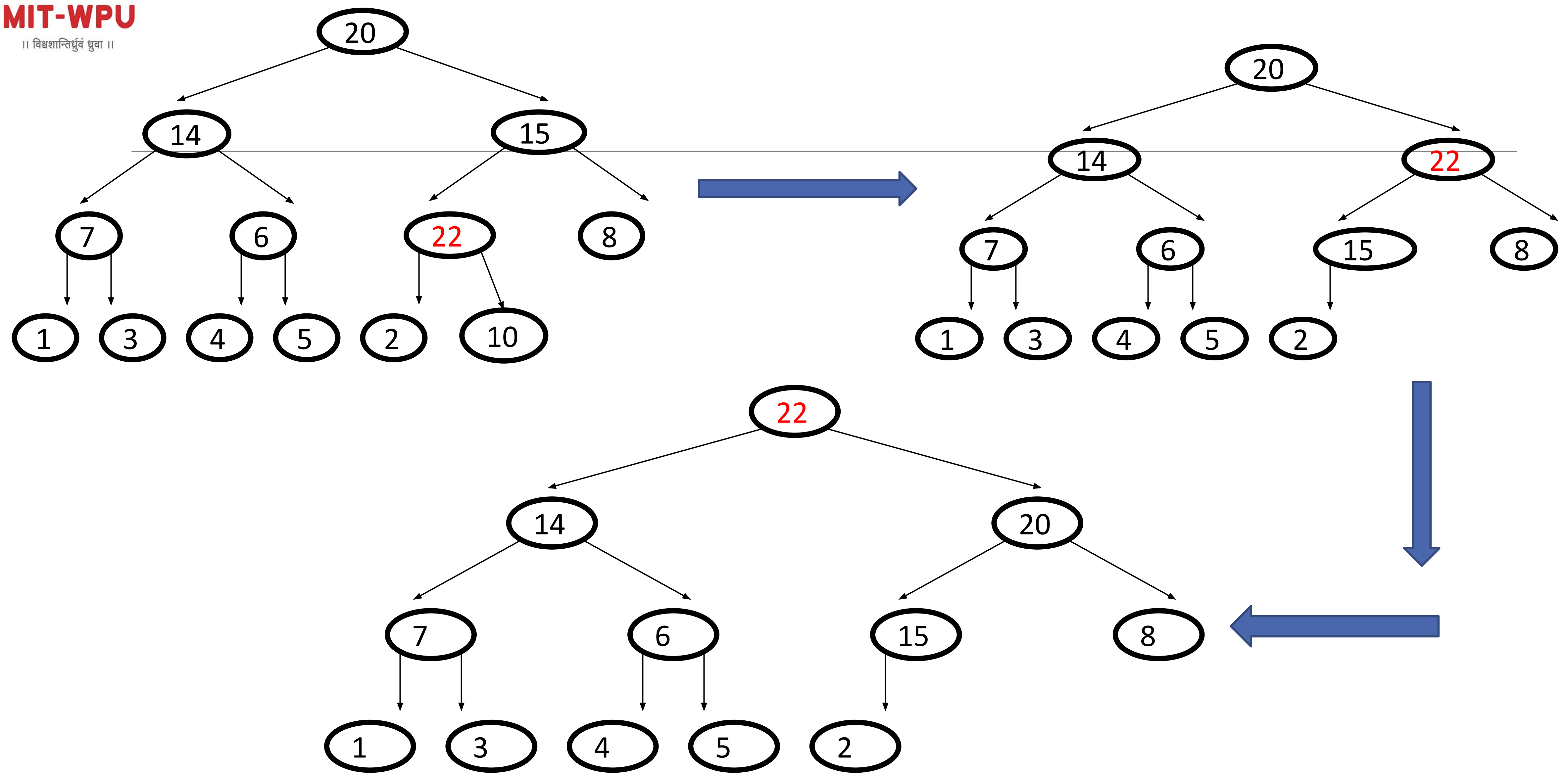
Insert (max heap)

```
insert(22)
```



Insert (max heap)

contd…

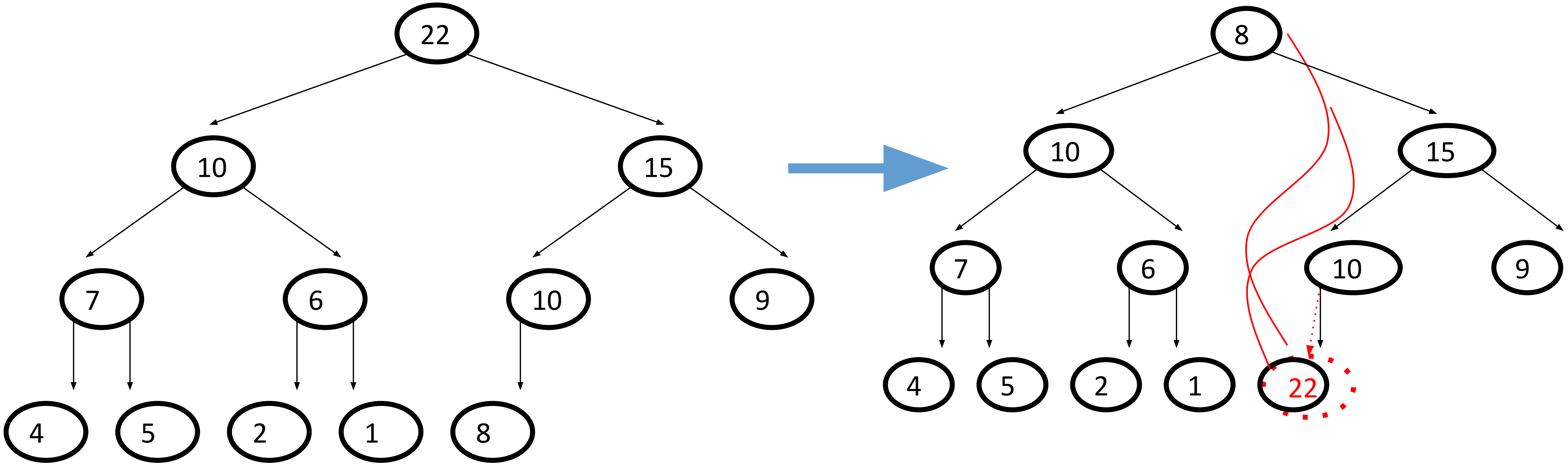


```
Algorithm add()
{      //index of a starting from 1
     j=1 ; a[20];
     Get Choice
    Repeat
    {
     a[j]=elem;
       insert(a,j);
       j++;
     }
    until(choice==‘n’);
}
```

```
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;
}
```

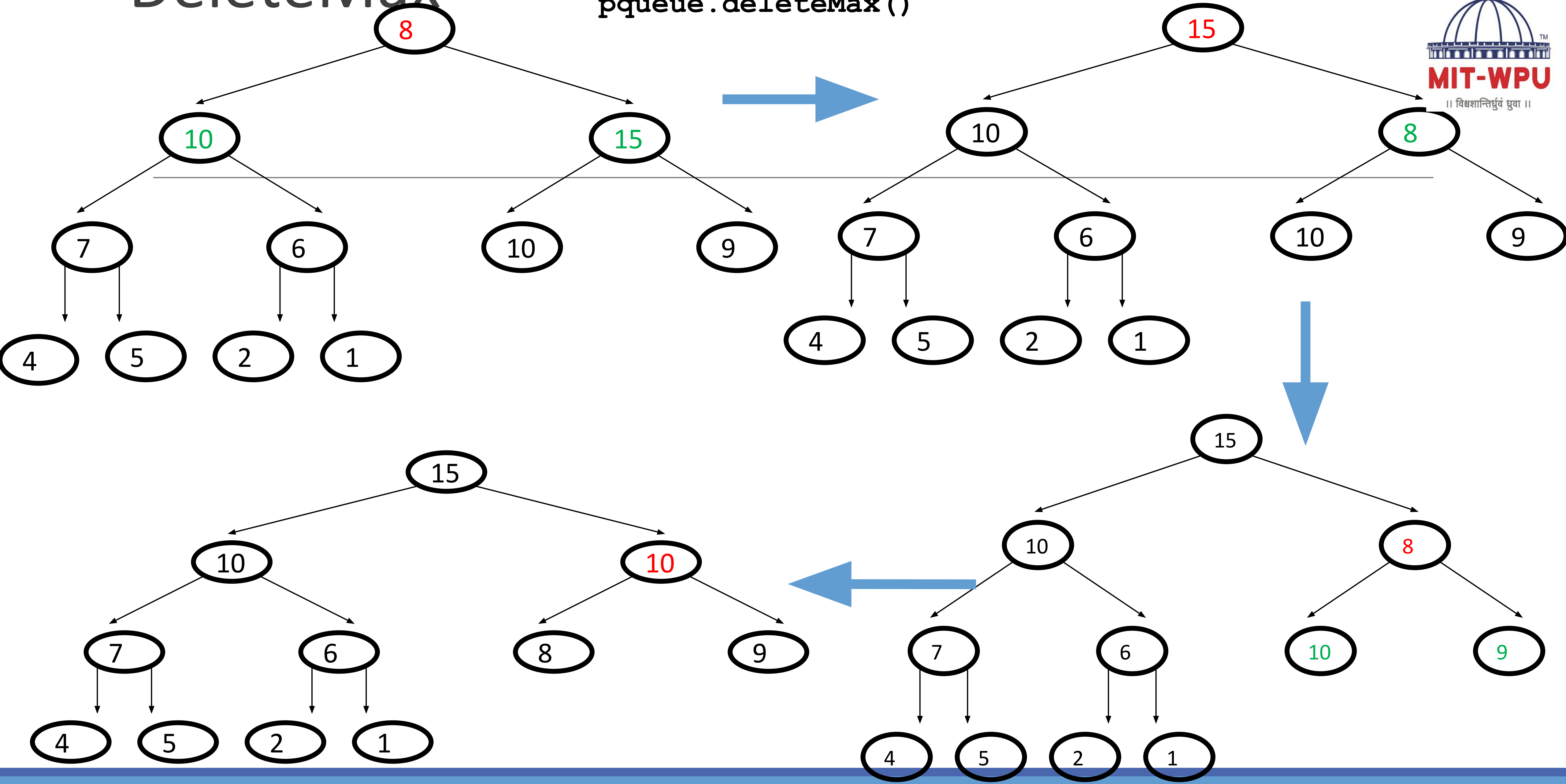
DeleteMax

`pqueue.deleteMax()`



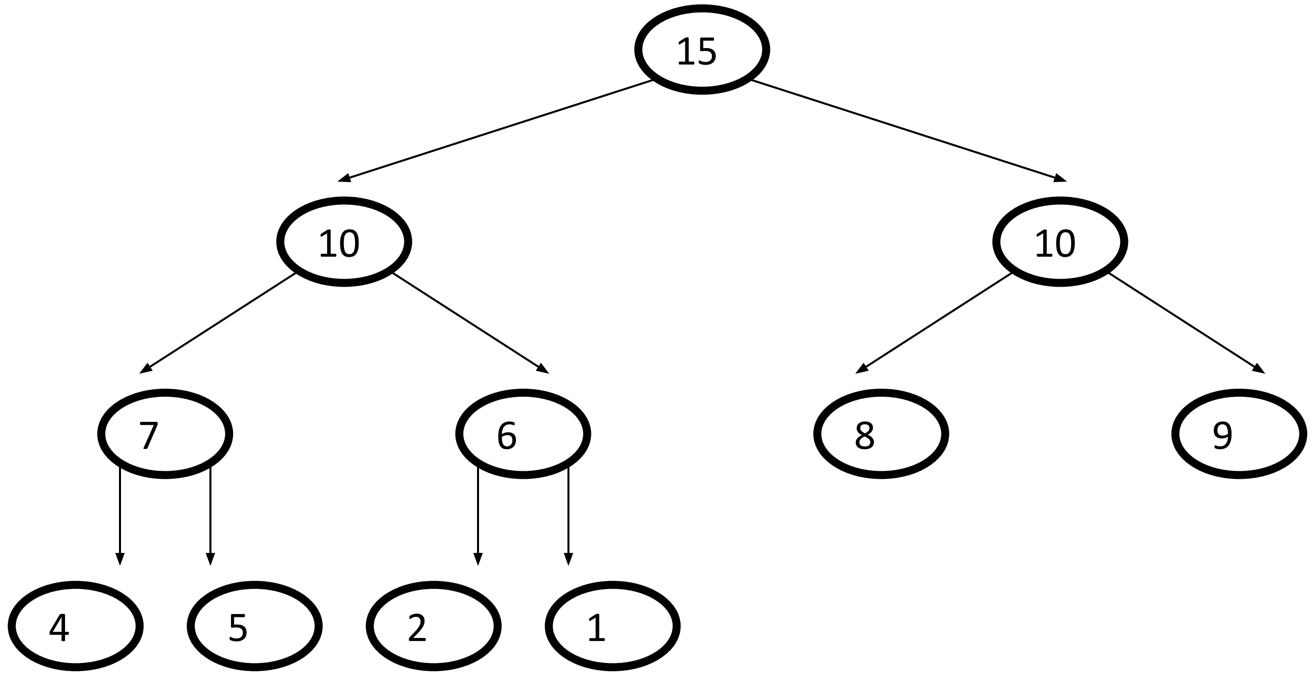
DeleteMax

```
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 chil
          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

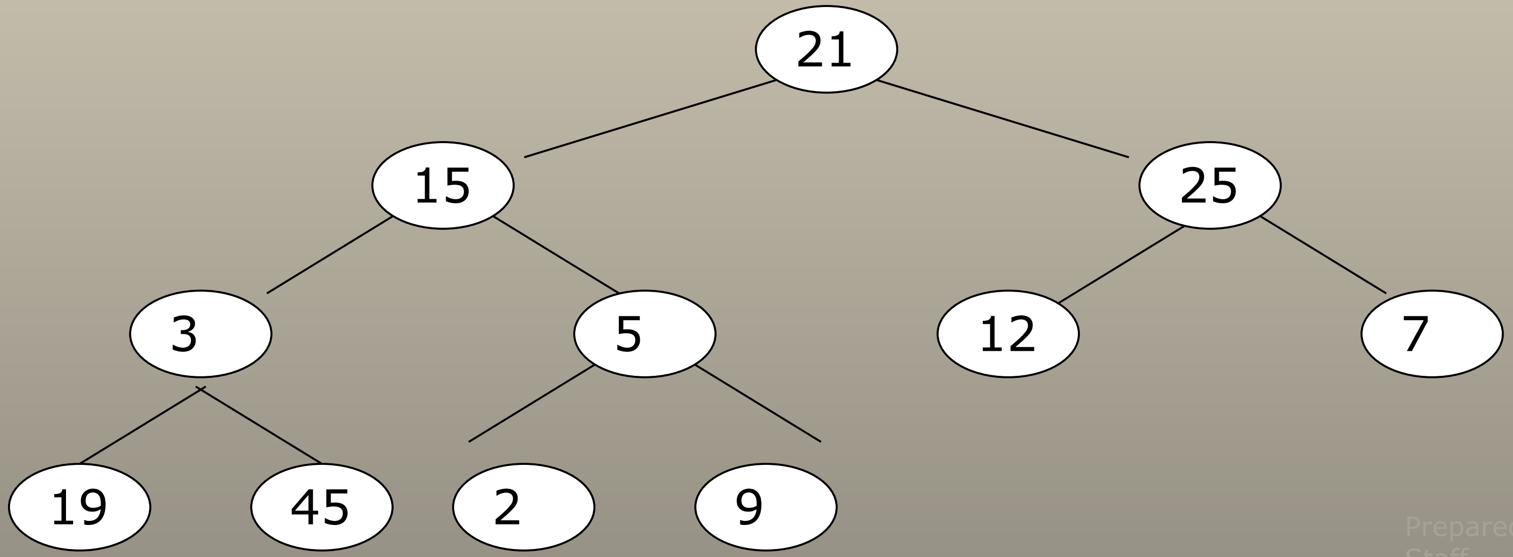
Sample Run

- Start with unordered array of data

Array representation:

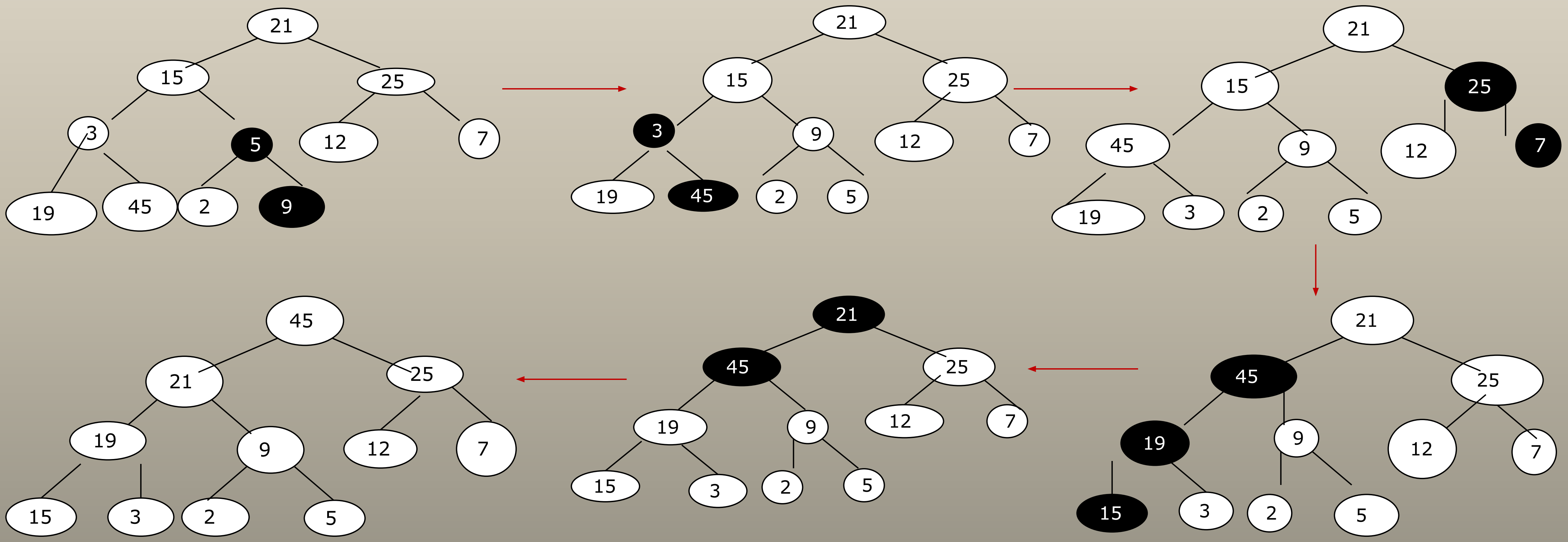
21	15	25	3	5	12	7	19	45	2	9

Binary tree representation:

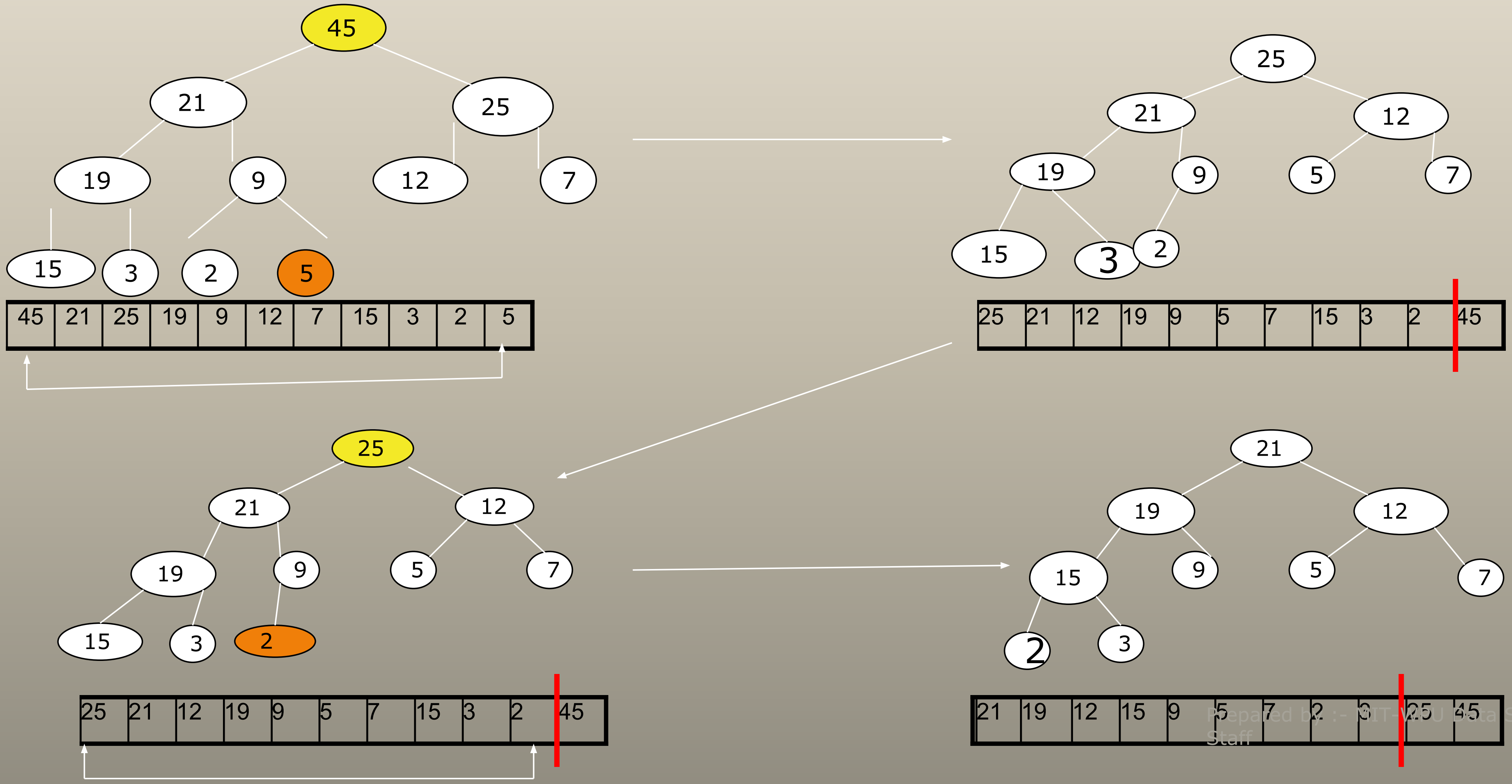


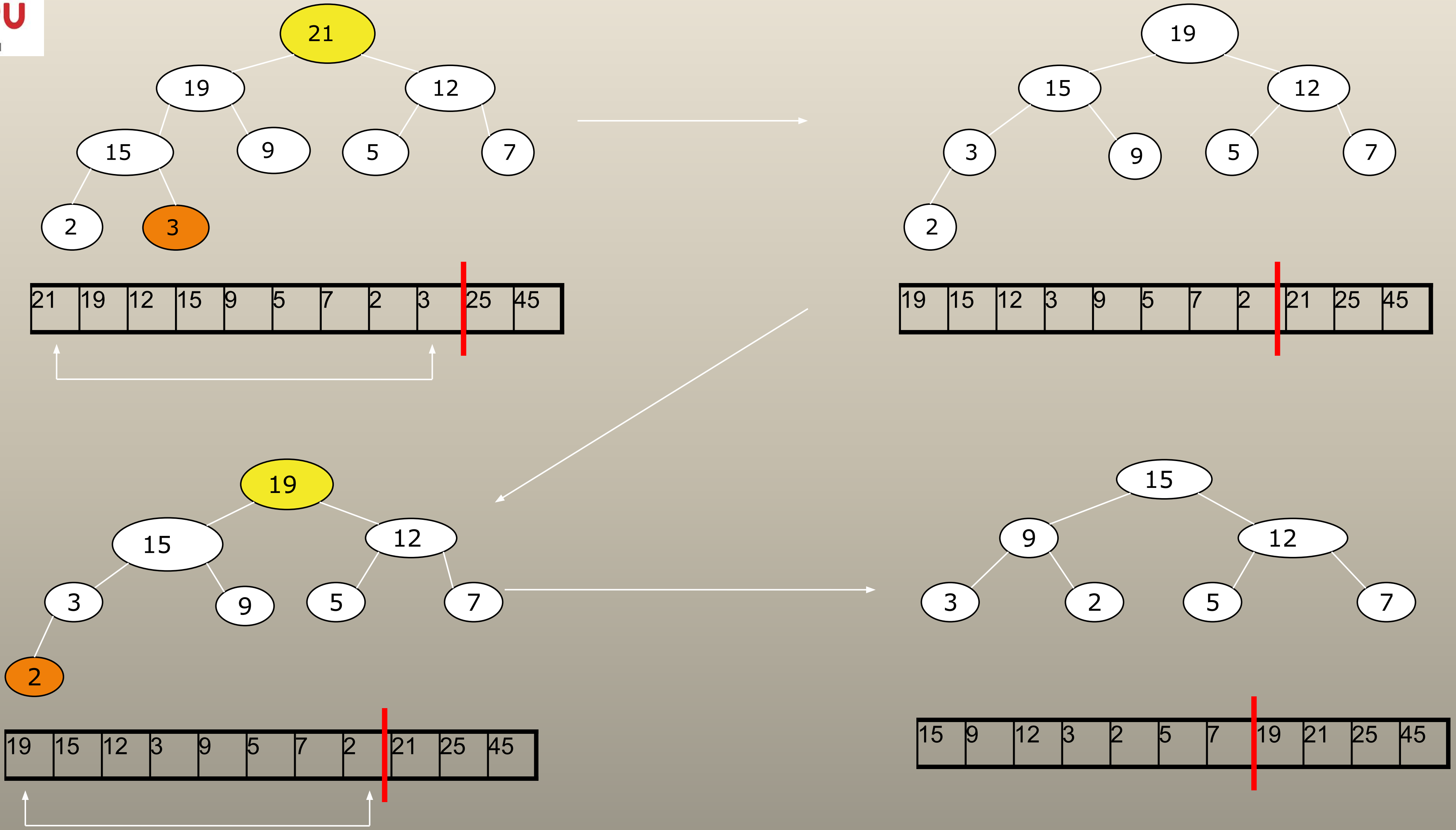
Sample Run

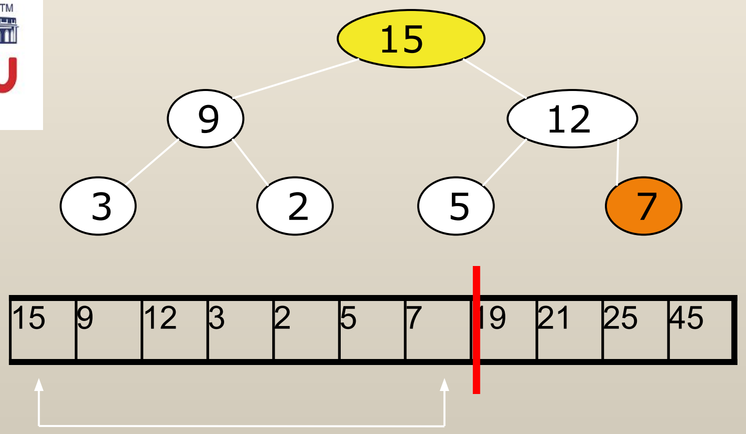
- Heapify the binary tree –(from n/2 to 1)(ReheapDown)

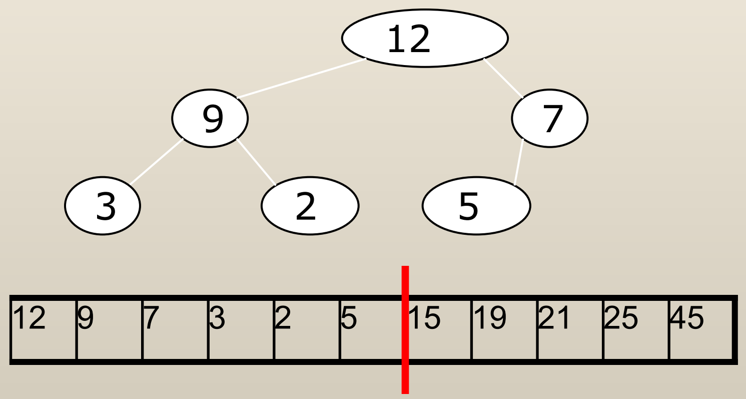


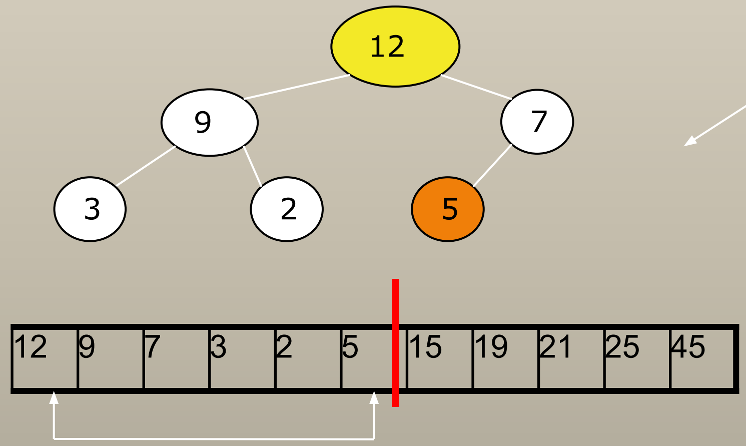
Step 2 – perform n – 1 deleteMax(es), and replace last element in heap with first, then re-heapify. Place deleted element in the last nodes position.

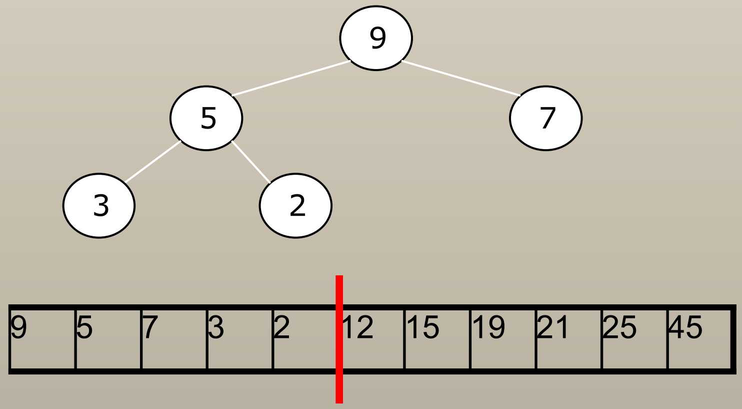



21
19
12
15
9
5
7
2
3
21 | 19 | 12 | 15 | 9 | 5 | 7 | 2 | 3 | 25 | 45
19
15
12
3
9
5
7
2
19 | 15 | 12 | 3 | 9 | 5 | 7 | 2 | 21 | 25 | 45
19
15
12
3
9
5
7
2
19 | 15 | 12 | 3 | 9 | 5 | 7 | 2 | 21 | 25 | 45
15
9
12
3
2
5
7
15 | 9 | 12 | 3 | 2 | 5 | 7 | 19 | 21 | 25 | 45



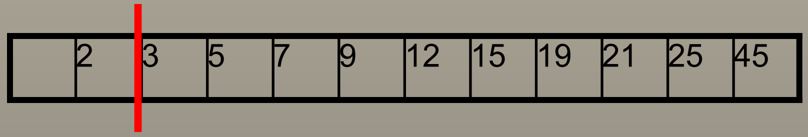






This continues till only one is left

...and finally

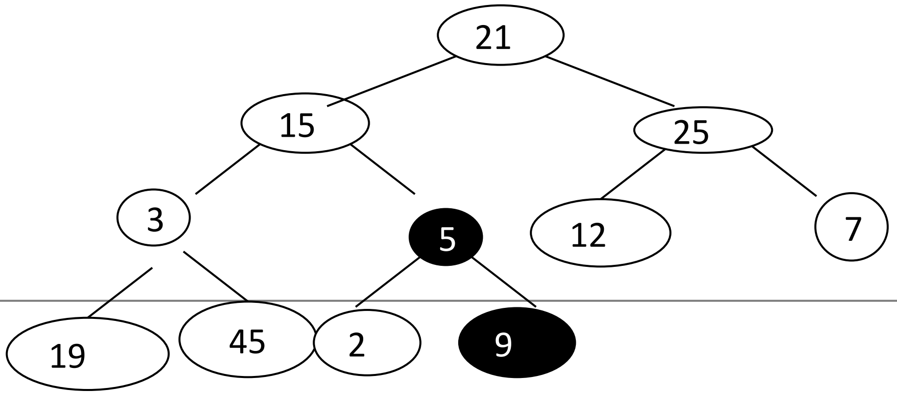


Heap sort Algorithm(index starting from 1)

```
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<= 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] >= 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 kth 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 kth 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(nlogn) 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**

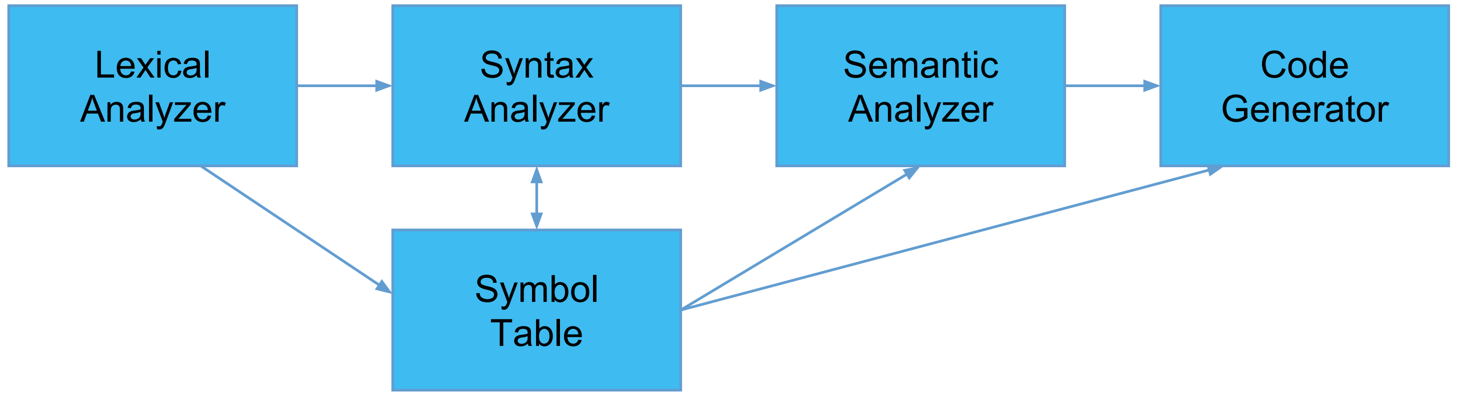
- For instance, a **type attribute is usually associated with each identifier**
- A symbol table is a necessary component
  - Definition (declaration) of identifiers appears once in a program
  - Use of identifiers may appear in many places of the program text
- Identifiers and attributes are entered 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

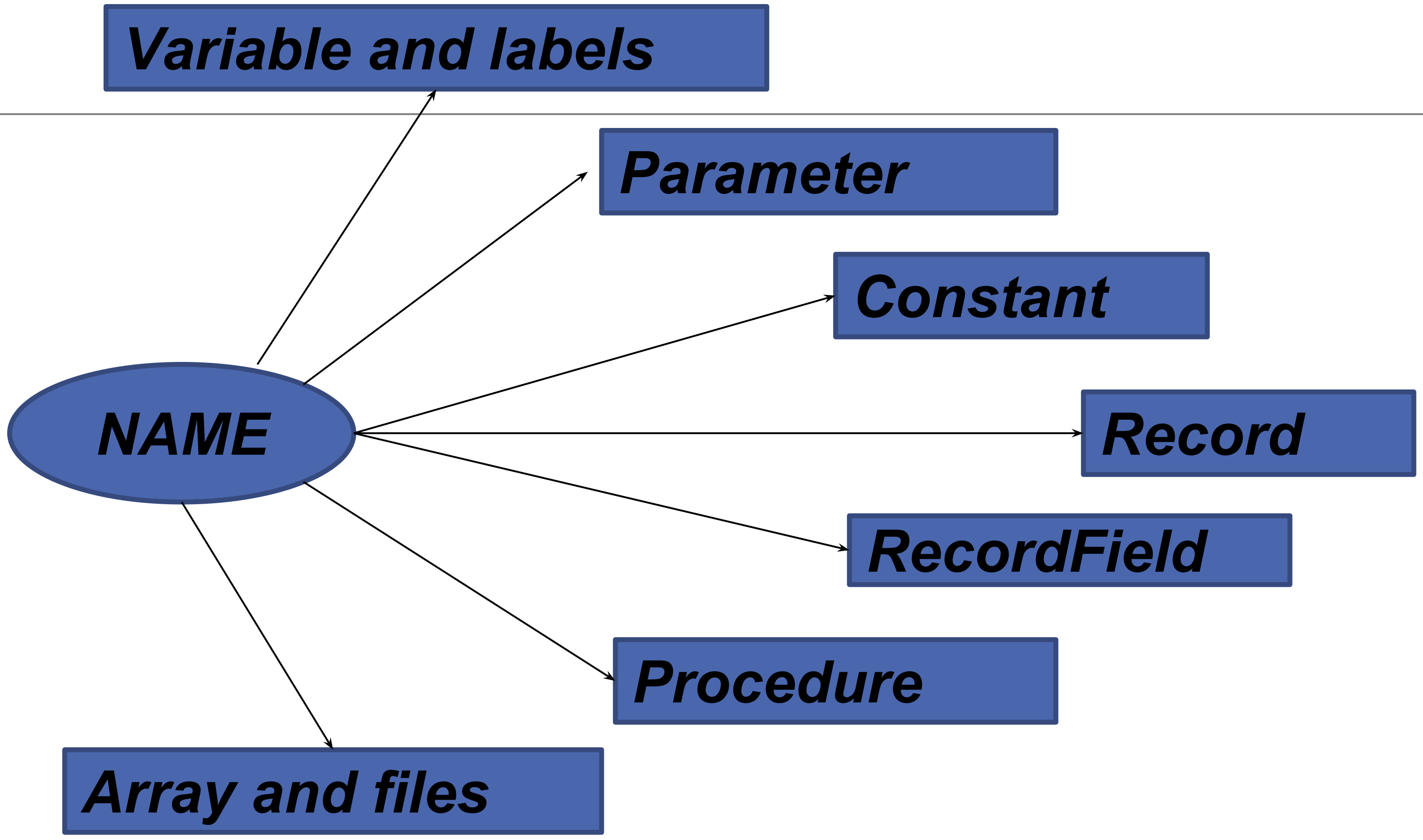
- Symbol table information is used by the analysis and synthesis phases
  - To verify that used identifiers have been defined (declared)
  - To verify that expressions and assignments are semantically correct – **type checking**
  - To generate intermediate or target code

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
•• Type , Line number where declared , Lines where referenced , Scope	•• Number of parameters, parameters themselves, result type.	•• # of Dimensions, Array bounds.

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(log2 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(log2 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,

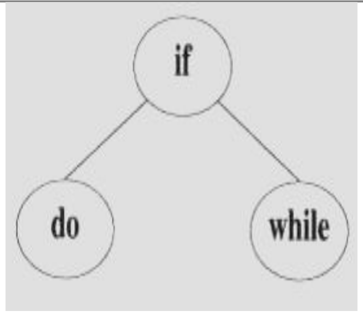
If we search for each identifier with equal Probability

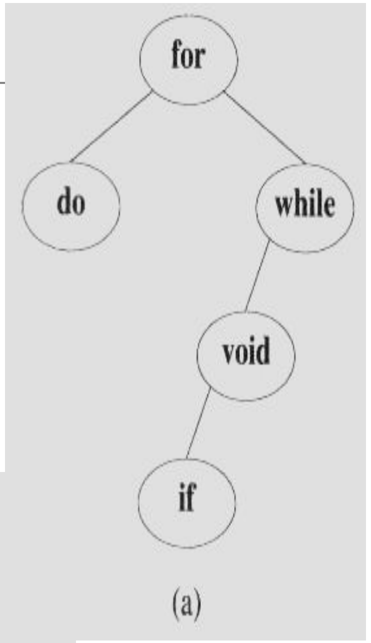
In first tree (a)

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

The second tree (b) has

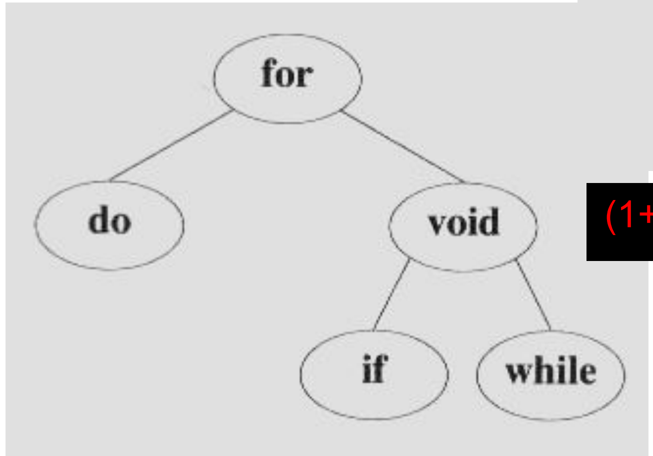
- a better worst case search time than the first tree.
- a better average behavior.





(a)

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



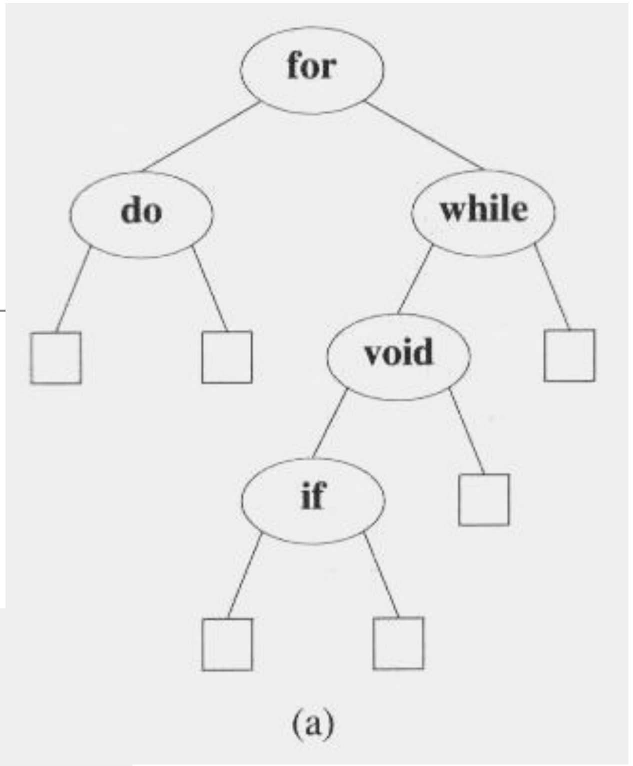
(b)

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

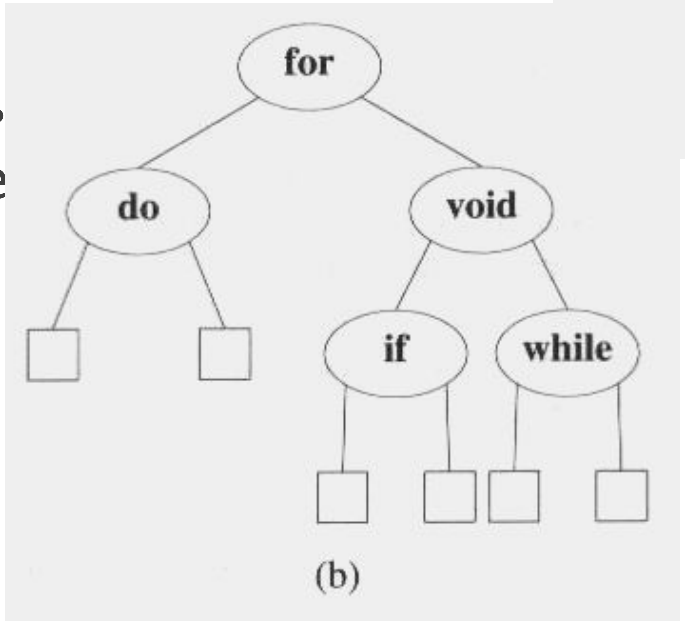
Optimal binary search trees

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*



(a)



(b)

Optimal binary search trees

External / internal path length

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

For example

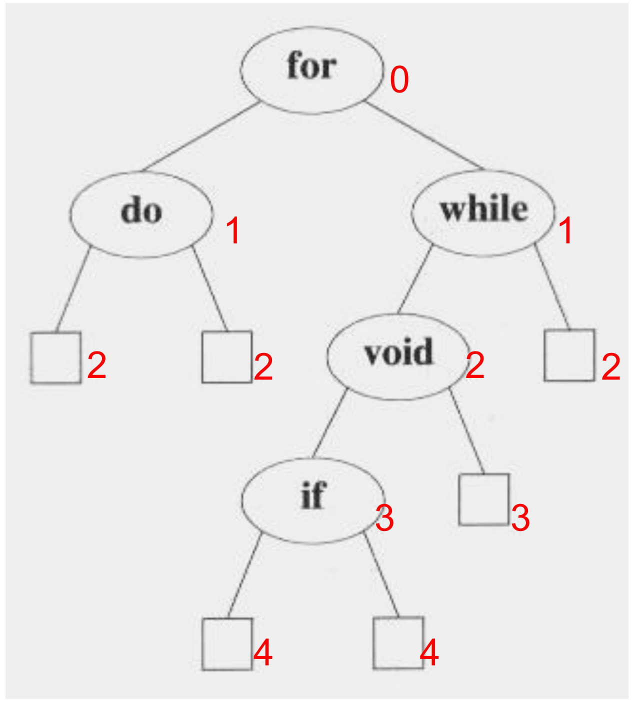
- Internal path length, *I*, is:

  $I = 0 + 1 + 1 + 2 + 3 = 7$
- 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 $a_1, a_2, \ldots, a_n$ with $a_1 < a_2 < \ldots < a_n$
- 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, $0 \le i \le n$

- 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 form 0 to *n* with *i* being for class E_i, $0 \le i \le n$.

- 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:

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

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

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

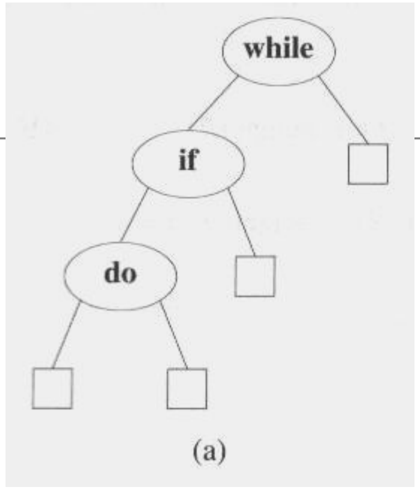
- An optimal binary search tree for the identifier set $a_1, \ldots, a_n$ is one that minimizes
  - Since all searches must terminate either successfully or unsuccessfully, we have

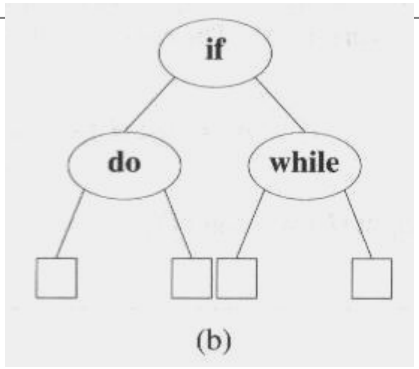
$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$

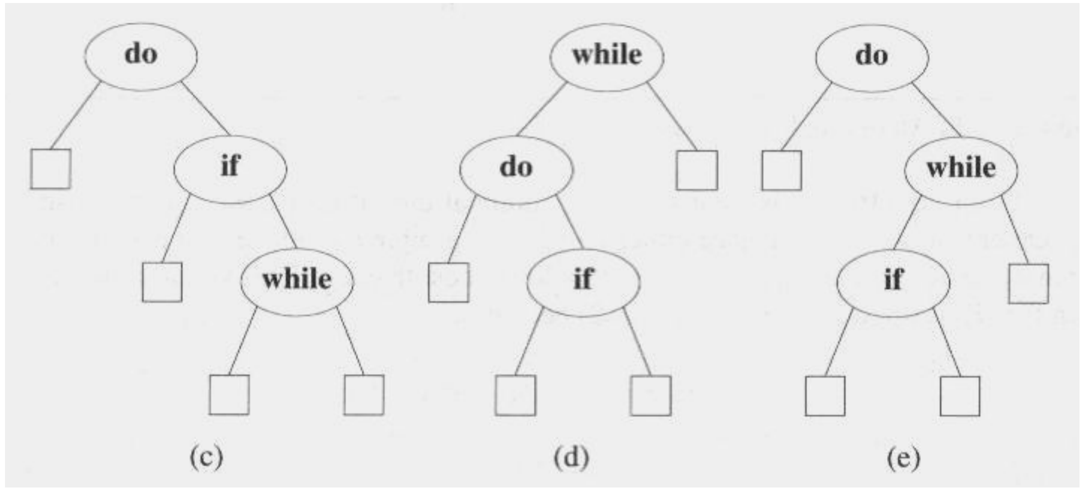
Optimal binary search trees

The possible binary search trees for the identifier set (a_1, a_2, a_3) = (**do**, **if**, **while**)

- The identifiers with equal probabilities, $p_i=q_j=1/7$ for all *i*, *j*,
  - cost(tree *a*) = 15/7;
  - cost(tree *b*) = 13/7; (optimal)
    cost(tree *c*) = 15/7;
  - cost(tree *d*) = 15/7;
    cost(tree *e*) = 15/7;
- $p_1 = 0.5$, $p_2 = 0.1$, $p_3 = 0.05$, $q_0 = 0.15$, $q_1 = 0.1$, $q_2 = 0.05$, $q_3 = 0.05$
  - cost(tree *a*) = 2.65;
    cost(tree *b*) = 1.9;
    cost(tree *c*) = 1.5; (optimal)
    cost(tree *d*) = 2.05;
    cost(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.

Cost(tree a)=∑P(i)*level a(i) +∑Q(i)*level (Ei) -1

1≤i≤n   0≤i≤n

(2-1)   (3-1)   (4-1)   (4-1)

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

Cost (tree b) =17[1+2+2+2+2+2+2] =13/7

Cost (tree c) =cost (tree d) =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) =.15 , Q(1) =.1, Q(2) =.05 and Q(3) =.05 find the OBST.

Cost (tree a) = .5 x 3 +.1 x 2 +.05 x 3 +.15x3 +.1x3 +.05x2 +.05x1 = 2.65

Cost (tree b) =1.9 , Cost (tree c) =1.5 ,Cost (tree d) =2.05 ,

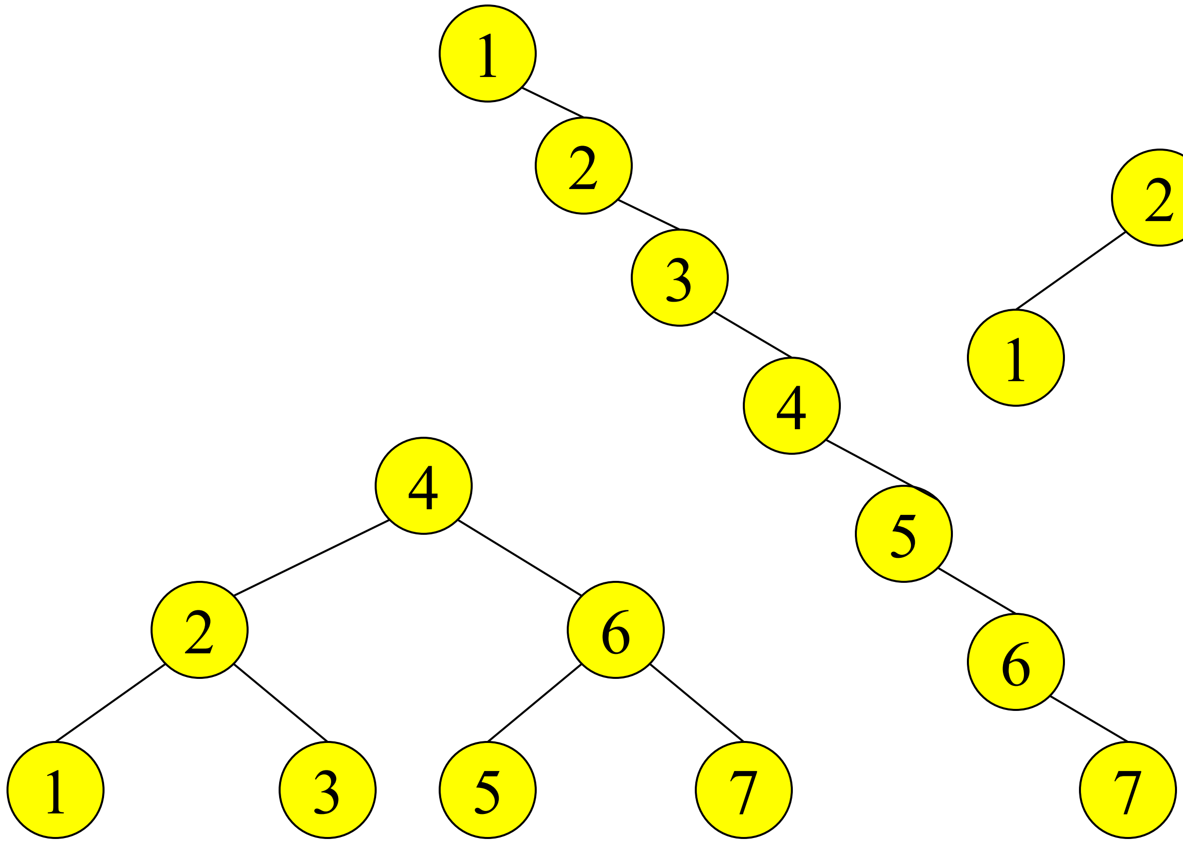
Cost (tree e) =1.6 Hence tree C is optimal.

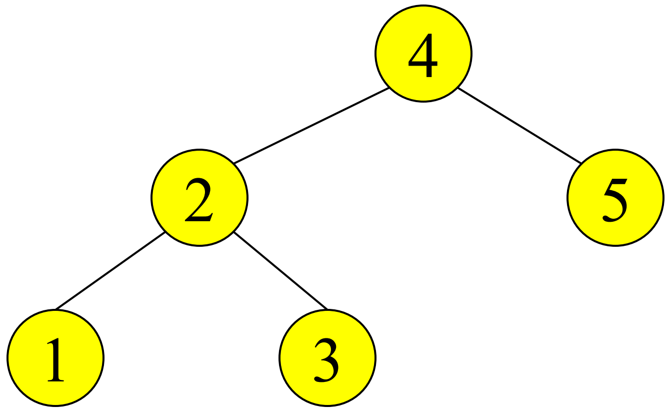
Binary Search Tree - Worst Time

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





Is this “balanced”?

Balancing Binary Search Trees

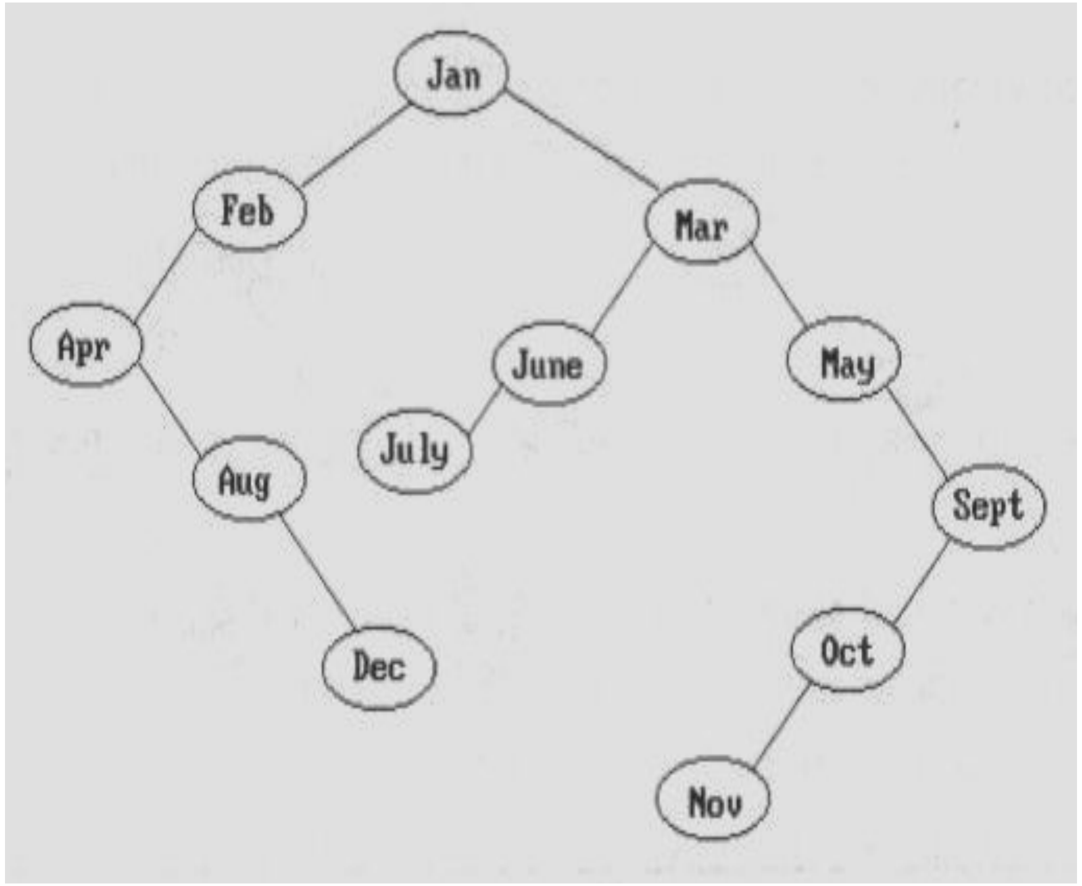
Many algorithms exist for keeping binary search trees balanced

- 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.

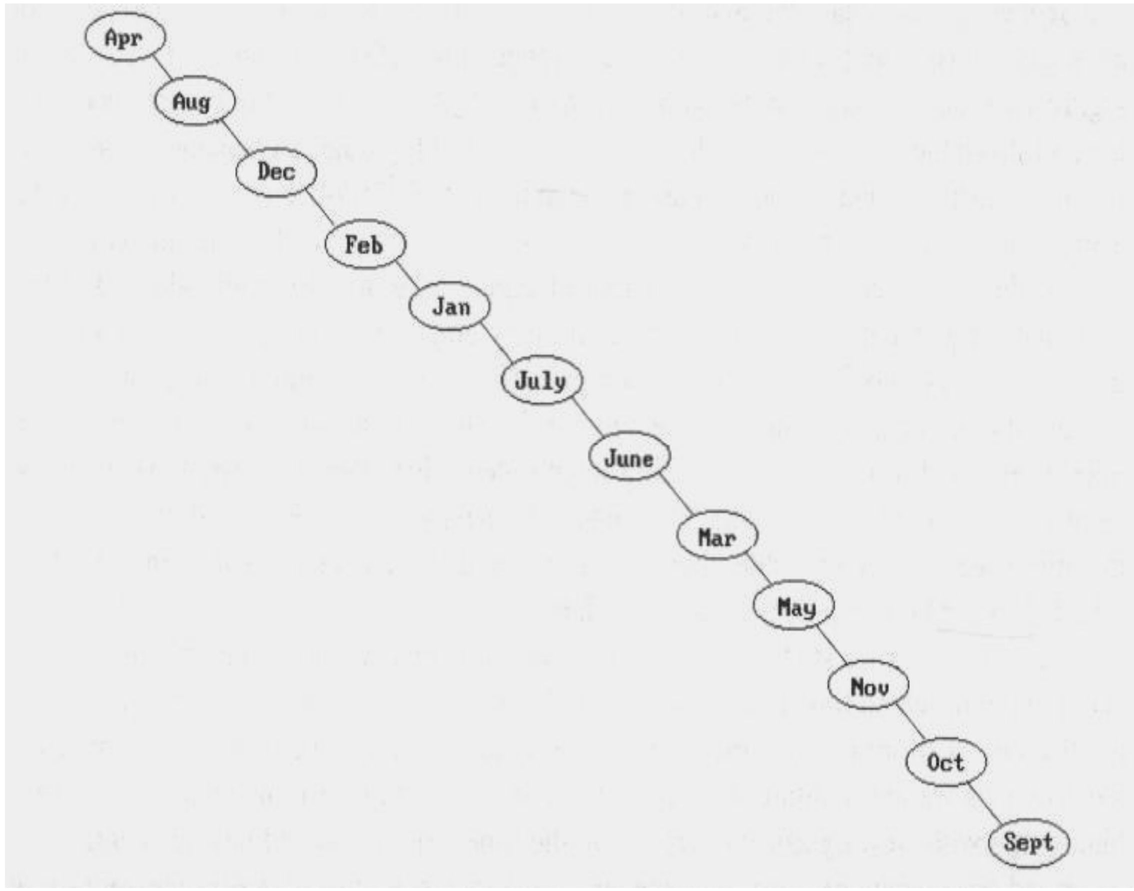
- The binary search tree obtained by entering the months *January* to *December*, in that order, into an initially empty binary search tree
- The maximum number of comparisons needed to search for any identifier in the tree (for *November*).
- Average number of comparisons is 42/12 = 3.5



AVL Trees

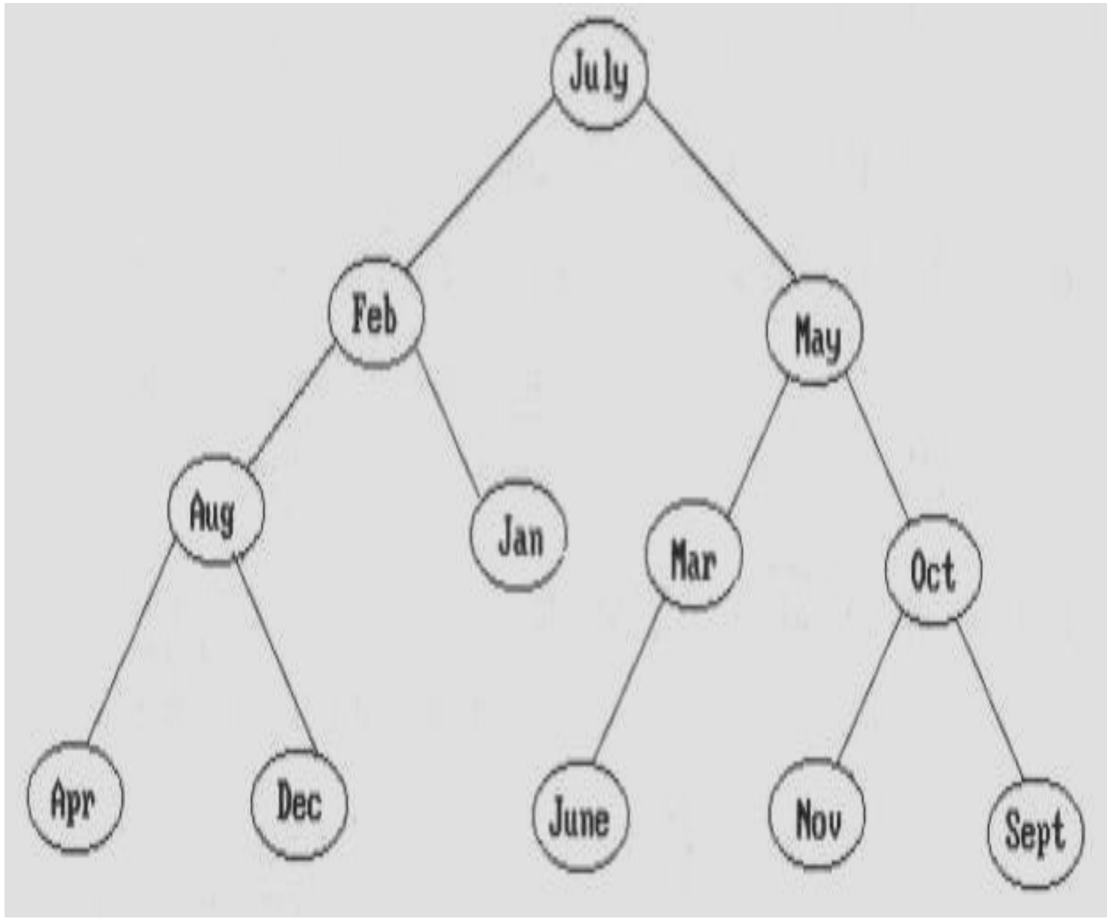
Suppose that we now enter the months into an initially empty tree in alphabetical order

- 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 ≈ 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 form it, in O(log*n*) time. The resulting tree remain height balanced.
- As with binary trees, we may define AVL tree recursively

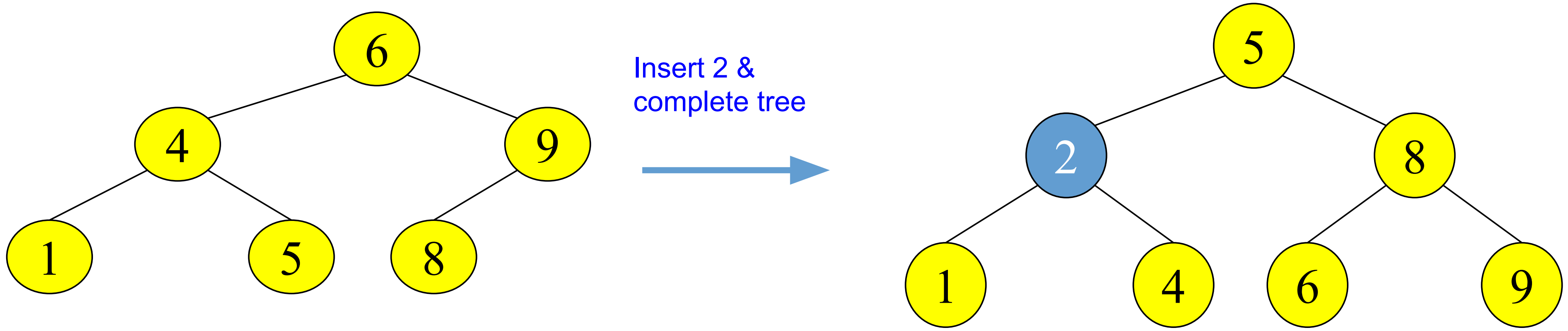
Perfect Balance

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

- height(left subtree) - height(right subtree)

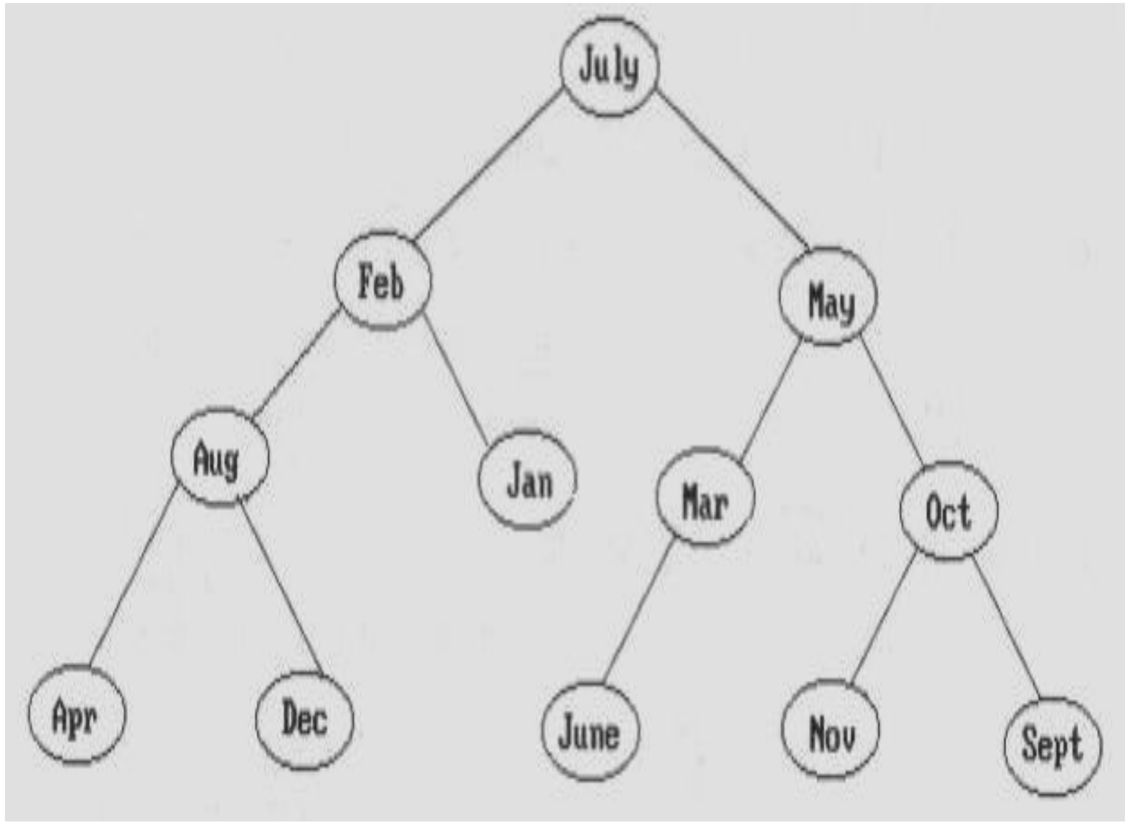
An AVL tree has balance factor calculated at every node

- 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 ±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 α.

There are 4 cases:

Outside Cases (require single rotation) :

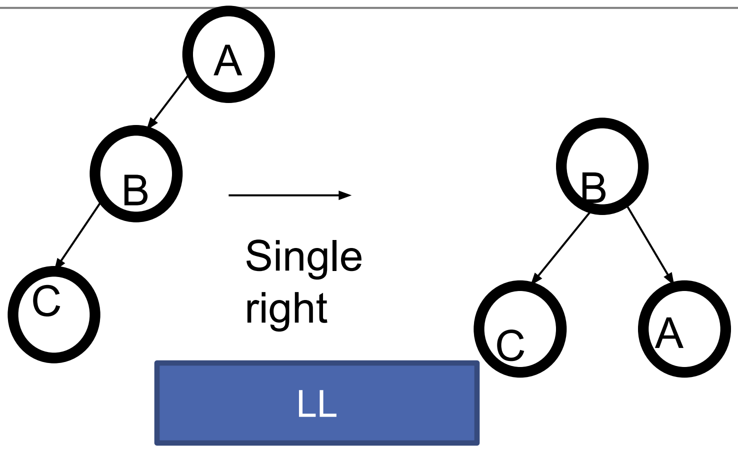
1. Insertion into left subtree of left child of α.
2. Insertion into right subtree of right child of α.

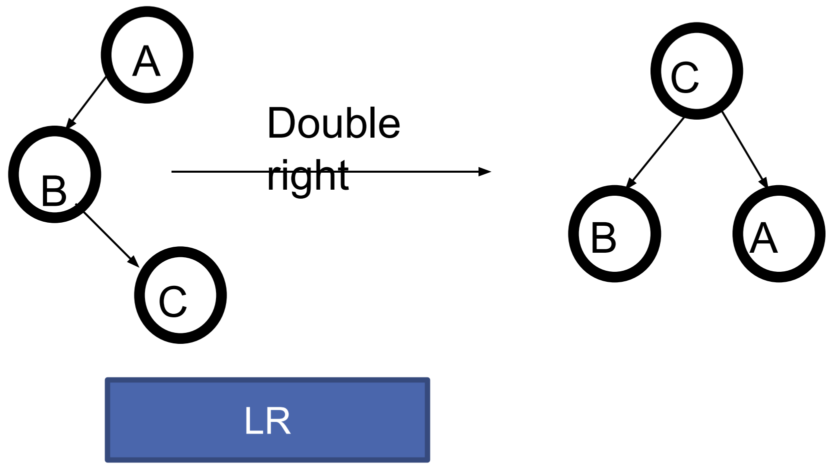
Inside Cases (require double rotation) :

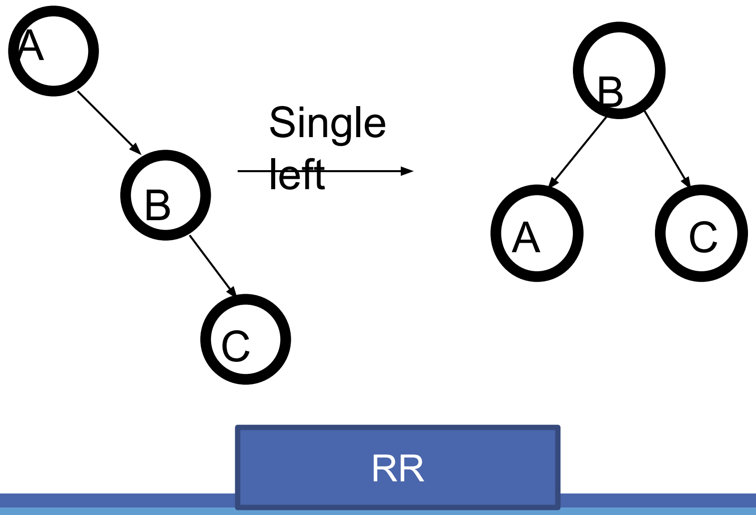
3. Insertion into right subtree of left child of α.
4. Insertion into left subtree of right child of α.

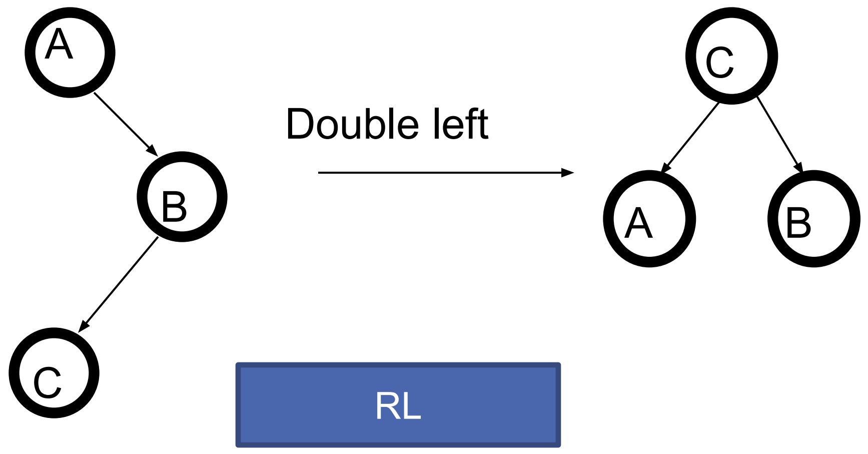
The rebalancing is performed through four separate rotation algorithms.

AVL Balancing : Four Rotations

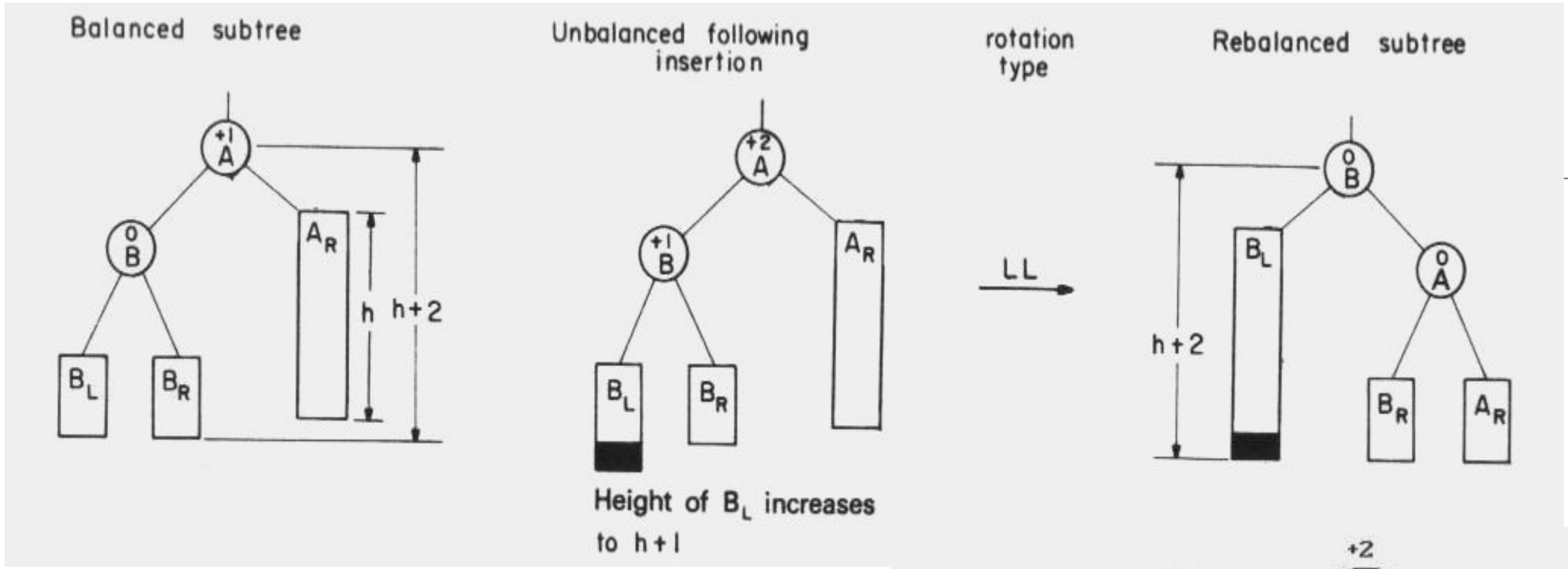


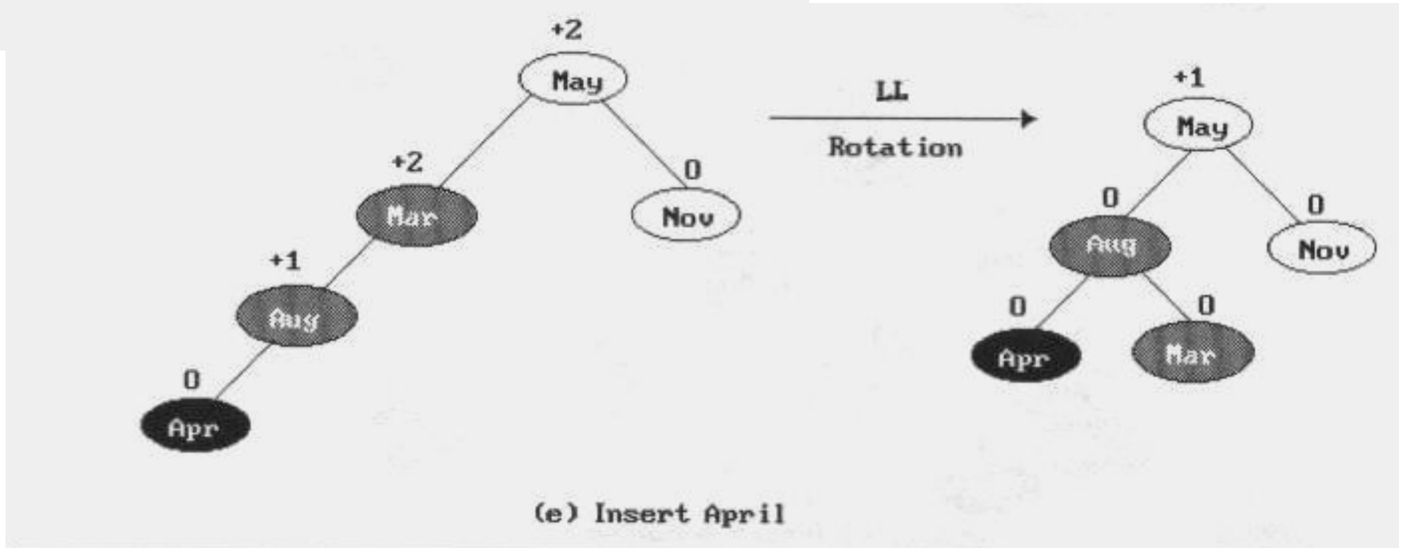






Rebalancing rotations

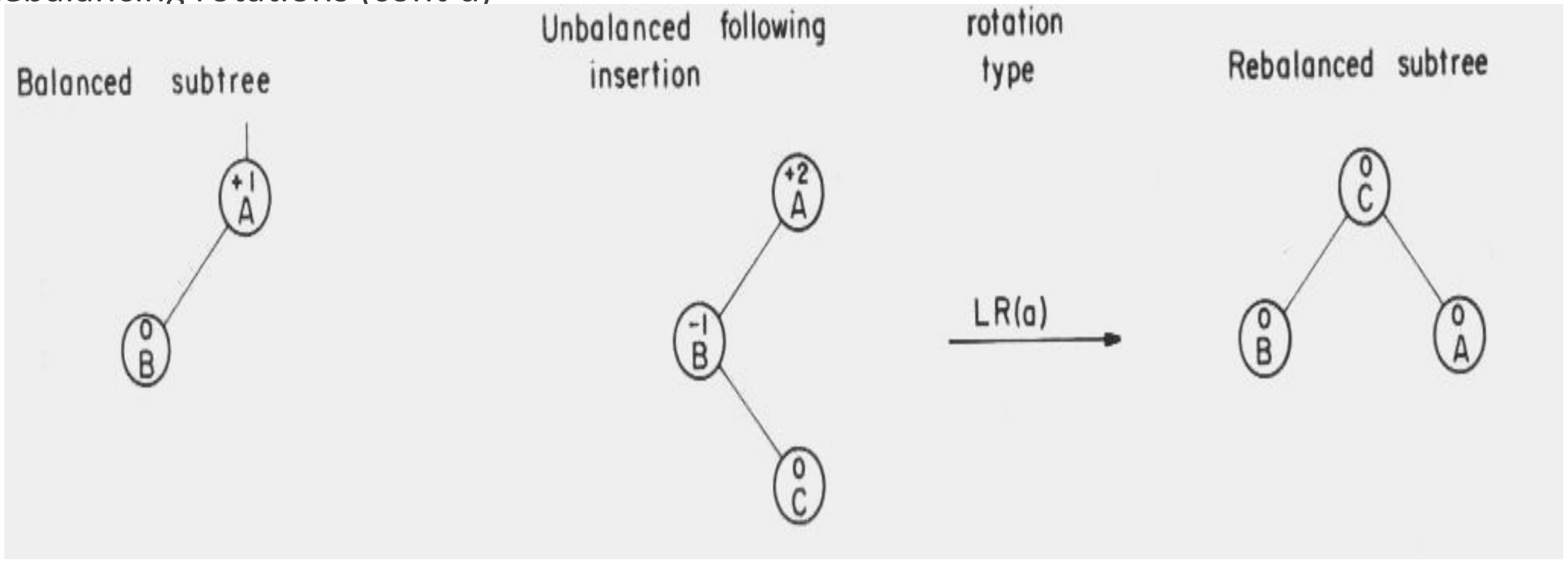




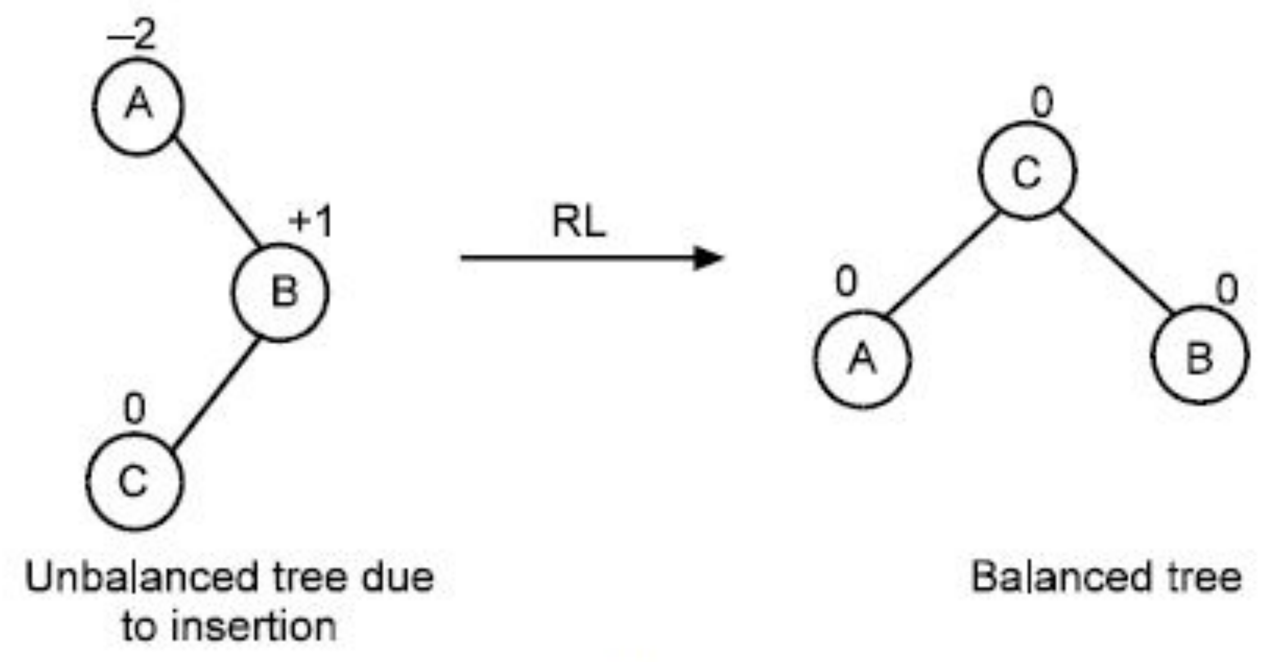
(e) Insert April

AVL Trees

Rebalancing rotations (cont'd)



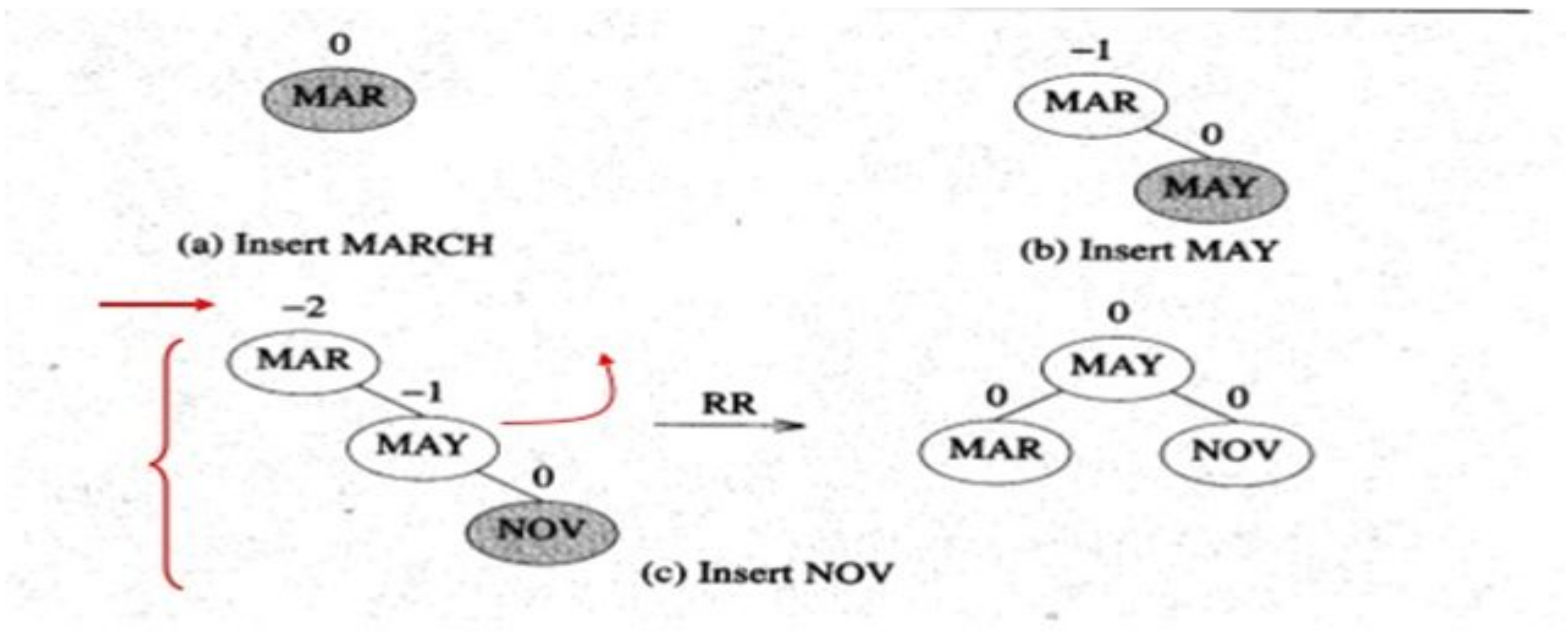
Case 4: RL (Left of Right)

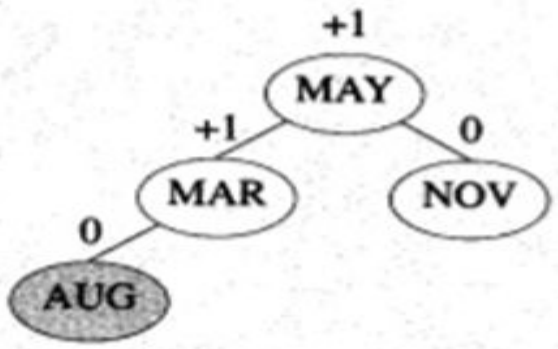


(a)

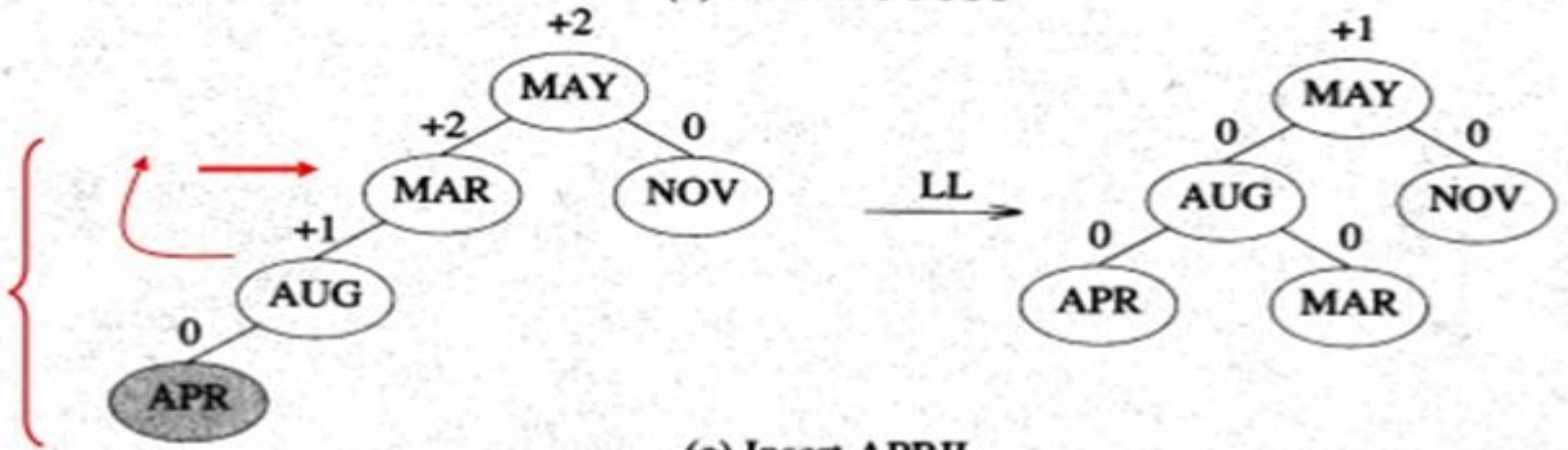
- This time we will insert the months into the tree in the order
- *Mar*, *May*, *Nov*, *Aug*, *Apr*, *Jan*, *Dec*, *Jul*, *Feb*, *Jun*, *Oct*, *Sep*

- It shows the tree as it grows, and the restructuring involved in keeping it balanced.



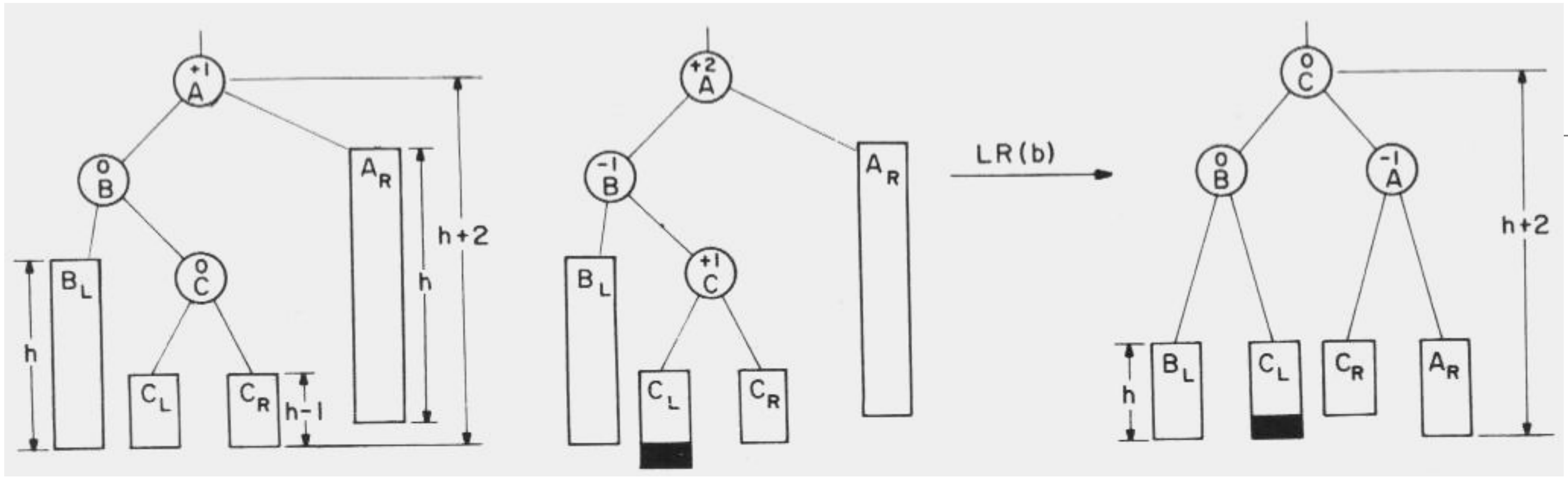


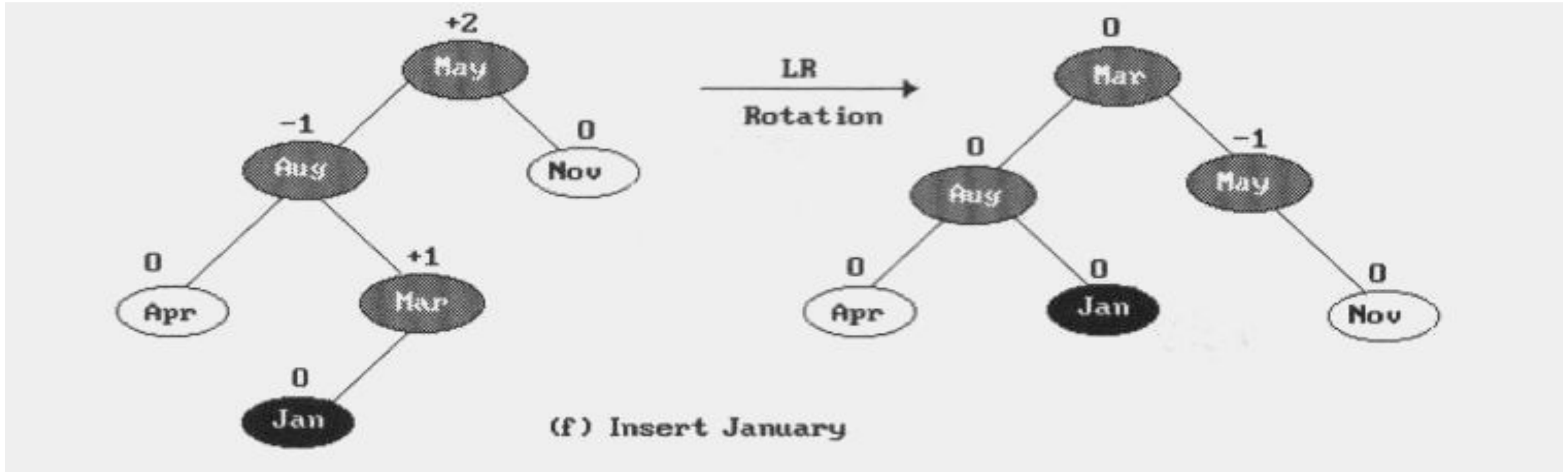
(d) Insert AUGUST



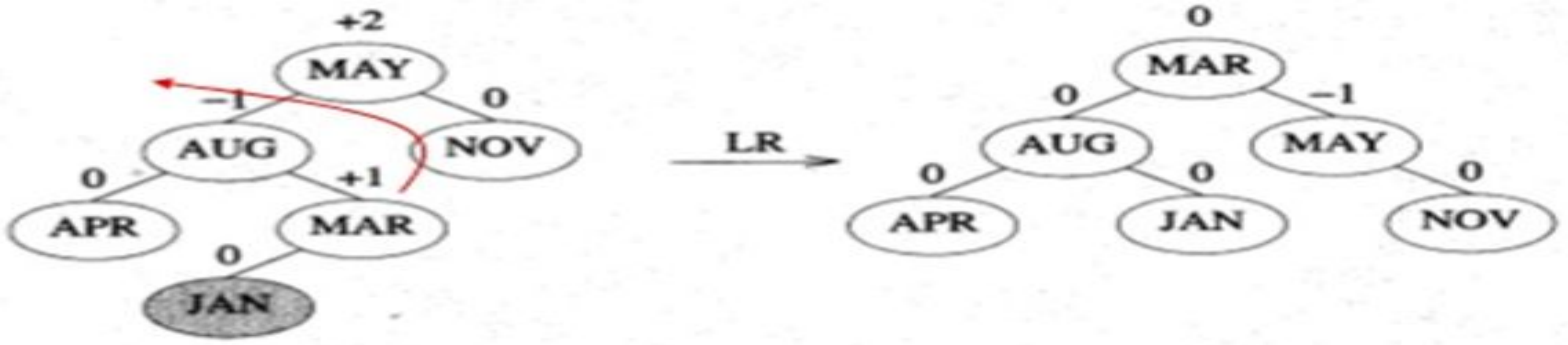
(e) Insert APRIL

Rebalancing rotations (cont'd)

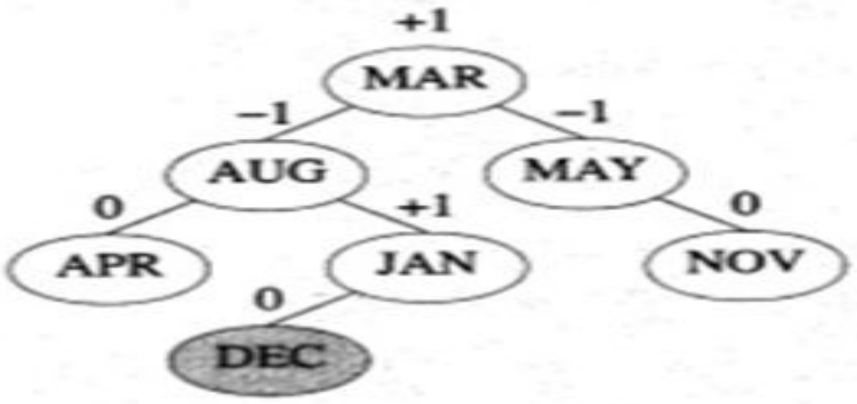




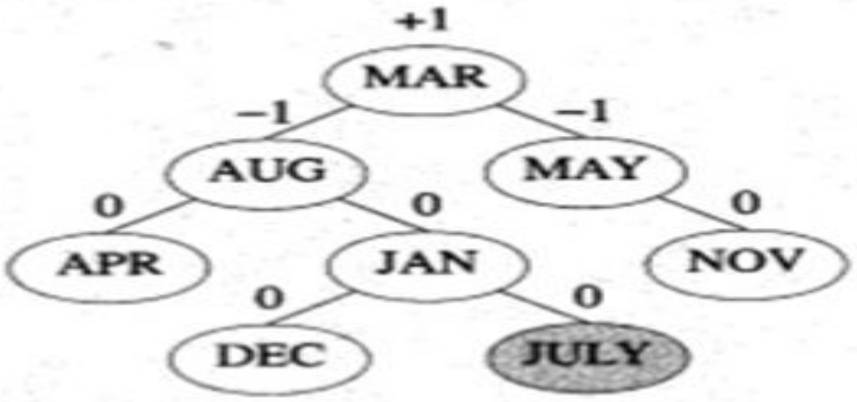
(f) Insert January



(f) Insert JANUARY

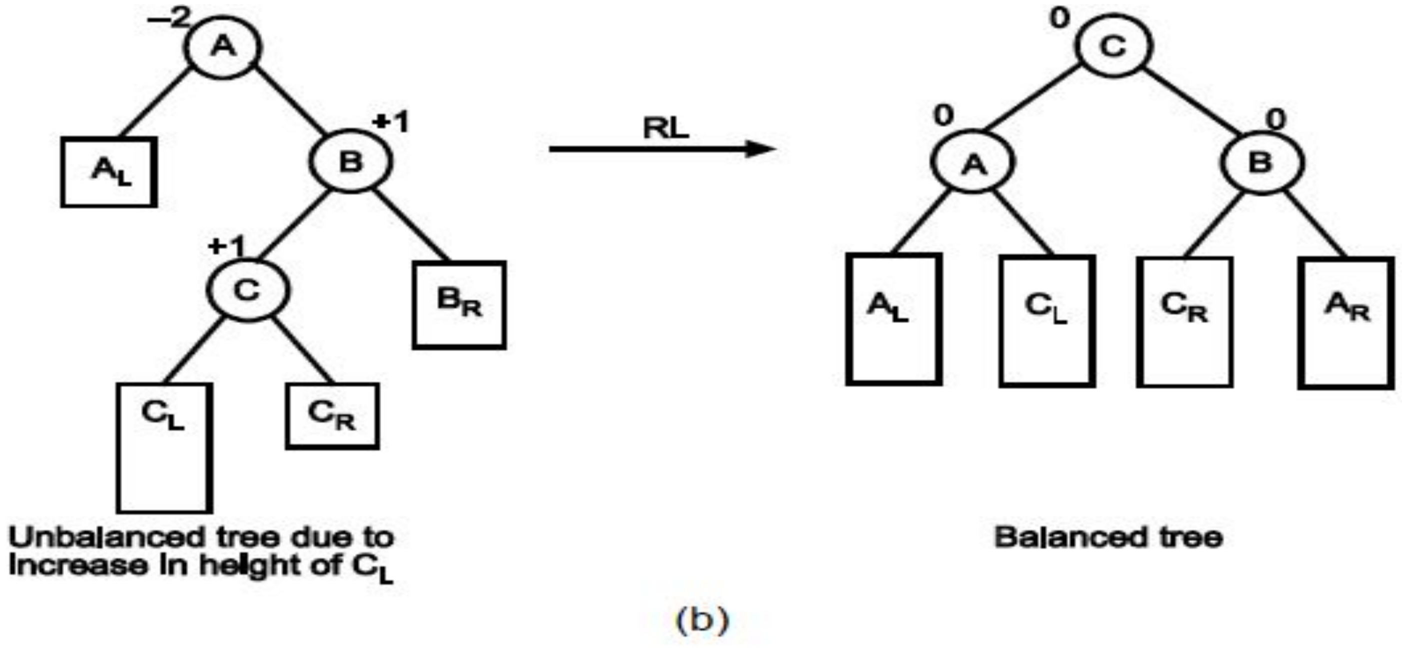


(g) Insert DECEMBER

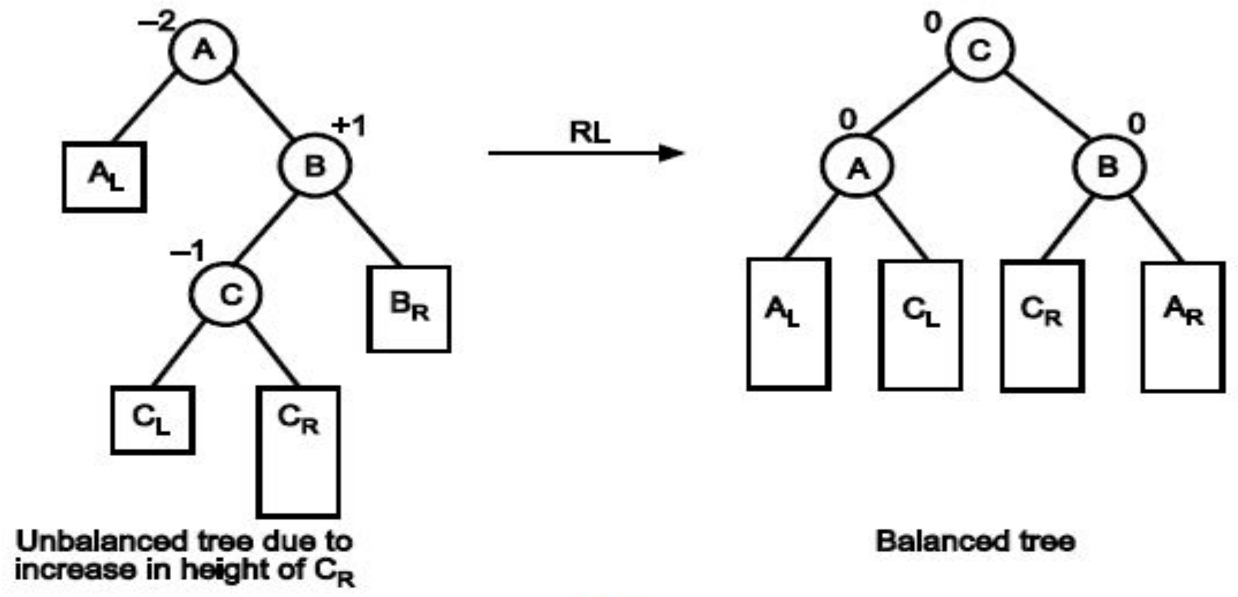


(h) Insert JULY

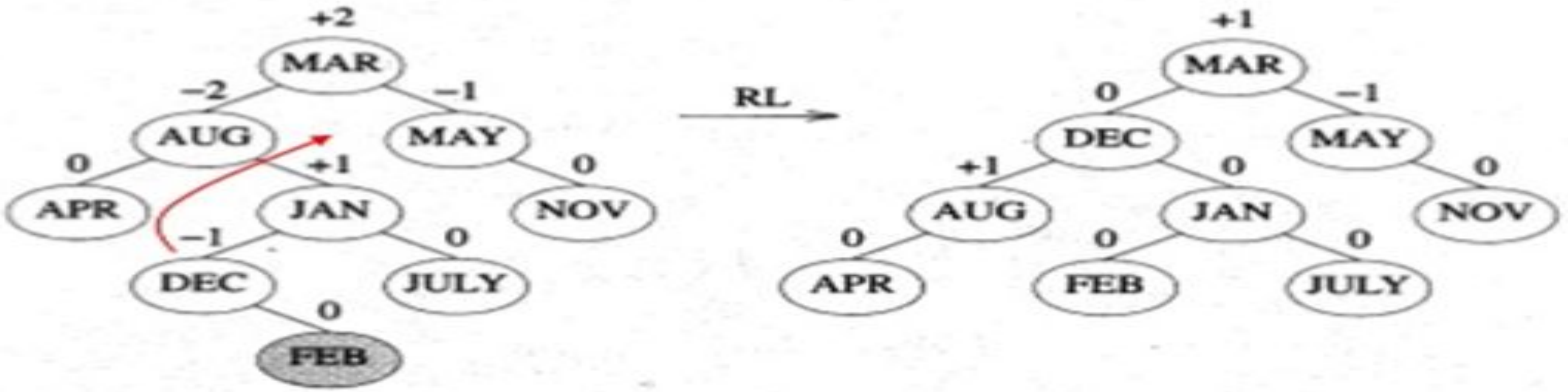
Case 4: RL(Left of Right)



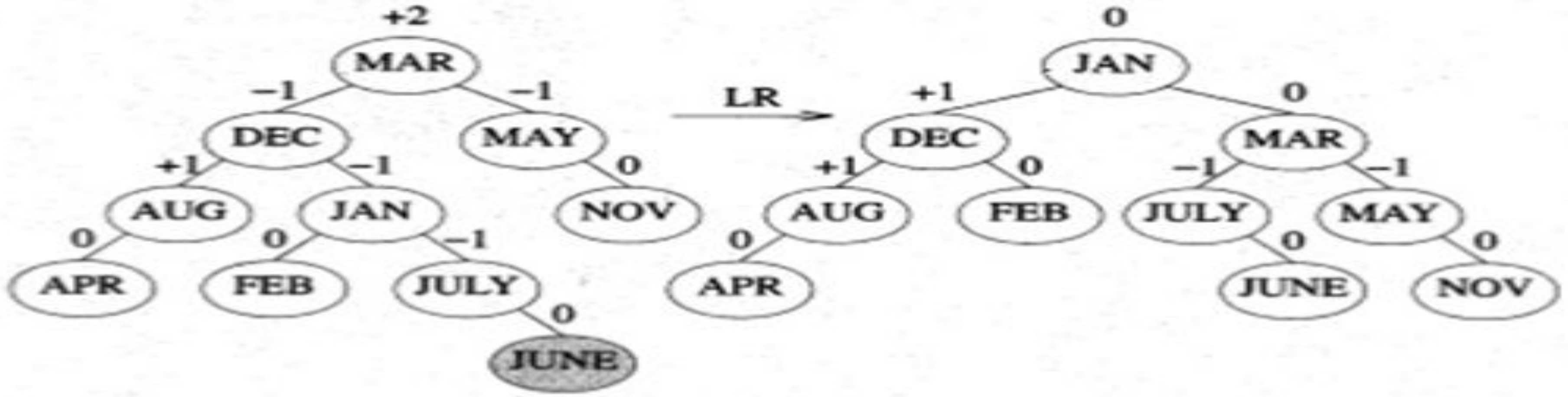
Case 4: RL (Left of Right)



(c)

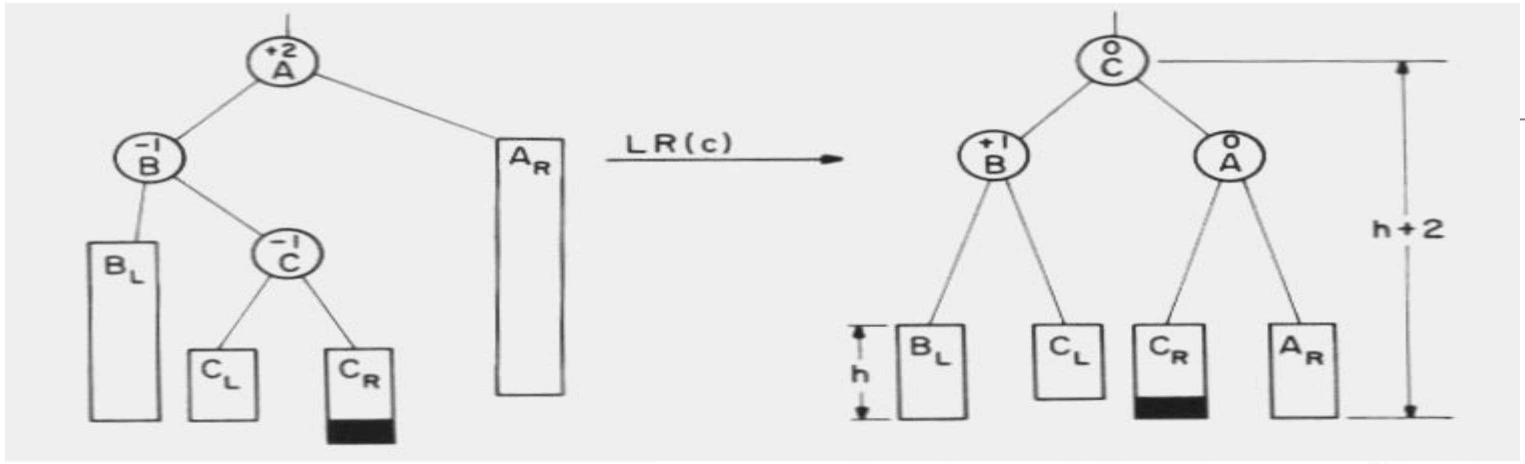


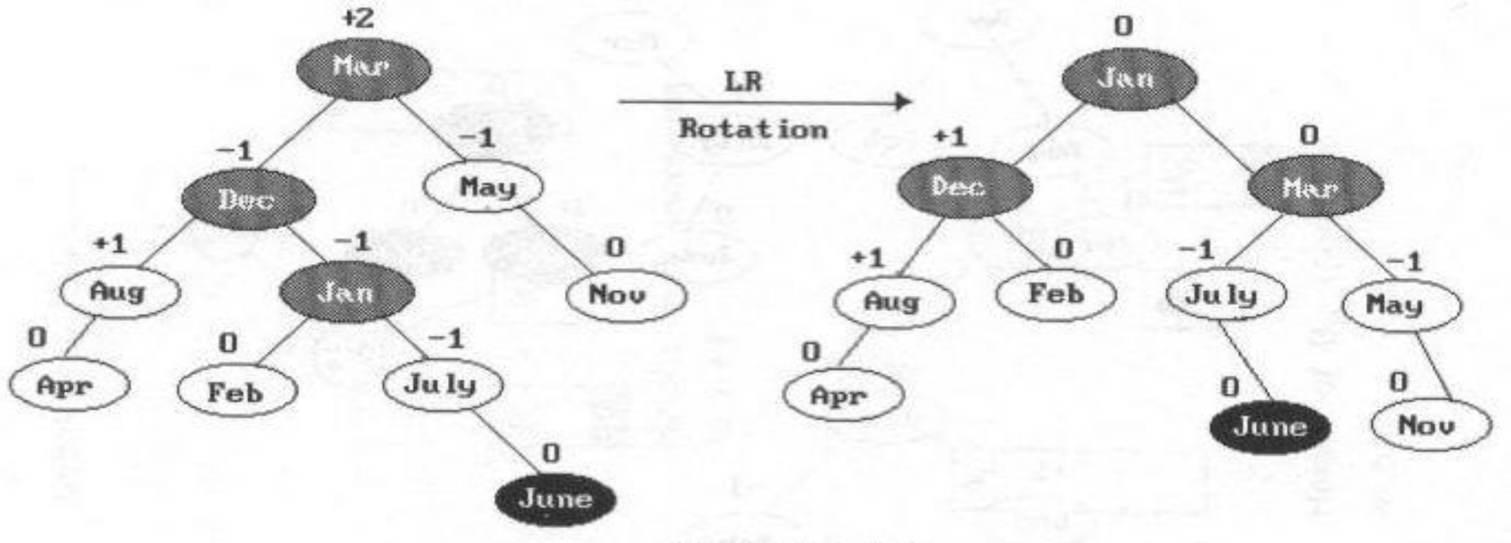
(i) Insert FEBRUARY



(j) Insert JUNE

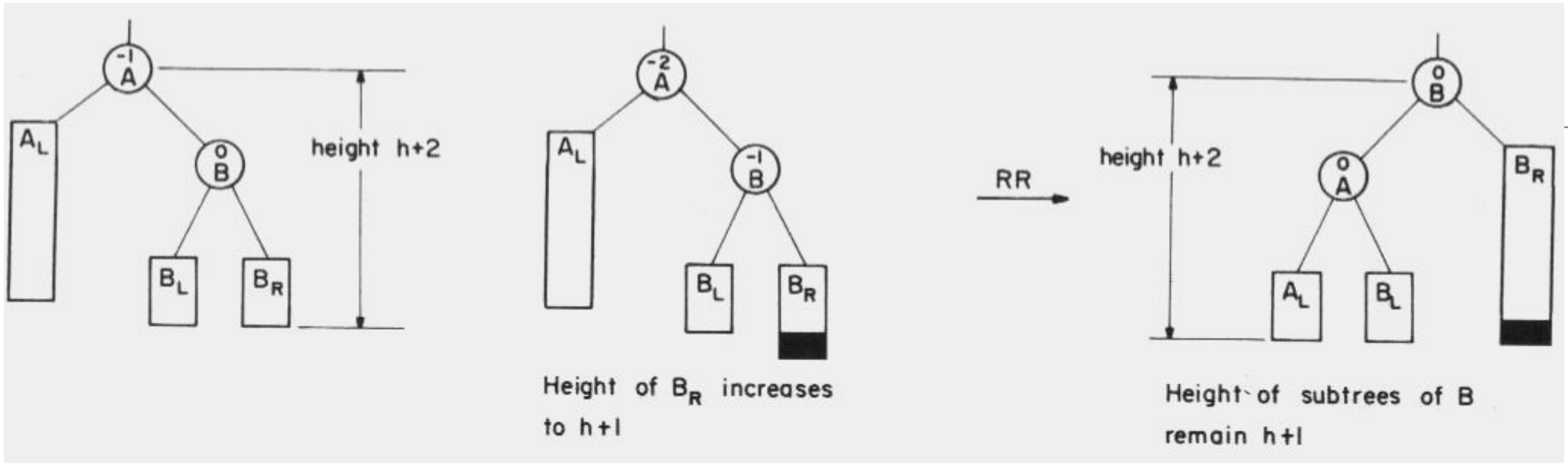
Case 3: LR (Left of Right)



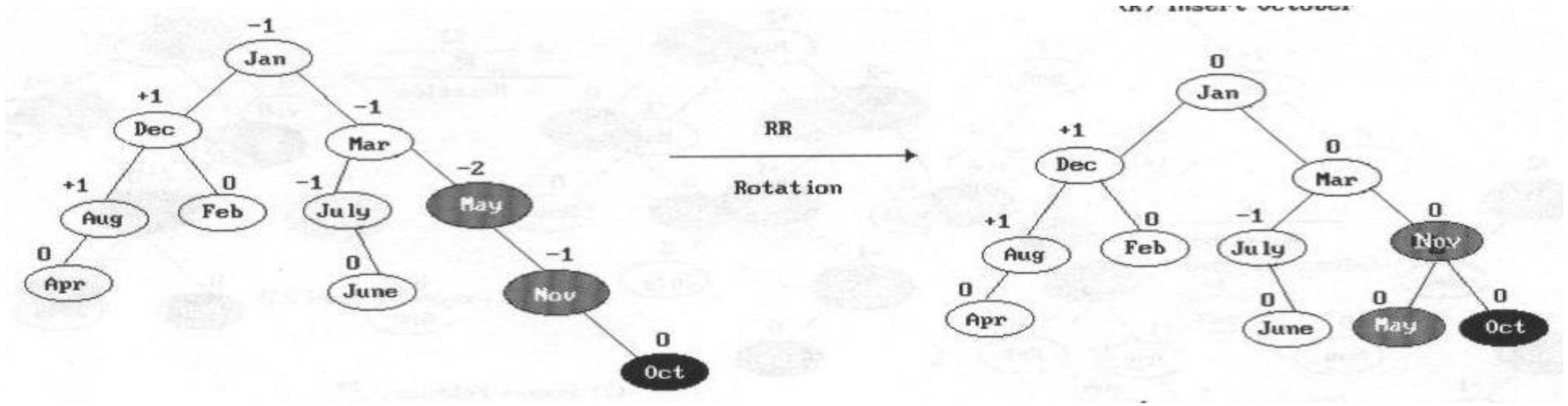


(j) Insert June

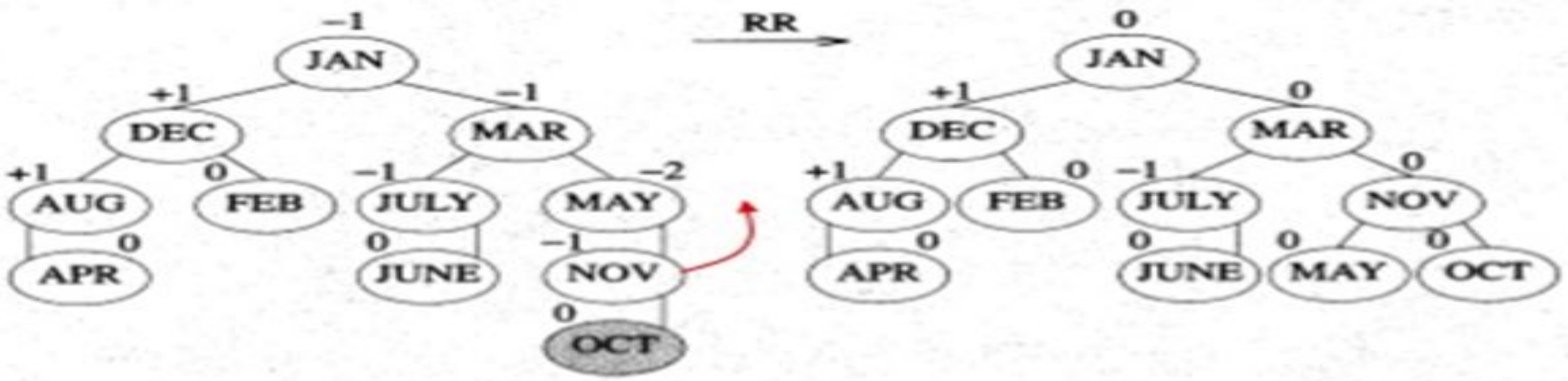
AVL Trees



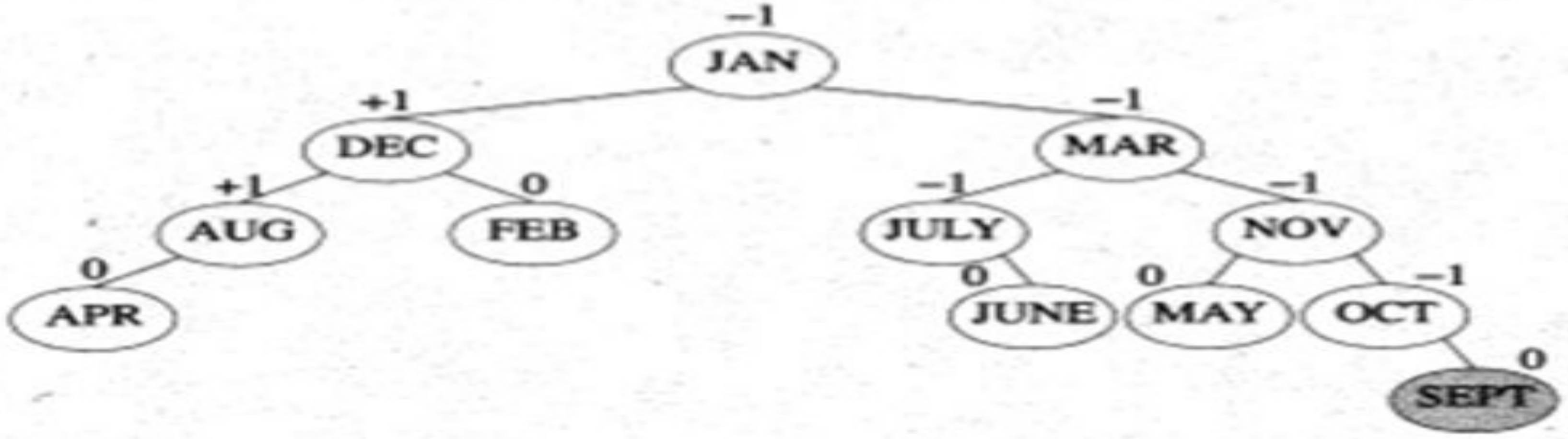
-1
A
A_L
0
B
B_L
B_R
height h+2
-2
A
A_L
-1
B
B_L
B_R
Height of B_R increases to h+1
RR
height h+2
0
B
0
A
A_L
B_L
B_R
Height of subtrees of B remain h+1



-1
Jan
+1
Dec
-1
Mar
+1
Aug
0
Feb
-1
July
-2
May
0
Apr
0
June
-1
Nov
0
Oct
RR
Rotation
0
Jan
+1
Dec
0
Mar
+1
Aug
0
Feb
-1
July
0
Nov
0
Apr
0
June
0
May
0
Oct



(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;
}
```

LR Rotation

```
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);
    }
}
```

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