JAIN DEEMED-TO-BE UNIVERSITY | SCHOOL OF COMPUTER SCIENCE AND IT

Master of Computer Applications

Data Structures

Module 4

Advanced Trees & Hashing

SCHOOL OF COMPUTER SCIENCE AND IT

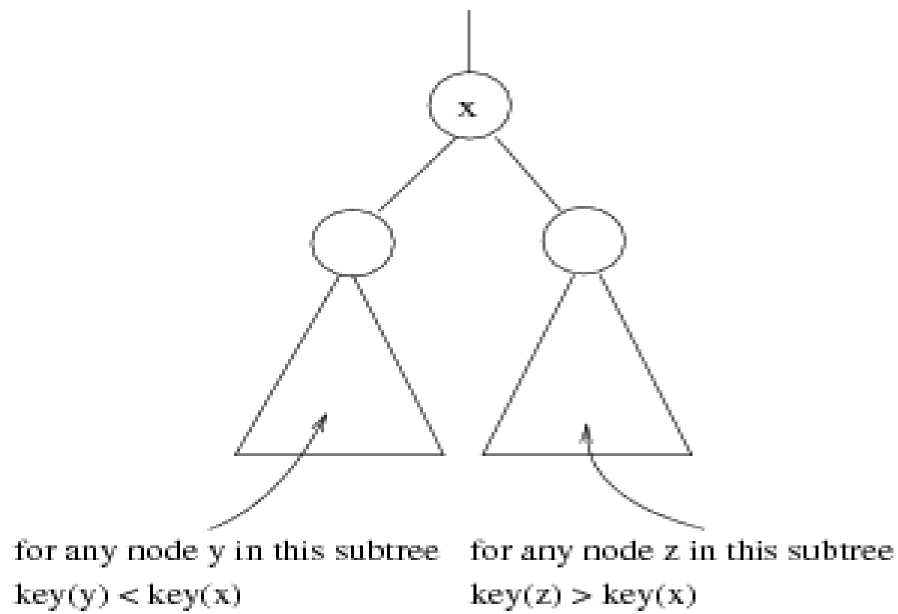
Syllabus Contents

- Binary Search Tree- Operations.
- AVL trees
- Threaded Binary Tree
- B Tree & B+ Tree,
- Heaps, Types, Operations and Applications
- Hashing, Hashing functions, Collision Strategy.

Binary Search Trees

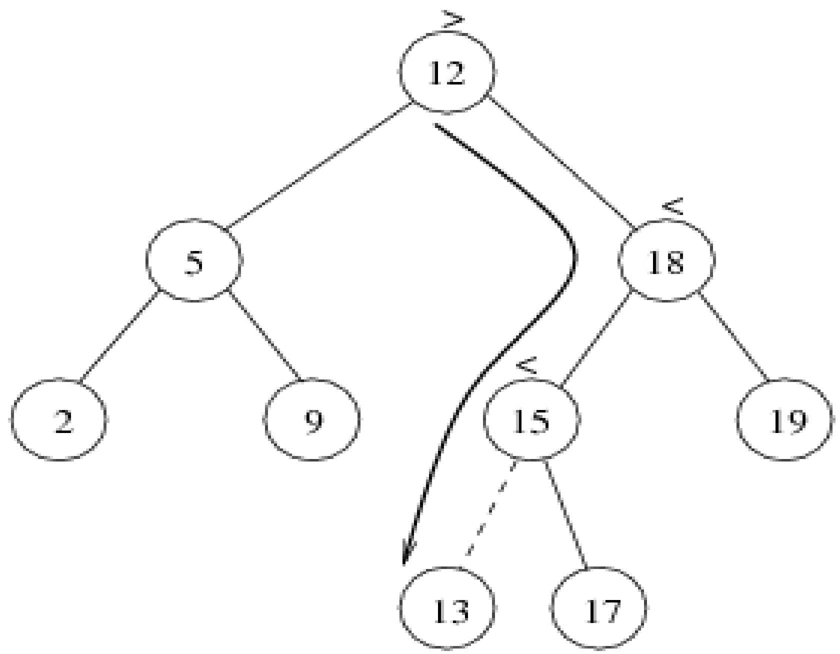
- **Stores keys in the nodes in a way so that searching, insertion and deletion** can be done efficiently.
- **Binary search tree** property

value(LST) < value (Root) < **value (RS)**



BST - Insert

- **Find the place** where the item to be attached
- **Once place is found attach the new item** on the traversed path



- **Time complexity** = O(height of the tree)

Steps for Inserting in BST

Using Recursive structure: insert(Root, data)

1. **'Root'** is NULL then **create new node** and **return it**

2. If **Root->data < data**

   Root->right = insert(Root->right, data)

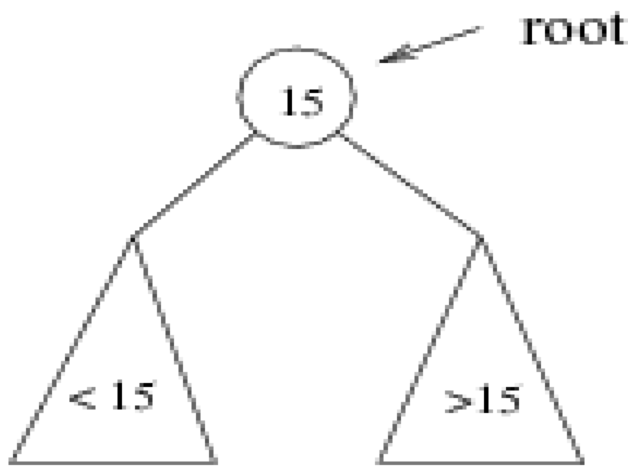
   else

   Root->left = insert(Root->left, data)

3. Return Root

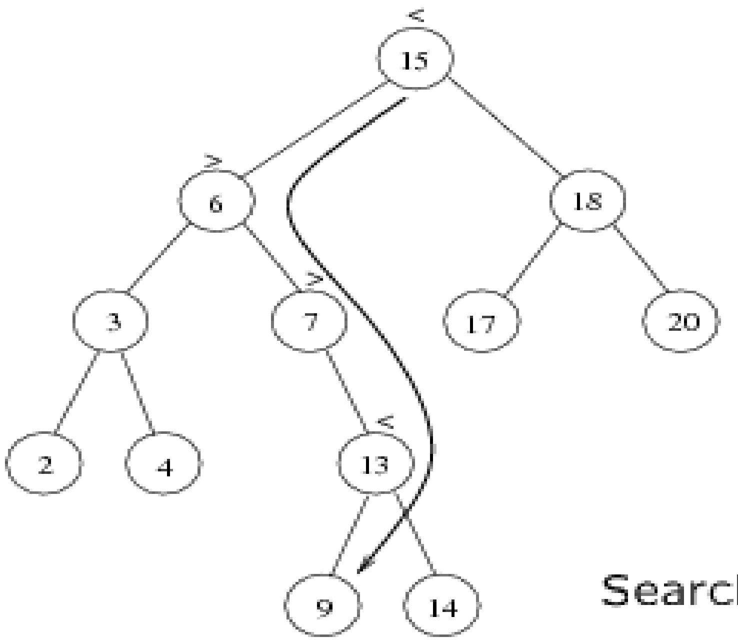
BST – Searching Operation

- If **we are searching for 15**, then **we are done.**
- If **we are searching for a key < 15**, then we **should search in the left subtree.**
- If **we are searching for a key > 15**, then we **should search in the right subtree**.



Example

Example: Search for 9 ...



Search for 9:

1. compare 9:15(the root), go to left subtree;
2. compare 9:6, go to right subtree;
3. compare 9:7, go to right subtree;
4. compare 9:13, go to left subtree;
5. compare 9:9, found it!

Steps for Searching in BST

Using Loop structure:

1. **Create** a **flag variable** and **initialize with 0**
2. Take a temp pointer for BST and assign the ‘Root’ address
3. **Loop until temp != NULL**
   if temp->data==key
   flag =1 go to step 6
4. **Else if temp->data < key**
   temp = temp->right
   go to step 3
5. **Else if temp->data>key**
   temp = temp->left
   go to step 3
6. if **flag==0,** Display **“Element is not found”**
   else
   Display **“the key is found”**
7. **Stop** the execution

Steps for Searching in BST

Using Recurisve Procedure: int SearchRec(root,key)

1. Take a **temp pointer** for BST and **assign the 'Root'** address
2. If **temp!=NULL**
   Compare temp->data with Key
   if temp->data==key then
   return 1
3. **Else if temp->data < key then**
   return **SearchRec(temp->right, key)**
4. **Else**
   return **SearchRec(temp->left, key)**

Note:

Check the return value in calling procedure to declare the result.

- **If returned value is 1 then key is found**
- **else the key is not found**

BST - Delete

3 cases:

Case -1: the node is a leaf

- **Delete it immediately**

EX: Delete(3)

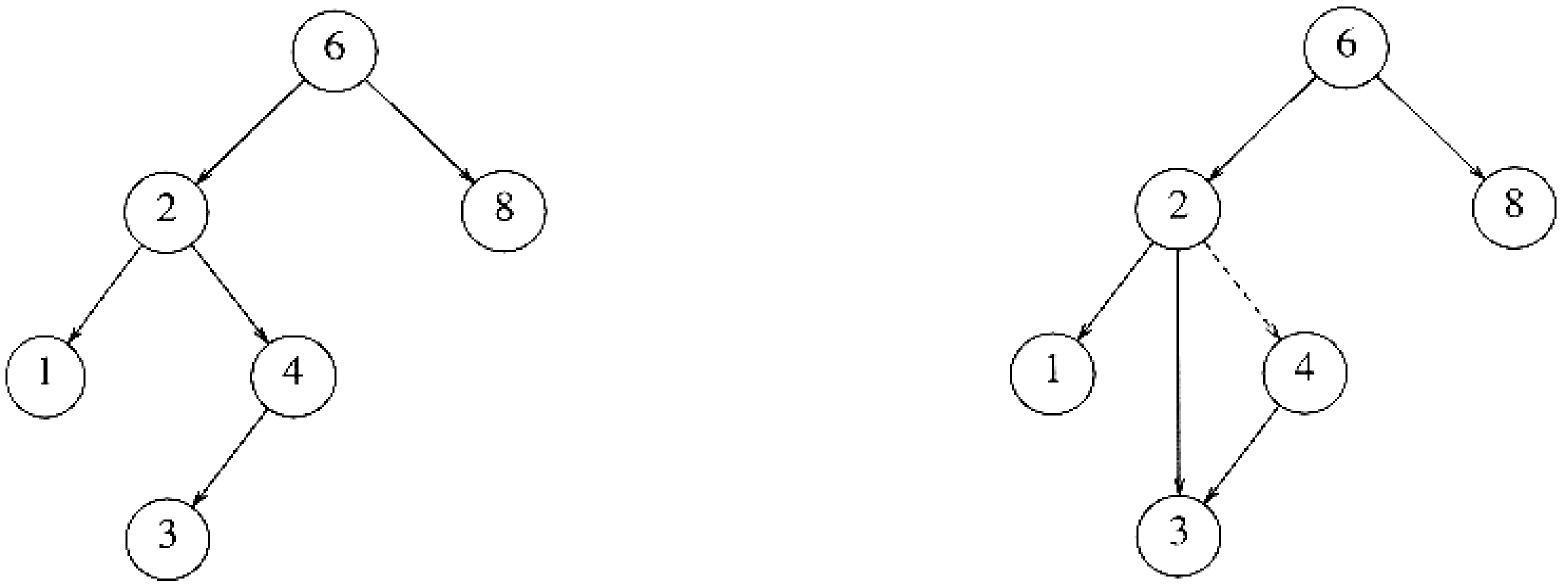


Figure 4.24 Deletion of a node (4) with one child, before and after

BST - Delete

Case 2:

the node has one child

- **Adjust a pointer from the parent** to **bypass that node**

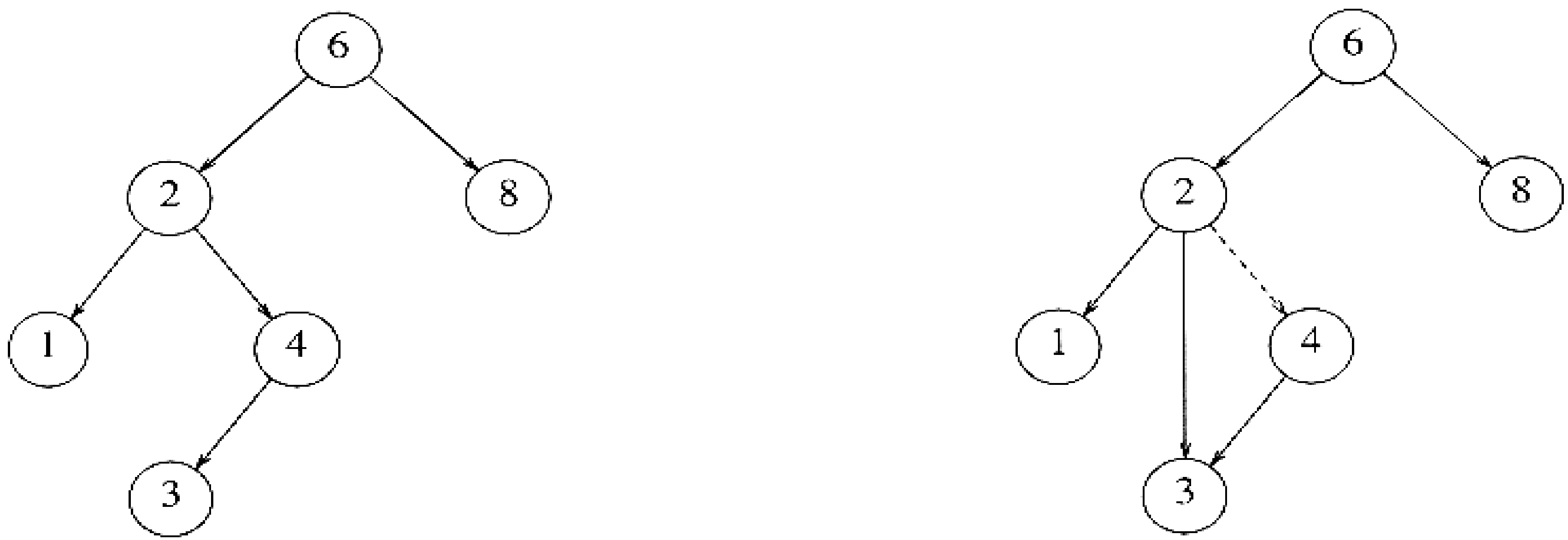


Figure 4.24 Deletion of a node (4) with one child, before and after

Contd..

Case-3:

the node has 2 children

- **replace the key of that node** with the **minimum element at the right subtree**

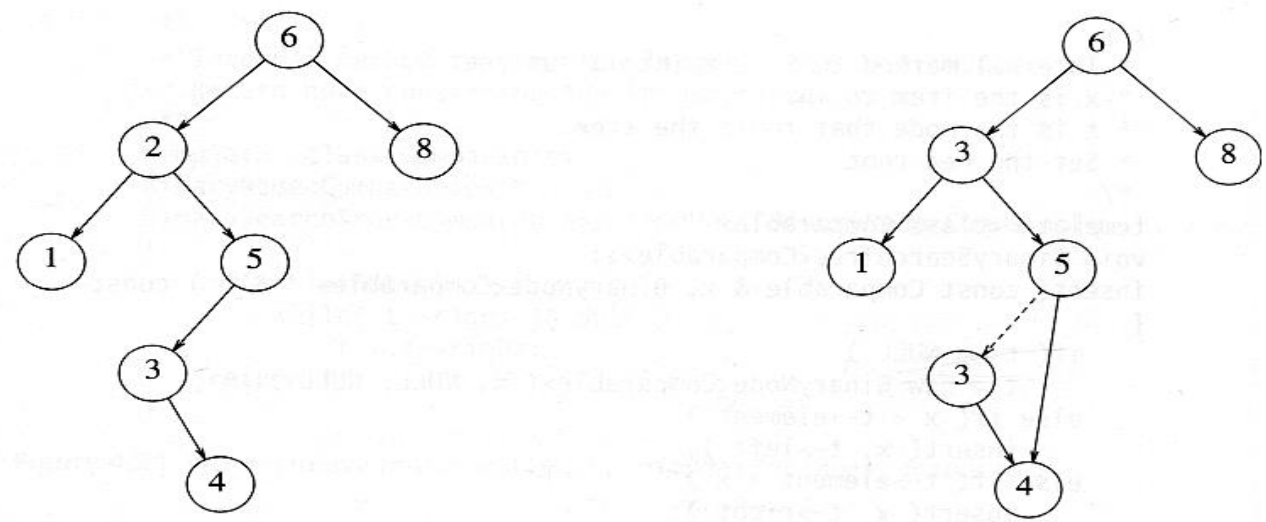
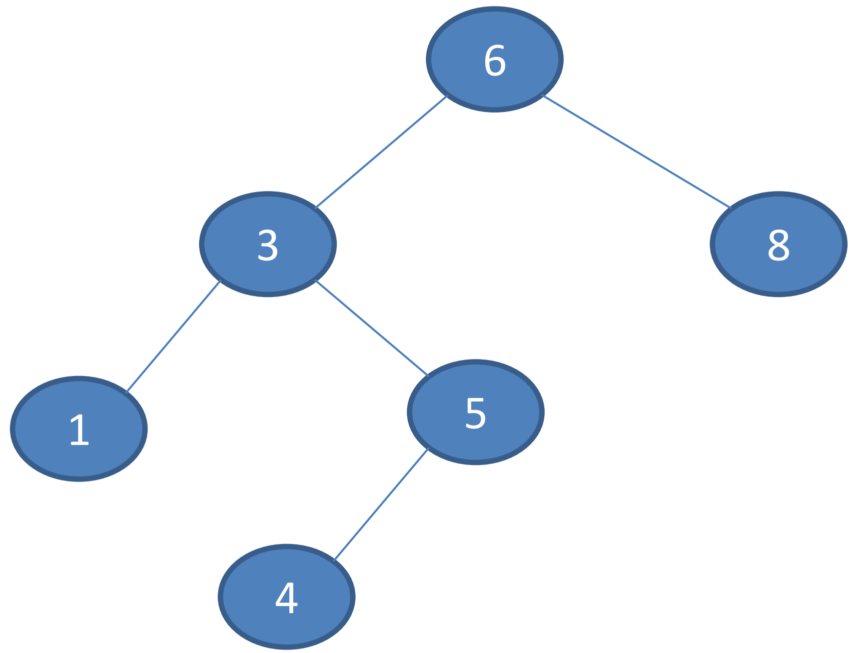


Figure 4.25 Deletion of a node (2) with two children, before and after

Contd..



After Deleting node(2)

Time complexity = O(height of the tree)

AVL Tree

The Property of AVL Tree is

The Balance Factor of each node in AVL tree may differ by at most 1 or 0. That means the acceptable values are 0, 1 or -1

The formula for **computing Balance Factor** is

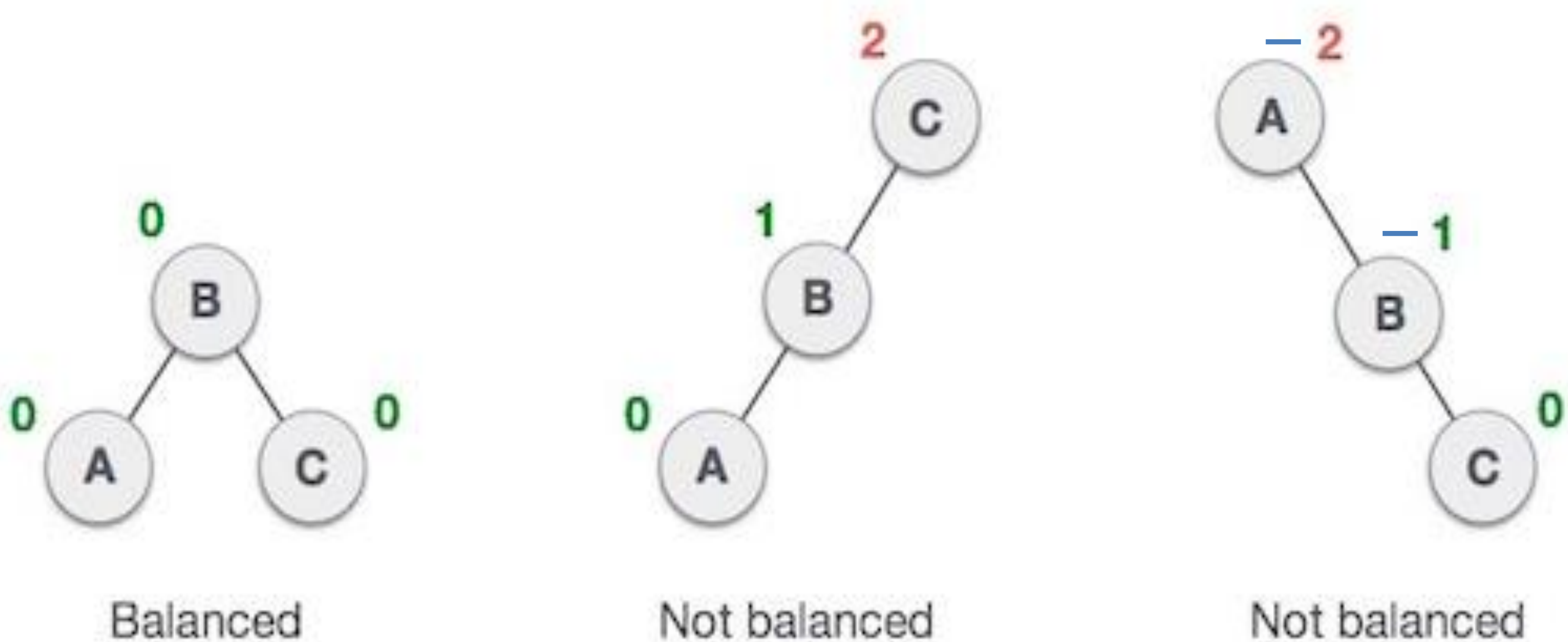
Balance Factor = Height(LST) – Height(RST)

AVL-Rotations

SCHOOL OF COMPUTER SCIENCE AND IT

- When the tree structure changes (e.g., insertion or deletion), we need to **transform the tree to restore the AVL tree property**.
- This is done using **Single rotations or Double rotations.**
- **The nodes are rearranged to have the leaf nodes at least in the same level**

Example



Rotations

- Inserting a new node will increase the height of the tree
- Deleting a node will decrease the height of the tree
- Thus, if the AVL tree property is violated at a node x, it means that the heights of left(x) and right(x) differ by exactly 2.
- Rotations will be applied to node 'x' to restore the AVL tree property.

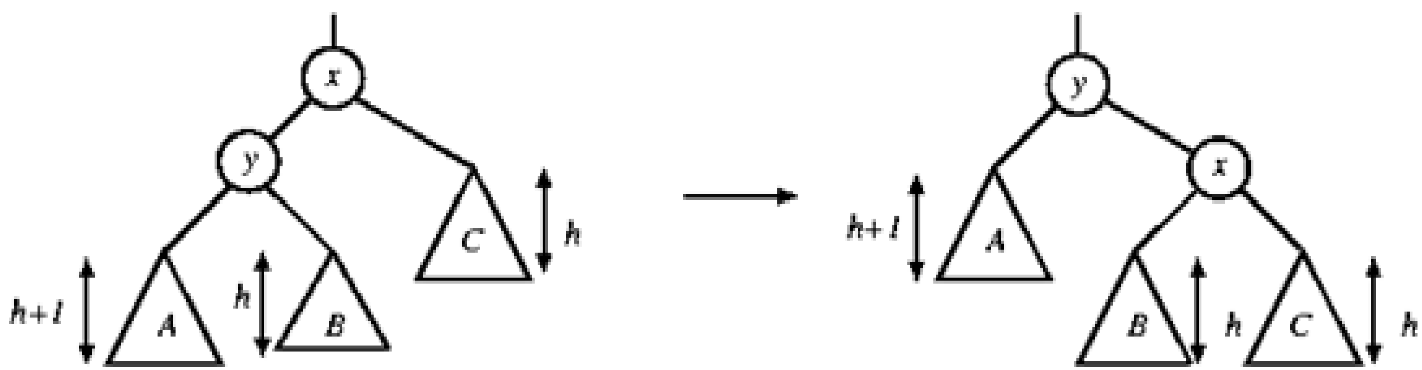
Insertion

- First, insert the new key as a new leaf just as in ordinary binary search tree
- Then trace the path from the new leaf towards the root. For each node x encountered, check if heights of left(x) and right(x) differ by at most 1.
- If yes, proceed to parent(x). If not, restructure by doing either a single rotation or a double rotation
- For insertion, once we perform a rotation at a node x, we won't need to perform any rotation at any ancestor of x.

Single Rotations

1. Single Rotation with Right (LL)

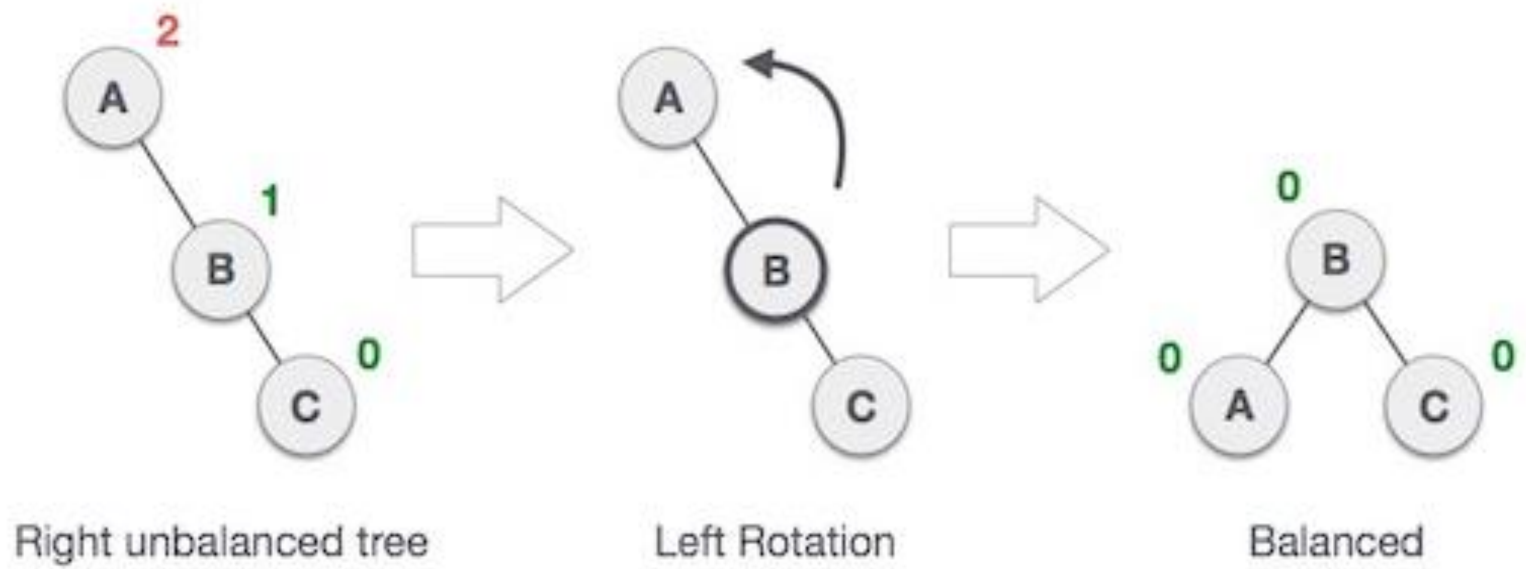
- The new key is inserted in the subtree A. i.e **new node is inserted in the left side of Left Sub Tree (LL)**
- The AVL-property is violated at x, because its balance factor is not accepted values.
- So, the **Right rotation** will be done on node x to balance the tree.



Single Left Rotation

Left Rotation

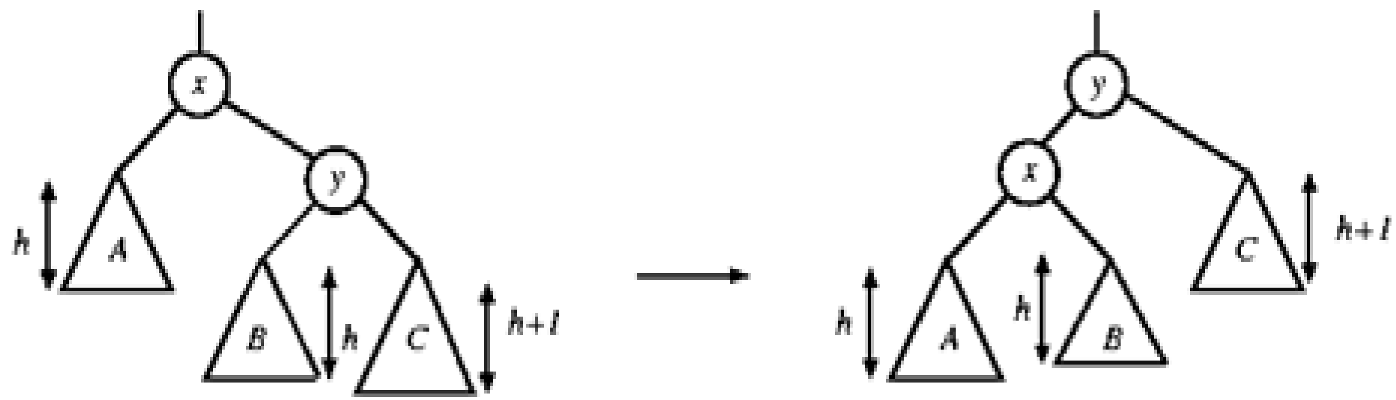
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation –



Contd..

2. Single Rotation with Left (RR)

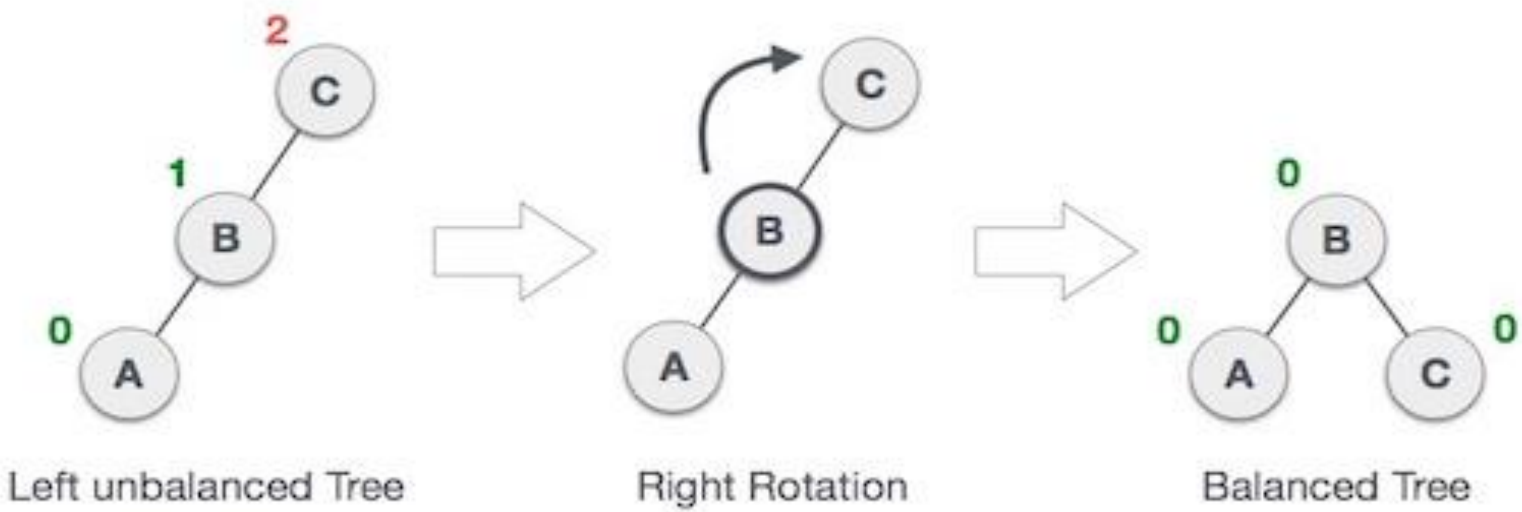
- The new key is inserted in the subtree C. i.e **new node is inserted in the right side of Right Sub Tree (RR)**
- The AVL-property is violated at x, because its balance factor is not accepted values.
- So, the **Left rotation** will be done on node x to balance the tree.



Single Right Rotation

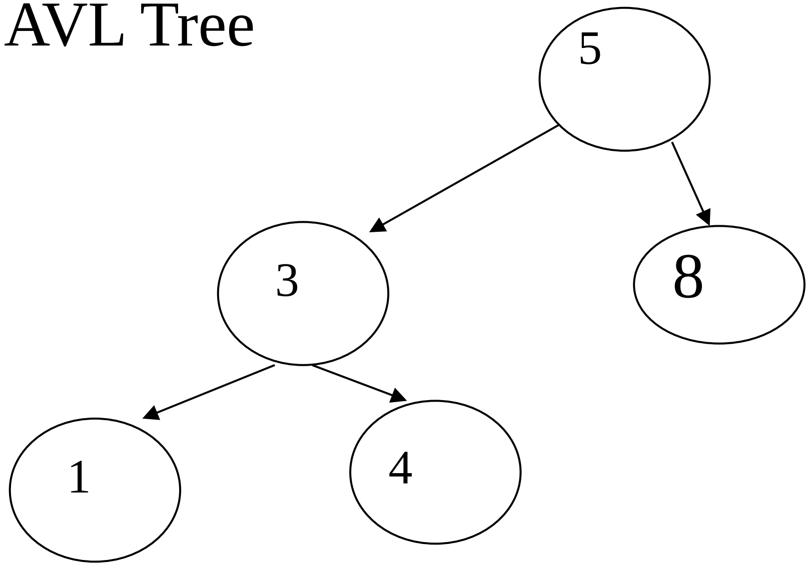
Right Rotation

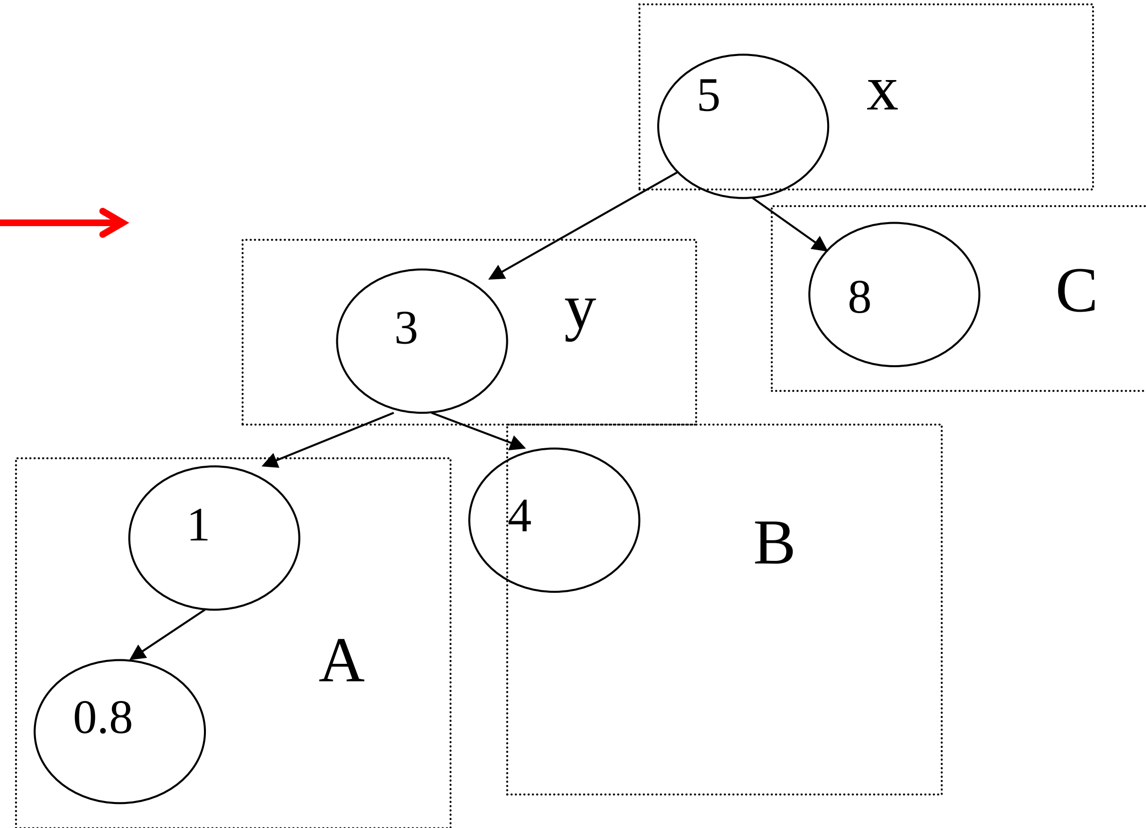
AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.



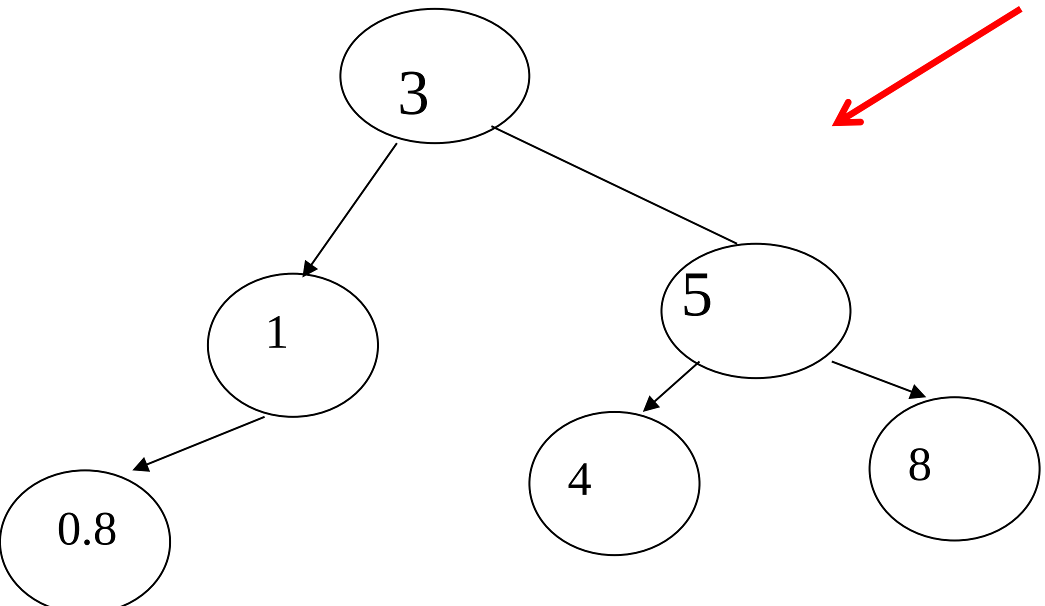
As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

AVL Tree



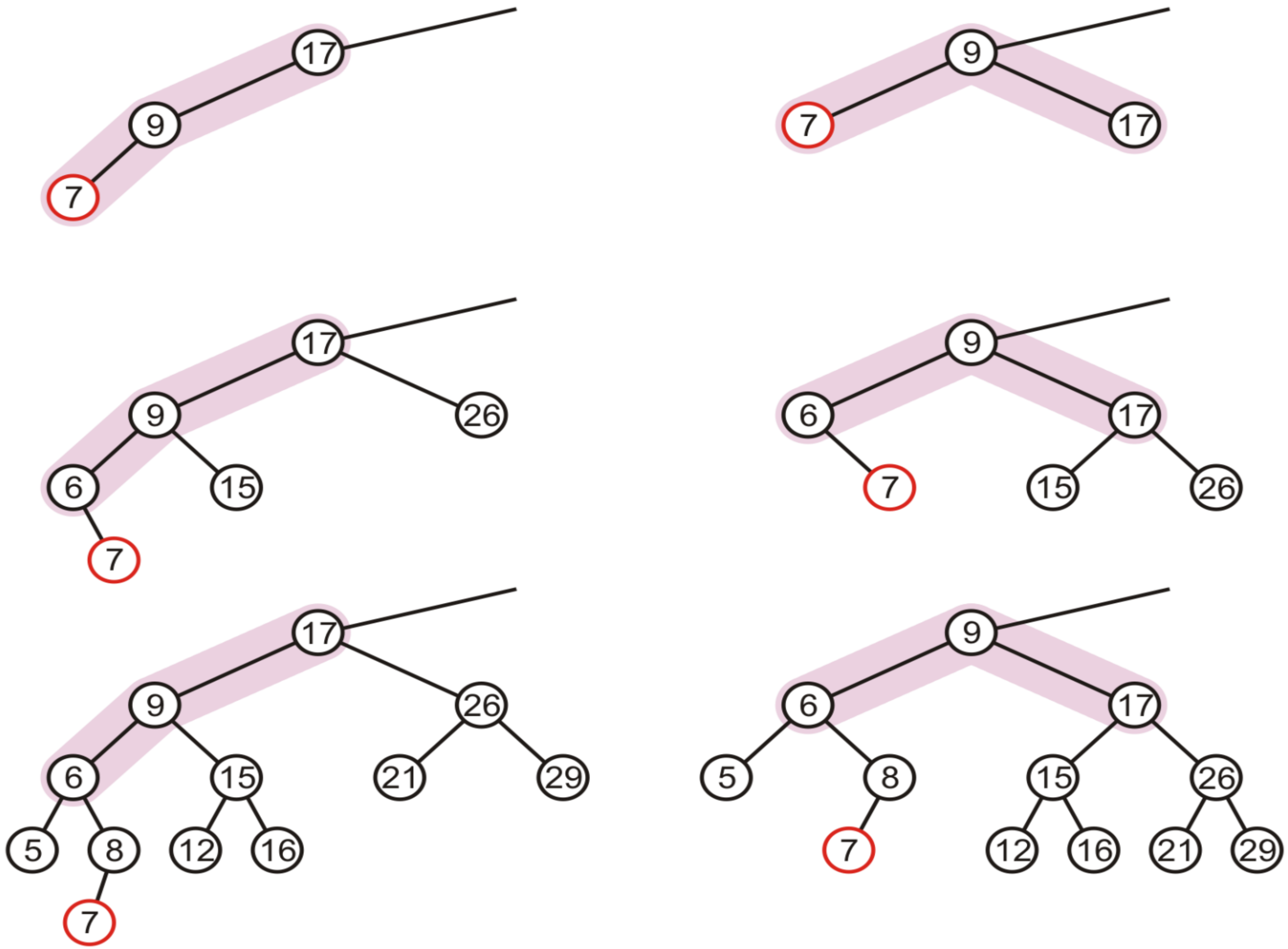


Insert 0.8



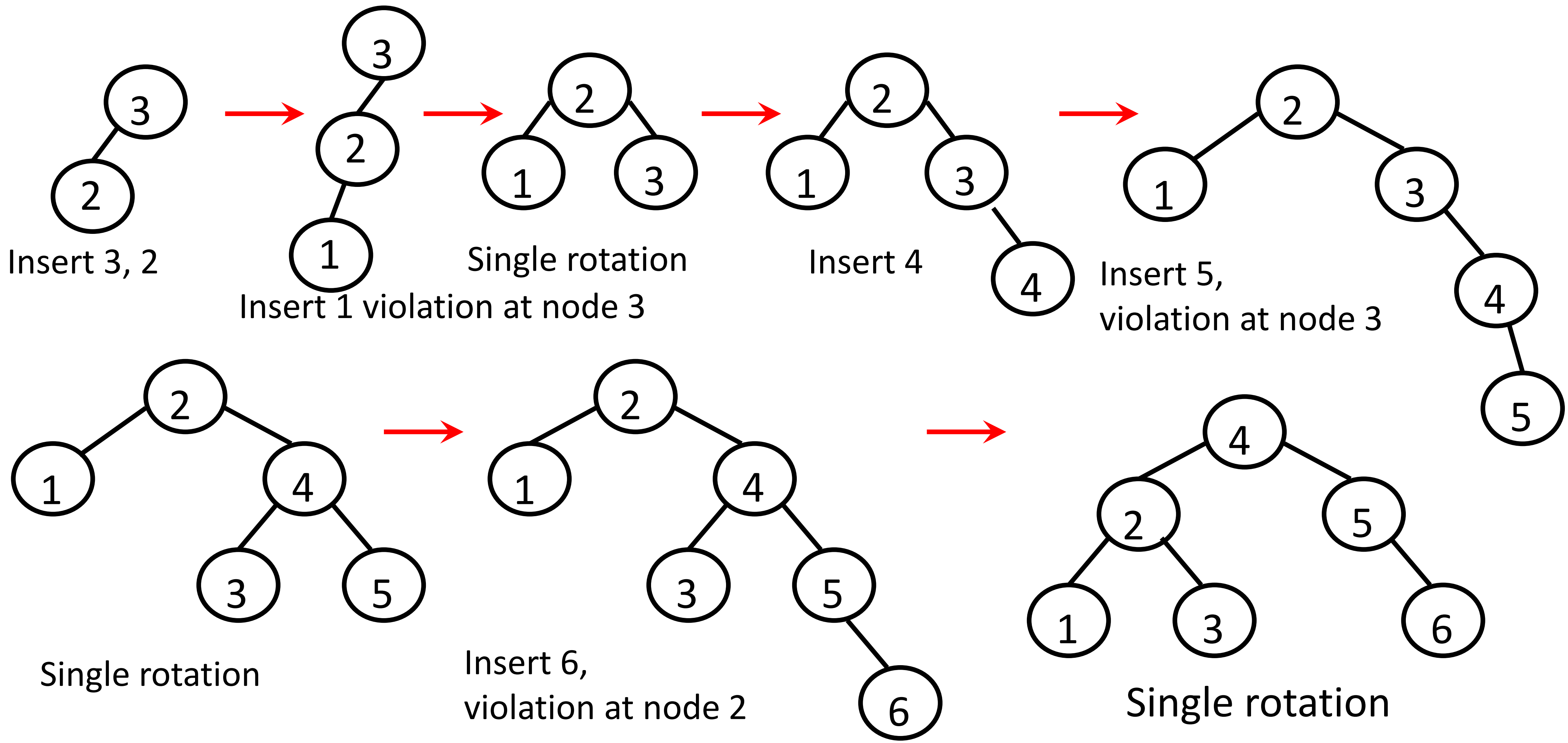
After rotation

More Examples



Ex: Single Rotation

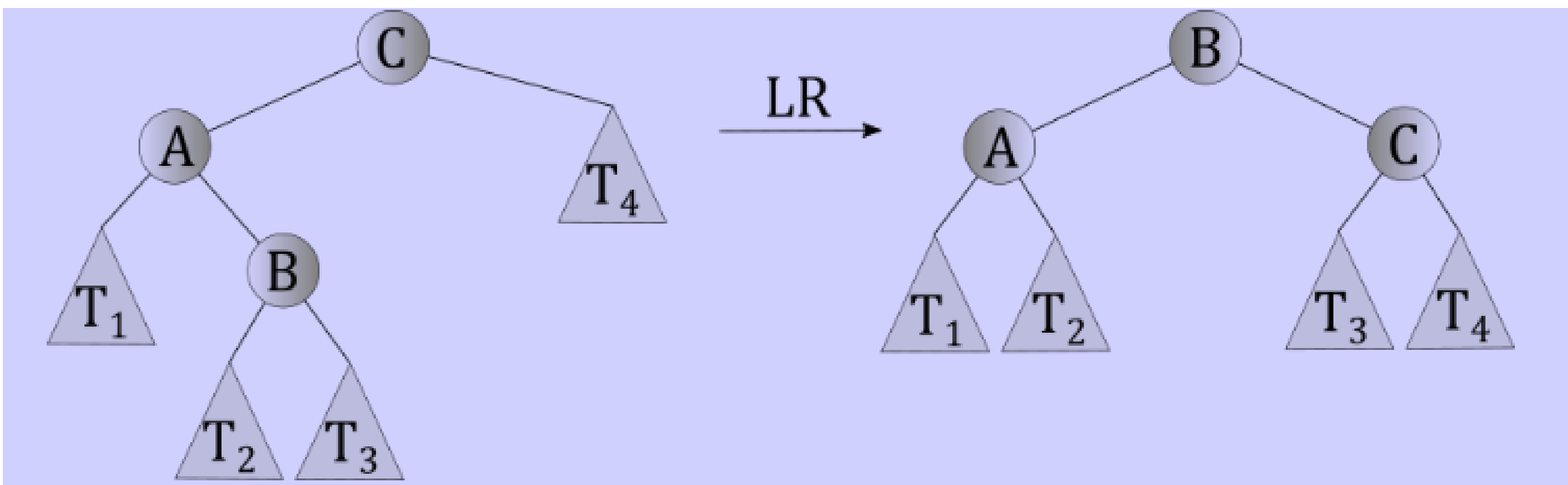
- ✓ Sequentially insert 3, 2, 1, 4, 5, 6 to an AVL Tree



Double Rotations

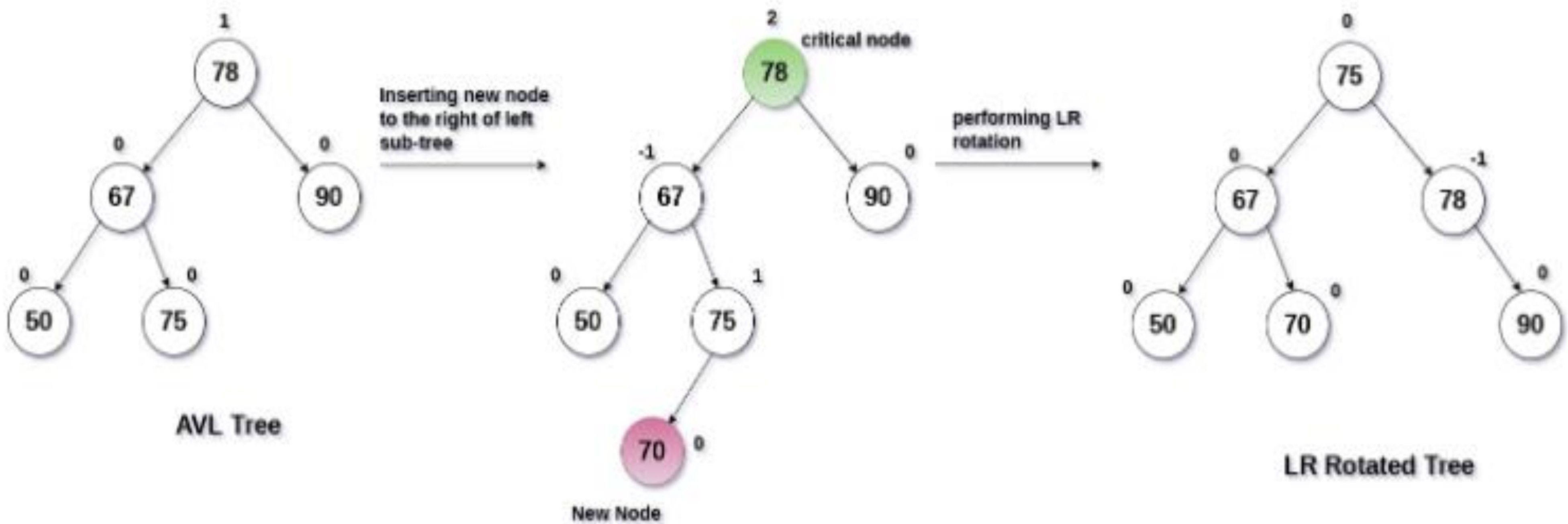
1. Left- Right Rotation (LR)

- **new key is inserted** in the **Right side of the Left Sub Tree (LR).**
- The AVL-property is violated at x.
- So, **apply the left rotation** on the node where balance factor is more.
- Then, **apply the right rotation** on the node where balance factor is more.



Double Rotations

Example: LR

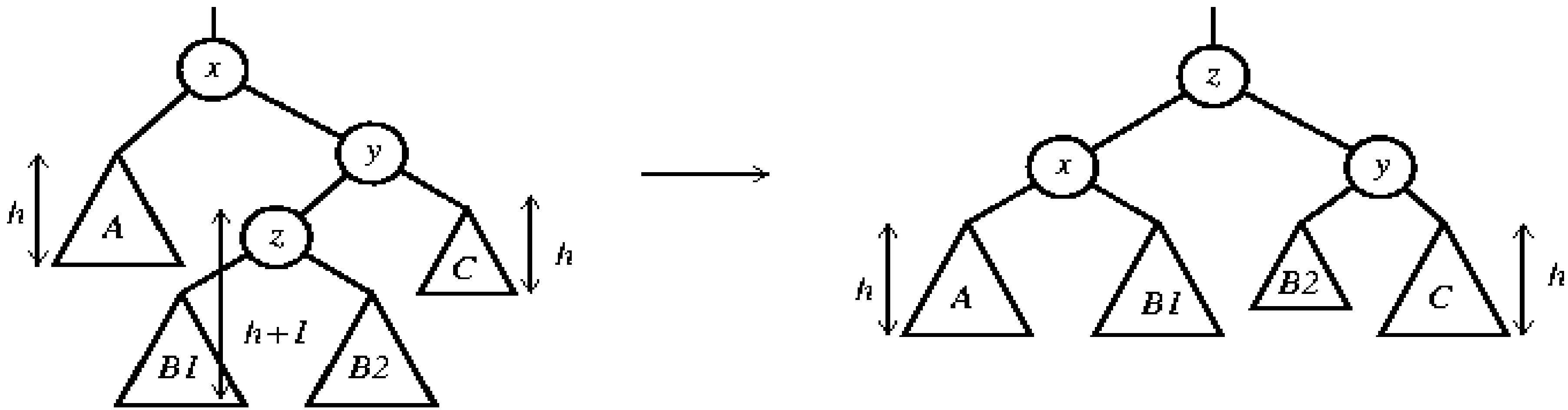


Contd..

2. Right-Left Rotation (RL)

The new key is inserted in the **left side of Right Sub Tree.**

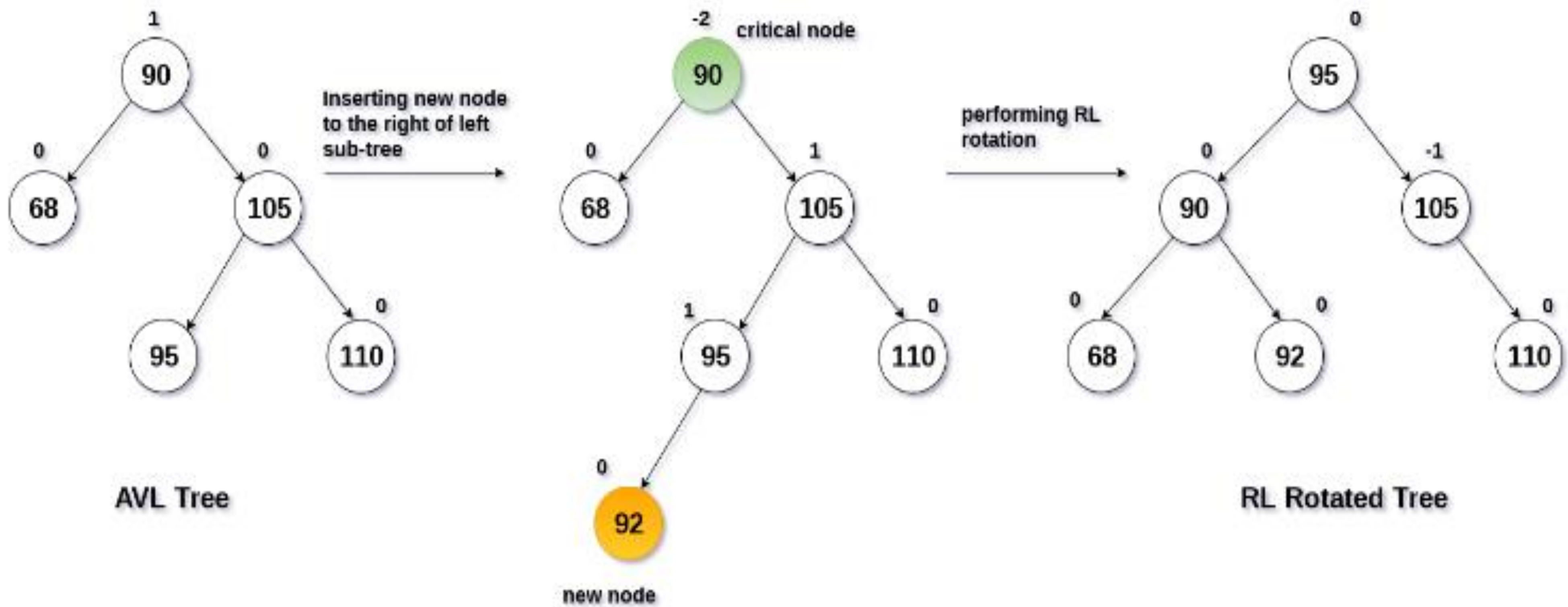
The AVL-property is violated at x.

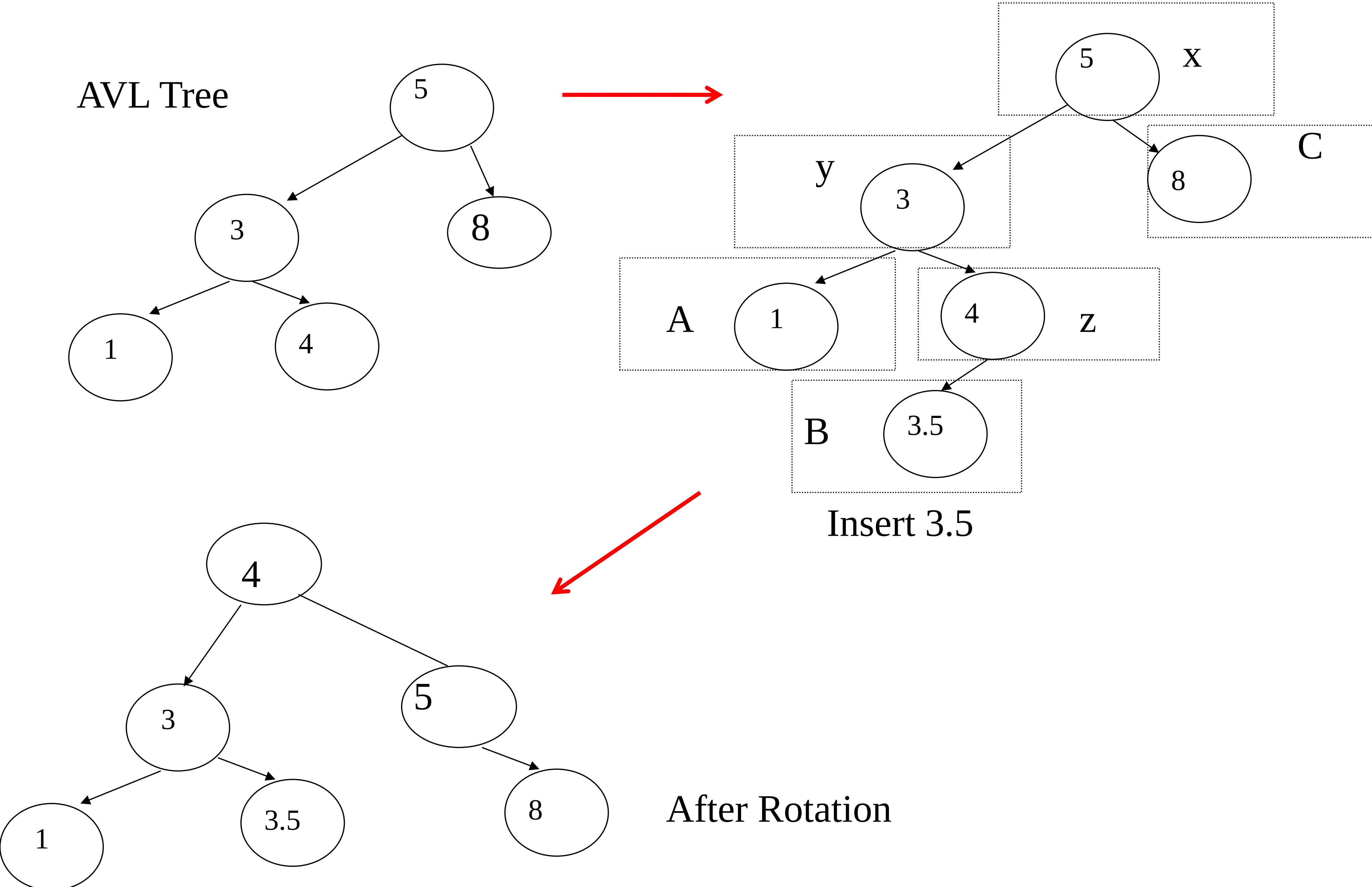


Double rotate with right child

Contd..

JGi JAIN DEEMED-TO-BE UNIVERSITY | SCHOOL OF COMPUTER SCIENCE AND IT




AVL Tree
5
3
8
1
4
x
5
y
3
C
8
A
1
4
z
B
3.5
Insert 3.5
4
3
5
1
3.5
8
After Rotation

Threaded Binary Trees

- Too many null pointers in current representation of binary trees

  n: No. of nodes

  Total links: 2n

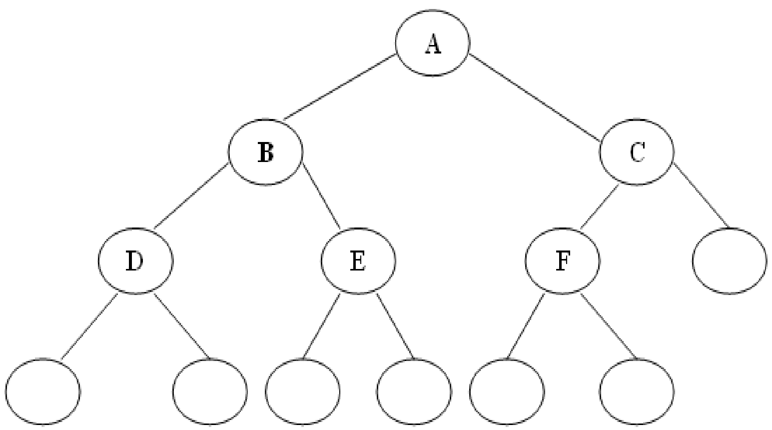
  No. of **non-null links**: n-1

  null links**:** **2n-(n-1)=n+1**

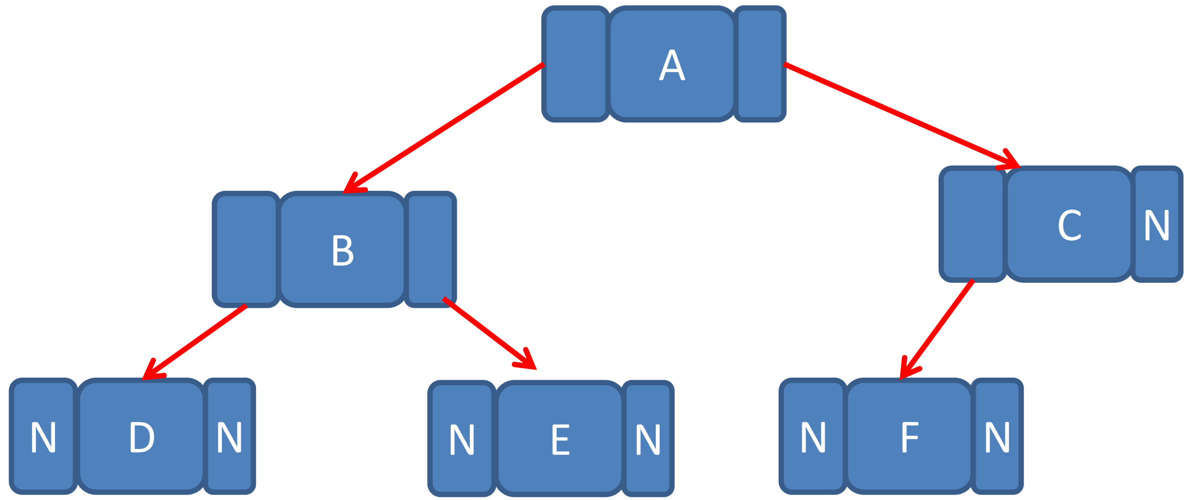
- Replace these null pointers with some useful "**threads**".

Threaded Binary Trees

- In a linked representation of a binary tree, the **number of null links (null pointers)** are actually **more than non-null pointers**.
- Consider the following binary tree:



A Binary tree with the null pointers



- In the given binary tree, **there are 7 null pointers** & **actual 5 pointers**.
- Objective: To make **effective use of these null pointers.**
- Proposed idea to **make effective use** of these null pointers.
- According to this idea we are going to **replace all the null pointers** by the **appropriate pointer values** called **threads. Such Trees are known as 'Threaded Binary Trees'.**

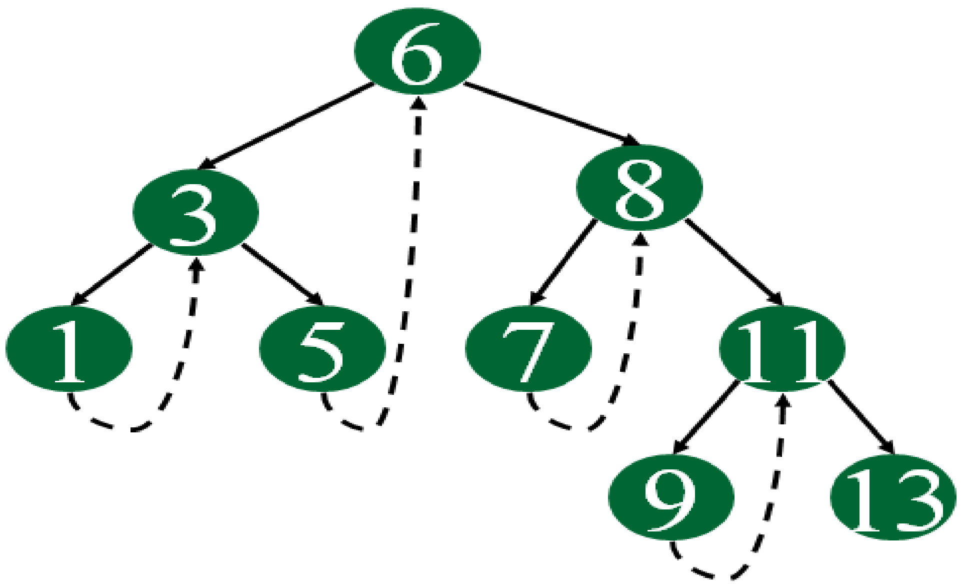
Types of Threaded BT

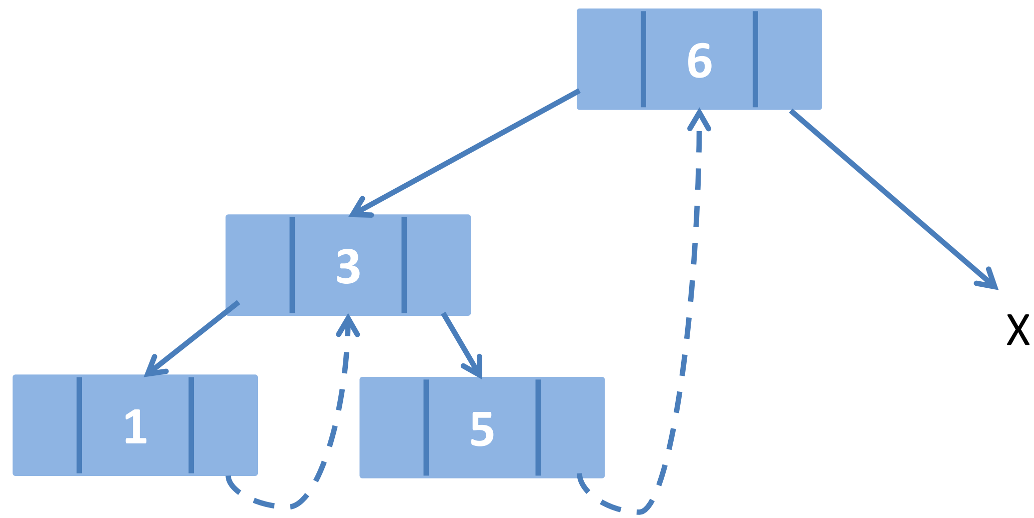
- Single Threaded Binary Tree
- Double Threaded Binary Tree

Single Threaded Binary Tree

- **Single-Threaded Binary Tree**

  - Where a **NULL Right pointers** is **made to point** to the **inorder successor** (if successor exists)





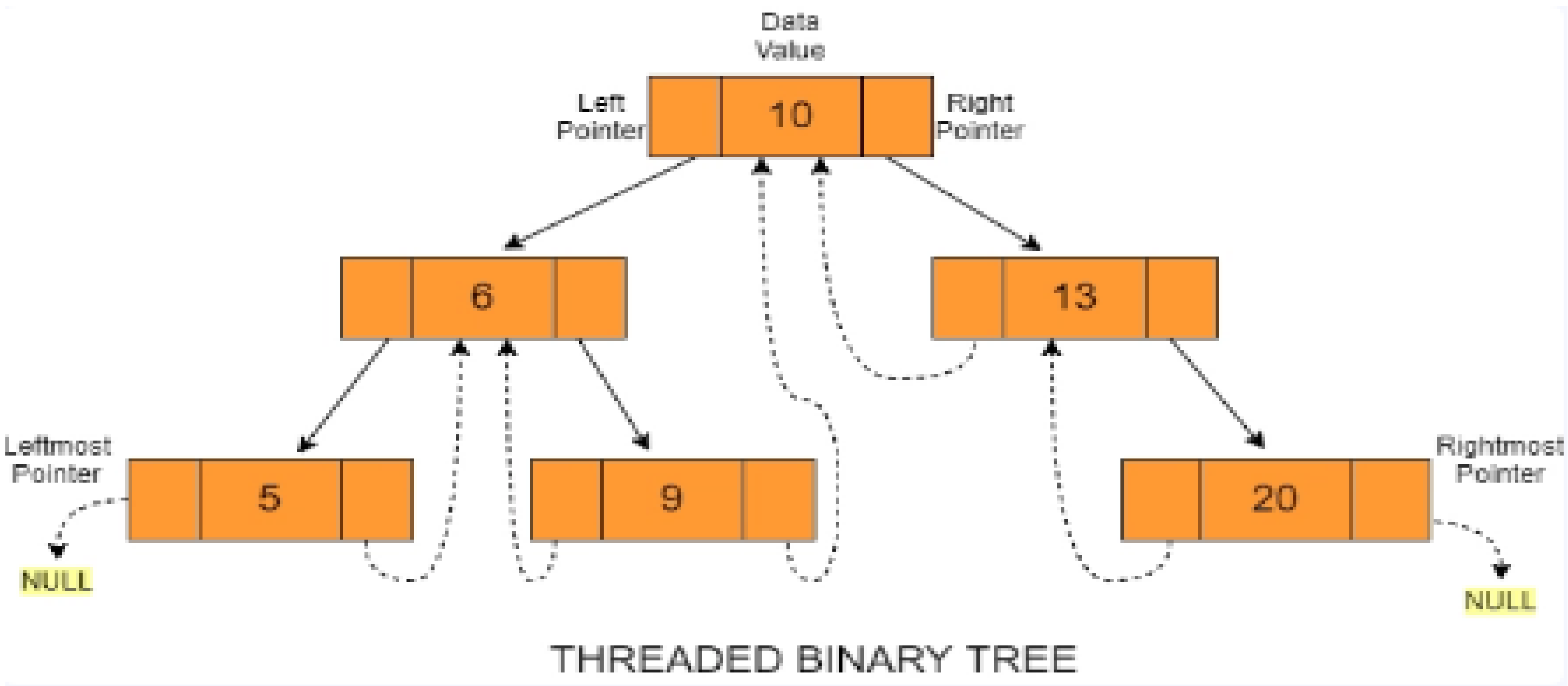
Inorder – 1, 3, 5, 6, 7, 8, 9, 11, 13

Double Threaded Binary Trees

- **Left NULL** pointer is made to **point to inorder predecessor**

- **Right NULL** pointer is made to **point to inorder successor** respectively.

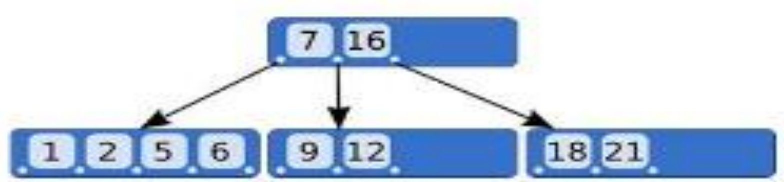
- Furthermore, the **left pointer of the first node** and the **right pointer of the last node** (in the **in-order** traversal of T) will **contain the null value**.



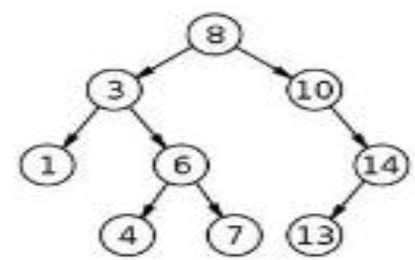
B Trees

- A **B tree** is a **sorted tree** because its **nodes are sorted** in **an inorder traversal**.
- A node in a B tree can have many children.
- If each internal node in the tree has M children, the height of the tree would be $\log_M n$ instead of $\log_2 n$.
- Thus, we can **speed up the search significantly**.

B-tree



Binary Search Tree



	Average	Worst Case
Space	O(*n*)	O(*n*)
Search	O(log *n*)	O(log *n*)
Insert	O(log *n*)	O(log *n*)
Delete	O(log *n*)	O(log *n*)

	Average	Worst Case
Space	O(*n*)	O(*n*)
Search	O(log *n*)	O(*n*)
Insert	O(log *n*)	O(*n*)
Delete	O(log *n*)	O(*n*)

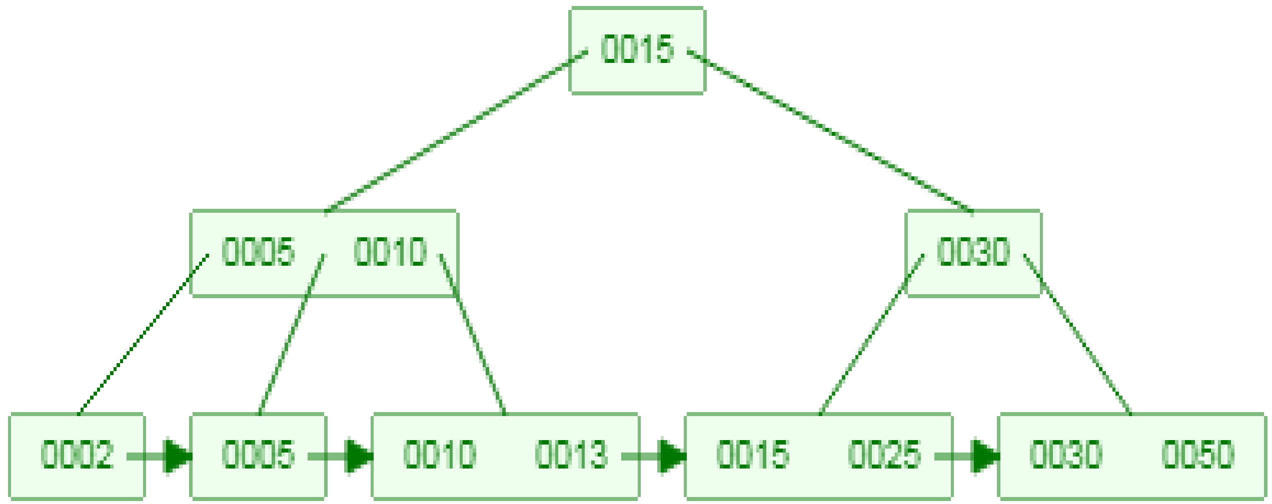
B+ Trees

- **B+ Tree is an extension of B Tree** which **allows efficient insertion, deletion and search operations.**
- In B Tree - **Keys and records both can be stored** in **the internal as well as leaf nodes**. **Whereas, in B+ tree**,
  - **records (data) can only be stored on the leaf nodes**
  - while **internal nodes can only store the key values.**
- The **leaf nodes of a B+ tree** are **linked together in the form of a singly linked lists** to make the **search queries more efficient**.

B+ Trees - Properties

- A B^+-tree of **order M ≥ 3 is an M-ary tree** with the following properties:
  - The **root** has **between 1 and M-1 keys**
  - Each **internal node has at most M children**
  - Each **internal node**, except the root, has **between ⌈M/2⌉-1 and M-1 keys**
  - The **keys at each node are ordered**
  - **Leaves can have M-1 keys**
  - The **data items** are **stored at the leaves.**
  - All **leaves are at the same depth**.

B$^+$ tree with M=3



Visualization: https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html

B+ Tree

Advantages

- Each **internal node/leaf is designed to fit into one I/O block of data**.

  - An I/O block usually can hold quite a lot of data. Hence, an **internal node can keep a lot of keys**, i.e., large M.

- This implies that the tree has **only a few levels** and **only a few disk accesses** can accomplish a **search, insertion, or deletion**.
- **B+-tree is a popular structure** used **in commercial databases**.
- To **further speed up** the search, the **first one or two levels of the B+-tree are usually kept in main memory**.

Disadvantages of B and B+ Trees:

- The disadvantage of B^+-tree is that **most nodes will have less than M-1 keys** most of the time. This could **lead to severe space wastage**.
- **B-tree** refers to the variant where the **actual records are kept at internal nodes** as well as **the leaves**. Such a **scheme is not practical**.
- **Keeping actual records at the internal nodes** will **limit the number of keys** stored there, and thus **increasing the number of tree levels**.

Heaps

- First of all, a heap is a **kind of** binary **tree** that offers **both insertion and deletion in O(log2n)** time.
- Heaps are **largely about priority queues**.

  - They are an **alternative data structure** to implement **priority queues** (we had arrays, linked lists…)

  - the **advantages and disadvantages** of queues implemented as arrays

    ->Insertions / deletions? **O(n) … O(1)!**

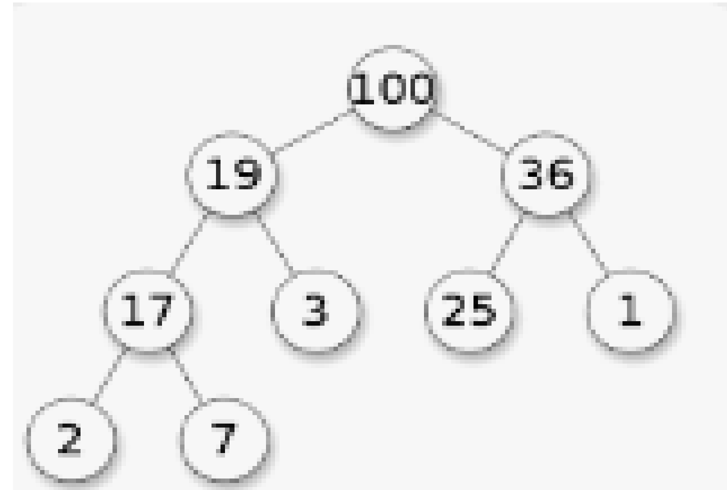
- **Priority queues** are **critical to many real-world applications**.

Introduction

SCHOOL OF COMPUTER SCIENCE AND IT

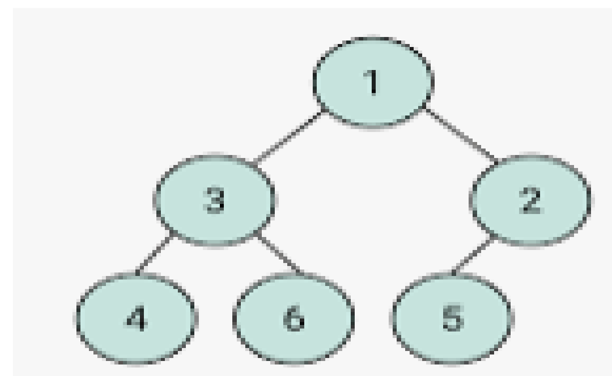
Definition (MAX HEAP):

A heap is a **complete binary tree** that either is empty or It's **root contains a value greater than or equal to the value in each of its children,** and **has heaps as its subtrees**.



Definition (Min HEAP):

A heap is a **complete binary tree** that either is empty or It's **root contains a value lesser than or equal to the value in each of its children**, and **has heaps as its subtrees**.



Properties of Heaps

- **Structure Property**

  A **heap is a binary tree** that is **completely filled**, with the **possible exception of the bottom level,**

  \- which **is filled from left to right**

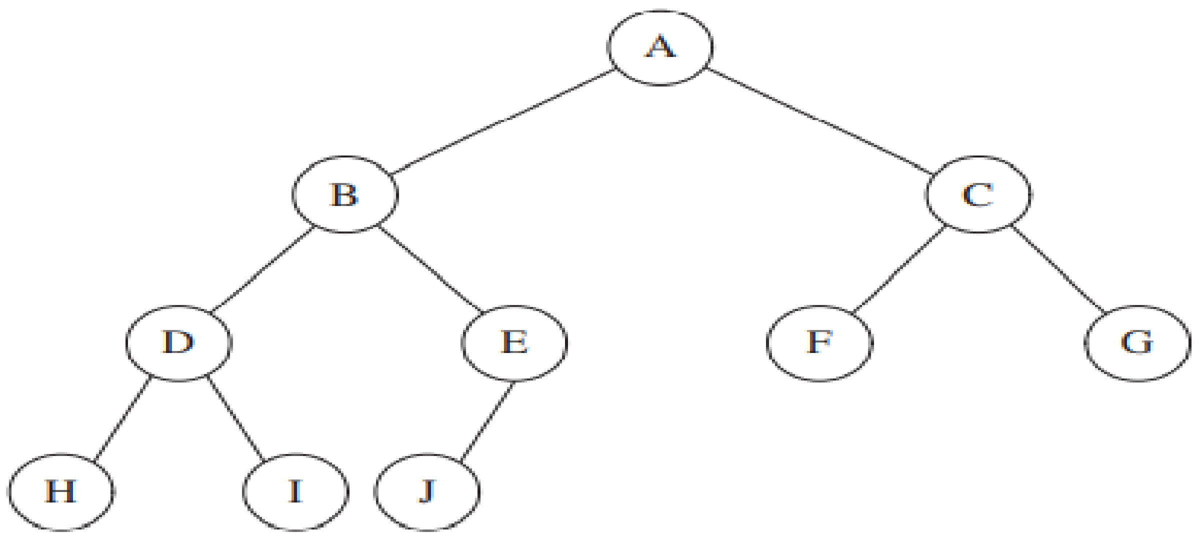


Figure 6.2 A complete binary tree

	A	B	C	D	E	F	G	H	I	J			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

- **<u>Heap Order Property</u>**

Each node in a heap **satisfies the 'heap condition,'** which states that **<u>every node's key is smaller than or equal to the keys of its children</u> <u>(min heap)</u>**.

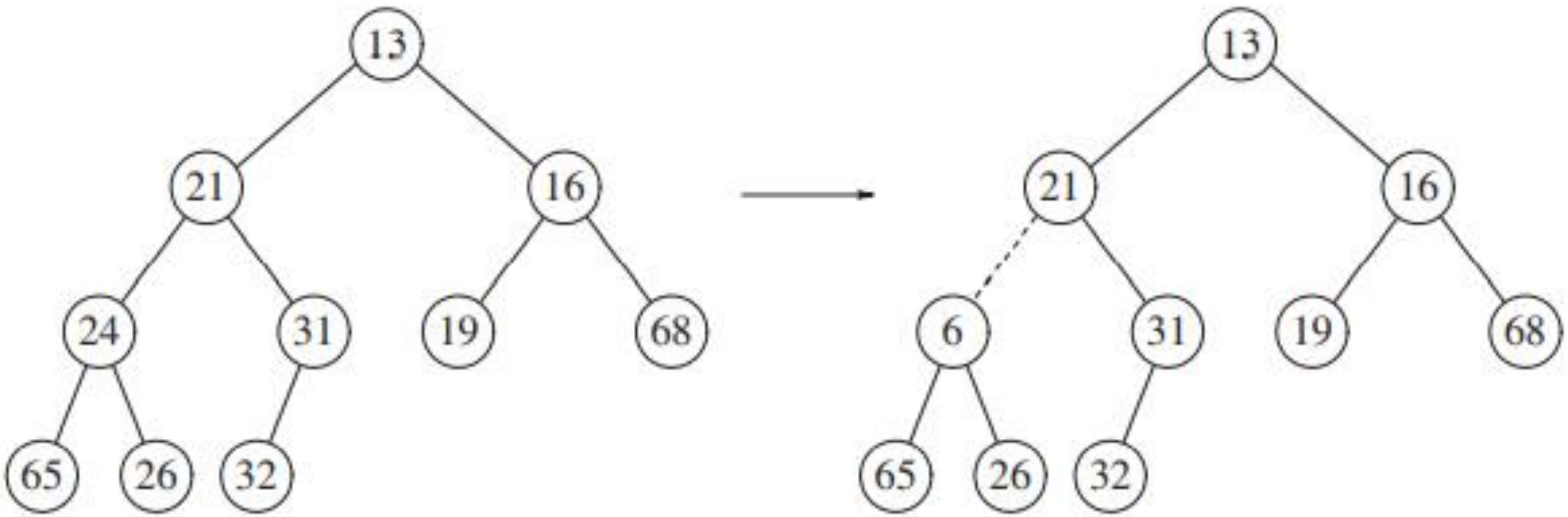
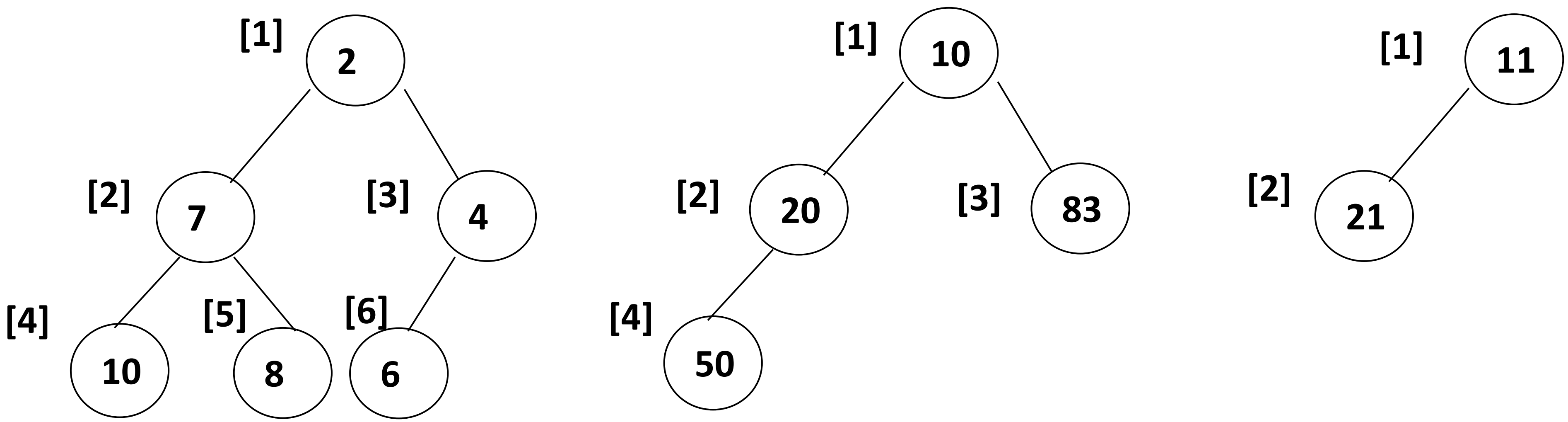


Figure 6.5 Two complete trees (only the left tree is a heap)

Min heaps



Property: The root of **min heap** contains the **smallest**.

Heaps - Insertion

- ✓Add **new item to the end**
- ✓Now **fix the heap** (float the new item up to the correct location)
- ✓**Move the element** to the **correct location** (**trickle up**)

- Start at the **bottom** (first open position) via code:

```
heapArray[n] = newNode;
n++;
```

***Inserting at bottom** will likely destroy the heap condition.

This will happen when the new node is not satisfying the Heap order property than its parent.

***Trickle upwards** until node is **satisying the heap order property**

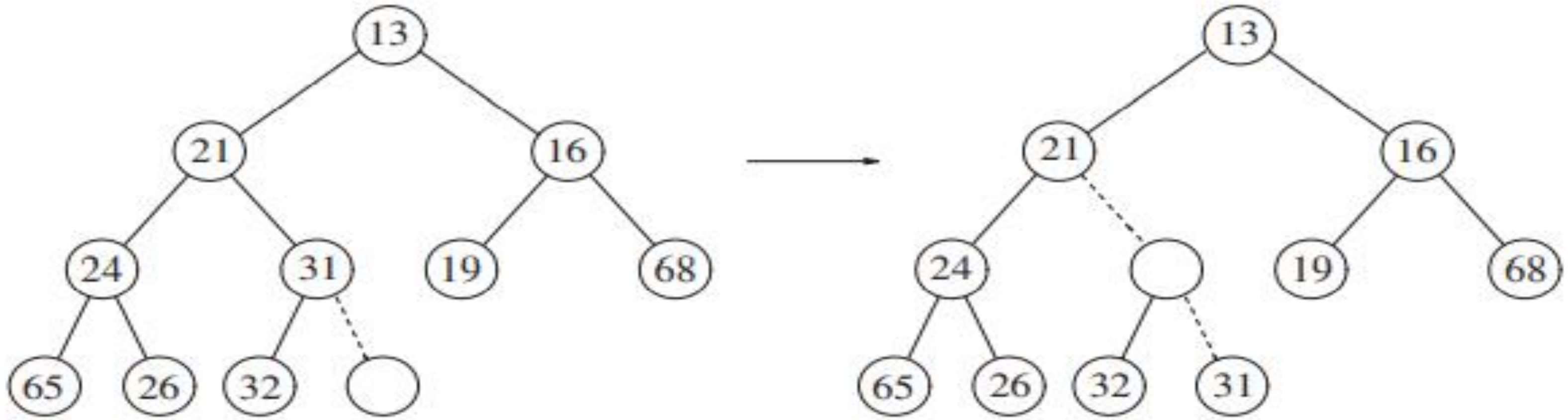


Figure 6.6 Attempt to insert 14: creating the hole, and bubbling the hole up

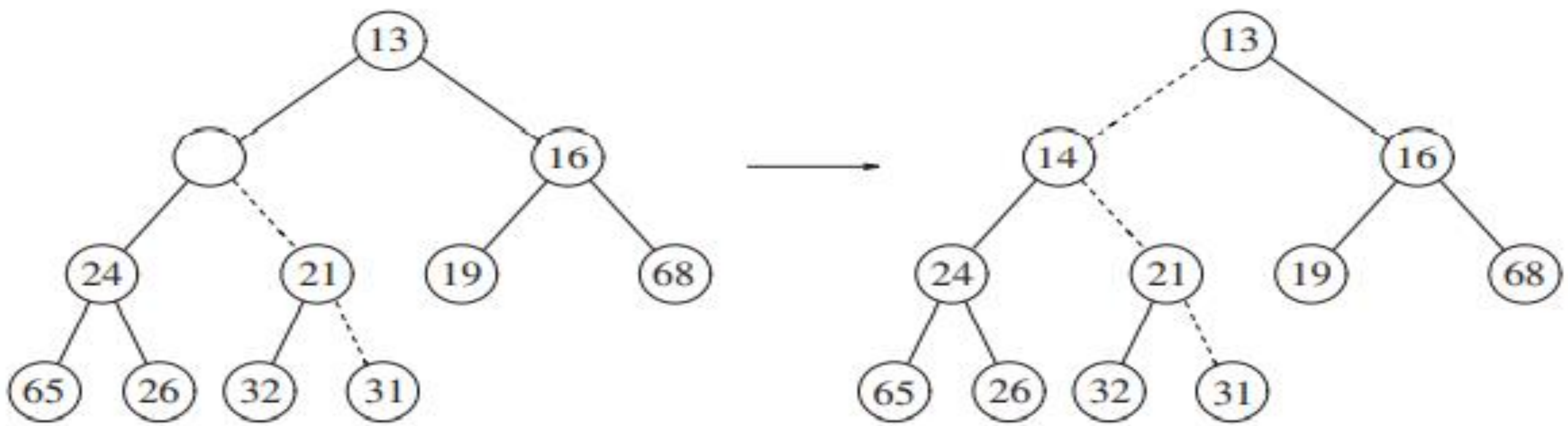


Figure 6.7 The remaining two steps to insert 14 in previous heap

Heaps - Removal

Removal 'Min' Node

- **Remove the Root Key**

  - When we remove from a heap, we always remove the node with the **root key**.

  – Hence, removal is quite easy and has **index 0 of the heap array**.

  – maxNode = heapArray[0]

- **Move 'last node' to root**.

  – Start by **moving** the **'last node' into the root**.

  – The 'last' node is the **recently inserted in the heap**.

  – This also corresponds to the **last filled cell** in the array (ahead).

- **Trickle-down or Percolate Down**:

  – Then **trickle this last node down** until **heap order property gets satisfied**.

Heaps - Removal

deleteMin()

✓The root element is **removed simply from the root**

✓**Last element** will be **moved to the root**

✓Verify heap order property. If it is violated **percolate down** the **element to correct position**

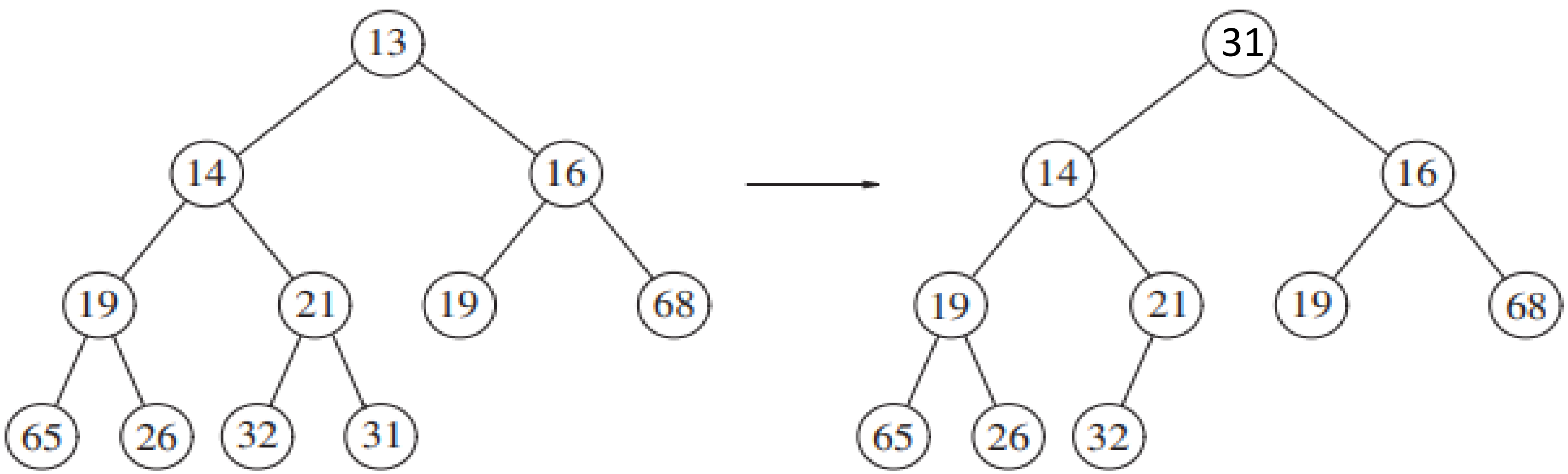


Figure 6.9 Creation of the hole at the root

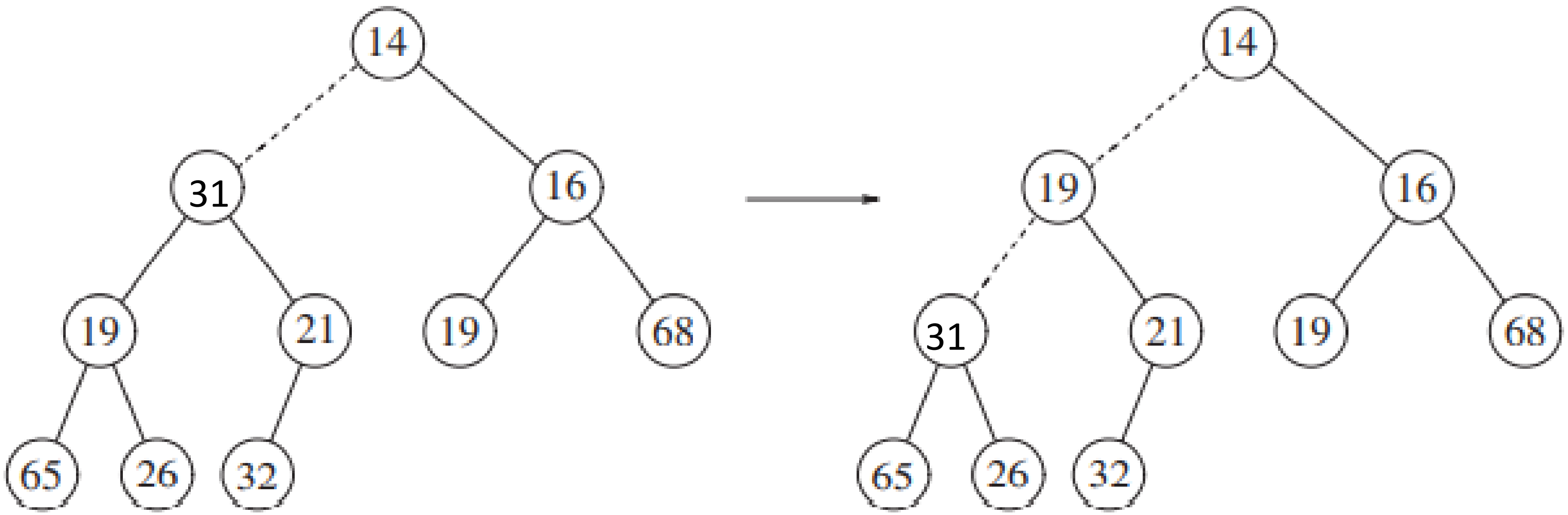


Figure 6.10 Next two steps in deleteMin

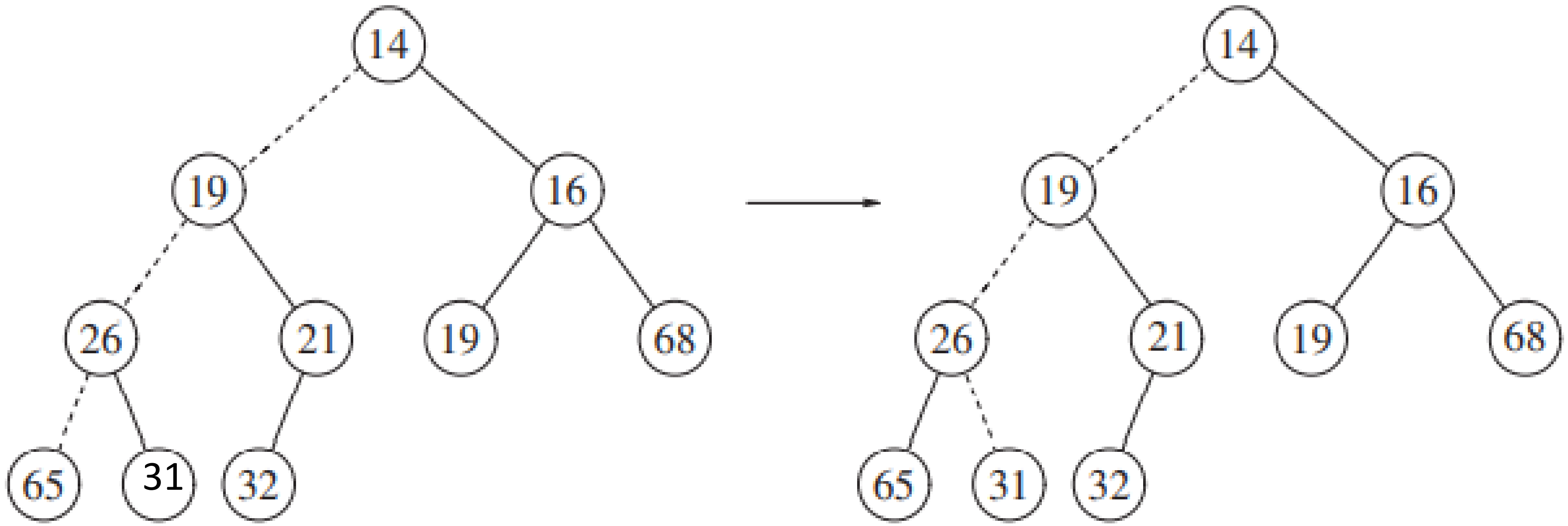


Figure 6.11 Last two steps in `deleteMin`

Applications of Heaps

- Used to obtain **improved running times** for **several network optimization algorithms.**
- Can be used to assist in **dynamically-allocating memory partitions.**
- A **heapsor**t is considered to be **one of the best sorting methods** being in-place with no quadratic worst-case scenarios.
- **Finding the min, max, both the min and max, median**, or even the **k-th largest element can be done in linear time** using heaps and etc.

Hashing

It is the process of mapping a key value to a position in a table

- ✓ Hashing is a technique used for performing insertions, deletions and searching in constant average time (i.e. O(1))
- ✓ Hash function is determining position of key in the array.
- ✓ Hashing is widely useful technique for implementing Dictionaries ADT.
- ✓ Hash table ADT is an alternative solution with O(1) expected query time and O(n + N) space, where N is the size of the table.

Hashing Functions

There are many hash functions that use numeric or alphanumeric keys.

Different hash functions:

1. **Division Method -** The hash function **divides the value k by M** and then **uses the remainder obtained**.

   h(K) = k mod M

   Here,

   ***k** is the key value, and*

   ***M** is the size of the hash table.*

It is best suited that **M is a prime number** as that can make sure the **keys are more uniformly distributed.**

2. Mid Square Method

It involves two steps to compute the hash value-

1. Square the value of the **key k i.e.** k^2

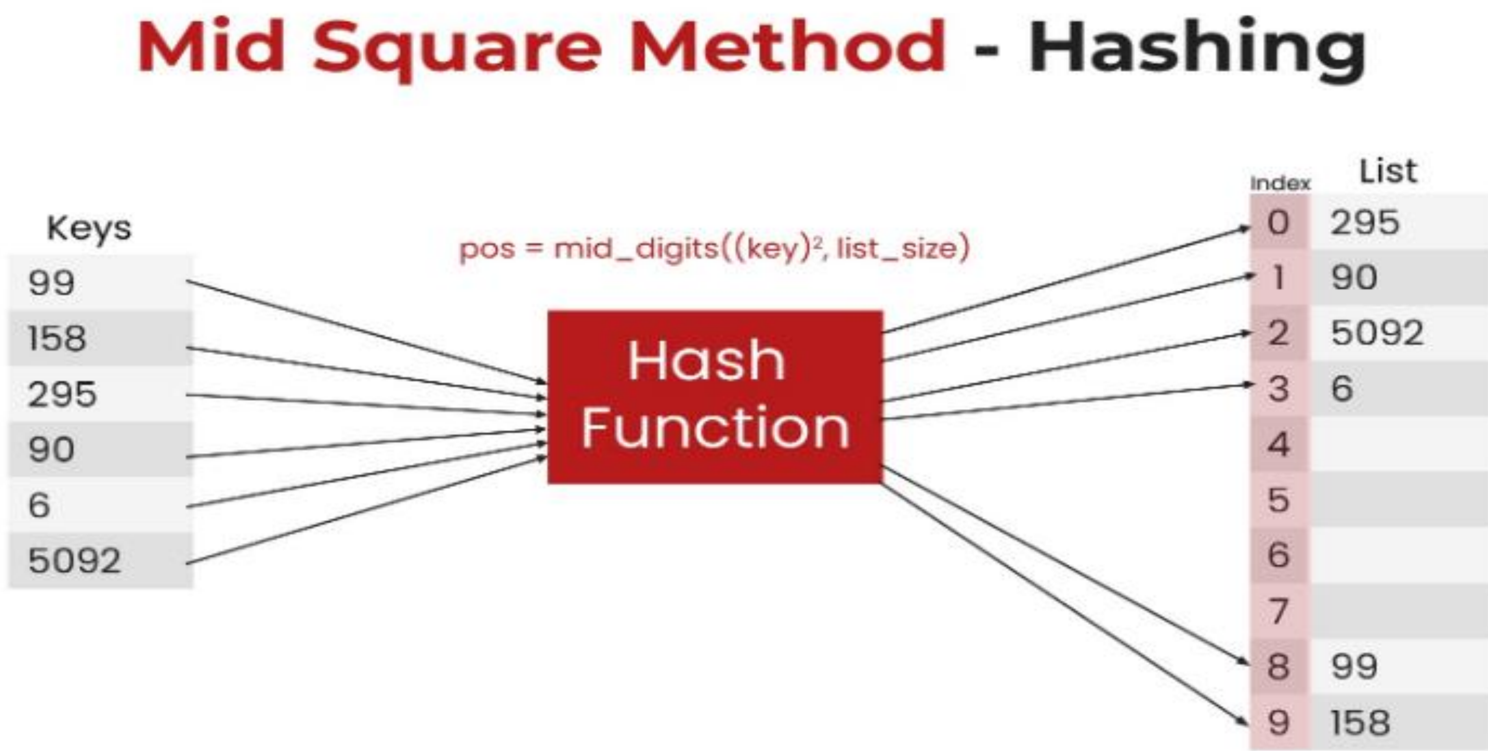
2. **Extract the middle r** digits as the hash value.

h(K) = h(k x k)

Here,

***k** is the key value.*

The value of **r** can be decided based on the size of the table.


Mid Square Method - Hashing
Keys
99
158
295
90
6
5092
pos = mid_digits((key)², list_size)
Hash Function
Index
List
0 295
1 90
2 5092
3 6
4
5
6
7
8 99
9 158

SCHOOL OF COMPUTER SCIENCE AND IT

3. Folding Method.

This method involves two steps:

1. Divide the key-value **k** into a number of parts i.e. **k1, k2, k3,….,kn**, where **each part has the same number of digits** **except for the last part** that can have lesser digits than the other parts.

2. Add the individual parts. The hash value is obtained by ignoring the last carry if any.

k = k1, k2, k3, k4, ….., kn

s = k1+ k2 + k3 + k4 +….+ kn

h(K)= s

Here,

s** is obtained by adding the parts of the key **k

Ex: Keys are 2103, 7148, 12345, Table size 100 (0 to 99)

Hash table index : **00 to 99** (2-digit hash table)
So, divide the Key into **k numbers of two digits**

K	2103	7148	12345
$k_1\ k_2\ k_3$	21, 03	71, 48	12, 34, 5
H(k) $= k_1 + k_2 + k_3$	H(2103) = 21+03 = 24	H(7148) = 71+48 = 19	H(12345) = 12+34+5 = 51

H(7148) = 71 + 48 = 119, here we will eliminate the leading carry (i.e., 1). So H(7148) = 71 + 48 = 19

Key 2103 is placed -> cell 24, 7148 -> cell 19, 12345 -> 51

4. Multiplication Method

Steps to follow -

- **Pick up a constant value A** (where $0 < A < 1$)
- **Multiply A with the key value**
- **Take the fractional part of kA**
- **Take the result of the previous step** and **multiply it by the size of the hash table**, M.

Formula - h(K) = floor (M (kA mod 1))

(Where, **M = size of the hash table**, **k = key value** and **A = constant value**)

Ex:

Suppose k=6, A=0.3, m=32

(1) $k \times A = 1.8$

(2) fractional part: $1.8 - \lfloor 1.8 \rfloor = 0.8$

(3) $m \times 0.8 = 32 \times 0.8 = 25.6$

(4) $\lfloor 25.6 \rfloor = 25$

h(6)=25

- **Problem: *COLLISION***
  - **two keys may hash to the same slot**
  - can we ensure that any two distinct keys get different cells?
    - No, if |U|>m, where m is the size of the hash table

➢ Design a **good hash function**

- that is **fast to compute** and
- can **minimize the number of collisions**

➢ Design a **method to resolve the collisions** when they occur

Hashing

Collision: Collision is the condition resulting when two or more keys produce the same hash location

✓ To avoid collision use **Collision Resolution Techniques**

There are two broad ways of collision resolution:

1. **Separate Chaining:** An **array of linked list implementation.**
2. **Open Addressing**: **Array-based implementation**.

(i) Linear probing (linear search)

(ii) Quadratic probing (nonlinear search)

(iii) Double hashing (uses two hash functions)

How Hashing works?

- ✓ **Ex: 23 ,24, 25, 26, 27, 28, 29, 30, 31, 32, 33 ,60**

Take n % 10 is a Hash Function

BUCKET: 0 -->30 -->60 Collision
BUCKET: 1 -->31
BUCKET: 2 -->32
BUCKET: 3 -->23 --> 33 Collision
BUCKET: 4 -->24
BUCKET: 5 -->25
BUCKET: 6 -->26
BUCKET: 7 -->27
BUCKET: 8 -->28
BUCKET: 9 →29

- ✓ To avoid Collision use Collision resolution Techniques

1. Separate Chaining

- ✓ To deal collision, set up an **array of links (a table),** indexed by the **keys to lists of items with the same key**
- ✓ **All keys that map to the same hash value** are **kept in a list**
- ✓ The **hash table is implemented as an array of linked lists**, inserting an item that hashes at index **is simply insertion into the linked list at position in the table**.
- ✓ **Store all elements that hash to the same slot in a linked list**, **store a pointer** to **the head of the linked list** in the hash table slot.

Separate Chaining

SCHOOL OF COMPUTER SCIENCE AND IT

- **To insert a key K**
  - **Compute h(K)** to determine **which list to traverse**
  - If **T[h(K)] contains a null pointer**, initialize this entry to point to a linked list that **contains K alone**.
  - If **T[h(K)] is a non-empty list**, we add **K at the beginning of this list or end of the list**.
- **To delete a key K**
  - compute h(K), then search for K within the list at T[h(K)].

  Delete K if it is found.

Separate Chaining

Example: Load the keys **23, 13, 21, 14, 7, 8, and 15** , in this order, in a hash table of size **7** using separate chaining with the hash function: **h(key) = key % 7**

$h(23) = 23 \% 7 = 2$

$h(13) = 13 \% 7 = 6$

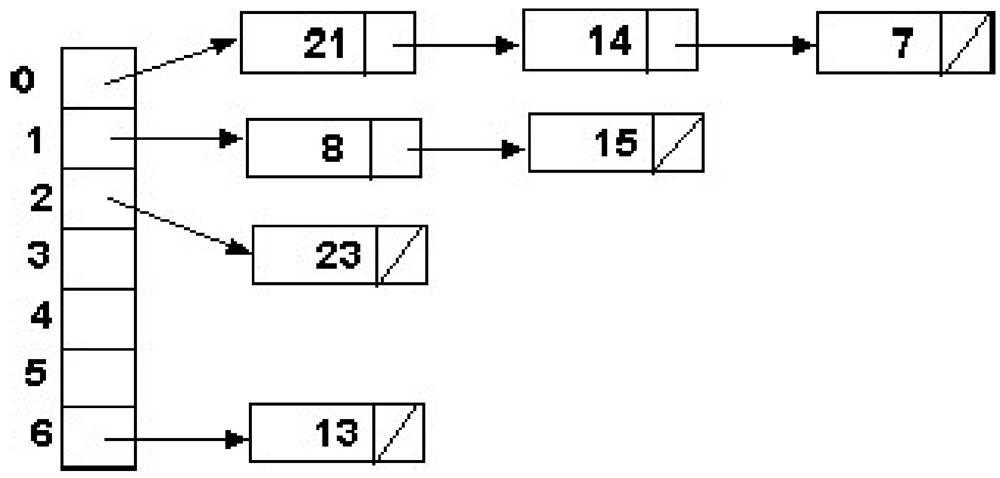
$h(21) = 21 \% 7 = 0$

$h(14) = 14 \% 7 = 0$ collision

$h(7) = 7 \% 7 = 0$ collision

$h(8) = 8 \% 7 = 1$

$h(15) = 15 \% 7 = 1$ collision



Separate Chaining

- Assume that we will be storing *n* keys. Then we should make *m* the next larger prime number. If the hash function works well, the number of keys in each linked list will be a small constant.
- Therefore, we expect that each search, insertion, and deletion can be done in constant time.
- **Disadvantage:**

  Memory allocation in linked list manipulation will slow down the program.
- **Advantage**: deletion is easy.

2. Open Addressing

✓ Open Addressing:

- (i) Linear probing (linear search)
- (ii) Quadratic probing (nonlinear search)
- (iii) Double hashing (uses two hash functions)

i. Linear Probing

- ✓ Linear probing is a strategy for resolving collisions, by placing the new key into the closest following empty cell.
- f(i) =i
  - cells are probed sequentially (with wraparound)
  - **$h_i(K)$ = (hash(K) + i) mod m**
- Insertion:
  - Let K be the new key to be inserted. We compute hash(K)
  - For **i = 1 to m-1**
    - compute L = **(hash(K) + i) mod m**
    - **T[L] is empty, then we put K there and stop**.
  - **If we cannot find an empty entry to put K, it means that the table is full and we should report an error**.

Linear Probing

✓ **Ex:** Insert items with keys: 89, 18, 49, 58, 9 into an empty hash table. Table size is 10. Hash function is hash(x) = x mod 10.

Hash Function:

89%10=9
18%10=8
49%10=9
58%10=8
9 % 10=9

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

Bucket 0	49
Bucket 1	58
Bucket 2	9
Bucket 3	
Bucket 4	
Bucket 5	
Bucket 6	
Bucket 7	
Bucket 8	18
Bucket 9	89

	After insert 89	*After insert 18*	*After insert 49*	*After insert 58*	*After insert 9*
0			49	49	49
1				58	58
2					9
3					
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

Fig: **Linear probing hash table after each insertion**

ii. Quadratic Probing

- **Quadratic Probing eliminates primary clustering problem** of linear probing.
- Collision function is quadratic.

  The popular choice is $f(i) = i^2$.
- If the hash function evaluates to h and a search in cell h is inconclusive, we try cells **$h + 1^2$, $h+2^2$, … $h + i^2$**.
- Remember that subsequent probe points are a **quadratic number of positions** from the original probe point.
- In case of a collision if the hash table is not full, attempt to store key in array elements **$(h+1^2)\%N$, $(h+2^2)\%N$, $(h+3^2)\%N$** … **until you find an empty slot**.

ii. Quadratic Probing

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

	After insert 89	After insert 18	After insert 49	After insert 58	After insert 9
0			49	49	49
1					
2				58	58
3					9
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

Fig: A quadratic probing hash table after each insertion

Double Hashing

- The idea of double hashing: **Make the offset to the next position** probed depend on the key value, so **it can be different for different keys**
  - **Need to introduce a second hash function H_2 (K),** which is **used as the offset in the probe sequence**

$$h_1(x) = x \bmod m$$

$$h_2(x) = x \bmod m'$$

Double Hashing

Suppose we have a list of size 10 (m = 10). We want to put some elements in linear probing fashion. The elements are {96, 48, 26, 68}

h1(x) = x mod 10

h2(x) = x mod 7

h(x,i) = (h1(x) + i h2(x)) mod 10

96%10 => 6

48%10 => 8

26%10=> 6, collision

H(26,1) = (h1(26) + i * h2(26)) %10

= (6 + 5) % 10 = 1 it is free.

26 will be moved to slot 1

Cell	Value	remarks
0		
1	26	Placed using Double Hashing
2		
3		
4		
5		
6	96	26 need this, collision
7		
8	48	
9		

Hashing - Applications

- ✓ Relational DB Query processing
- ✓ File Organization, Telephone Dictionaries.
- ✓ Symbol table of a compiler.
- ✓ Memory-management tables in operating systems.
- ✓ Large-scale distributed systems.
- ✓ Online spelling checkers.
- ✓ Indexes
- ✓ Search engine databases
- ✓ Game programs - (transposition table)