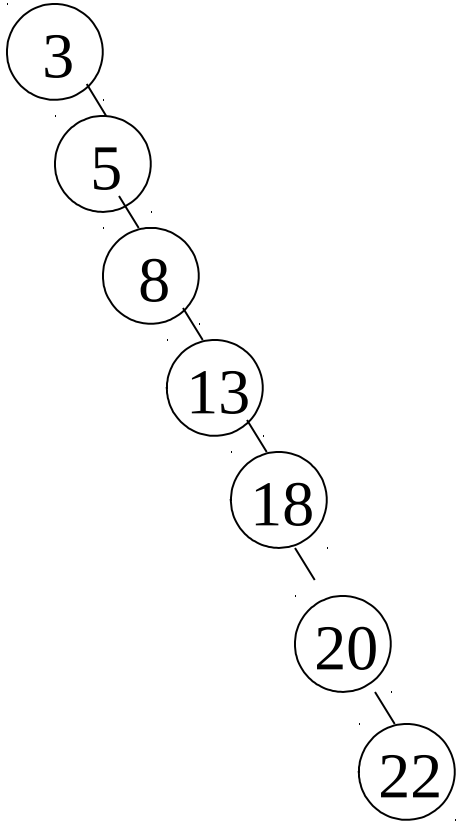
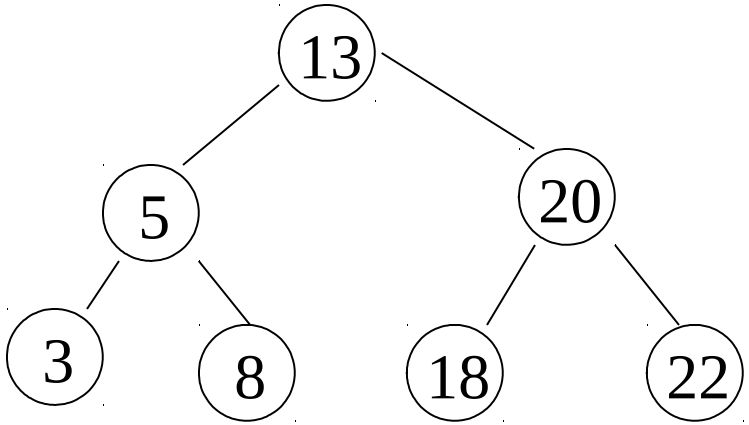
Balanced Search Trees

Chapter 15

Motivation

When building a binary search tree, what type of trees would we like? Example: 3, 5, 8, 20, 18, 13, 22





Balanced binary tree

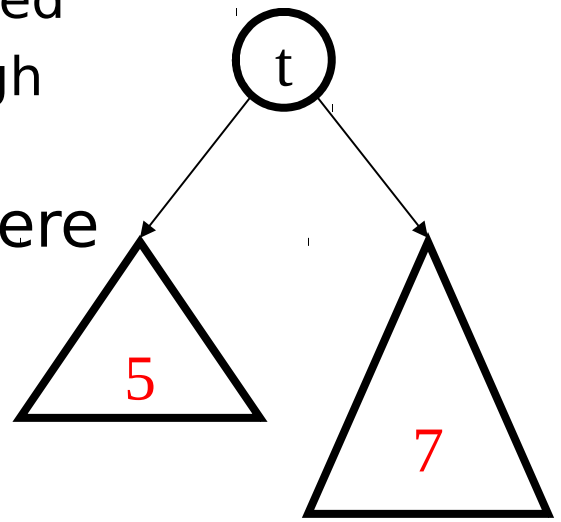
- The disadvantage of a binary search tree is that its height can be as large as N-1
- This means that the time needed to perform insertion and deletion and many other operations can be O(N) in the worst case
- We want a tree with small height
- A binary tree with N node has height at least $\Theta(\log N)$
- Thus, our goal is to keep the height of a binary search tree O(log N)
- Such trees are called balanced binary search trees. Examples are AVL tree, red-black tree.

Balance

Balance == height(left subtree) - height(right subtree)

- zero everywhere ⇒ perfectly balanced
- small everywhere ⇒ balanced enough

Balance between -1 and 1 everywhere
⇒ maximum height of ~1.44 log n

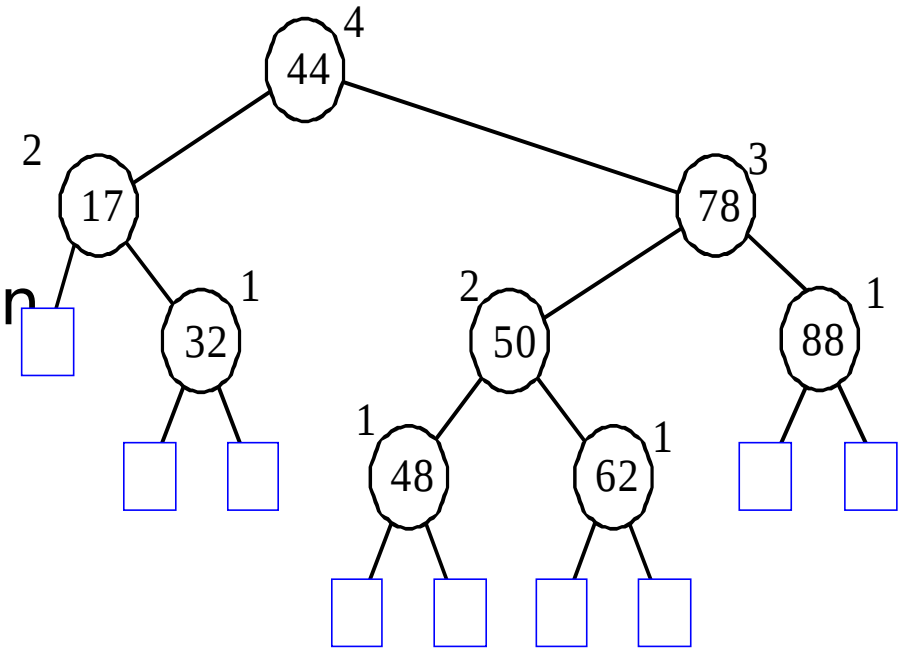


AVL Trees

A binary search tree is said to be AVL balanced if:

- The difference in the heights between the left and right sub-trees is at most 1, and
- Both sub-trees are themselves AVL trees

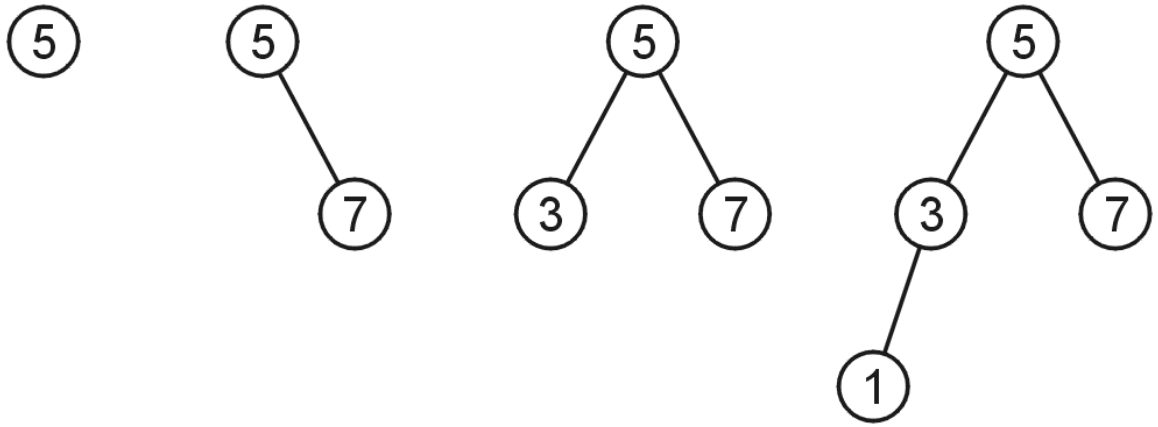
Operations : Rotation, Insertion and Deletion



An example of an AVL tree where the heights are shown next to the nodes:

AVL Trees

AVL trees with 1, 2, 3, and 4 nodes:



Height of an AVL tree

- Theorem: The ***height*** of an AVL tree storing n keys is O(log n).
- **Proof**:
  - Let us bound **n(h),** the minimum number of internal nodes of an AVL tree of height h.
  - We easily see that n(0) = 1 and n(1) = 2
  - For h > 2, an AVL tree of height h contains the root node, one AVL subtree of height h-1 and another of height h-2 (at worst).
  - That is, **n(h) >= 1 + n(h-1) + n(h-2)**
  - Knowing n(h-1) > n(h-2), we get n(h) > 2n(h-2). So

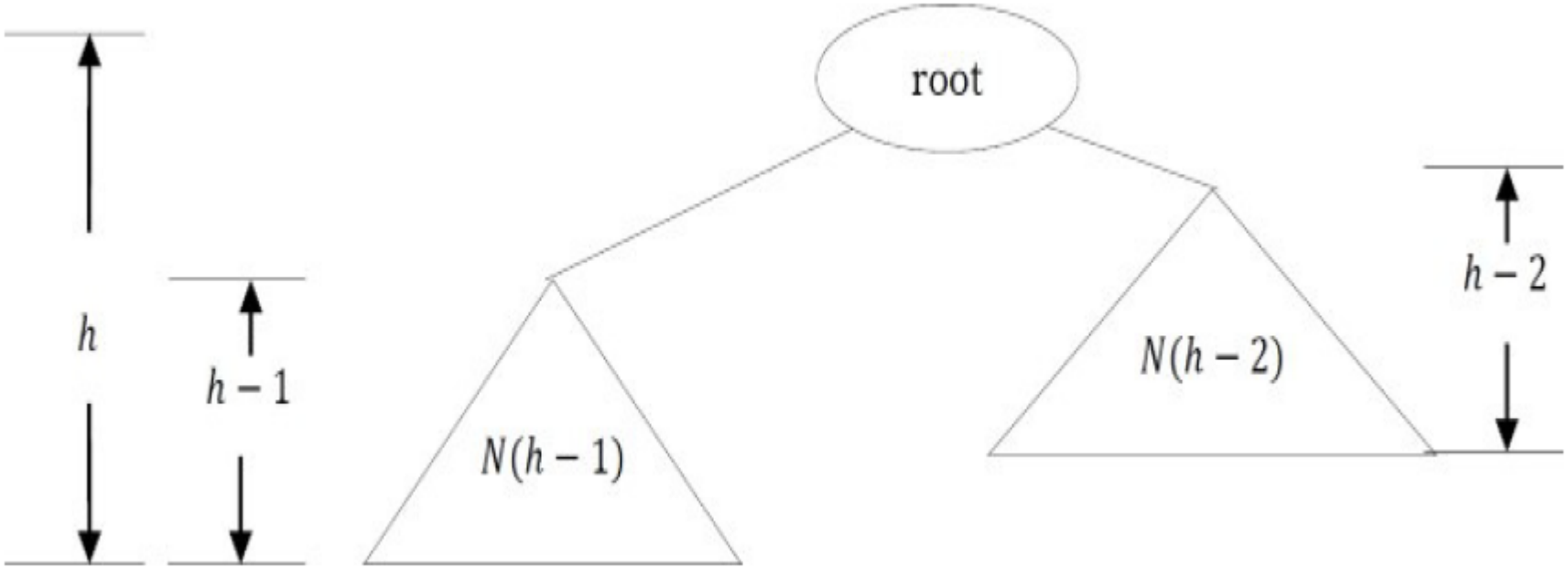
    n(h) > 2n(h-2), n(h) > 4n(h-4), n(h) > 8n(n-6), … (by induction),

    $n(h) > 2^i n(h-2i)$
  - Solving the base case we get: $n(h) > 2^{h/2-1}$
  - Taking logarithms: h < 2log n(h) +2
  - Since n>=n(h), h < 2log(n)+2 and the height of an AVL tree is O(log n)

Minimum/Maximum Number of Nodes in AVL Tree

If we fill the left subtree with height h – 1 then we should fill the right subtree with height h – 2. As a result, the minimum number of nodes with height h is:

$N(h) = N(h - 1) + N(h - 2) + 1$



AVL Tree Insert and Remove

- Do binary search tree insert and remove
- The balance condition can be violated sometimes
  - Do something to fix it : rotations
  - After rotations, the balance of the whole tree is maintained

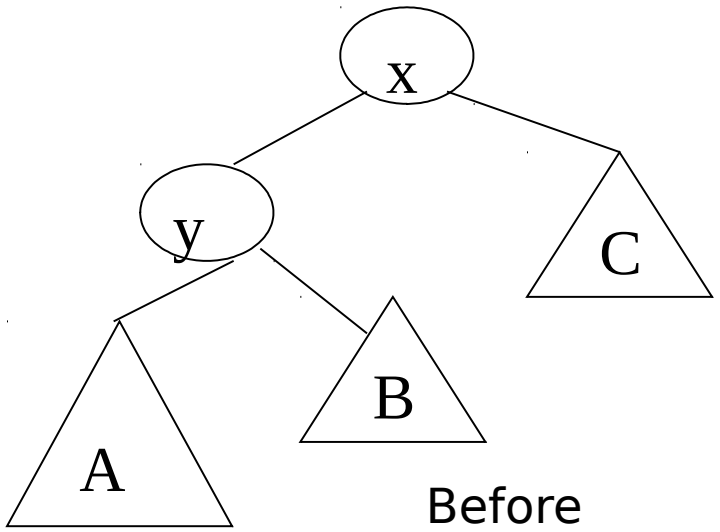
Rotations

- Since an insertion/deletion involves adding/deleting a single node, this can only increase/decrease the height of some subtree by 1
- 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 x to restore the AVL tree property.

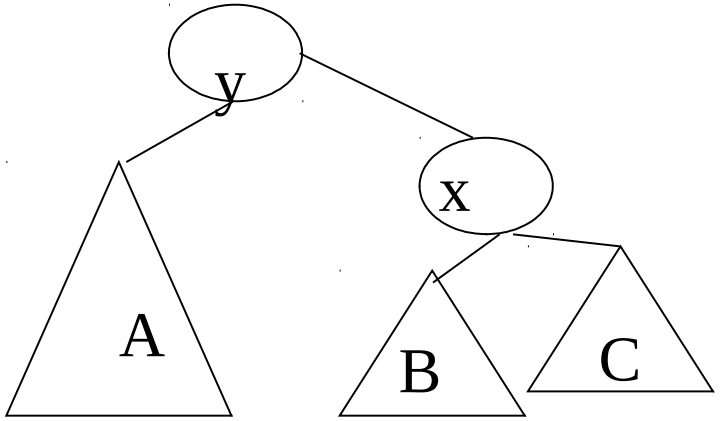
Rotations

- 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.

e.g. Single Rotation



Before Rotation

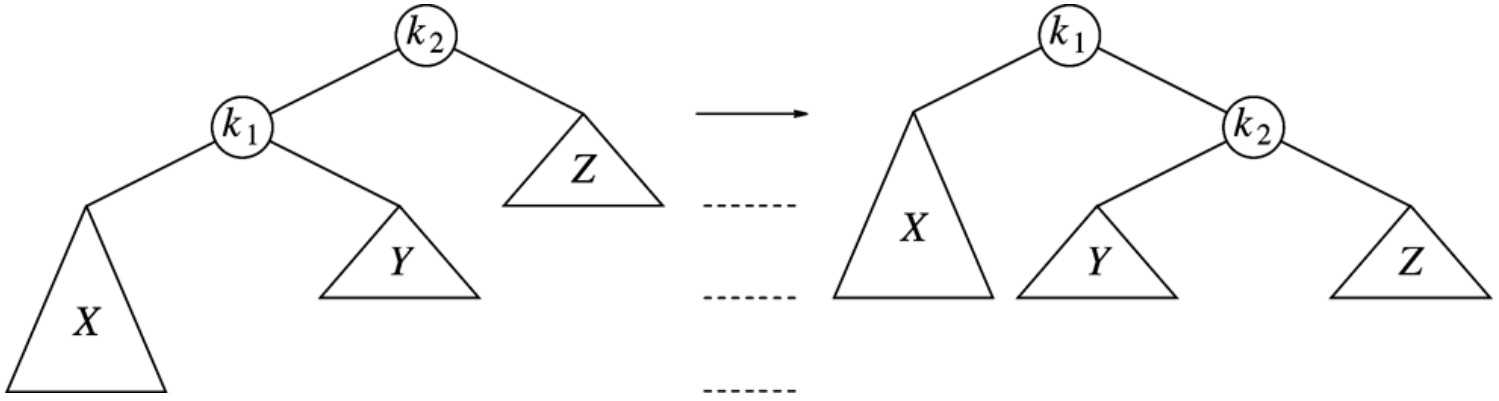


After Rotation

AVL Trees Complexity

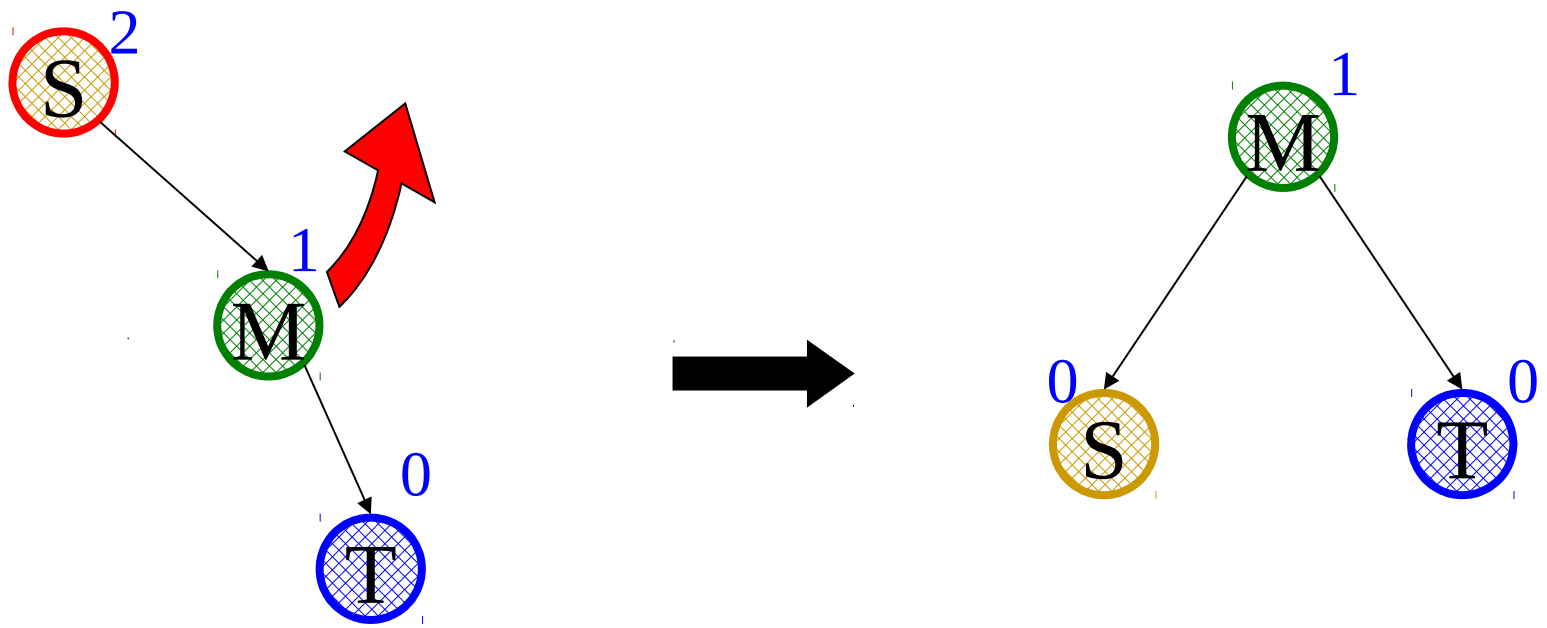
- Overhead
  - Extra space for maintaining height information at each node
  - Insertion and deletion become more complicated, but still O(log N)
- Advantage
  - Worst case O(log(N)) for insert, delete, and search

Single Rotation (Case 1)



- Replace node k_2 by node k_1
- Set node k_2 to be right child of node k_1
- Set subtree Y to be left child of node k_2

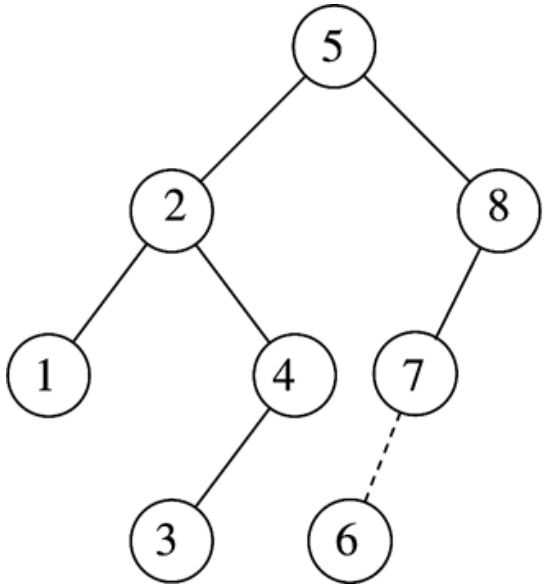
Single Rotation

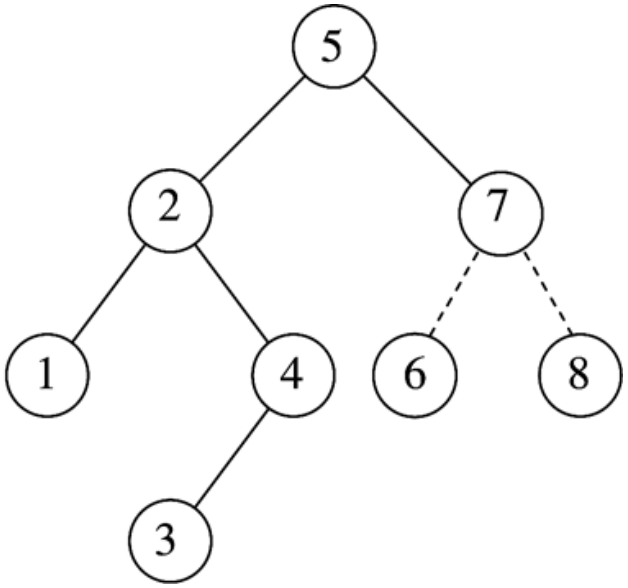


Basic operation used in AVL trees:

A right child could legally have its parent as its left child.

Example





- After inserting 6
  - Balance condition at node 8 is violated

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.

Insertion

- Let x be the node at which left(x) and right(x) differ by more than 1
- Assume that the height of x is h+3
- There are 4 cases
  - Height of left(x) is h+2 (i.e. height of right(x) is h)
    - Height of left(left(x)) is h+1 ⇒ single rotate with left child
    - Height of right(left(x)) is h+1 ⇒ double rotate with left child
  - Height of right(x) is h+2 (i.e. height of left(x) is h)
    - Height of right(right(x)) is h+1 ⇒ single rotate with right child
    - Height of left(right(x)) is h+1 ⇒ double rotate with right child

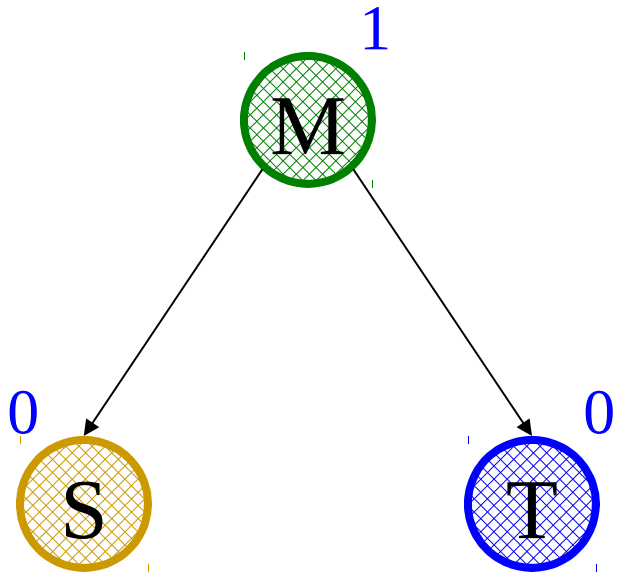
Good Insert Case:
Balance Preserved

Good case: insert middle, then small,then tall

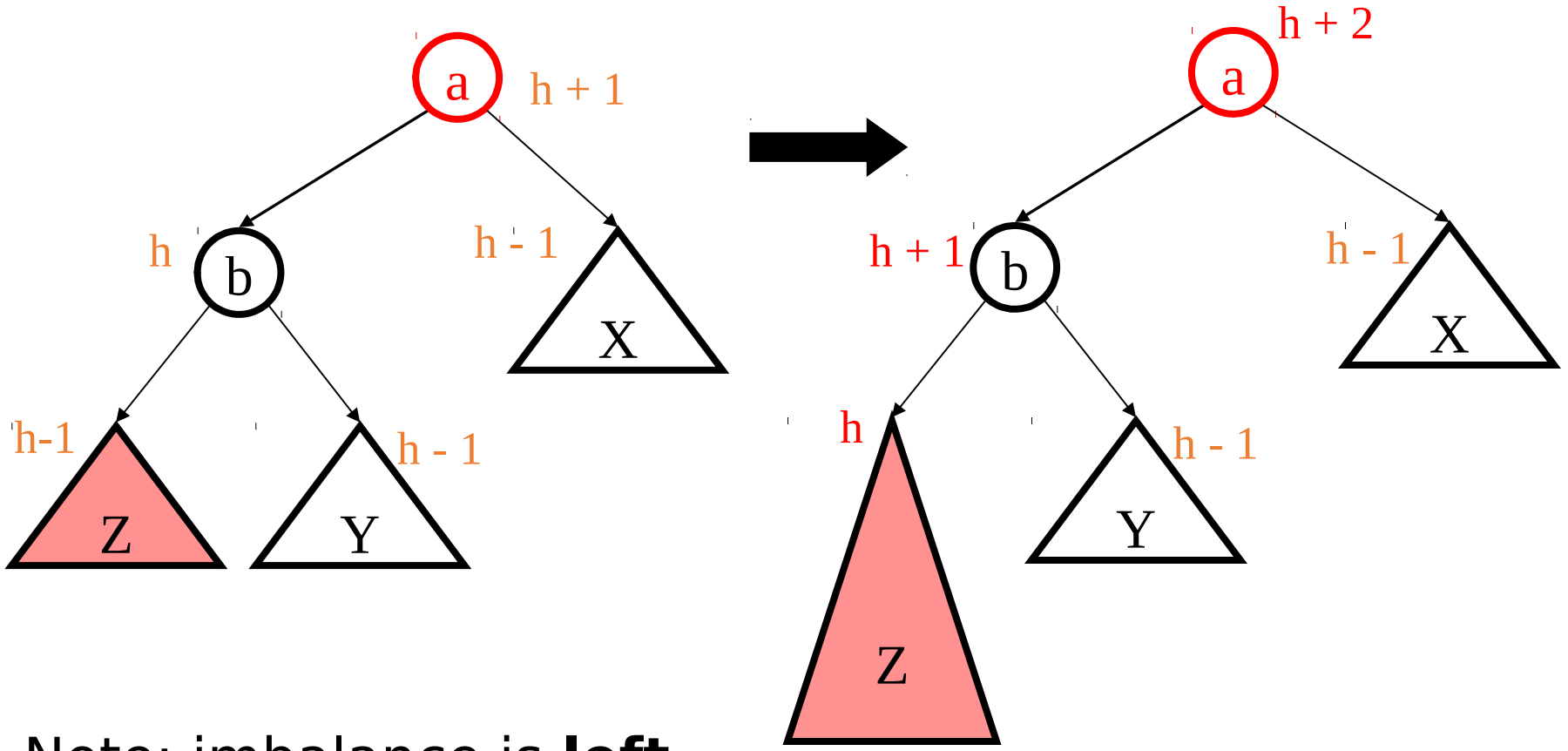
Insert(middle)

Insert(small)

Insert(tall)



General Bad Case #1



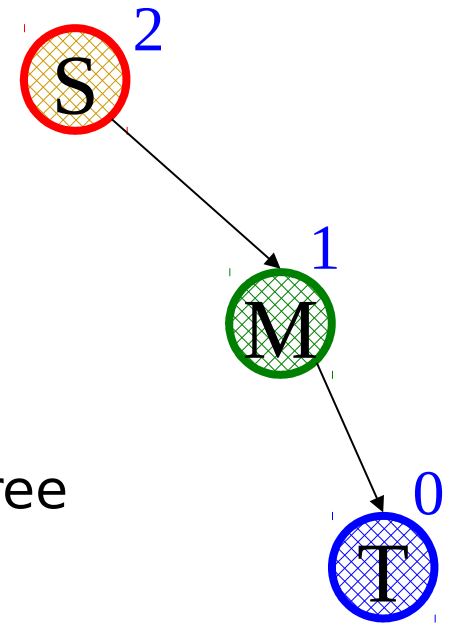
Note: imbalance is **left-left**

Bad Insert Case #1: Left-Left or Right-Right Imbalance

Insert(small)

Insert(middle)

Insert(tall)

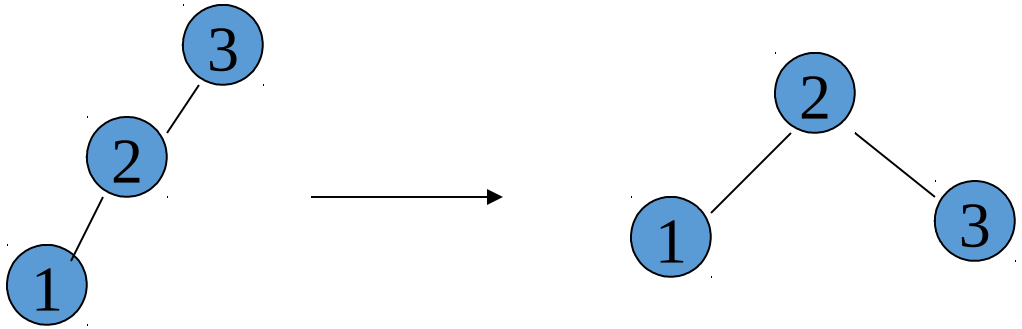


BC#1 Imbalance caused by either:

- Insert into **left** child's **left** subtree
- Insert into **right** child's **right** subtree

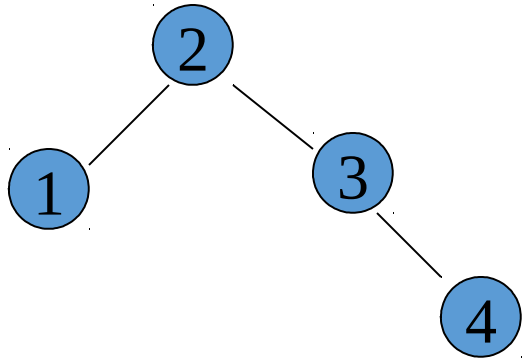
Example

- Inserting 3, 2, 1, and then 4 ,5 sequentially into empty AVL tree

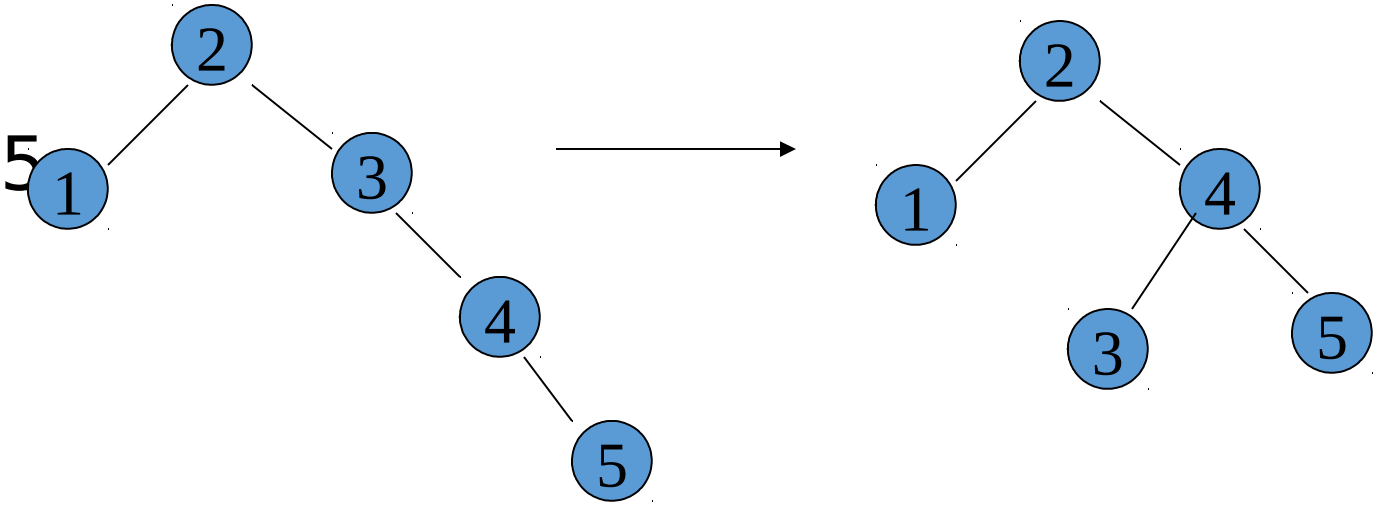


Example (Cont'd)

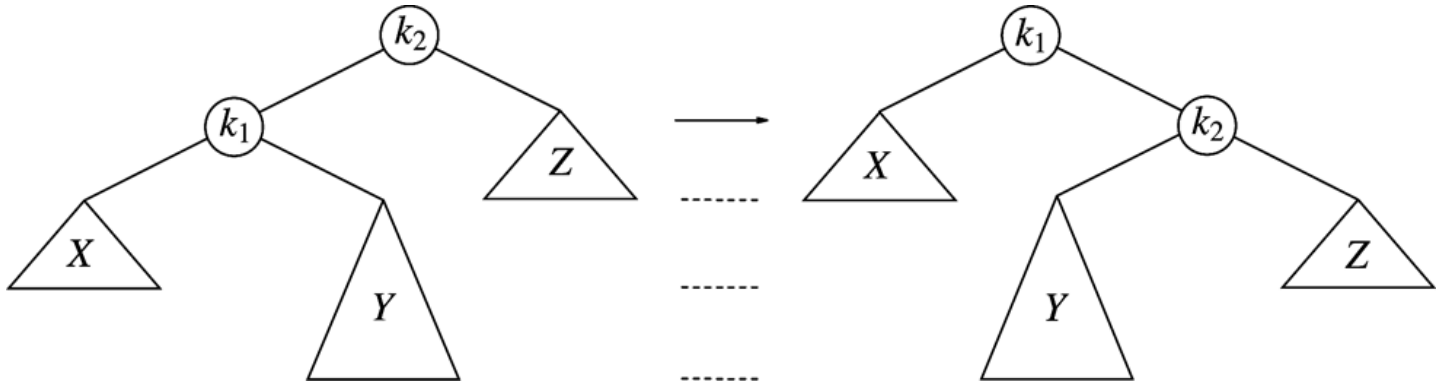
- Inserting 4



- Inserting 5

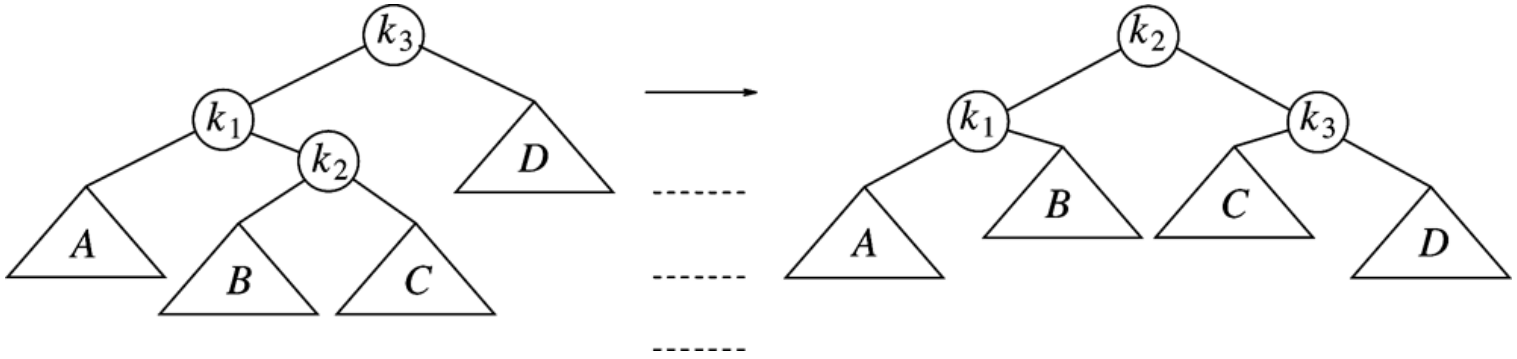


Single Rotation Will Not Work for the Other Case



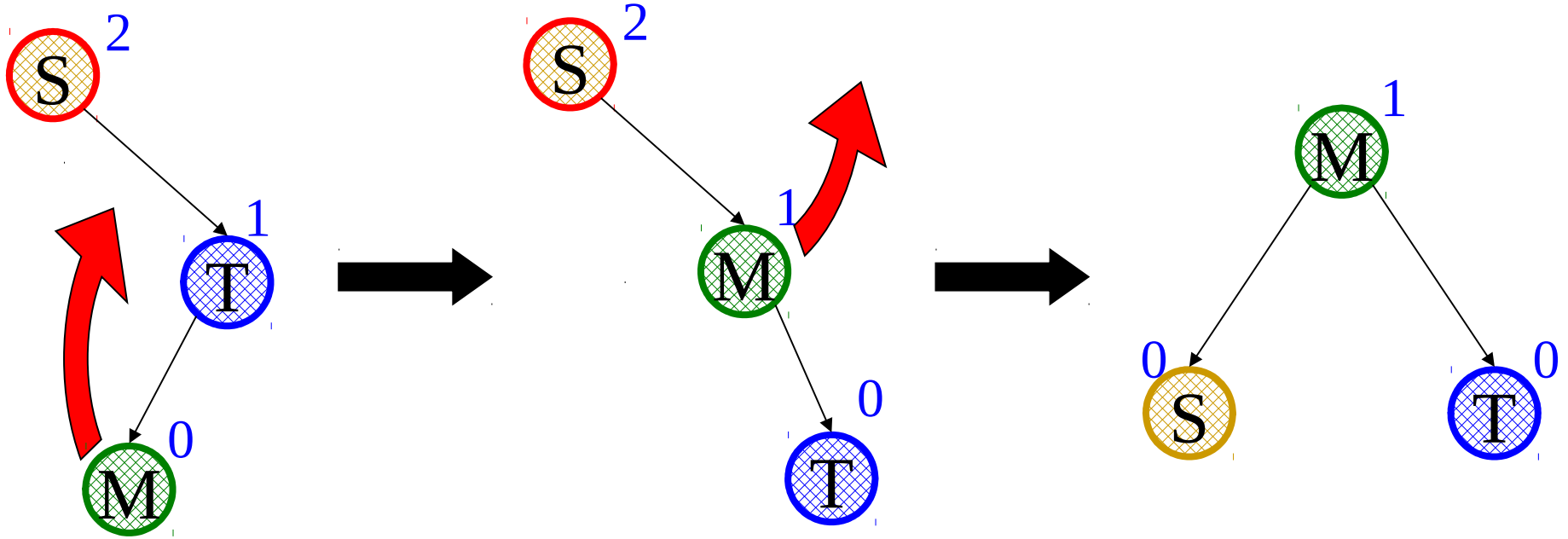
- After single rotation, k_1 still not balanced
- Double rotations needed

Double Rotation (Case 2)

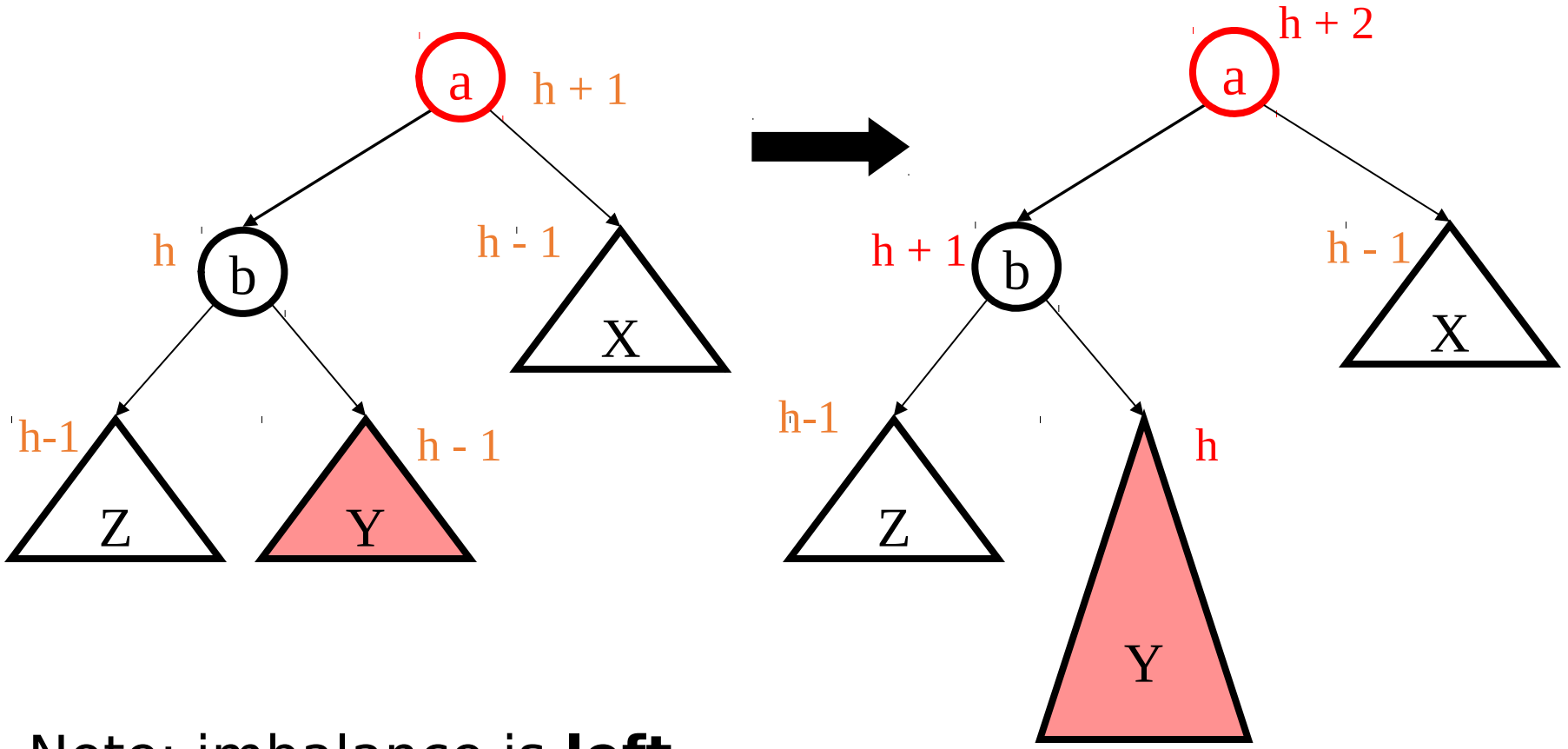


- Left-right double rotation to fix case 2
- First rotate between **k_1** and **k_2**
- Then rotate between **k_2** and **k_3**

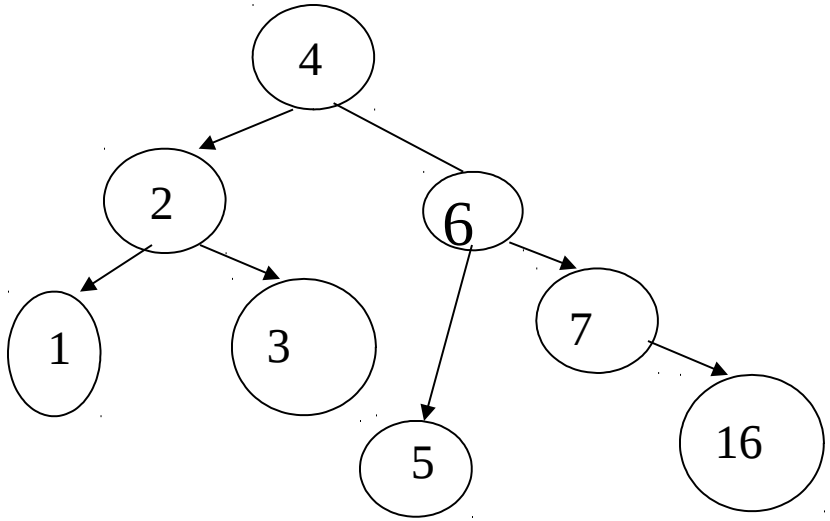
Double Rotation

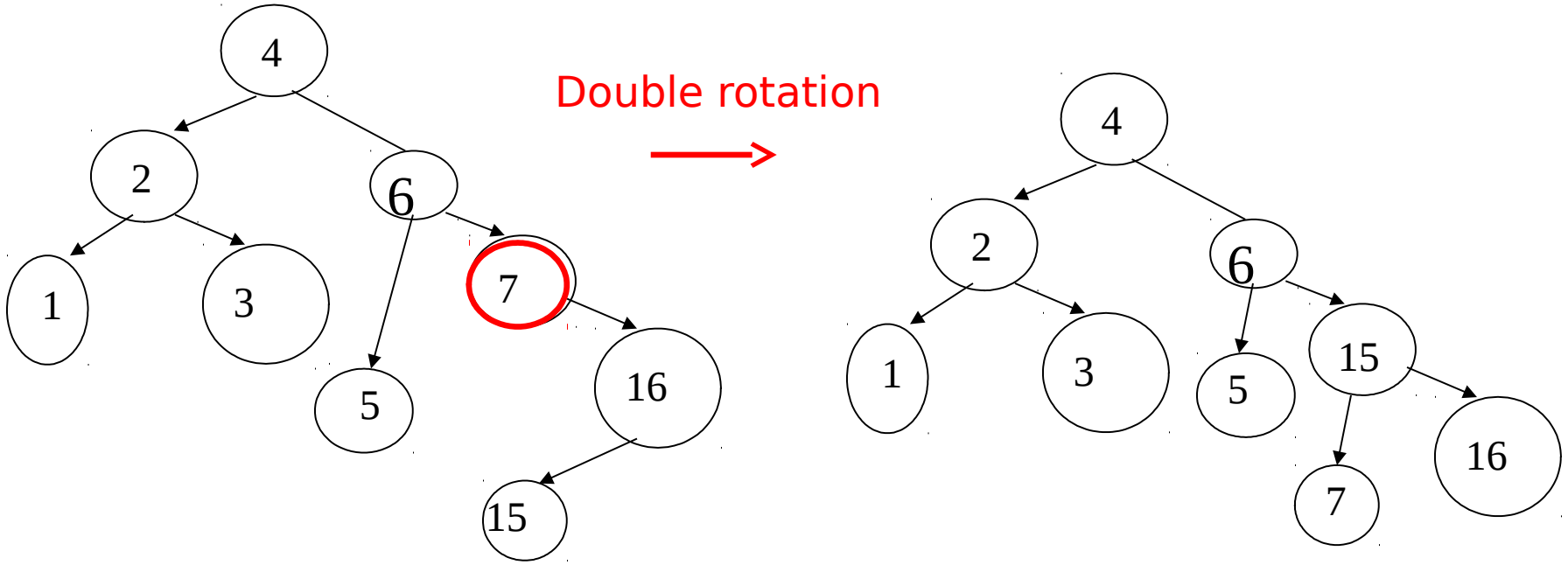


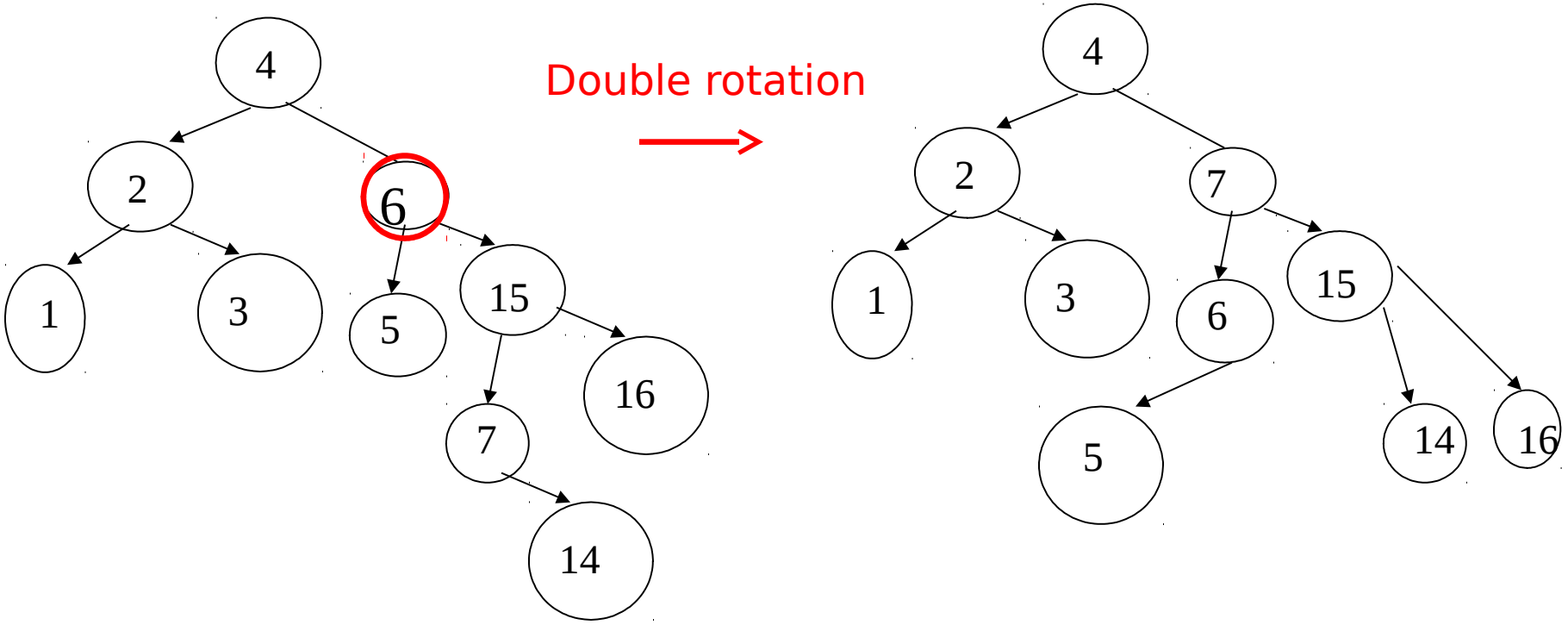
General Bad Case #2



Note: imbalance is **left-right**

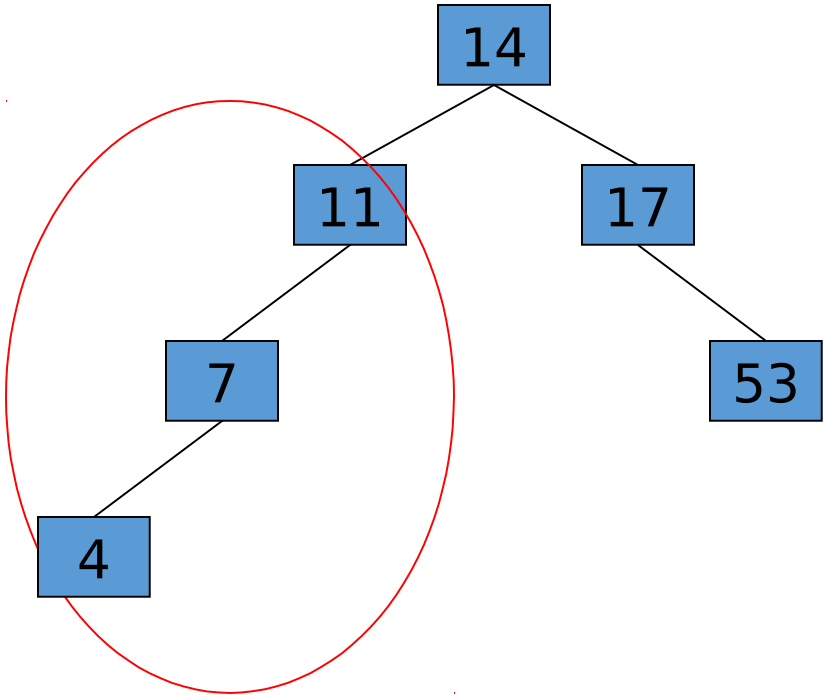

4
2
6
1
3
7
5
16


4
2
6
1
3
7
5
16
15
Double rotation
4
2
6
1
3
5
15
7
16


4
2
6
1
3
5
15
16
7
14
Double rotation
4
2
7
1
3
6
15
5
14
16

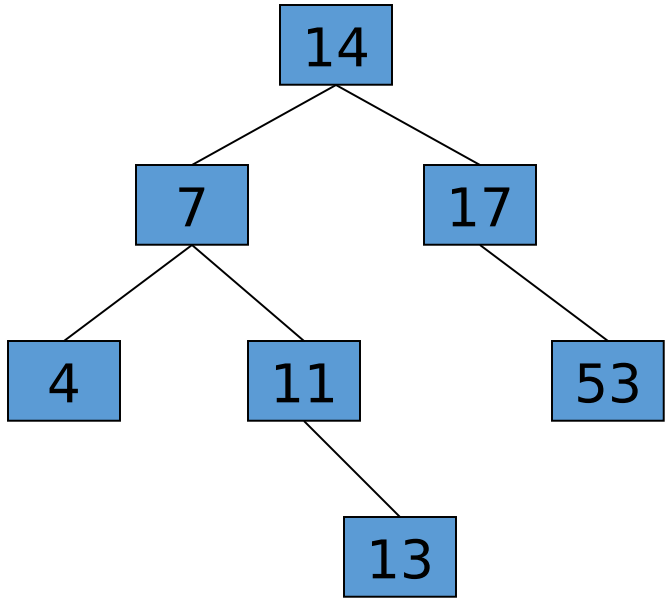
AVL Tree Example - 2

Insert 14, 17, 11, 7, 53, 4, 13 into an empty AVL tree



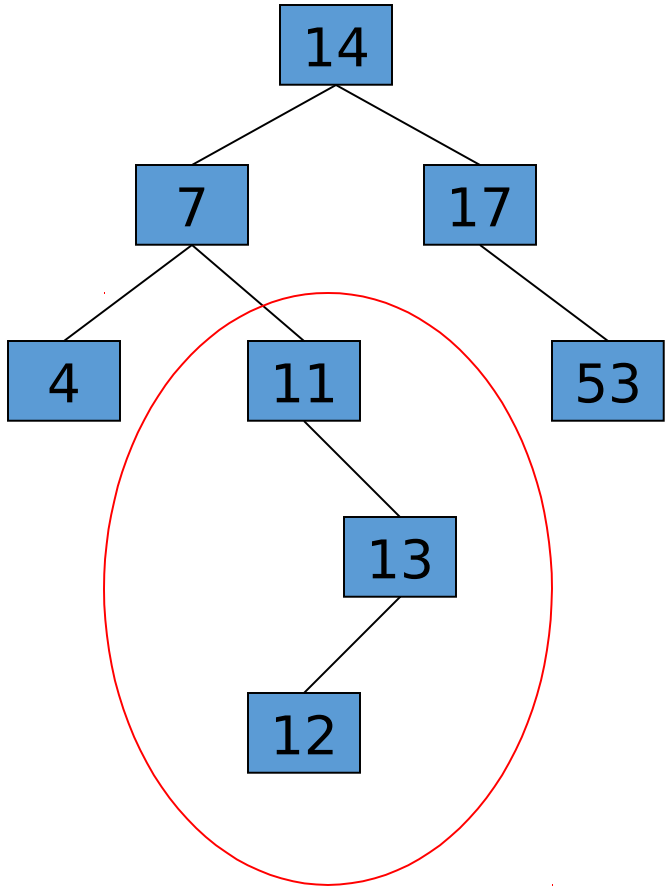
AVL Tree Example:

- **Insert 14, 17, 11, 7, 53, 4, 13 into an empty AVL tree**



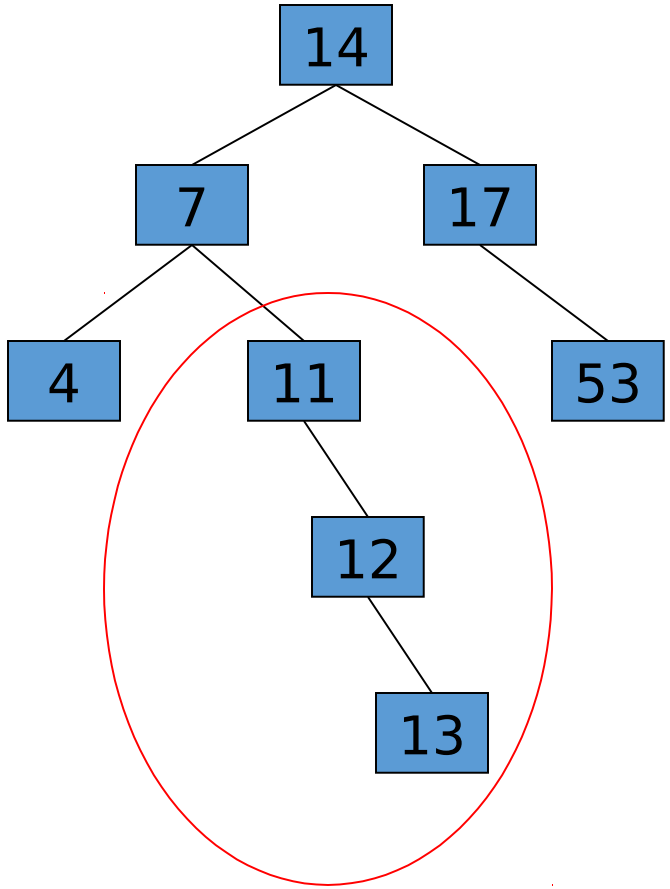
AVL Tree Example:

- **Now insert 12**



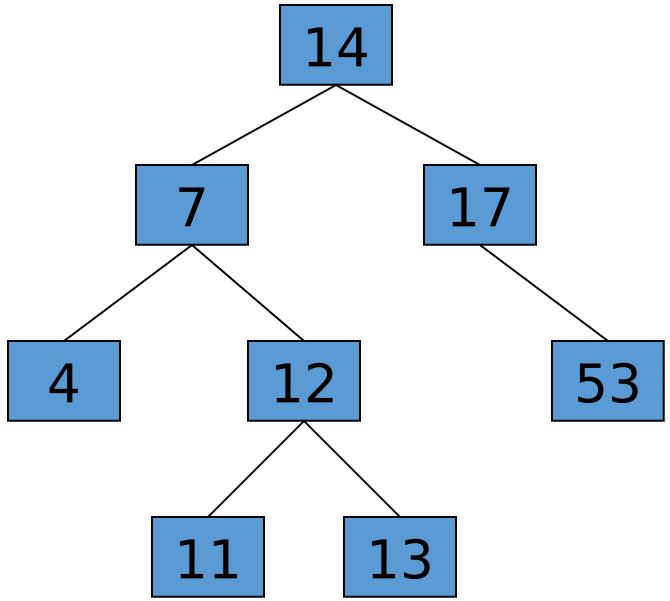
AVL Tree Example:

- **Now insert 12**



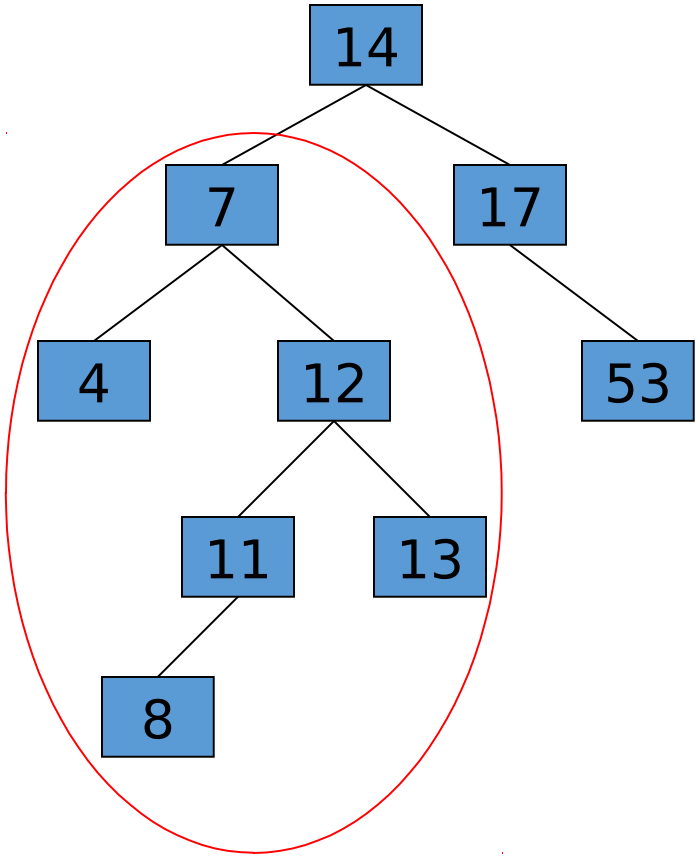
AVL Tree Example:

- **Now the AVL tree is balanced.**



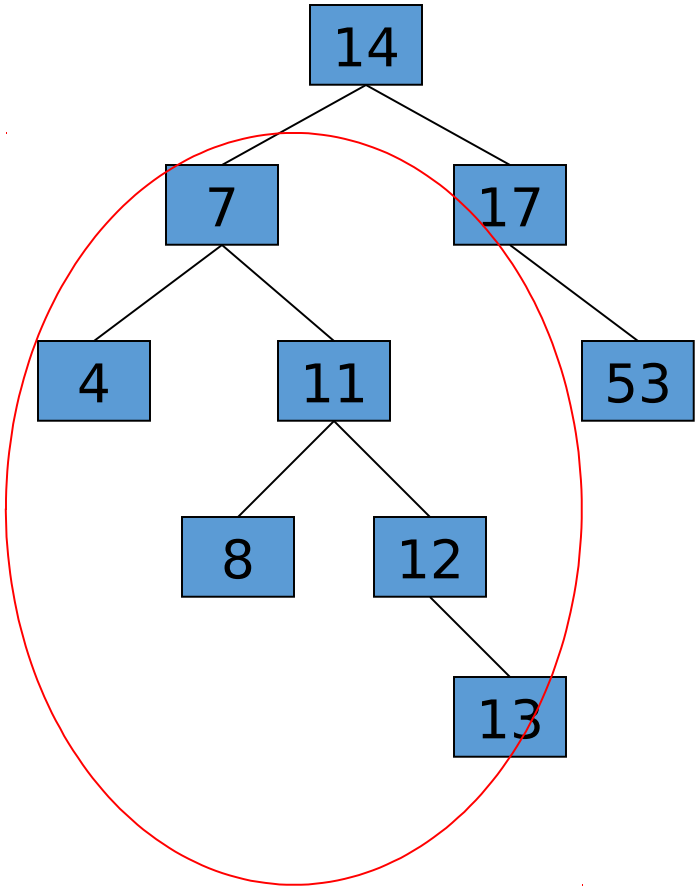
AVL Tree Example:

- **Now insert 8**



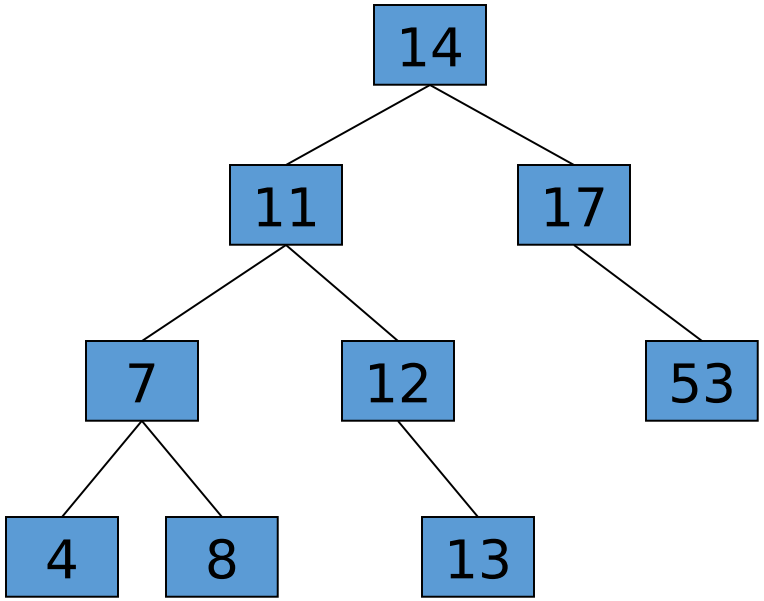
AVL Tree Example:

- **Now insert 8**



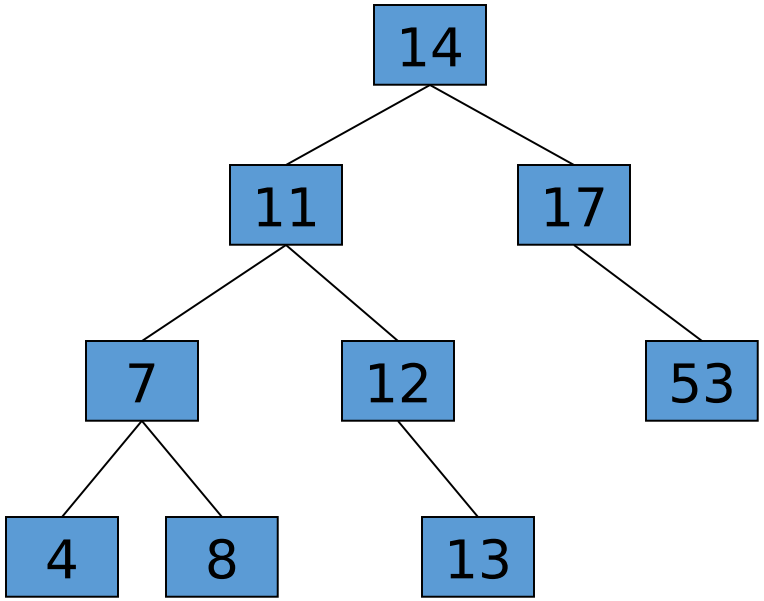
AVL Tree Example:

- **Now the AVL tree is balanced.**



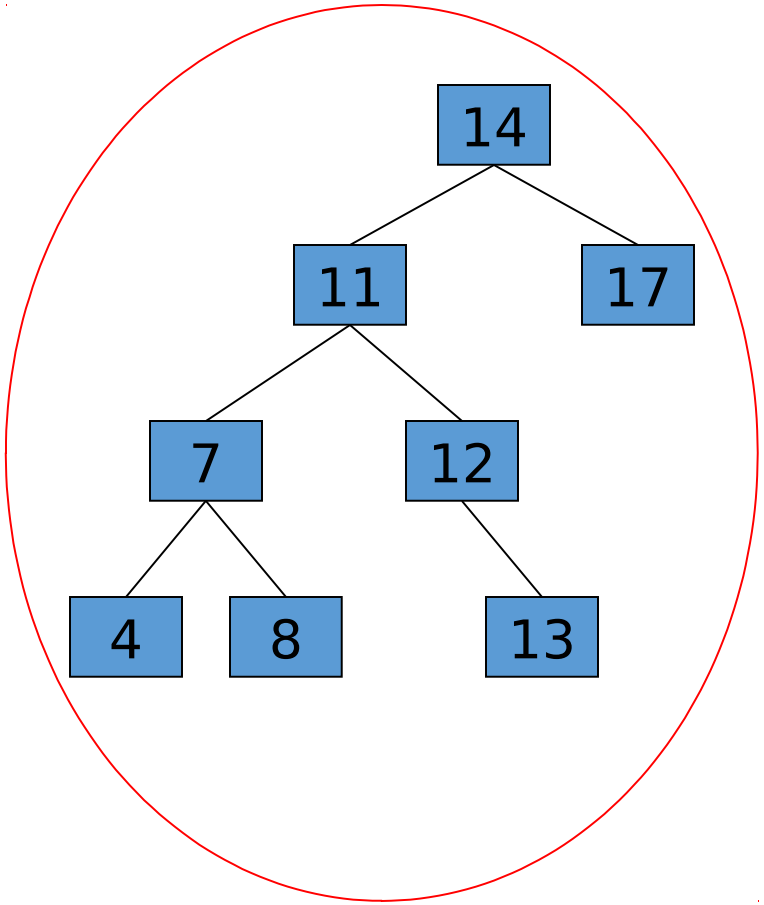
AVL Tree Example:

- **Now remove 53**



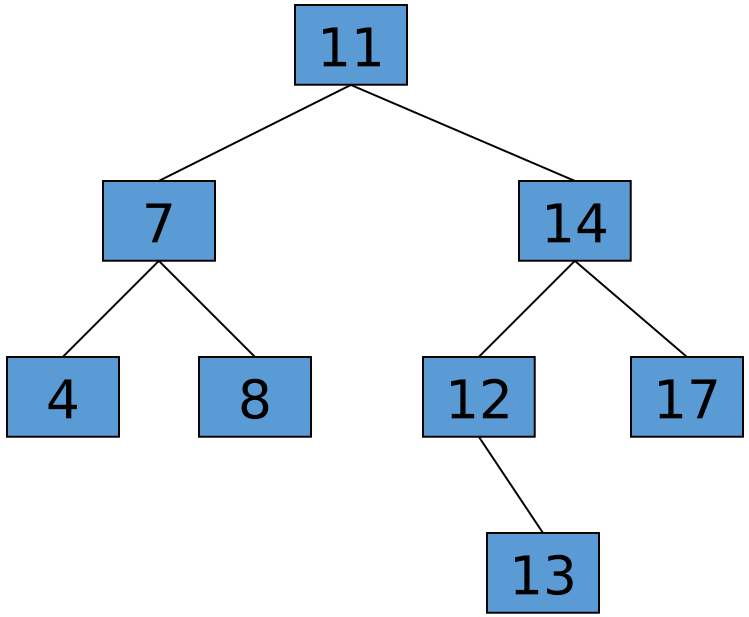
AVL Tree Example:

- **Now remove 53, unbalanced**



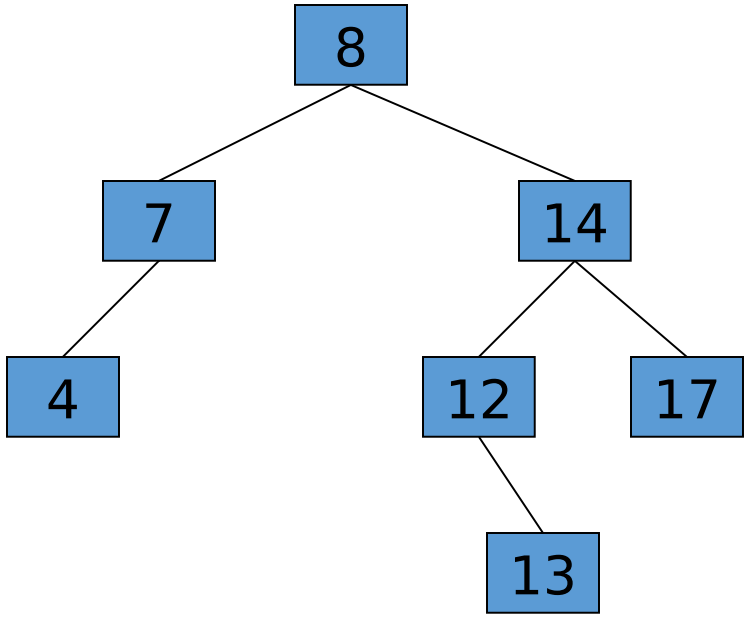
AVL Tree Example:

- **Balanced! Remove 11**



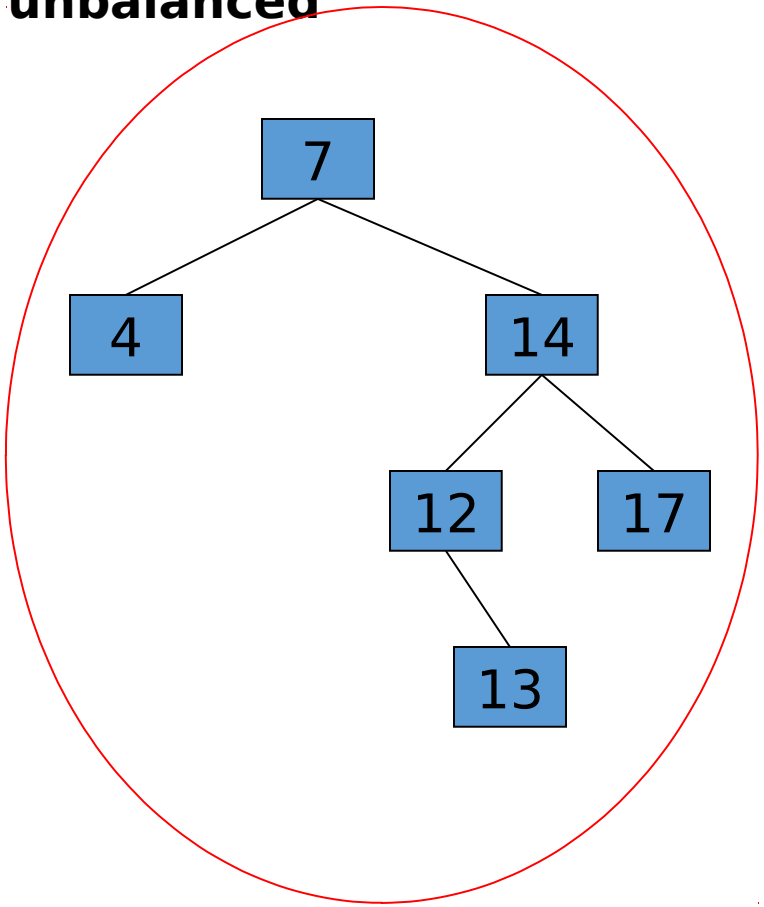
AVL Tree Example:

- **Remove 11, replace it with the largest in its left branch**



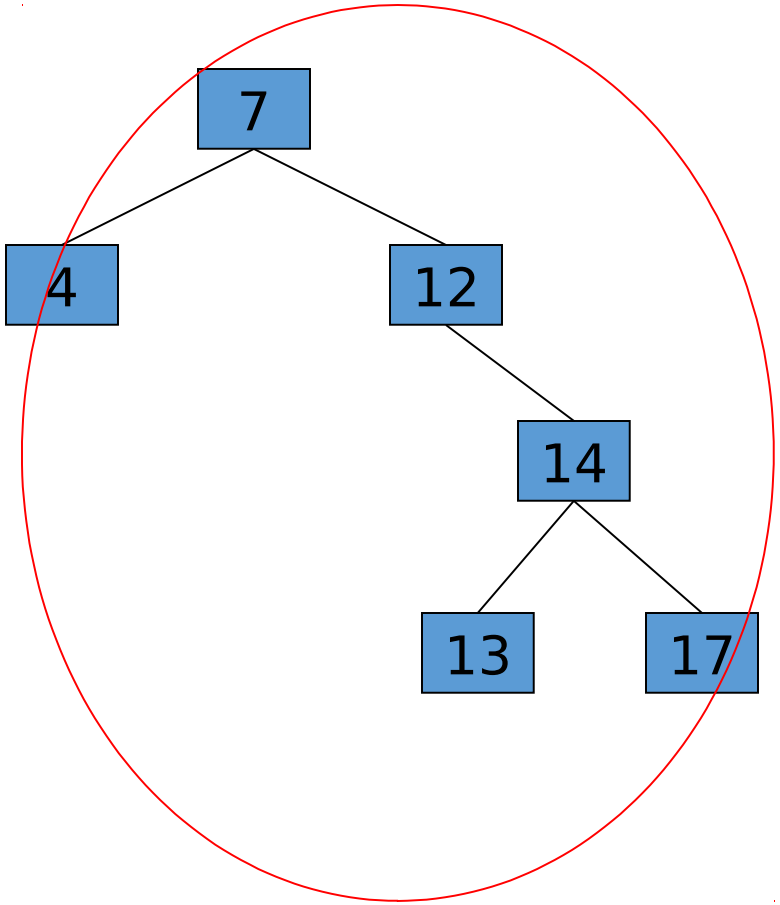
AVL Tree Example:

- **Remove 8, unbalanced**



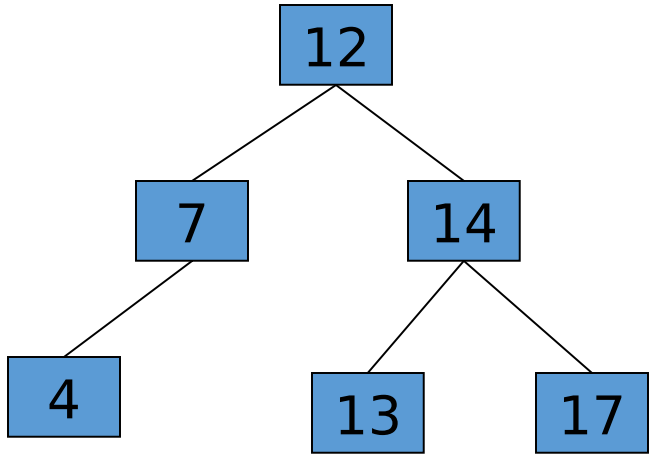
AVL Tree Example:

- **Remove 8, unbalanced**



AVL Tree Example:

- **Balanced!!**



Deletion

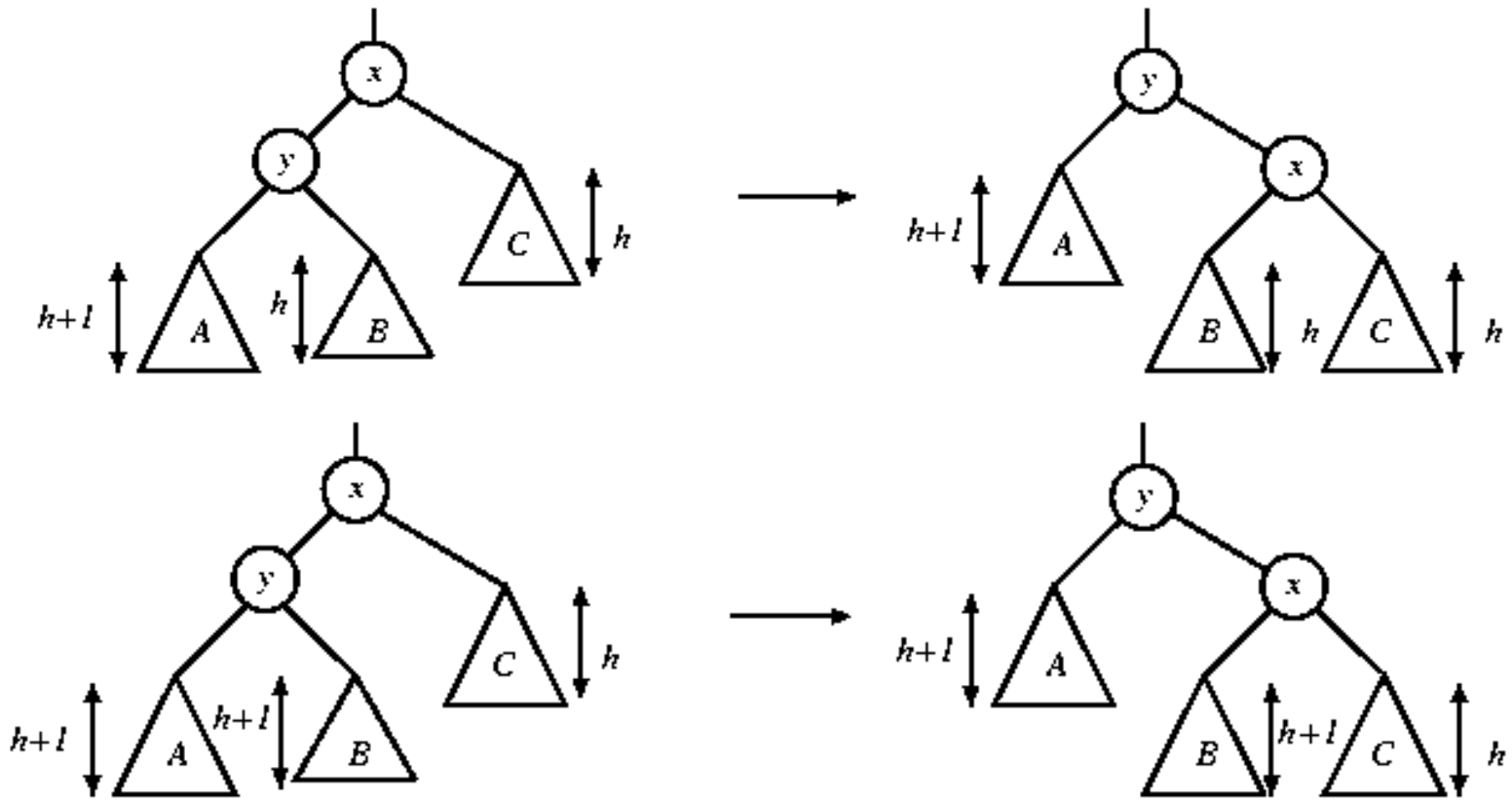
- Delete a node x as in ordinary binary search tree. Note that the last node deleted is a leaf.
- Then trace the path from the new leaf towards the root.
- For each node x encountered, check if heights of left(x) and right(x) differ by at most 1. If yes, proceed to parent(x). If not, perform an appropriate rotation at x. There are 4 cases as in the case of insertion.
- For deletion, after we perform a rotation at x, we may have to perform a rotation at some ancestor of x. Thus, we must continue to trace the path until we reach the root.

Deletion

- On closer examination: the single rotations for deletion can be divided into 4 cases (instead of 2 cases)
  - Two cases for rotate with left child
  - Two cases for rotate with right child

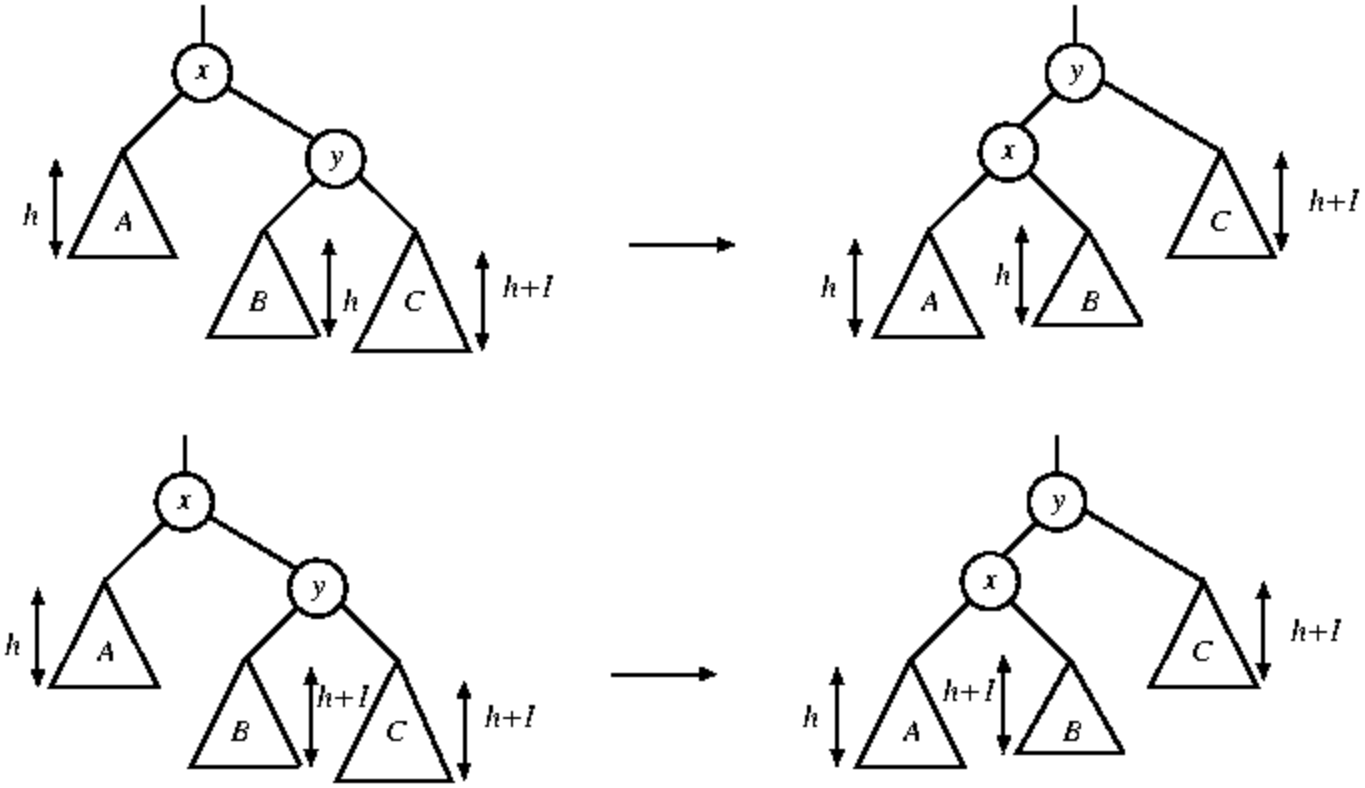
Single rotations in deletion

In both figures, a node is deleted in subtree C, causing the height to drop to h. The height of y is h+2. When the height of subtree A is h+1, the height of B can be h or h+1. Fortunately, the same single rotation can correct both cases.



Single rotations in deletion

In both figures, a node is deleted in subtree A, causing the height to drop to h. The height of y is h+2. When the height of subtree C is h+1, the height of B can be h or h+1. A single rotation can correct both cases.



rotate with right child

Pros and Cons of AVL Trees

Pros of AVL trees:

1. Search is O(log N) since AVL trees are always balanced.
2. Insertion and deletions are also O(logn)
3. The height balancing adds no more than a constant factor to the speed of insertion.

Cons of AVL trees:

4. Difficult to program & debug; more space for balance factor.
5. Asymptotically faster but rebalancing costs time.
6. Most large searches are done in database systems on disk and use other structures (e.g. B-trees).
7. May be OK to have O(N) for a single operation if total run time for many consecutive operations is fast (e.g. Splay trees).

Red Black Trees

Colored Nodes Definition

- Binary search tree.
- Each node is colored red or black.
- Root and all external nodes are black.
- No root-to-external-node path has two consecutive red nodes.
- All root-to-external-node paths have the same number of black nodes

Red Black Trees

Colored Edges Definition

- Binary search tree.
- Child pointers are colored red or black.
- Pointer to an external node is black.
- No root to external node path has two consecutive red pointers.
- Every root to external node path has the same number of black pointers.

Definition: A Red-black tree is a binary search tree that satisfies the following properties:

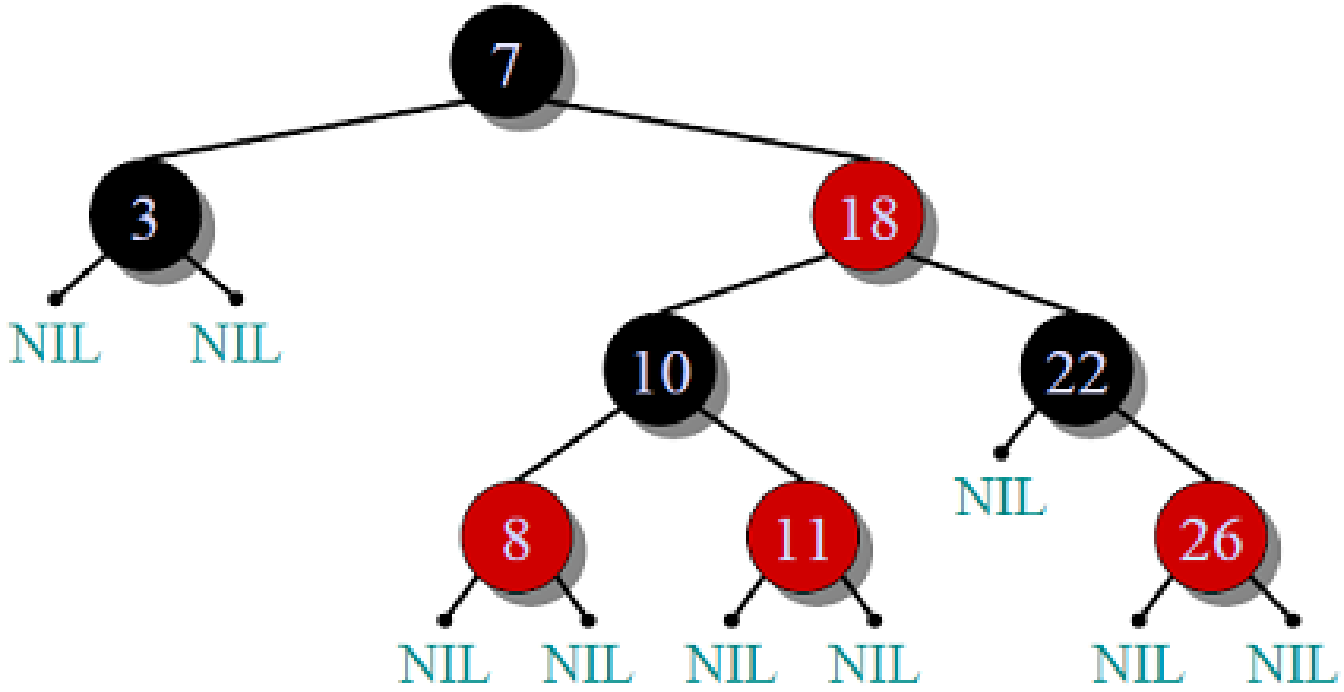
- Root Property: the root is black
- External Property: every leaf is black
- Internal Property: the children of a red node are black
- Depth Property: all the leaves have the same black

The height of a red black tree that has n (internal) nodes is between $\log_2(n+1)$ and $2\log_2(n+1)$.

Why Red-Black Trees?

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that height of the tree remains O(Logn) after every insertion and deletion, then we can guarantee an upper bound of O(Logn) for all these operations. The height of a Red-Black tree is always O(Logn) where n is the number of nodes in the tree.

Example Red-Black Tree



Operations: Insertion and Deletion

Properties

- Start with a red black tree whose height is h;
- collapse all red nodes into their parent black nodes to get a tree whose node-degrees are between 2 and 4, height is >= h/2, and all external nodes are at the same level.

Properties

- Let h’>= h/2 be the height of the collapsed tree.
- In worst-case, all internal nodes of collapsed tree have degree 2.
- Number of internal nodes in collapsed tree >= $2^{h'}$-1.
- So, n >= $2^{h'}$-1
- So, h <= 2 $\log_2$ (n + 1)

Properties

- At most 1 rotation and O(log n) color flips per insert/delete.
- Priority search trees.
  - Two keys per element.
  - Search tree on one key, priority queue on other.
  - Color flip doesn’t disturb priority queue property.
  - Rotation disturbs priority queue property.
  - O(log n) fix time per rotation O($\log^2 n$) overall time.

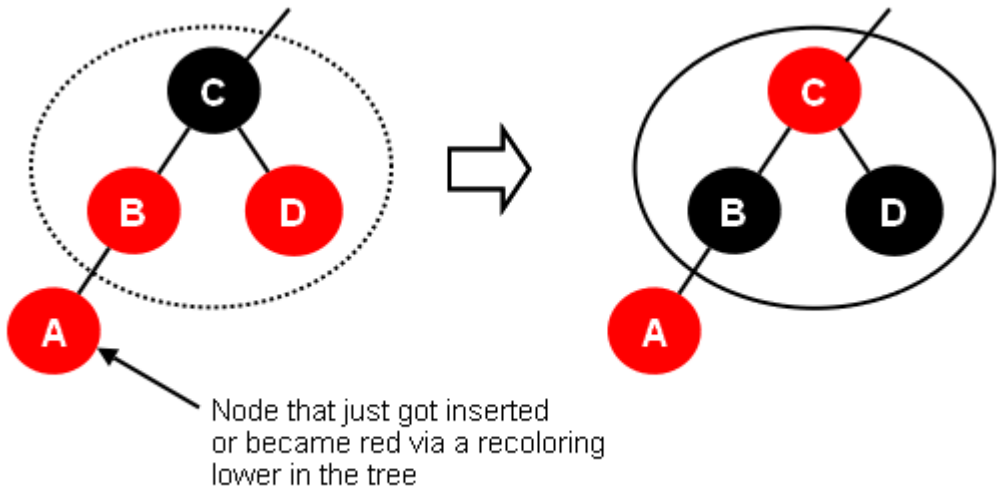
Insertion

The algorithm has three steps:

- Insert as you would into a BST, coloring the node red.
- If the parent of the node you just inserted was red, you have a double-red problem which you must correct.
- Color the root node black.
- A double red problem is corrected with zero or more **recolorings** followed by zero or one **restructuring**.

Recoloring

- Recolor whenever *the sibling of a red node's red parent is red*:

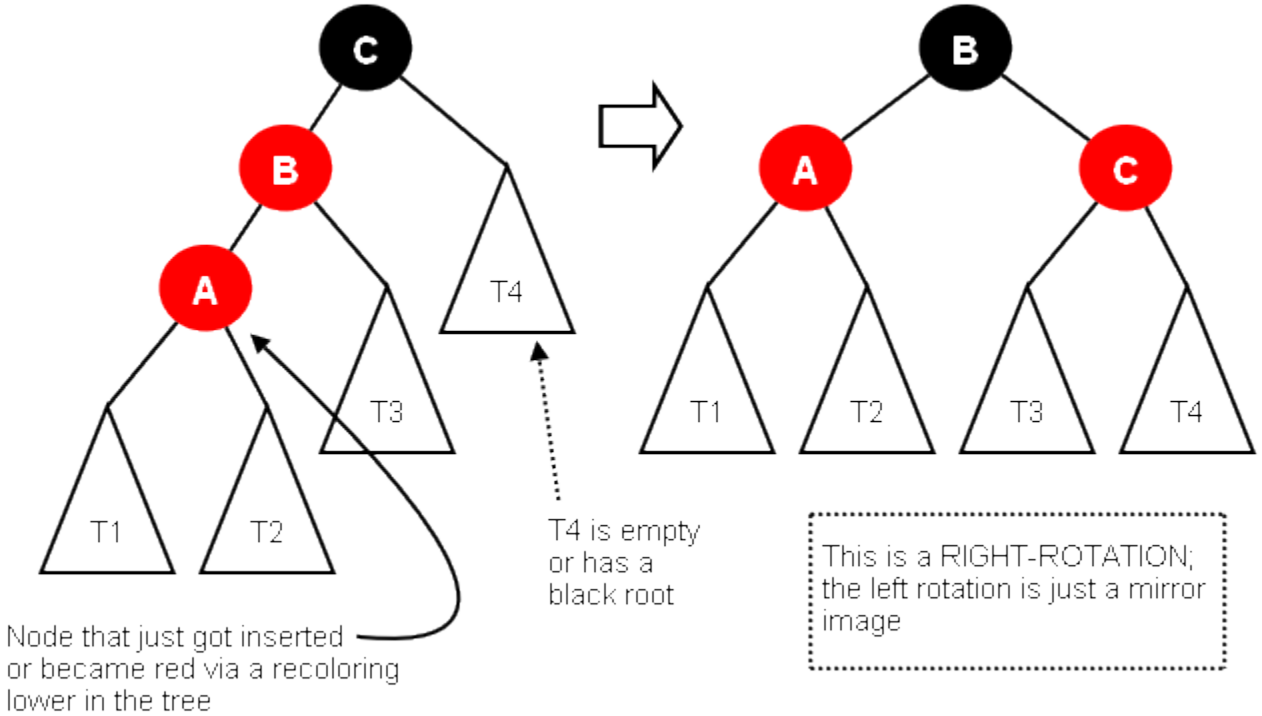


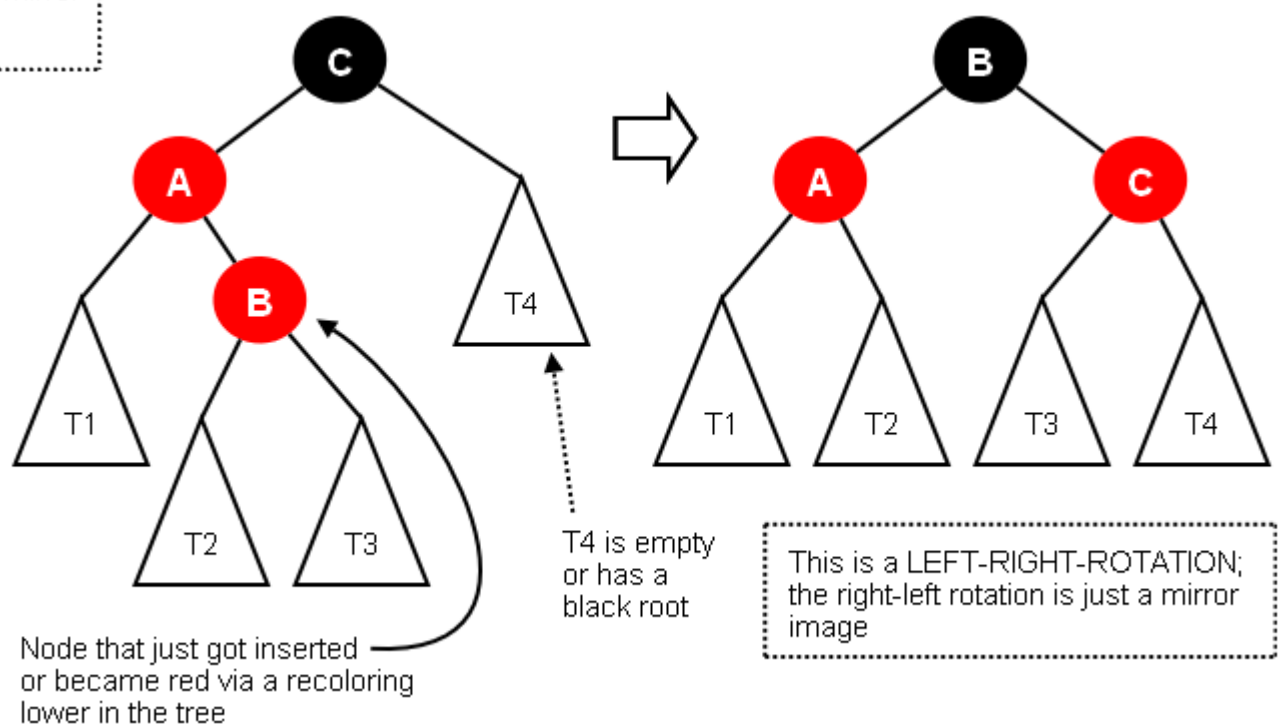
Restructuring

Restructure whenever *the red child's red parent's sibling is black or null*. There are four cases:

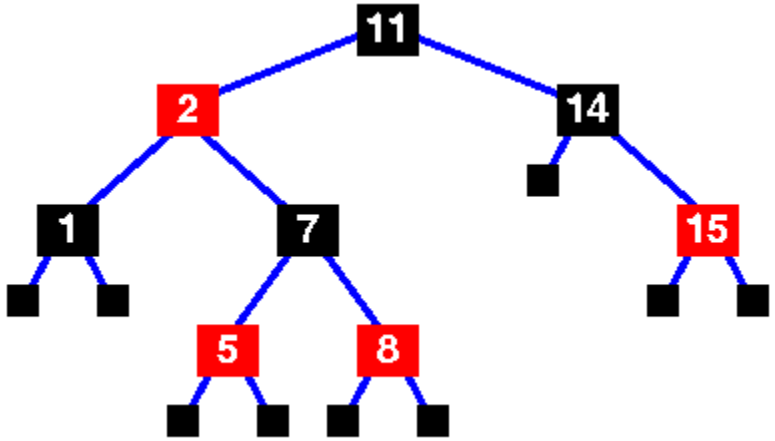
- Right
- Left
- Right-Left
- Left-Right

When you restructure, the root of the restructured subtree is colored black and its children are colored red.


C
B
A
T1
T2
T3
T4
B
A
C
T1
T2
T3
T4
Node that just got inserted or became red via a recoloring lower in the tree
T4 is empty or has a black root
This is a RIGHT-ROTATION; the left rotation is just a mirror image

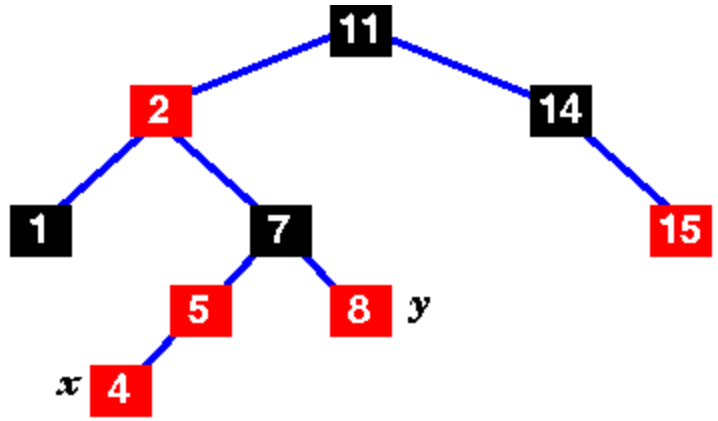

C
A
B
T1
T2
T3
T4
B
A
C
T1
T2
T3
T4
Node that just got inserted or became red via a recoloring lower in the tree
T4 is empty or has a black root
This is a LEFT-RIGHT-ROTATION; the right-left rotation is just a mirror image

Insertion Example

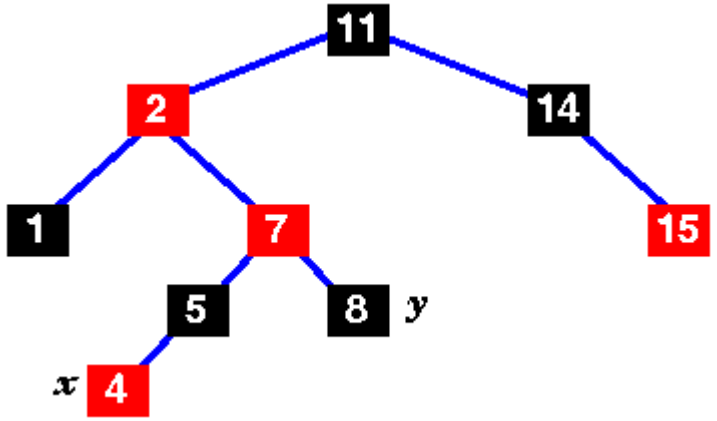


Assume this is the original tree.

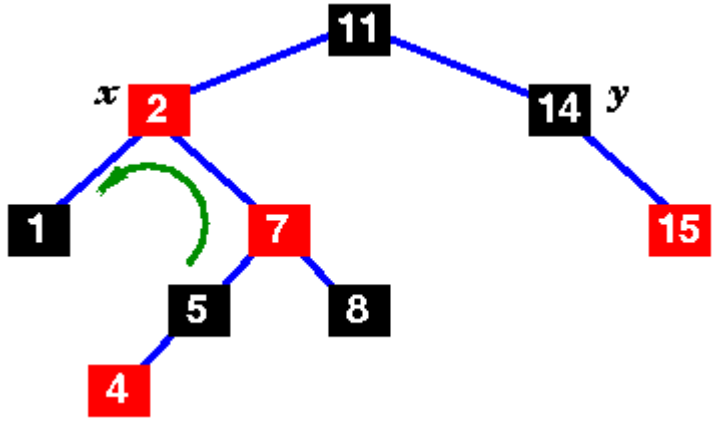
Insert 4 to this Tree



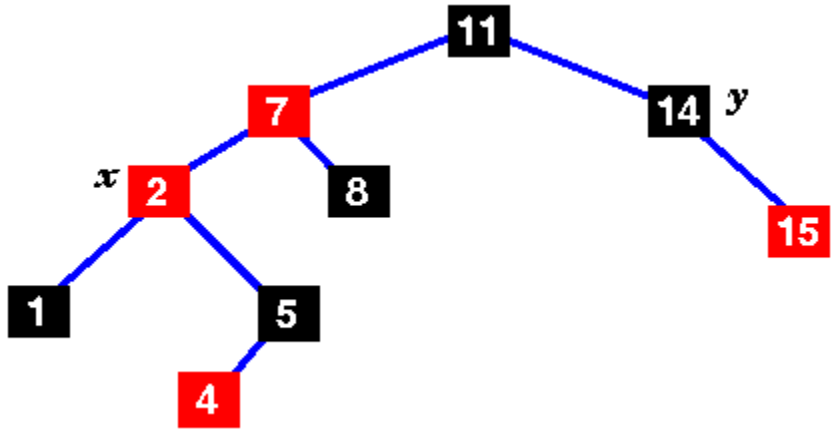
This causes consecutive red nodes. Hence recolor



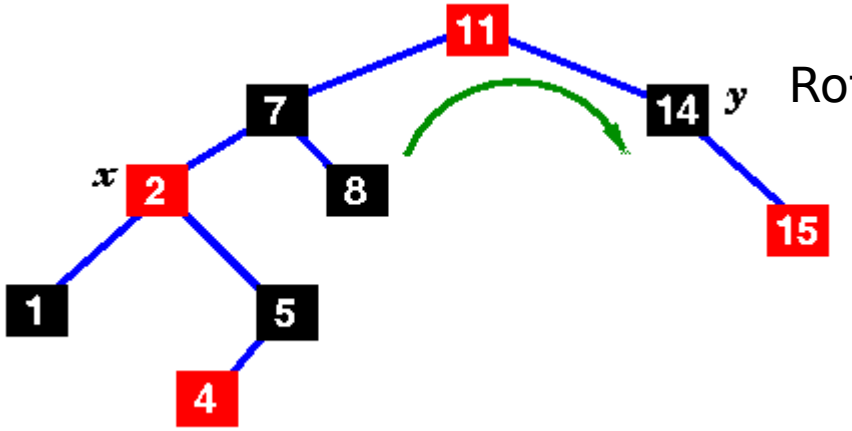
Recolor node 5 and 8. This also violates Red – Black tree property. Node 2 and Node 7 are both red. (Continuous red nodes not allowed)



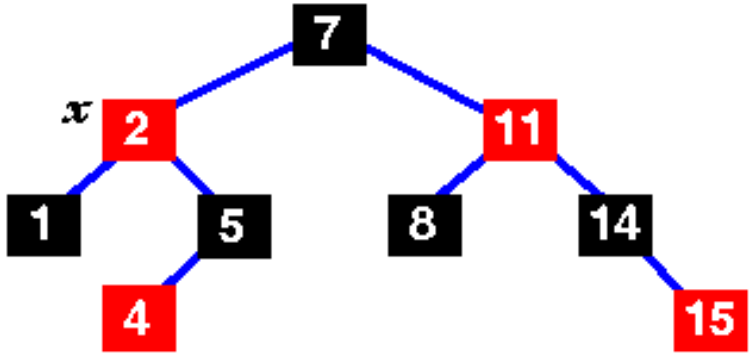
Rotate left



This still violates Red Black tree property.
So recolor nodes and perform rotation again



Rotation about node 11.



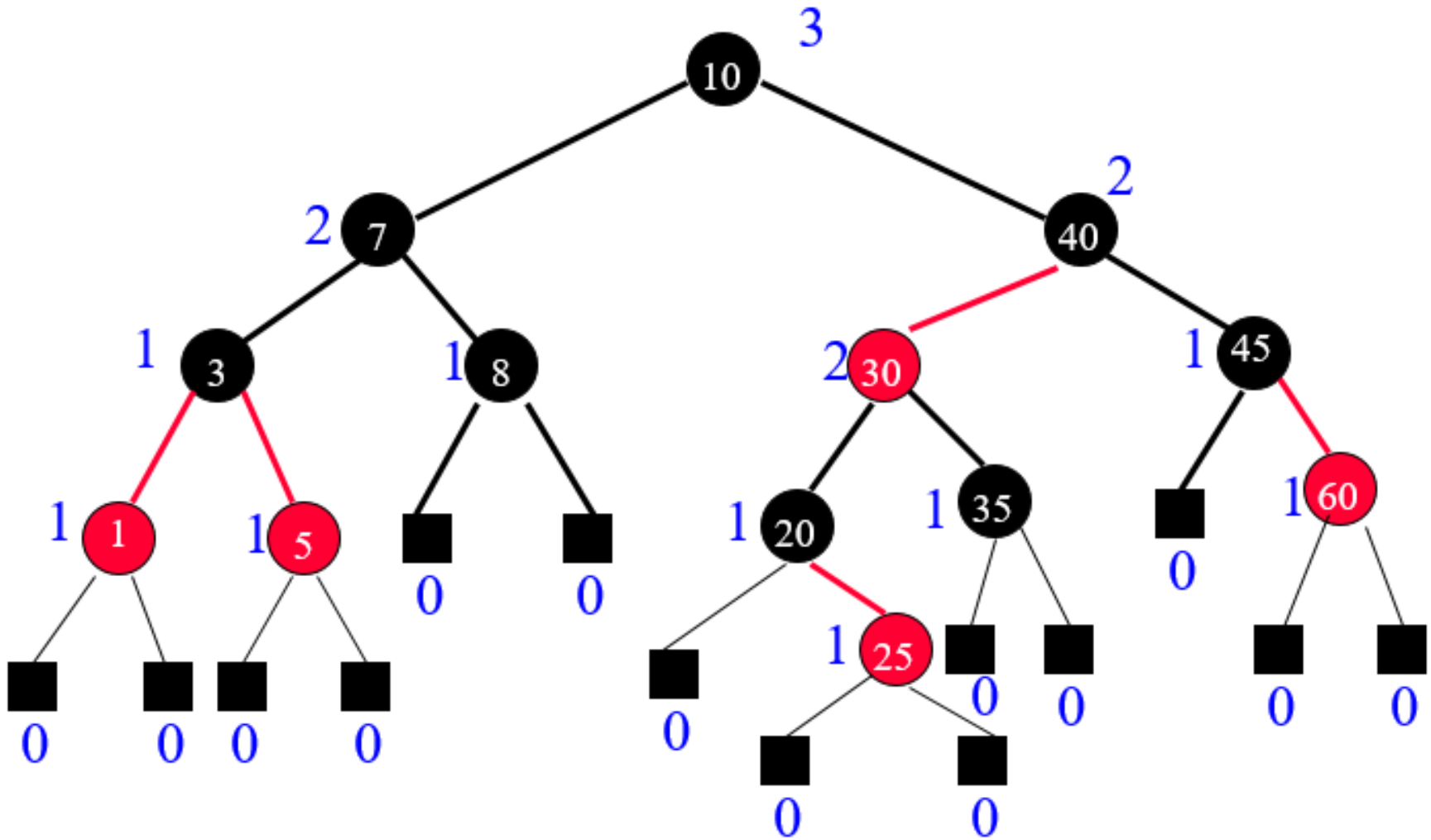
Final Reb Black tree with node 4 inserted.

Red-Black Trees

- rank(x) = # black pointers on path from x to an external node.
- Same as #black nodes (excluding x) from x to an external node.
- rank(external node) = 0.

- rank(x) = 0 for x an external node.
- rank(x) = 1 for x parent of external node.
- p(x) exists rank(x) <= rank(p(x)) <= rank(x) + 1.
- g(x) exists rank(x) < rank(g(x)).

An Example

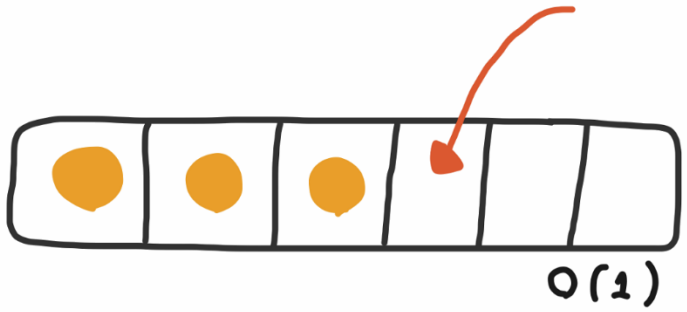

3
10
2
7
2
40
1
3
1
8
2
30
1
45
1
1
1
5
0
0
1
20
1
35
0
1
60
0
0
0
0
0
1
25
0
0
0
0
0
0
0
0

Amortized Complexity

- In an ***amortized analysis***, we average the time required to perform a sequence of datastructure operations over all the operations performed.
- With amortized analysis, we can show that the average cost of an operation is small, if we average over a sequence of operations, even though a single operation within the sequence might be expensive.
- Amortized analysis differs from average-case analysis in that probability is not involved; an amortized analysis guarantees the *average performance of each operation in the worst case*.
- Three most common techniques used in amortized analysis are : Aggregate, Accounting and Potential methods

Amortized Complexity

- The amortized cost of n operations is the total cost of the operations divided by n.
- Example: When the maximum size of array has reached, to insert a new element, create a new array with double the size.
- **The insertion takes O(2X) w**hen the capacity of X has been reached, **and the amortized time for each insertion is O(1).**



Splay Tree

- Splay Tree is a self - adjusted Binary Search Tree in which every operation on element rearranges the tree so that the element is placed at the root position of the tree
- In a splay tree, every operation is performed at the root of the tree. All the operations in splay tree are involved with a common operation called **"Splaying"**.
- By splaying elements we bring more frequently used elements closer to the root of the tree so that any operation on those elements is performed quickly

Splay Trees

- **Every Splay tree must be a binary search tree but it is need not to be balanced tree.**
- Search, insert, delete, and split have amortized complexity O(log n) & actual complexity O(n).
- Actual and amortized complexity of join is O(1).
- Priority queue and double-ended priority queue versions outperform heaps etc. over a sequence of operations.
- Two varieties.
  - Bottom up.
  - Top down.

Bottom-Up Splay Trees

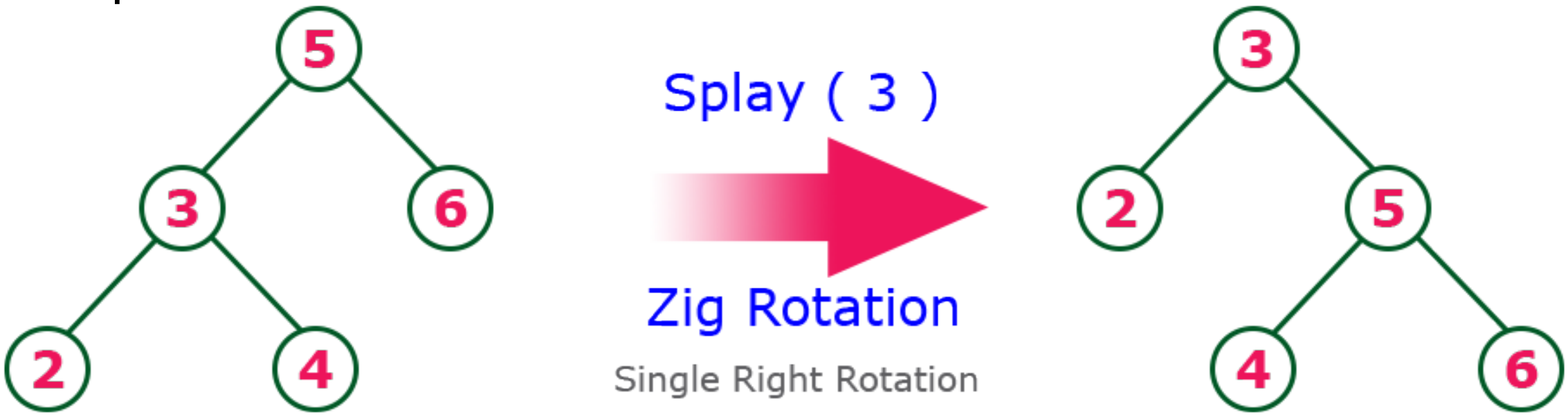
- Search, insert, delete, and join are done as in an unbalanced binary search tree.
- Search, insert, and delete are followed by a splay operation that begins at a splay node.
- When the splay operation completes, the splay node has become the tree root.
- Join requires no splay (or, a null splay is done).
- For the split operation, the splay is done in the middle (rather than end) of the operation.

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 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.

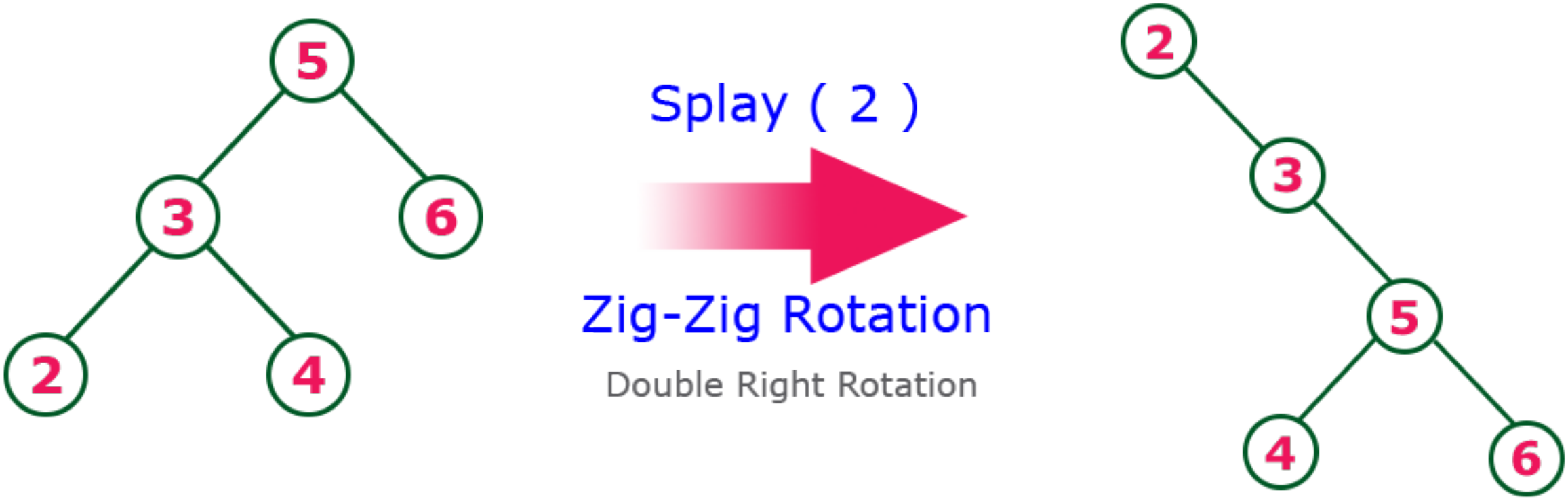


Zag Rotation

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

Zig-Zig Rotation

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

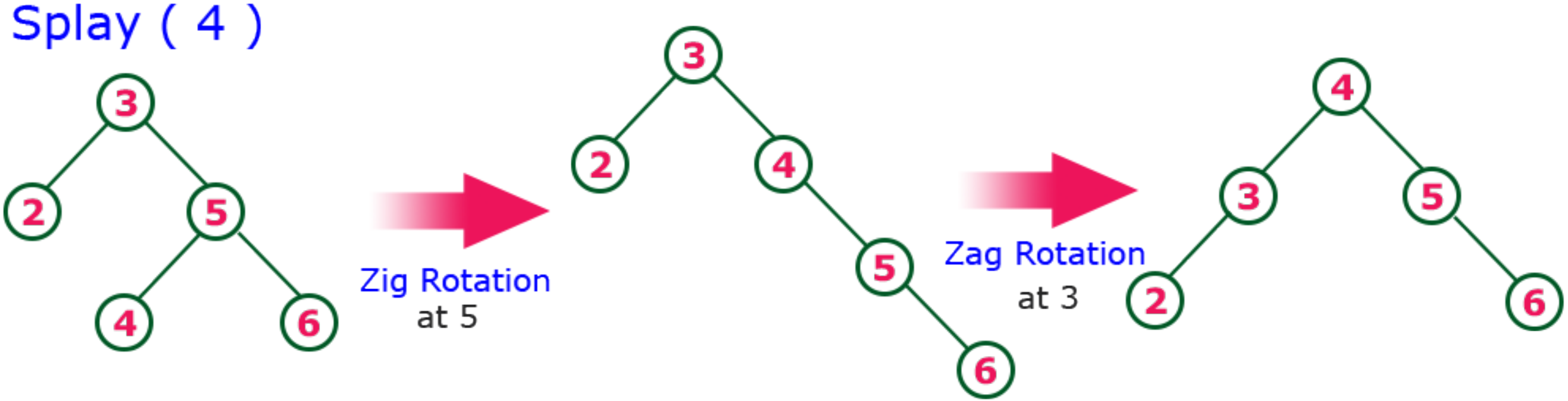


Zag Zag Rotation

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

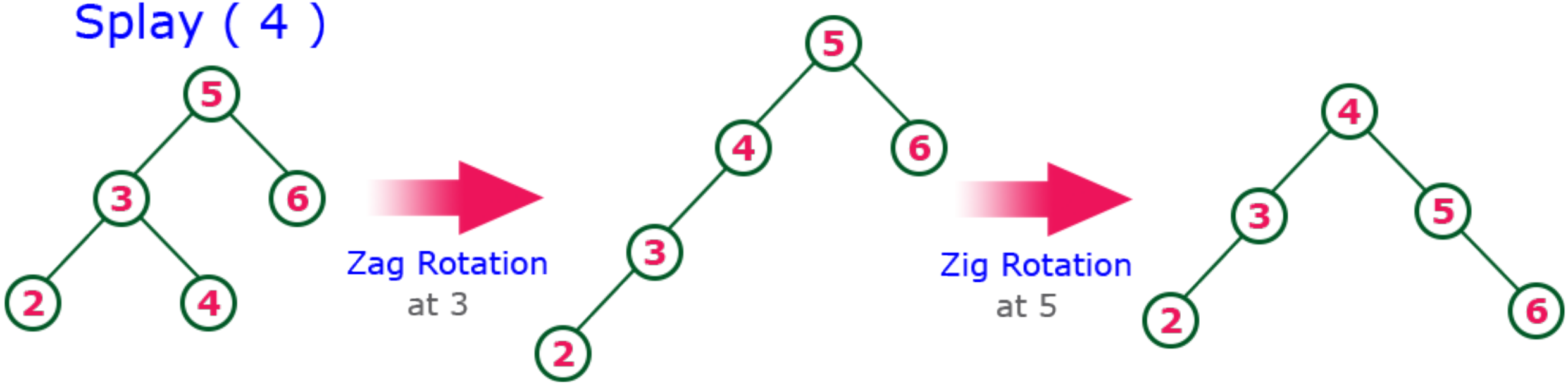
Zig – Zag Rotation

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



Zag – Zig Rotation

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



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 leaf node using Binary Search tree insertion logic.

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

Deletion Operation in Splay Tree

The deletion operation in splay tree is similar to deletion operation in Binary Search Tree. But before deleting the element, we first need to **splay** that element and then delete it from the root position. Finally join the remaining tree using binary search tree logic.

Comparison

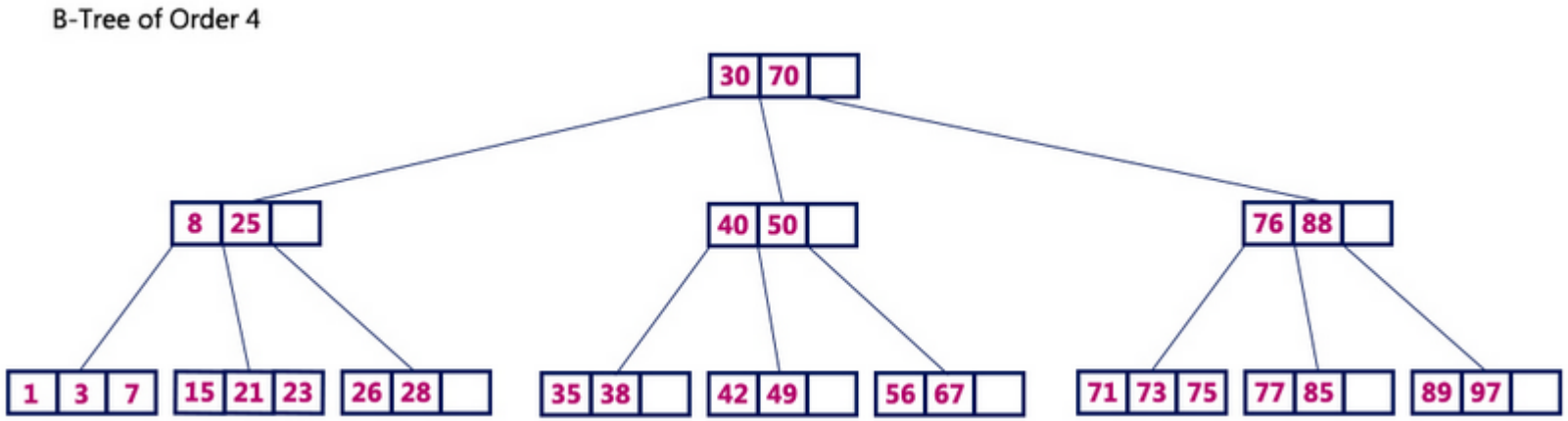
Metric	RB Tree	AVL Tree	Splay Tree
Insertion in worst case	**O(1)**	**O(logn)**	**Amortized O(logn)**
Maximum height of tree	**2*log(n)**	**1.44*log(n)**	**O(n)**
Search in worst case	**O(logn), Moderate**	**O(logn), Faster**	**Amortized O(logn), Slower**
Efficient Implementation requires	**Three pointers with color bit per node**	**Two pointers with balance factor per node**	**Only two pointers with no extra information**
Deletion in worst case	**O(logn)**	**O(logn)**	**Amortized O(logn)**
Mostly used	**As universal data structure**	**When frequent lookups are required**	**When same element is retrieved again and again**
Real world Application	**Multiset, Multimap, Map, Set, etc.**	**Database Transactions**	**Cache implementation, Garbage collection Algorithms**

B - Tree

- B-Tree is m way search tree
- It is a self-balanced search tree in which every node contains multiple keys and has more than two children.
- Large degree B-trees used to represent very large dictionaries that reside on disk.
- Smaller degree B-trees used for internal-memory dictionaries to overcome cache-miss penalties

Properties of B-Tree

- **#1** - All **leaf nodes** must be **at same level**.
- **#2** - All nodes except root must have at least **[m/2]-1** keys and maximum of **m-1** keys.
- **#3** - All non leaf nodes except root (i.e. all internal nodes) must have at least **m/2** children.
- **#4** - If the root node is a non leaf node, then it must have **atleast 2** children.
- **#5** - A non leaf node with **n-1** keys must have **n** number of children.
- **#6** - All the **key values in a node** must be in **Ascending Order**.



Operations on a B-Tree

- **Search**
- **Insertion**
- **Deletion**

Search Operation in B-Tree

Step 1 - Read the search element from the user.

Step 2 - Compare the search element with first key value of root node in the tree.

Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function

Step 4 - If both are not matched, then check whether search element is smaller or larger than that key value.

Step 5 - If search element is smaller, then continue the search process in left subtree.

Step 6 - If search element is larger, then compare the search element with next key value in the same node and repeate steps 3, 4, 5 and 6 until we find the exact match or until the search element is compared with last key value in the leaf node.

Step 7 - If the last key value in the leaf node is also not matched then display "Element is not found" and terminate the function.

Insertion Operation in B-Tree

Step 1 - Check whether tree is Empty.

Step 2 - If tree is **Empty**, then create a new node with new key value and insert it into the tree as a root node.

Step 3 - If tree is **Not Empty**, then find the suitable leaf node to which the new key value is added using Binary Search Tree logic.

Step 4 - If that leaf node has empty position, add the new key value to that leaf node in ascending order of key value within the node.

Step 5 - If that leaf node is already full, **split** that leaf node by sending middle value to its parent node. Repeat the same until the sending value is fixed into a node.

Step 6 - If the spilting is performed at root node then the middle value becomes new root node for the tree and the height of the tree is increased by one.

Example Construct a **B-Tree of Order 3** by inserting numbers from 1 to 10.

insert(1)

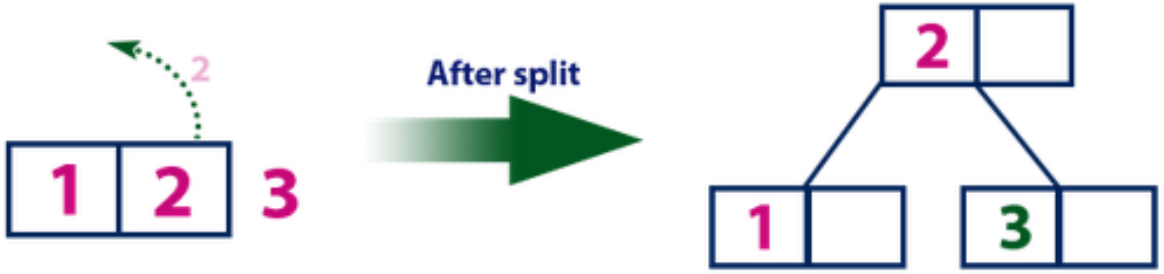
Since '1' is the first element into the tree that is inserted into a new node. It acts as the root node.

insert(2)

Element '2' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node has an empty position. So, new element (2) can be inserted at that empty position.

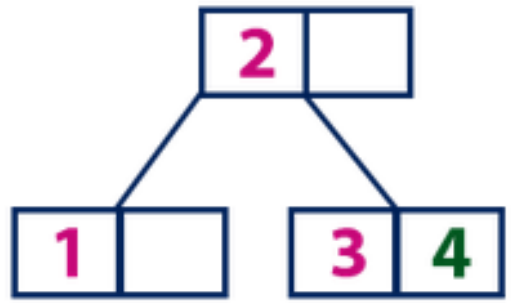
insert(3)

Element '3' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node doesn't has an empty position. So, we split that node by sending middle value (2) to its parent node. But here, this node doesn't has parent. So, this middle value becomes a new root node for the tree.



insert(4)

Element '4' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node with value '3' and it has an empty position. So, new element (4) can be inserted at that empty position.

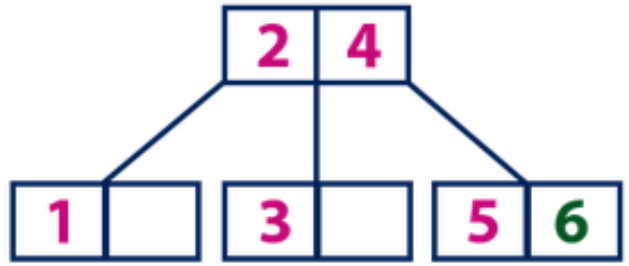


insert(5)

Element '5' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (4) to its parent node (2). There is an empty position in its parent node. So, value '4' is added to node with value '2' and new element '5' added as new leaf node.

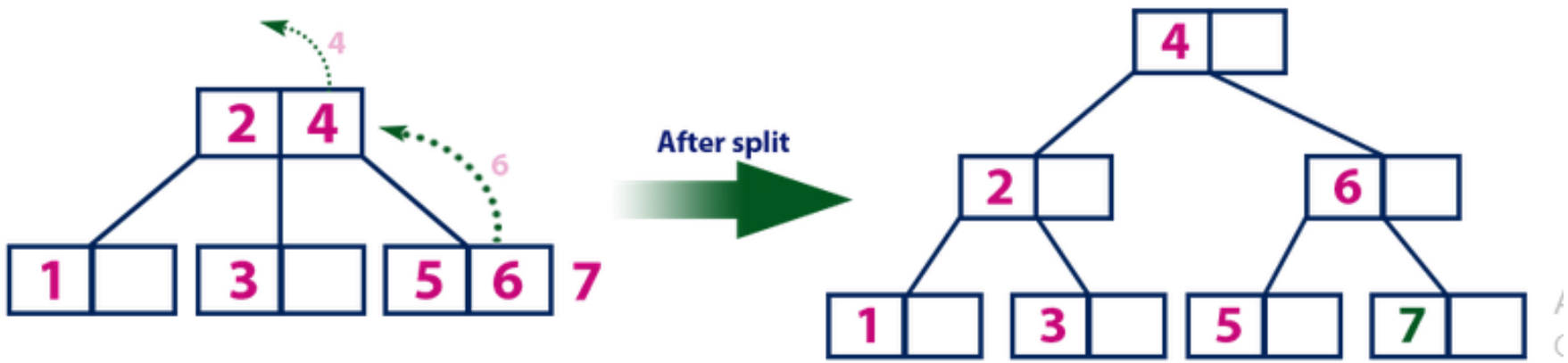
insert(6)

Element '6' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node with value '5' and it has an empty position. So, new element (6) can be inserted at that empty position.



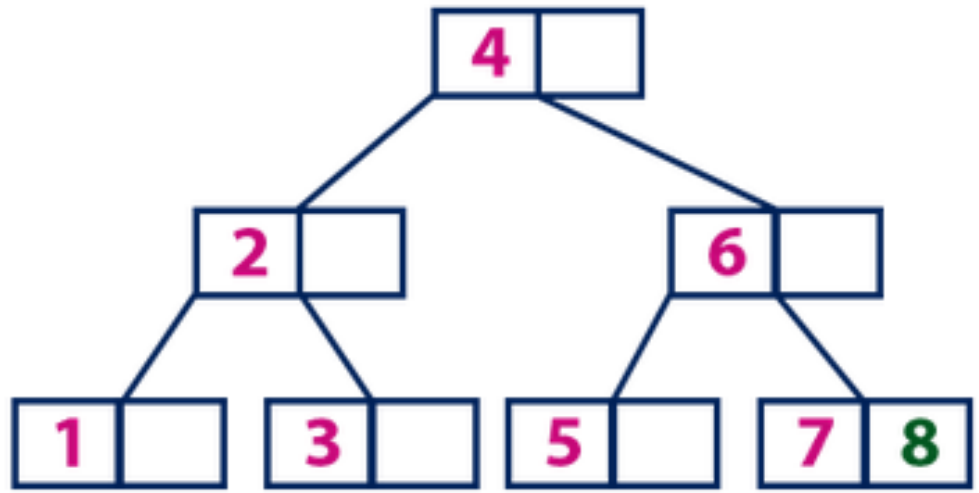
insert(7)

Element '7' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (6) to its parent node (2&4). But the parent (2&4) is also full. So, again we split the node (2&4) by sending middle value '4' to its parent but this node doesn't have parent. So, the element '4' becomes new root node for the tree.



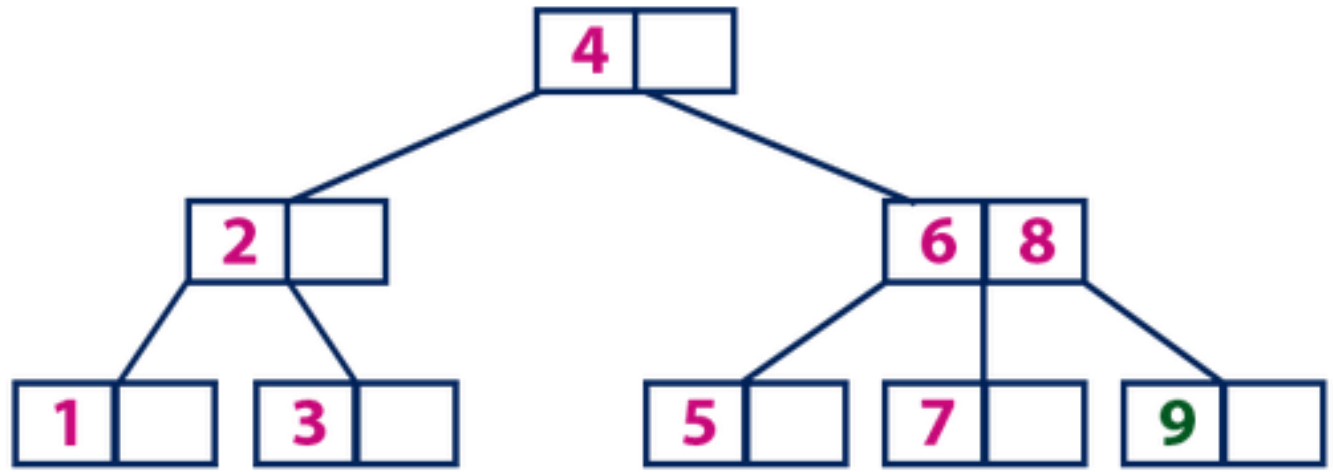
insert(8)

Element '8' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '8' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7) and it has an empty position. So, new element (8) can be inserted at that empty position.



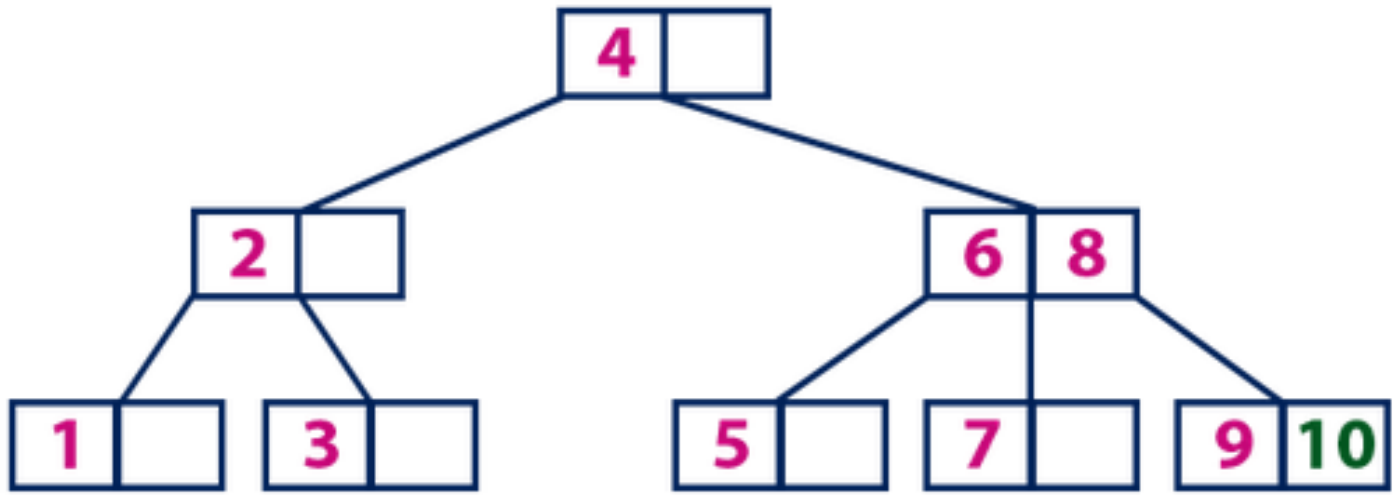
insert(9)

Element '9' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '9' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7 & 8). This leaf node is already full. So, we split this node by sending middle value (8) to its parent node. The parent node (6) has an empty position. So, '8' is added at that position. And new element is added as a new leaf node.



insert(10)

Element '10' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with values '6 & 8 '. '10' is larger than '6 & 8' and it is also not a leaf node. So, we move to the right of '8'. We reach to a leaf node (9). This leaf node has an empty position. So, new element '10' is added at that empty position.



Comparison

Basis for comparison	B-tree	Binary tree
Essential constraint	A node can have at max M number of child nodes(where M is the order of the tree).	A node can have at max 2 number of subtrees.
Used	It is used when data is stored on disk.	It is used when records and data are stored in RAM.
Height of the tree	$\log_M N$ (where m is the order of the M-way tree)	$\log_2 N$
Application	Code indexing data structure in many DBMS.	Code optimization, Huffman coding, etc

opics Discussed So Far

- Asymptotic Notations
- Array – Row and Column Major
- Linked List – Singly, Doubly, Circular
- Matrices – Special matrices
- Stacks - LIFO
- Queues – FIFO
- Infix, Postfix, Prefix expression
- Skip Lists
- Hashing
- Trees
  - Binary – Tree Traversal (Inorder, Postorder and Preorder)
  - Binary Search Tree
  - Balanced Tree - AVL Tree, Red Black Tree, Splay Tree, B-Tree
  - Heaps
  - Leftist Tree – HBLT, WBLT
- Priority queue
- Breadth First and Depth First Search algorithms
- Insertion Sort and Heap Sort mechanisms

Most common operations on each ADT

- Creation
- Insertion
- Find / Search
- Deletion
- Size / Empty/ Full

Time Complexity of each operation