Data Structures and Algorithms I SCS1201 - CS

Dr. Dinuni Fernando

Senior Lecturer

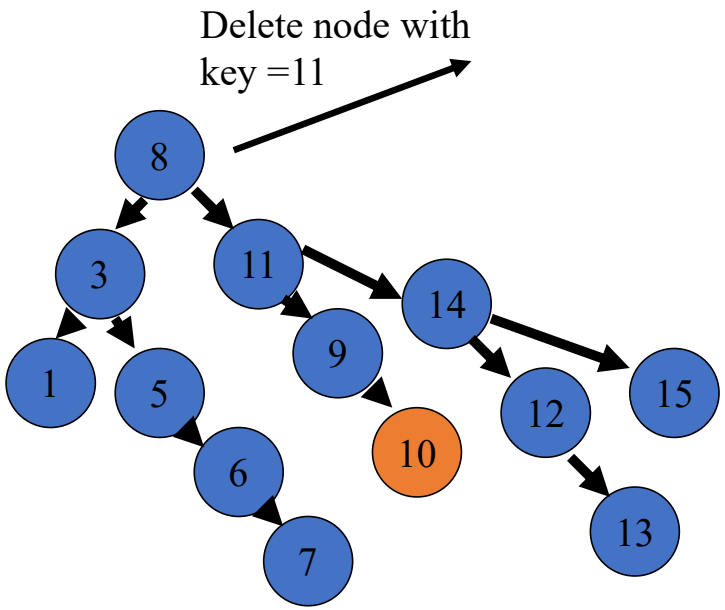
Lecture 9

UCSC

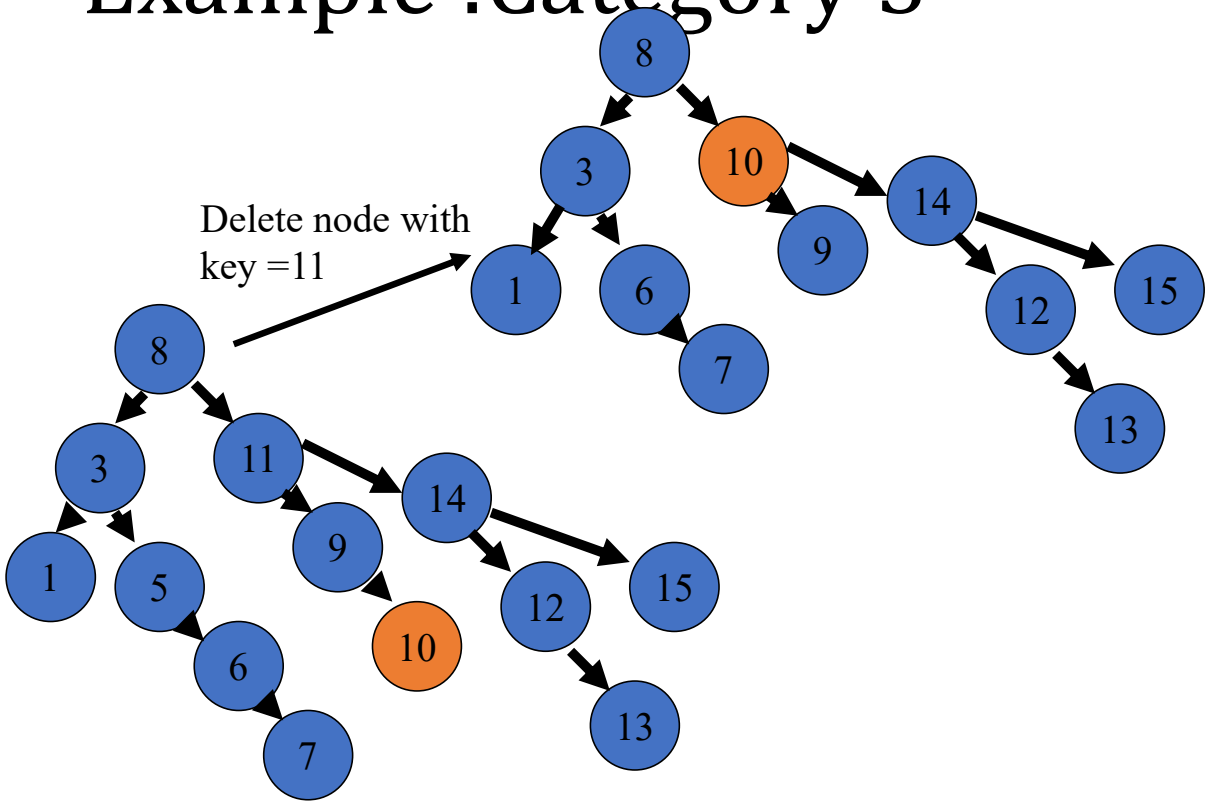
Category 3 : The node to be deleted has two sub-trees.

- Solution : this method we shall use is to replace the node being deleted by the rightmost node in its left sub-tree
- Or Leftmost node in its right-sub-tree.

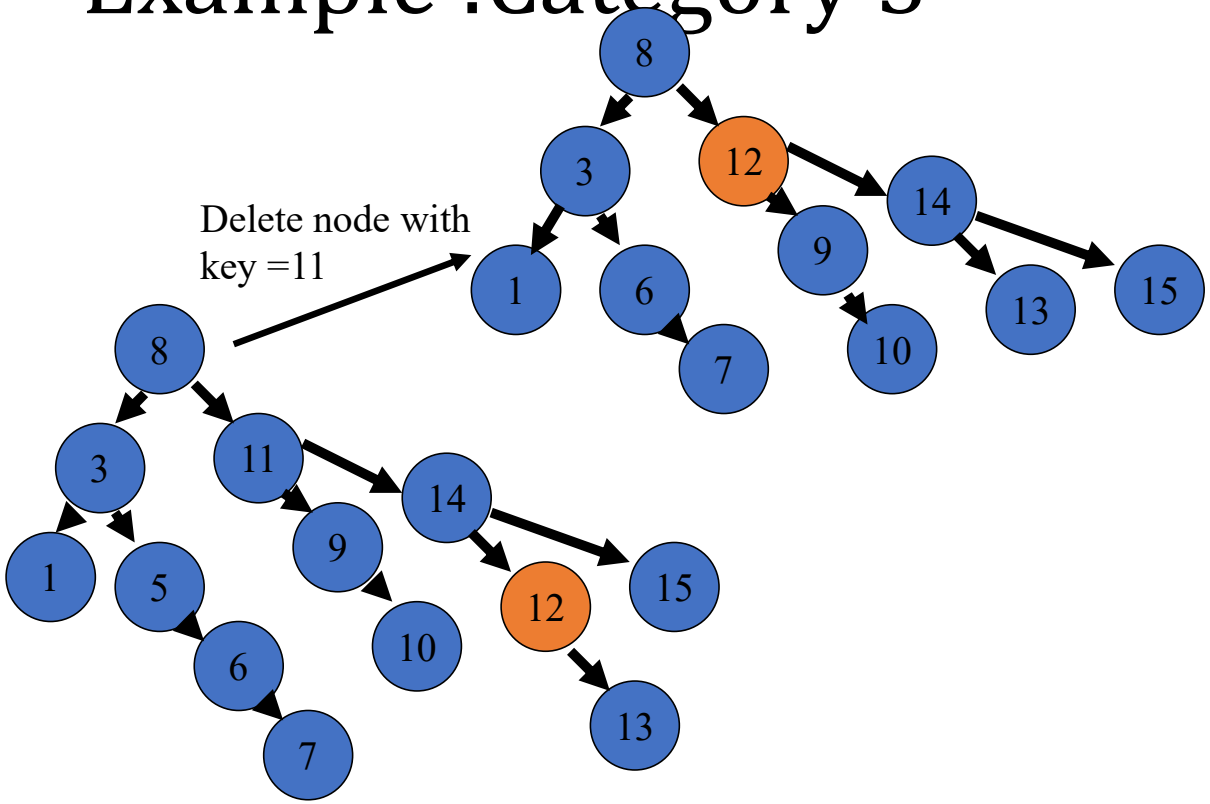
Example :Category 3



Example :Category 3



Example :Category 3



Algorithm to delete a node from a binary search tree

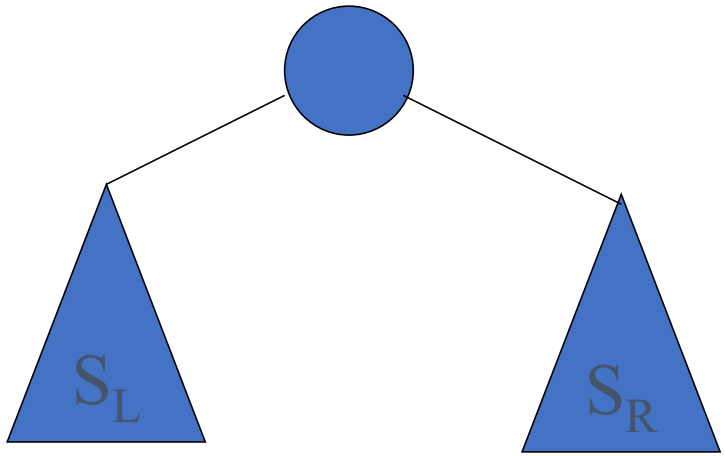
```
Delete (TREE, VAL)

Step 1: IF TREE = NULL
          Write "VAL not found in the tree"
        ELSE IF VAL < TREE -> DATA
          Delete(TREE->LEFT, VAL)
        ELSE IF VAL > TREE -> DATA
          Delete(TREE -> RIGHT, VAL)
        ELSE IF TREE -> LEFT AND TREE -> RIGHT
          SET TEMP = findLargestNode(TREE -> LEFT)
          SET TREE -> DATA = TEMP -> DATA
          Delete(TREE -> LEFT, TEMP -> DATA)
        ELSE
          SET TEMP = TREE
          IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
              SET TREE = NULL
          ELSE IF TREE -> LEFT != NULL
               SET TREE = TREE -> LEFT
          ELSE
               SET TREE = TREE -> RIGHT
          [END OF IF]
          FREE TEMP
        [END OF IF]
Step 2: END
```

Determining the Height of a Binary Search Tree

- In order to determine the height of a binary search tree, we calculate the height of the left sub-tree and the right sub-tree.

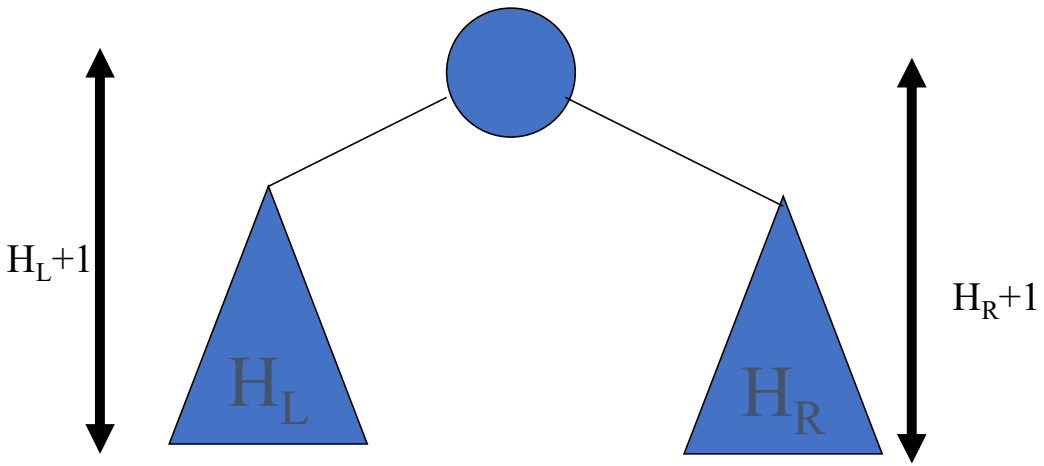
Recursive view used to calculate the size of a tree : $S_T=S_L+S_R+1$



Routine to compute the size of a node

```
//* Return the size of the binary tree rooted at t.
if  (t = = null)
 {
        return 0
else
        return 1+size(t->left)+ size(t->right)
}
```

Recursive view used to calculate the height of a tree : $H_T = \max(H_L + 1, H_R + 1)$



Routine to compute the height of a node

```
//* Return the height of the binary tree rooted at t.

If (t = = null) {
        return 0;
else
        return 1+maximum(height(t->left),height(t->right);
}
```

Other operations

- We can perform, find operation by starting at the root and then repeatedly branching either left or right, depending result of a comparison.

Findmin and findmax

- To perform a findmin, we start at root, and repeatedly branch left as long as there is a left child. The stopping point is the smallest element.
- Findmax routine is same, except that branching is to right child.

Findmin operation

```
If (t = = null)
        return null;
else if (t->left= = null)
        return t;
return findMin(t->left);
```

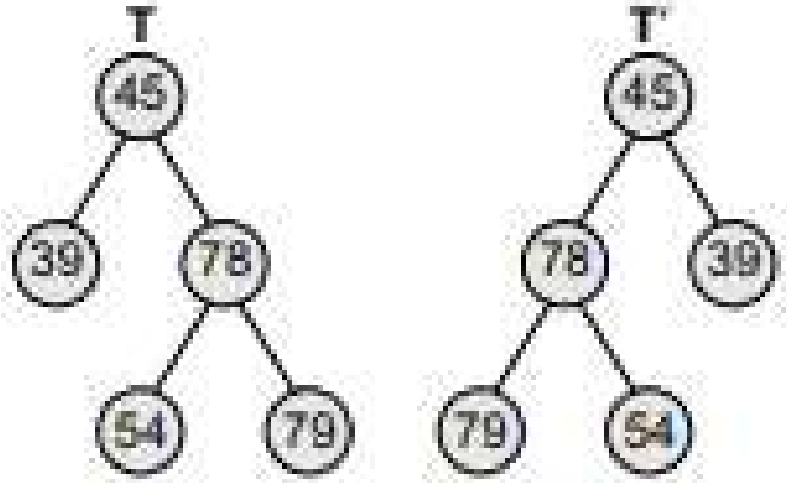
findMax(Non recursive)

```
{
if (t!=null)
        While (t->right!=null)
                t=t->right;
return t;
}
```

Finding the Mirror Image of a Binary Search Tree

- Mirror image of a binary search tree is obtained by interchanging the left sub-tree with the right sub-tree at every node of the tree.
- For example, given a tree T the mirror image of T can be obtained as T'.

Consider the following tree.



Finding the Mirror Image of a Binary Search Tree (cont'd)

- A recursive algorithm more convenient to obtain the mirror image of a binary search tree.
- In the algorithm, if TREE != NULL, that is if the current node in the tree has one or more nodes, then the algorithm is recursively called at every node in the tree to swap the nodes in its left and right sub-trees.

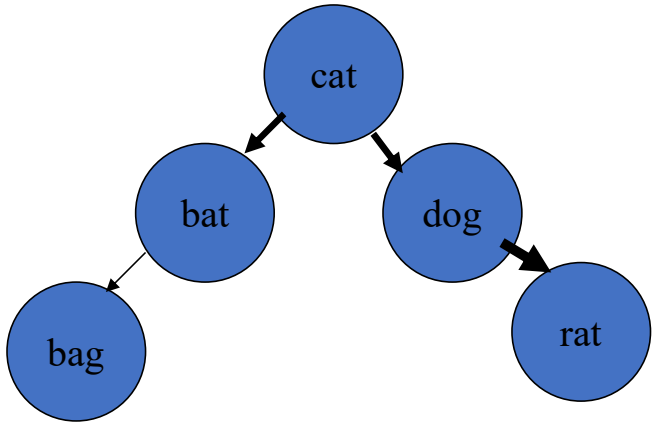
Finding the Mirror Image of a Binary Search Tree (cont'd)

```
MirrorImage(TREE)

Step 1: IF TREE != NULL
            MirrorImage(TREE -> LEFT)
            MirrorImage(TREE -> RIGHT)
            SET TEMP = TREE -> LEFT
            SET TREE -> LEFT = TREE -> RIGHT
            SET TREE -> RIGHT = TEMP
        [END OF IF]
Step 2: END
```

Application : Implementation of a Dictionary [words]

- Eg : cat, bat, dog,rat,bag etc..



Homework

- Write a program to create a binary search tree and perform all the operations discussed in the preceding sections.
  - FindHeight()
  - FindSize()
  - findMin()
  - findMax()
  - MirrorImage()

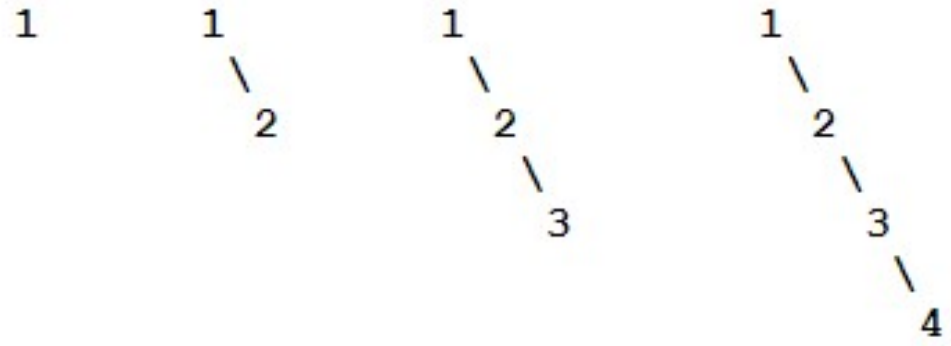
Balanced Binary Search Trees

Balanced Binary Search Trees

- Binary search trees allow **binary search** for fast lookup, insertion and deletion of data items, and can be used to implement **dynamic sets** and **lookup tables**.
- The order of nodes in a BST means that each comparison skips about half of the remaining tree, so the whole lookup takes time proportional to the binary logarithm of the number of items stored in the tree.

Balanced Binary Search Trees cont..

- Binary search trees are a nice idea, but they fail to accomplish our goal of doing lookup, insertion and deletion each in time $O(\log_2(n))$, when there are n items in the tree.
- Imagine starting with an empty tree and inserting 1, 2, 3 and 4, in that order.



Balanced Binary Search Trees (cont'd)

- You do not get a branching tree, but a linear tree.
- All of the left subtrees are empty. Because of this behavior, *in the worst case* each of the operations (lookup, insertion and deletion) takes time $\Theta(n)$.
- From the perspective of the worst case, we might as well be using a linked list and linear search.
- That bad worst-case behavior can be avoided by using an idea called *height balancing*, sometimes called *AVL trees*.

What are Self-Balancing Binary Search Trees?

- A self-balancing binary search tree (BST) is a binary search tree that automatically tries to keep **its height as minimal as possible at all times** (even after performing operations such as insertions or deletions).
- Hence having the height as small as possible is better when it comes to performing a large number of operations. Hence, self-balancing BSTs were introduced which automatically maintain the height at a minimum.
- However, you may think having to self-balance every time an operation is performed is not efficient, but this is compensated by ensuring a large number of fast operations which will be performed later on the BST.

What are Self-Balancing Binary Search Trees?

A binary tree with height h can have at most $2^0 + 2^1 + \cdots + 2^h = 2^{h+1} - 1$ nodes.

$$n \leq 2^{(h+1)}\text{-}1$$

$$h \geq |log_2(n+1)| - 1 \geq |log_2(n)|$$

- Hence, for self-balancing BSTs, the minimum height must always be $\boldsymbol{log_2(n)}$ rounded down. Moreover, a binary tree is said to be **balanced** if the height of left and right children of every node differ by either **-1**, **0** or **+1**. This value is known as the **balance factor**.

> *Balance factor = Height of the left subtree - Height of the right subtree*

How do Self-Balancing Binary Search Trees Balance?

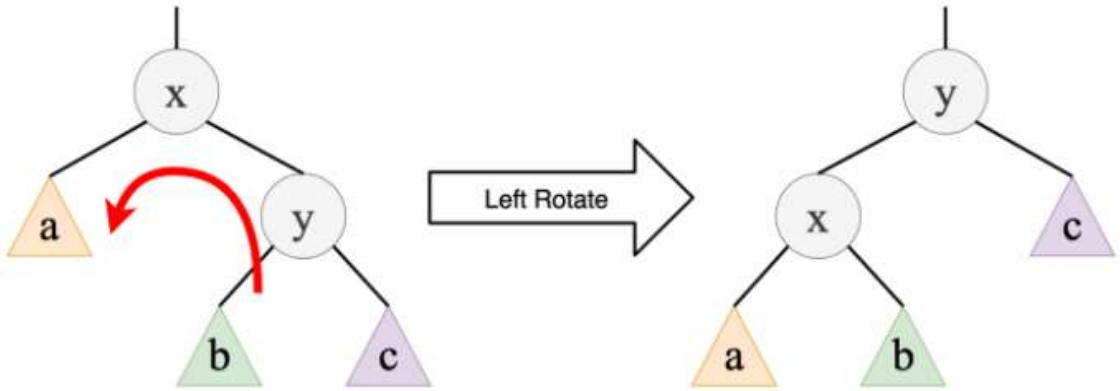
When it comes to self-balancing, BSTs perform **rotations** after performing insert and delete operations.

Given below are the two types of rotation operations that can be performed to balance BSTs without violating the binary-search-tree property.

How do Self-Balancing Binary Search Trees Balance?

1. Left rotation

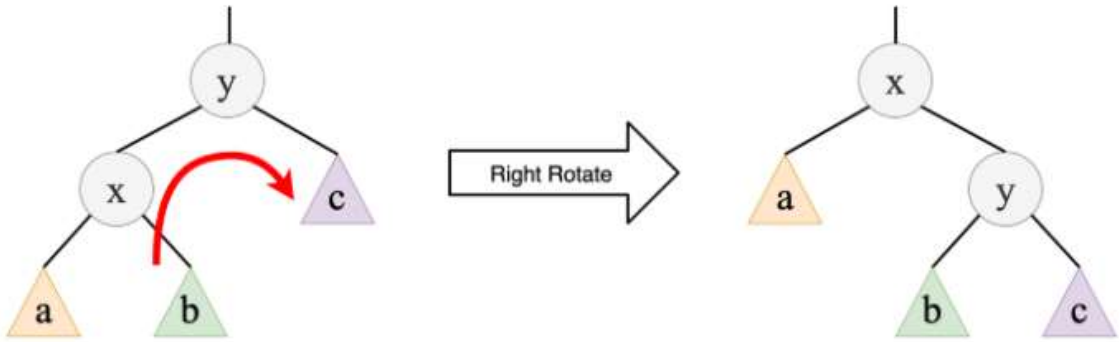
When we left rotate about node *x*, node *y* becomes the new root of the subtree. Node *x* becomes the left child of node *y* and subtree *b* becomes the right child of node *x*.



How do Self-Balancing Binary Search Trees Balance?

2. Right rotation

When we right rotate about node *y*, node *x* becomes the new root of the subtree. Node *y* becomes the right child of node *x* and subtree *b* becomes the left child of node *y*.



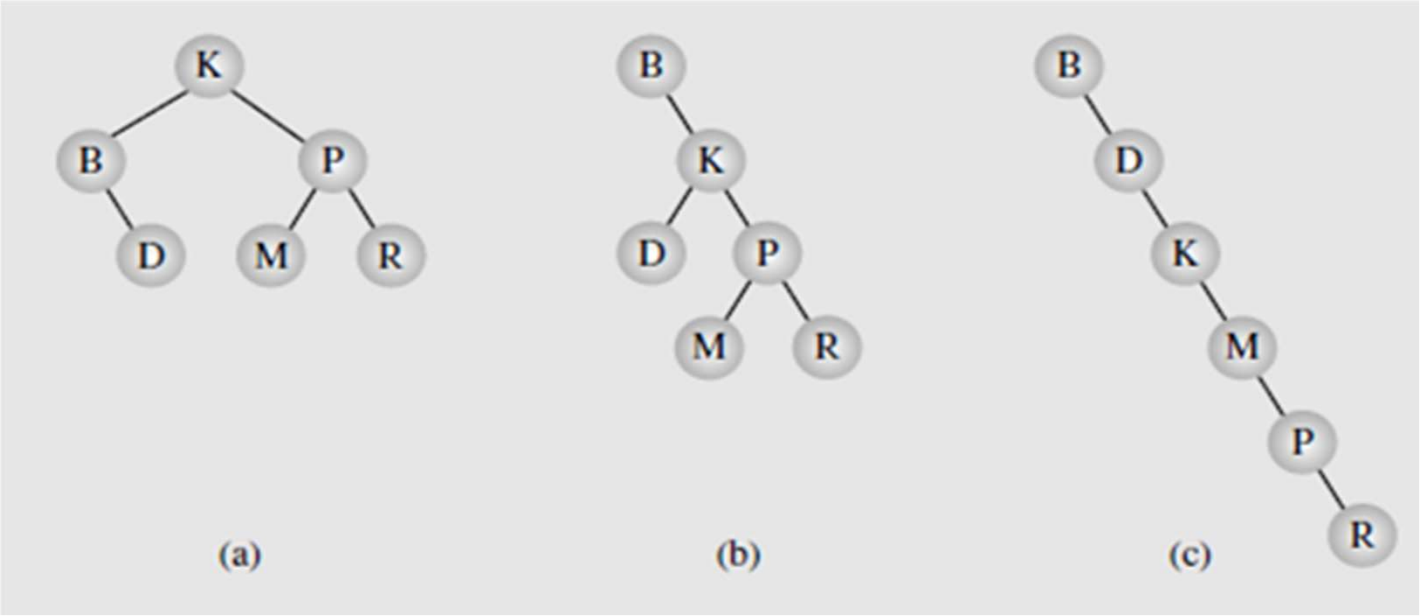
Types of Self-Balancing Binary Search Trees

Given below are a few types of BSTs that are self-balancing trees.

1. AVL trees
2. Red-black trees
3. Splay trees
4. Treaps (A Randomized Binary Search Tree)

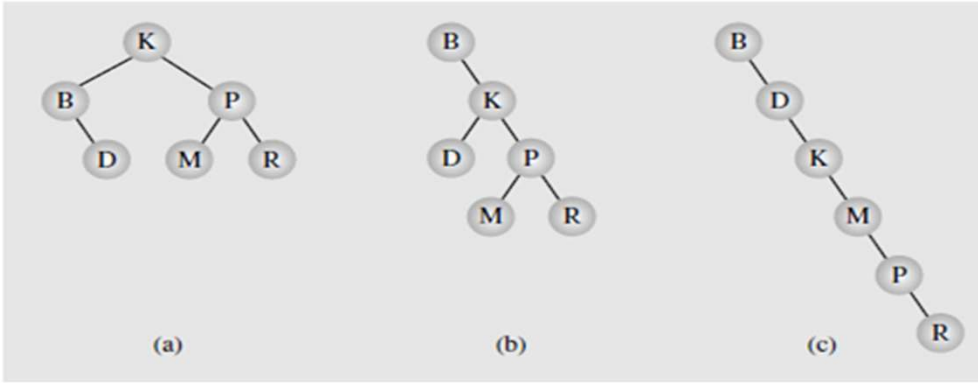
More detail about the tree balancing

- Consider the following three examples.



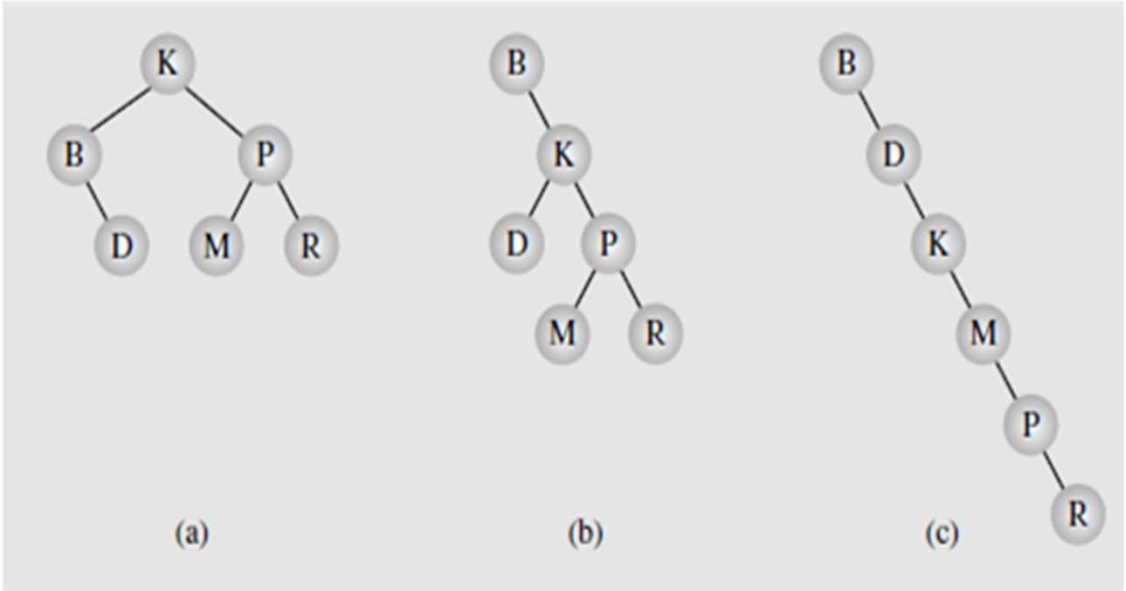
More detail about the tree balancing

- A binary tree is height-balanced or simply balanced if the difference in height of **both sub trees of any node in the tree is either zero or one.**
- For example, for node K in Figure, the difference between the heights of its sub trees being equal to one is acceptable. But for node P this difference is 3, which means that the entire tree is unbalanced.



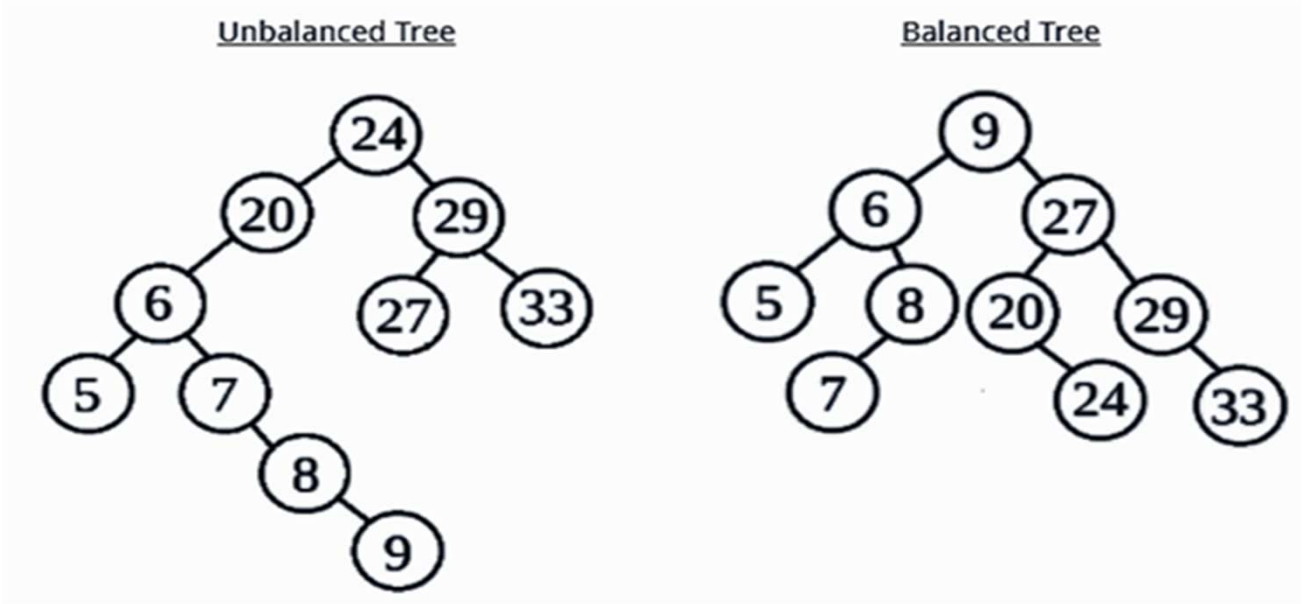
More detail about the tree balancing

- a tree is considered perfectly balanced if it is balanced and all leaves are to be found on one level or two levels.



More detail about the tree balancing

- Examples



Global tree balancing

- **Balancing a tree**
- We have seen the searching benefits of a balance tree, but how to we have a balance a tree.
- Consider a tree with 10,000 nodes.
- At its least efficient (linked list) at most 10,000 elements need to be tested find the element.
- When balance, just 14 tests need to be performed
- The height of the tree is lg(10,000)=13.2877123795=approx.. 14.

Creating a balanced tree

Assume all the data is in a stored array.

- The middle element becomes the root.
- The middle element of one half becomes a child.
- The idle element of the other half becomes the other child.
- Then recursively add elements.
- E.g. 1,2,3,4,5,6,7 (elements stored in an array).

How to balance?

```
balance (Data[], int first, int last){
If (first<=last) {
        int middle=(first+last)/2;
        insert(data[middle]);
        balance(data, first,middle-1);
        balance (data,middle+1, last);
   }
}
```

Weakness of this method

- This algorithm is quite inefficient as it relies on using extra space to store an array of values, and all values must be in this array(perhaps by in-order traversal).
- Let's look at some alternative.

Global tree balancing -The DSW (Day-Stout-Warren) **algorithm**

- The very elegant DSW algorithm was devised by Colin **Day** and later improved by Quentin F. **Stout** and Bette L.**Warren**.
- In this algorithm, first the tree is stretched into a linked list like tree. Then is it balanced?.
- The building block for tree transformations in this algorithm is the rotation.
- The key to the operation comes from a rotation function, where a child is rotated around its parent.
- There are two types of rotation, left and right, which are symmetrical to one another.

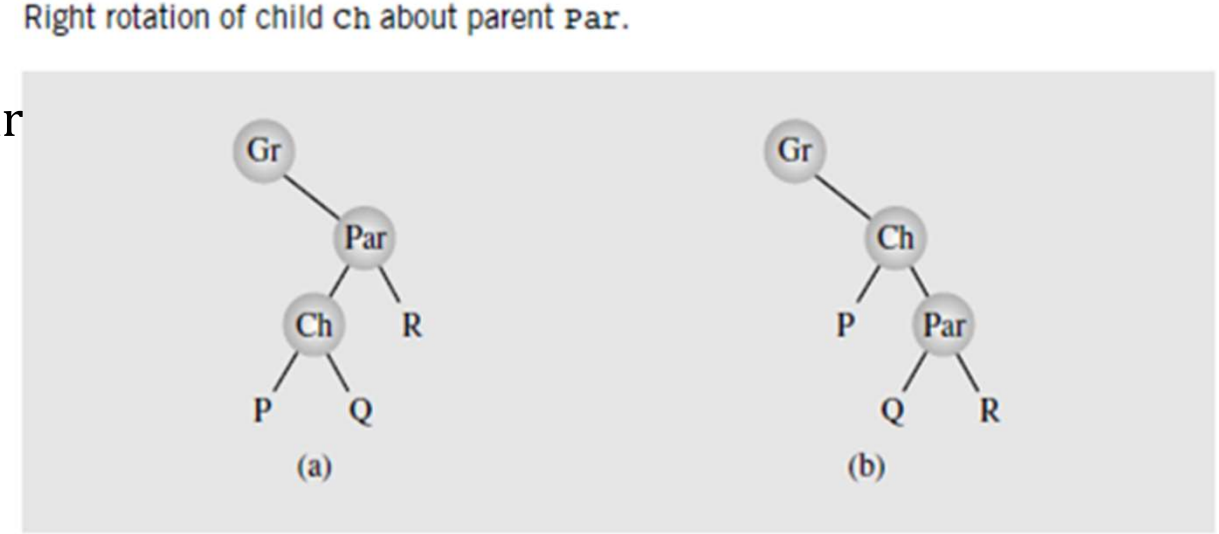
Global tree balancing -The DSW algorithm

- So, the idea is to take a tree and perform some rotations to it to make the balanced.
- First create a backbone or vine.
- Then you transform the backbone into nicely balance tree.

Global tree balancing -The DSW algorithm

- Tree balancing - the steps involved in this compound operation are shown in the image below.

The right rotation of the node Ch about its parent Par is performed according to the following tree:

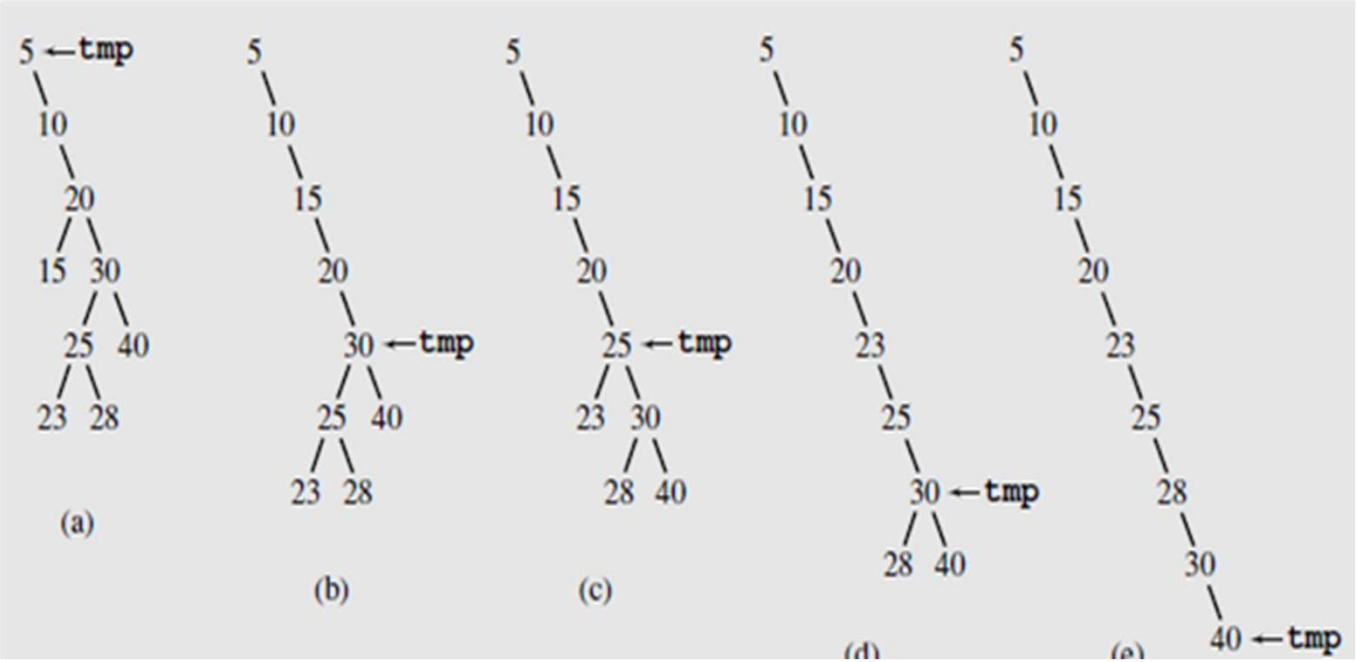


- createBackbone(root, n)
  - Tmp = root
  - While (Tmp != 0)
    - If Tmp has a left child
      - Rotate this child about Tmp
      - Set Tmp to the child which just became parent
    - Else set Tmp to its right child

- createPerfectTree(n)
  - M = $2^{\text{floor}[\lg(n+1)]}-1$;
  - Make n-M rotations starting from the top of the backbone;
  - While (M > 1)
    - M = M/2;
    - Make M rotations starting from the top of the backbone;

Global tree balancing -The DSW algorithm

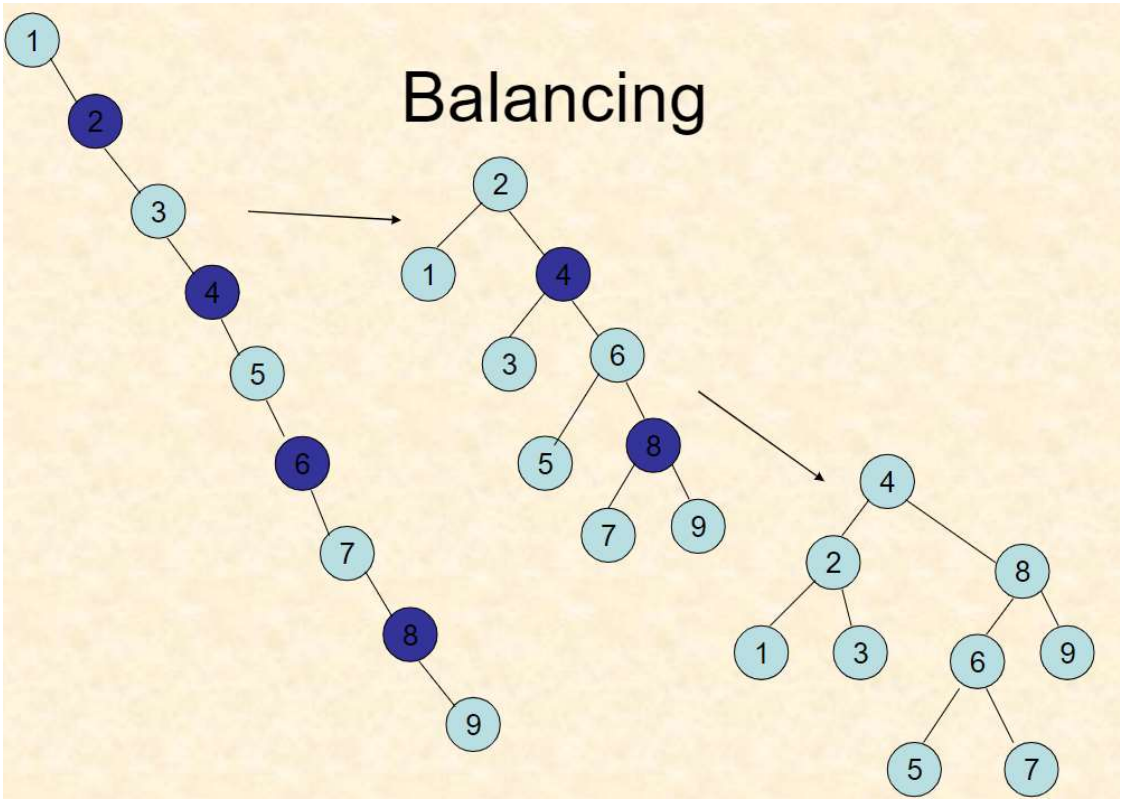
- Phase 1 : Transforming a binary search tree into a backbone.



Global tree balancing -The DSW algorithm

- Phase 2 – turn the lined list like tree into a perfectly balanced tree, once again using the rotation function.
- Here every other node is rotated around its parent.
- This process is repeated down the right branch until a balanced tree is reached.

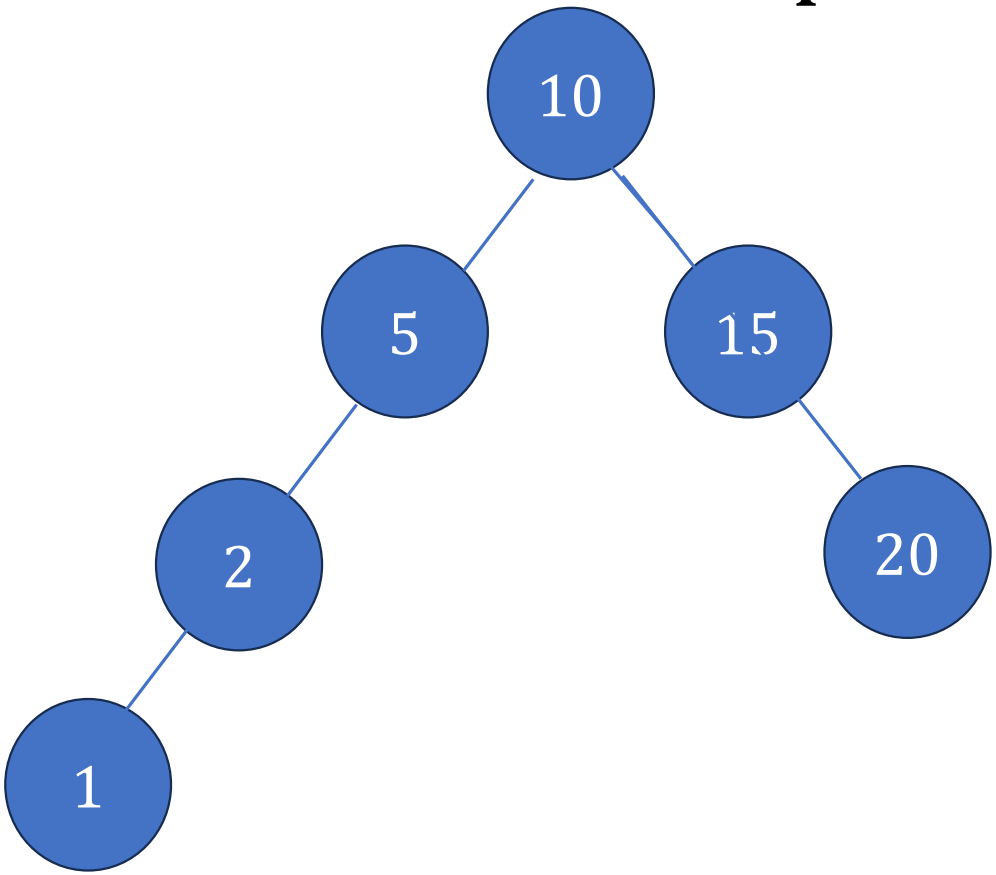
Global tree balancing -The DSW algorithm



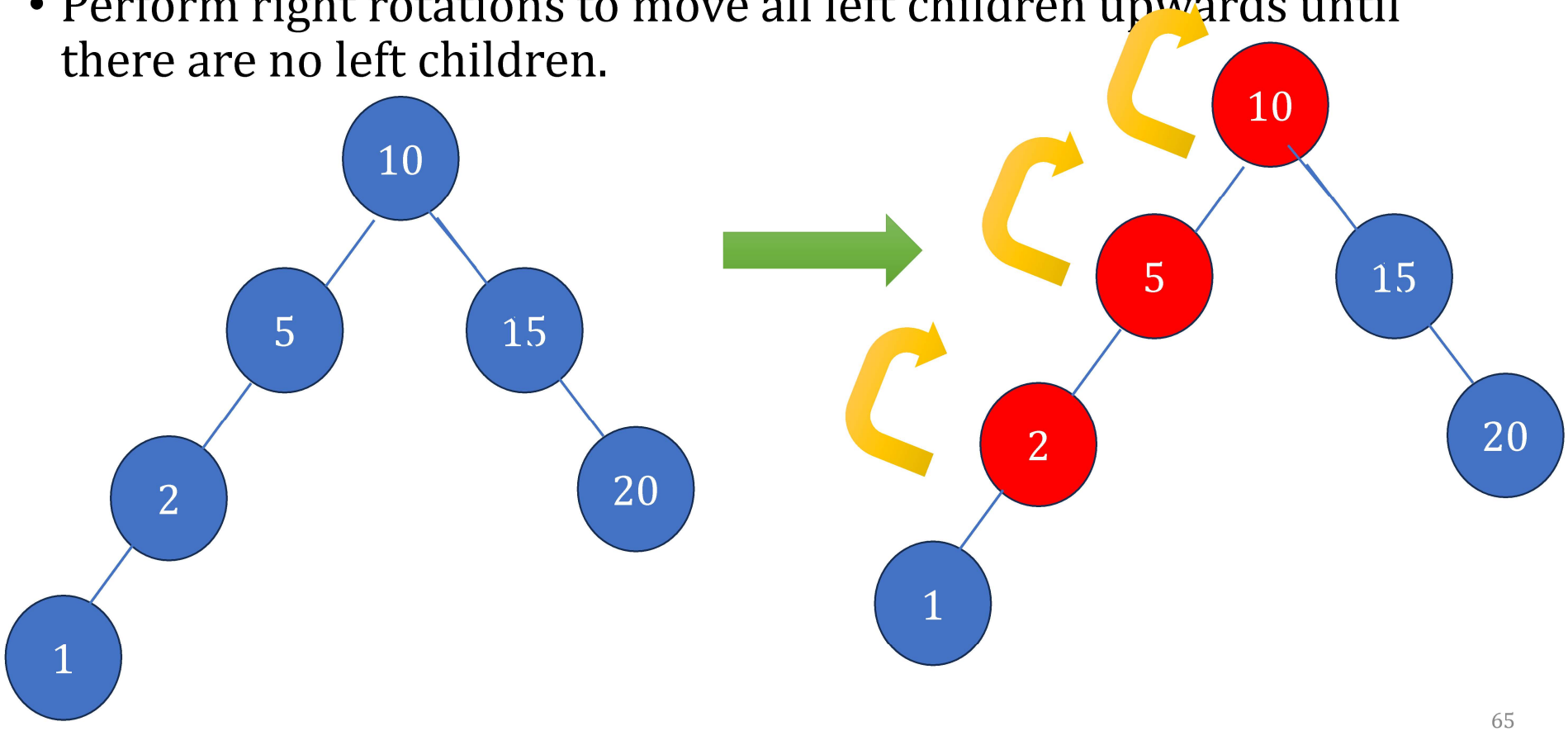
DSW Algorithm – Steps

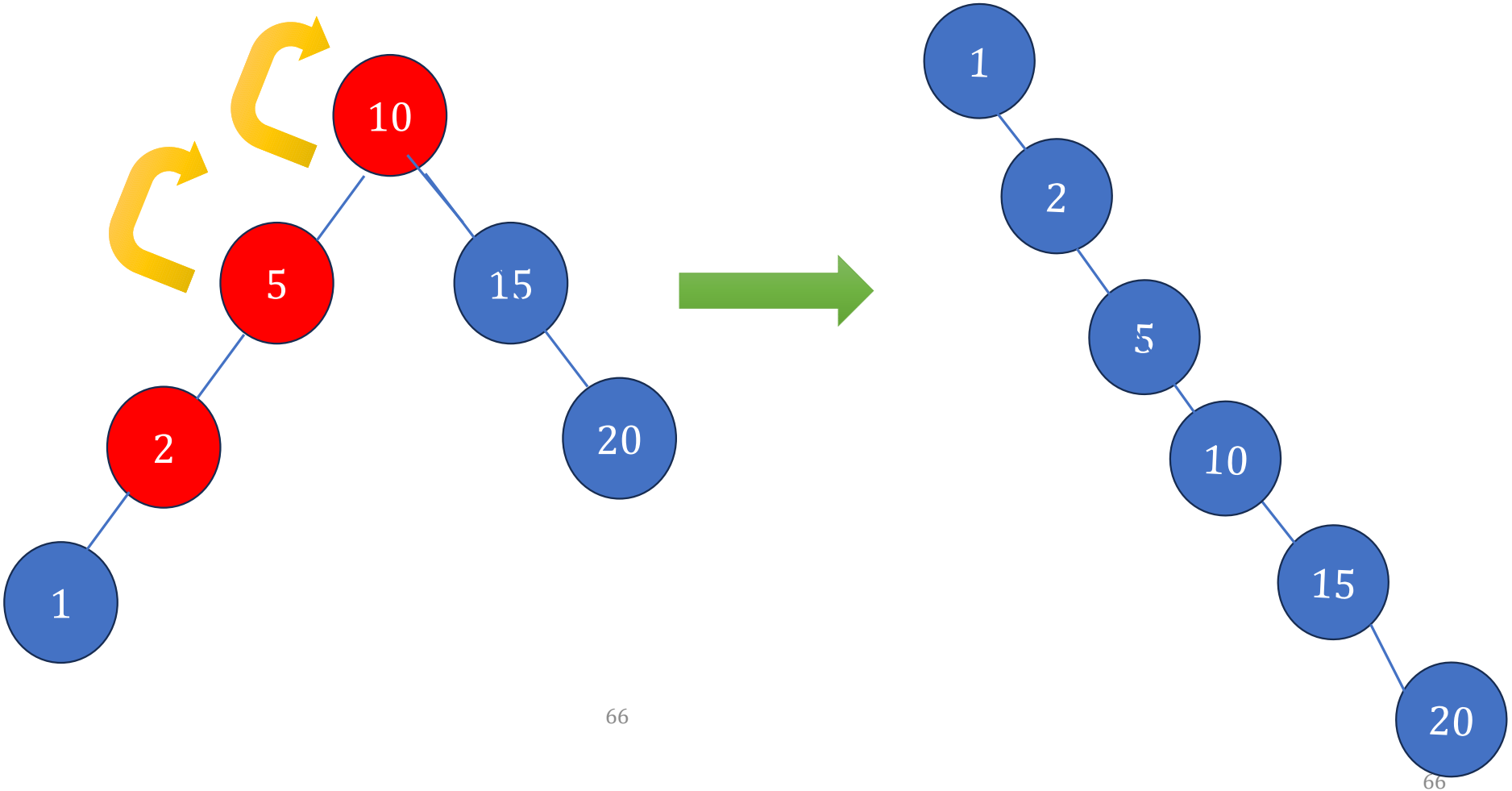
1. Convert the BST into a right-skewed tree (Vine)
   - Flattens the BST into a linked-list like structure where each node has only a right child.
   - Right rotations at each node to move left children upwards.
2. Convert the right-skewed tree into a balanced BST
   - Right skewed tree is converted into a balanced BST using left rotations.
   - Reduces tree height progressively.

Unbalanced BST :example



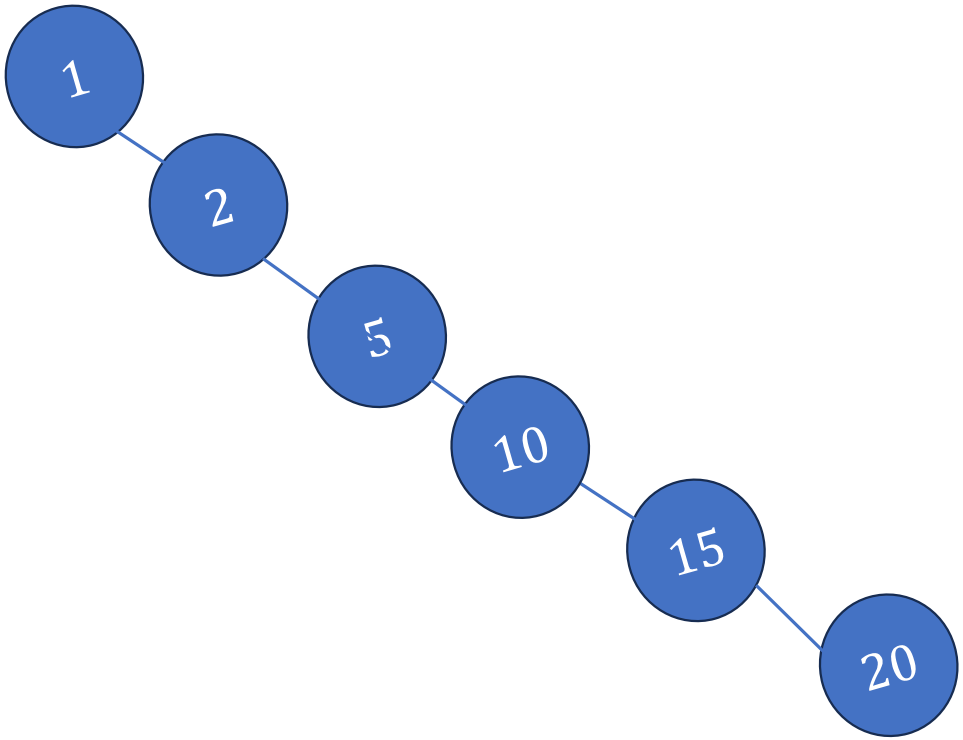
- **Step1** : convert BST to a right skewed Tree (Vine formation)
- Perform right rotations to move all left children upwards until there are no left children.





66

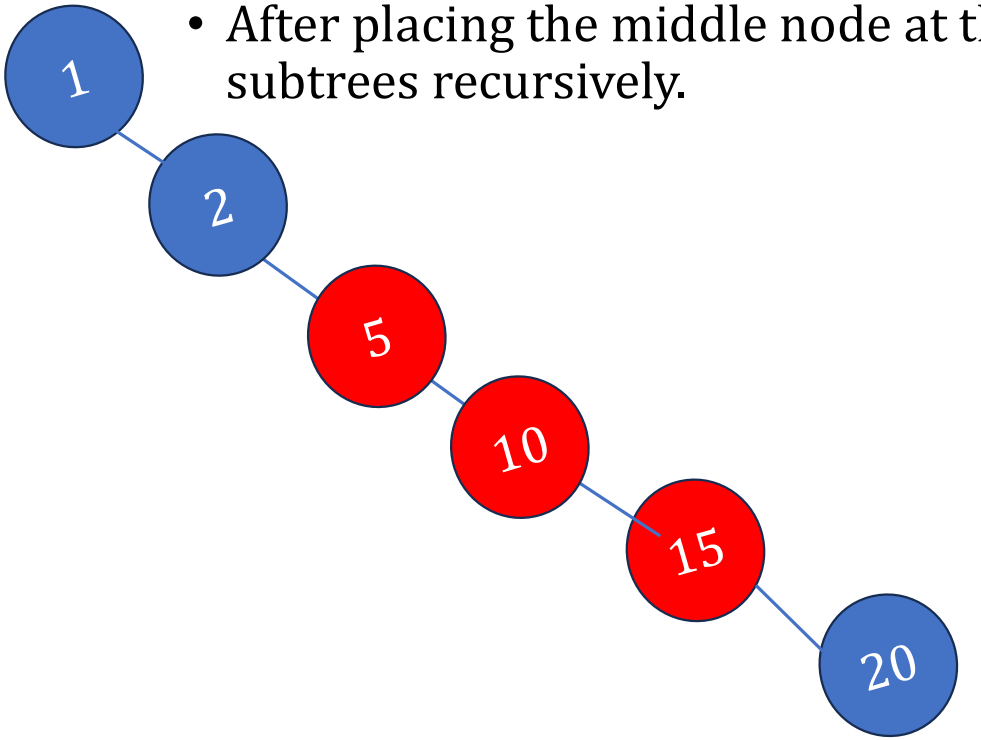
- Now BST is converted into a right-skewed (**link list like structure**)
- **Step2** : convert right-skewed tree into a balanced BST
  - Rebalance the tree using left rotations in multiple passes.



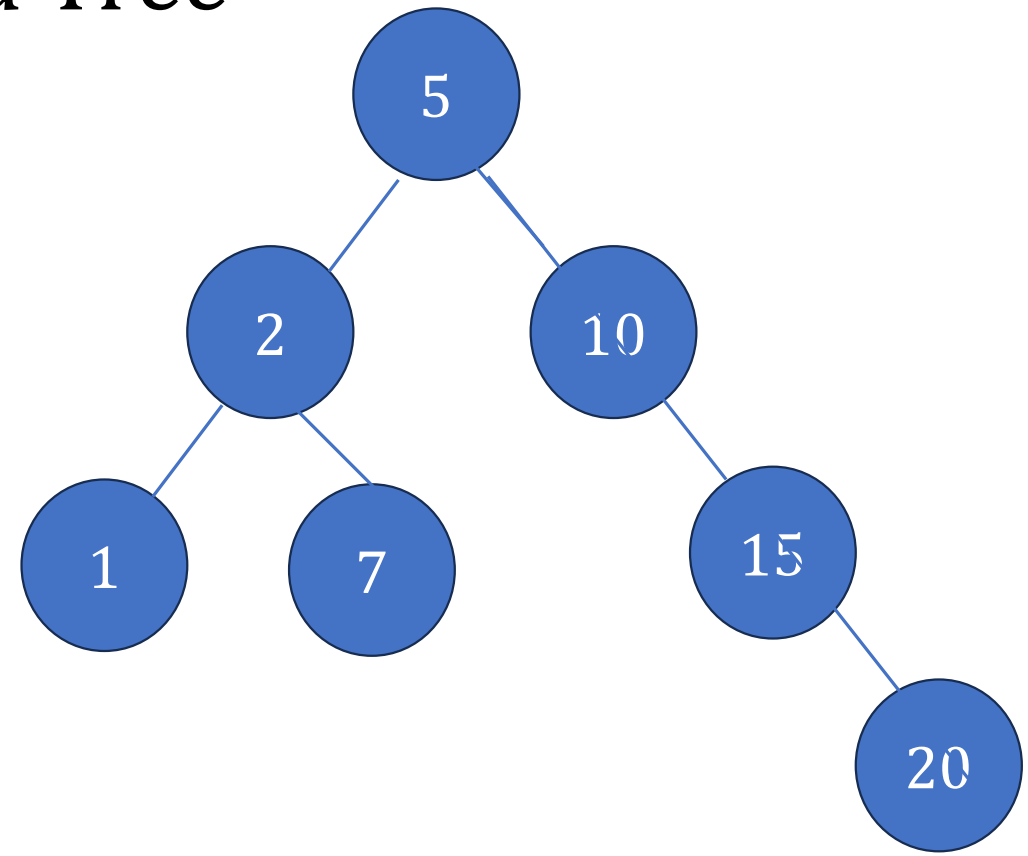
Which nodes to rotate ?

- n = total number of nodes
- Height of the balanced BST is $log_2 n$
- First left rotations are applied to create an almost complete binary tree
  - Addition rotations refine it further

- **Step3** : Process of left rotations
- Find the middle node
  - Rotate the n/2 th node upwards to make it the new root
- Apply left rotations in a structured way
  - After placing the middle node at the top, perform left rotations on subtrees recursively.



Balanced Tree



How to Decide Which Nodes to Rotate?

- For Right Rotations (Vine Formation):
  - Rotate any node that has a left child (move left child up).
- For Left Rotations (Balancing Phase):
  - Rotate the middle node (n/2-th node) up first.
  - Then apply rotations on remaining nodes in a structured way to maintain balance.

Global tree balancing -The DSW algorithm

- DSW is effective at balancing an entire tree (Actually O(n).
- Sometimes trees need only be balanced periodically, in which this cost can be amortized.
- Alternatively, the tree may only become unstable after a series of insertions and deletions, in which case a DSW balancing may be appropriate.

Global tree balancing -The DSW algorithm – Alternatives

- An alternative approach is to ensure the tree remains balanced by incorporating balancing into any insertion/deletion algorithms
- Using AVL trees – insertion and deletion considers the structure of tree.
  - Balanced factor must remain at 1,0 or -1
- However, AVL tree may not be appear completely balanced as after the DSW algorithm.

Why was AVL introduced when DSW already existed ?

- Both **AVL trees** and the **Day-Stout-Warren (DSW) algorithm** are used to maintain balance in a Binary Search Tree (BST)
  - But they serve different **purposes** and have **different use cases**

  - AVL Provides Continuous Balancing (Unlike DSW)
  - DSW only balances the tree when explicitly applied (batch processing).
  - AVL trees balance the tree immediately after every insertion/deletion.
  - If a BST frequently changes, using DSW repeatedly is inefficient, while AVL ensures a logarithmic height at all times.
  - AVL is More Suitable for Dynamic Data

When to use AVL vs DSW ?

AVL

- The data is frequently updated (insertions/deletions).
- Real-time balance is needed (databases, search engines, routing tables).
- Logarithmic search time is always required.

DSW

- You have a static BST that only needs balancing occasionally.
- Bulk insertion has caused an unbalanced tree, and you need a one-time fix.
- The BST is used mostly for read-heavy operations, and rebalancing isn't needed often.