Tree

Structures and Algorithms

Hanchen Wang • 09.09.2018

Outline

Introduction

- Definition & Properties
- Basic Operations

Problem Sets

- Traversing Tree
- Balancing Binary Tree
- Querying Binary Tree

Advanced Structures

- Red-Black Tree
- Balanced Search Tree
- van Emde Boas Tree

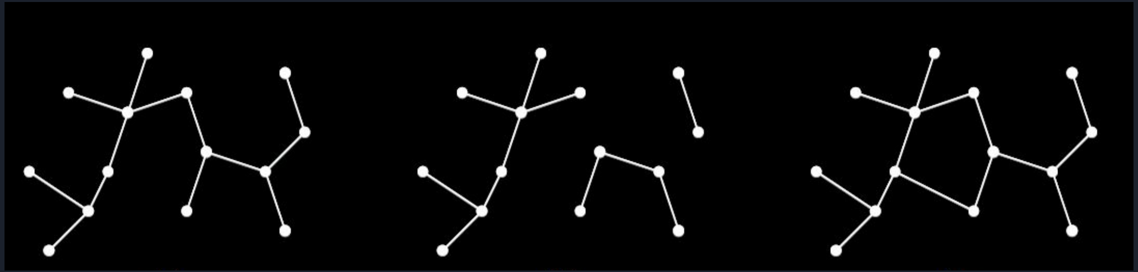
Introduction

Math Definition
(Free) Tree;
Rooted and Ordered Tree;
Binary and Positional Tree;

Practical Instance
Search;
Minimum/Maximum;
Successor/Predecessor
Insert/Delete

Introduction - Definition & Property - (Free) Tree

(Free) Tree -> Connected, Acyclic, Undirected Graph

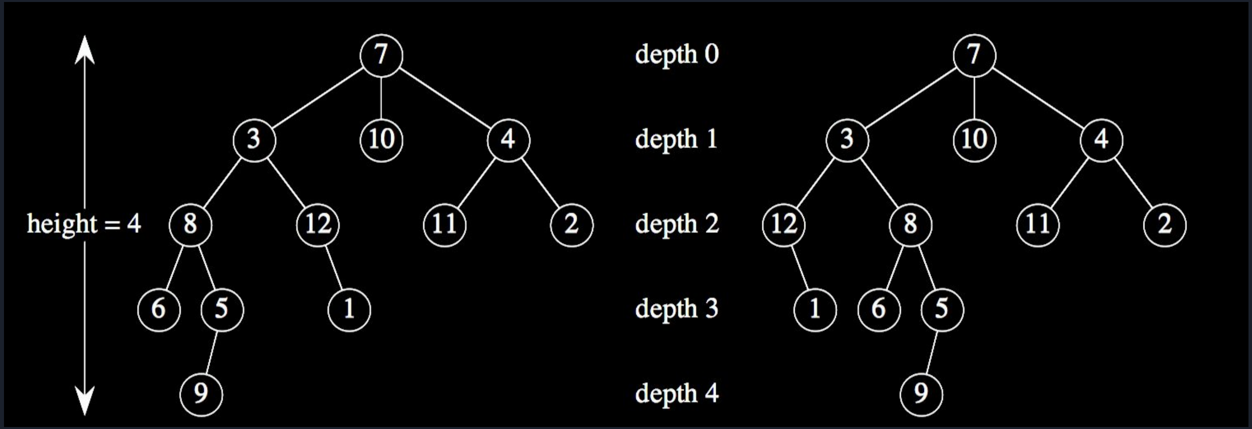


(a) Tree (b) Forest (c) Neither

- G = (V, E)
- Any two vertices in G are connected by a unique simple path
- G is connected, but if any edge is removed from E, the resulting graph is disconnected
- G is connected, and |E| = |V| - 1
- G is acyclic, and |E| = |V| - 1
- G is acyclic, but if any edge is added to E, the resulting graph contains a cycle

Introduction - Definition & Property - R* Tree

Rooted Tree -> a free tree in which one of the vertices is distinguished from the others



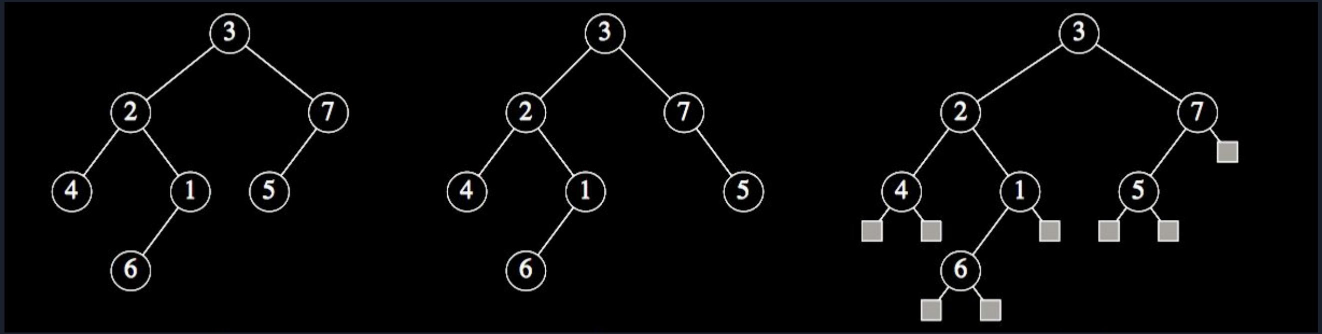
- **Terms**: Root/Leaf; Ancestor/Descendent; Parent/Child; Degree/Depth/Height

Ordered Tree -> a rooted tree in which the children of each node are ordered.

Introduction - Definition & Property - B* Tree

Binary Tree -> A Structure that defined on a finite set of nodes that:

- either contains no nodes,
- or is composed of three disjoint sets of nodes: a root node, a binary tree called its left subtree, and a binary tree called its right subtree.



- **Terms:** empty(null)/full tree; NIL

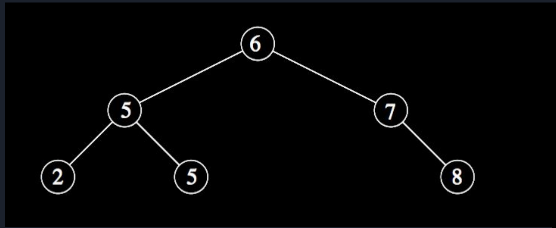
Positional Tree -> Extended version of Binary Tree, with more than two children per nodes, with index

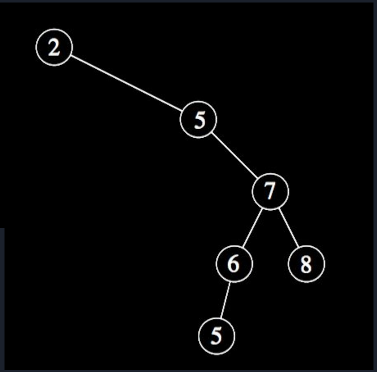
- **Terms**: k-ary tree, complete k-ary tree

Introduction - Basic Operations (of Binary Search Tree)

Binary Search Tree

- A Binary Tree
- Each node contains attributes *key*, *left*, *right*, and *p* that point to the nodes corresponding to its *stored data, left child, right child,* and *parent*, respectively
- For any node X, If Y is a node in the left subtree of X, then y.key < x.key; If Y is a node in the right subtree of x, then Y.key > X.key





Dynamic-Set Operations

- Search, Minimum, Maximum, Predecessor, Successor, Insert and Delete
- Time Complexity -> $O(\log_2 n)$
- Recursive Method: Inorder/Preorder/Postorder

Introduction - Basic Operations (of Binary Search Tree)

Ergodic Method

- Inorder: Left Subtree -> Root -> Right Subtree
- Preorder: Root -> Subtrees
- Postorder: Subtrees -> Root

Tree Walk

```
INORDER-TREE-WALK(x)
if x ≠ NIL
    INORDER-TREE-WALK(x.left)
    print x.key
    INORDER-TREE-WALK(x.right)
```

How about Preorder
/Postorder Tree Walk?

Introduction - Basic Operations (of Binary Search Tree)

Search

-Recursively

```
TREE-SEARCH(x, k)
if x == NIL or k == x.key
    return x
if k < x.key
    return TREE-SEARCH(x.left, k)
else return TREE-SEARCH(x.right, k)
```

-Iteratively(better efficiency)

```
ITERATIVE-TREE-SEARCH(x, k)
while x ≠ NIL and k ≠ x.key
    if k < x.key
        x = x.left
    else x = x.right
return x
```

Introduction - Basic Operations (of Binary Search Tree)

Minimum

TREE-MINIMUM(x)
```
while x.left ≠ NIL
    x = x.left
return x
```

Maximum

TREE-MAXIMUM(x)
```
while x.right ≠ NIL
    x = x.right
return x
```

Introduction - Basic Operations (of Binary Search Tree)

Successor

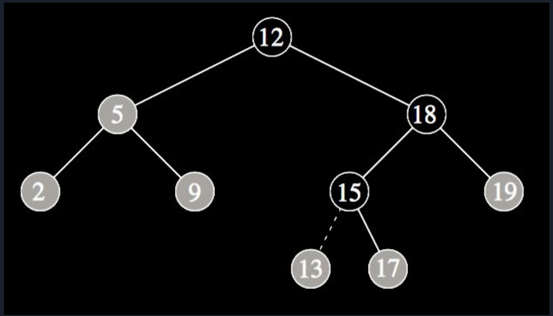
```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
    return TREE-MINIMUM(x.right)
y = x.p
while y ≠ NIL and x == y.right
    x = y
    y = y.p
return y
```

How about Predecessor?

Introduction - Basic Operations (of Binary Search Tree)

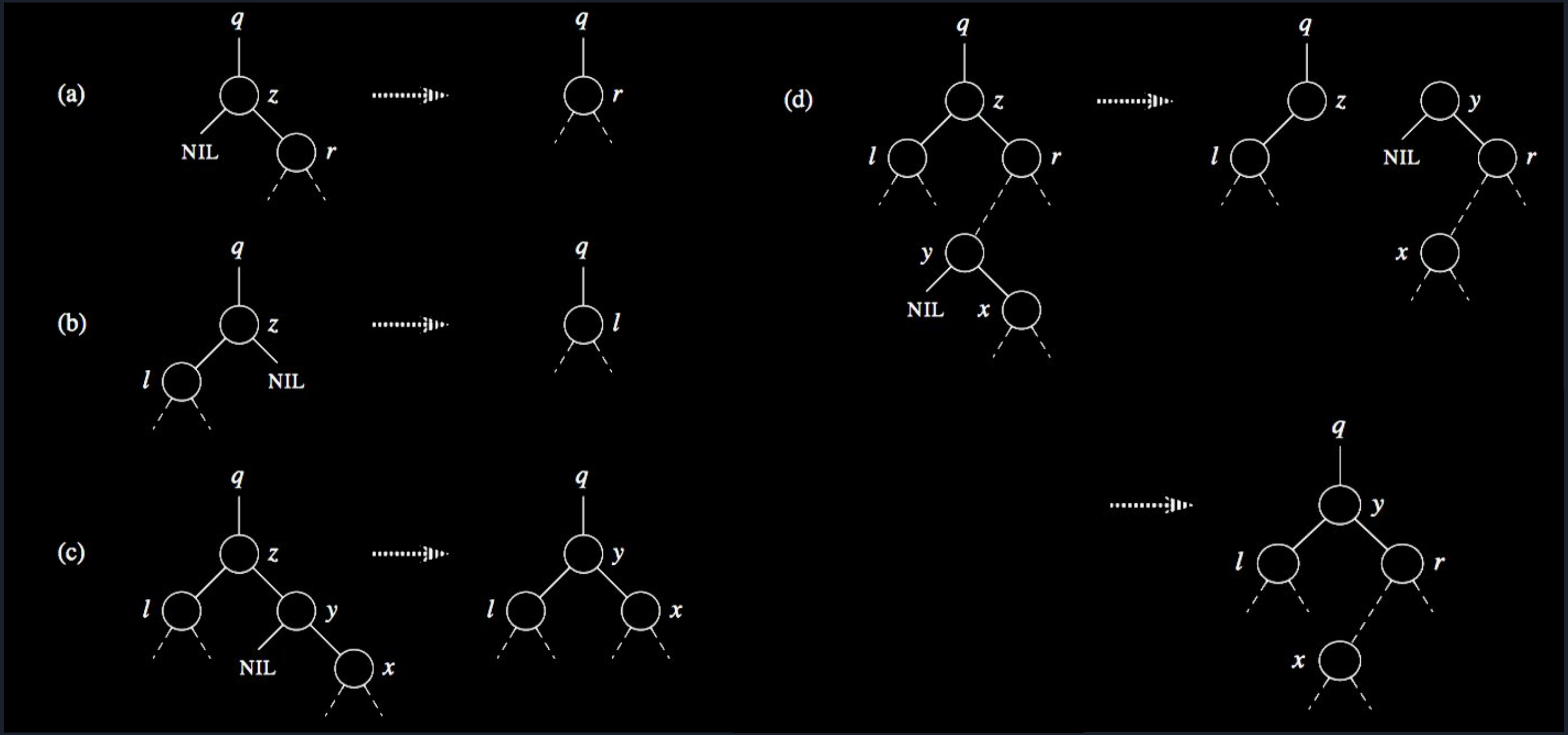
Insert

```
TREE-INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z          // tree T was empty
elseif z.key < y.key
    y.left = z
else y.right = z
```



Introduction - Basic Operations (of Binary Search Tree)

Delete



Introduction - Basic Operations (of Binary Search Tree)

Delete

```
TRANSPLANT(T, u, v)
if u.p == NIL
    T.root = v
elseif u == u.p.left
    u.p.left = v
else u.p.right = v
if v ≠ NIL
    v.p = u.p
```

```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    if y.p ≠ z
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
```

Introduction - *Randomly Built Binary Search Tree

- Each of the basic operations on a binary search tree runs in O(h) time, where h is the height of the tree
- Height changes with insertions and deletions
- $h >= \lfloor \log_2 n \rfloor$

- **randomly built binary search tree** on n keys means inserting the keys in random order into an initially empty tree
- It can be proved that the **expected height of a such with n distinct key values is $O(\log_2 n)$**.

Problem Sets

Definition
Traversing Tree
Balancing Binary Tree
Querying Binary Tree

Problem Sets - 104. Maximum Depth of Binary Tree

Given a binary tree, find its maximum depth.

The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

```python
# Definition for a  binary tree node
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None
```

```python
class Solution:
    def maxDepth(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        if root is None:
            return 0
        left_  = self.maxDepth(root.left)
        right_ = self.maxDepth(root.right)
        return 1 + max(left_, right_)
```

Problem Sets - 111. Minimum Depth of Binary Tree

> Given a binary tree, find its minimum depth.
>
> The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

```python
class Solution:
    # @param root, a tree node
    # @return an integer
    def minDepth(self, root):
        if root is None:
            return 0

        if root.left and root.right:
            return min(self.minDepth(root.left), self.minDepth(root.right)) + 1
        else:
            return max(self.minDepth(root.left), self.minDepth(root.right)) + 1
```

Problem Sets - 94. Binary Tree Inorder Traversal

Given a binary tree, return the inorder traversal of its nodes' values.

Example:

```
Input: [1,null,2,3]
   1
    \
     2
    /
   3

Output: [1,3,2]
```

```python
class Solution:
    def inorderTraversal(self, root):
        """
        :type root: TreeNode
        :rtype: List[int]
        """
        result, curr = [], root
        while curr:
            if curr.left is None:
                result.append(curr.val)
                curr = curr.right
            else:
                node = curr.left
                while node.right and node.right != curr:
                    node = node.right

                if node.right is None:
                    node.right = curr
                    curr = curr.left
                else:
                    result.append(curr.val)
                    node.right = None
                    curr = curr.right

        return result
```

Problem Sets -144. Binary Tree Preorder Traversal

Given a binary tree, return the preorder traversal of its nodes' values.

Example:

```
Input: [1,null,2,3]
   1
    \
     2
    /
   3

Output: [1,2,3]
```

```python
class Solution:
    def preorderTraversal(self, root):
        """
        :type root: TreeNode
        :rtype: List[int]
        """
        result, curr = [], root
        while curr:
            if curr.left is None:
                result.append(curr.val)
                curr = curr.right
            else:
                node = curr.left
                while node.right and node.right
                    node = node.right

                if node.right is None:
                    result.append(curr.val)
                    node.right = curr
                    curr = curr.left
                else:
                    node.right = None
                    curr = curr.right

        return result
```

Problem Sets -145. Binary Tree Postorder Traversal

Given a binary tree, return the postorder traversal of its nodes' values.

Example:

```
Input: [1,null,2,3]
   1
    \
     2
    /
   3

Output: [3,2,1]
```

```python
class Solution:
    def postorderTraversal(self, root):
        """
        :type root: TreeNode
        :rtype: List[int]
        """
        dummy = TreeNode(0)
        dummy.left = root
        result, cur = [], dummy
        while cur:
            if cur.left is None:
                cur = cur.right
            else:
                node = cur.left
                while node.right and node.right != cur:
                    node = node.right

                if node.right is None:
                    node.right = cur
                    cur = cur.left
                else:
                    result += self.traceBack(cur.left, node)
                    node.right = None
                    cur = cur.right

        return result
```

```python
    def traceBack(self, frm, to):
        result, cur = [], frm
        while cur is not to:
            result.append(cur.val)
            cur = cur.right
        result.append(to.val)
        result.reverse()
        return result
```

Problem Sets -102. Binary Tree Level Order Traversal

Given a binary tree, return the level order traversal of its nodes' values.

For example:

Given binary tree `[3,9,20,null,null,15,7]`,

```
    3
   / \
  9  20
    /  \
   15   7
```

return its level order traversal as:

```
[
  [3],
  [9,20],
  [15,7]
]
```

```python
class Solution:
    # @param root, a tree node
    # @return a list of lists of integers
    def levelOrder(self, root):
        if root is None:
            return []
        result, current = [], [root]
        while current:
            next_level, vals = [], []
            for node in current:
                vals.append(node.val)
                if node.left:
                    next_level.append(node.left)
                if node.right:
                    next_level.append(node.right)
            current = next_level
            result.append(vals)
        return result
```

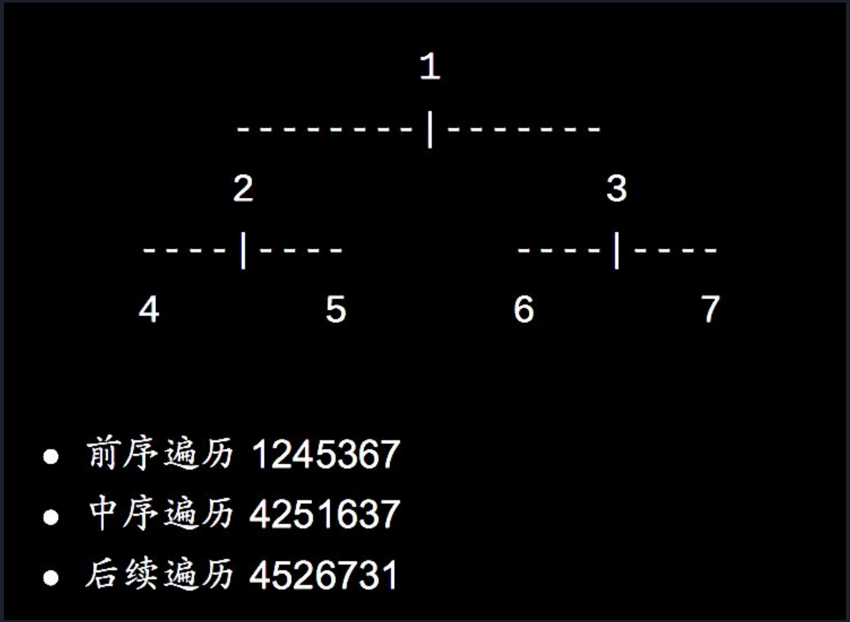
Problem Sets - 889. Construct Binary Tree from Preorder and Postorder Traversal

Given preorder and postorder traversal of a tree, construct the binary tree.

```
# Definition for a  binary tree node
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None
```

Problem Sets - 889. Construct Binary Tree from Preorder and Postorder Traversal

Given preorder and postorder traversal of a tree, construct the binary tree.



Problem Sets - 889. Construct Binary Tree from Inorder and Postorder Traversal

Given preorder and postorder traversal of a tree, construct the binary tree.

```python
# Definition for a  binary tree node
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None
```

```python
class Solution:
    def constructFromPrePost(self, pre, post):
        """
        :type pre: List[int]
        :type post: List[int]
        :rtype: TreeNode
        """
        stack = [TreeNode(pre[0])]
        j = 0
        for i in range(1, len(pre)):
            node = TreeNode(pre[i])
            while stack[-1].val == post[j]:
                stack.pop()
                j += 1
            if not stack[-1].left:
                stack[-1].left = node
            else:
                stack[-1].right = node
            stack.append(node)
        return stack[0]
```

Problem Sets - 105. Construct Binary Tree from Preorder and Inorder Traversal

Given preorder and inorder traversal of a tree, construct the binary tree.

```python
class Solution:
    # @param preorder, a list of integers
    # @param inorder, a list of integers
    # @return a tree node
    def buildTree(self, preorder, inorder):
        lookup = {}
        for i, num in enumerate(inorder):
            lookup[num] = i
        return self.buildTreeRecu(lookup, preorder, inorder, 0, 0, len(inorder))

    def buildTreeRecu(self, lookup, preorder, inorder, pre_start, in_start, in_end):
        if in_start == in_end:
            return None
        node = TreeNode(preorder[pre_start])
        i = lookup[preorder[pre_start]]
        node.left  = self.buildTreeRecu(lookup, preorder, inorder, pre_start + 1, in_start, i)
        node.right = self.buildTreeRecu(lookup, preorder, inorder, pre_start + 1 + i - in_start, i + 1, in_end)
        return node
```

Problem Sets - 110. Balanced Binary Tree

Given a binary tree, determine if it is height-balanced

a binary tree in which the depth of the two subtrees of every node never differ by more than 1.

```python
class Solution:
    # @param root, a tree node
    # @return a boolean
    def isBalanced(self, root):
        def getHeight(root):
            if root is None:
                return 0
            left_height, right_height = \
                getHeight(root.left), getHeight(root.right)
            if left_height < 0 or right_height < 0 or \
               abs(left_height - right_height) > 1:
                return -1
            return max(left_height, right_height) + 1
        return (getHeight(root) >= 0)
```

Problem Sets - 109. Convert Sorted List to Binary Search Tree

> Given a singly linked list where elements are sorted in ascending order, convert it to a height balanced BST.

Example:

```
Given the sorted linked list: [-10,-3,0,5,9],

One possible answer is: [0,-3,9,-10,null,5], which represents the following height balanced BST:

      0
     / \
   -3   9
   /   /
 -10  5
```

Problem Sets - 109. Convert Sorted List to Binary Search Tree

> Given a singly linked list where elements are sorted in ascending order, convert it to a height balanced BST.

```
class Solution:
    head = None
    # @param head, a list node
    # @return a tree node
    def sortedListToBST(self, head):
        current, length = head, 0
        while current is not None:
            current, length = current.next, length + 1
        self.head = head
        return self.sortedListToBSTRecu(0, length)

    def sortedListToBSTRecu(self, start, end):
        if start == end:
            return None
        mid = start + (end - start) / 2
        left = self.sortedListToBSTRecu(start, mid)
        current = TreeNode(self.head.val)
        current.left = left
        self.head = self.head.next
        current.right = self.sortedListToBSTRecu(mid + 1, end)
        return current
```

Problem Sets - 114. Flatten Binary Tree to Linked List

Given a binary tree, flatten it to a linked list in-place.

For example, given the following tree:

```
    1
   / \
  2   5
 / \   \
3   4   6
```

The flattened tree should look like:

```
1
 \
  2
   \
    3
     \
      4
       \
        5
         \
          6
```

Problem Sets - 114. Flatten Binary Tree to Linked List

Given a binary tree, flatten it to a linked list in-place.

```python
class Solution:
    # @param root, a tree node
    # @return nothing, do it in place
    def flatten(self, root):
        return self.flattenRecu(root, None)

    def flattenRecu(self, root, list_head):
        if root != None:
            list_head = self.flattenRecu(root.right, list_head)
            list_head = self.flattenRecu(root.left, list_head)
            root.right = list_head
            root.left = None
            return root
        else:
            return list_head
```

Problem Sets - 257. Binary Tree Path

Given a binary tree, return all root-to-leaf paths.

Example:

```
Input:

   1
 /   \
2     3
 \
  5

Output: ["1->2->5", "1->3"]

Explanation: All root-to-leaf paths are: 1->2->5, 1->3
```

Problem Sets - 257. Binary Tree Path

Given a binary tree, return all root-to-leaf paths.

```
class Solution:
    # @param {TreeNode} root
    # @return {string[]}
    def binaryTreePaths(self, root):
        result, path = [], []
        self.binaryTreePathsRecu(root, path, result)
        return result

    def binaryTreePathsRecu(self, node, path, result):
        if node is None:
            return

        if node.left is node.right is None:
            ans = ""
            for n in path:
                ans += str(n.val) + "->"
            result.append(ans + str(node.val))

        if node.left:
            path.append(node)
            self.binaryTreePathsRecu(node.left, path, result)
            path.pop()

        if node.right:
            path.append(node)
            self.binaryTreePathsRecu(node.right, path, result)
            path.pop()
```

Advanced Structures

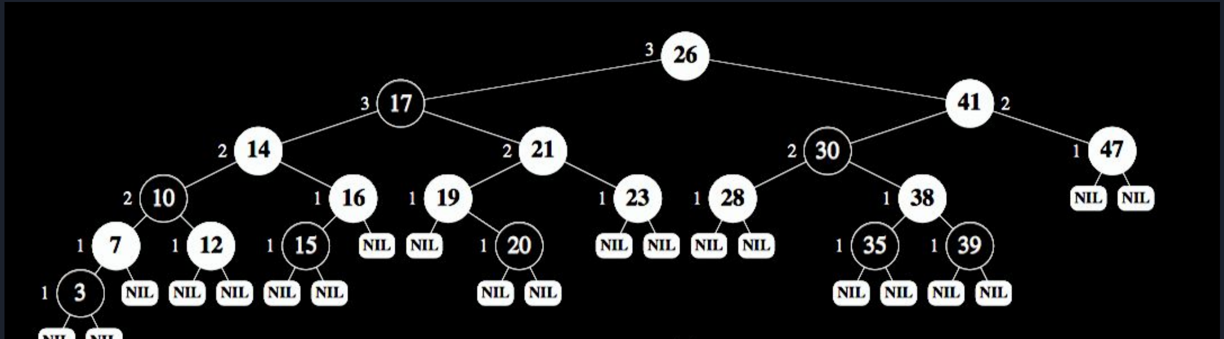
-to bring more balance

- Red-Black Tree
- Balanced Search Tree
- van Emde Boas Tree

Advanced Structures - Red-Black Tree

Red-Black Tree -> A Binary Tree(each node with 1 more attribute, color) that satisfies red-black properties:

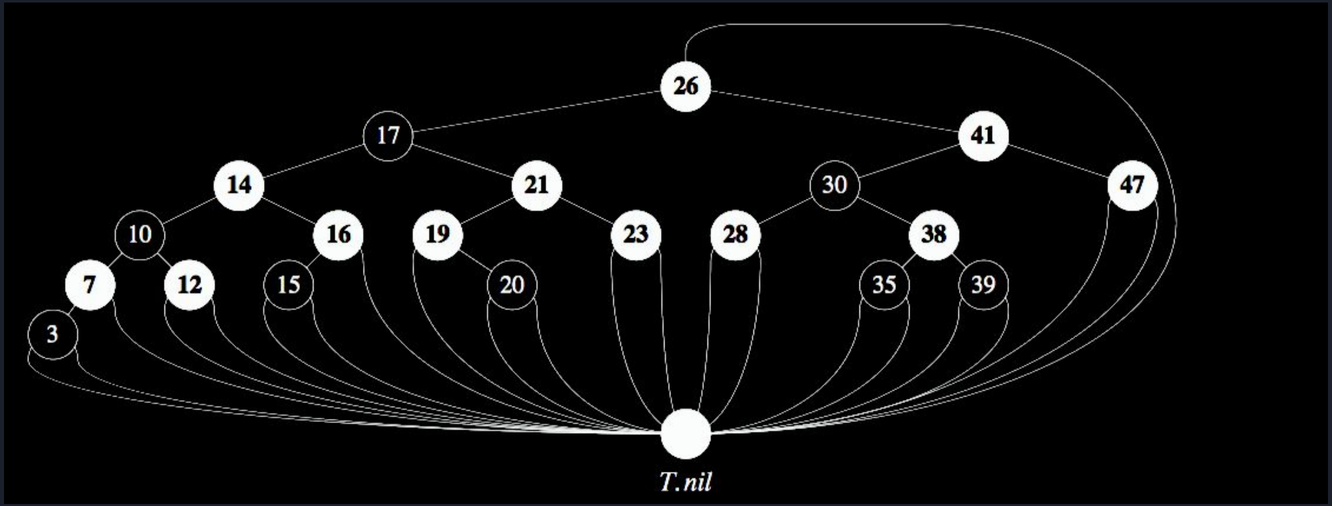
- Every node is either red or black
- The root is black
- Every leaf(NIL) is black
- If a node is red, the both its children are black
- For each node, all simple paths from the node to descendant leaves contain the same number of black nodes



(a)

Advanced Structures - Red-Black Tree

Balanced Property -> A red-black tree with n internal nodes has height at most 2 lg n
Dynamic Operation -> Rotation, Insertion, Deletion



Advanced Structures - Balanced Search Tree

Balanced Search Tree -> better at minimizing disk I/O operations, it is a rooted tree satisfied several specific properties(pp.488-489, *Introduction to Algorithms*, 3rd Ed).

Balanced Property -> the height h of a n-key, t-degree B-tree satisfies:

$$H <= \log_t (n+1)/2 \ (n>=1, t>=2)$$

Dynamic Operation

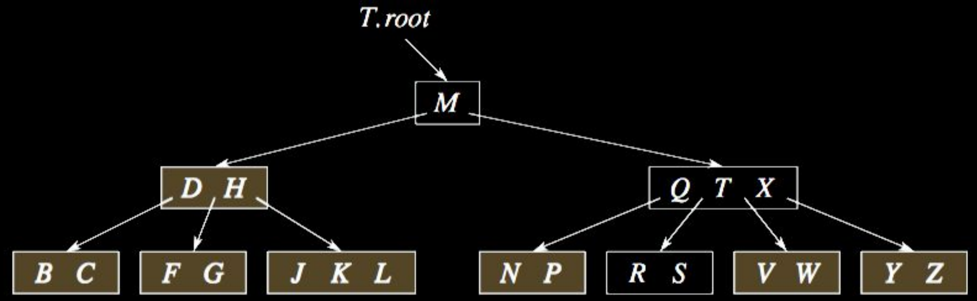


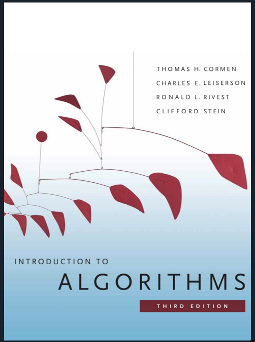
Figure 18.1 A B-tree whose keys are the consonants of English. An internal node x containing $x.n$ keys has $x.n + 1$ children. All leaves are at the same depth in the tree. The lightly shaded nodes are examined in a search for the letter R.

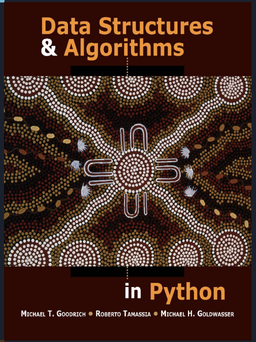
References

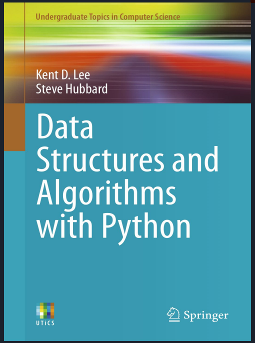
“Introduction to Algorithms” by Thomas H. Cormen et al.

“Data Structures & Algorithms in Python” by Michael T. Goodrich et al.

“Data Structures and Algorithms with Python” by Kent D. Lee et al.







Thanks