Contents

[1. Reverse Linked List in k groups 2](#_Toc471670586)

[2. Sort Linked list using Merge sort 2](#_Toc471670587)

[3. Find intersection of two LinkedLists 3](#_Toc471670588)

[4. Valid Parentheses 3](#_Toc471670589)

[5. WildCard 3](#_Toc471670590)

[6. Longest Valid Parentheses 4](#_Toc471670591)

[7. Expression evaluator 4](#_Toc471670592)

[8. Palindrome partitions 5](#_Toc471670593)

[9. Is Bst 5](#_Toc471670594)

[10. Flip a Tree (create mirror image) 6](#_Toc471670595)

[11. Print root to leaf path 6](#_Toc471670596)

[12. Build Tree (InOrder-Pre/PostOrder) 6](#_Toc471670597)

[13. BST to Doubly Circular Linked List (InOrder sort) 8](#_Toc471670598)

[14. BST Iterator (small to larger number – InOrder sort) 9](#_Toc471670599)

[15. New problem 10](#_Toc471670600)

[16. New problem 10](#_Toc471670601)

# Reverse Linked List in k groups

<todo>

# Sort Linked list using Merge sort

**Tags:** LinkedList, Sorting, Merge sort, LeetCode

Merge sort’s merge operation usually requires additional space for merging into new array. But, using LinkstList that need can be avoided.

public int getLen(ListNode h) { }

public ListNode merge(ListNode l, ListNode r) { }

public ListNode mergeRecursive(ListNode l, ListNode r)

{

// Stack overflow problem on LeetCode with around 31k size length

}

public ListNode sortList(ListNode head, int len)

{

int mid = len/2;

ListNode left = sortList(firstHalf, mid);

ListNode right = sortList(secondHalf, len - mid);

return merge(left, right);

}

# Find intersection of two LinkedLists

**Tags:** LinkedList, LeetCode

Given that m and n are lengths of two LLs -

1. Time O(m\*n); Space O(1)

For each node in m, search entire LinkedList n. If you find a match return that node.

1. **Time O(m+n); Space O(1)**

Find length of each list; Then for longest list traverse abs(m-n) nodes; last step - traverse both lists one at a time till they are same;

1. Time O(m+n); space O(m+n)

Using hashing (set can be used too)

Traverse first list and hash each node; for second list while traversing each node check in hash if it is present; if matched report that node as intersection

1. Time O(m+n); space O(m+n)

Use stack; Idea is use the fact that both list share same size from the end.

While traversing both the list put them in 2 separate stacks; now start poping each element from two stacks simultaneously, if they match report that node as intersection.

# Valid Parentheses

**Tags:** Stack, LeetCode

Given a string containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

The brackets must close in the correct order, "()" and "()[]{}" are all valid but "(]" and "([)]" are not

Solution:

Use a stack to push (, { and ]. Pop during closing brackets.

# WildCard

**Tags:** Recursion

Given a string 123?, replace ? with ‘0’ or ‘1’. So, output will be 1230, 1231

# Longest Valid Parentheses

**Tags:** Stack, LeetCode

**Problem statement:**

Given a string containing just the characters '(' and ')', find the length of the longest valid (well-formed) parentheses substring.

For "(()", the longest valid parentheses substring is "()", which has length = 2.

Another example is ")()())", where the longest valid parentheses substring is "()()", which has length = 4.

**Solution [1]**: Time O(n); Space O(n)

Keep pushing **index** of ‘(‘ in the stack. On finding ‘)’ pop the index and calculate the length (current\_index – index\_of\_top\_element\_from\_stack)

# Expression evaluator

**Tags:** Leetcode, recursion

**Problem statement:**

Given a string that contains only digits 0-9 and a target value, return all possibilities to add binary operators (not unary) +, -, or \* between the digits so they evaluate to the target value.

Examples:

"123", 6 -> ["1+2+3", "1\*2\*3"]

"232", 8 -> ["2\*3+2", "2+3\*2"]

"105", 5 -> ["1\*0+5","10-5"]

"00", 0 -> ["0+0", "0-0", "0\*0"]

"3456237490", 9191 -> []

**Solution:**

Time: O(3^(n-1)); n – number of digits and 3 – number of operators

Space: O(n); size of stack

Using backtracking approach, one can quickly code it’s solution

# Palindrome partitions

**Tags:** LeetCode, recursion

**Problem statement:**

If given string is ‘aab’, then return palindrome partitions like this [ [“aa”, “b”], [“a”, “a”, “b”]]

**Solution:**

Time: O(n\* 2^n); n – Palindrome check; 2^n total substring partitioning.

Space: O(n) –

T(n) = T(n-1) + T(n-2) + … + 1

T(n+1) = T(n) + T(n-1) + … +1 = T(n) + T(n) = 2\*T(n)

Using backtracking, it can be solved easily.

public void findPartitions(string s, int start, List<IList<string>> res, List<string> buff) {

if (s.Length == start) {

res.Add(new List<string> (buff));

return;

}

for(int i=start ; i<s.Length ; i++) {

if (isPalindrome(s, start, i)) {

buff.Add(s.Substring(start, i-start+1));

findPartitions(s, i+1, res, buff);

buff.RemoveAt(buff.Count-1);

}

}

}

# Is Bst

**Tags:** LeetCode, homework

**Problem statement:** Check if a BST is a BST. That tree does not have any duplicates

**Answer:** Time O(n), Space O(n)

<http://articles.leetcode.com/determine-if-binary-tree-is-binary>

Start with left and right values and keep reducing based on sub tree we explore.

# Flip a Tree (create mirror image)

**Tags:** LeetCode, homework

**Problem statement:** Convert a tree into its mirror image

**Answer:**

[1] Recursive approach: Time O(n); Space O(h)

Change left and right from its root. After that make two recursive calls to same function with left and right subtrees.

[2] Iterative approach: Time O(n); Space O(n/2)

Use BFS using queue. Then while putting before left and right into queue, change pointer of left and right tree

<https://leetcode.com/articles/invert-binary-tree/>

# Print root to leaf path

**Tags:** LeetCode, Homework

**Solution:** Time O(n\*h) (n: Total nodes; h: string from root to leaf); Space O(h)

Recursive solution.

# Build Tree (InOrder-Pre/PostOrder)

**Tags:** Homework

**Question:** Given the in-order and pre-order traversing results of a binary tree (as arrays), write a function to rebuild the tree. The function should return the pointer to the root node of the tree. Then take that pointer, and print your tree level by level (level order).

**Trivia:** Generally speaking, one needs to be given in-order traversal (with either pre or post or level), as input, in order to re-construct a binary tree. Without in-order traversal given, it’s not possible to re-construct a binary tree without ambiguity, even if all other 3 traversal orders are given. The only exception, is if we know something more about the tree e.g. if the binary tree is full and complete, then we can resolve the ambiguity without having to know the in-order traversal. [Something to read: <http://www.geeksforgeeks.org/if-you-are-given-two-traversal-sequences-can-you-construct-the-binary-tree/> ]

**Solutions:**

<http://articles.leetcode.com/2011/04/construct-binary-tree-from-inorder-and-preorder-postorder-traversal.html>

<http://edwardliwashu.blogspot.com/2013/01/construct-binary-tree-from-preorder-and.html>

<https://www.youtube.com/watch?v=PAYG5WEC1Gs>

**Two approaches:**

* First node in preorder represents root node. We have to search that node in inorder array. Left side of remaining nodes represents left subtree and right side represents right subtree

static TreeNode create(int[] inArr, int[] preArr, Dictionary<int, int> hash, int remaining, int preStart, int inStart)

{

if (remaining == 0)

{

return null;

}

int val = preArr[preStart];

int i = hash[val] - inStart;

TreeNode root = new TreeNode(val);

root.left = create(inArr, preArr, hash, i, preStart + 1, inStart);

Console.WriteLine($"start: {preStart + i + 1}, offset: {hash[val] + 1}");

root.right = create(inArr, preArr, hash, remaining - i - 1, preStart + i + 1, hash[val] + 1);

return root;

}

static TreeNode create\_using\_Length(int[] inArr, int inStart, int inEnd, int[] preArr, int preStart, int preEnd,

Dictionary<int, int> hash)

{

if (inStart > inEnd)

{

return null;

}

int val = preArr[preStart];

int len = hash[val] - inStart;

TreeNode root = new TreeNode(val);

root.left = create\_using\_Length(inArr, inStart, hash[val] - 1, preArr, preStart + 1, preStart + len - 1, hash);

root.right = create\_using\_Length(inArr, hash[val] + 1, inEnd, preArr, preStart + len + 1, preEnd, hash);

return root;

}

# BST to Doubly Circular Linked List (InOrder sort)

**Tags**: Homework, LeetCode

**Question:** Convert a BST to a sorted circular doubly-linked list (list will be sorted InOrder manner). Think of the left and right pointers as synonymous to the previous and next pointers in a doubly-linked list.

**Solution**:

[1] We can use divide and conquer approach. From the root node, we’ll split the problem to left and right subtree. After that, we stitch left-subtree-result with root and right-subtree-result. (<http://cslibrary.stanford.edu/109/TreeListRecursion.html> )

TreeNode BstToDLLHelper(TreeNode node) {

if (node == null) {

return null;

}

var lList = BstToDLLHelper(node.left);

var rList = BstToDLLHelper(node.right);

var selfLoop = SelfLoop(node);

lList = append(lList, selfLoop);

lList = append(lList, rList);

return lList;

}

[2] There is 2nd approach as well refer: <http://articles.leetcode.com/convert-binary-search-tree-bst-to/>

# BST Iterator (small to larger number – InOrder sort)

**Tags:** LeetCode, Homework, InterviewBit

**Question:** Implement an iterator over a binary search tree (BST). Your iterator will be initialized with the root node of a BST.

1. Calling next() will return the next smallest number in the BST.

2. Calling hasNext() should return whether the next element exists.

Both functions should run in average O(1) time and uses O(h) memory, where h is the height of the tree.

**Solutions:**

Choice of the solution will depend on what the interviewer asks you to do. #2 is generally preferred i.e. without assuming there is a parent pointer.

1. With parent pointer: <http://stackoverflow.com/questions/12850889/in-order-iterator-for-binary-tree>

2. Without parent pointer, but with stack: <https://leetcode.com/discuss/20001/my-solutions-in-3-languages-with-stack>

With approach #2, there is amortized cost associated. So, even if it looks otherwise; the average time complexity is O(1).

Basically, we put every element from root to leaf on left edge of tree. As we pop the element we do same thing for popped element’s right node. If stack is non empty, it will have next element available.

Stack<TreeNode> stk = new Stack<TreeNode>();

Solution(TreeNode root) {

stackAll(root);

}

void stackAll(TreeNode node) {

for (; node != null ; node = node.left) {

stk.Push(node);

}

}

TreeNode next() {

TreeNode node = stk.Pop();

stackAll(node.right);

return node;

}

bool hashNext() {

return (stk.Count > 0);

}

# New problem

# New problem