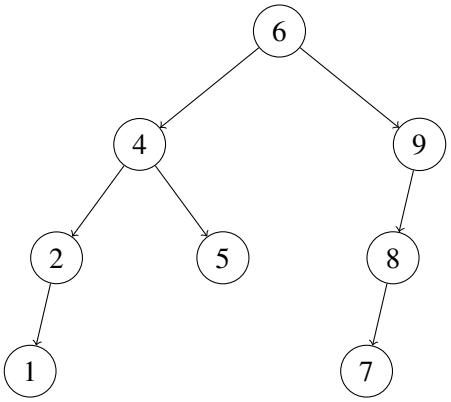
CS 61B Summer 2020 | Small Group Tutoring Section 6 Trees, Stacks, and Queues | Worksheet 6

1 Tree-versal



a) What is the pre-order traversal of the tree? **Solution:** 6 4 2 1 5 9 8 7

b) What is the post-order traversal of the tree? **Solution:** 1 2 5 4 7 8 9 6

c) What is the in-order traversal of the tree? **Solution:** 1 2 4 5 6 7 8 9

d) What is the breadth-first traversal of the tree?
Solution: 6 4 9 2 5 8 1 7

2 Sum Paths

Define a root-to-leaf path as a sequence of nodes from the root of a tree to one of its leaves. Write a method `printSumPaths(TreeNode T, int k)` that prints out all root-to-leaf paths whose values sum to k. For example, if T is the binary tree in the diagram below and k is 13, then the program will print out `10 2 1` on one line and `10 4 -1` on another.

```
      10
     /  \
    2    4
   / \    \
  5   1    -1
```

(a) Provide your solution by filling in the code below:

```
public static void printSumPaths(TreeNode T, int k) {
    if (T != null) {
        sumPaths(T, k, "");
    }
```

```
}

public static void sumPaths(TreeNode T, int k, String path) {
    if (T.left == null && T.right == null && k == T.val) {
        System.out.println(path + T.val);
    } else {
        path += T.item + " ";
        if (T.left != null) {
            sumPaths(T.left, k - T.item, path);
        }
        if (T.right != null) {
            sumPaths(T.right, k - T.item, path);
        }
    }
}
```

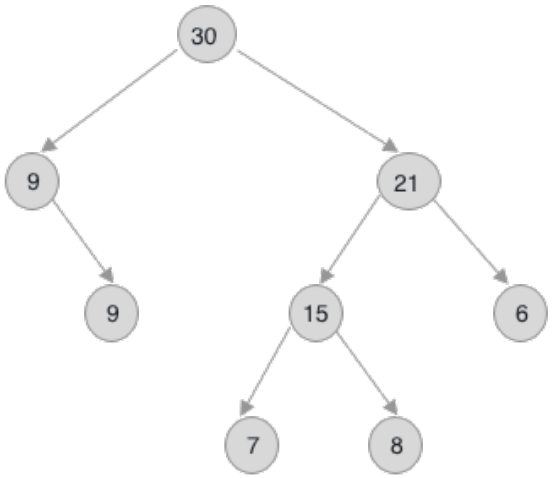
(b) What is the worst case runtime of `printSumPaths` in terms of N, the number of nodes in the tree? What is the worst case runtime in terms of h, the height of the tree?

Solution: In the worst case, the height of the tree is N and we perform a string contatenation at each level. If we assume that all nodes in the tree have itemues bounded by some constant, then at level l we perform a string concatenation of a string of length l (the length of the path from the root to that node) and a string whose length is bounded by some constant. Since string concatenation is linear, we get a runtime of $1+2+\ldots+N = \Theta(N^2)$.

The worst case runtime in terms of h is when we have a complete binary tree. In this case, there are 2^h leaves, all at the bottom level of the tree. Each string concatenation on this level takes $\Theta(h)$ time (again assuming that the values in the tree are bounded by some constant). Thus the total runtime is $\Theta(h2^h)$, since there are at most 2^h non-leaf nodes and the string concatenation for these nodes takes $\emptyset(h)$ time.

3 Sum Tree

Given a binary tree, check if it is a sum tree or not. In a sum tree, value at each non-leaf node is equal to the sum of all elements presents in its left and right subtree. For example, the following binary tree is a sum tree -




```
public boolean isSumTree(TreeNode t) {
        if (t == null || (t.left==null && t.right==null)) {
            return true;
        }
        int left;
        if (t.left == null){
            left = 0;
        }
        else {
            left = t.left.item;
        }
        int right;
        if (t.right  == null) {
            right = 0;
        }
        else {
            right = t.right.item;
        }

        return (t.item == left + right) &&
                isSumTree(t.left) &&
                isSumTree(t.right);
    }
```

4 When am I useful Senpai?

Based on the description, choose the data structure which would best suit our purposes. Choose from: **A - arrays, B - linkedlists, C - stacks, D - queues** (excluding dequeue's cause they're too OP).

1. Keeping track of which customer in a line came first.
Solution: D. Queues have a first in first out (FIFO) ordering.

2. We will expect many inserts and deletes on some dataset, but not too many searches and lookups.
Solution: B. Linked Lists aren't great for searching, but they're quite fast for insert and delete operations.

3. We gather a lot of data of a fixed length that will remain relatively unchanged overtime, but we access its contents very frequently.
Solution: A. Arrays support constant time access to an element (by index).

4. Maintaining a history of the last actions on Word in case I need to undo something.
Solution: C. Stacks support a first in last out (FILO) ordering.

5 Pseudo Stack

Implement a stack's `push` and `pop` methods using Queues. Assume that we have some MyIntQueue class, with API :

```
boolean isEmpty() //returns true if the queue is empty
void enqueue(int item) //adds item to the back of the queue
int dequeue() //removes the item at the front of the queue
int size() //returns the size of the queue
```

Solution: For first solution, q1 will hold top element of the stack at its end. For alternative solution, basically reversing q1 as items are pushed.

```
public class MyIntStack {
    MyIntQueue q1 = new MyIntQueue();
    MyIntQueue q2 = new MyIntQueue();
    public boolean isEmpty() {
        //Implementation not shown
    }
    public int size() {
        //Implementation not shown
    }
    public void push(int item) {
        q1.enqueue(item);
    }
    public int pop() {
        while (q1.size() > 1) {
            q2.enqueue(q1.dequeue());
        }
        int temp = q1.dequeue();
        MyIntQueue tempQ = q1;
        q1 = q2;
        q2 = tempQ;
        return temp;
    }

    //alt solution below: slower push, but faster pop than first solution
    public void push(int item) {
        q2.enqueue(item);
        while (!q1.isEmpty()) {
            q2.enqueue(q1.dequeue());
        }
        MyIntQueue tempQ = q1;
        q1 = q2;
        q2 = tempQ;
    }
    public int pop() {
        if (!q1.isEmpty()) {
```

```
            return q1.dequeue();
        } else {
            //throw an error, which we shouldn't worry about
        }
    }
}
```

6 A Balancing Act

Given a string *str*, containing just the characters `(`, `)`, {, }, `[`, and `]`, implement a method `hasValidParens` which determines if the string is valid.

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

You may use the getRightParen method provided below.

Solution: The idea is to keep track of which closing parentheses we are looking for in the order that they are needed to correctly match up to their corresponding opening parentheses. Therefore, we use a stack to accommodate for the nested nature of parentheses ordering.

```
private static boolean hasValidParens(String str) {
    Stack s = new Stack();
    for (int i = 0; i < str.length(); i++) {
        char c = str.charAt(i);
        if (c == '{' || c == '(' || c == '[') {
            s.push(getRightParen(c));
        } else {
            if (s.isEmpty()) {
                return false;
            }
            if (c != s.pop()){
                return false;
            }
        }
    }
    return s.isEmpty();
}


/**
    The method getRightParen takes in the left parenthesis
    and returns the corresponding right parenthesis.
**/
private static char getRightParen(char leftParen){
    if (leftParen == '(') {
        return ')';
    } else if (leftParen == '{') {
        return '}';
    } else if (leftParen == '[') {
        return ']';
    } else {
        //not one of the valid parenthesis characters
        throw new IllegalArgumentException();
    }
}
```