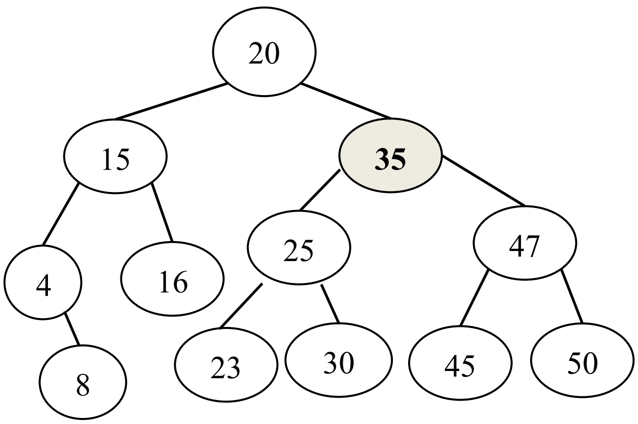
Trees & Search Trees Practice

P1.

a) Could the sequence below (where 30 is the root) be a valid search in a BST for number **65**?
Justify your answer. **30, 80, 60, 50, 70, 65**

b) In the Binary Search Tree below do **a right rotation** at node **35** (rotate 35 to the right). Redraw the tree.

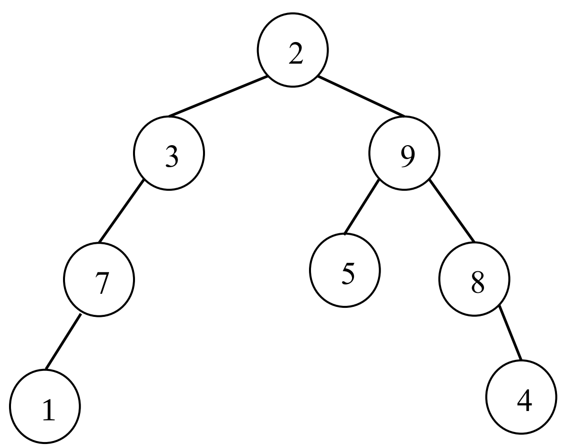


P2. Given the tree on the right, list the nodes in:

(The tree is NOT a BST)

Also fill in the order in which you visit the Root,

And the children (Left and Right)



1. Preorder (________, ________, ________)

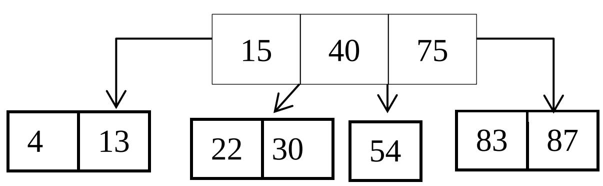
2. Inorder (________, ________, ________)

3. Postorder (________, ________, ________)

P3. (10 points) Given an empty 2-3-4 tree, insert numbers 4, 6, 2, 8, 7, 9 <u>in this order</u>. In the table below, <u>draw the tree after each insertion.</u>

N	Tree after inserting number N	N	Tree after inserting number N
4	Tree after inserting 4 in an empty tree: [4]	**8**	
6	Above tree after inserting 6	**7**	
2		**9**	

P4. Given the 2-3-4 tree:



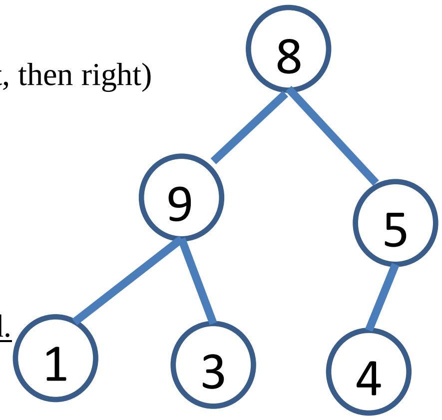
a) (4 points) Insert 80 and redraw the tree:

b) (4 points) After 80, insert 90 and redraw the tree:

P5. (25 points) Write a function `int follow_path(link root, int * A, int F)` that takes as arguments a tree (given by the root node, `root`), an array, `A`, and a number, F, and returns the item found in the tree, following the path given by the first `F` elements of `A`:

- if you see a **<u>0</u>** in `A`, go **left** in the tree;
- if you see a **<u>1</u>** in `A` go **right** in the tree.

Continue the same way. For the function calls below, assume that `root` is pointing to the root, 8, of the tree below.



- `follow_path(root, [0,1], 2)` returns 3 (from root, go left, then right)
- `follow_path(root, [1,0], 2)` returns 4
- `follow_path(root, [0,1,1,0], 2)` returns 3
- `follow_path(root, [1], 1)` returns 5
- `follow_path(root, [0,1], 0)` returns 8

<u>You can **only** assume that `A` has at least `F` elements in it whenever it is called.</u>

<u>If anything goes wrong in the function call, it should return -1.</u>

You can use a helper function if you want. The nodes and links are defined as follows:

```
typedef struct node * link;
struct node{
     int data;
     link left;
     link right;
};
```

a) (6 points) What special cases are there? Focus especially on cases that can crash your code. Give an example (function call and tree) that show each special case.

b) (4 points) **Fill in** the time complexity of the code you wrote for part c). (If needed, you can assume the tree has N nodes in total.)

O (____________)

c) (15 points) Write the **<u>complete code</u>** for the function. It should **<u>deal with the special cases</u>** that you mentioned. Each line or section that deals with a special case, should clearly indicate what case it deals with.

P6. (14 points) Write a function **`list print_reverse(link root)`** that takes a binary search tree (BST) as an argument and **returns a list** with the items in the tree in **reverse order**. If it helps, you can assume the following list functions are provided:

- `list new_list()` - creates and returns an empty list.
- `void add(list L, int x)`, - adds integer `x` **to the end** of the given list, `L`.

You can use a helper function if you want.

The nodes and links are defined as follows. (Do not worry about how the list is represented and implemented.)

```
typedef struct node * link;
struct node{
     int data;
     link left;
     link right;
};
```

a) (3 points) **Fill in** the time complexity of the code you wrote for part b). (Assume the tree has N nodes.)

O (____________)

b) (11 points) Write the **complete code** for the function. It should **not crash**. Do not use any global variables. 3 points will be deducted if any global variable is used.