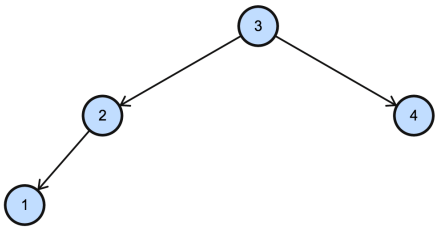
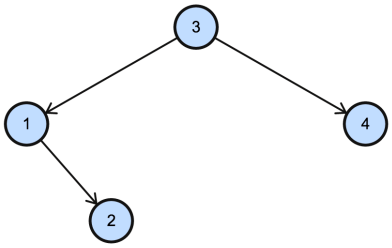
Avik Bag
Professor Peysakhov
CS 260 – Assignment #5
26th July, 2015

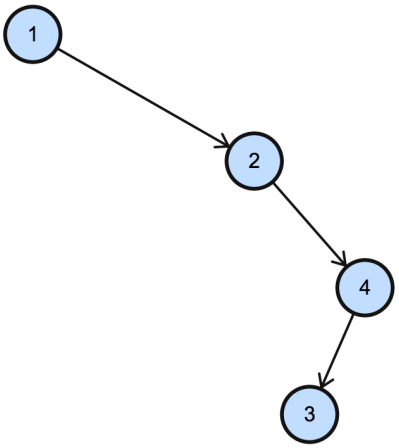
Written Problems

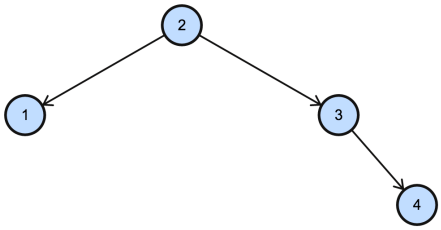
Question 1

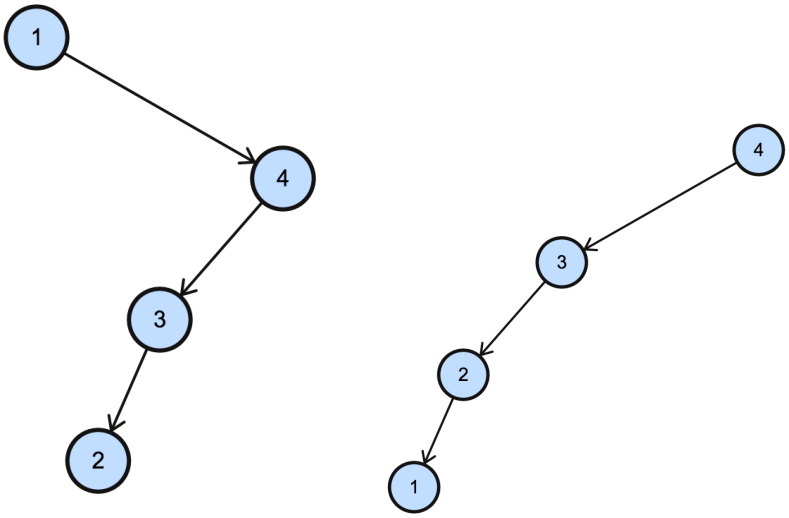
These are all the combinations of binary search trees that can be formed with the numbers 1, 2, 3 and 4.

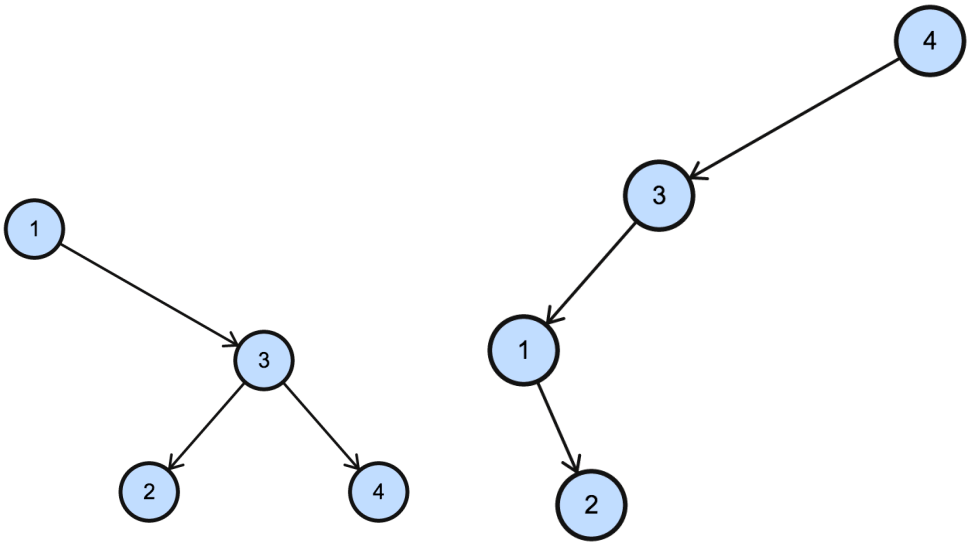


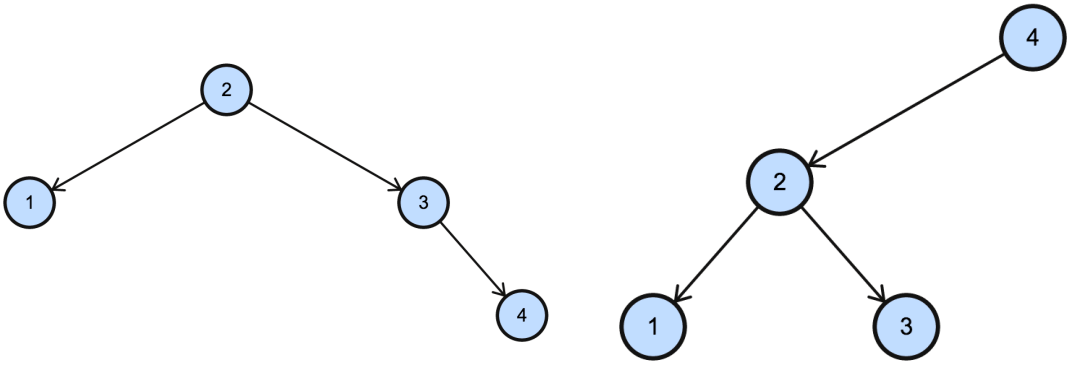


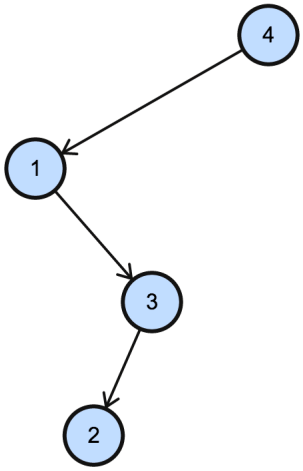


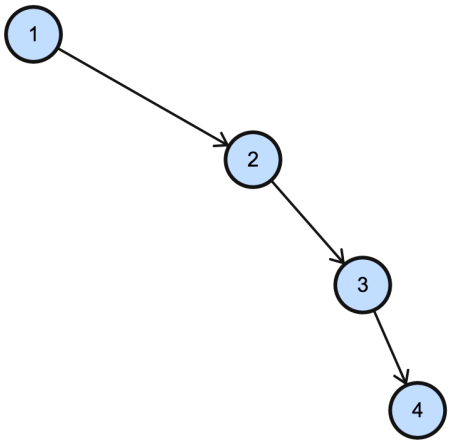


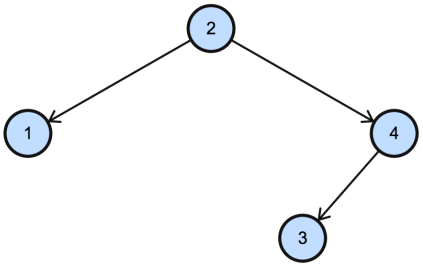

1
4
3
2
4
3
2
1

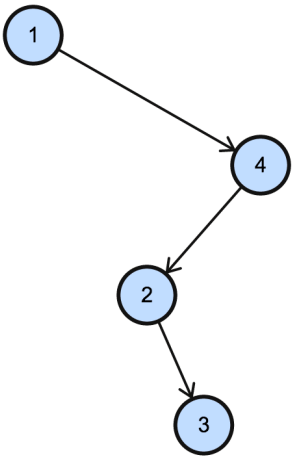

1
3
2
4
4
3
1
2

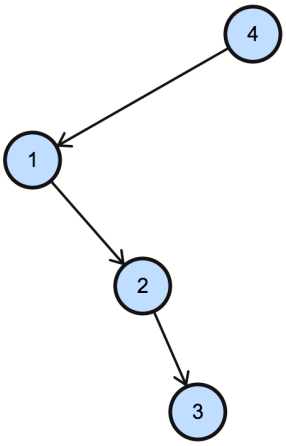

2
1
3
4
4
2
1
3


4
1
3
2

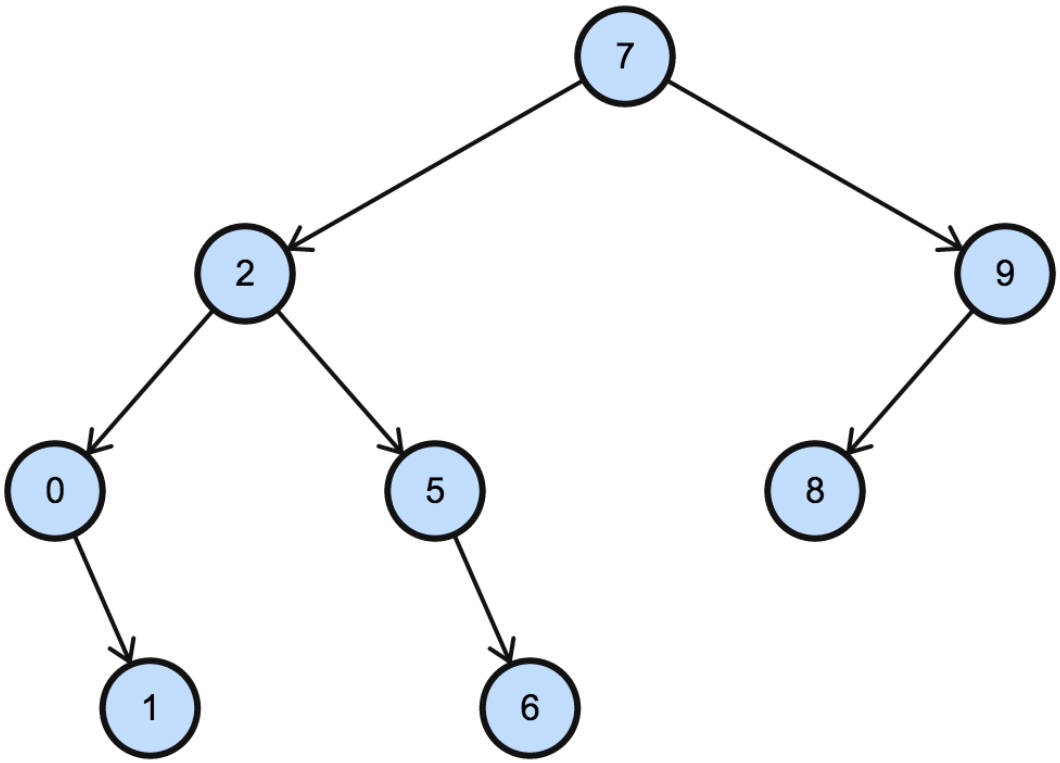

1
2
3
4


2
1
4
3


1
4
2
3

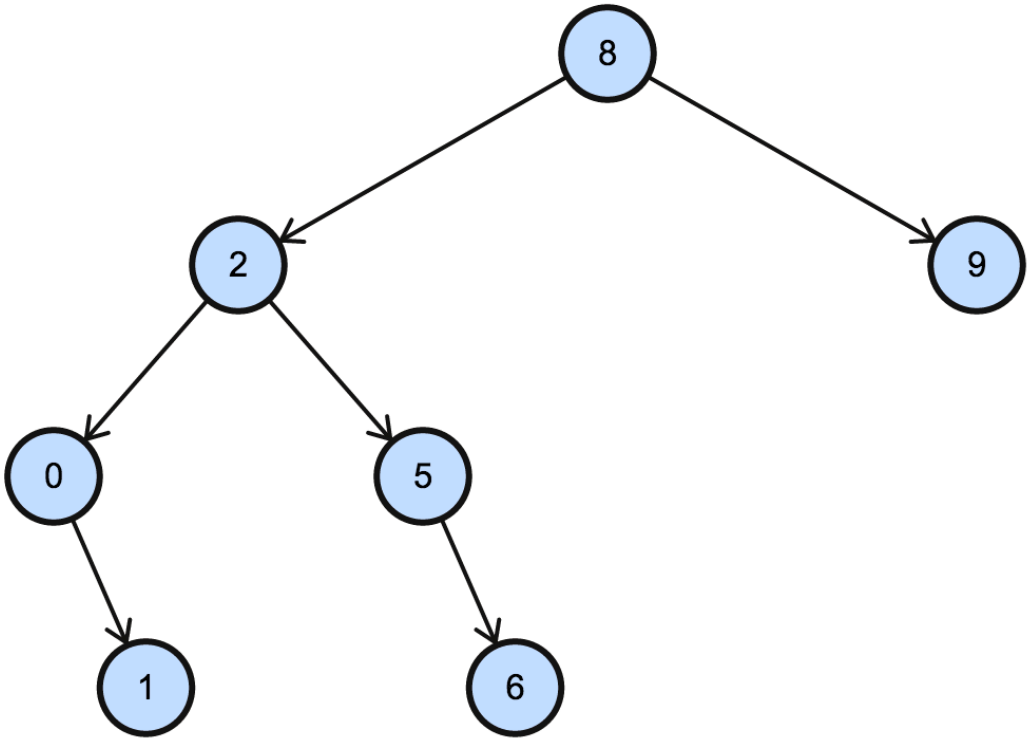

4
1
2
3

Question 2

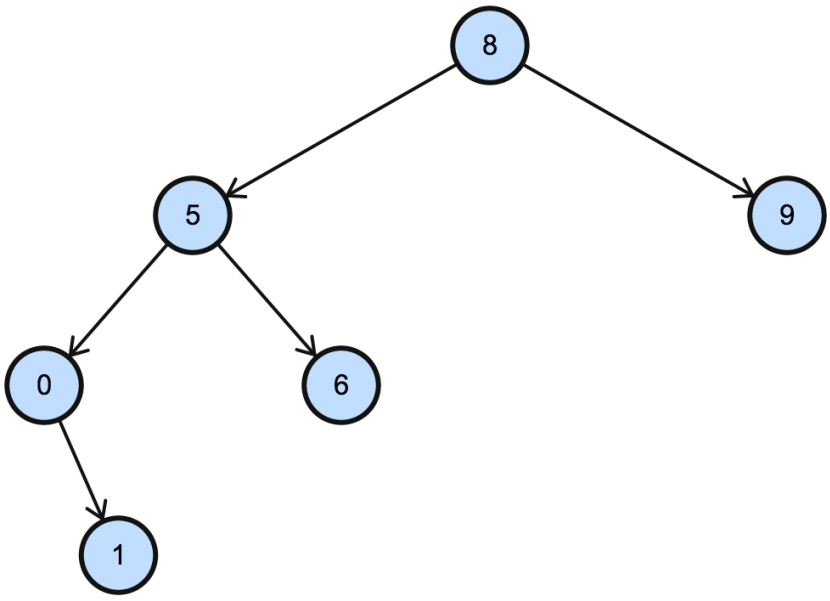


This is the resulting BST after calling the INSERT() function on the given data.

Question 3



This is the result after removing the node containing the value 7.



This is the final result after removing 7 and then removing 2 from the BST.

Question 4

The delete() function does not depend on the order in which the nodes are deleted as it still maintains the order of the BST after deletion of any node. After deleting the mentioned node, the delete function looks for the smallest value within the right sub-tree of the node and replaces it with the deleted node.

Question 5

n	time (ms)
100	0.228
200	0.388
300	0.633
400	0.973
500	1.066
600	1.393
700	1.476
800	1.654
900	1.863
1000	2.293

This is the time output for the heapify function which orders the input value based on the heap concept that iterates over the length of the input array.

Question 6

Both merge sort and heap sort have the same time complexity of $O(n\ log\ n)$. In the case for merge sort, the array gets split into smaller halves and then merged back in order in another array. For heap sort, the array is organized in comparison to it's parent node. If it's greater then the parent node, it is swapped. This is recursively carried out to maintain a heap with the minimum value as the root.

Question 7

```
function algorithm()
        array ar[], height[]
        data = first()
        while data != NULL
                ar.append(data)
                heapify(ar) # heapify based on height value
                data = data->next()

        for x in ar
```

```
        height.append(x.height)

result[] = impressive_A(height[])

for i in 0 to length(ar) - 1
        ar[i].computed_value = result[i]
```

The above function will carry out the job of storing an array of pointers to the struct my_data based on the first iteration using first() and next(). This order of pointers is then organized based on the heapify() function that sorts the array of pointers in a heap based on the height value within my_data. After it is organized in the heap, the array is then traversed and the height values are extracted in that order. Once this array of floats is obtained, impressive_A() is called with the array of floats as the arguments. This will return an array of computed_value in the same order. Then the array containing the ordered array of pointers to my_data is iterated and the computed_value is stored in that order.

Question 8

The concept of stacks works as LIFO (Last in First out) while Queues work on the concept of FIFO (First in First out). By using two stacks, a Queue can be implemented. To insert() a value, the value can be pushed onto one stack. For every insert() call, the value should be pushed to one stack. We can call this the old stack. When the remove() method is called, all the values from the old stack is popped and then immediately pushed onto the new stack. Once all the values from the old stack is completely popped and pushed onto the new stack sequentially, the first value popped from the new stack will represent the first value in the dequeue. Thus that is how a queue is implemented using two stacks. Another important point to remember is that when the remove() method is called, the new stack needs to be empty before the old stack is entirely popped and pushed onto the new stack, as this is necessary to ensure order within the queue .

Question 9

The problem here is that the server side of the system provides a list of checksums as a block of 512 byte, after which it moves on to the next block of 512 bytes, and sequentially goes on and sends a list. There is no over lap in the checksum data. But for the client side of the list, it moves through every possible 512 byte of checksum data. This makes the client side checksum list redundant. Therefore to account for the right blocks of checksum for comparison, we check the byte start and end values in the server side list. We use this value to search for the right checksum block in the client side list. When there is a match in the start and end byte value, the checksum blocks are then compared. If they are the same, it moves on to the next block of 512 bytes, if not the appropriate action can be executed.