CS 260 – Homework #5

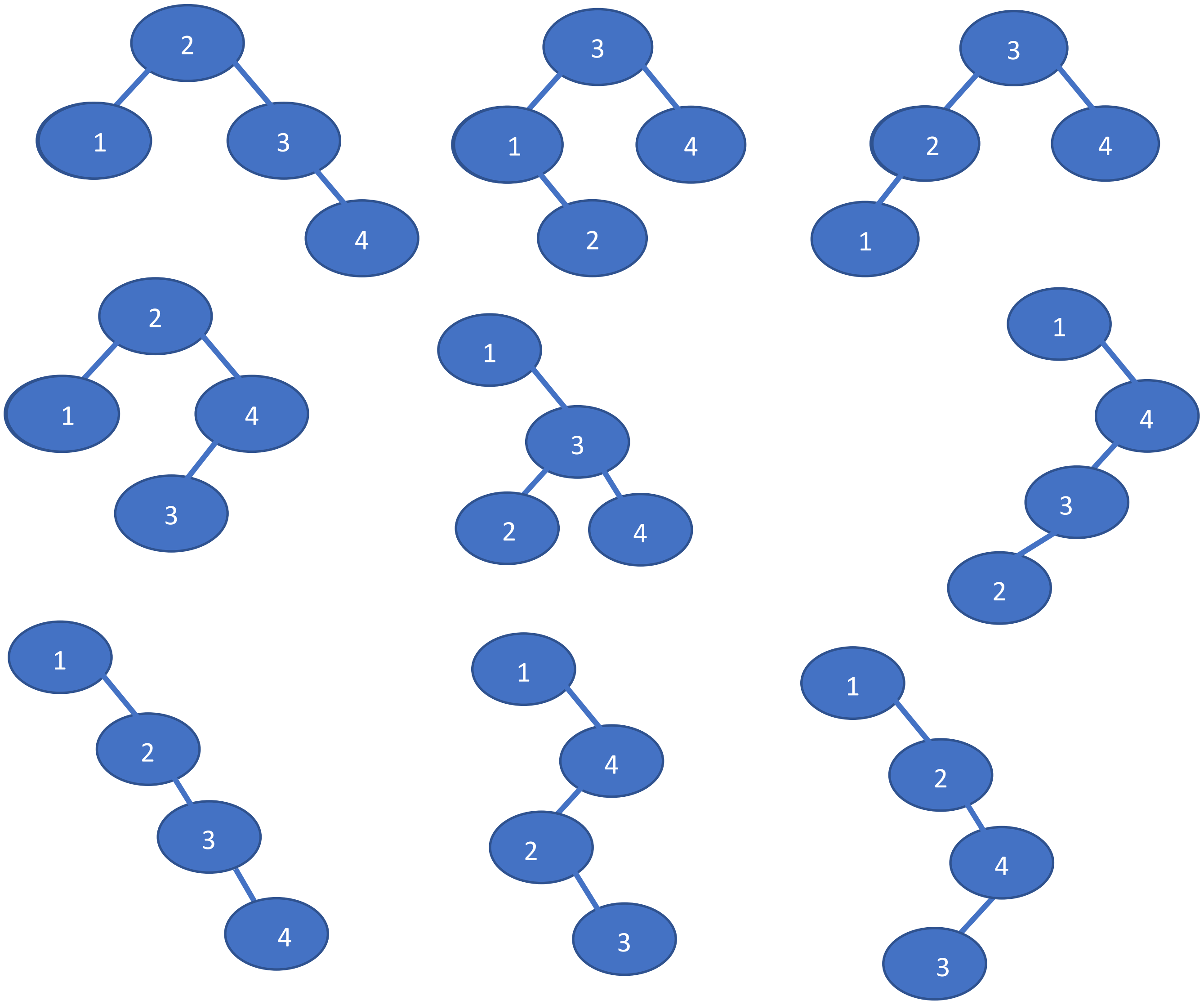
Chris Kasper

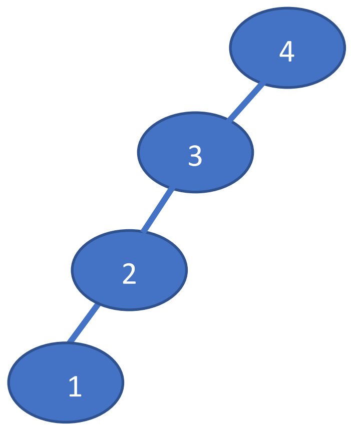
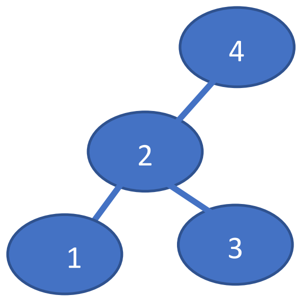
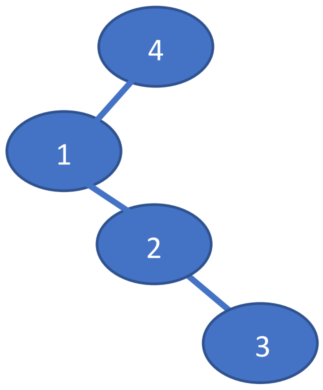
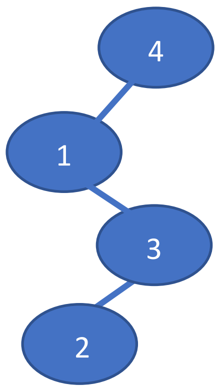
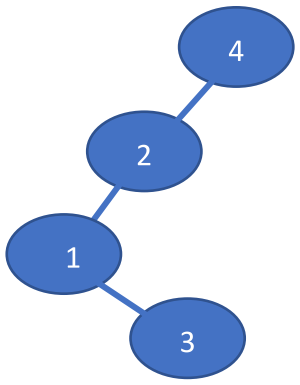
February 10, 2018

Problem 1 (5.1 from text)

Draw all possible binary search trees containing the four elements 1, 2, 3, 4.

Solution

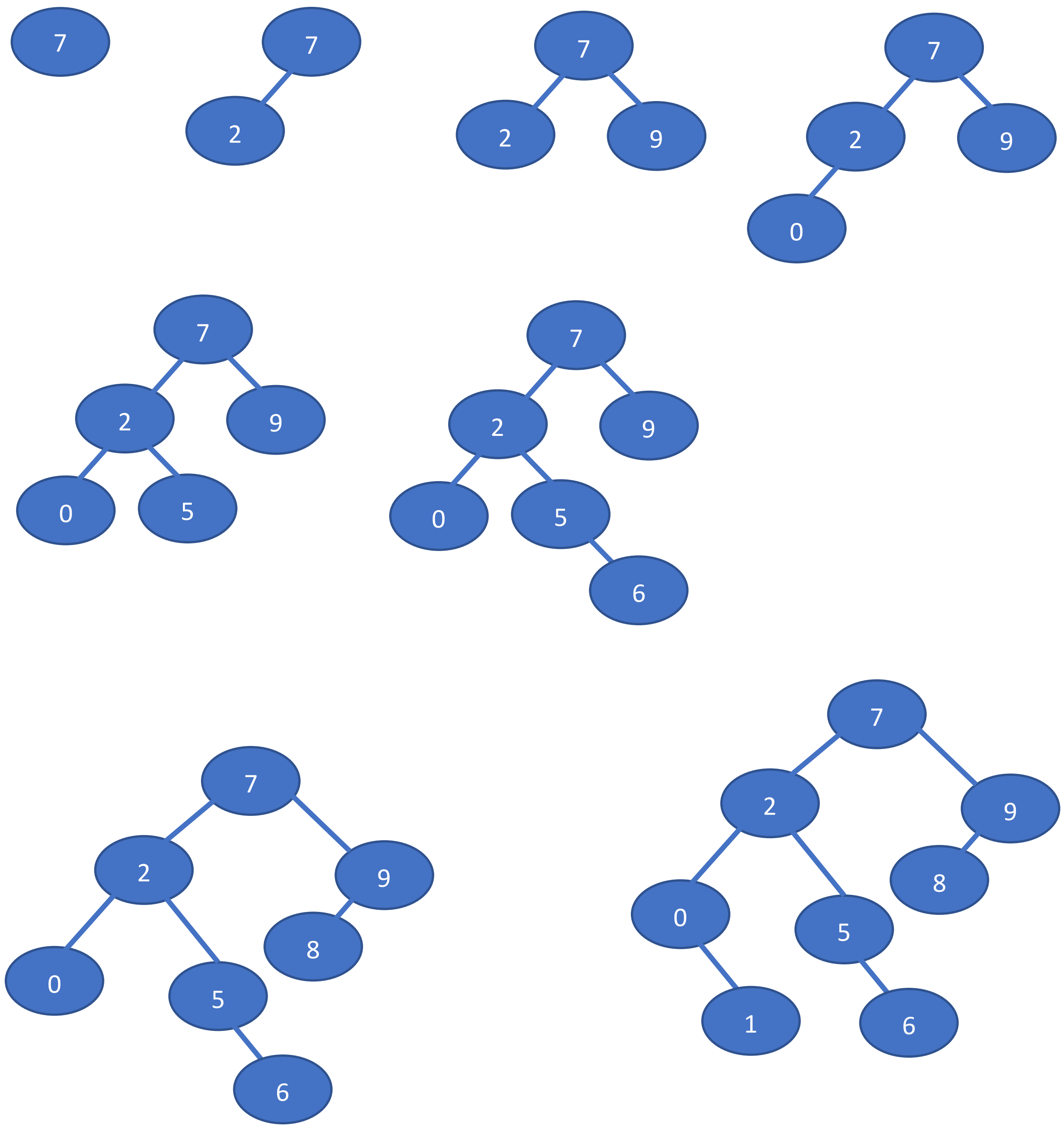



4
3
2
1

4
2
1
3

4
1
2
3

4
1
3
2

4
2
1
3

Problem 2 (5.2 from text)

Insert the integers 7, 2, 9, 0, 5, 6, 8, 1 into a binary search tree by repeated application of the procedure INSERT of Fig. 5.3.

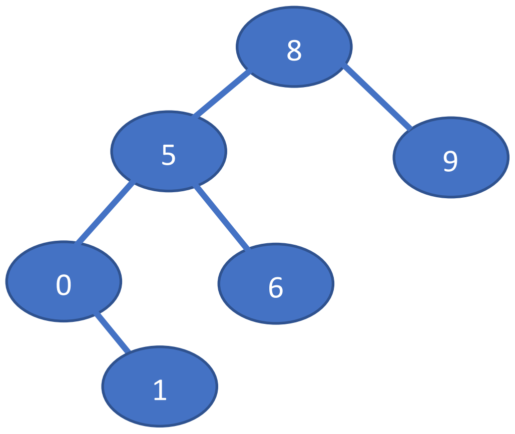
Solution



Problem 3 (5.3 from text)

Show the result of deleting 7, then 2 from the final tree of Exercise 5.2.

Solution



Problem 4 (5.4 from text)

When deleting two elements from a binary search tree using the procedure of Fig. 5.5, does the final tree ever depend on the order in which you delete them?

Solution

Changing the order of deleting two elements can affect the final tree because the successor to the nodes being deleted have to be determined. Changing the order could affect which elements get promoted.

Problem 5

See the programming portion below. Provide your observed behavior of the make_heap() algorithm.

Solution

```
def make_heap(x):
    for i in range((len(x)/2)+1,-1,-1): # n
        downheap(x,i) # log(n)
```

The make_heap() algorithm can create heaps from unordered arrays in O(n) time.

Problem 6

One way to sort numbers is to build a heap and then pop the numbers off the heap. Since each element is the least element left on the heap when you pop it, the stream of numbers popped off must be in sorted order. Analyze the time and space complexity of this method of sorting numbers. (Don't forget to include the time it takes to build the heap!) How does this compare to merge sort ($O(n \lg n)$ time and $O(n)$ space)?

Solution

Making a heap costs O(n) and popping (extracting) costs O(log(n)). However, this method is dominated by making the heap, and has a time complexity of O(n). The space complexity is O(n). This way of sorting proves to be faster than the merge sort as the number of elements increases.

Problem 7

Describe an algorithm (pseudo-code and high-level description, not C) that uses first and next, the above structure definition, and the function impressive_A() to fill in the computed value members. Analyze the time and space complexity of your algorithm, including the fact that your algorithm must call impressive_A().

*More details about problem on course website

Solution

```
Procedure A ( ):
    data = first() O(1)?
    names[] O(1)
    values[] O(1)
    while data != NULL: O(n)
        Insert(values, data->height) O(1)
        Insert(names, data->name) O(1)
        next() O(1)?
    new_values = impressive_A(values) O(n)
    data = first() O(1)?
    while data != NULL: O(n)
        idx = Locate(data->name, names) O(n)
        data->computed value = values[idx] O(1)
        next() O(1)?
```

Time complexity = O(n) + O(n)*O(n) = O(n) + O(n^2) = O(n^2)
Space complexity = O(n) (because it depends on n number of data structures)

Problem 8

Design a queue using only two stacks. What is the time complexity for insert? For remove()? What is the space complexity?

Solution

Have two stacks: one is the inbox, one is the outbox.

Insert(element, inbox stack, outbox stack): would work in the following matter: first add the element to the inbox stack. Then once finished, pop the element from the inbox stack and add them to the outbox stack.

Time Complexity: 2*O(1) = O(1) / Space Complexity: O(n)

Remove(element, outbox stack): would just pop the elements from the outbox stack.

Time Complexity: O(1) / Space Complexity: O(n)

Problem 9

Design an algorithm, given the above two lists of blocks, that comes up with a list of blocks on the server for which no block on the client can possibly match. (We are not asking you to describe how rsync actually works, only to describe an algorithm that could work.)

*More details about problem on course website

Solution

```
CheckBlocks(s_cs, c_cs): // s_cs = server checksum , c_cs = client
    no_match[]          //                          checksum
    bool flag
    for i = 0 to sizeof(s_cs)-1:
        for j to sizeof(c_cs)-1:
            if s_cs[i] == c_ck[j]:
                flag = true
        if flag != true:
            Insert(s_cs[i], no_match)
        flag = false
    return no_match
```