Q4 a)

```
newSkipLevel()
returns a skip list with -∞ and ∞ as the two nodes
 1. x = Node(-∞)
 2. y = Node(∞)
 3. x.after = y
 4. return x

insertAfter(p, k)
p: The new node will be embedded after this node
k: Element to insert
returns the node of element k
 1. x ← Node(k)
 2. x.after ← after(p)
 3. p.after ← x
 4. return x

skipify(A)
A: A sorted array in ascending order
returns a deterministic skip list with elements from A, of height
log(A.size)
 // Get an initial skip list with -∞ and ∞
 1. L ← newSkipLevel()
 2. p ← L
    // Add all elements to first level. Complexity O(n)
 3. for i ← 0 to A.size-1 do
 4.     p ← insertAfter(p, A[i])
 // Loop until where we get to a level with just -∞ and ∞. Complexity O(n)
 5. while after(L) != null and after(after(L)) != null do
 6.     M ← newSkipLevel()
 7.     q ← M
        // Link negative infinity to the level below
 8.     q.below ← L
 9.     p ← after(after(L))
        // Promote every next element in current level
10.     while p != null and after(p) != null do
11.         q ← insertAfter(q, key(p))
12.         q.bottom ← p
            // Break if next element is ∞
13.         if after(after(p)) = null do
14.             p ← after(p)
14.             break
13.         p ← after(after(p))
        // Link positive infinity to the level below
14.     q.below ← p
15.     L ← M
16. return L
```

b)

Balance skip list: Enumerate skip list levels from 0 to s with level 0 being the bottom-most level (containing all elements) and level s being the top-most level (containing only -∞ and ∞). Excluding -∞ and ∞:

- Level 0 contains all n elements
- Level `i+1` (for all `0 < i < s-1`) contains the alternate elements from level i

Nodes in different levels are linked as they would be in the skip list discussed in class.

Best case: `O(1)` when element at index $2^{k-1} - 1$ is accessed (not including -∞ and ∞). This is because this element exists in the second level of skip list and is found in the second iteration.

Worst case: The search algorithm needs to go down all levels (height) of the skip list which is `log(n)`, therefore the runtime is `O(log n)`.

c)

Please see below.

d)

For worst case, the LSBL would contain only 2 BSLs of `n/2` elements each. On insertion, both would be replaced by a bigger BSL including the inserted element generated using `skipify`. Since the total number of elements after insertion is n+1 and we can perform a merge-sort kind of merging and insertion, as we saw in part a, the run time would be `Θ(n+1)`.

e)

```
LBSLSearch(L, k)
L: an LBSL
k: item to be searched
returns a tuple (BSL, Stack) if key was found, after(top(tuple[1])) contains
the key node; returns null if not found
 1. for every M ← BSL in L do
 2.     S ← skip-search(M, k)
 3.     if key(after(S)) = k do
 4.         return (M, S)
 5. return null
```

The worst case runtime would be when we need to look through every BSL in the LBSL. The maximum number of BSLs is <= `log n`. Since the height of each BSL will be at most `log n`, from part b it follows that the worst case run time would be $O((\log n)^2)$.

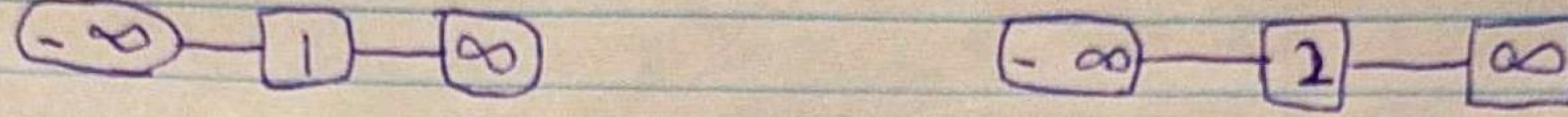
Theres is no advantage to searching the BSLs from largest to smallest or vice versa as there is no relation between the sorting of elements and the size of BSLs.

Every BSL contains another top level (-infinity, infinity) not shown for simplicity.

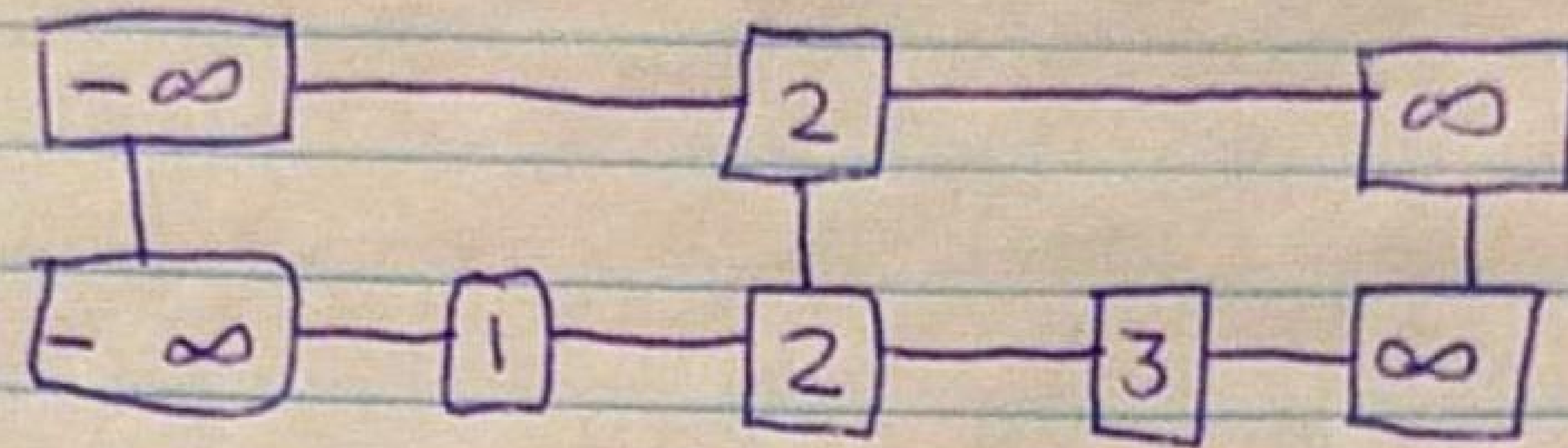
4. c

Insert 1:

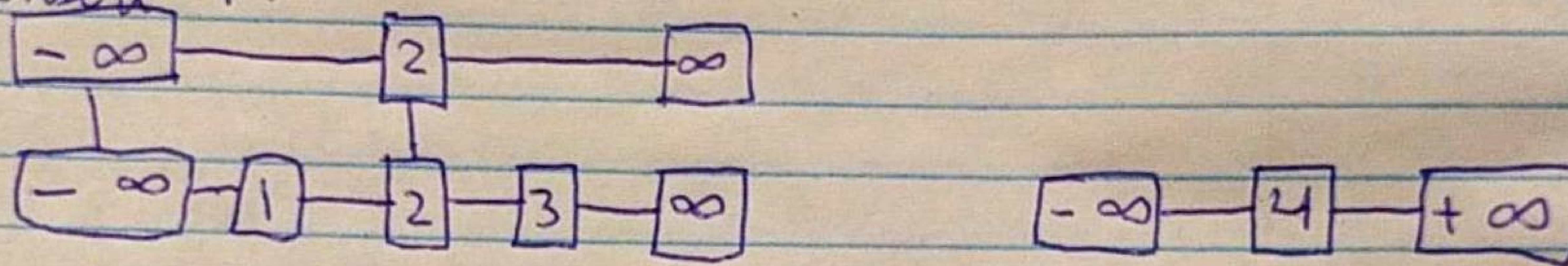
Insert 2:



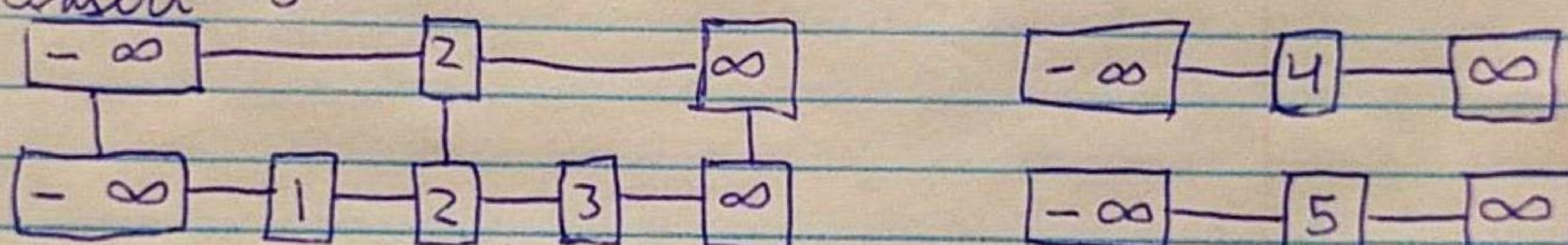
Insert 3:



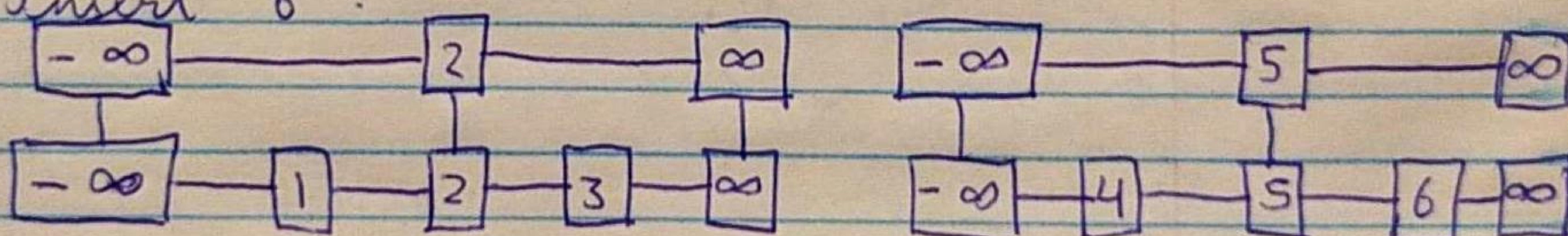
Insert 4:



Insert 5:



Insert 6:



Insert 7: