1.

where head ≡ front, tail ≡ rear

head: [1] :tail

head: [1] ⟷ [3] :tail

head: [1] ⟷ [3] ⟷ [2] :tail

head: [5] ⟷ [1] ⟷ [3] ⟷ [2] :tail

head: [1] ⟷ [3] ⟷ [2] :tail

head: [1] ⟷ [3] :tail

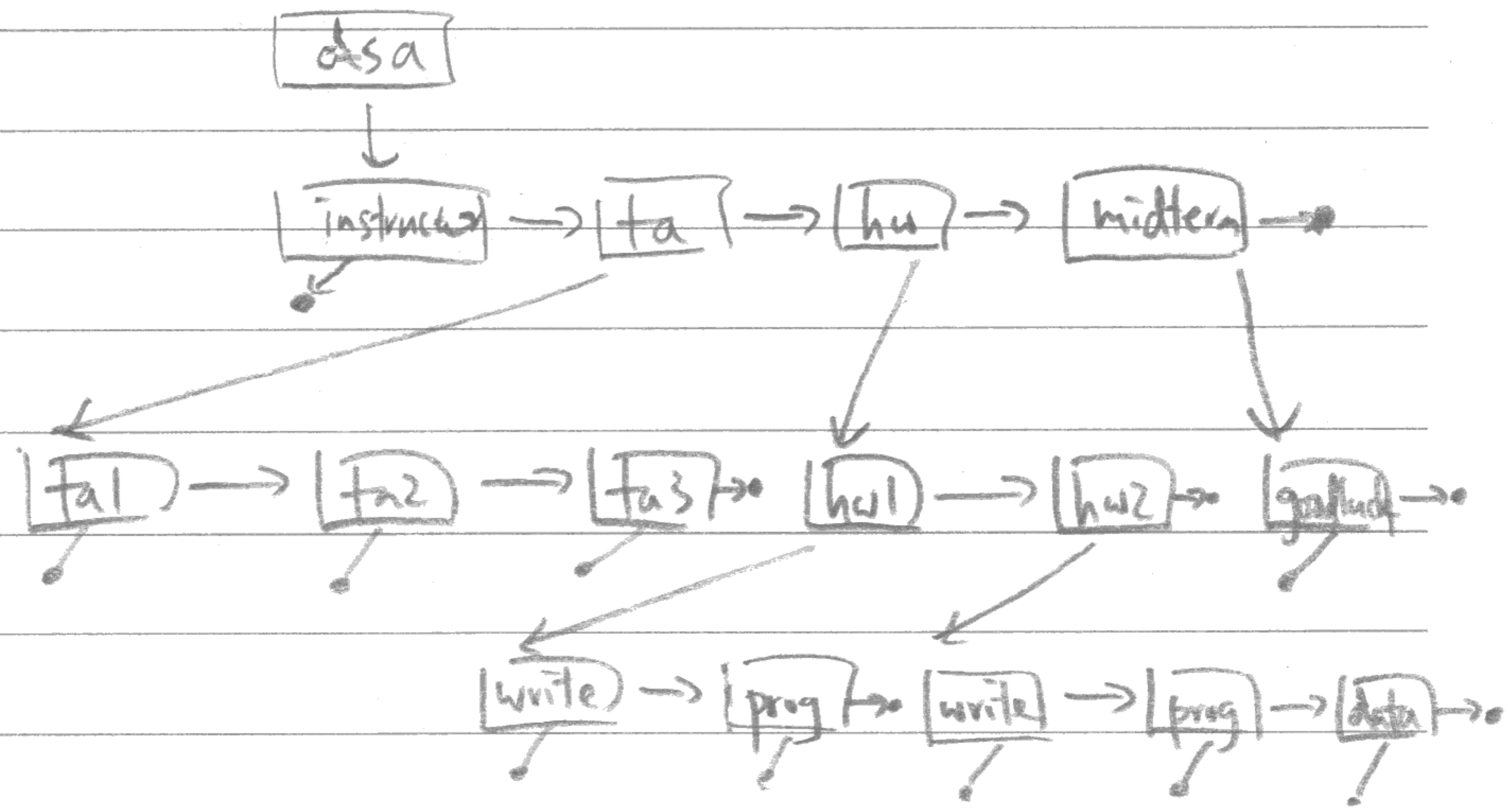
head: [1] ⟷ [3] ⟷ [7] :tail

head: [3] ⟷ [7] :tail

head: [3]

head: [3] ⟷ [6] :tail

2.



3. (a)

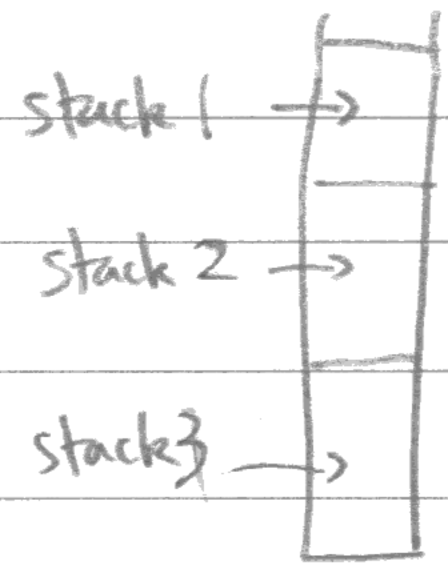
1	2
3	4
2	4
5	6
6	7
4	7

(b)

0	0
0	0
0	0
0	0
0	0
0	0

4 (a) store all data on blue, while using green for temp operations

on the blue stack, the layout is like



We use three additional counters c[1], c[2], c[3] to store the # elements in each stack

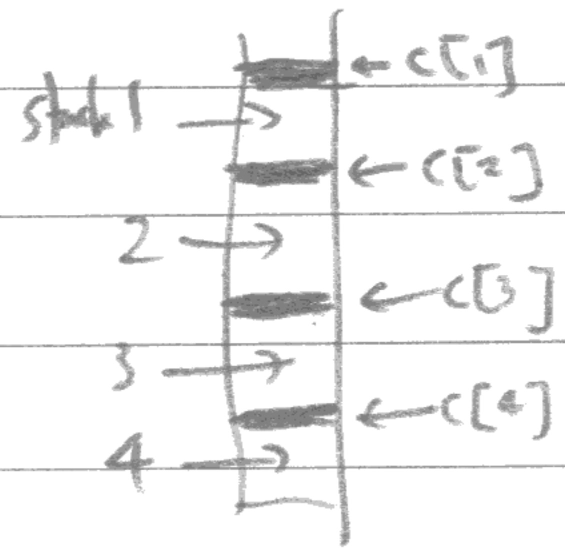
```
function operate on stack (num)
    for i ← 1 to num-1
      for t ← 1 to c[i]
        push "pop from stack blue" to stack green

function push to stack (num, element)
    operate on stack (num)
    push element to stack blue
    c[num] ← c[num] + 1
    cleargreen()

function pop of stack (num)
    operate on stack (num)
    pop element from stack blue
    c[num] ← c[num] - 1
    cleargreen()
    return the popped element

function cleargreen()
    while stack green not empty
        push "pop from stack green" to stack blue
```

(b)



We use a similar scheme, but put C[i] on the top of each stack segment instead

operation on stack can be changed such that the #elements to be popped is determined by popping C[i] first

push to stack can be changed to pop C[num] first, do the pushing, and then push the updated C[num] to the stack before cleargreen

pop of stack can be done similarly to push to stack

5.
```
res ← 0
up ← u , vp ← v
while up ≠ NULL and vp ≠ NULL
    if up → index == vp → index
        res ← res + (up → value − vp → value)^2
        up ← (up → next) , vp ← (vp → next)
    else if up → index < vp → index
        res ← res + (up → value)^2
        up ← (up → next)
    else
        res ← res + (vp → value)^2
        vp ← (vp → next)
while up ≠ NULL
    res ← res + (up → value)^2 , up ← (up → next)
while vp ≠ NULL
    res ← res + (vp → value)^2 , vp ← vp → next
return res
```

define unvisited being the # of non-zero terms in $\underline{u}$ and $\underline{v}$ that have not been calculated for res, we see that unvisited is either discounted by 1 or 2 in each iteration of (any of the) three "while"s. Each iteration of while only has constant # of primitive operations, So the time complexity is

$O(\text{initial unvisited}) = O(\#nz \text{ in } \underline{u} \text{ and } \underline{v})$

6.(a) Because $1 + |a_k n^k + a_{k-1} n^{k-1} + \cdots + a_0| = O(n^k)$

$\exists\, n_1, c_1$ s.t.

$$1 + |a_k n^k + \cdots + a_0| \leq c_1 \cdot n^k \quad \text{for } n \geq n_1$$

that is

$$f(n) \leq \log c_1 + g_k(n) \quad \text{for } n \geq n_1$$

because $g_k(n) = \log_2 n^k \geq 1$ for $n \geq 2$

$$\max(\log c_1, 1)\, g_k(n) \geq \max(\log c_1, 1) \quad \text{for } n \geq 2$$

Let $n_0 = \max(n_1, 2)$

$$\begin{aligned} f(n) &\leq \log c_1 + g_k(n) \\ &\leq \max(\log c_1, 1) + g_k(n) \\ &\leq \max(\log c_1, 1)\, g_k(n) + g_k(n) \end{aligned}$$

Let $c = \max(\log c_1, 1) + 1 \geq 2 > 0$

we see that

$$f(n) \leq c\, g_k(n) \quad \text{for } n \geq n_0$$

that is, $f(n) = O(g_k(n))$

(b) By $f(n) = O(g_k(n))$

$\exists$ n_0, c s.t.

$f(n) \leq c \log_2 n^k$ for $n \geq n_0$

that is,

$f(n) \leq c \cdot k \log_2 n$ for $n \geq n_0$

let $c' = c \cdot k$, $n_0' = n_0$

we see that

$f(n) \leq c' \cdot g(n)$ for $n \geq n_0'$

that is, $f(n) = O(g(n))$

7. disprove

Let $f(n) = n$

$g(n) = \frac{1}{2}n$

so $f(n) = O(g(n))$ by choosing $n_0 = 1$, $c = 2$

Let $h(n) = 4^n$ which is non-decreasing

then $h(f(n)) = 4^n$

$h(g(n)) = 2^n$

if $h(f(n)) = O(h(g(n)))$

$\exists$ $n_0 > 0$ $c > 0$ s.t.

$4^n \leq c \cdot 2^n$ for all $n \geq n_0$

take $\log_2$ on both sides

$2n \leq \log_2 c + n$

$\Rightarrow$ $n \leq \log_2 c$

that is, take $n' = \max(n_0, \lceil \log_2 c + 1 \rceil)$

because $n' > \log_2 c$ $4^n > c \cdot 2^n$

because $n' \geq n_0$ $4^n \leq c \cdot 2^n$

$\rightarrow\leftarrow$

So $h(f(n)) \neq O(h(g(n)))$.

8. perform binary search, but changing the condition to be

```
if arr[mid] = NULL
    search for k in [left, mid)
else
    search for k in [mid, right)
```

binary search maintains the invarince
" to-be-searched within [left, right) "
we maintain the invarince
" k within [left, right) "
So correctness & time complexity
are both $\approx$ binary search

One small detail is to check whether
arr[0] is NULL first
If so, simply return 0
without going into the main algorithm
to maintain the invariance.