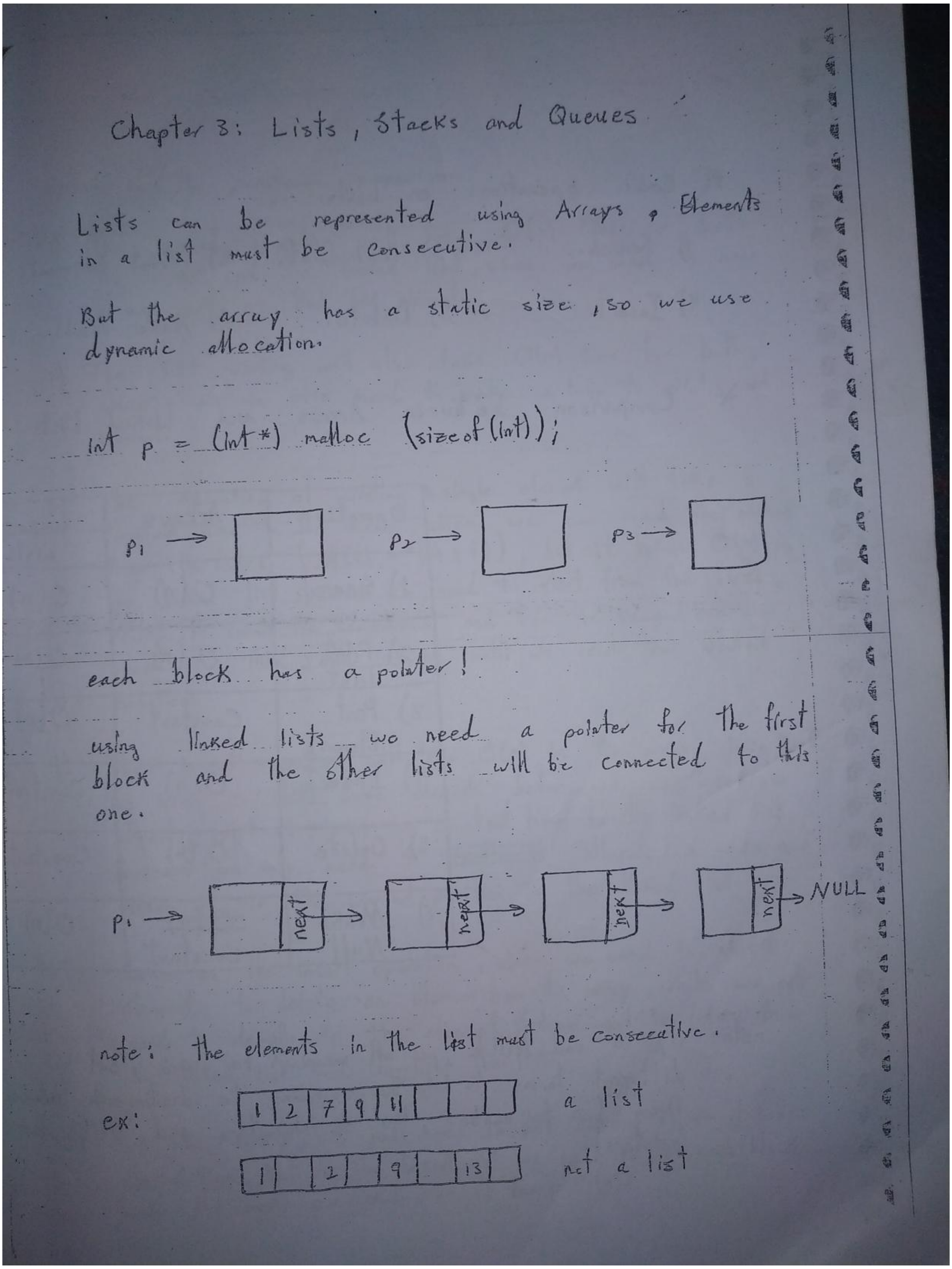


Chapter 3: Lists , Stacks and Queues

Lists can be represented using Arrays , Elements in a list must be consecutive.

But the array has a static size , so we use dynamic allocation.

```
Int p = (Int*) malloc (sizeof(Int));
```

p1 →
p2 →
p3 →

each block has a pointer !

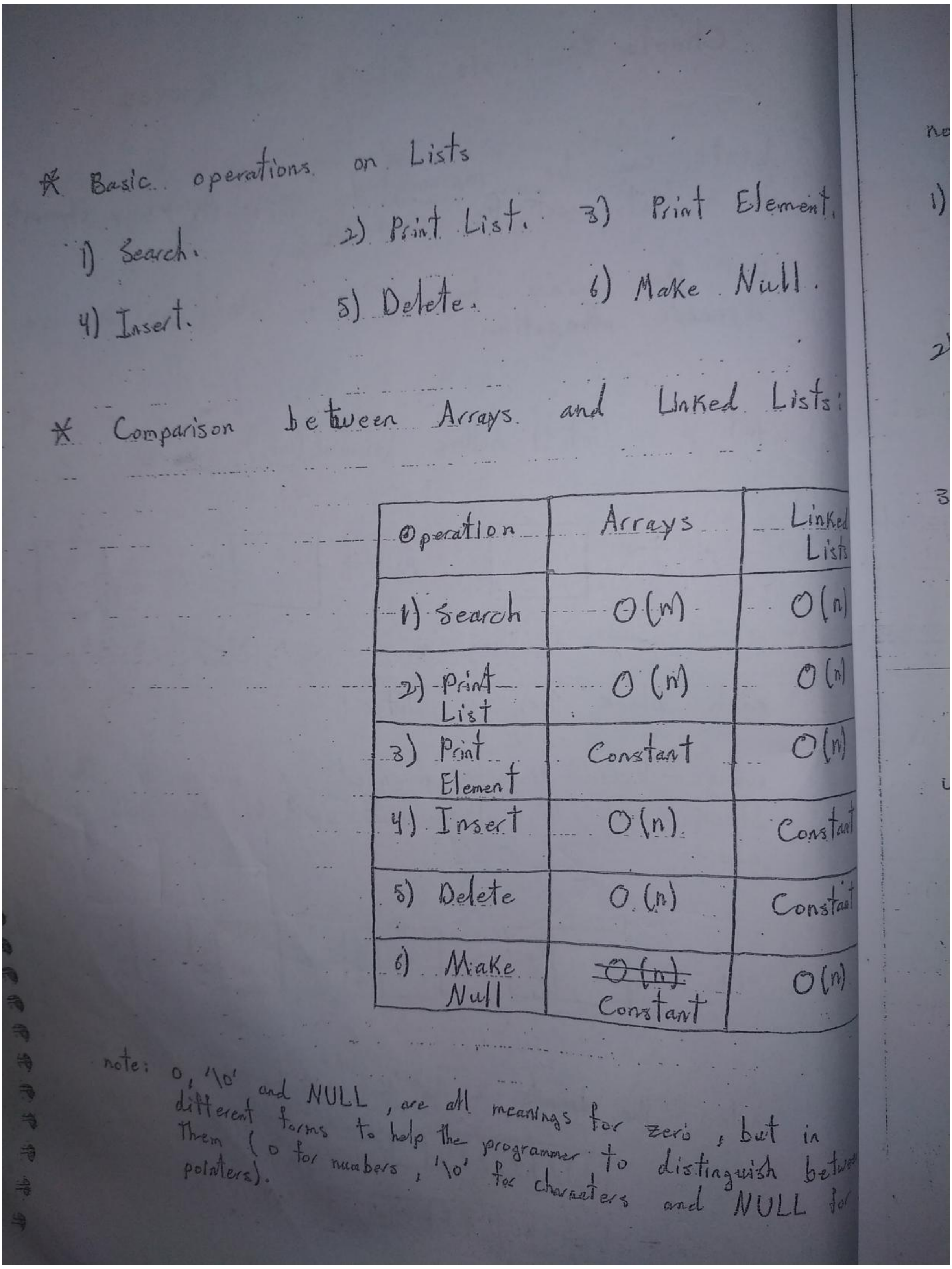
using linked lists we need a pointer for the first block and the other lists will be connected to this one.

p1 →
next
next
next
next
NULL

note : the elements in the list must be consecutive.

ex:

1	2	7	9	11				a list
1		2		9		13		not a list



* Basic operations on Lists

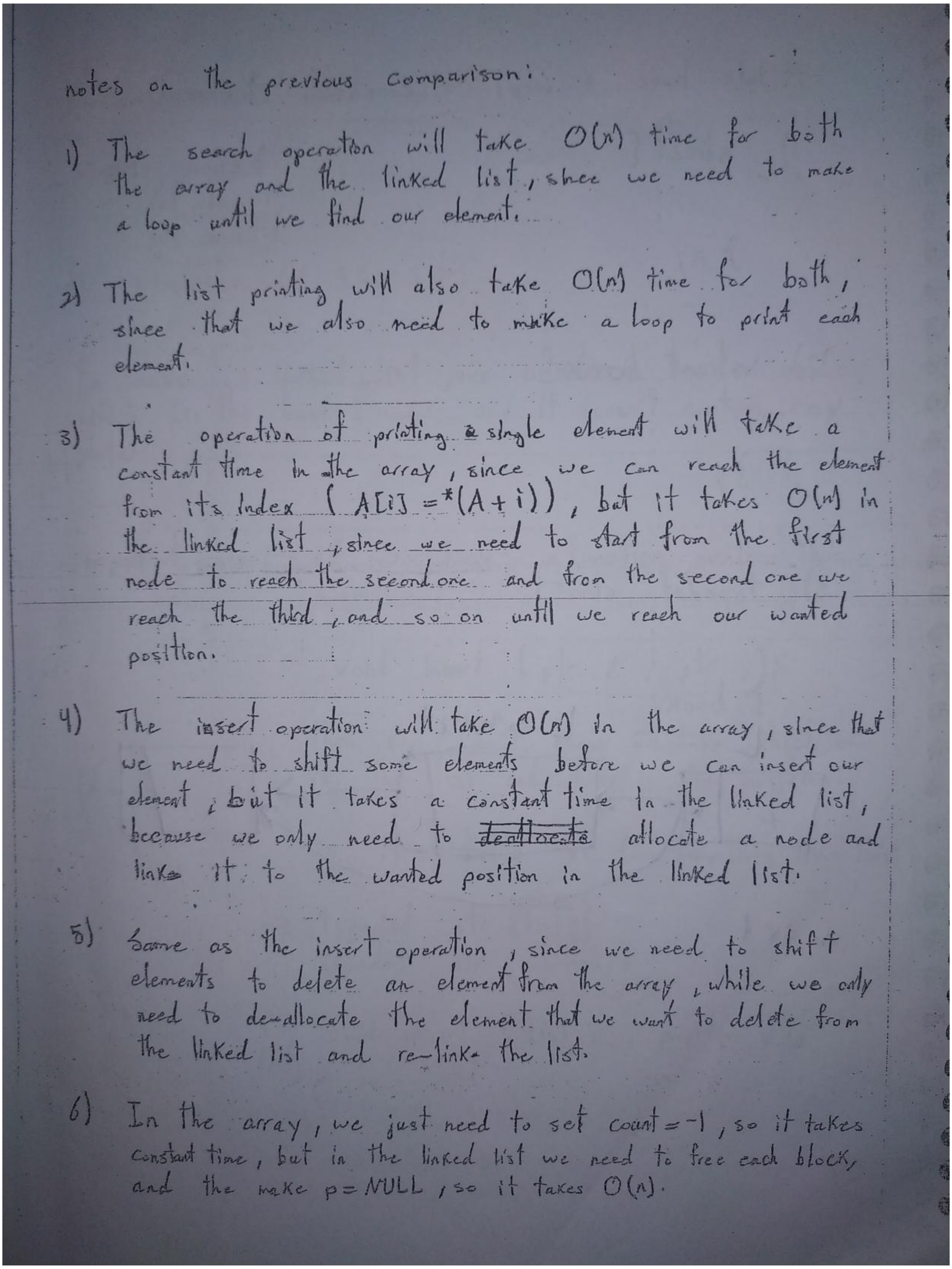
1) Search. 2) Print List. 3) Print Element.

4) Insert. 5) Delete. 6) Make Null.

* Comparison between Arrays and Linked Lists:

Operation	Arrays	Linked Lists
1) Search	O(n)	O(n)
2) Print List	O(n)	O(n)
3) Print Element	Constant	O(n)
4) Insert	O(n)	Constant
5) Delete	O(n)	Constant
6) Make Null	~~O(n)~~ Constant	O(n)

note: 0, '\0' and NULL , are all meanings for zero , but in different forms to help the programmer to distinguish betw them (0 for numbers , '\0' for characters and NULL for pointers).



notes on the previous comparison:

1) The search operation will take $O(n)$ time for both the array and the linked list, since we need to make a loop until we find our element.

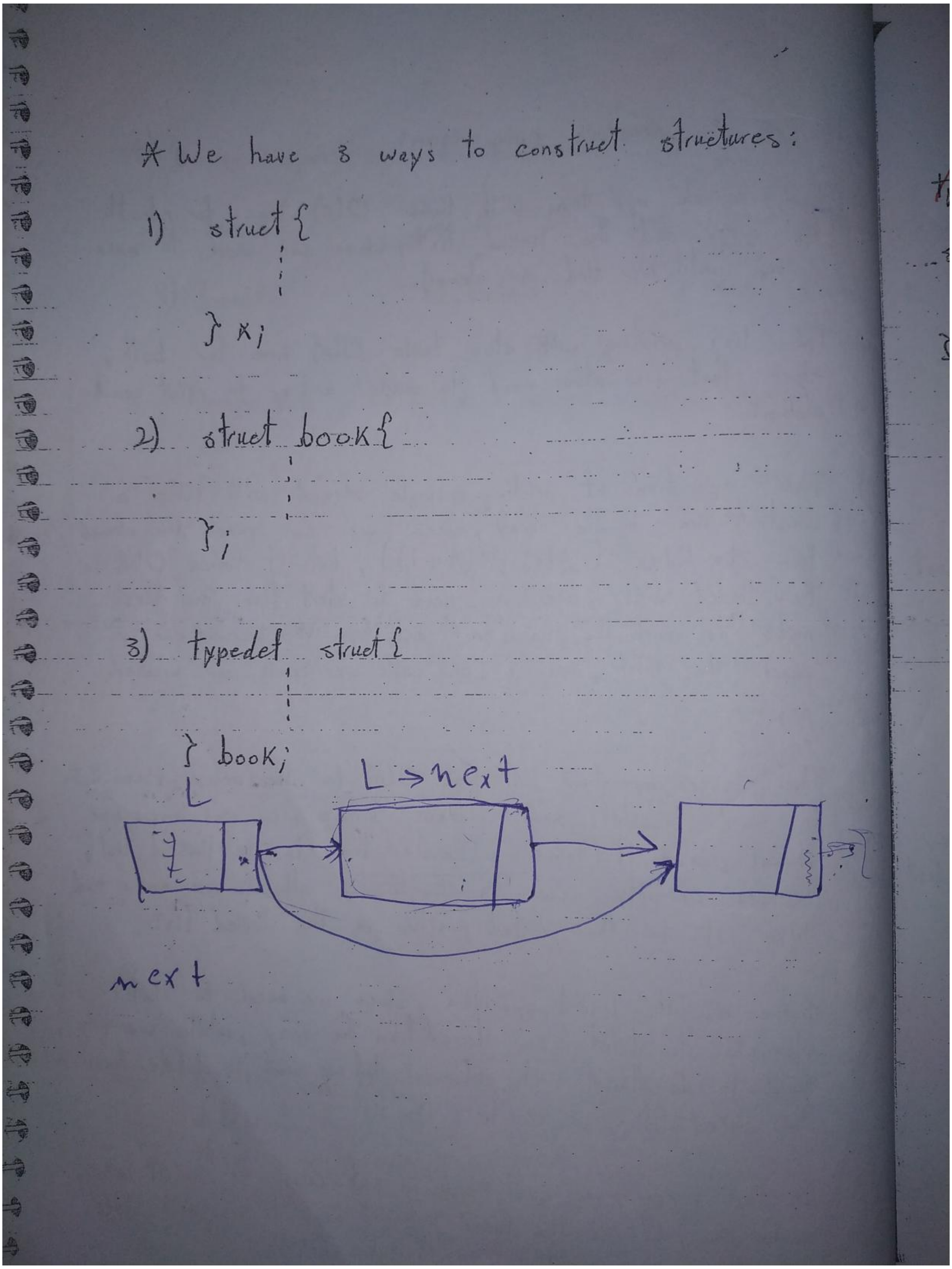
2) The list printing will also take $O(n)$ time for both, since that we also need to make a loop to print each element.

3) The operation of printing a single element will take a constant time in the array, since we can reach the element from its index ($A[i] = *(A+i)$), but it takes $O(n)$ in the linked list, since we need to start from the first node to reach the second one and from the second one we reach the third, and so on until we reach our wanted position.

4) The insert operation will take $O(n)$ in the array, since that we need to shift some elements before we can insert our element, but it takes a constant time in the linked list, because we only need to ~~deallocate~~ allocate a node and link it to the wanted position in the linked list.

5) Same as the insert operation, since we need to shift elements to delete an element from the array, while we only need to de-allocate the element that we want to delete from the linked list and re-link the list.

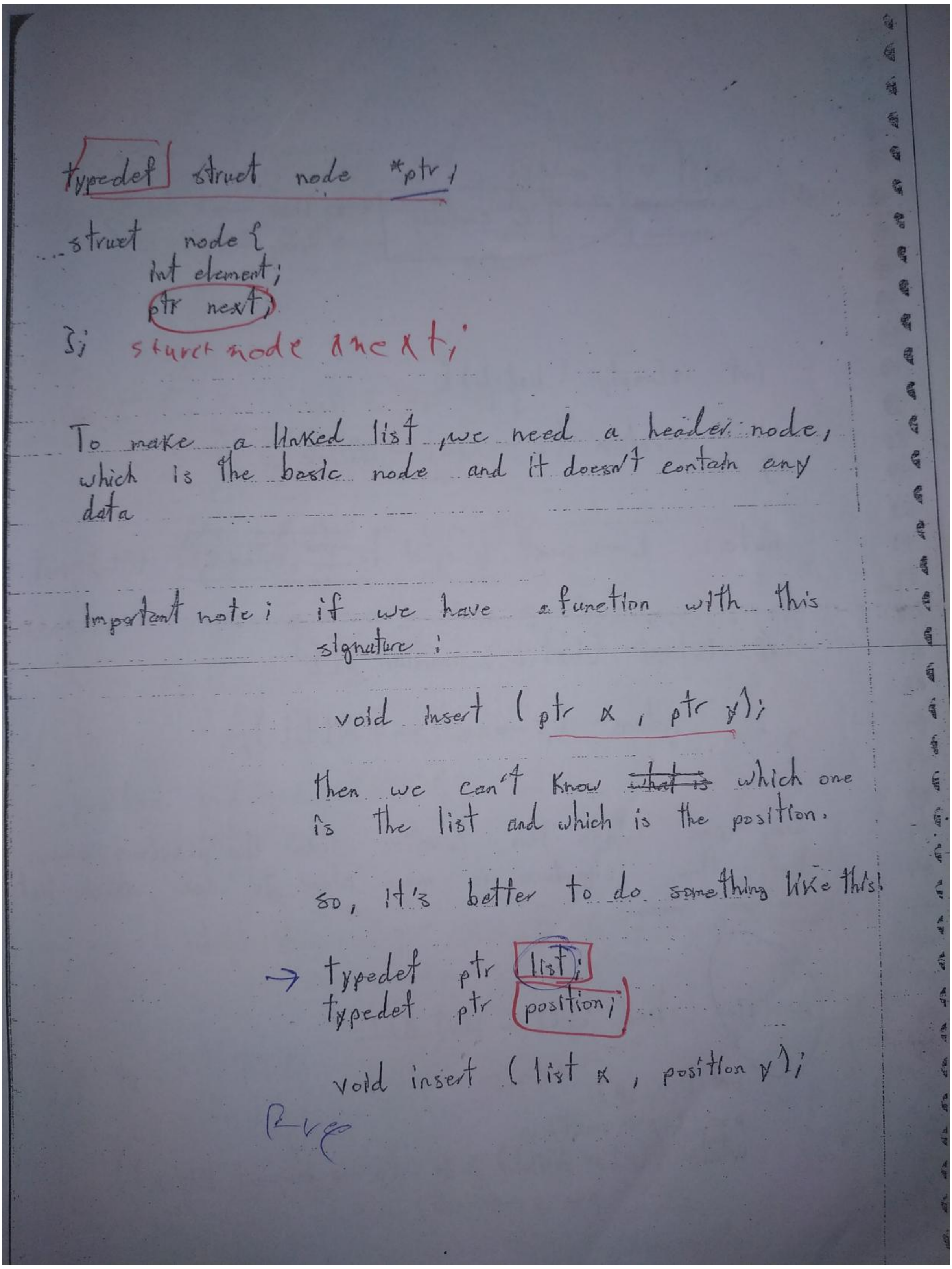
6) In the array, we just need to set $count = -1$, so it takes constant time, but in the linked list we need to free each block, and the make $p = NULL$, so it takes $O(n)$.



* We have 3 ways to construct structures:

```
1) struct {
       :
   } x;

2) struct book {
       :
   };

3) typedef struct {
       :
   } book;
```

L
L -> next
next



```
typedef struct node *ptr;

struct node {
      int element;
      ptr next;
};   struct node *next;
```

To make a linked list we need a header node, which is the basic node and it doesn't contain any data

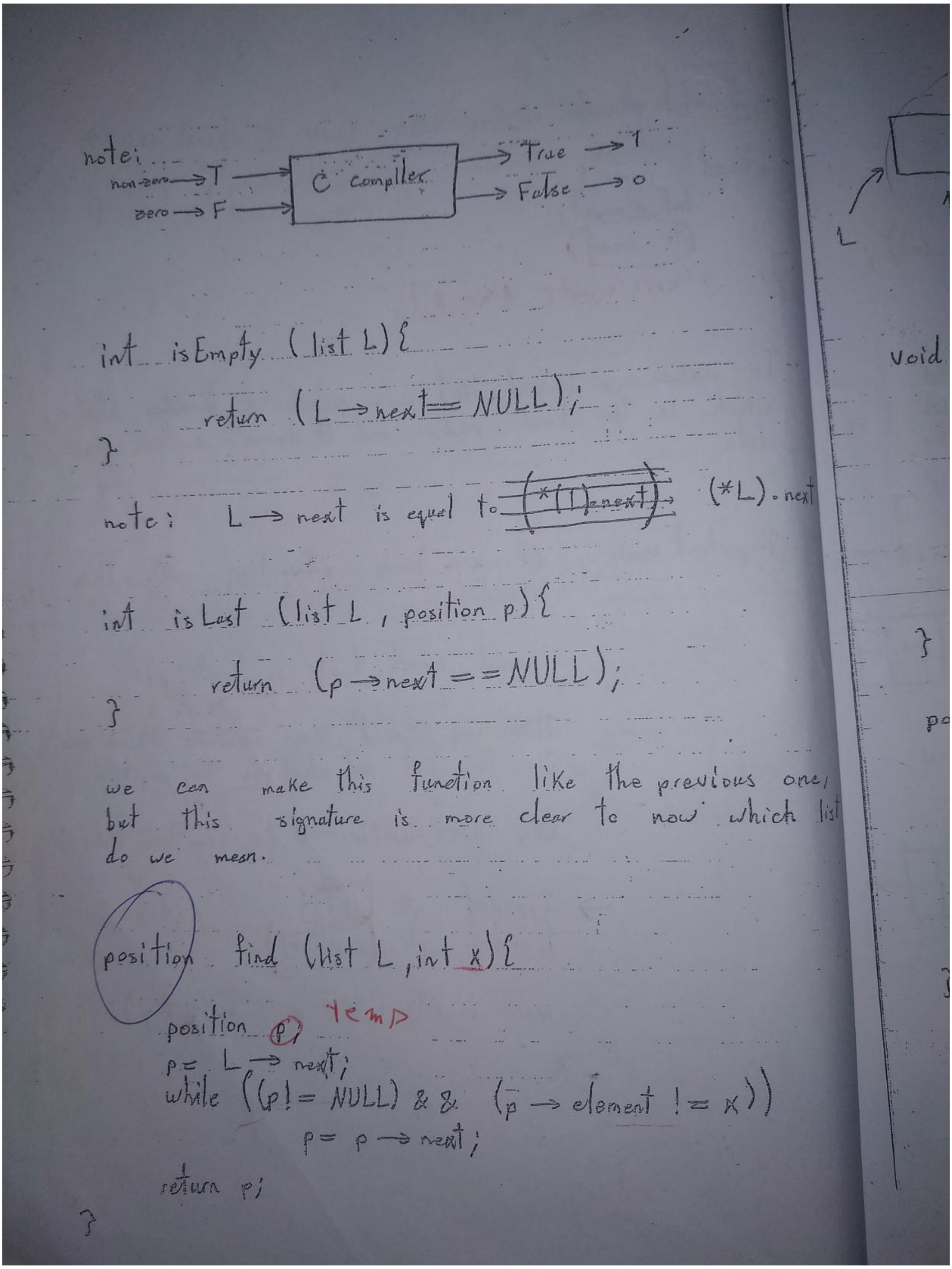
Important note: if we have a function with this signature:

```
void insert (ptr x, ptr y);
```

then we can't know ~~what is~~ which one is the list and which is the position.

so, it's better to do something like this:

```
→ typedef ptr list;
  typedef ptr position;

  void insert (list x, position y);
```



note:

non-zero → T → C compiler → True → 1

zero → F → C compiler → False → 0

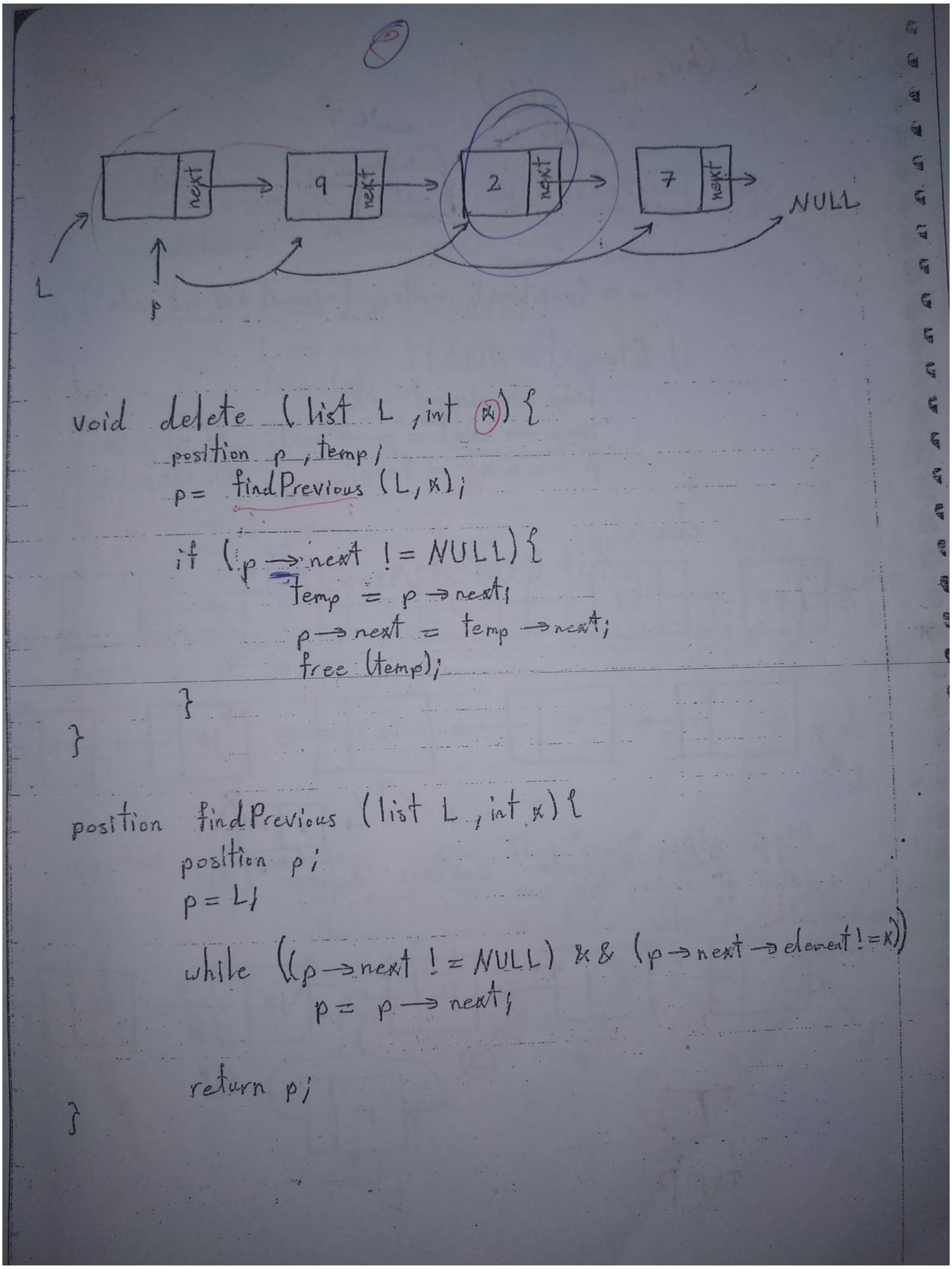
```
int  isEmpty  (list L){
        return  (L→next== NULL);
}
```

note: L → next is equal to ~~(*(L).next)~~ (*L).next

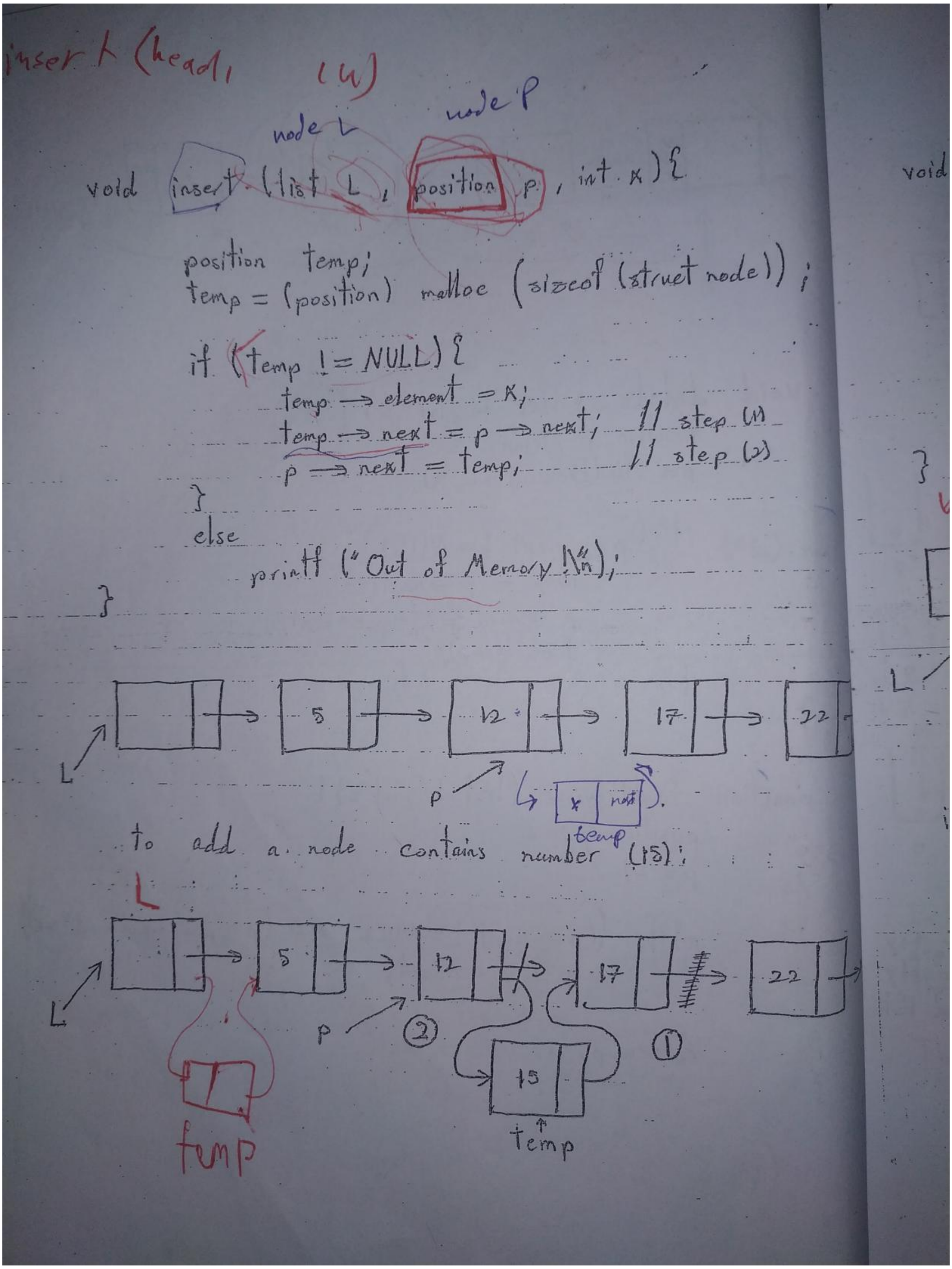
```
int  isLast  (list L , position p){
        return  (p→next ==NULL);
}
```

we can make this function like the previous one, but this signature is more clear to now which list do we mean.

```
position  find (list L ,int x){
     position  p;   temp
     p= L→ next;
     while ((p!= NULL) && (p → element != x))
             p= p → next;
     return p;
}
```


next
9
next
2
next
7
next
NULL
L
p

```
void delete (list L ,int x) {
    position p , temp ;
    p= findPrevious (L, x);

    if (p -> next != NULL) {
          Temp = p -> next;
          p -> next = temp -> next;
          free (temp);
    }
}

position findPrevious (list L , int x) {
    position p;
    p = L;
    while ((p -> next != NULL) && (p -> next -> element != x))
           p = p -> next;

    return p;
}
```



insert (head, (W)

node L   node P

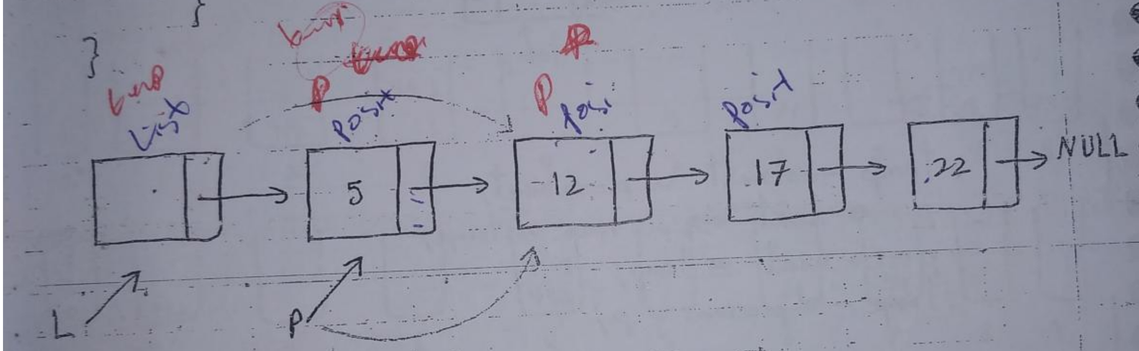
```
void insert (list L, position p, int x){

    position temp;
    temp = (position) malloc (sizeof (struct node));

    if (temp != NULL){
        temp -> element = x;
        temp -> next = p -> next;   // step (1)
        p -> next = temp;           // step (2)
    }
    else
        printf ("Out of Memory !\n");
}
```

L
5
12
17
22
p
x
next
temp

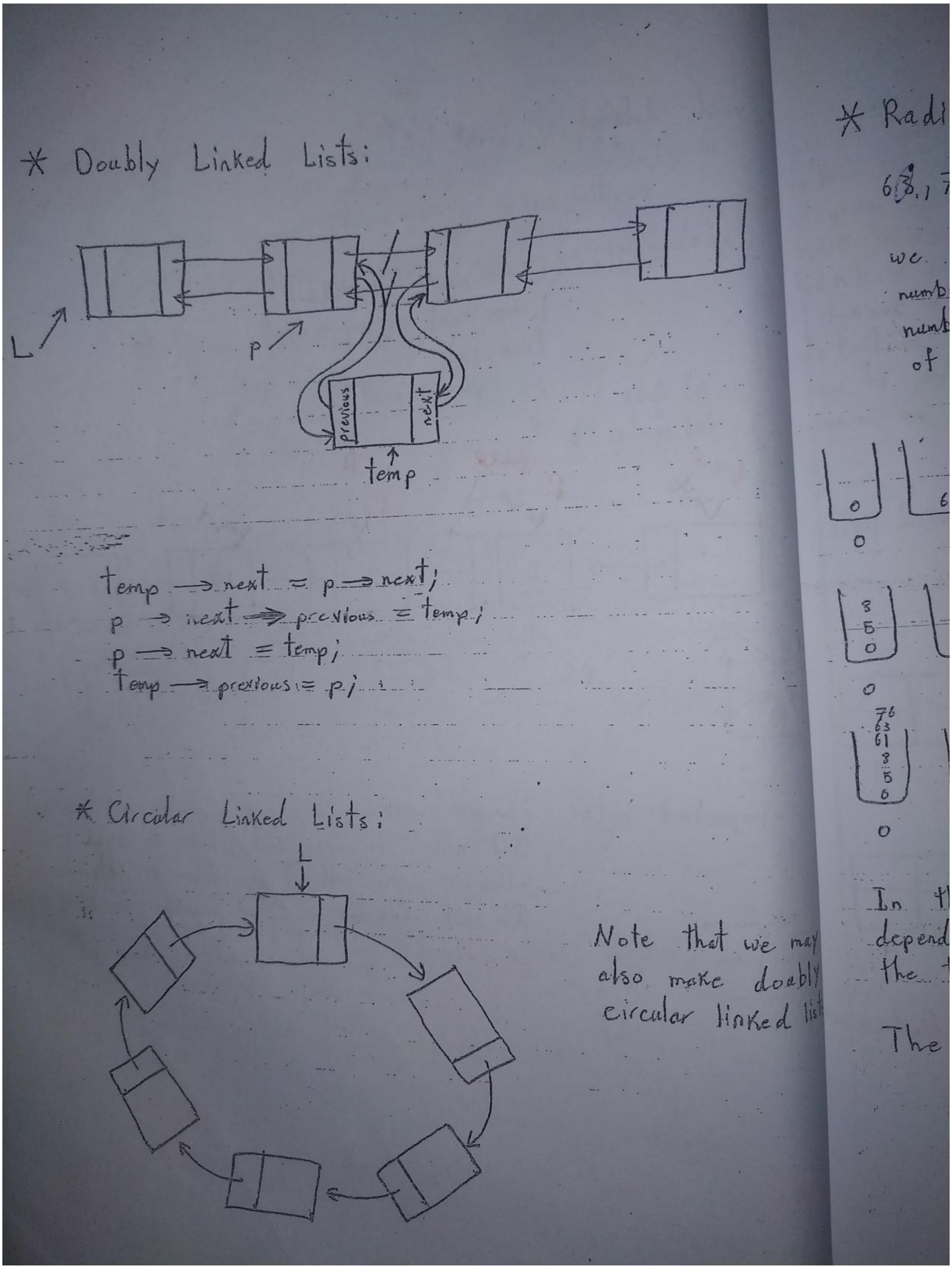
to add a node contains number (15);

L
L
5
12
17
22
p
②
①
15
temp
temp

```
void deleteList (list L){
    position p = L→next;
    while (L→next != NULL){
        L→next = p→next;
        free (p);
        p=L~~p~~→next;
    }
}
```



important note: always make the "next" pointer in the last node in any list points to NULL, otherwise, you will not be able to determine the last node or use the previous functions properly.



* Doubly Linked Lists:

L
p
previous
next
temp

```
temp → next = p → next;
p → next → previous = temp;
p → next = temp;
temp → previous = p;
```

* Circular Linked Lists:

L

Note that we may also make doubly circular linked list

* Radi

63,1 7

we
numb
numb
of

0
0
6
8
5
0
0
76
63
61
8
5
6
0

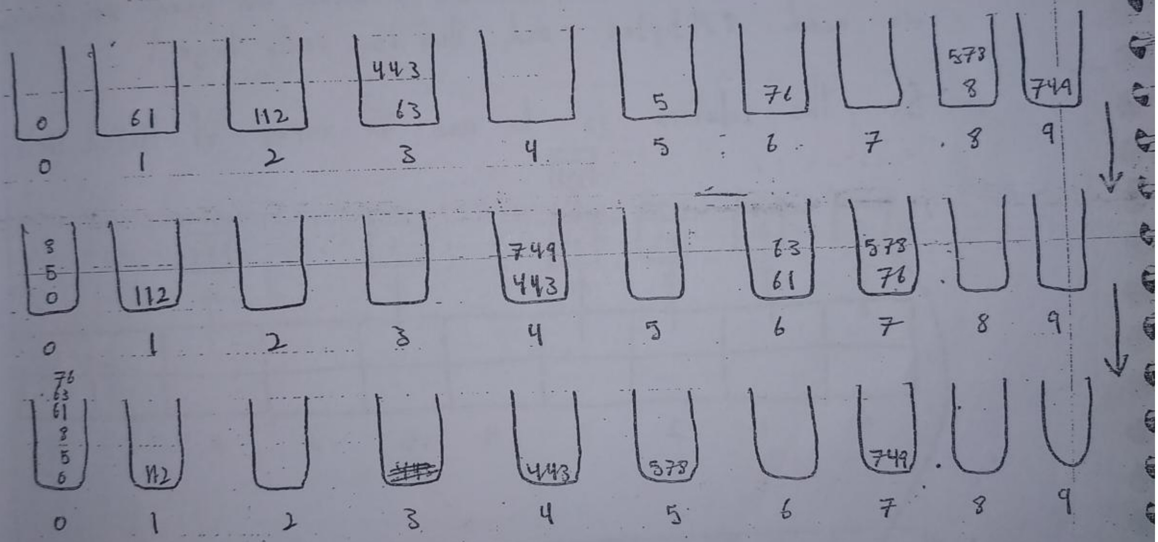
In th
depend
the

The

* Radix Sort:

63, 749, 5, 8, 76, 443, 0, 112, 61, 578

we need 3 steps ~~for~~ sort these numbers (the maximum number of digits in these numbers), and we need 10 buckets numbered from (0-9) also (since each digit has a range of 0-9).

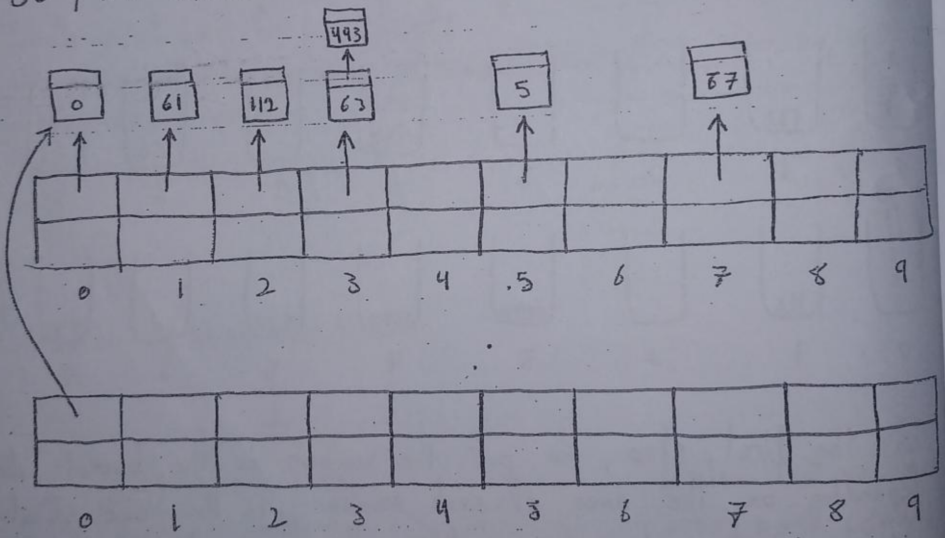


In the first step, we put the numbers in the buckets ~~depend~~ depending on the ones of each number, in the second step on the tens of each number, and in the third step on the hundreds.

The time needed is $O(n)$.

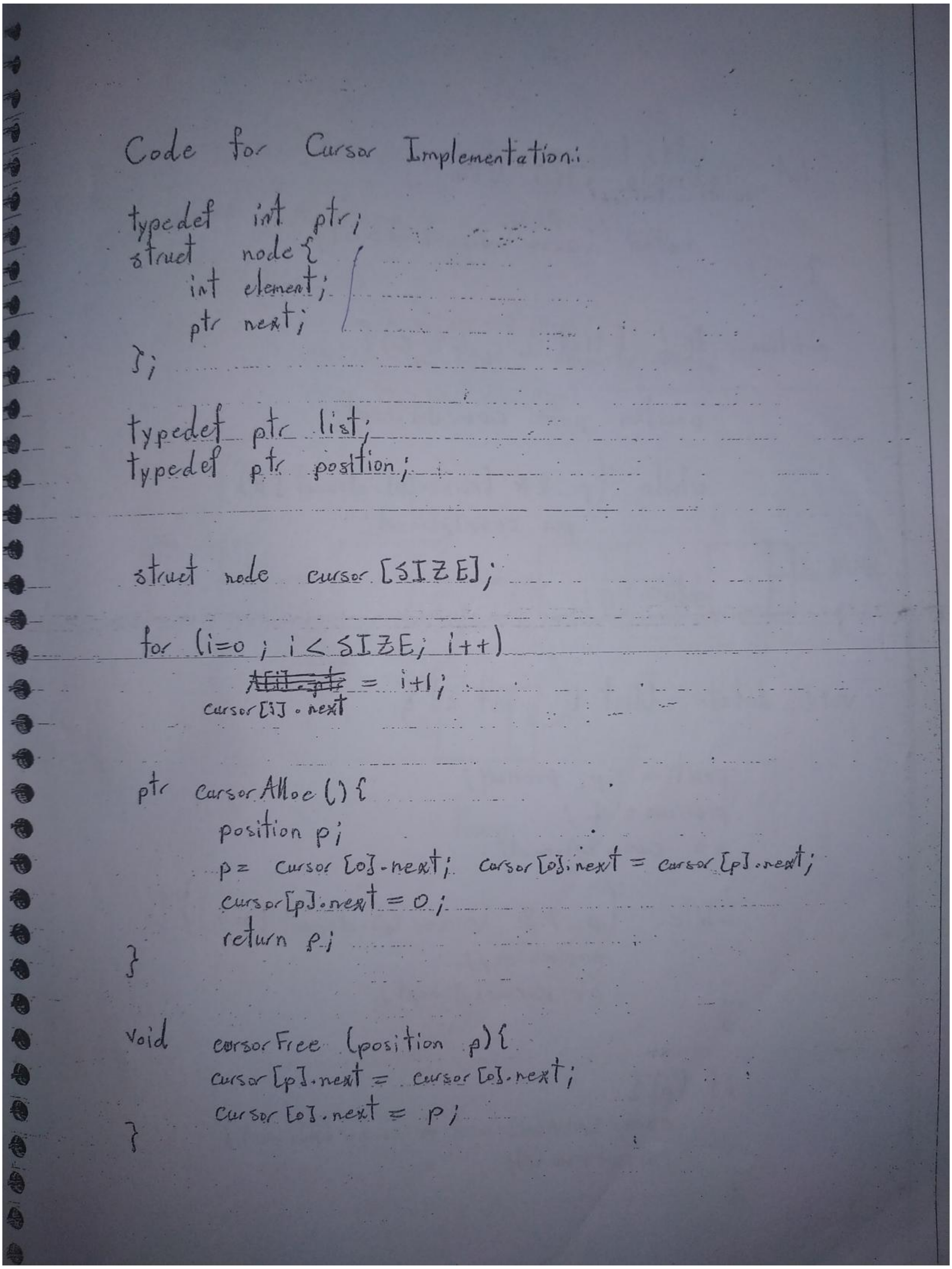
If we want to make this sort using arrays, then if we have 100 000 numbers that we want to sort then we will ne 10 arrays, each of size 100000 and let these array h an integer type then those array will take size of 4(byt * 10 (arrays) * 100.000 (locations) = $4 * 10^6$ bytes = 4 M byt and since we have 3 steps here, we need 20 buckets (arrays) and thats because the buckets used in the first st can be reused in the third one (and the same thing if we need more than 3 steps). So, since we need 20 arrays we need 8 M bytes and that is really huge!

So, the solution is to use array of linked lists.



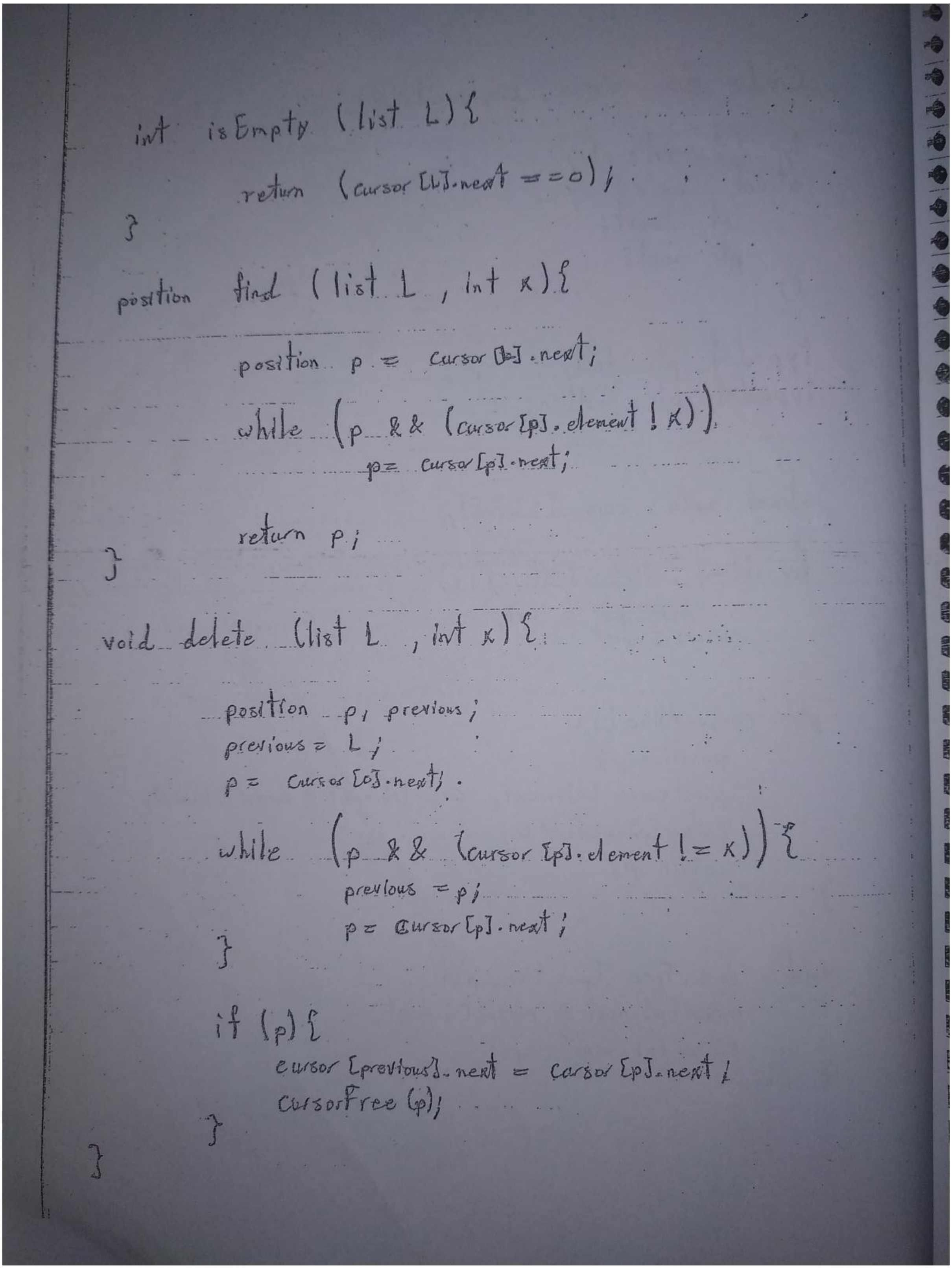
And we will need the following function to get each digit:

```
int getDigit (int n, int i){
        return (n/pow(10,i))%10 ;
}
```



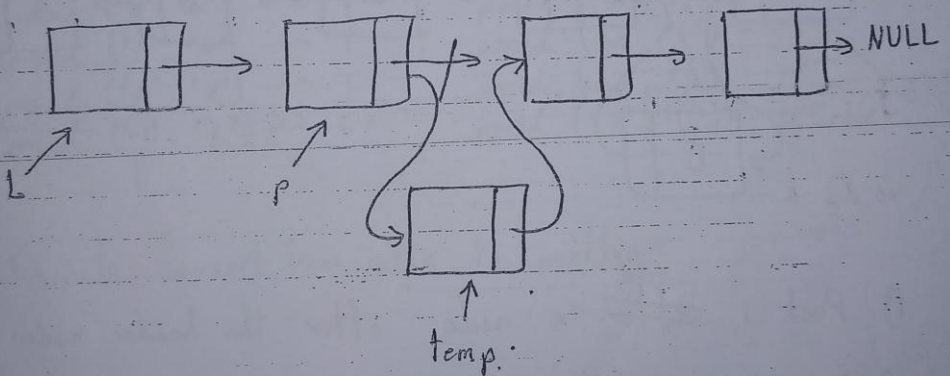
Code for Cursor Implementation:

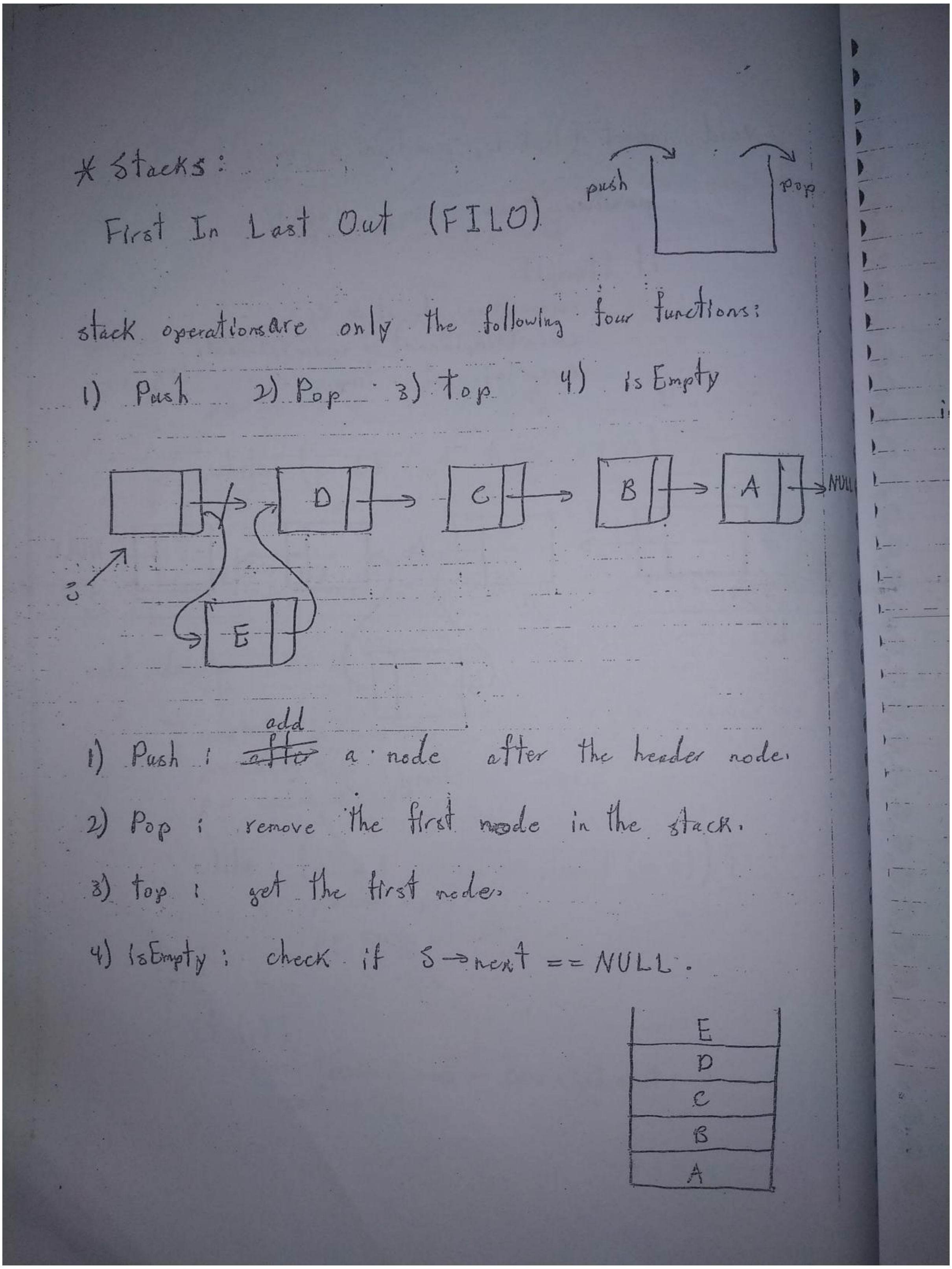
```
typedef int ptr;
struct node{
    int element;
    ptr next;
};

typedef ptr list;
typedef ptr position;

struct node cursor[SIZE];

for (i=0 ; i<SIZE; i++)
        ~~A[i].ptr~~ = i+1;
    cursor[i] . next

ptr cursorAlloc(){
      position p;
      p= cursor[0]-next; cursor[0].next = cursor[p].next;
      cursor[p].next = 0;
      return p;
}

void   cursorFree (position p){
      cursor[p].next = cursor[0].next;
      cursor[0].next = p;
}
```



```
int isEmpty (list L){
        return (cursor[L].next ==0);
}

position find (list L , int x){

        position p = cursor[L].next;

        while (p && (cursor[p].element ! x))
                p= cursor[p].next;

        return p;
}

void delete (list L , int x){

        position p, previous;
        previous = L;
        p = cursor[0].next;

        while (p && (cursor[p].element != x)){
                previous = p;
                p = cursor[p].next;
        }

        if (p){
            cursor[previous].next = cursor[p].next;
            cursorFree (p);
        }
}
```

```
void insert (list L, position p , int x){

    position temp = cursorAlloc();

    if (temp){
        cursor[temp].element = x;
        cursor[temp].next = cursor[p].next;
        cursor[p].next = temp;
    }
```





* Stacks:

First In Last Out (FILO)

push
pop

stack operations are only the following four functions:

1) Push 2) Pop 3) top 4) is Empty

D
C
B
A
NULL
S
E

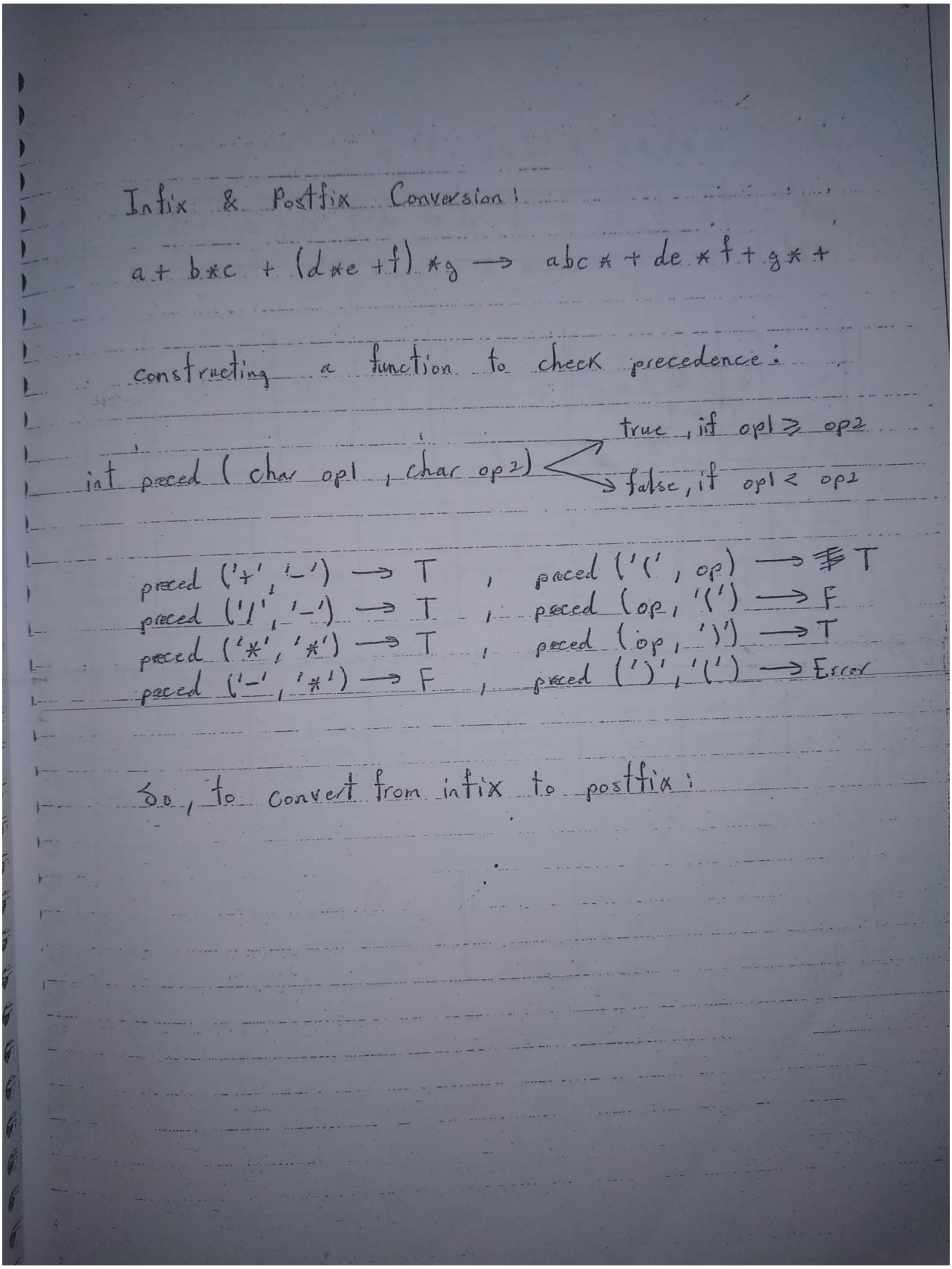
1) Push : ~~after~~ add a node after the header node.

2) Pop : remove the first node in the stack.

3) top : get the first node.

4) IsEmpty : check if S→next == NULL.

E
D
C
B
A



Infix & Postfix Conversion:

a + b*c + (d*e + f) *g → abc* + de*f+g*+

constructing a function to check precedence:

$$\text{int preced (char op1 , char op2)} \begin{cases} \text{true , if op1} \geq \text{op2} \\ \text{false, if op1} < \text{op2} \end{cases}$$

preced ('+', '-') → T , preced ('(', op) → T

preced ('/', '-') → T , preced (op, '(') → F

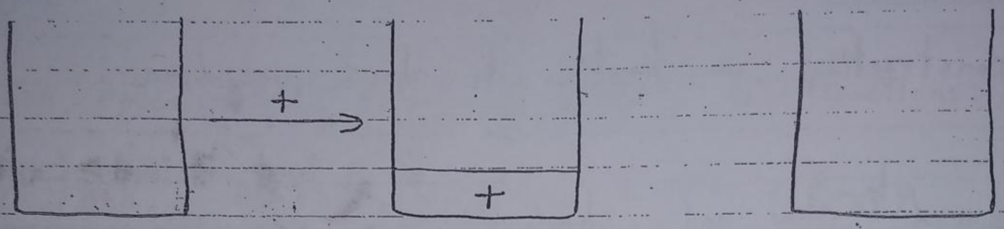
preced ('*', '*') → T , preced (op, ')') → T

preced ('-', '*') → F , preced (')', '(') → Error

So, to convert from infix to postfix:

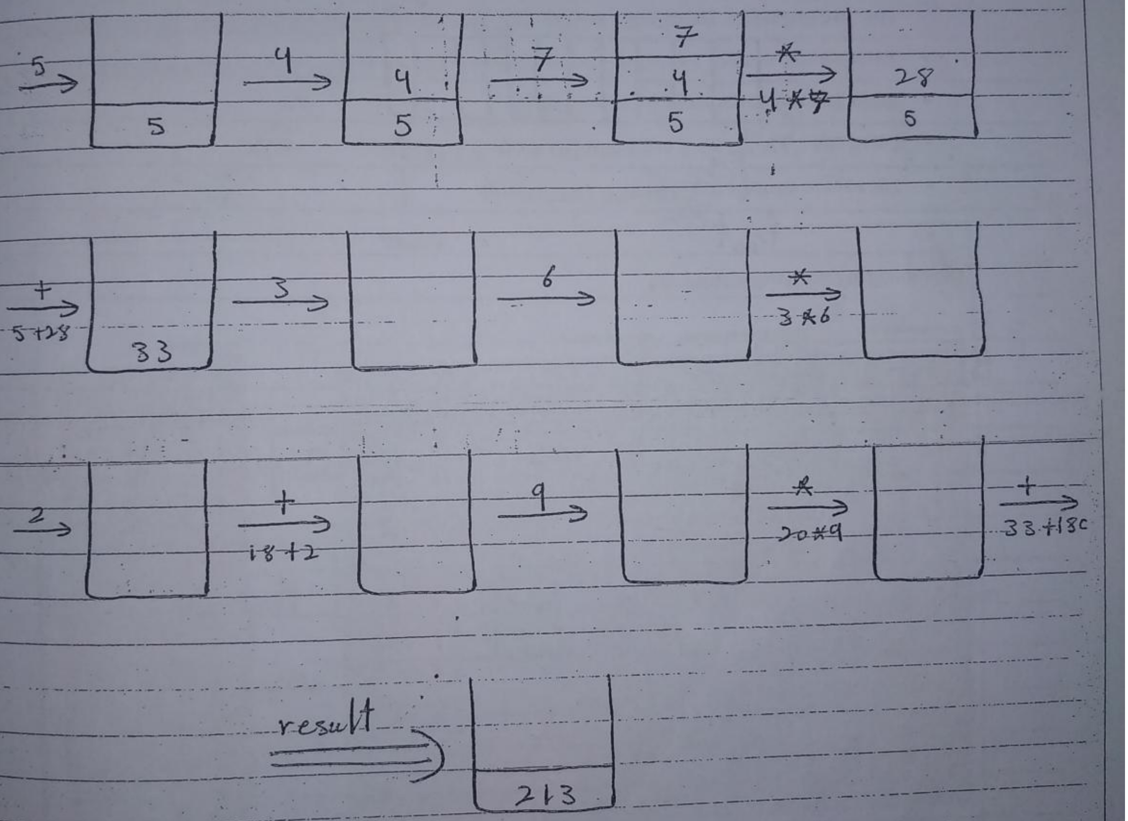
input : a + b * c + (d * e + f) * g

output : $\boxed{a\ b\ c\ *\ +\ d\ e\ *\ f\ +\ g\ *\ +}$
(or string)



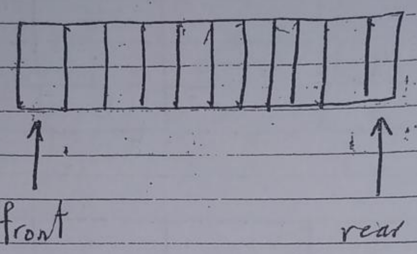
ex: 5 + 4 * 7 + (3 * 6 + 2) * 9 (infix)

⟹ 5 4 7 * + 3 6 * 2 + 9 * + (post fix)



* Queue:

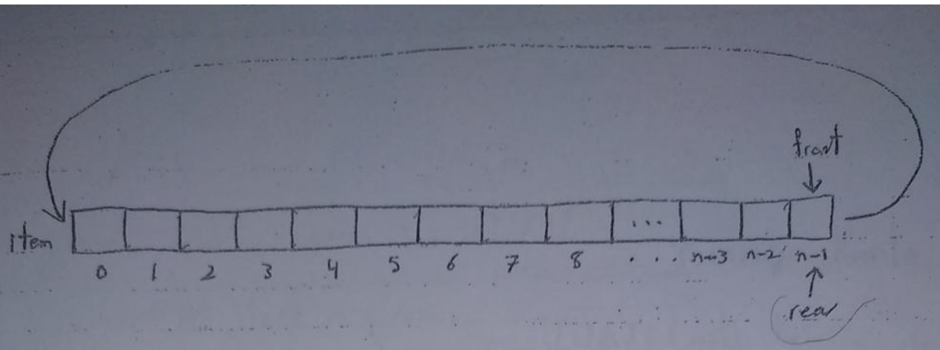
FIFO (First In First Out)



Basic Operations:

1) Insert (enqueue) 2) Delete (Dequeue) 3) isEmpty

note: its better to insert from the rear and delete from the front

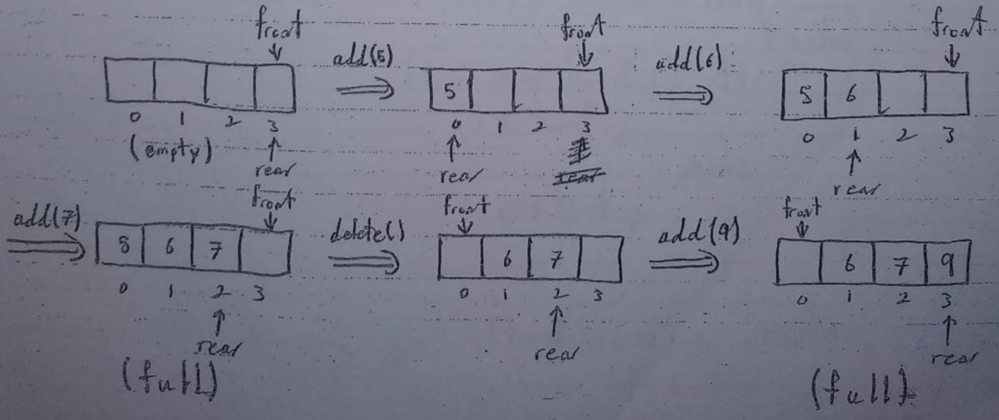


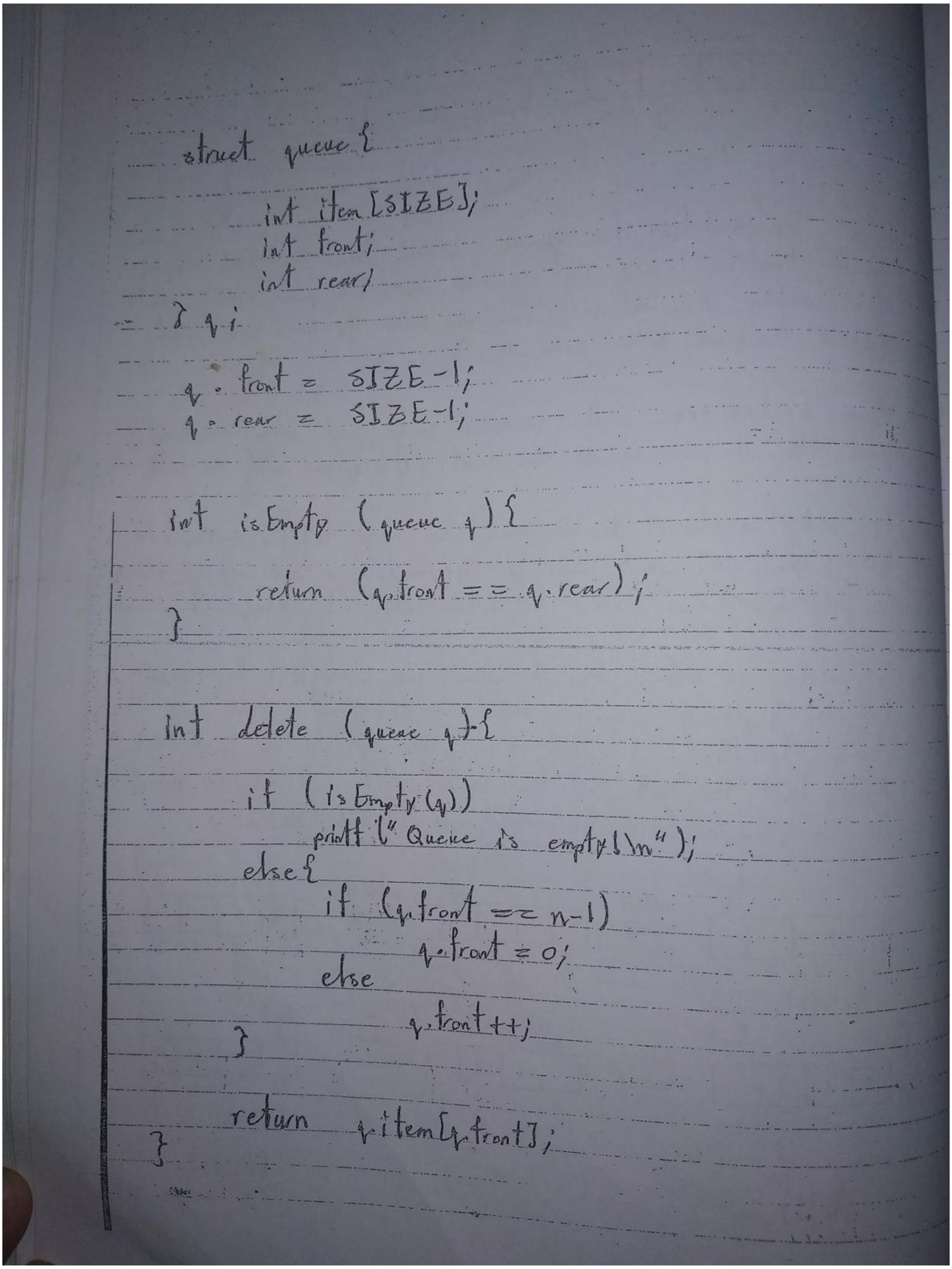
In our implementation for queues we will make the front=rear=n-1 at the begining, since that we must make the queue circular, that is after reaching the last index, we will return to index zero, and similarly, the index before zero will be n-1.

When front== rear, the queue is empty, and when the next index after the rear equals the front, the queue will be full.

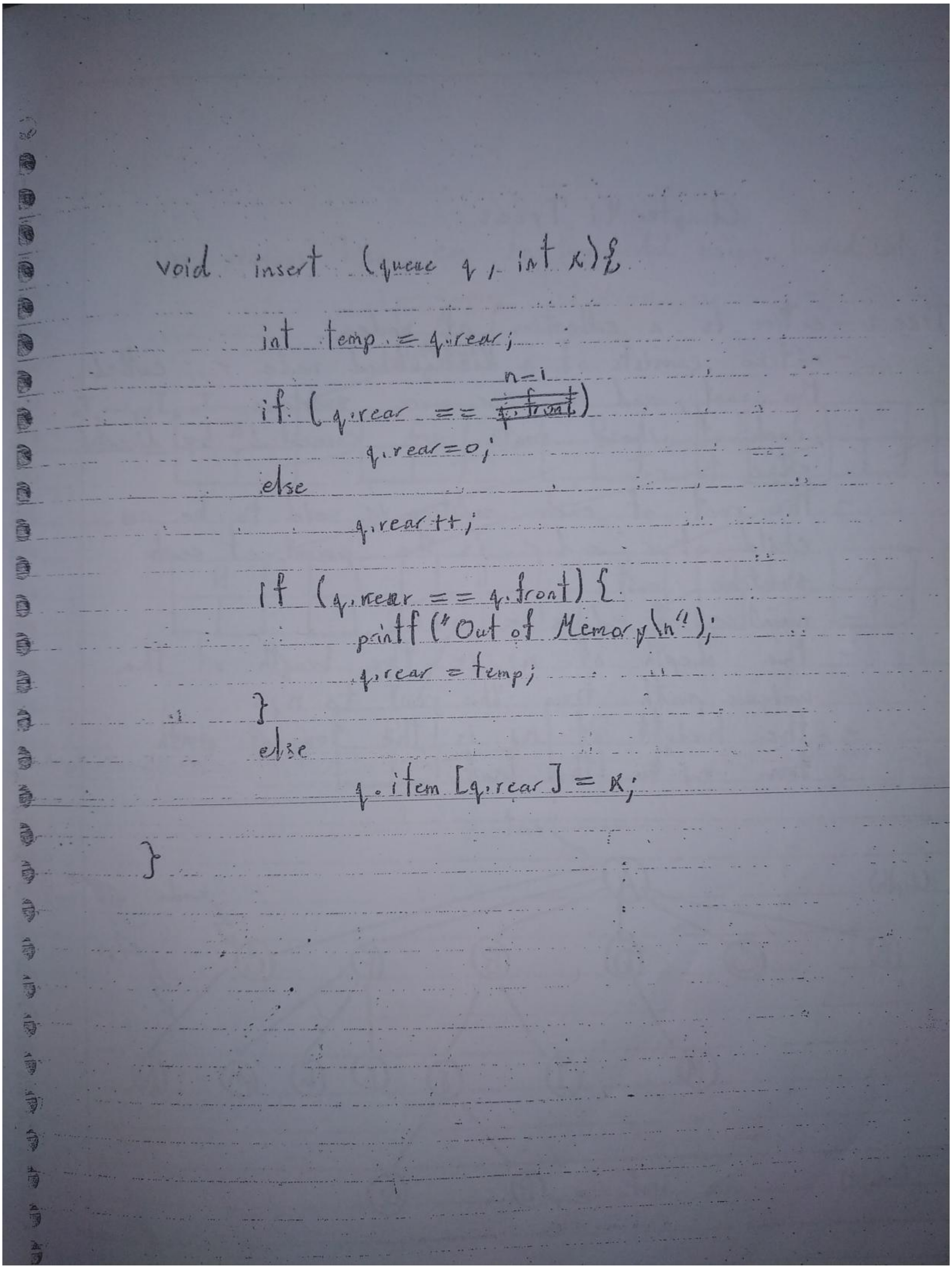
To add an element to the queue, we need first to make sure that it isn't full, and then we increment rear and save our element in item[rear] (if rear == n-1, the next will be rear = 0).

To delete an element from the queue, we need first to check that the queue isn't empty, then we ~~de~~ in crement front and return item[front] (if front == n-1, the next will be front = 0). Note that the cell that front is pointing to will not be used for storing data (like the header node in linked lists).





```
struct queue{
        int item[SIZE];
        int front;
        int rear;
} q;

q.front = SIZE-1;
q.rear = SIZE-1;

int isEmpty (queue q){
        return (q.front == q.rear);
}

int delete (queue q){
        if (isEmpty(q))
              printf("Queue is empty!\n");
        else{
              if (q.front == n-1)
                      q.front = 0;
              else
                      q.front++;
        }
        return q.item[q.front];
}
```



```
void insert (queue q, int x){

        int temp = q.rear;
                              n-1
        if (q.rear == ~~q.front~~)
                q.rear=0;
        else
                q.rear++;

        if (q.rear == q.front){
                printf("Out of Memory\n");
                q.rear = temp;
        }
        else
                q.item [q.rear] = x;

}
```