**Questions:-** Kth node from last, given list cyclic or null-terminated

**Operations**:-

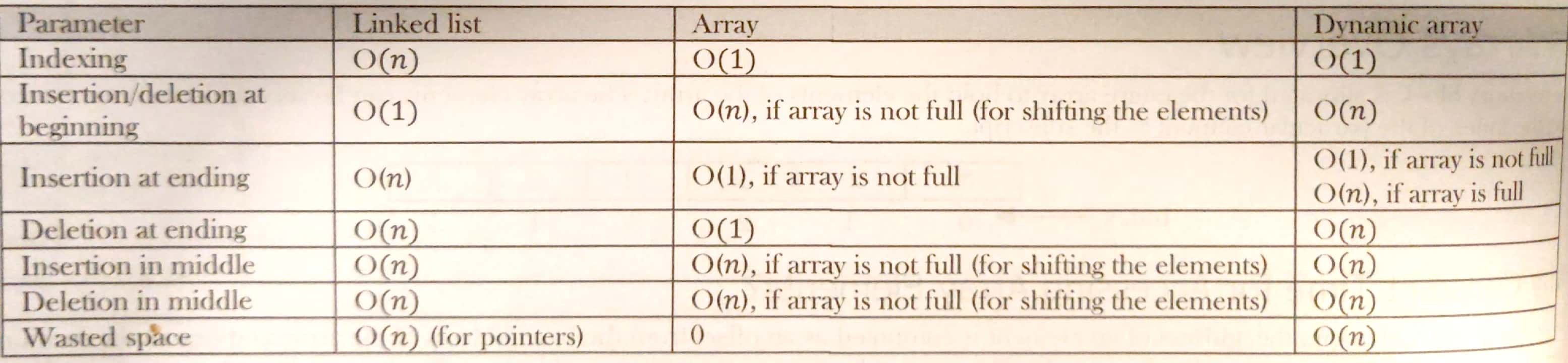
* Insert : inserts an element into the list
* Delete: removes and returns the specified position element from the list
* Delete List: Removes all elements of the list
* Count: Returns the number of elements in the list
* Find nth node from the end of the list

**Advantages**:-

* Expanded in constant time

**Disadvantages**:-

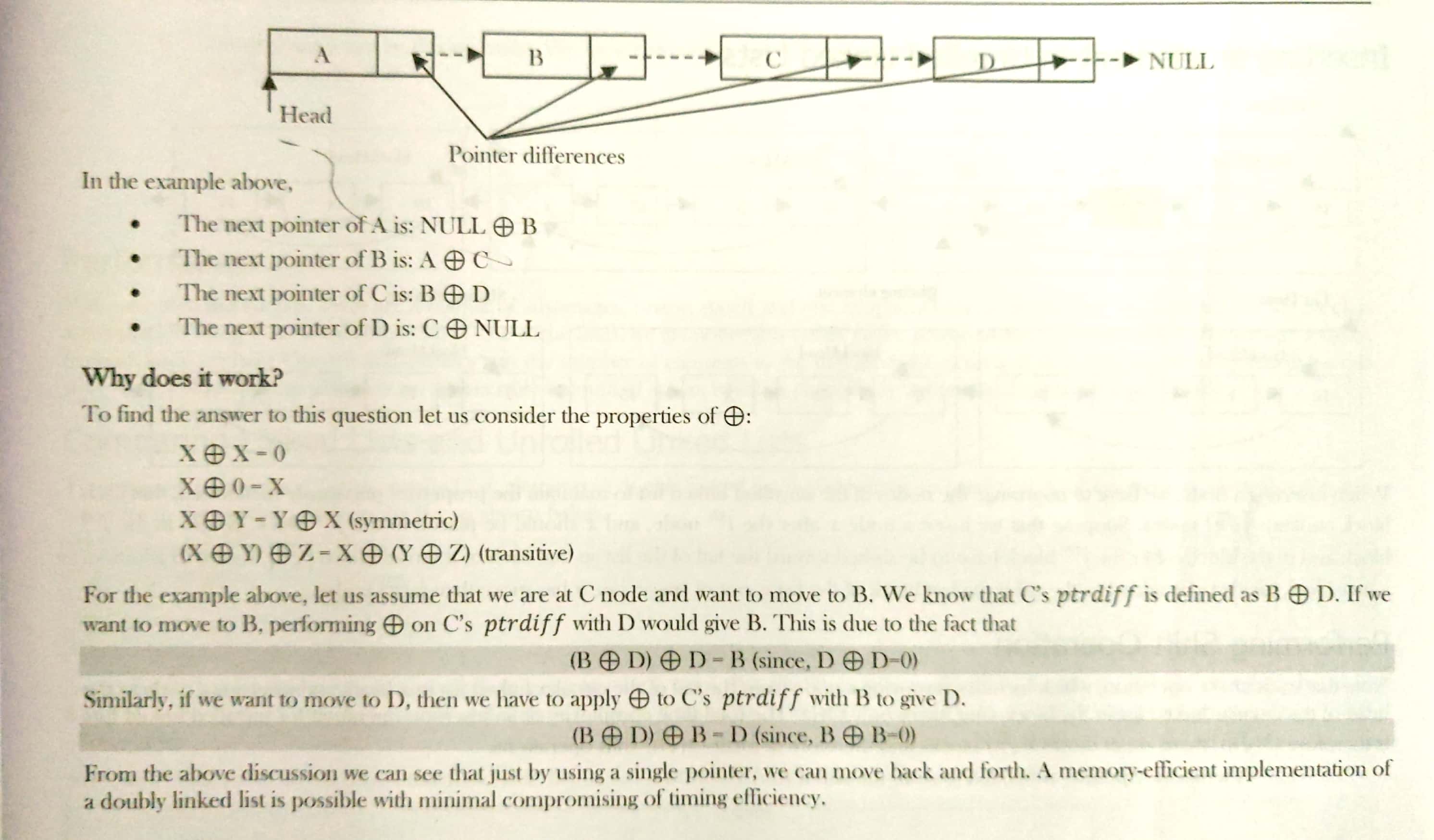
* Linked lists take O(n) for access to an element in the list in the worst case
* Linked lists waste memory in terms of extra reference points.



**Singly Linked List**

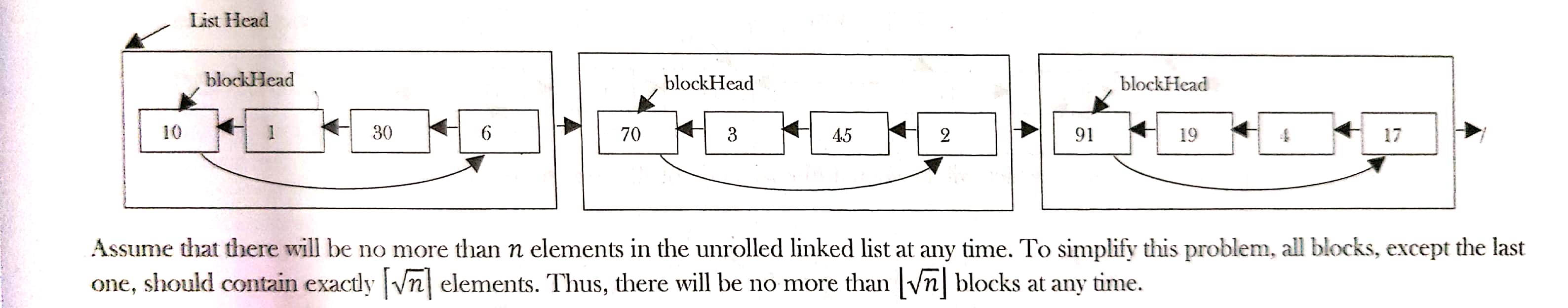
The first part of the record is a field that stores the data, and the second part of the record is a field that stores a pointer to a node.

Each node is allocated in the heap with a call to malloc(), so the node memory continues to exist until it is explicitly deallocated with a call to free().

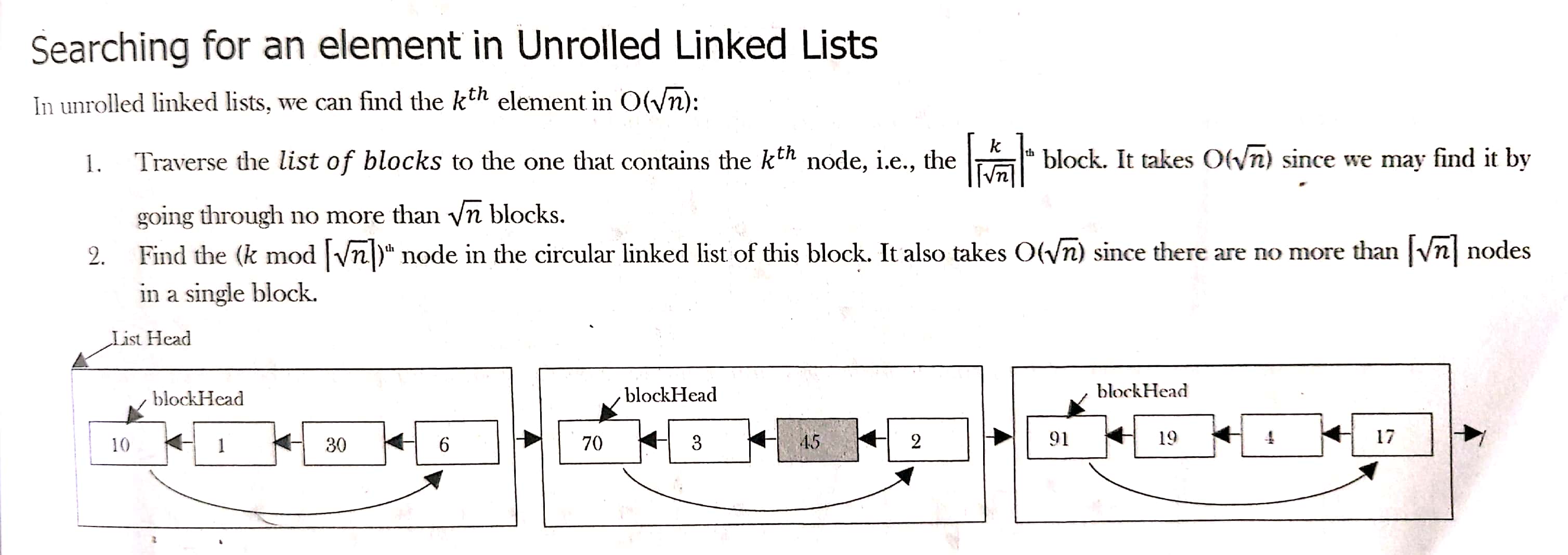


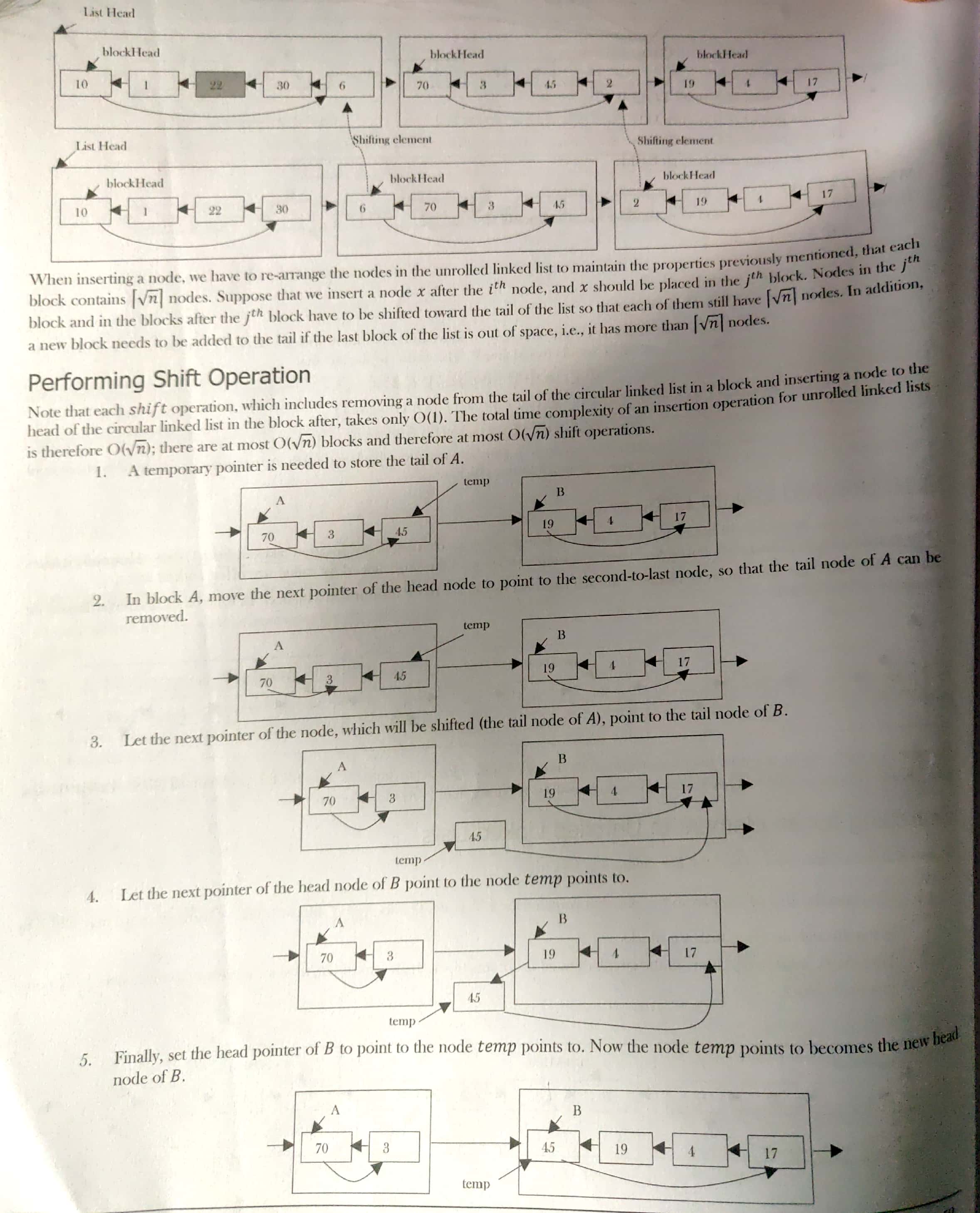
**Unrolled Linked List**

An unrolled linked list stores multiple element in each node (let us call it a block for our convenience). In each block, a circular linked list is used to connect all nodes.



**Searching for an element in Unrolled Linked List**





6. temp pointer can be thrown away. We have completed the shift operation to move the original tail node of A to become the new head node of B.

**Performance**

With unrolled linked lists, there are a couple of advantages, one in speed and one in space. First, if the number of elements in each block is appropriately sized (at most the size of one cache line), we get noticeably better cache performance from the improved memory locality. Second, since we have O(n/m) links, where n is number of elements in the unrolled linked list and m is the number of elements we can store in any block, we can also save an appreciable amount of space, which is particularly noticeable if each element is small.

**Kth node from last**:

*Approach-1*

Count the numbers of nodes. If the number of node is < k-1 then return saying “fewer number of nodes in list”. If the number of nodes > k-1 then go to next node. Continue this until the number of nodes after current node are k-1.

Time Complexity: O(n2), for scanning the remaining list (from current node) for each node

Space Complexity: O(1)

*Approach - 2*

In this approach, create a **hash table** whose entries are <position of node, node address>. That means, key is the position of the node in the list and value is the address of that node.

By the time we traverse the complete list (for creating the hash table), we can find the list length. Let us say the list length is M. To find nth from the end of list, convert it to M-n+1th from beginning. Since we already know the length of the list, it is just a matter of returning M-n+1th key value from the hash table.

Time Complexity: O(m), time for creating the hash table

Space Complexity: O(m), hash table of size m

*Appproach-3*

Find the length without creating hash table. Compute n-k+1 and with one more scan get the n-k+1 node from the beginning.

Time Complexity: O(n)+O(n)=O(n), time for finding the length+ time for finding the n-k+1 node from the beginning.

Space Complexity: O(1)

*Efficient Approach-4*

Use two pointes kthNode and pTemp. Initially, both point to head node. kthNode starts moving only after pTemp has made k-1 moves. From there both move forward until pTemp reaches the end of the list. As a result kthNode points to kth from the end of the linked list.

Time Complexity: O(n)

Space Complexity: O(1)

**Check whether the given list is null-terminated or ends in cycle (cyclic).**

*Approach – 1*

**Hash table**.

Traverse the linked list nodes one by one. Check if the address of the node is available in the hash table or not. If it is already available in the hash table, that indicates that we are visiting the node that was already visited. This is only possible if given linked list has a loop in it. If the address of the node is not available in hash table, insert that node’s address into the hash table. Continue the process until we reach the end of the linked list or find the loop.

Time Complexity: O(n) for scanning the linked list. Only do the scan of input

Space Complexity: O(n) for hash table

**Note –** Sorting technique will not work with this question. Length is needed for sorting technique, if list has loop then user may end up in an infinite loop.

*Efficient Approach – 2*

**Floyd Cycle finding algorithm**

The space complexity can be reduced to O(1) by considering two pointers at different speed – a slow pointer and a fast pointer. The slow pointer moves one step at a time while the fast pointer moves two steps at time. It uses two pointers moving at different speeds to walk the linked list. If there is no cycle in list, the fast pointer will eventually reach the end and we can return false in this case. When there is a loop, once they enter it, they are expected to meet, which denotes that there is a loop.

This works because the only way a faster moving pointer would point to the same location as a slower moving pointer is if somehow the entire list or a part of it is circular.

Time Complexity: O(n)

Space Complexity: O(1)

**Check whether the given list is null-terminated or ends in cycle (cyclic) and find the start node of loop if exist**

After finding the loop using Floyd Cycle Algorithm, initialize the slow to the head of the linked list. From that point onwards both slow and fast move only one node at a time. The point at which they meet is the start of a loop. Generally, this method is used to remove the loop.

Reason – This problem is at heart of the number theory. In the Floyd algorithm, slow and fast will meet when they are n X L, where L is loop length. Furthermore, the slow is at the midpoint between the fast and the beginning of the sequence because of the way they move. Therefore, slow is n X L away from the beginning of the sequence as well. If we move both one step at time, from the position of the slow and from the start of the sequence, they will move as soon as both are in the loop, since they are n X L, a multiple of the loop length, apart. One of them is already in the loop, so just move the other one in one step until, it enters the loop, keeping the other n X L away from it at all times.

Time Complexity: O(n)

Space Complexity: O(1)

**Check whether the given list is null-terminated or ends in cycle (cyclic) and find the length of loop if exist**

*Approach-1*

After detecting the loop in linked list, keep the slow as it is. The fast keeps on moving until it again comes back to slow. While moving fast, use a counter variable which increments at the rate of 1.

Time Complexity: O(n)

Space Complexity: O(1)

**Insert a node in sorted linked list**

Traverse the list and find a position for the element and insert it.

**Reverse a linked list**

*Iterative version*:

While traversing the list, change the current node’s next pointer to its previous element. Since a node doesn’t have reference to its previous node, we must store its previous element beforehand. We also need another pointer to store the next node before changing the reference. Use three pointers “past”, “present” and “future” to keep track of previous, current and next node during linked list reversal.

Time Complexity: O(n)

Space Complexity: O(1)

*Recursion version*:

Divide the list in two parts – first node and rest of the linked list.

Time Complexity: O(n)

Space Complexity: O(n), the extra space comes from implicit stack space due to recursion. The recursion could go up to n level.

**Two linked list with m nodes and n nodes before it reaches at some point and become a single linked list (m and n maybe m=n, m<n or m>n)**

*Approach – 1*

Compare every node pointer in the first list with every other node pointer in the second list by which the matching node pointers will lead to the intersecting node.

Time Complexity: O(mn)

Space Complexity: O(1)

**Note –** Sorting algorithm will not work. In the algorithm, we are sorting all the node pointers of both the lists and sorting. But we are forgetting the fact that there can be many repeated elements. This is because after the merging point, all node pointers are the same for both the lists. The algorithm works fine only in one case and it is when both lists have the ending node at their merging point.

*Approach – 2*

**Hash Table**

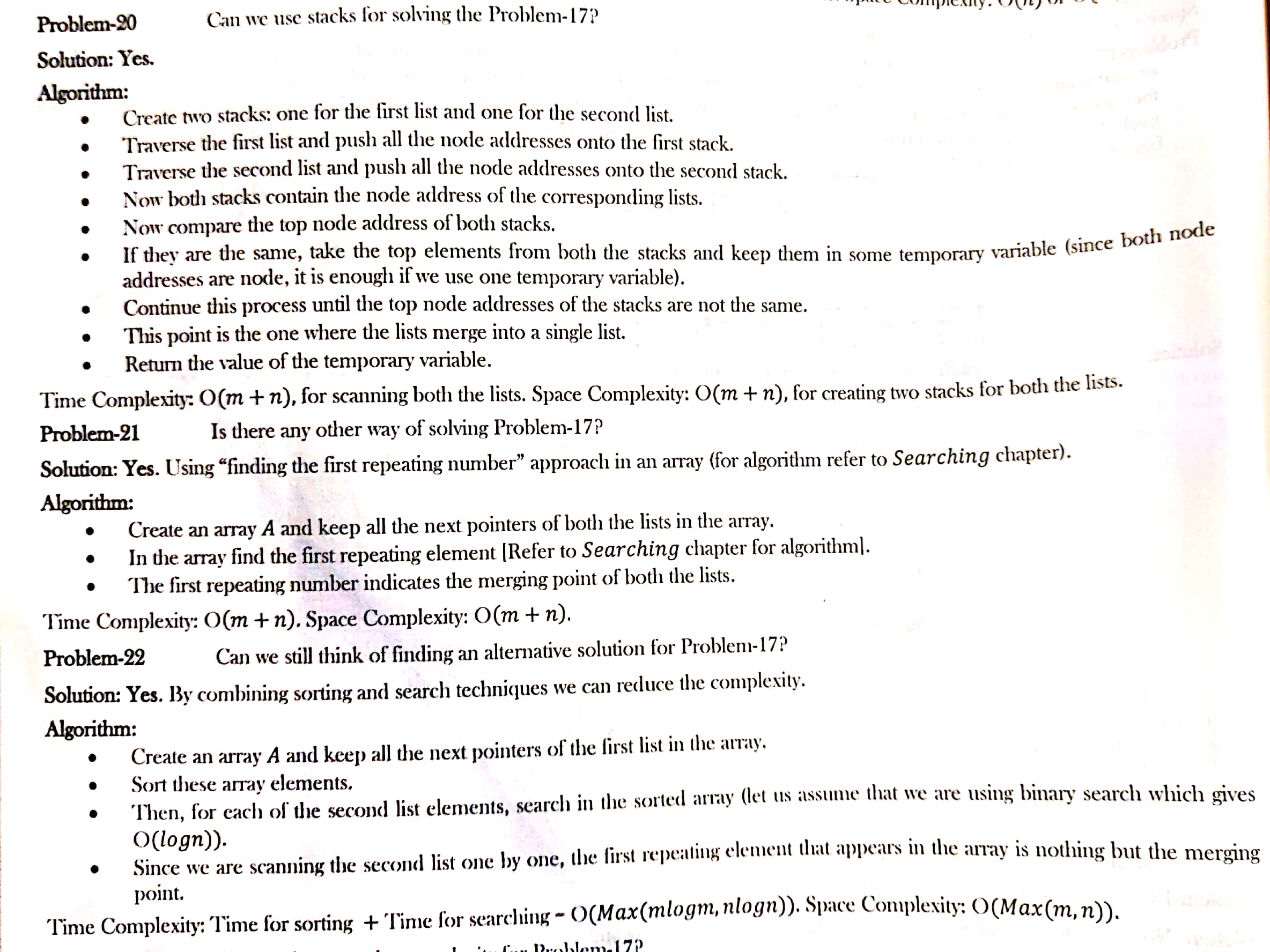
Select a list which has less number of nodes or select one randomly if lengths are unknown. Traverse the other list and for each node pointer of this list check whether the same node pointer exists in the hash table. If there is a merge point for the given lists then we will definitely encounter the node pointer in the hash table.

Time Complexity: O(m)+O(n) = Time for creating hash table + Time for scanning the second list

Space Complexity: O(m) or O(n)

*Approach – 3, 4 and 5*

**Stacks**



*Efficient Approach-6*

Maintain two pointers head1 and head2 initialized at the head of list1 and list2, respectively. Then let them both traverse through the lists, one node at a time. First calculate the length of two lists and find the difference. Then start from the longer list at the diff offset, iterate though 2 lists and find the node.

* Find lengths (L1 and L2) of both lists – O(n) + O(m) = O(max(m, n))
* Take the difference d of the length – O(1)
* Make d steps in longer list – O(d)
* Step in both lists in parallel until links to next node match – O(min(m, n))

Time Complexity: O(max(m, n))

Space Complexity: O(1)

**Middle of the linked list**

*Approach – 1*

For each of the node, count how many nodes are there in the list, and see whether it is middle node of the list.

Time Complexity: O(n2)

Space Complexity: O(1)

*Approach – 2*

Traverse the list and find the length of the list. After finding the length, again scan the list and locate n/2 node from the beginning.

Time Complexity: O(n) = Time for finding the length of the list + Time for locating middle node

Space Complexity: O(1)

*Approach – 3*

**Hash table**