**Operations**:-

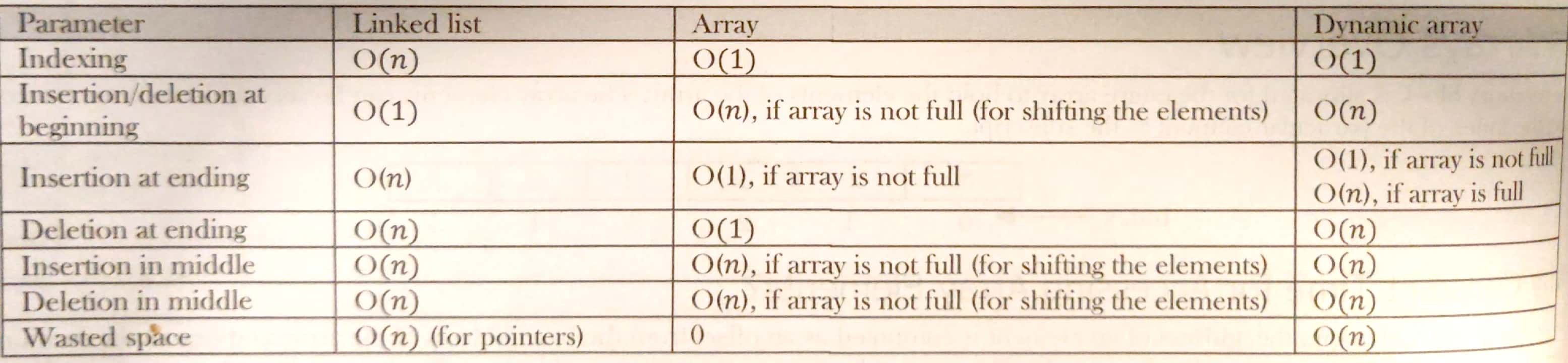
* Insert : inserts an element into the list
* Delete: removes and returns the specified position element from the list
* Delete List: Removes all elements of the list
* Count: Returns the number of elements in the list
* Find nth node from the end of the list

**Advantages**:-

* Expanded in constant time

**Disadvantages**:-

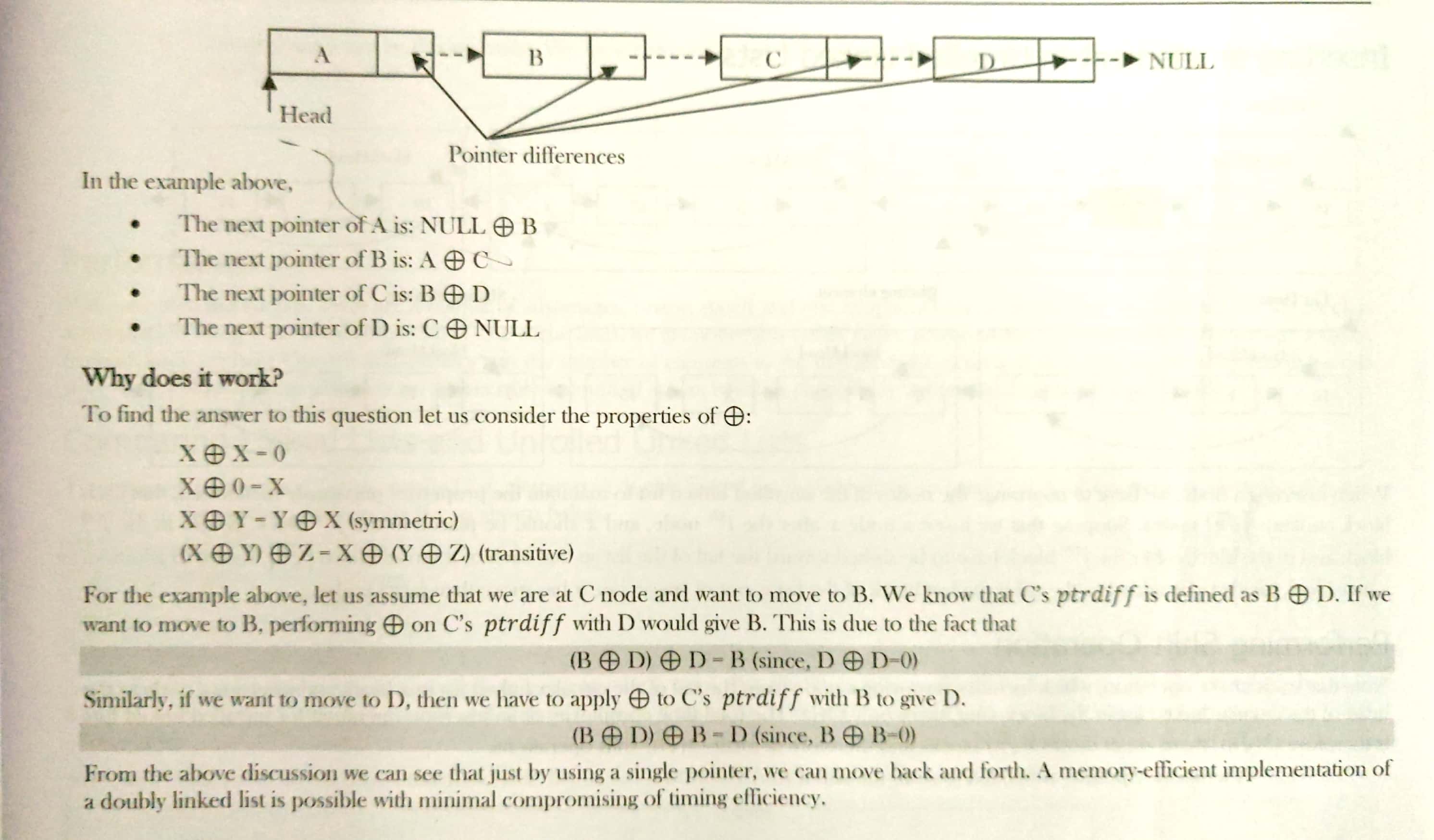
* Linked lists take O(n) for access to an element in the list in the worst case
* Linked lists waste memory in terms of extra reference points.



**Singly Linked List**

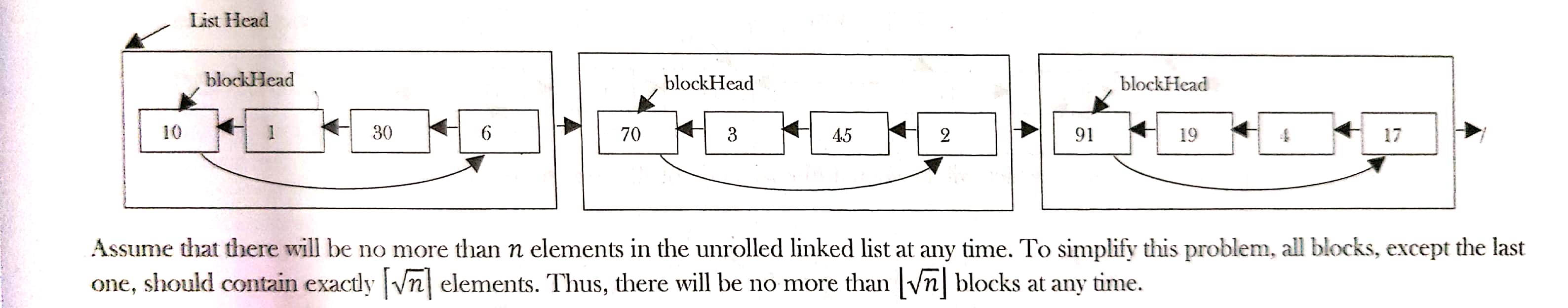
The first part of the record is a field that stores the data, and the second part of the record is a field that stores a pointer to a node.

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().

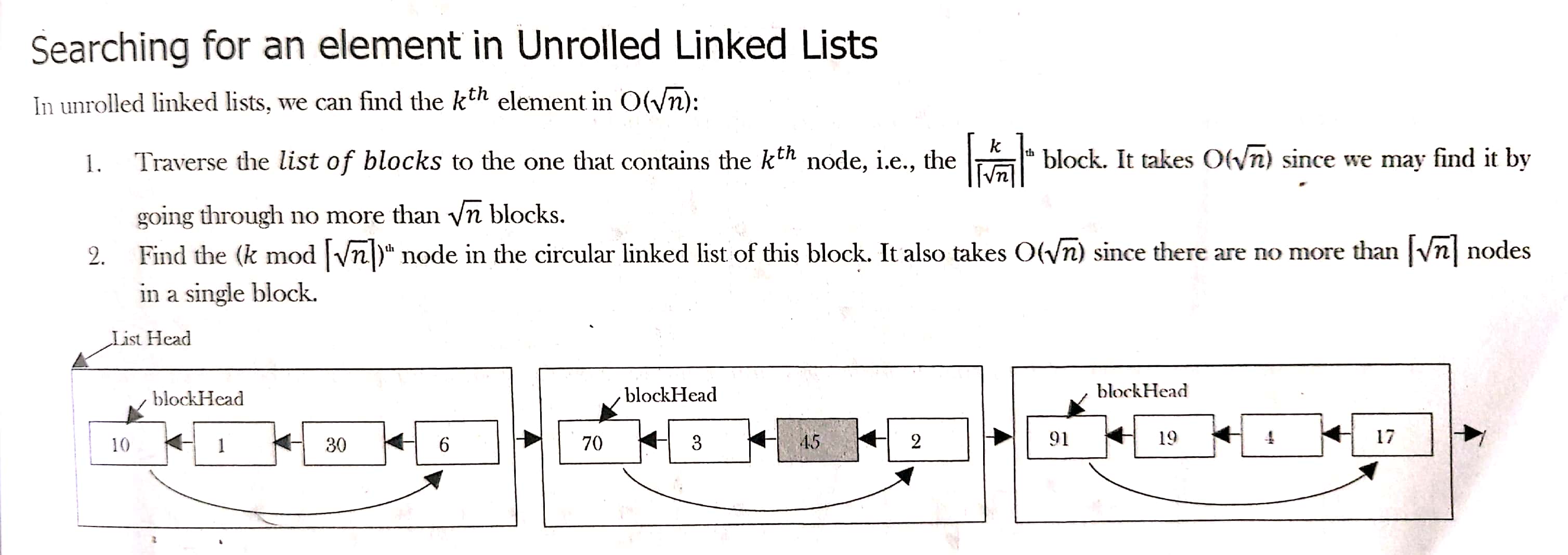


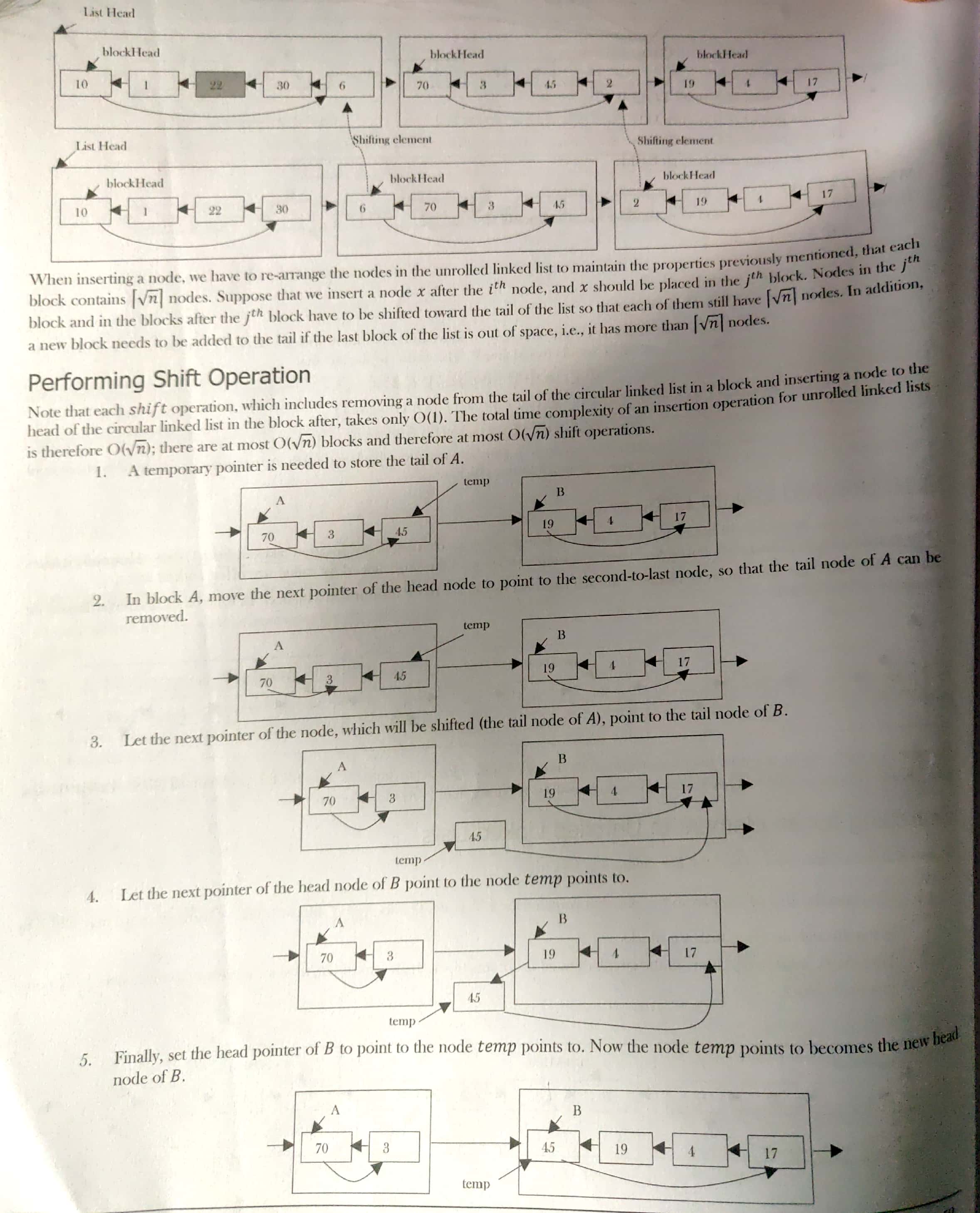
**Unrolled Linked List**

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)