CHAPTER

7 Linked Lists

The S-expressions are formed according to the following recursive rules.

1. *The atomic symbols p_1, p_2, etc., are S-expressions.*
2. *A null expression Λ is also admitted.*
3. *If e is an S-expression so is (e).*
4. *If e_1 and e_2 are S-expressions so is (e_1, e_2).*

— "*Recursive Functions Of Symbolic Expressions*,"
J. McCarthy, 1959

A list implements an ordered collection of values, which may include repetitions. Sepcifically, a *singly linked list* is a data structure that contains a sequence of nodes such that each node contains an object and a reference to the next node in the list. The first node is referred to as the *head* and the last node is referred to as the *tail*; the tail's next field is null. The structure of a singly linked list is given in Figure 7.1. There are many variants of linked lists, e.g., in a *doubly linked list*, each node has a link to its predecessor; similarly, a sentinel node or a self-loop can be used instead of null to mark the end of the list. The structure of a doubly linked list is given in Figure 7.2.

A list is similar to an array in that it contains objects in a linear order. The key differences are that inserting and deleting elements in a list has time complexity $O(1)$. On the other hand, obtaining the kth element in a list is expensive, having $O(n)$ time complexity. Lists are usually building blocks of more complex data structures. However, as we will see in this chapter, they can be the subject of tricky problems in their own right.

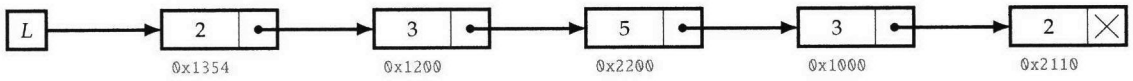


Figure 7.1: Example of a singly linked list. The number in hex below a node indicates the memory address of that node.

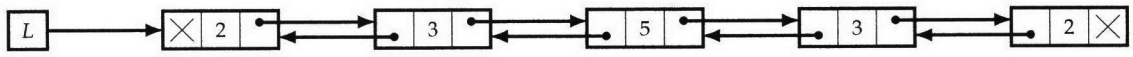


Figure 7.2: Example of a doubly linked list.

For all problems in this chapter, unless otherwise stated, each node has two entries—a data field, and a next field, which points to the next node in the list, with the next field of the last node being null. Its prototype is as follows:

```
class ListNode:
    def __init__(self, data=0, next_node=None):
        self.data = data
        self.next = next_node
```

Linked lists boot camp

There are two types of list-related problems—those where you have to implement your own list, and those where you have to exploit the standard list library. We will review both these aspects here, starting with implementation, then moving on to list libraries.

Implementing a basic list API—search, insert, delete—for singly linked lists is an excellent way to become comfortable with lists.

Search for a key:

```
def search_list(L, key):
    while L and L.data != key:
        L = L.next
    # If key was not present in the list, L will have become null.
    return L
```

Insert a new node after a specified node:

```
# Insert new_node after node.
def insert_after(node, new_node):
    new_node.next = node.next
    node.next = new_node
```

Delete a node:

```
# Delete the node past this one. Assume node is not a tail.
def delete_after(node):
    node.next = node.next.next
```

Insert and delete are local operations and have $O(1)$ time complexity. Search requires traversing the entire list, e.g., if the key is at the last node or is absent, so its time complexity is $O(n)$, where n is the number of nodes.

List problems often have a simple brute-force solution that uses $O(n)$ space, but a subtler solution that uses the **existing list nodes** to reduce space complexity to $O(1)$.
Very often, a problem on lists is conceptually simple, and is more about **cleanly coding what's specified**, rather than designing an algorithm.
Consider using a **dummy head** (sometimes referred to as a sentinel) to avoid having to check for empty lists. This simplifies code, and makes bugs less likely.
It's easy to forget to **update next** (and previous for double linked list) for the head and tail.
Algorithms operating on singly linked lists often benefit from using **two iterators**, one ahead of the other, or one advancing quicker than the other.

Table 7.1: Top Tips for Linked Lists

Know your linked list libraries

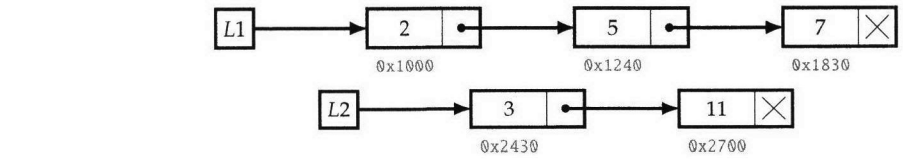
We now review the standard linked list library, with the reminder that many interview problems that are directly concerned with lists require you to write your own list class.

Under the hood, the Python `list` type is typically implemented as a dynamically resized array, and the key methods on it are described on Page 38. This chapter is concerned specifically with

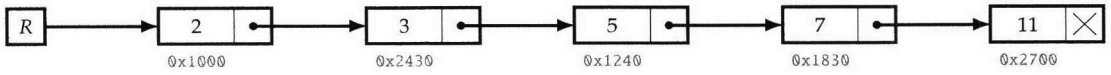
linked lists, which are not a standard type in Python. We define our own singly and doubly linked list types. Some of the key methods on these lists include returning the head/tail, adding an element at the head/tail, returning the value stored at the head/tail, and deleting the head, tail, or arbitrary node in the list.

7.1 Merge two sorted lists

Consider two singly linked lists in which each node holds a number. Assume the lists are sorted, i.e., numbers in the lists appear in ascending order within each list. The *merge* of the two lists is a list consisting of the nodes of the two lists in which numbers appear in ascending order. Merge is illustrated in Figure 7.3.



(a) Two sorted lists.



(b) The merge of the lists in (a).

Figure 7.3: Merging sorted lists.

Write a program that takes two lists, assumed to be sorted, and returns their merge. The only field your program can change in a node is its next field.

Hint: Two sorted arrays can be merged using two indices. For lists, take care when one iterator reaches the end.

Solution: A naive approach is to append the two lists together and sort the resulting list. The drawback of this approach is that it does not use the fact that the initial lists are sorted. The time complexity is that of sorting, which is $O((n+m)\log(n+m))$, where n and m are the lengths of each of the two input lists.

A better approach, in terms of time complexity, is to traverse the two lists, always choosing the node containing the smaller key to continue traversing from.

```
def merge_two_sorted_lists(L1, L2):
    # Creates a placeholder for the result.
    dummy_head = tail = ListNode()

    while L1 and L2:
        if L1.data < L2.data:
            tail.next, L1 = L1, L1.next
        else:
            tail.next, L2 = L2, L2.next
        tail = tail.next

    # Appends the remaining nodes of L1 or L2
    tail.next = L1 or L2
    return dummy_head.next
```

The worst-case, from a runtime perspective, corresponds to the case when the lists are of comparable length, so the time complexity is $O(n + m)$. (In the best-case, one list is much shorter than the other and all its entries appear at the beginning of the merged list.) Since we reuse the existing nodes, the space complexity is $O(1)$.

Variant: Solve the same problem when the lists are doubly linked.

7.2 Reverse a single sublist

This problem is concerned with reversing a sublist within a list. See Figure 7.4 for an example of sublist reversal.

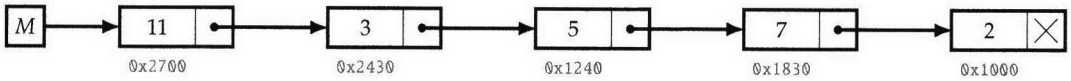


Figure 7.4: The result of reversing the sublist consisting of the second to the fourth nodes, inclusive, in the list in Figure 7.5 on the following page.

Write a program which takes a singly linked list L and two integers s and f as arguments, and reverses the order of the nodes from the sth node to fth node, inclusive. The numbering begins at 1, i.e., the head node is the first node. Do not allocate additional nodes.

Hint: Focus on the successor fields which have to be updated.

Solution: The direct approach is to extract the sublist, reverse it, and splice it back in. The drawback for this approach is that it requires two passes over the sublist.

The update can be performed with a single pass by combining the identification of the sublist with its reversal. We identify the start of sublist by using an iteration to get the sth node and its predecessor. Once we reach the sth node, we start the reversing process and keep counting. When we reach the fth node, we stop the reversion process and link the reverted section with the unreverted sections.

```
def reverse_sublist(L, start, finish):
    dummy_head = sublist_head = ListNode(0, L)
    for _ in range(1, start):
        sublist_head = sublist_head.next

    # Reverses sublist.
    sublist_iter = sublist_head.next
    for _ in range(finish - start):
        temp = sublist_iter.next
        sublist_iter.next, temp.next, sublist_head.next = (temp.next,
                                                           sublist_head.next,
                                                           temp)
    return dummy_head.next
```

The time complexity is dominated by the search for the fth node, i.e., $O(f)$.

Variant: Write a function that reverses a singly linked list. The function should use no more than constant storage beyond that needed for the list itself. The desired transformation is illustrated in Figure 7.5 on the next page.

Variant: Write a program which takes as input a singly linked list L and a nonnegative integer k, and reverses the list k nodes at a time. If the number of nodes n in the list is not a multiple of k, leave the last n mod k nodes unchanged. Do not change the data stored within a node.

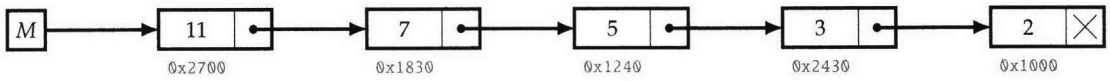


Figure 7.5: The reversed list for the list in Figure 7.3(b) on Page 84. Note that no new nodes have been allocated.

7.3 Test for cyclicity

Although a linked list is supposed to be a sequence of nodes ending in null, it is possible to create a cycle in a linked list by making the next field of an element reference to one of the earlier nodes.

Write a program that takes the head of a singly linked list and returns null if there does not exist a cycle, and the node at the start of the cycle, if a cycle is present. (You do not know the length of the list in advance.)

Hint: Consider using two iterators, one fast and one slow.

Solution: This problem has several solutions. If space is not an issue, the simplest approach is to explore nodes via the next field starting from the head and storing visited nodes in a hash table—a cycle exists if and only if we visit a node already in the hash table. If no cycle exists, the search ends at the tail (often represented by having the next field set to null). This solution requires $O(n)$ space, where n is the number of nodes in the list.

A brute-force approach that does not use additional storage and does not modify the list is to traverse the list in two loops—the outer loop traverses the nodes one-by-one, and the inner loop starts from the head, and traverses as many nodes as the outer loop has gone through so far. If the node being visited by the outer loop is visited twice, a loop has been detected. (If the outer loop encounters the end of the list, no cycle exists.) This approach has $O(n^2)$ time complexity.

This idea can be made to work in linear time—use a slow iterator and a fast iterator to traverse the list. In each iteration, advance the slow iterator by one and the fast iterator by two. The list has a cycle if and only if the two iterators meet. The reasoning is as follows: if the fast iterator jumps over the slow iterator, the slow iterator will equal the fast iterator in the next step.

Now, assuming that we have detected a cycle using the above method, we can find the start of the cycle, by first calculating the cycle length C. Once we know there is a cycle, and we have a node on it, it is trivial to compute the cycle length. To find the first node on the cycle, we use two iterators, one of which is C ahead of the other. We advance them in tandem, and when they meet, that node must be the first node on the cycle.

The code to do this traversal is quite simple:

```
def has_cycle(head):
    def cycle_len(end):
        start, step = end, 0
        while True:
            step += 1
            start = start.next
            if start is end:
                return step
```

```
    fast = slow = head
    while fast and fast.next and fast.next.next:
        slow, fast = slow.next, fast.next.next
        if slow is fast:
            # Finds the start of the cycle.
            cycle_len_advanced_iter = head
            for _ in range(cycle_len(slow)):
                cycle_len_advanced_iter = cycle_len_advanced_iter.next

            it = head
            # Both iterators advance in tandem.
            while it is not cycle_len_advanced_iter:
                it = it.next
                cycle_len_advanced_iter = cycle_len_advanced_iter.next
            return it  # iter is the start of cycle.
    return None  # No cycle.
```

Let F be the number of nodes to the start of the cycle, C the number of nodes on the cycle, and n the total number of nodes. Then the time complexity is $O(F) + O(C) = O(n)$—$O(F)$ for both pointers to reach the cycle, and $O(C)$ for them to overlap once the slower one enters the cycle.

Variant: The following program purports to compute the beginning of the cycle without determining the length of the cycle; it has the benefit of being more succinct than the code listed above. Is the program correct?

```
def has_cycle(head):
    fast = slow = head
    while fast and fast.next and fast.next.next:
        slow, fast = slow.next, fast.next.next
        if slow is fast:  # There is a cycle.
            # Tries to find the start of the cycle.
            slow = head
            # Both pointers advance at the same time.
            while slow is not fast:
                slow, fast = slow.next, fast.next
            return slow  # slow is the start of cycle.
    return None  # No cycle.
```

7.4 Test for overlapping lists—lists are cycle-free

Given two singly linked lists there may be list nodes that are common to both. (This may not be a bug—it may be desirable from the perspective of reducing memory footprint, as in the flyweight pattern, or maintaining a canonical form.) For example, the lists in Figure 7.6 on the following page overlap at Node I.

Write a program that takes two cycle-free singly linked lists, and determines if there exists a node that is common to both lists.

Hint: Solve the simple cases first.

Solution: A brute-force approach is to store one list's nodes in a hash table, and then iterate through the nodes of the other, testing each for presence in the hash table. This takes $O(n)$ time and $O(n)$ space, where n is the total number of nodes.

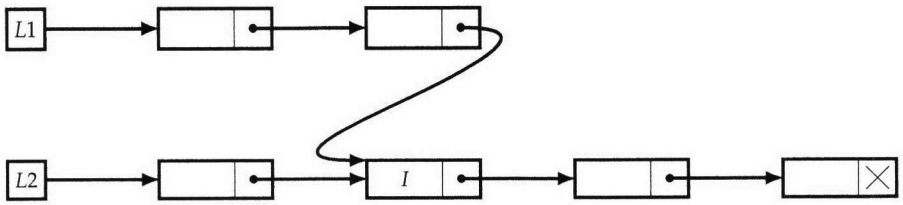


Figure 7.6: Example of overlapping lists.

We can avoid the extra space by using two nested loops, one iterating through the first list, and the other to search the second for the node being processed in the first list. However, the time complexity is $O(n^2)$.

The lists overlap if and only if both have the same tail node: once the lists converge at a node, they cannot diverge at a later node. Therefore, checking for overlap amounts to finding the tail nodes for each list.

To find the first overlapping node, we first compute the length of each list. The first overlapping node is determined by advancing through the longer list by the difference in lengths, and then advancing through both lists in tandem, stopping at the first common node. If we reach the end of a list without finding a common node, the lists do not overlap.

```
def overlapping_no_cycle_lists(L1, L2):
    def length(L):
        length = 0
        while L:
            length += 1
            L = L.next
        return length

    L1_len, L2_len = length(L1), length(L2)
    if L1_len > L2_len:
        L1, L2 = L2, L1  # L2 is the longer list
    # Advances the longer list to get equal length lists.
    for _ in range(abs(L1_len - L2_len)):
        L2 = L2.next

    while L1 and L2 and L1 is not L2:
        L1, L2 = L1.next, L2.next
    return L1  # None implies there is no overlap between L1 and L2.
```

The time complexity is $O(n)$ and the space complexity is $O(1)$.

7.5 Test for overlapping lists—lists may have cycles

Solve Problem 7.4 on the previous page for the case where the lists may each or both have a cycle. If such a node exists, return a node that appears first when traversing the lists. This node may not be unique—if one node ends in a cycle, the first cycle node encountered when traversing it may be different from the first cycle node encountered when traversing the second list, even though the cycle is the same. In such cases, you may return either of the two nodes.

For example, Figure 7.7 on the facing page shows an example of lists which overlap and have cycles. For this example, both A and B are acceptable answers.

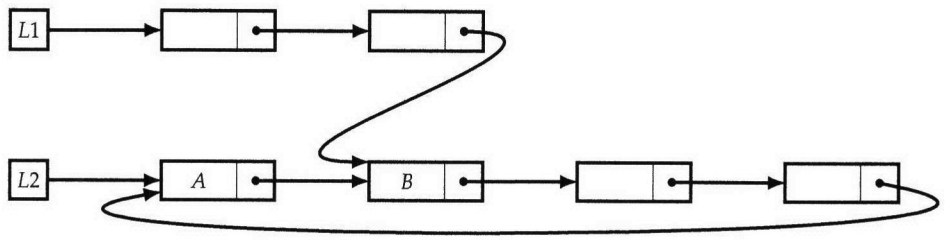


Figure 7.7: Overlapping lists.

Hint: Use case analysis. What if both lists have cycles? What if they end in a common cycle? What if one list has cycle and the other does not?

Solution: This problem is easy to solve using $O(n)$ time and space complexity, where n is the total number of nodes, using the hash table approach in Solution 7.4 on Page 87.

We can improve space complexity by studying different cases. The easiest case is when neither list is cyclic, which we can determine using Solution 7.3 on Page 86. In this case, we can check overlap using the technique in Solution 7.4 on Page 87.

If one list is cyclic, and the other is not, they cannot overlap, so we are done.

This leaves us with the case that both lists are cyclic. In this case, if they overlap, the cycles must be identical.

There are two subcases: the paths to the cycle merge before the cycle, in which case there is a unique first node that is common, or the paths reach the cycle at different nodes on the cycle. For the first case, we can use the approach of Solution 7.4 on Page 87. For the second case, we use the technique in Solution 7.3 on Page 86.

```
def overlapping_lists(L1, L2):
    # Store the start of cycle if any.
    root1, root2 = has_cycle(L1), has_cycle(L2)

    if not root1 and not root2:
        # Both lists don't have cycles.
        return overlapping_no_cycle_lists(L1, L2)
    elif (root1 and not root2) or (not root1 and root2):
        # One list has cycle, one list has no cycle.
        return None
    # Both lists have cycles.
    temp = root2
    while True:
        temp = temp.next
        if temp is root1 or temp is root2:
            break

    # L1 and L2 do not end in the same cycle.
    if temp is not root1:
        return None  # Cycles are disjoint.

    # Calculates the distance between a and b.
    def distance(a, b):
        dis = 0
        while a is not b:
            a = a.next
```

```
            dis += 1
        return dis

    # L1 and L2 end in the same cycle, locate the overlapping node if they
    # first overlap before cycle starts.
    stem1_length, stem2_length = distance(L1, root1), distance(L2, root2)
    if stem1_length > stem2_length:
        L2, L1 = L1, L2
        root1, root2 = root2, root1
    for _ in range(abs(stem1_length - stem2_length)):
        L2 = L2.next
    while L1 is not L2 and L1 is not root1 and L2 is not root2:
        L1, L2 = L1.next, L2.next

    # If L1 == L2 before reaching root1, it means the overlap first occurs
    # before the cycle starts; otherwise, the first overlapping node is not
    # unique, we can return any node on the cycle.
    return L1 if L1 is L2 else root1
```

The algorithm has time complexity $O(n + m)$, where n and m are the lengths of the input lists, and space complexity $O(1)$.

7.6 Delete a node from a singly linked list

Given a node in a singly linked list, deleting it in $O(1)$ time appears impossible because its predecessor's next field has to be updated. Surprisingly, it can be done with one small caveat—the node to delete cannot be the last one in the list and it is easy to copy the value part of a node.

Write a program which deletes a node in a singly linked list. The input node is guaranteed not to be the tail node.

Hint: Instead of deleting the node, can you delete its successor and still achieve the desired configuration?

Solution: Given the pointer to a node, it is impossible to delete it from the list without modifying its predecessor's next pointer and the only way to get to the predecessor is to traverse the list from head, which requires $O(n)$ time, where n is the number of nodes in the list.

Given a node, it is easy to delete its successor, since this just requires updating the next pointer of the current node. If we copy the value part of the next node to the current node, and then delete the next node, we have effectively deleted the current node. The time complexity is $O(1)$.

```
# Assumes node_to_delete is not tail.
def deletion_from_list(node_to_delete):
    node_to_delete.data = node_to_delete.next.data
    node_to_delete.next = node_to_delete.next.next
```

7.7 Remove the kth last element from a list

Without knowing the length of a linked list, it is not trivial to delete the kth last element in a singly linked list.

Given a singly linked list and an integer k, write a program to remove the kth last element from the list. Your algorithm cannot use more than a few words of storage, regardless of the length of the list. In particular, you cannot assume that it is possible to record the length of the list.

Hint: If you know the length of the list, can you find the kth last node using two iterators?

Solution: A brute-force approach is to compute the length with one pass, and then use that to determine which node to delete in a second pass. A drawback of this approach is that it entails two passes over the data, which is slow, e.g., if traversing the list entails disc accesses.

We use two iterators to traverse the list. The first iterator is advanced by k steps, and then the two iterators advance in tandem. When the first iterator reaches the tail, the second iterator is at the $(k+1)$th last node, and we can remove the kth node.

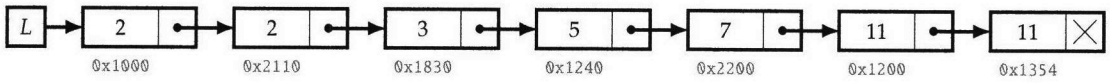
```
# Assumes L has at least k nodes, deletes the k-th last node in L.
def remove_kth_last(L, k):
    dummy_head = ListNode(0, L)
    first = dummy_head.next
    for _ in range(k):
        first = first.next

    second = dummy_head
    while first:
        first, second = first.next, second.next
    # second points to the (k + 1)-th last node, deletes its successor.
    second.next = second.next.next
    return dummy_head.next
```

The time complexity is that of list traversal, i.e., $O(n)$, where n is the length of the list. The space complexity is $O(1)$, since there are only two iterators.

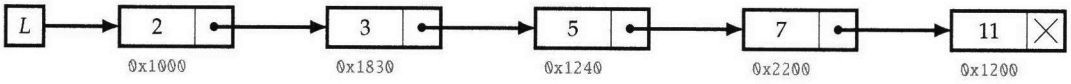
Compared to the brute-force approach, if k is small enough that we can keep the set of nodes between the two iterators in memory, but the list is too big to fit in memory, the two-iterator approach halves the number of disc accesses.

7.8 Remove duplicates from a sorted list

This problem is concerned with removing duplicates from a sorted list of integers. See Figure 7.8 for an example.



(a) List before removing duplicates.



(b) The list in (a) after removing duplicates.

Figure 7.8: Example of duplicate removal.

Write a program that takes as input a singly linked list of integers in sorted order, and removes duplicates from it. The list should be sorted.

Hint: Focus on the successor fields which have to be updated.

Solution: A brute-force algorithm is to create a new list, using a hash table to test if a value has already been added to the new list. Alternatively, we could search in the new list itself to see if

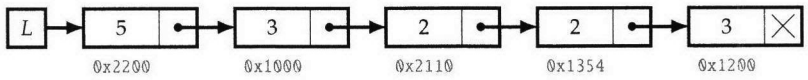


Figure 7.9: The result of applying a right cyclic shift by 3 to the list in Figure 7.1 on Page 82. Note that no new nodes have been allocated.

the candidate value already is present. If the length of the list is n, the first approach requires $O(n)$ additional space for the hash table, and the second requires $O(n^2)$ time to perform the lookups. Both allocate n nodes for the new list.

A better approach is to exploit the sorted nature of the list. As we traverse the list, we remove all successive nodes with the same value as the current node.

```
def remove_duplicates(L):
    it = L
    while it:
        # Uses next_distinct to find the next distinct value.
        next_distinct = it.next
        while next_distinct and next_distinct.data == it.data:
            next_distinct = next_distinct.next
        it.next = next_distinct
        it = next_distinct
    return L
```

Determining the time complexity requires a little amortized analysis. A single node may take more than $O(1)$ time to process if there are many successive nodes with the same value. A clearer justification for the time complexity is that each link is traversed once, so the time complexity is $O(n)$. The space complexity is $O(1)$.

Variant: Let m be a positive integer and L a sorted singly linked list of integers. For each integer k, if k appears more than m times in L, remove all nodes from L containing k.

7.9 Implement cyclic right shift for singly linked lists

This problem is concerned with performing a cyclic right shift on a list.

Write a program that takes as input a singly linked list and a nonnegative integer k, and returns the list cyclically shifted to the right by k. See Figure 7.9 for an example of a cyclic right shift.

Hint: How does this problem differ from rotating an array?

Solution: A brute-force strategy is to right shift the list by one node k times. Each right shift by a single node entails finding the tail, and its predecessor. The tail is prepended to the current head, and its original predecessor's successor is set to null to make it the new tail. The time complexity is $O(kn)$, and the space complexity is $O(1)$, where n is the number of nodes in the list.

Note that k may be larger than n. If so, it is equivalent to shift by k mod n, so we assume $k < n$. The key to improving upon the brute-force approach is to use the fact that linked lists can be cut and the sublists reassembled very efficiently. First we find the tail node t. Since the successor of the tail is the original head, we update t's successor. The original head is to become the kth node from the start of the new list. Therefore, the new head is the $(n - k)$th node in the initial list.

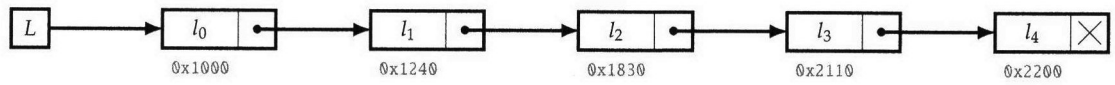
```
def cyclically_right_shift_list(L, k):
    if not L:
```

```
    return L

# Computes the length of L and the tail.
tail, n = L, 1
while tail.next:
    n += 1
    tail = tail.next

k %= n
if k == 0:
    return L

tail.next = L  # Makes a cycle by connecting the tail to the head.
steps_to_new_head, new_tail = n - k, tail
while steps_to_new_head:
    steps_to_new_head -= 1
    new_tail = new_tail.next

new_head = new_tail.next
new_tail.next = None
return new_head
```

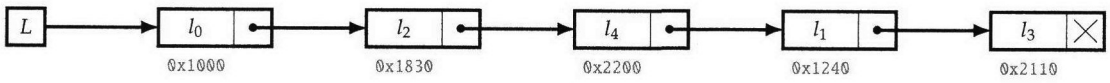
The time complexity is $O(n)$, and the space complexity is $O(1)$.

7.10 Implement even-odd merge

Consider a singly linked list whose nodes are numbered starting at 0. Define the even-odd merge of the list to be the list consisting of the even-numbered nodes followed by the odd-numbered nodes. The even-odd merge is illustrated in Figure 7.10.



(a) The initial list is L. The number in hex below a node indicates the memory address of that node.



(b) The even-odd merge of L—note that no new nodes have been allocated.

Figure 7.10: Even-odd merge example.

Write a program that computes the even-odd merge.

Hint: Use temporary additional storage.

Solution: The brute-force algorithm is to allocate new nodes and compute two new lists, one for the even and one for the odd nodes. The result is the first list concatenated with the second list. The time and space complexity are both $O(n)$.

However, we can avoid the extra space by reusing the existing list nodes. We do this by iterating through the list, and appending even elements to one list and odd elements to another list. We use an indicator variable to tell us which list to append to. Finally we append the odd list to the even list.

```
def even_odd_merge(L):
    if not L:
        return L

    even_dummy_head, odd_dummy_head = ListNode(0), ListNode(0)
    tails, turn = [even_dummy_head, odd_dummy_head], 0
    while L:
        tails[turn].next = L
        L = L.next
        tails[turn] = tails[turn].next
        turn ^= 1  # Alternate between even and odd.
    tails[1].next = None
    tails[0].next = odd_dummy_head.next
    return even_dummy_head.next
```

The time complexity is $O(n)$ and the space complexity is $O(1)$.

7.11 Test whether a singly linked list is palindromic

It is straightforward to check whether the sequence stored in an array is a palindrome. However, if this sequence is stored as a singly linked list, the problem of detecting palindromicity becomes more challenging. See Figure 7.1 on Page 82 for an example of a palindromic singly linked list.

Write a program that tests whether a singly linked list is palindromic.

Hint: It's easy if you can traverse the list forwards and backwards simultaneously.

Solution: A brute-force algorithm is to compare the first and last nodes, then the second and second-to-last nodes, etc. The time complexity is $O(n^2)$, where n is the number of nodes in the list. The space complexity is $O(1)$.

The $O(n^2)$ complexity comes from having to repeatedly traverse the list to identify the last, second-to-last, etc. Getting the first node in a singly linked list is an $O(1)$ time operation. This suggests paying a one-time cost of $O(n)$ time complexity to get the reverse of the second half of the original list, after which testing palindromicity of the original list reduces to testing if the first half and the reversed second half are equal. This approach changes the list passed in, but the reversed sublist can be reversed again to restore the original list.

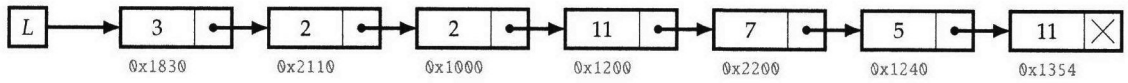
```
def is_linked_list_a_palindrome(L):
    # Finds the second half of L.
    slow = fast = L
    while fast and fast.next:
        fast, slow = fast.next.next, slow.next

    # Compares the first half and the reversed second half lists.
    first_half_iter, second_half_iter = L, reverse_linked_list(slow)
    while second_half_iter and first_half_iter:
        if second_half_iter.data != first_half_iter.data:
            return False
        second_half_iter, first_half_iter = (second_half_iter.next,
                                             first_half_iter.next)
    return True
```

The time complexity is $O(n)$. The space complexity is $O(1)$.

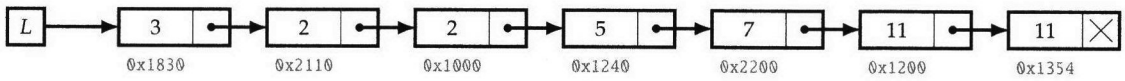
Variant: Solve the same problem when the list is doubly linked and you have pointers to the head and the tail.

7.12 Implement list pivoting

For any integer k, the pivot of a list of integers with respect to k is that list with its nodes reordered so that all nodes containing keys less than k appear before nodes containing k, and all nodes containing keys greater than k appear after the nodes containing k. See Figure 7.11 for an example of pivoting.



(a) A list to be pivoted.



(b) The result of pivoting the list in (a) with $k = 7$.

Figure 7.11: List pivoting.

Implement a function which takes as input a singly linked list and an integer k and performs a pivot of the list with respect to k. The relative ordering of nodes that appear before k, and after k, must remain unchanged; the same must hold for nodes holding keys equal to k.

Hint: Form the three regions independently.

Solution: A brute-force approach is to form three lists by iterating through the list and writing values into one of the three new lists based on whether the current value is less than, equal to, or greater than k. We then traverse the list from the head, and overwrite values in the original list from the less than, then equal to, and finally greater than lists. The time and space complexity are $O(n)$, where n is the number of nodes in the list.

A key observation is that we do not really need to create new nodes for the three lists. Instead we reorganize the original list nodes into these three lists in a single traversal of the original list. Since the traversal is in order, the individual lists preserve the ordering. We combine these three lists in the final step.

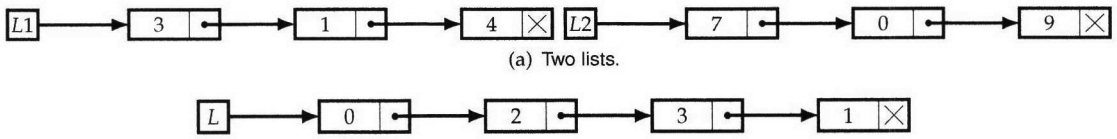
```
def list_pivoting(L, x):
    less_head = less_iter = ListNode()
    equal_head = equal_iter = ListNode()
    greater_head = greater_iter = ListNode()
    # Populates the three lists.
    while L:
        if L.data < x:
            less_iter.next = L
            less_iter = less_iter.next
        elif L.data == x:
            equal_iter.next = L
            equal_iter = equal_iter.next
        else:  # L.data > x.
            greater_iter.next = L
            greater_iter = greater_iter.next
        L = L.next
```

```
    # Combines the three lists.
    greater_iter.next = None
    equal_iter.next = greater_head.next
    less_iter.next = equal_head.next
    return less_head.next
```

The time to compute the three lists is $O(n)$. Combining the lists takes $O(1)$ time, yielding an overall $O(n)$ time complexity. The space complexity is $O(1)$.

7.13 Add list-based integers

A singly linked list whose nodes contain digits can be viewed as an integer, with the least significant digit coming first. Such a representation can be used to represent unbounded integers. This problem is concerned with adding integers represented in this fashion. See Figure 7.12 for an example.



(a) Two lists.

(b) The sum of the two lists in (a).

Figure 7.12: List-based interpretation of 413 + 907 = 1320.

Write a program which takes two singly linked lists of digits, and returns the list corresponding to the sum of the integers they represent. The least significant digit comes first.

Hint: First, solve the problem assuming no pair of corresponding digits sum to more than 9.

Solution: Note that we cannot simply convert the lists to integers, since the integer word length is fixed by the machine architecture, and the lists can be arbitrarily long.

Instead we mimic the grade-school algorithm, i.e., we compute the sum of the digits in corresponding nodes in the two lists. A key nuance of the computation is handling the carry-out. The algorithm continues processing input until both lists are exhausted and there is no remaining carry.

```
def add_two_numbers(L1, L2):
    place_iter = dummy_head = ListNode()
    carry = 0
    while L1 or L2 or carry:
        val = carry + (L1.data if L1 else 0) + (L2.data if L2 else 0)
        L1 = L1.next if L1 else None
        L2 = L2.next if L2 else None
        place_iter.next = ListNode(val % 10)
        carry, place_iter = val // 10, place_iter.next
    return dummy_head.next
```

The time complexity is $O(n + m)$ and the space complexity is $O(\max(n, m))$, where n and m are the lengths of the two lists.

Variant: Solve the same problem when integers are represented as lists of digits with the most significant digit comes first.