Chapter 2: Linked List

2.1 Introduction

When studying the array, we were quite annoyed by the limitations of these arrays:

- **Fixed capacity:** Once created, an array cannot be resized. The only way to "resize" is to create a larger new array, copy the elements from the original array into the new one, and then change the reference to the new one.
- **Shifting elements in insert:** Since we do not allow gaps in the array, inserting a new element requires that we shift all the subsequent elements right to create a hole where the new element is placed. In the worst case, we "disturb" every slot in the array when we insert it at the beginning of the array!
- **Shifting elements in removal:** Removing may also require shifting to plug the hole left by the removed element. If the ordering of the elements does not matter, we can avoid shifting by replacing the removed element with the element in the last slot (and then treating the last slot as empty).

The answer to these problems is the **Linked List** data structure. It is another primary data structure which is a sequence of nodes connected by links. The links allow the insertion of new nodes anywhere in the list or to remove of an existing node from the list without having to disturb the rest of the list (the only nodes affected are the ones adjacent to the node being inserted or removed). Since we can extend the list one node at a time, we can also resize a list until we run out of resources. However, we'll find out soon enough what we lose in the process — random access, and space! A linked list is a sequence container that supports sequential access only and requires additional space for at least n references for the links.

We maintain a linked list by referring to the first node in the list, conventionally called the **head** reference. All the other nodes can then be reached by traversing the list, starting from the head. An empty list is represented by setting the head reference to None. Given a head reference to a list, how do you count the number of nodes in the list? You have to iterate over the nodes, starting from the head, and count until you reach the end.

As mentioned earlier, a linked list is a sequence of nodes. To put anything (primitive type or object reference) in the list, we must first put this "thing" in a node, and then put the node in the list. Compare this with an array – we simply put the "thing" (our primitive type or object reference) directly into the array slot, without the need for any extra scaffolding. Now to put the “thing” in our list and to store the next item’s information, we need to design our own data type by designing a **Node** class. Our Node class is quite simple — it contains an element, and a reference to the next node in the list. Below is a code snippet of a node class given.

```
Class Node:
   Initialize data
   Initialize elem as NULL
```

This allows you to create a singly-linked list, and as a result, we can only move forward in the list by following the next links. To be able to move backward as well, we'll have to add another reference to the previous node, increasing the space usage even more. If you are wondering how we can do that, give it a thought (We will cover it later).

2.2 Operations of Linked List

To understand the operations, first, we will need to understand how to create a linked list. After creation, we can perform different operations on it like an array (e.g: iteration, insertion, removal, and so on).

2.2.1 Creation:

In order to create a linked list, we just need to create objects of the Node class and then connect one node to another using that next variable. We will only store the first Node-type object and call it head. Whenever we want to add a new node-type object, we just need to go to the last node and add the new node after that.
In the example code below, we are taking an array and converting it to a Linked List. To keep things easier, we are using the tail variable so that we do not need to go to the end of the list every time we want to add a new item.

```
Method createFromArray(arr):
    if arr is null or empty:
        return
    head = new Node(arr[0])
    current = head
    for i from 1 to arr.length - 1:
        current.next = new Node(arr[i])
        current = current.next
```

2.2.2 Iteration:

Iteration is one of the most important aspects of every data structure. We will use an iterative approach to traverse a linked list. The important thing is that to do this traversal we are using a variable (temp in this case) to store the location of node and so we are updating this temp for every iteration. Below is an example of an iteration function that takes the head of a linked list as a parameter. Time Complexity O(n) and Space Complexity O(1).

```
Method iterate():
    current = head
    while current is not null:
        print current.elem
        current = current.next
    print newline
```

2.2.3 Count:

In order to count the number of nodes present in a linked list, we just need to count how many times the iteration process is running during a traversal. Time Complexity O(n) and Space Complexity O(1).

```
Method count():
    count = 0
    current = head
    while current is not null:
        count = count + 1
        current = current.next
    return count
```

2.2.4 Retrieving the index of an element:

In the memory level, a linked list is not indexed as it does not book any memory block. However, the concept of indexing is very beneficial in order to access values, insert a value, and remove a value (we have seen it in the array part). For this reason, we will implement indexing in our linked list where the head will be a value at index 0 and the next node will be 1, and so on (just like an array). Below an example code is shown where the head of the indexOf function find out the corresponding first index of an element. Time Complexity O(n) and Space Complexity O(1).

```
Method indexOf(elem):
    index = 0
    current = head
    while current is not null:
        if current.elem == elem:
            return index
        current = current.next
        index = index + 1
    return -1
```

2.2.5 Retrieving a node from an index:

Similar to getting the element, we can also get the node of that index. Below the example, nodeAt function is given which is similar to the elemAt. Time Complexity O(n) and Space Complexity O(1).

```
Method getNode(index):
    currentIndex = 0
    current = head
    while current is not null:
        if currentIndex == index:
            return current
        current = current.next
        currentIndex = currentIndex + 1
    return null
```

2.2.6 Update value of specific index:

By tuning the elemAt function, we can also update the element of a linked list using an index. Time Complexity O(n) and Space Complexity O(1).

```
Method update(index, newValue):
    node = getNode(index)
    if node is not null:
        node.elem = newValue
        return true
    return false
```

2.2.7 Searching an element in the list:

Searching for an element in a list can be done by sequentially searching through the list. There are two typical variants: return the index of the given element (indexOf), or return true if the element exists. Time Complexity O(n) and Space Complexity O(1).

```
Method search(elem):
    current = head
    while current is not null:
        if current.elem == elem:
            return true
        current = current.next
    return false
```

2.2.8 Inserting an element into a list:

There are three places in a list where we can insert a new element: in the beginning ("head" changes), in the middle, and at the end. To insert a new node in the list, you need the reference to the predecessor to link in the new node. There is one "special" case however — inserting a new node at the beginning of the list because the head node does not have a predecessor. Inserting, in the beginning, has an important side effect — it changes the head reference! Inserting in the middle or at the end is the same — we first find the predecessor node and link in the new node with the given element. To find the specific node, we can take the help of the nodeAt function mentioned above. Below is the example code of insertion. However, this code can be made more efficient!!! Try that. Time Complexity O(n) and Space Complexity O(1).

```
Method insert(index, elem):
    newNode = new Node(elem)
    if index == 0:
        newNode.next = head
        head = newNode
        return
    prev = getNode(index - 1)
    if prev is not null:
        newNode.next = prev.next
        prev.next = newNode
```

2.2.9 Removing an element from a list:

Removing an element from the list is done by removing the node that contains the element. If we only know the element that we want to remove, then we have to sequentially search the list to find the node which contains the element (if it exists in the list that is); or else, we may already have a reference to the node which contains the element to remove; or, we have an index of the element in the list. Just like inserting a new node in a list, removing requires that you have the reference to the predecessor node. And just like in insertion, removing the 1st node in the list is a "special" case — it does not have a predecessor, and removing it has the side-effect of changing the head. You can try to make this more efficient too!!! Time Complexity O(n) and Space Complexity O(1).

```
Method remove(index):
    if index == 0 and head is not null:
        head = head.next
        return
    prev = getNode(index - 1)
    if prev is not null and prev.next is not null:
        prev.next = prev.next.next
```

2.2.10 Copying a list:

Copying the elements of a source list to a destination list is simply a matter of iterating over the elements of the source list, and inserting these elements at the end of the destination list. Time Complexity O(n) and Space Complexity O(n).

```
Method copy():
    newList = new SinglyLinkedList()
    if head is null:
        return newList
    newList.head = new Node(head.elem)
    current = head.next
    newCurrent = newList.head
    while current is not null:
        newCurrent.next = new Node(current.elem)
        current = current.next
        newCurrent = newCurrent.next
    return newList
```

2.2.11 Reversing a list:

Since a linked list does not support random access, it is difficult to reverse a list in place without changing the head reference. Instead, we'll create a new list with its own head reference, and copy the elements in the reverse order. This method does not modify the original list, so we can call it an out-of-place method. Time Complexity O(n) and Space Complexity O(n).

```
Method reverseOutOfPlace():
    reversedList = new SinglyLinkedList()
    current = head
    while current is not null:
        newNode = new Node(current.elem)
        newNode.next = reversedList.head
        reversedList.head = newNode
        current = current.next
    return reversedList
```

The problem with this approach is that it creates a copy of the whole list, just in reverse order. We would like an in-place approach instead — re-order the links instead of copying the nodes! That would of course change the original list and would have a new head reference (which is the reference to the tail node in the original list). Time Complexity O(n) and Space Complexity O(1).

```
Method reverseInPlace():
    prev = null
    current = head
    while current is not null:
        next = current.next
        current.next = prev
        prev = current
        current = next
    head = prev
```

2.2.12 Rotating a list left:

Rotating a list left is much simpler than rotating an array — the 2nd node becomes the new head, and the 1st becomes the new tail node. We don't have to actually move the elements, it's just a matter of rearranging a few links. Time Complexity O(n) and Space Complexity O(1).

```
Method rotateLeft(k):
    if head is null or k <= 0:
        return
    size = count()
    k = k % size
    if k == 0:
        return
```

```
current = head
for i from 1 to k-1:
    current = current.next

newHead = current.next
current.next = null

tail = newHead
while tail.next is not null:
    tail = tail.next
tail.next = head
head = newHead
```

2.2.13 Rotating a list right:

Rotating a list right is almost the same as rotating left — the current last node becomes the new head and the current second last node becomes the last node. Time Complexity O(n) and Space Complexity O(1).

```
Method rotateRight(k):
    if head is null or k <= 0:
        return
    size = count()
    k = k % size
    if k == 0:
        return
    rotateLeft(size - k)
```

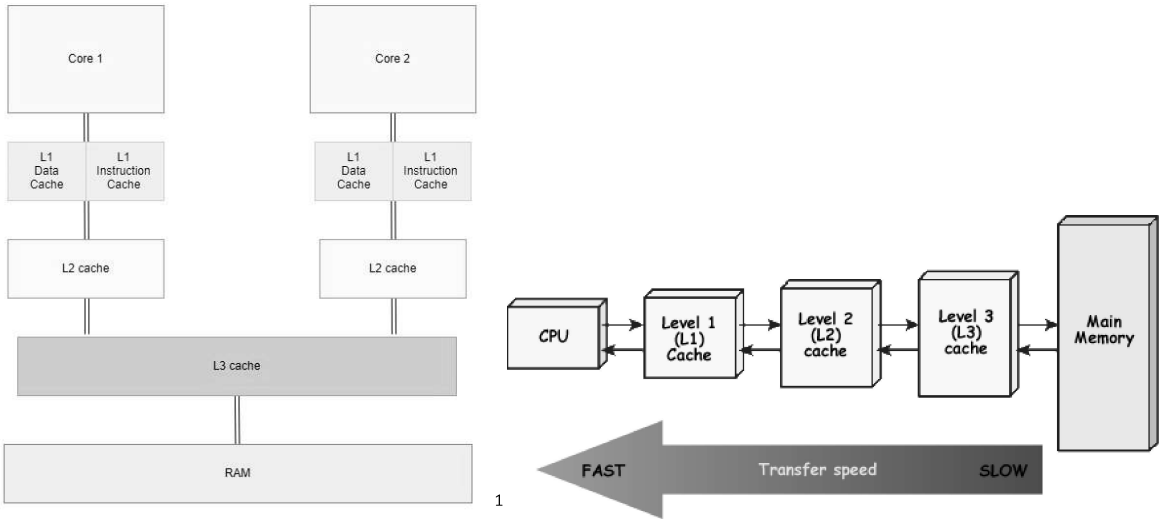
2.3 Understanding the Hidden Cost of Data Structures

By now you know that traversing all the elements of a linear array and doing that on a linked list have the same computational cost if the number of elements in the array and the list are the same. The cost is some multiple of the number of elements, N. However, what multiple? That is a big question and in a real world setting that matters a lot. To understand the issue, you need a little bit of initiation on how the computer CPU and memory are organized.

From the early days of computers, CPU is many times faster than the RAM (which is the memory) which in turn many times faster than the DISK (which is the computer's data storage). Since, to do any computation in the CPU, you need data; the computer designers applied a simple optimization to avoid hurting computer's performance due to the slowness of data read/write. The idea is that the RAM loads a large block of data at once, which is several KB or MB, from the storage even if the CPU asks for a single byte of data. The reason is, we expect in any typical program, CPU will ask for more data from locations nearby to its first request.

Surprisingly, or not, this assumption happens to be true for more than 98% of time for all data accesses of a program. However, even with this strategy, hardware designer realized that the slowness of the RAM (no longer the slowness of the disk, which has been taken care of) hurts

overall program performance greatly. So they just reused the same idea by introducing something called caches. Caches are like RAM but much smaller and faster (their small size is related to their faster speed).



[1]

So when a program runs, if the CPU ask for a single byte of data that is available in the RAM, the hardware actually load multiple consecutive bytes, at the range of 256 bytes or more from the RAM to the cache. The reason is the same; the hardware hopes that the program will make request for data among the additional bytes in near future. At that time, the request can be served from the cache. This idea of loading more data in closer memory works so good that now a days, all computers has multiple levels of caches between the CPU and the RAM.

This strategy for optimization affects the performance of element traversals of an array and a list differently. As you know, array elements are allocated in consecutive memory cells during the creation of the array. Therefore, traversal is extremely fast. There are just occasional data transfers between the cache and the RAM. However, space for a list element is allocated when you add the element to the list. The hardware gives whatever the next free memory cells available at that time. As a result, the elements are actually in different places of the RAM and are only connected together by the links of the linked list. So when you traverse a linked list, the hardware has to fetch data from the RAM to caches many times more. Therefore, traversing a linked list is significantly slower than traversing a linear array of the same size. It is like using Google MAP to go from a source location to a destination location. It asks you whether you will walk, bike (bicycle), take a car, or a bus despite the routes (aka intermediate locations you need to cross) being the same. Traversing the array is like taking a bus or a car. Traversing a linked list is like walking or biking.

Why do you not see the performance difference in your assignment programs? The reason it, the arrays and lists you use in your assignments are so small that both typically fit within the amount of data hardware load from the RAM to the cache in a few reads.

[1] Images: https://www.cybercomputing.co.uk/Languages/Hardware/cacheCPU.html, https://miro.medium.com/max/1030/1*Do424lmt3V06kJcmU1CEgg.png

2.4 Types of Linked List

Now we have an idea about Linked List and how a linked list works. One important thing to remember is that everything regarding the linked list is created and maintained manually by us. We have designed the node class and also created the linked list. After that, we implemented the idea of indexing along with different operations such as insert, removal, rotation, and so on. If you think about it all these operations are a bit complicated because of our node class design which only has the option of moving from one node to another in a forward manner.

This brings the question can we design our linked list in such a way that we can traverse from one node to another in a forward and backward manner? In addition, can we make our linked list circular and is there any way where we will not need to handle the head explicitly?

To answer all these questions, the types of the linked list have been introduced. Linked Lists type can be determined by considering three different categories. Each category is independent of the other two options. We need to understand first what these options mean.

Category 1	Category 2	Category 3
Non-Dummy Headed	Singly	Linear
Dummy Headed	Doubly	Circular

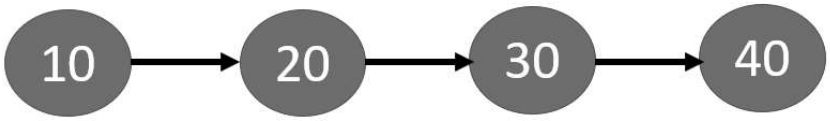
2.4.1 Non-Dummy Headed: It means that the linked list's first node (the head) contains an item along with the information of the next node. For example, the lists we have used till now are Non-Dummy headed.

Example: 10 → 20 → 30 → 40. In this Linked list, the first node stores the value head.

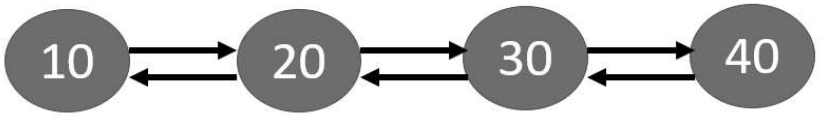
2.4.2 Dummy-Headed: This refers to the linked list where a head node is a node-type object but it does not store any element on its own. Rather it stores the information of the next node where the starting value is stored. The benefit of this is that we do not need to handle the first item of the data carefully now. To illustrate, we can remove or add items at beginning of the list without handling the head delicately. The reason is the first item is stored after the dummy head.

Example: DH → 10 → 20 → 30 → 40. In this linked list, DH refers dummy head which is a node without any element. The first item is stored after the dummy head.

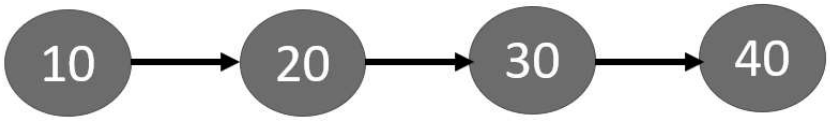
2.4.3 Singly: It means every node has only the information of its next node. The reason is the design of the Node class has only one variable (next variable). Up until now all the list we worked on are Singly Linked List.



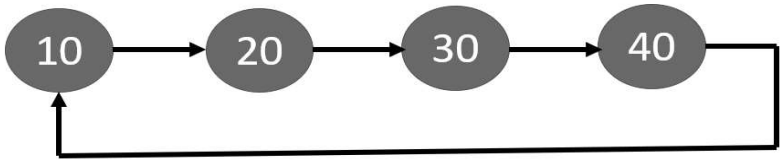
2.4.4 Doubly: It means a node knows its next node and its previous node's information. To achieve this we need to modify the Node class design where we store the previous node's information.



2.4.5 Linear: Linear linked list refers to the linked list where the linked list is linear in structure. To illustrate, the last node of the linked list will refer to None.



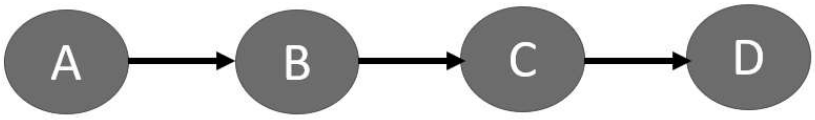
2.4.6 Circular: Circular linked list refers to a linked list where the linked list is circular in structure. To illustrate, the last node of the linked list will refer to the starting node.



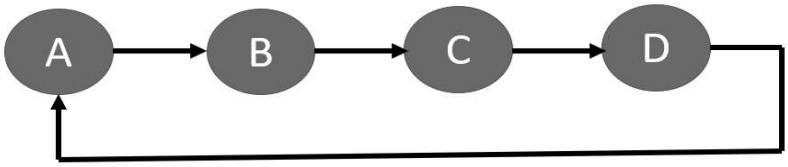
2.4.7 Explanation of the different linked list types
Now that we understand the options of each category, we can easily identify the type of a linked list. The main idea is that a linked list will contain one option from each category mentioned above. The option of one category does not have any impact on other categories.

The linked list we are taught until now is Non-Dummy Headed Singly Linear Linked List. You can think of this as a default linked list. It means if not mentioned, think of the head as Non-Dummy Head, Connection as Singly, and the structure as Linear. Now combining the categories the linked list can be 8 types. DH means dummy head in below diagram

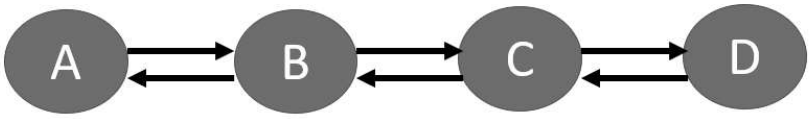
1. **Non-Dummy Headed Singly Linear Linked List:**



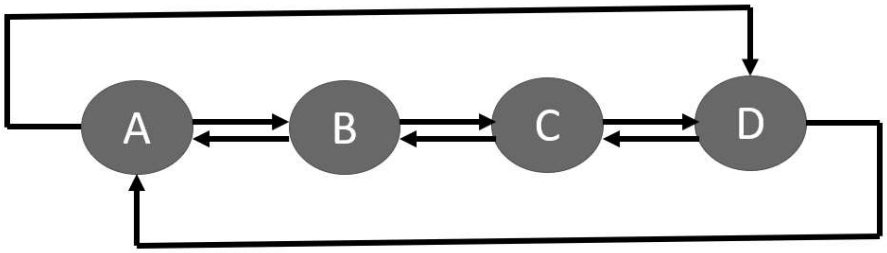
2. **Non-Dummy Headed Singly Circular Linked List:**



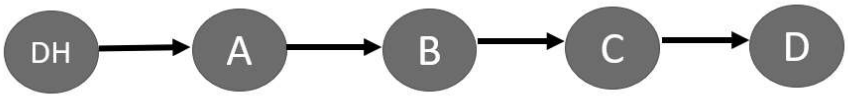
3. **Non-Dummy Headed Doubly Linear Linked List:**



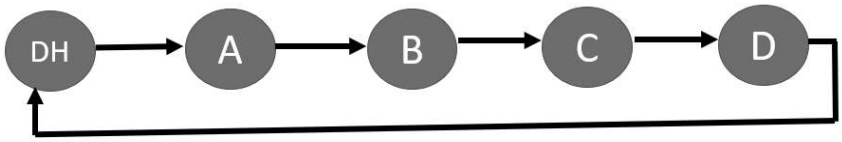
4. **Non-Dummy Headed Doubly Circular Linked List:**



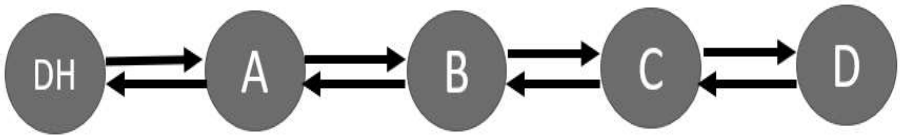
5. **Dummy Headed Singly Linear Linked List:**



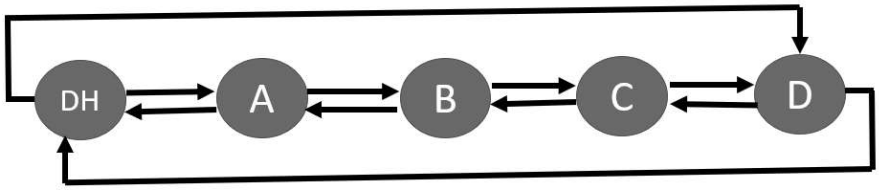
6. **Dummy Headed Singly Circular Linked List:**



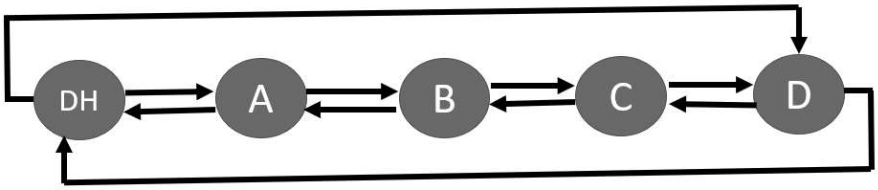
7. **Dummy Headed Doubly Linear Linked List:**



8. **Dummy Headed Doubly Circular Linked List:**



2.5 Dummy Headed Doubly Circular Linked List Operations



To practice we will learn about the dummy-headed doubly circular linked list. As the title suggests, the list will have a dummy head, doubly connection, and circular structure. For this reason, the Node class will be

```
Node:
    elem: Integer
    next: Node
    prev: Node
```

2.5.1 Creation:

It is similar to creating a linked list. The difference is that we need to ensure the connections in a proper way.

The below function takes an array and creates a dummy-headed doubly circular linked list.

```
CreateFromArray(array):
    FOR each element in array:
        Call InsertAtEnd(element)
```

2.5.2 Iteration:

It is similar to the previous linked list. The difference is that it will start from the next item of the dummy head and run till the pointer comes back to the dummy head. Time Complexity O(n) and Space Complexity O(1).

```
Iterate():
    Set current = head.next
    WHILE current != head:
        Print current.elem
        Set current = current.next
```

2.5.3 NodeAt:

The nodeAt method is similar to the previous one. The difference is that the dummy head does not represent any index as it does not have any value. Time Complexity O(n) and Space Complexity O(1).

```
GetNode(index):
   IF index < 0:
      RETURN NULL

   Set current = head.next
   Initialize currentIndex = 0

   WHILE current != head:
      IF currentIndex == index:
         RETURN current
      Increment currentIndex by 1
      Set current = current.next

   RETURN NULL (Index out of bounds)
```

Note that, if you want to count the number of total nodes, you should ignore the dummy head too.

2.5.4 Insertion:

To insert a new node in the list, you need the reference to the predecessor to link in the new node. Unlike for a singly-linked linear list, there is no "special" case here, since there is always a valid predecessor node available, thanks to the dummy head. Time Complexity O(n) and Space Complexity O(1).

```
InsertAtStart(element):
   Create newNode with elem = element
   Set first = head.next

   Set newNode.next = first
   Set newNode.prev = head
   Set head.next = newNode
   Set first.prev = newNode
```

```
InsertAtEnd(element):
   Create newNode with elem = element
   Set last = head.prev

   Set newNode.next = head
   Set newNode.prev = last
   Set last.next = newNode
   Set head.prev = newNode
```

```
InsertAtIndex(index, element):
   IF index < 0:
      RETURN

   IF index == 0:
      Call InsertAtStart(element)
      RETURN

   Set current = head.next
   Initialize currentIndex = 0

   WHILE current != head:
      IF currentIndex == index:
         Create newNode with elem = element
         Set prevNode = current.prev

         Set newNode.next = current
         Set newNode.prev = prevNode
         Set prevNode.next = newNode
         Set current.prev = newNode
         RETURN

      Increment currentIndex by 1
      Set current = current.next

   IF currentIndex == index: (Insert at end)
      Call InsertAtEnd(element)
```

2.5.5 Removal:

Removing an element from the list is done by removing the node that contains the element. Just like inserting a new node in a list, removing requires that you have the reference to the predecessor node. Since we're using a doubly-linked list, finding a predecessor of a node is trivial — it's n.prev. And thanks to the dummy head, there is no "special" case here as well. Time Complexity O(n) and Space Complexity O(1).

```
Remove(index):
   IF index < 0:
      RETURN

   Set current = head.next
   Initialize currentIndex = 0

   WHILE current != head:
      IF currentIndex == index:
```

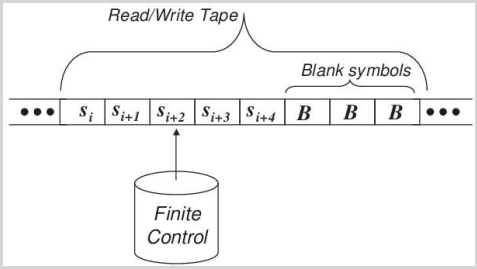
```
        Set prevNode = current.prev
        Set nextNode = current.next

        Set prevNode.next = nextNode
        Set nextNode.prev = prevNode
        RETURN

    Increment currentIndex by 1
    Set current = current.next

Print "Index out of bounds"
```

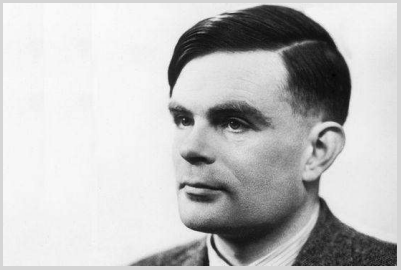
2.6 Reason for Doubly Linked List

By now, you know the difference between arrays and linked lists and their relative advantages and disadvantages. You learned about two types of linked list: single-linked list and double-linked list. Here we will discuss why double linked lists are important.

Although a single-linked list allows us to handle a linear sequence of elements that can be of unknown length and dynamic[2] in our program, there are some limitations in the traversal of the list. Suppose we want to do read/remove/insert near at the end of the list. Then we have to traverse all the way from the head to our position of interest then do the desired operation. When the list is long then this traversal cost is a huge problem. Furthermore, we have to be very careful regarding where to stop the traversal. For example, if you want to insert at position n, then we have to stop traversal at position n – 1. If you want to read then we stop traversal at n. If we make one more wrong step then we cannot backtrack. You have to restart from the head.

This second problem of not being able to move back-and-forth during our traversal is a major concern. In later courses, you will learn that moving backward on a list can be considered as going back in time to investigate the context of future events. This is highly useful for complex language parsing, intelligent agent design, and statistical modeling.





Some of you know about the famous mathematician and computer scientist Alan Turing. He investigated and proved many important properties about computers. One of his most fascinating contributions is the Turing Machine that can simulate any computer. By doing so, the Turing machine allows us to investigate the fundamental limitations and capabilities of all computers ever being built. The machine is

[2] means the sequence may change during program execution

surprisingly simple. It has an infinite tape of symbols and a read-write head that can make one-step forward or backward on that tape in each step or change the content of the tape cell where the head is located at that moment. The read-write head, also called the finite control, keeps track of its current state and decides what to do based-on the symbol on the tape in its position and its state.

So what kind of list you need to implement this tape?

Exercises

You can assume your node class is given.

```
class Node:
        def __init__(self, elem, next):
                self.elem = elem
                self.next = next
```

However, you may need to change the node class according to your problem.

2.1: Write a function that moves the last node to the front in a given Singly Linked List.

Examples:

Input: 1 → 2 → 3 → 4 → 5
Output: 5 → 1 → 2 → 3 → 4

Input: 3 → 8 → 1 → 5 → 7 → 12
Output: 12 → 3 → 8 → 1 → 5 → 7

2.2: Given two lists sorted in increasing order, create and return a new list representing the intersection of the two lists. The new list should be made with its own memory — the original lists should not be changed.

Example:

Input:
First linked list: 1 → 2 → 3 → 4 → 6
Second linked list be 2 → 4 → 6 → 8,
Output: 2 → 4 → 6.
The elements 2, 4, 6 are common in
both the list so they appear in the
intersection list.

Input:
First linked list: 1 → 2 → 3 → 4 → 5
Second linked list be 2 → 3 → 4,
Output: 2 → 3 → 4
The elements 2, 3, 4 are common in
both the list so they appear in the
intersection list.

2.3: You are given a linked list that contains N integers. You have performed the following reverse operation on the list:

Select all the subparts of the list that contain only even integers. For example, if the list is {1, 2, 8, 9, 12, 16}, then the selected subparts will be {2, 8}, {12, 16}.
Reverse the selected subparts such as {8, 2} and {16, 12}.
Now, you are required to retrieve the original list.

Input format

First line: N
Next line: N space-separated integers that denote elements of the reverse list
Output format

Print the N elements of the original list.

Sample Input
9
2 18 24 3 5 7 9 6 12

Sample Output
24 18 2 3 5 7 9 12 6

Explanation
In the sample, the original list is {24, 18, 2, 3, 5, 7, 9, 12, 6,} which when reversed according to the operations will result in the list given in the sample input.

2.4: Given a Linked List of integers, write a function to modify the linked list such that all even numbers appear before all the odd numbers in the modified linked list. Also, keep the order of even and odd numbers the same.

Examples:

Input: 17 → 15 → 8 → 12 → 10 → 5 → 4 → 1 → 7 → 6 → None
Output: 8 → 12 → 10 → 4 → 6 → 17 → 15 → 5 → 1 → 7 → None

Input: 8 → 12 → 10 → 5 → 4 → 1 → 6 → None
Output: 8 → 12 → 10 → 4 → 6 → 5 → 1 → None

2.5: Given a Linked List and a number N, write a function that returns the value at the Nth node from the end of the Linked List.

Example:
Input:
10 → 20 → 30 → 40 → 50 → None
N = 2
Output: 40

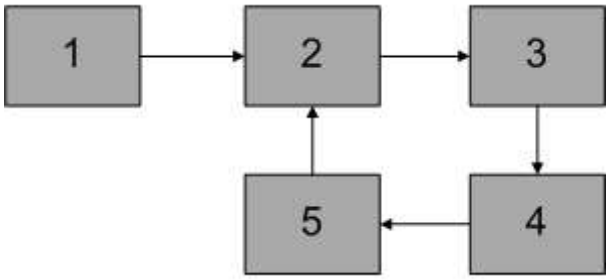
Input:
35 → 15 → 4 → 20 → 45 → None
N = 4
Output: 15

2.6: Given a singly linked list, find the middle of the linked list. If there are even nodes, then there would be two middle nodes, we need to print the second middle element.

Example:
Input: 10 → 20 → 30 → 40 → 50 → None
Output: 30

Input: 1 → 2 → 3 → 4 → 5 → 6 → None
Output: 4

2.7: Given a linked list, check if the linked list has a loop or not. The below diagram shows a linked list with a loop.



2.8: After getting her PhD, Christie has become a celebrity at her university, and her facebook profile is full of friend requests. Being the nice girl she is, Christie has accepted all the requests.

Now Kuldeep is jealous of all the attention she is getting from other guys, so he asks her to delete some of the guys from her friend list.

To avoid a 'scene', Christie decides to remove some friends from her friend list, since she knows the popularity of each of the friend she has, she uses the following algorithm to delete a friend.

```
Algorithm Delete(Friend):
   DeleteFriend=false
   for i = 1 to Friend.length-1
       if (Friend[i].popularity < Friend[i+1].popularity)
          delete i th friend
          DeleteFriend=true
          break
   if(DeleteFriend == false)
      delete the last friend
```

Input:

First line contains a T number of test cases. First line of each test case contains N, the number of friends Christie currently has and K ,the number of friends Christie decides to delete. Next line contains the popularity of her friends separated by space.

Output:
For each test case print N-K numbers which represent popularity of Christie friend's after deleting K friends.
NOTE:
Order of friends after deleting exactly K friends should be maintained as given in input.

Sample Input
3
3 1
3 100 1
5 2
19 12 3 4 17
5 3
23 45 11 77 18

Sample Output
100 1
19 12 17
77 18

2.9: A number is Palindrome if it reads same from front as well as back.For example,
2332 is a palindrome number as it reads the same from both sides.

Linked List can also be palindrome if they have the same order when it traverses from forward as well as backward.

Write a function that will take a linked list as input and return True if the list is a palindrome and return False otherwise.

Sample Input 1:
1 → 2 → 3 → 2 → 1 → None

Sample Output 1:
True

Sample Input 2:
1 → 2 → 3 → 1 → 1 → None

Sample Output 2:
False

2.10: Given a Singly Linked List, starting from the second node delete all alternate nodes of it.

Example:
Input: 10 → 20 → 30 → 40 → 50 → None
Ouput: 10 → 30 → 50 → None

Input: 1 → 2 → 3 → 4 → 5 → 6 → None
Output: 1 → 3 → 5 → None

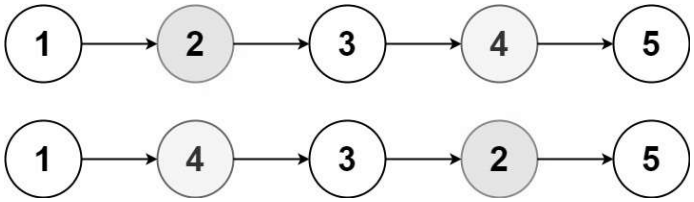
2.11: Two Linked Lists are identical when they have the same data and the arrangement of data is also the same. Write a function to check if the given two linked lists are identical.

Examples:
Input:
a = 1 → 2 → 3 → 4 → 5 → None
b = 1 → 2 → 3 → 4 → 5 → None
Output: Identical

Input:
a = 1 → 2 → 3 → 4 → None
b = 1 → 7 → 4 → 5 → None
Output: Not Identical

2.12: You are given a linked list, and an integer k.

Return the head of the linked list after swapping the values of the kth node from the beginning and the kth node from the end (the list is 1-indexed).



Example:
Input:
a = 1 → 2 → 3 → 4 → 5 → None
k = 2
Output: 1 → 4 → 3 → 2 → 5 → None

Input:
a = 1 → 7 → 4 → 5 → None
k = 2
Output: 1 → 4 → 7 → 5 → None

You can assume your node class is given for Doubly Linked List problems.

```
class Node:
    def __init__(self, elem, next, prev):
        self.elem = elem
        self.next = next
        self.prev = prev
```

However, you may need to change the node class according to your problem.

2.13: Given a non-dummy headed doubly non-circular linked list, write a function that returns true if the given linked list is a palindrome, else false.

Example:
Input: 1 ⇔ 7 ⇔ 7 ⇔ 1 ⇔ None
Output: True

Input: 1 ⇔ 7 ⇔ 4 ⇔ 5 ⇔ None
Output: False

2.14: Given a non-dummy headed doubly non-circular linked list, reverse the list.

Example:
Input: 10 ⇔ 20 ⇔ 30 ⇔ 40 ⇔ 50 ⇔ None
Output: 50 ⇔ 40 ⇔ 30 ⇔ 20 ⇔ 10 ⇔ None

2.15: Given a dummy headed doubly circular linked list, find the largest node in the doubly linked list.

Example:
Input: dummy_head ⇔ 10 ⇔ 70 ⇔ 40 ⇔ 15 ⇔ dummy_head (consider it circular)
Output: 70

2.16: Given a dummy headed doubly circular linked list, rotate it left by k node (where k is a positive integer)

2.17: Given a dummy headed doubly circular linked list, rotate it right by k node (where k is a positive integer)