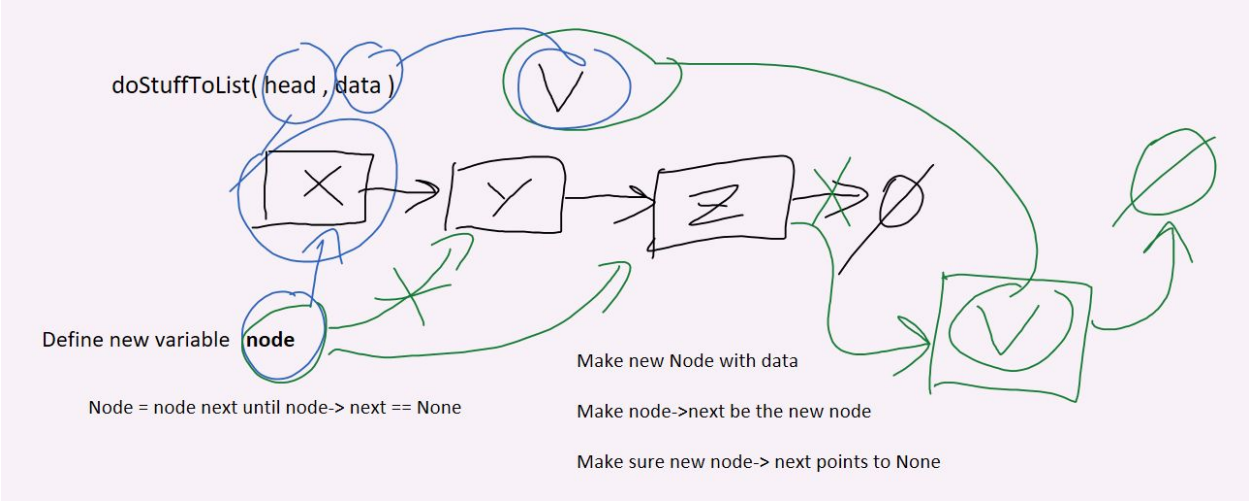
Before implementing each of these operations draw the functionality on a piece of paper, step through what happens and how the links connect, both with an empty list (head == None), a list with one node and a list with several nodes. *Explain this process to a peer!* Then take the steps from your drawing and translate into lines of code.

Example:



It is recommended to start with some fellow students, each pick some SLL functionality and do this exercise, explaining the different operations to each other. Then do this exercise for each new functionality that you implement.
This process will truly make the writing of the code much faster and more direct, and help deepen the understanding of what the code itself does.

1. **Node lists without an encapsulating class**

Here we will work with lists simply as a reference to the first node.

- Make the class ***Node***
  - Make a constructor with the instance variables ***data*** and ***next***
  - If needed you can take values for the variables as parameters in the constructor, but default them to ***None***.
- Implement a function that takes a node (***head***) and a value (***data***) and adds a node to the front of head
  - Return the new list (head)
  - Does your implementation work if the node that is sent in is ***None*** (empty list)?
- Implement an operation that takes a node (***head***) as a parameter and prints the value of all items in the list to the screen
  - Try to make an ***iterative*** version and a ***recursive*** one as well

- Implement a function that takes a node (***head***) as a parameter and removes the first item off of it
    - Return the new list (head)
    - Does this work if the node is the only item in the list (***head.next == None***)?
    - What if the head itself is ***None*** (empty list)?
- Implement an operation that takes a node (***head***) and a value (***data***) as parameters and adds the item to the back of the list
    - Is the time complexity of this as good as adding to the front?
    - What can be done in order to add as quickly to the back of the list as to the front?
- Implement an operation that ***removes*** an item from the ***back*** of a list
    - Is this more tricky than the other three (add to both ends and remove from front)?

2. Nodes and lists with an encapsulating class

Here we work with a class that has instance variables with nodes as well as other optional helper variables. For both classes implement ***__str__(self)*** that returns a string with data from all the nodes. You can also implement ***__str__(self)*** for the Node itself, if that helps.

- Make a class that implements a ***stack*** using singly-linked list nodes
    - The class must have an instance of ***Node***
        - Is it sufficient to have ***one*** Node variable to implement a ***stack***?
    - Implement a function that adds to the top of the stack (**push**)
    - Implement a function that returns data from and removes the top (**pop**)
    - What is the most efficient way to implement a ***get_size()*** function?
        - *It returns the number of items currently on the stack*
- Make a class that implements a ***queue*** using singly-linked list nodes
    - The class must have an instance of ***Node***
        - Is it sufficient to have ***one*** Node variable to implement a ***queue***?
    - Implement a function that adds to the back of the queue (**push**)
    - Implement a function that returns data from and removes the front (**pop**)
    - What is the most efficient way to implement a ***get_size()*** function?
        - *It returns the number of items currently in the queue*