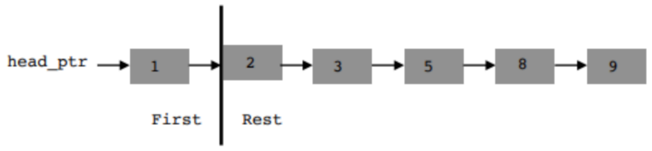
The objective is to update a Singly-Linked list class to support inversion and rotation functionality.

Task 1

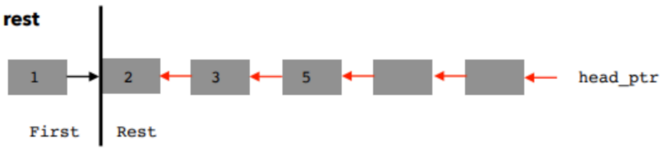
"A singly-linked list can be inverted" was once a song of dreams, a tale from a time before the invention of the doubly-linked list. Then, the budding programmer could iterate from head to tail, but they could not iterate from tail to head - that is, until one thought of a way to take a singly-linked list and invert it in order to make all of the links point in the opposite direction. Implement this inversion within the provided LinkedList files. Your implementation must be in O(n) time using O(1) extra space. This can be achieved by splitting your list into two parts, the first and the rest.
Here is an example:

To do so recursively, you can think of the solution to the problem as:

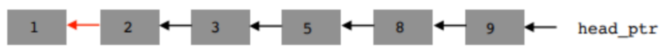
- **Split the list into first and rest**



- **Invert the rest**



- **Reconnect rest to first in the appropriate direction**

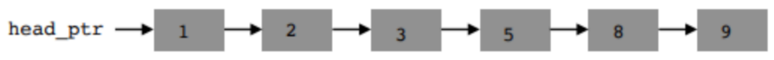


Write a public and non-recursive function invert( ) that calls a private and recursive invertRest( ). invertRest( ) must be implemented recursively and you may not create additional lists, containers, or copies of nodes to aid you in your venture.

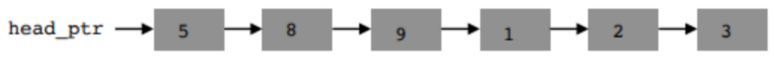
```
/**
   A wrapper to a recursive method that inverts the contents of the list

   @post the contents of the list are inverted such that:
      the item previously at position 1 is at position item_count_,
      the item previously at position 2 is at position item_count_-1 ...
      the item previously at position ⌊item_count/2⌋ is at position
         ⌈item_count_/2⌉
*/
void invert();


/**
   private function to invert, used for safe programming to avoid
   exposing pointers to list in public methods

   @post the contents of the list are inverted such that:
      the item previously at position 1 is at position item_count_,
      the item previously at position 2 is at position item_count_-1 ...
      the item previously at position ⌊item_count/2⌋ is at position
         ⌈item_count_/2⌉
*/
void invertRest(Node<T>* current_first_ptr);
```

Task 2

Implement a rotate function that, given any k, will shift all items in the caller LinkedList k positions to the right. Items that move beyond the final position must wrap around to the beginning.
Here is an example:



`rotate(3)` will produce:



rotate( ) is a public and recursive function that should be implemented with O(1) extra space, which means no additional lists, containers, or nodes are needed to implement it. This can be achieved by relinking the nodes currently in the list.

```
/**
   @pre k >= 0
   @post the contents of the list are rotated to the right by k places,
       so that every element at position i shifts to position i+k % item_count_
*/
void rotate(int k);
```

Hint: Think of the problem as "perform the smallest rotation (i.e. k = 1) by moving the last node to be the first and then rotate the rest recursively (k-1)."

You will submit the following files: **LinkedList.cpp** & **LinkedList.hpp**

For each class

- Implement one function/method and test it thoroughly (multiple test cases + edge cases if applicable).
- Implement the next function/method and test in the same fashion. How do you do this? Write your own main( ) function to test your classes. In this course you will never submit your test program, but you must always write one to test your classes. Choose the order in which you implement your methods so that you can test incrementally: i.e. implement mutator functions before accessor functions. Sometimes functions depend on one another. If you need to use a function you have not yet implemented, you can use stubs: a dummy implementation that always returns a single value for testing Don't forget to go back and implement the stub!!! If you put the word STUB in a comment, some editors will make it more visible.