※ 리스트 연습문제 **20230523 김난영**

1. 더미헤드를 사용하면 첫 번째 노드에 접근할 때 다른 노드들과 동일한 방식으로 접근할 수 있어 노드를 삽입하거나 할 때 노드 앞에 항상 직전 노드가 존재해 코드가 조금더 간단해진다는 장점이 있다.
하지만 이 더미노드가 존재하는 이유가 명확하지 않다면, 오히려 더미노드가 추가됨으로써 코드를 읽는 사람에게 혼란을 가져올 수 있다는 단점도 존재한다.

2. 리스트의 처음부터 끝까지 순서대로 값을 확인하면서 찾으면 True, 아니면 False를 반환.

```
current = self.head
while current is not None:
    if current.data == x:
        return True
    current = current.next
return False
```

3. getNode()를 활용하여 i번째 노드를 찾고, 그 이후 j번째 노드까지 순차적으로 출력하는 방식을 사용한다. 순차적으로 노드를 확인할 때, 현재 노드의 item을 출력하고 다음 노드로 이동하는 방식으로 작동하게 한다.

```
current = self._getNode(i)
for index in range(i, j + 1):
    print(current.item, end=" ")
    current = current.next
print()
```

4. 계속 돌아가는 리스트이기 때문에 노드가 원하는 범위를 초과하지 않게 범위를 설정해주고, 초과하면 오류메세지가 나오게 해준다. 그 이후로는 i번째 노드를 찾은 뒤, 그 이후로 j번째 노드까지 출력하면서 head로 돌아가면 종료되게 설정을 해준다.

```
if i < 0 or j < 0 or i > j:
    print("잘못된 인덱스 범위입니다.")
    return
current = self._getNode(i)
count = 0
while count <= j - i:
    print(current.item, end=" ")
    current = current.next
    count += 1
    if current == self._head:
        break
print()
```

5. __numItems는 리스트에 들어있는 원소의 개수를 의미함.
5-1) 재귀를 사용하지 않기위해 반복문을 사용할 것이다. 리스트를 순서대로 탐색하면서 각 노드를 하나씩 확인하고, 원소의 개수를 카운트 하는 방식을으로 알고리즘을 구현한다.
5-2) 재귀를 사용하는 알고리즘도 우선 노드를 하나씩 처리하며 리스트의 끝까지 원소의 개수를 계산한다.

지르고 재귀 호출을 사용하여 각 노드를 하나씩 순차적으로 처리하면서 카운트 값을 증가시켜나가는 방식으로 알고리즘을 구현한다.

5-1) 재귀를 사용하지 않는 알고리즘

```
def numItems(self):
    count = 0
    current = self._head.next
    while current is not None:
        count += 1
        current = current.next
    return count
```

5-2) 재귀 알고리즘

```
def numItems(self):
    return self._numItemsRecursive(self._head.next)

def _numItemsRecursive(self, node):
    if node is None:
        return 0
    else:
        return 1 + self._numItemsRecursive(node.next)
```

6. 제일 먼저 I가 유효한 범위 내에 있는지 확인한다. 그 다음에는 I번째 노드를 찾기 위해 prev는 I-1번째 노드를 가리키면서, curr이 지울 I번째 노드를 가리키게 설정한다. 이후에는 k개의 노드를 지우기위해 k번 반복하며 수행하게 만들고, prev.next를 curr.next로 연결해서 curr를 지운 후 다음 노드를 이동하며 원소의 수도 하나 감소시키는 방식으로 알고리즘이 진행되게 구현한다.

```
if i < 0 or i >= self._numItems:
    return None
prev = self._getNode(i - 1) if i > 0 else self._head
curr = prev.next
for _ in range(k):
    if curr is None:
        break
    prev.next = curr.next
    curr = curr.next
    self._numItems -= 1
```

7. 새 노드의 prev는 기존의 마지막 노드, next는 헤드노드를 가리키고, 기존 마지막 노드의 next는 새노드를 가리키며 헤드노드의 prev는 새 노드를 가리키는 방식으로 연결되게 구상한다.
그래서 새로운 노드 생성 후 기존 마지막노드의 next를 새 노드로 연결, 새로운 노드를 마지막 노드로 설정한뒤에 리스트 원소수를 증가시키는 순서로 알고리즘을 구현한다.

```
new_node = BidirectNode(x, self._head, self._head.prev)
self._head.prev.next = new_node
self._head.prev = new_node
self._numItems += 1
```

8. 우선 두 노드가 유효한 노드인지 검사를 한다. 그리고 그 이후에 current를 첫 번째 노드로 놓고 node1을 찾고, 만약 찾았다면 그 이후에 node2가 있는지도 계속 반복해서 탐색을 하여 최종적으로 두 노드가 같은 연결리스트에 있는 것이 확인되면 True를 return한다.

```
def isSameList(self, node1, node2) -> bool:
    if node1 is None or node2 is None:
        return False
```

```
    current = self._head.next
    while current:
        if current == node1:
            while current:
                if current == node2:
                    return True
                current = current.next
            return False
        current = current.next
    return False
```

9. 리스트를 처음부터 순차적으로 탐색하면서 x가 나올때마다 그 인덱스를 기록하는데, 제일 마지막에 나온 x의 인덱스를 반환하는 방법으로 코드를 구상할 것이다. 만약 x가 없으면 -1 반환하도록도 설정함.

```
  index = -1
  current = self._head.next
  current_index = 0
  while current:
      if current.item == x:
          index = current_index
      current = current.next
      current_index += 1
  return index
```

10. linkedlistbasic클래스에 추가하는 9번과 달리 순환구조인 cercularlinkedlist에 추가해야하니 이번엔 종료 조건을 명확히 리스트의 크기만큼으로 딱 정하고 반복하게 코드를 만드는 것에 중점을 둔다.

```
def lastIndexOf(self, x):
    curr = self._head.next
    last_index = -1
    for i in range(self._numItems):
        if curr.item == x:
            last_index = i
        curr = curr.next
    return last_index
```

※ 스택 연습문제

1. [15, 25] -> [40] -> [40, 10, 2] -> [40, 20] -> [20]
=> 15, 20이 차례대로 스택에 들어감, +연산자가 오면 들어있던 숫자 꺼내서 더한 후인 40을 다시 집어넣음, 다음으로 10, 2차례대로 스택에 들어감, *연산자가 오면 나중에 들어간 2, 10을 꺼내 곱한 후인 20을 다시 집어넣음, -연산자가 오면 먼저 나오는 20을 연산자 뒤로 40을 연산자 앞으로 해서 계산하면 40-20이니 그 결과 20을 다시 스택에 넣으면 최종적으로 20이 스택에 들어있는 상태가 된다.

2. push는 새로 추가되는 요소를 맨 앞에 추가해야하고, pop은 맨 앞에 있는 요소를 제거하고 반환해야

하며, top은 리스트의 첫 번째 요소가 탑이니 self.__stack[0]을 반환하도록 바꾸면 된다. 나머지 메서드들은 스택의 맨 앞이나, 뒤 관계없이 정상적으로 작동할 수 있기 때문에 바꾸지 않아도 괜찮다.

```
 def push(self, x):
     self.__stack.insert(0, x)

 def pop(self):
     return self.__stack.pop(0)

 def top(self):
     if self.isEmpty():
         print("No element in stack")
         return None
     else:
         return self.__stack[0]
```

3. 문자열을 입력받으면 함수로 넘어가 형태를 확인한다. 함수에서 $를 기준으로 문자열을 두 부분으로 나눠 앞 부분 w를 스택에 저장한 뒤, 뒷부분 w^R이 w의 순서를 뒤집어 놓은 형태인지 확인하여 True일 경우 '입력된 문자열은 집합의 원소입니다'를 출력하는 방법으로 알고리즘을 구상한다.

```
def isValidString(s: str) -> bool:
    parts = s.split('$')
    if len(parts) != 2:
        return False
    w, w_rev = parts
    stack = []
    for char in w:
        stack.append(char)
    for char in w_rev:
        if not stack:
            return False
        if stack.pop() != char:
            return False
    return True

input_string = input("문자열을 입력하세요: ")

if isValidString(input_string):
    print("입력된 문자열은 집합의 원소입니다.")
else:
    print("입력된 문자열은 집합의 원소가 아닙니다.")
```

4. LinkedStack의 클래스가 이미 구현이 되어있고, 객체 a도 이미 존재하는 상태라면 그걸 활용하여 b로 복사하는 코드는 b를 새로 만들고, a의 값을 꺼내 임시 스택에 저장한 뒤에 그걸 다시 b값에 넣는 방식으로 복사를 할 수 있다.

```
b = LinkedStack()
```

```
temp_stack = LinkedStack()
while not a.isEmpty():
    temp_stack.push(a.pop())
while not temp_stack.isEmpty():
    value = temp_stack.pop()
    b.push(value)
print(b.items)
```

5. 여는 소괄호가 나오면 스택에 넣고, 닫는 소괄호가 나오면 짝이 맞는 확인 후 틀리면 False를 출력하며 계속 반복하여 스택이 비어 있으면 모든 괄호가 짝을 맞춘 것이 되도록 코드를 구상한다.

```
    stack = LinkedStack()
    for char in s:
        if char == '(':
            stack.push(char)
        elif char == ')':
            if stack.isEmpty() or stack.pop() != '(':
                return False
    if stack.isEmpty():
        return True
    else:
        return False
```

6. 로직이 조금 더 복잡해진다.
=> 아래와 같이 괄호들의 짝을 서로 정해주는 딕셔너리를 추가해야하니 로직이 조금더 복잡해질 수 있다.

```
    stack = []
    matching_brackets = {')': '(', '}': '{', ']': '['}
    for char in s:
        if char in matching_brackets.values():
            stack.append(char)
        elif char in matching_brackets:
            if not stack or stack.pop() != matching_brackets[char]:
                return False
    return not stack
```

7. 최대 101개의 호출이 스택에 저장된다.
=> 처음 factorial(100)이 호출되면, 그 다음으로 factorial(99)가 호출되고 이렇게 계속해서 최대 factorial(0)이 될 때까지 호출을 하게 된다면 스택에는 최대 101개의 함수 호출이 스택에 저장된다고 할 수 있다.

8. 최대 50개의 호출이 스택에 저장된다.
=> fib(50)이 호출되면, fib(49), fib(48)이 호출되고, 그 과정에서 또다시 fib(49)는 fib(48), fib(47)을 호출 / fib(48)은 fib(47), fib(46)호출하는 방식처럼 중복되기에 이를 전부 고려하면 답은 2^{50}이라고 볼 수도 있지만 우리는 스택에 쌓이는 최대 함수 호출 수를 중복 없이 고려할때를 보자면 마지막에 호출되는 fib(1)이나 fib(2)가 포함될때까지 최대 50개의 함수 호출이 저장된다고 할 수 있다.