6차시

```
# 연결된 구조는 흩어진 데이터를 링크로 연결해서 관리한다.
# 용량이 고정되어 있지 않다.
# 중간에 자료를 삽입하거나 삭제하는 것이 용이하다.

# 노드 : 중간 다리 같은 역활을 하며, 데이터를 담을 데이터 필드와 링크를 담을 하나 이상의 링크 필드가 필요하다.
# 헤드포인터 : 시작부분의 링크를 담아두는 곳이다.

# 연결 리스트의 종류
# 단순 연결 리스트
# 원형 연결 리스트
# 이중 연결 리스트

# 단순연결 리스트의 응용 : 연결된 스택
# 노드 클래스 구현
class Node: # 단순연결리스트를 위한 노드 클래스
    def __init__(self, elem, link=None): # 생성자, 디폴트 인수 사용
        self.data = elem # 데이터 멤버 생성 및 초기화
        self.link = link # 링트 생성 및 초기화

# 연결된스택 클래스 구현
class LinkedStack:
    def __init__(self): # 생성자
        self.top = None # top 생성 및 초기화

    def isEmpty(self): # 공백상태 검사
        return self.top == None

    def clear(self): # 스택 초기화
        self.top = None

    def push(self, item): # 연결된 스택의 삽입연산
        n = Node(item, self.top) # 새로운 공간(노드)에 추가할 값과 시작링크를 대입한다.
        self.top = n # 시작링크를 새롭게 만든 공간에 연결해준다.

    def pop(self): # 연결된 스택의 삭제연산
        if not self.isEmpty(): # 공백이 아니면
            n = self.top # 현재 가장 위에 있는 데이터를 새로운 공간(노드)에 대입한다.
            self.top = n.link # 시작링크을 가장 위에 있던 데이터 다음의 링크로 걸어둔다.
            return n.data # 새로운 공간에 저장해둔 노드의 데이터를 리턴한다.

    def size(self): # 스택의 항목수 계산
        node = self.top # 시작 노드
        count = 0
        while not node == None: # node가 None이 아닐 때 까지
            node = node.link # 다음 노드로 이동
            count += 1 # count 증가
        return count # count 반환

    def display(self, msg='LinkedStack'):
        print(msg, end=" ")
        node = self.top # 시작노드를 새로운 공간(노드)에 저장한다.
        while not node == None: # 노드에 값이 있으면 반복
            print(node.data, end=" ") # 값을 출력
            node = node.link # 다음노드를 가리키는 링크를 노드에 대입한다.
        print()
```

```
    def peek(self): # 가장 위에 있는 값 반환
        if not self.isEmpty(): # 비어있지 않으면
            return self.top.data # 시작 노드의 데이터를 리턴

# 본문
if __name__ == "__main__" :

    odd = LinkedStack()
    even = LinkedStack()

    for i in range(10):
        if i%2 == 0:
            even.push(i)
        else:
            odd.push(i)

    even.display(' 스택 even push 5회: ')
    odd.display(' 스택 odd  push 5회: ')

    print(' 스택 even peek: ', even.peek())
    print(' 스택 odd  peek: ', odd.peek())

    for _ in range(2):
        even.pop()
    for _ in range(3):
        odd.pop()

    even.display(' 스택 even pop 2회: ')
    odd.display(' 스택 odd  pop 3회: ')

'''
 스택 even push 5회:  8 6 4 2 0
 스택 odd  push 5회:  9 7 5 3 1
 스택 even peek:  8
 스택 odd  peek:  9
 스택 even pop 2회:  4 2 0
 스택 odd  pop 3회:  3 1
'''
```

```
# 단순연결 리스트의 응용 : 연결된 리스트
# 연결된 리스트

# 노드 : 중간 다리 같은 역활을 하며
# 데이터를 담을 데이터 필드와 링크를 담을 하나 이상의 필드가 필요하다.

# 연결 리스트 클래스

class Node: # 단순연결리스트를 위한 노드 클래스
    def __init__(self, elem, link=None): # 생성자, 디폴트 인수 사용
        self.data = elem # 데이터 멤버 생성 및 초기화
        self.link = link # 링트 생성 및 초기화

class LinkedList: # 연결된 리스트 클래스
    def __init__(self):
        self.head = None

    def isEmpty(self): # 공백상태 검사
        return self.head == None

    def clear(self): # 리스트 초기화
        self.head = None

    def size(self): # 스택의 항목수 계산
        node = self.head # 시작노드를 새로운 공간(노드)에 저장한다.
        count = 0
        while not node == None: # node가 None이 아닐 때 까지
            node = node.link # 다음 노드로 이동
            count += 1 # count 증가
        return count # count 반환

    def getNode(self, pos): # pos번째 노드 반환
        if pos < 0: # 입력된 pos값이 1보다 작을경우
            return None
        node = self.head; # 시작노드를 새로운 공간(노드)에 저장한다.
        while pos > 0 and node != None: # pos가 0보다 크고 노드가 비어있지 않으면 반복
            node = node.link # node를 다음 노드로 이동
            pos -= 1 # 남은 반복 횟수 줄임
        return node # 최종 노드 반환

    def getEntry(self, pos): # pos번째 노드의 데이터 반환
        node = self.getNode(pos) # pos번째 있는 데이터를 찾아서 새로운 공간(노드)에 저장한다.
        if node == None: # 찾는 노드가 없는 경우
            return None
        else: # 그 노드의 데이터를 반환
            return node.data

    def insert(self, pos, elem):
        before = self.getNode(pos-1) # 앞에 있는 노드를 찾아서 비포노드에 저장
        if before == None: # 비포노드에 값이 없으면, 시작노드 앞에 값을 대입.
            self.head = Node(elem, self.head) # 새롭게 만든 노드를 사작노드에 대입
        else: # 중간에 앞에 삽입하는 경우
            # 새로운 공간(노드)에 새로운 값과, 3번째 노드를 가리키던 before.link를 연결
            node = Node(elem, before.link)
            # 세번째 노드를 가리키던 before.link에 새로운 공간(노드)를 연결한다.
            before.link = node
```

```
def delete(self, pos):
    before = self.getNode(pos-1) # before 노드를 찾음
    if before == None: # 삭제하고 싶은 노드 앞에 노드가 없다면(시작노드를 삭제하는 경우)
        if self.head is not None : # 시작노드가 공백이 아니면(공백이면 안에 들어있는 값이 없다고 판단)
            self.head = self.head.link # 시작노드를 시작노드에 연결된 노드로 연결
    elif before.link != None: # 중간에 있는 노드의사제 삭제
        before.link = before.link.link # 지우고 싶은 노드를 가리키는 링크(before.link)에 다음 노드를 연결한다.

def display(self, msg='LinkedStack'): # 메세지를 출력(디폴트값을 미리 대입해 놓는다.)
    print(msg, end=" ") # 입력받은 값을 먼저 출력을 하고
    node = self.head # 헤드 포인트의 주소를 저장
    while not node == None: # 연결포인트에 값이 없지않으면 무한반복
        print(node.data, end=" -> ")
        node = node.link # 다음 연결포인트를 저장
    print()

def peek(self):
    if not self.isEmpty(): # 비어있지 않으면
        return self.head.data # 시작노드의 데이터를 반환

def replace(self, pos, elem): # 값을 대체
    node = self.getNode(pos) # 값을 대체할 노드의 위치를 찾는다.
    if node != None: # 그 노드가 존재한다면
        node.data = elem # 그 노드에 값에 새로 입력할 데이터를 삽입한다.

def find(self, val):
    count = 0
    node = self.head # 시작노드를 새로운 공간(노드)에 저장한다.
    while not node == None: # 노드에 더이상 값이 없을 경우까지 반복
        count += 1
        if node.data == val: # 만약 찾고있는 값이랑 같은경우
            break
        node = node.link
    return count
```

```
# 본문
if __name__ == "__main__" :

    s = LinkedList()
    s.display('단순연결리스트로 구현한 리스트(초기상태):')

    s.insert(0,10)
    s.insert(0,20)
    s.insert(1,30)
    s.insert(s.size(), 40)
    s.insert(2,50)
    s.display('단순연결리스트로 구현한 리스트(삽입*5):')

    s.replace(2,90)
    s.display('단순연결리스트로 구현한 리스트(교체*1):')

    print('단순연결리스트로 구현한 리스트(값찾기): %d는 %d번째에 있다.'%(90, s.find(90)))

    s.delete(2);
    s.delete(s.size()-1)
    s.delete(0)
    s.display('단순연결리스트로 구현한 리스트(삭제*3):')

    s.clear()
    s.display('단순연결리스트로 구현한 리스트(정리후):')

'''
단순연결리스트로 구현한 리스트(초기상태):
단순연결리스트로 구현한 리스트(삽입*5): 20 -> 30 -> 50 -> 10 -> 40 ->
단순연결리스트로 구현한 리스트(교체*1): 20 -> 30 -> 90 -> 10 -> 40 ->
단순연결리스트로 구현한 리스트(값찾기): 90는 3번째에 있다.
단순연결리스트로 구현한 리스트(삭제*3): 30 -> 10 ->
단순연결리스트로 구현한 리스트(정리후):
'''
```

```
# 원형연결리스트의 응용 : 연결된 큐

# 단순연결리스트로 구현한 큐 : 크기를 미리 정하고, front와 rear를 이용하여 구현
# 원형연결리스트로 구현한 큐 : 크기를 미리 정의할 필요가 없고, tail를 이용하여 rear와 front에 바로 접근할 수 있어 효율적이다.
# front -> tail.link
# rear -> tail
# 용량 제한이 없고, 삽입/ 삭제가 모두 O(1) 이다.

# 원형 연결 큐 구현

class Node: # 단순연결리스트를 위한 노드 클래스
    def __init__(self, elem, link=None): # 생성자, 디폴트 인수 사용
        self.data = elem # 데이터 멤버 생성 및 초기화
        self.link = link # 링트 생성 및 초기화

class CircularLinkedQueue:

    def __init__(self): # 생성자 함수
        self.tail = None # tail : 유일한 데이터

    def isEmpty(self): # 공백상태 검사
        return self.tail == None

    def clear(self): # 큐 초기화
        self.tail = None

    def peek(self): # Peek연산
        if not self.isEmpty(): # 공백이 아니면
            return self.tail.link.data # front(tail.link)의 data를 반환

    def enqueue(self, item):
        # 일단 대입할 데이터를 담은 공간(노드)를 생성한다. 위치(링크)는 아직 모르므로 None로 가정한다.
        node = Node(item, None)
        if self.isEmpty(): # 만약 큐가 비어져 있으면(값이 지금 대입한 값밖에 없으면)
            node.link = node # 새 공간의 링크를 자기자신으로 설정
            self.tail = node # 꼬리 노드를 새공간에 설정한다.
        else:
            # 새공간이 새로운 꼬리노드가 될것이므로 새로운공간(노드)의 링크에 front(tail.link)를 연결한다.
            node.link = self.tail.link
            self.tail.link = node # 원래 front를 가리키던 tail.link에 새롭게 생성된 노드를 연결해준다.
            self.tail = node # 꼬리 노드에 새로운 공간(노드)를 연결한다.

    def dequeue(self):
        if not self.isEmpty(): # 비워져있지 않으면
            data = self.tail.link.data # front(tail.link)의 값을 출력해야하므로
            if self.tail.link == self.tail: # front(tail.link)와 rear(tail)가 같을경우, 즉 큐에 값이 하나밖에 없을 경우
                self.tail = None # 값을 반환하면 이제 큐에 감긴 원소가 업으므로 None
            else:
                # 원소가 2개 이상있다면, rear(tail)에 front(tail.link)의 다음 노드(tail.link.link)를 연결한다.
                self.tail.link = self.tail.link.link
            return data # 처음에 저장해 놨던 값을 반환
```

```
    def size(self):
        if self.isEmpty(): # 공백 : 0반환
            return 0
        else: # 공백이 아니면
            count = 1 # count는 최소 1
            node = self.tail.link # front(tail.link) 노드를 새로운 공간(노드)에 저장한다.
            while not node == self.tail: # node가 rear(tail)를 만날 때까지
                node = node.link # 노드를 차례대로 이동
                count += 1 # count 증가
            return count # 최종 count 반환

    def display(self, msg='CircularLinkedQueue'): # 메세지를 출력(디폴트값을 미리 대입해 놓는다.)
        print(msg, end=" ") # 입력받은 값을 먼저 출력을 하고
        if not self.isEmpty(): # 큐가 비어있지 않으면
            node = self.tail.link # front(tail.link) 노드를 새로운 공간(노드)에 저장한다.
            while not node == self.tail : # node가 rear(tail)를 만날 때까지
                print(node.data, end=" -> ")
                node = node.link # 다음 노드로 이동
            # self.tail의 node는 비교에 의해 while문에 들어가지 못하므로
            # 반복이 끝나고 한번더 출력을 진행해야한다.
            print(node.data, end=" (fin) ")
        print()

# 본문
if __name__ == "__main__":
    q = CircularLinkedQueue() # 객체 생성
    MAX_QSIZE = 10 # 원형 연결 큐의 크기

    for i in range(8):
        q.enqueue(i) # 0, 1, ... , 7 대입

    q.display()

    for i in range(5):
        q.dequeue() # 전위부터 5개 반환하고 삭제

    q.display()

    for i in range(8,14):
        q.enqueue(i) # 8, 9, ... , 13 대입

    q.display()

'''
CircularLinkedQueue 0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 (fin)
CircularLinkedQueue 5 -> 6 -> 7 (fin)
CircularLinkedQueue 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13 (fin)
'''
```

```
# 이중연결리스트의 응용 : 연결된 덱

# 연결된 덱을 이중연결리스트로 구현하는 이유
# 단순연결리스트로 구현한 덱에선 DeleteRear를 진행할 때 바로 후단의 링크를 알수가 없어 불가능하다. O(n)
# 이를 해결하기 위해 이중연결리스트를 사용한다.
# 이중연결리스트로 구현한 덱에서 후단의 링크를 바로 알수 있으므로 DeleteRear를 문제없이 사용할 수 있다. O(1)

# 이중연결리스트를 구현하기 위한 양방향 노드 믈래스 생성
class DNode:
    def __init__(self, elem, prev = None, next = None): # 전단과 후단의 링크는 일단 None로 디폴트 값을 설정한다.
        self.data = elem
        self.prev = prev # 전단의 링크
        self.next = next # 후단의 링크

# 이중연결리스트를 이용한 덱 구현
class DoublyLinkedDeque:
    def __init__(self):
        self.front = None # 맨 앞에 있는 노드를 가리킨다.
        self.rear = None # 맨 뒤에 있는 노드를 가리킨다.

    def isEmpty(self):
        return self.front == None

    def clear(self):
        self.front = self.front = None

    def size(self): # 스택의 항목수 계산
        node = self.front # 맨 앞에 있는 노드의 값을 새로운 공간(노드)에 저장한다.
        count = 0
        while not node == None: # node가 None이 아닐 때 까지
            node = node.next # 다음 노드로 이동
            count += 1 # count 증가
        return count # count 반환

    def display(self, msg='LinkedQueue'): # 메세지를 출력(디폴트값을 미리 대입해 놓는다.)
        print(msg, end=" ") # 입력받은 값을 먼저 출력을 하고
        node = self.front # 맨 앞에 있는 노드의 값을 새로운 공간(노드)에 저장한다.
        while not node == None:
            print(node.data, end=" ")
            node = node.next # 다음의 노드로 링크를 옮긴다.
        print()

    def addFront(self, item): # 앞에서 입력
        # 입력되는 값과 맨 앞에 삽입되는 것이므로 전단은 None, 루단에 이어질 링크는 원래 맨 앞이였던 front를 대입한다.
        node = DNode(item, None, self.front)
        if (self.isEmpty()): # 덱스가 비어있으면 원소가 하나밖에 없다는 뜻이다.
            self.front = self.rear = node # front와 rear의 링크를 자기자신으로 가리킨다.
        else: # 공백이 아니면, 값이 덱스에 들어있다는 뜻
            self.front.prev = node # 현재 front는 앞에서 두번째 노드이므로 첫번째 노드와 전단 링크를 연결한다.
            self.front = node # front에 새롭게 생성한 노드를 위치 시킨다.

    def addRear(self, item): # 후단에서 입력
        # 입력되는 값과 맨 뒤에서 삽입되는 것이므로 후단은 None, 전단은 원래 맨 뒤에 있던 rear를 대입한다.
        node = DNode(item, self.rear, None)
        if(self.isEmpty()): # 덱스가 비어있으면 원소가 하나밖에 없다는 뜻이다.
            self.front = self.rear = node # front와 rear의 링크를 자기자신으로 가리킨다.
```

```
        else: # 공백이 아니면, 덱스에 값이 들어있다면
            self.rear.next = node # 현재 rear는 뒤에서 두번째 노드이므로 후단에 현재 맨 뒤 노드인 node를 연결한다.
            self.rear = node # rear에 현재 맨 뒤 노드인 node를 연결한다.

    def deleteFront(self): # 전단에서 삭제
        if not self.isEmpty(): # 덱스가 비어져 있지 않다면
            data = self.front.data # front에 있던 노드의 데이터를 출력
            self.front = self.front.next # front에 두번째 앞에 있던 노드(front.next)를 연결한다.
            if self.front == None: # 만약 전단 노드가 없다면, 즉 덱스에 아무 값도 없다면
                self.rear = None # rear도 None값을 대입
            else: # 덱스에 남아 있는 값이 있다면
                self.front.prev = None # 이제 앞에서 2번째 노드가 맨 앞의 노드가 되었으므로 front의 전단링크를 None로 바꾼다.
            return data # 값을 리턴

    def deleteRear(self): # 후단에서 삭제
        if not self.isEmpty(): # 덱스가 비어져 있다면
            data = self.rear.data # 맨 후단의 데이터를 삽입
            self.rear = self.rear.prev # 맨 뒤에서 2번째로 있던 값에 rear를 배치한다.
            if self.rear == None: # 만약 후단 노드가 없다면, 즉 덱스에 아무 값도 없다면
                self.front = None # front도 None으로 설정
            else: # 덱스에 값이 들어있으면
                self.rear.next = None # 이제 뒤에서 2번째 노드가 맨 뒤에 노드가 되었으므로 rear의 후단링크를 None로 바꾼다.
            return data # step4

# 본문
if __name__ == "__main__":
    dq = DoublyLinkedDeque()
    for i in range(9):
        if i%2 == 0: # 짝수는 뒤에서 입력
            dq.addRear(i)
        else: # 홀수는 앞에서 입력
            dq.addFront(i)

    dq.display('DoublyLinkedDeque :')

    for i in range(2): # 전단에서 삭제
        dq.deleteFront()

    for i in range(3): # 후단에서 삭제
        dq.deleteRear()

    dq.display('DoublyLinkedDeque :')

    for i in range(9,14):
        dq.addFront(i)

    dq.display('DoublyLinkedDeque :')
'''
DoublyLinkedDeque : 7 5 3 1 0 2 4 6 8
DoublyLinkedDeque : 3 1 0 2
DoublyLinkedDeque : 13 12 11 10 9 3 1 0 2
9 대신 20 으로 대입했을 경우
DoublyLinkedDeque : 19 17 15 13 11 9 7 5 3 1 0 2 4 6 8 10 12 14 16 18
DoublyLinkedDeque : 15 13 11 9 7 5 3 1 0 2 4 6 8 10 12
DoublyLinkedDeque : 13 12 11 10 9 15 13 11 9 7 5 3 1 0 2 4 6 8 10 12
'''
```

```
# 회문을 검사라는 함수(연결된스택 사용)

from LinkedStack import LinkedStack
from LinkedStack import Node

def Palindrome(list):
    list1 = list
    list = list.lower() # 대문자로 입력을 할 것을 대비하여 소문자로 변경
    list2 = [] # 콤마, 마침표, 괄호, 특수기호, 숫자 등을 제거한 리스트를 저장하기 위해 공간 생성

    for s in list: # 콤마, 마침표, 괄호, 특수기호, 숫자 등이 입력 되는 것을 막기 위해 아스키코드 값을 사용
        if s >="a" and s <="z":
            strs.push(s) # 연결스택에 값을 대입한다.
            list2.append(s) # 리스트에 값을 대입한다.

    print("입력된 문자열 : ", ''.join(list1))
    print("정제된 문자열 : ",''.join(list2))

    num = 1 # 참일 경우 1을 유지
    for i in range(strs.size()):
        if list2[i] != strs.pop(): # 양쪽에 값을 비교하고 삭제하며 회문을 검사
            num = 0 # 거짓일 경우 0으로 변경
    return num # 참:1 / 거짓:0 을 반환

# 본문
if __name__ == "__main__":
    strs = LinkedStack()
    list = input("회문인지 확인할 출력할 문자열을 입력하세요 : ")
    num = Palindrome(list) # 사용자 지정 함수를 이용하여 회문 검사
    print("회문입니다.") if num==1 else print("회문이 아닙니다.") # 출력문 작성

'''
회문인지 확인할 출력할 문자열을 입력하세요 : madam, i'm Adam
입력된 문자열 :  madam, i'm Adam
정제된 문자열 :  madamimadam
회문입니다.
'''
'''
회문인지 확인할 출력할 문자열을 입력하세요 : a"sDf..gF-d,Ea
입력된 문자열 :  a"sDf..gF-d,Ea
정제된 문자열 :  asdfgfdea
회문이 아닙니다.
'''
```