7차시

```
# 정렬이란 데이터를 순서대로 재배열하는 것
# 가장 기본적이고 중요한 알고리즘으로 오름차순과 내림차순이 있다.

# 용어정리
# 레코드 : 정렬시켜야 될 대상으로 여러개의 필드로 이루져있다.
# 정렬 키 : 정렬의 기준이 되는 필드
# 정렬 : 레코드들을 키의 순서로 재배열하는 것

# 정렬 장소에 따른 분류(데이터의 크기가 크면 메모리에 넣기 힘들다.)
# 내부정렬 : 모든 데이터가 메인 메모리
# 외부정렬 : 외부 기억 장치에 대부분의 레코드

# 단순하지만 비효율적인 방법
# 삽입, 선택, 버블 정렬 등

# 복잡하지만 효율적인 방법
# 퀵, 힙, 병합, 기수 정렬, 팀 등

# 정렬 알고리즘의 안전성(stability)
# 동일한 값을 가진 데이터를 정렬할때 두 데이터의 위치가 바뀌면 안정성을 충족하지 않는다.
```

```
# 간단한 정렬 알고리즘

# 선택 정렬
# 오른쪽 리스트에서 가장 작은 숫자(최솟값)를 선택하여 왼쪽 리스트의 맨 뒤로 이동하는 작업을 반복

# 시간 복잡도 : O(n^2) -> for문이 두번 돌아가므로

def selection_sort(A) : # 반복을 진행하며 가장 작은 값을 왼쪽으로 배치한다.
    n = len(A)
    for i in range(n-1): # 0부터 -2까지의 인덱스 차례로 선택
        least = i; # key값 입력
        for j in range(i+1, n): # i보다 큰 인덱스를 차례로 선택
            if(A[j] < A[least]): # j값이 나타내는 값이 더 작으면
                least = j # j값을 저장
        A[i], A[least] = A[least], A[i] # 가장작은 값은 나타내는 인덱스와 시작 인덱스를 이용하여 값 스왑
        printStep(A, i+1); # 출력문 선언

def printStep(arr, val):
    print("    step %2d = " % (val), end='')
    print(arr)

if __name__ == "__main__":
    data = [5, 3, 8, 4, 9, 1, 6, 2, 7]
    print("Original :  ", data)
    selection_sort(data)
    print("Selection : ", data)

# 출력
# Original :   [5, 3, 8, 4, 9, 1, 6, 2, 7]
#     step  1 = [3, 5, 8, 4, 9, 1, 6, 2, 7]
#     step  2 = [3, 5, 8, 4, 9, 1, 6, 2, 7]
#     step  3 = [3, 4, 5, 8, 9, 1, 6, 2, 7]
#     step  4 = [3, 4, 5, 8, 9, 1, 6, 2, 7]
#     step  5 = [1, 3, 4, 5, 8, 9, 6, 2, 7]
#     step  6 = [1, 3, 4, 5, 6, 8, 9, 2, 7]
#     step  7 = [1, 2, 3, 4, 5, 6, 8, 9, 7]
#     step  8 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
# Selection :  [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
# 삽입 정렬
# 정렬되어 있는 부분에 새로운 레코드를 올바른 위치에 삽입하는 과정

# 복잡도 분석 : O(n^2) -> 역순으로 정렬되어 있는 경우

# 특징
# 많은 이동 필요 - >  레코드가 큰 경우 불리
# 안정된 정렬방법
# 대부분 정렬되어 있으면 매우 효율

# 알고리즘
# i=1 key=3 j= 0 T 5 3 8 4 9 1(변환 전)
#                  5 5 8 4 9 1(변환 후)
#           j=-1 F 5 5 8 4 9 1(변환 전)
#                  3 5 8 4 9 1(변환 후) key값 대입
# i=2 key=8 j= 1 F 3 5 8 4 9 1(변환 전)
#                  3 5 8 4 9 1(변환 후) key값 대입
# i=3 key=4 j= 2 T 3 5 8 4 9 1(변환 전)
#                  3 5 8 8 9 1(변환 후)
#           j= 1 T 3 5 8 8 9 1(변환 전)
#                  3 5 5 8 9 1(변환 후)
#           j= 0 T 3 5 5 8 9 1(변환 전)
#                  3 4 5 8 9 1(변환 후) key값 대입
# i=4 key=9 j= 3 F 3 4 5 8 9 1(변환 전)
#                  3 4 5 8 9 1(변환 후) key값 대입
# i=5 key=1 j= 4 T 3 4 5 8 9 1(변환 전)
#                  3 4 5 8 9 9(변환 후)
#           j= 3 T 3 4 5 8 9 9(변환 전)
#                  3 4 5 8 8 9(변환 후)
#           j= 2 T 3 4 5 8 8 9(변환 전)
#                  3 4 5 5 8 9(변환 후)
#           j= 1 T 3 4 5 5 8 9(변환 전)
#                  3 4 4 5 8 9(변환 후)
#           j= 0 T 3 4 4 5 8 9(변환 전)
#                  3 3 4 5 8 9(변환 후)
#           j=-1 F 3 3 4 5 8 9(변환 전)
#                  1 3 4 5 8 9(변환 후) key값 대입


def insertion_sort(A):
    n = len(A)
    for i in range(1, n):
        key = A[i]
        j = i-
        # j >= 0 : 인덱스는 0이상 이므로 음수가 되면 반복을 중단한다.
        # A[j] > key : 인덱스를 줄이면서 key값과 비교, key값보다 값이 크면 반복
        while j >= 0 and A[j] > key :
            A[j+1] = A[j] #
            j -= 1
        A[j+1] = key # 비교하는 key값보다 작은 값이 나오면 그 인덱스 바로 오른쪽에 삽입
        printStep(A, i)

def printStep(arr, val):
    print("    step %2d = " % (val), end='')
    print(arr)
```

```
if __name__ == "__main__":
    data = [5, 3, 8, 4, 9, 1, 6, 2, 7]
    print("Original :  ", data)
    insertion_sort(data)
    print("Selection : ", data)

# 출력
# Original :  [5, 3, 8, 4, 9, 1, 6, 2, 7]
#     step  1 = [3, 5, 8, 4, 9, 1, 6, 2, 7]
#     step  2 = [3, 5, 8, 4, 9, 1, 6, 2, 7]
#     step  3 = [3, 4, 5, 8, 9, 1, 6, 2, 7]
#     step  4 = [3, 4, 5, 8, 9, 1, 6, 2, 7]
#     step  5 = [1, 3, 4, 5, 8, 9, 6, 2, 7]
#     step  6 = [1, 3, 4, 5, 6, 8, 9, 2, 7]
#     step  7 = [1, 2, 3, 4, 5, 6, 8, 9, 7]
#     step  8 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
# Selection :  [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
# 버블정렬

# 인접한 2개의 레코드를 비교하여 순서대로 교환
# 끝으로 이동한 레코도를 제외하고 다시 스캔 반복

# 이동연산은 비교연산 보다 더 많은 시간이 소요됨

def bubble_sort(A):
    n = len(A)
    for i in range(n-1, 0, -1):
        bChanged = False # 병렬의 중단을 나타내는 기준값
        for j in range(i): # step1은 i가 8이라 j가 0, ... , 7이다. step2은 i가 7이라 j가 0, ... , 6이다.
            if (A[j] > A[j+1]) : # 연속된 값들 중 초반에 있는 수가 더 크면
                A[j], A[j+1] = A[j+1], A[j] # 스왑
                bChanged = True # 기준값에 T값 저장
        if not bChanged: # False면 실행, 즉 한줄의 반복이 되는동안 스왑이 한번도 진행되지 않으면
            break;
        printStep(A, n-i);

def printStep(arr, val):
    print("    step %2d = " % (val), end='')
    print(arr)

if __name__ == "__main__":
    data = [5, 3, 8, 4, 9, 1, 6, 2, 7]
    print("Original :  ", data)
    bubble_sort(data)
    print("Selection : ", data)

# 출력
# Original :   [5, 3, 8, 4, 9, 1, 6, 2, 7]
#     step  1 = [3, 5, 4, 8, 1, 6, 2, 7, 9]
#     step  2 = [3, 4, 5, 1, 6, 2, 7, 8, 9]
#     step  3 = [3, 4, 1, 5, 2, 6, 7, 8, 9]
#     step  4 = [3, 1, 4, 2, 5, 6, 7, 8, 9]
#     step  5 = [1, 3, 2, 4, 5, 6, 7, 8, 9]
#     step  6 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
# Selection :  [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
# 비교연산

# 3장에서 구현한 집합 자료구조 수정하기, 집합의 원소들을 항상 정렬된 순으로 저장
# 삽입 연산은 더 복잡해 짐
# 집합의 비교나 합집합, 차집합, 교집합 -> 효율적 구현 가능

# 집합에 원소 삽입, 같은집합찾기, 합집합, 차집합, 교집합에 복잡도 비교
# insert(e)            정렬되지않음: O(n)    /  정렬됨: O(n)
# __eq__(setB)         정렬되지않음: O(n^2) /  정렬됨: O(n)
# union(setB)          정렬되지않음: O(n^2) /  정렬됨: O(n)
# intersect(setB)    정렬되지않음: O(n^2) /  정렬됨: O(n)
# difference(setB)   정렬되지않음: O(n^2) /  정렬됨: O(n)

class Sort:

    def insert(self, elem): # 정렬된 상태를 유지하면서 elem을 삽입

        if elem in self.items: # 삽입하려는 값이 이미 집합에 있으면
            return # 삽입 중단

        for idx in range(len(self.items)): # 0부터 집합의 크기-1까지의 수를 대입
            if elem < self.items[idx]: # 왼쪽부터(작은값) 비교하다가 삽입하려는 값보다 큰 값이 나오면
                self.items.insert(idx, elem) # 그 위치에 삽입하려는 값을 대입한다.
                return
        self.items.append(elem) # 집합에 삽입하려는 값보다 큰 값이 없다면 맨 오른쪽에 추가한다.

    def __eq__(self, set):
        if self.size() != setB.size(): # 두 집합의 크기가 다르면 같지 않다.
            return False
        for idx in range(len(self.items)): # 두 집합의 크기만큼 반복을 한다.
            if self.items[idx] != setB.items[idx]: # 각 인덱스 위치에 값들을 비교
                return False
        return True

    def union(self. setB):
        newSet = Set() # 두 집합의 합집합을 저장하는 공간
        a = 0
        b = 0
        # 비교하는 집합의 인덱스가 두 집합의 길이가 보다 작을때만 반복
        while a < len(self.items) and b < len(setB.items):
            valueA = self.items[a]
            valueB = setB.items[b]
            if valueA < valueB:
                newSet.items.append(valueA)
                a += 1
            elif valueA > valueB:
                newSet.items.append(valueB)
                b += 1
            else:
                newSet.items.append(valueA)
                a += 1
                b += 1
        # 반복이 끝났을 때, self.items의 집합 남았으면, 즉 a의 인덱스가 오버가 안됐으면
        while a < len(self.items):
            newSet,items.append(self.items[a])
            a += 1
```

```
        # 반복이 끝났을 때, setB.items의 집합 남았으면, 즉 b의 인덱스가 오버가 안됐으면
        while b < len(setB.items):
            newSet.items.append(setB.items[b])
            b += 1
        return newSet

# 알고리즘
# step=0 newSet                      # step=5 newSet 0 1 2 3 4
# a=0 self.items 0 1 2 5             # a=3 self.items 0 1 2 5
# b=0 setB.items 1 2 3 4 5 6 7       # b=4 setB.items 1 2 3 4 5 6 7
#                                    #
# step=1 newSet 0                    # step=6 newSet 0 1 2 3 4 5
# a=1 self.items 0 1 2 5             # a=4(종료) self.items 0 1 2 5
# b=0 setB.items 1 2 3 4 5 6 7       # b=5 setB.items 1 2 3 4 5 6 7
#                                    #
# step=2 newSet 0 1                  # step=7 newSet 0 1 2 3 4 5 6
# a=2 self.items 0 1 2 5             # a=4(종료) self.items 0 1 2 5
# b=1 setB.items 1 2 3 4 5 6 7       # b=6 setB.items 1 2 3 4 5 6 7
#                                    #
# step=3 newSet 0 1 2                # step=8 newSet 0 1 2 3 4 5 6 7
# a=3 self.items 0 1 2 5             # a=4(종료) self.items 0 1 2 5
# b=2 setB.items 1 2 3 4 5 6 7       # b=7(종료) setB.items 1 2 3 4 5 6 7
#
# step=4 newSet 0 1 2 3
# a=3 self.items 0 1 2 5
# b=3 setB.items 1 2 3 4 5 6 7
```

```
# 탐색, 맵, 엔트리, 딕셔너리

# 탐색
# 테이블에서 원하는 탐색키를 가진 레코드를 찾는 작업

# 맵또는 딕셔너리
# 탐색을 위한 자료구조
# 엔트리 또는 키를 가진 레코드의 집합

# 맵 ADT
# search(key) : 탐색키(key)를 가진 레코드를 찾아 반환한다.
# insert(entry) : 주어진 entry를 앱에 삽입한다.
# delete(key) : 탐색키를 가진 레코드를 찾아 삭제한다.

# 엔트리
# 키 : 영어 단어와 같은 레코드를 구분할 수 있는 탐색키
# 값 : 단어의 의미와 같이 같이 탐색키와 관련된 값

# 앱을 구현하는 방법
# 리스트 사용 : 정렬 / 비정렬
# 이진 탐색 트리 사용 (9장)
# 해싱 구조 이용



# 순차탐색

# 정렬되지 않은 배열에 적용 가능
# 가장 간단하고 직접적인 탐색 방법
```

```
# 시간 복잡도 : O(n)

# A에서 key값을 low인덱스부터 high인덱스까지 찾아서 반환한다.
def sequential_search(A, key, low, high):
    for i in range(low, high+1):
        if A[i] == key:
            return i
    return None

if __name__ == "__main__":
    data = [5, 3, 8, 4, 9, 1, 6, 2, 7]
    print("Original :  ", data)
    idx = sequential_search(data, 1, 0, 9)
    print("sequential_search : ", idx+1)

'''
출력
Original :   [5, 3, 8, 4, 9, 1, 6, 2, 7]
sequential_search :  6
'''
```

```
# 이진탐색

# 정렬된 배열의 탐색에 적합
# 배열의 중앙에 있는 값을 비교해가면서 탐색의 범위를 절반씩 줄여가는 탐색법

# 시간 복잡도 : O(log n)
# 반복으로 구현가능

def binary_search(A, key, low, high):
    if(low <= high): # 항목들이 남아있다면 (종료조건)
        middle = (low + high) // 2 # 중간값을 저장, 정수 나눗셈 //에 주의 할 것
        if key == A[middle]:
            return middle # 탐색 성공
        elif (key < A[middle]): # 찾는 값이 중간값보다 작으면
            return binary_search(A, key, low, middle-1) # high 값을 수정하여 재귀
        else: # 찾는 값이 중간값보다 크면
            return binary_search(A, key, middle+1, high) # low 값을 수정하여 재귀
    return None # 탐색 실패

if __name__ == "__main__":
    data = [2, 26, 11, 13, 18, 20, 22, 27, 29, 30, 34, 38, 41, 42, 45, 47]
    print("Original : ", data)
    idx = binary_search(data, 20, 0, len(data))
    print("binary_search : ", idx+1)

'''
Original :  [2, 26, 11, 13, 18, 20, 22, 27, 29, 30, 34, 38, 41, 42, 45, 47]
binary_search :  6
'''

# 보간 탐색(InterPolation Search)
# 탐색키가 존재 할 위치를 예측하여 탐색
# 리스트를 불균등하게 분할하여 탐색

# 있다고만 하고 보간탐색 pass
```

```
# 해싱 : 키 값에 대한 산술적 연산에 의해 테이블의 주소를 계산
# 해시 테이블 : 키 값의 연산에 의해 직접 접근이 가능한 구조, 해시 함수가 키 값을 생성할때 참조하는 테이블
# 버킷 : 하나의 주소를 갖는 파일의 한 구역
# 슬롯 : 한 개의 레코드를 저장 할 수 있는 공간, 한 버킷 안에 여러 개의 슬롯이 있다.
# 충돌 : 서로 다른 키가 해시 함수에 의해 같은 주소로 계산되는 상황. 레코드는 버킷의 다음 슬롯 중 빈곳에 들어가게 된다.
# 동의어 : 충돌이 일어난 레코드의 집합. 키값이 같은 레코드의 집합으로, 동의어가 슬롯의 개수보다 많다면 오버플로우가 생긴다.
# 오버플로 : 한 홈 주소의 버킷 내에 더이상의 레코드를 저장할 슬롯이 없는 상태, 충돌이 슬롯 수보다 많이 발생하는 것
# 해시 함수 : 탐색키를 입력 받아 해시 주소 생성

# 선형 조사에 의한 오버플로 처리, 삽입 연산
# key     45 27 88  9 71 60 46 38 24
# h(key)  6  1 10  9  6  8  7 12 11

# .. 27 .. .. .. .. 45 .. .. 09 88 .. ..
# 00 01 02 03 04 05 06 07 08 09 10 11 12 // 45, 27, 88, 9 대입

# .. 27 .. .. .. .. 45 71 .. 09 88 .. ..
# 00 01 02 03 04 05 06 07 08 09 10 11 12 // 71 대입시 충돌 발생, 다음 슬롯인 07번에 저장

# .. 27 .. .. .. .. 45 71 60 09 88 46 ..
# 00 01 02 03 04 05 06 07 08 09 10 11 12 // 60 대입 후 46 대입시 충돌 발생, 다음 슬롯인 11에 저장

# 24 27 .. .. .. .. 45 71 60 09 88 46 38
# 00 01 02 03 04 05 06 07 08 09 10 11 12 // 38 대입 후 24 대입시 충돌 발생, 다음 슬롯인 00에 저장

# 선형 조사의 탐색 연산
# 순서대로 값을 찾는다.

# 선형 조사의 삭제 연산
# 빈 버킷을 만나면 연산을 정지하므로 앞서서 값이 있다가 삭제된 버킷은 원래 빈 버킷과 다르게 분류하여 표시한다.

# 선형조사 군집화 완화 방법
# 이차 조사법
# 이중 해시법 - 재행싱 방법으로 충동이 발생하면, 다른 해시 함수를 이용해 다음 위치 계산

# 체이닝에 의한 오버플로 처리
# 하나의 버킷에 여러 개의 레코드를 저장 할 수 있도록 하는 방법
# 예) h(k)) = k%7 라는 해시 함수를 이용해 0~7인덱스를 가지는 버킷에 값들을 입력
# 8, 1, 9, 6, 13을 대입할 때,
# 8과 1은 나머지가 1로 같으므로 충돌 발생 -> 같은 1슬롯에 새로운 노드를 생성하여 순서대로(8 -> 1) 저장
# 6과 13은 나머지가 6로 같으므로 충돌 발생 -> 같은 6슬롯에 새로운 노드를 생성하여 순서대로(6 -> 13) 저장

# 좋은 해시 함수의 조건
# 충돌이 적어야 한다.
# 함수 값이 케이블의 주소 영역 내에서 고르게 분포되어야 한다.
# 계신이 빨라야 한다.

# 제산 함수
# h(k) = k mod M
# 해시 테이블의 크기는 M은 소수(prime number) 선택

# 폴딩 함수 (16바이트 공간에 30바이트 변수를 대입하려고 할때 등 사용)
# 탐색키를 이용하여 이동폴딩, 경계폴딩을 진행할 수 있다.

# 해시함수
# 중간 제곱 함수 : 탐색키를 제곱한 다음, 중간의 몇 비트를 취해서 해시 주소 생성
```

비트 추출 함수 : 키를 이진수로 간주, 임의의 위치의 k개의 비트를 사용

탐색 방법들의 성능 비교
해싱의 적재 밀도 or 적재 비율 = (저장된 항목의 개수) / M(버킷의 개수)

탐색방법		탐색	삽입	삭제
순차탐색		O(n)	O(1)	O(n)
이진탐색		O(log n)	O(n)	O(n)
이진탐색트리	균형트리	O(log n)	O(log n)	O(log n)
	경사트리	O(n)	O(n)	O(n)
해싱	최선의 경우	O(1)	O(1)	O(1)
	최악의 경우	O(n)	O(n)	O(n)

맵의 응용 : 리스트를 이용한 순차탐색 맵
맵은 바이너리 이다.

나의 단어장
```
def sequential_search(A, key, low, high):
    for i in range(low, high+1):
        if A[i].key == key: # A는 딕셔너리라서 각 인덱션에 들어있는 key 값을 비교하려면 A[i].key로 표현해야 한다.
            return i
    return None

class Entry: # 입력된 key와 value를 딕셔너리로 저장시켜 대입해주는 class
    def __init__(self, key, value):
        self.key = key
        self.value = value

    def __str__(self): # 출력 지정자도 딕셔너리로 출력을 하도록 지정
        return str("%s:%s" % (self.key, self.value))

class SequentialMap:
    def __init__(self):
        self.table = []

    def insert(self, key, value):
        self.table.append(Entry(key, value)) # 각 요소에 딕셔너리로 key와 value를 대입한다.

    def search(self, key):
        pos = sequential_search(self.table, key, 0, len(self.table)-1)
        if pos is not None: # pos, 즉 key값을 찾으면
            return self.table[pos]
        else: # pos, 즉 key값을 찾지 못하면
            return None

    def delete(self, key):
        for i in range(len(self.table)):
            if self.table[i].key == key: # 각 요소에 key값이 일치하면
                self.table.pop(i)
                return

    def display(self, msg):
        print(msg)
        for i in range(len(self.table)):
            print("      ",self.table[i]) # for문으로 돌면서 첫번째 요소(key, value)부터 출력한다.
```

```
if __name__ == "__main__":
    map = SequentialMap()
    map.insert('data','자료')
    map.insert('structure','구조')
    map.insert('sequential search','선형 탐색')
    map.insert('game','게임')
    map.insert('data','자료')
    map.insert('binary search','이진 탐색')
    map.display("나의 단어장: ")

    print("탐색:game --> ", map.search('game'))
    print("탐색:over --> ", map.search('over'))
    print("탐색:data --> ", map.search('data'))

    map.delete('game')
    map.display("나의 단어장: ")

'''
나의 단어장:
      data:자료
      structure:구조
      sequential search:선형 탐색
      game:게임
      data:자료
      binary search:이진 탐색
탐색:game -->  game:게임
탐색:over -->  None
탐색:data -->  data:자료
나의 단어장:
      data:자료
      structure:구조
      sequential search:선형 탐색
      data:자료
      binary search:이진 탐색
'''
```

```
# 체이닝에 의한 오버플로 처리
# 하나의 버킷에 여러 개의 레코드를 저장 할 수 있도록 하는 방법
# 예) h(k)) = k%7 라는 해시 함수를 이용해 0~7인덱스를 가지는 버킷에 값들을 입력
# 8, 1, 9, 6, 13을 대입할 때,
# 8과 1은 나머지가 1로 같으므로 충돌 발생 -> 같은 1버킷에 새로운 노드를 생성하여 순서대로(8 -> 1) 저장
# 6과 13은 나머지가 6로 같으므로 충돌 발생 -> 같은 6버킷에 새로운 노드를 생성하여 순서대로(6 -> 13) 저장

# 앱의 응용 : 채이닝을 이용한 해시 맵
# 나의 단어장

from SequentialMap import Entry

class Node:
    def __init__(self, data, link=None): # 생성자
        self.data = data
        self.link = link

    def __str__(self): # 출력지정자
        return str("%s:%s" % (self.key, self.value))

class HashChainMap:
    def __init__(self, M):
        self.table = [None]*M # 입력받은 버킷의 수(M)만큼 공간 생성
        self.M = M

    def hashFn(self, key):
        sum = 0
        for c in key:            # 문자열의 모든 문자에 대해
            sum = sum + ord(c) # 그 문자의 아스키 코드 값을 sum에 더함
        return sum % self.M     # M으로 나눠서 M크기의 버킷에 알맞은 배정 인덱스를 구한다.

    def insert(self, key, value):
        idx = self.hashFn(key) # 해당 key값이 어느 버킷에 해당되는지 해시함수를 통해 찾는다.
        # 찾은 버킷에 노드 형식으로 데이터부분에 딕셔너리로 묶은 key와 value를, 링크 부분에 버킷을 넣는다.
        self.table[idx] = Node(Entry(key, value), self.table[idx])
        # entry = Entry(key, value) 위에 식을 아래와 같이 여러 줄로 표현 가능
        # node = Node(entry)
        # node.link = self.table[idx]
        # self.table[idx] = node

    def delete(self, key):
        idx = self.hashFn(key) # key가 해당되는 버킷 인덱스 반환받기
        node = self.table[idx] # 해당 인덱스의 버킷 해드를 저장
        before = None
        while node is not None: # 노드가 빈 값일때까지 반복
            if node.data.key == key: # 해당 노드의 data부분의 key값이 찾는 key값과 같으면
                if before == None: # 버킷에 노드가 자기 자신 밖에 없으면
                    self.table[idx] = node.link
                else: # 버킷에 노드가 2개 이상 있는 경우
                    before.link = node.link
                return
            before = node # 이전에 링크에 현재 링크 노드를 저장
            node = node.link # 노드에 다음 노드 링크를 저장

    def search(self, key):
        idx = self.hashFn(key) # key가 해당되는 버킷 인덱스 반환받기
```

```
        node = self.table[idx] # 해당 인덱스의 버킷 해드를 저장
        while node is not None: # 노드가 비어 있을 때까지 반복
            if node.data.key == key: # 해당 노드의 data부분의 key값이 찾는 key값과 같으면
                return node.data # 값 반환
            node = node.link # 같지 않으면 다음 노드 링크를 현재 노드에 저장


    def display(self, msg):
        print(msg)
        for idx in range(len(self.table)):
            node = self.table[idx]
            if node is not None:
                print("[%2d] -> " % (idx) , end='' )
                while node is not None:
                    print(node.data, end=' ->' )
                    node = node.link
                print()


if __name__ == "__main__":

    map = HashChainMap(13)
    map.insert('data','자료')
    map.insert('structure','구조')
    map.insert('sequential search','선형 탐색')
    map.insert('game','게임')
    map.insert('binary search','이진 탐색')
    map.display("나의 단어장: ")

    print("탐색:game --> ", map.search('game'))
    print("탐색:over --> ", map.search('over'))
    print("탐색:data --> ", map.search('data'))

    map.delete('game')
    map.display("나의 단어장: ")

'''
[ 3] -> sequential search:선형 탐색 ->
[ 7] -> binary search:이진 탐색 ->game:게임 ->data:자료 ->
[ 8] -> structure:구조 ->
탐색:game -->  game:게임
탐색:over -->  None
탐색:data -->  data:자료
나의 단어장:
[ 3] -> sequential search:선형 탐색 ->
[ 7] -> binary search:이진 탐색 ->data:자료 ->
[ 8] -> structure:구조 ->
'''
```

```
# 맵의 응용 : 딕셔너리를 이용한 구현
# 딕셔너리의 초기형식이 맵이다.

d = {}
d['data'] = '자료'
d['structure'] = '구조'
d['sequential search'] = '선형 탐색'
d['game'] = '게임'
d['binary search'] = '이진 탐색'
print("나의 단어장: ")
print(d)

if d.get('game') :
    print("탐색:game --> ", d['game'])
if d.get('over') :
    print("탐색:over --> ", d['over'])
if d.get('data') :
    print("탐색:data --> ", d['data'])

d.pop('game') # 게임 삭제
print("나의 단어장: ")
print(d)

'''
나의 단어장:
{'data': '자료', 'structure': '구조', 'sequential search': '선형 탐색', 'game': '게임', 'binary search': '이진 탐색'}
탐색:game -->  게임
탐색:data -->  자료
나의 단어장:
{'data': '자료', 'structure': '구조', 'sequential search': '선형 탐색', 'binary search': '이진 탐색'}
'''
```