알고리즘과 자료구조 워크북

교과목명 : 알고리즘과 자료구조
차시명: 12차시 해시 테이블

◈ 담당교수: 신일훈 **(서울과학기술대학교)**

◉ 세부목차
- 해시 테이블 개념
- 충돌 및 충돌의 해결 방안 이해
- 선형 조사 구현

학습에 앞서

■ 학습개요

큐와 스택은 각각 FIFO, LIFO 등의 정책을 구현하기에 효율적인 자료구조이지만, 대규모의 데이터에 대한 검색을 수행하기에는 비효율적이다. 해시 테이블은 대규모의 데이터에 대한 검색을 수행할 때 탐색 공간의 크기를 줄여서 검색의 효율성을 높일 수 있는 자료구조이다. 본 강에서는 해시 테이블의 개념과 충돌 문제, 충돌을 해결할 수 있는 방안을 이해하고, 충돌 해결 방안 중 하나인 선형조사를 파이썬으로 구현한다.

■ 학습목표

1	해시 테이블의 개념을 이해한다.
2	충돌 및 충돌의 해결 방안을 이해한다.
3	충돌 해결 방안 중 선형 조사를 구현할 수 있다.

■ 주요용어

용어	해설
해시 테이블	해시 함수를 이용하여 검색 공간을 줄임으로써 검색의 효율성을 높이는 자료구조
충돌	해시 함수의 입력으로 사용되는 키(key)가 다름에도 불구하고 해시 함수의 출력이 동일한 경우이다. 해시 테이블의 동일한 인덱스로 매핑된다.
선형 조사	충돌이 발생하면 순차적으로 다음의 빈 인덱스를 탐색하여 저장하는 충돌 해결 기법이다.
이중 해싱	충돌이 발생하면, 2차 해시 함수를 적용하여 그 결과값만큼 떨어진 인덱스에 데이터를 저장함으로써 충돌을 해결하는 기법이다. (빈 인덱스를 발견할 때까지 반복함)
체이닝	해시 테이블의 각 인덱스에 여러 아이템을 저장할 수 있도록 함으로

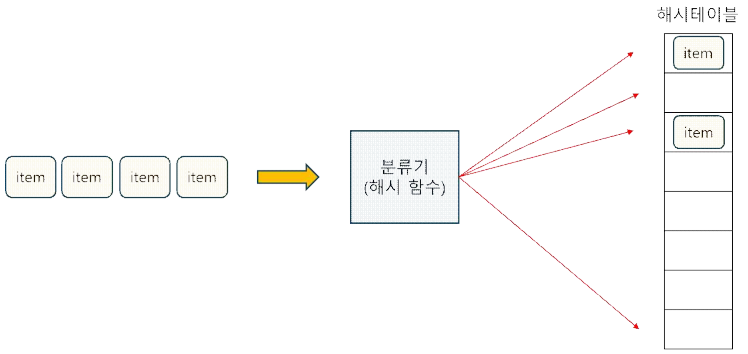
	써 충돌을 해결하는 기법이다. 연결 리스트를 사용하는 경우, 동일 인덱스로 매핑된 아이템들을 연결 리스트로 연결하여 관리한다.

학습하기

1. 해시 테이블 개념
   - 연결리스트, 큐, 스택 등의 선형 자료구조는 구현이 용이한 장점이 있음.
   - 특히 스택과 큐는 각각 LIFO, FIFO 등의 정책을 구현하기에 효율적인 자료구조임.
   - 반면, 탐색에는 적합하지 않은 한계가 있음.
   - 가령 N개의 아이템을 연결리스트나, 스택, 큐 등을 사용하여 저장한다면, 특정 아이템을 탐색 시에 최악의 경우, N개의 아이템을 모두 검사해야 함
   - 가령, 1만개의 전화번호를 큐의 형태로 저장한다면 특정 이름의 전화번호를 검색할 때, 별수없이 가장 앞쪽부터 뒤쪽까지 발견할 때까지 검사해야 함.
   - 따라서 저장된 아이템의 개수가 많다면 탐색 시간이 길어지는 단점이 있음.

   - 이를 해결하기 위해서는 탐색 시 탐색할 아이템의 개수(탐색공간의 크기)를 줄여야 함.
   - 가령 전화번호를 저장할 때 각 알파벳마다 별도의 큐를 사용한다면, 즉 26개의 큐를 사용하다면 탐색할 아이템의 개수가 감소함. 즉 이름이 Shin으로 시작한다면 S로 시작하는 큐만 검사하면 됨. 따라서 평균적으로 검사할 아이템의 개수가 1/26으로 감소함. 하지만 여전히 최악 시간복잡도는 O(N).

   - 해시 테이블은 해시 함수를 사용하여 탐색공간의 크기를 줄이는 아이디어이다.
   - 즉 아래 그림에서 보이듯이 해시 테이블은 각 아이템들을 저장하는 공간이다 (가령, 배열이나 리스트로 이해하면 됨)
   - 특정 아이템을 찾을 때 아이템의 key를 입력 인자로 하여 해시 함수를 호출하면 해시 테이블에서 아이템의 위치가 해시 함수의 결과값이 됨.
   - 따라서 아이템을 발견하기 위해 해시 테이블을 처음부터 끝까지 검사하지 않고, 해시 함수를 통해 아이템이 저장된 위치를 한번에 찾게 된다.



   - 따라서 이상적인 해시 함수는 가능한 아이템들을 해시 테이블 상에서 균등하게 분산해야 함.
   - 또한 해시 함수의 실행 시간이 길지 않아야 한다. (계산 오버헤드가 낮아야 한다.)
   - 간단한 해시 테이블에서 해시 함수로 많이 사용되는 것 중의 하나가 나머지를 구하는 % 연산자이다. % 연산자를 사용하여 key를 해시 테이블의 크기(L)로 나누면 결과값이 0 ~ L-1이 되어 이 값을 아이템을 저장할 해시 테이블의 인덱스로 사용한다.
   - % 연산자는 실행 시간이 짧은 장점이 있지만 아이템들을 균등하게 분산하지는 못할 수 있다. 균등 분산을 위해서는 다른 해시 함수를 사용하는 것을 고려할 수 있다.

2. 충돌 및 해결 기법

가. 충돌

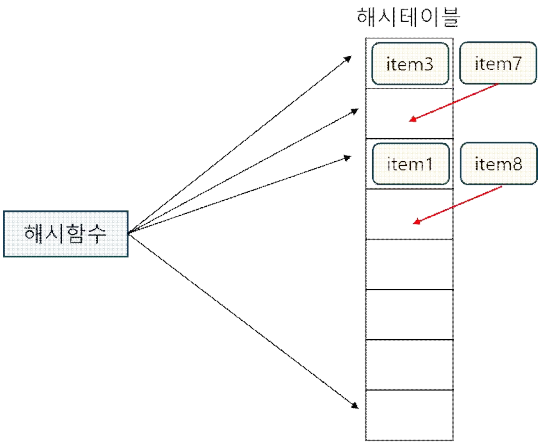
- key가 다름에도 불구하고 해시 함수의 출력이 동일할 때 충돌이라 지칭함.
- 즉, key가 다름에도 불구하고 동일한 인덱스로 매핑됨.
- 가령, %를 해시 함수로 사용하고, 해시 테이블의 크기가 13이라면, 7, 20, 33, … 등의 key는 모두 해시 함수 출력이 동일하므로 충돌이 발생하게 됨.

나. 충돌 해결 기법

- 선형 조사
- 이중 해싱
- 체이닝

다. 선형 조사(linear probing)

- 충돌이 발생하면, 순차적으로 다음의 빈 인덱스를 탐색하여 저장하는 방법이다.
- 아래 그림에서 item3을 저장할 때는 타겟 인덱스가 비어 있으므로 해당 인덱스에 item3이 저장됨.
- 하지만 item7을 저장할 때는 타겟 인덱스가 item3와 동일한 인덱스로 매핑되어, 타겟 인덱스가 비어 있지 않은 충돌이 발생함.
- 이 경우에 item7은 다음의 빈 인덱스에 저장됨.
- item1, item8도 마찬가지이다.
- item1은 타겟 인덱스가 비어 있으므로 충돌이 발생하지 않아 해당 인덱스에 저장된다.
- 하지만 item8은 타겟 인덱스가 item1과 동일하여 충돌이 발생하므로 다음의 빈 인덱스에 저장된다.



- 선형 조사에서 검색 연산은 다음과 같이 처리된다.

1. 해시함수를 통해 타겟 인덱스를 발견한다
2. 타겟 인덱스의 아이템이 찾는 아이템인지를 검사하여 발견하면 이를 반환하고 종료한다.
3. 찾는 아이템이 아니면 그 다음 인덱스를 타겟 인덱스로 수정하여 2번을 반복한다.
4. 2-3 작업을 반복하다, 타겟 인덱스가 비어 있거나 타겟 인덱스가 해시 함수의 출력과 동일하면, 즉 가장 처음 인덱스에 도달하면 찾는 아이템이 없다는 의미이므로 실패를 반환하고 종료한다.

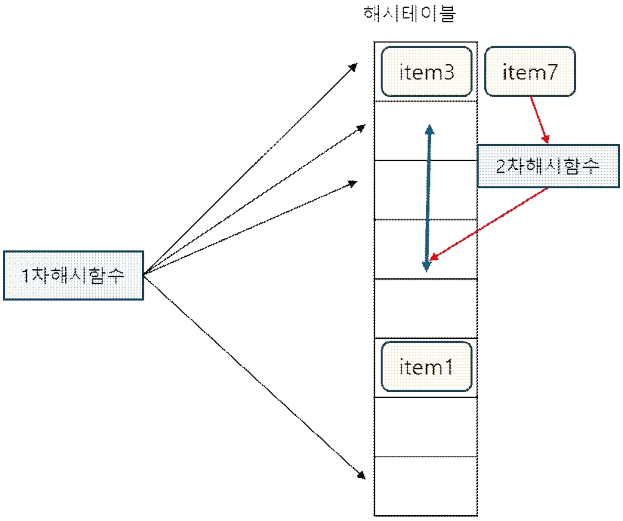
- 선형 조사는 clustering (군집화)을 유발할 수 있으며 그로 인해 검색 연산의 성능이 저하될 수 있다.

라. 이중 해싱(double hashing)

- 충돌이 발생하면, 2차 해시 함수를 적용하여 해당 distance 만큼 떨어진 위치를 다음 자리로 결정
- 아래 그림에서 item3을 저장할 때는 타겟 인덱스가 비어 있으므로 해당 인덱스에 item3이 저장됨.
- 하지만 item7을 저장할 때는 타겟 인덱스가 item3와 동일한 인덱스로 매핑되어, 타겟 인덱스가 비어

있지 않은 충돌이 발생함.

- 이 경우에 item7의 타겟 인덱스를 결정하기 위해 2차 해시 함수를 적용함.
- 2차 해시 함수의 결과만큼 떨어진 인덱스를 다음의 타겟 인덱스로 정함.
- 타겟 인덱스가 비어 있다면 해당 인덱스에 item7을 저장하고, 그렇지 않다면 빈 타겟 인덱스를 만날 때까지 2차 해시 함수를 반복하여 적용함.
- 만약 빈 인덱스를 만나지 못하고, 가장 처음의 타겟 인덱스로 돌아가면, 저장할 곳이 없으므로 저장은 실패로 끝난다.



- 검색 연산은 다음과 같이 처리된다.

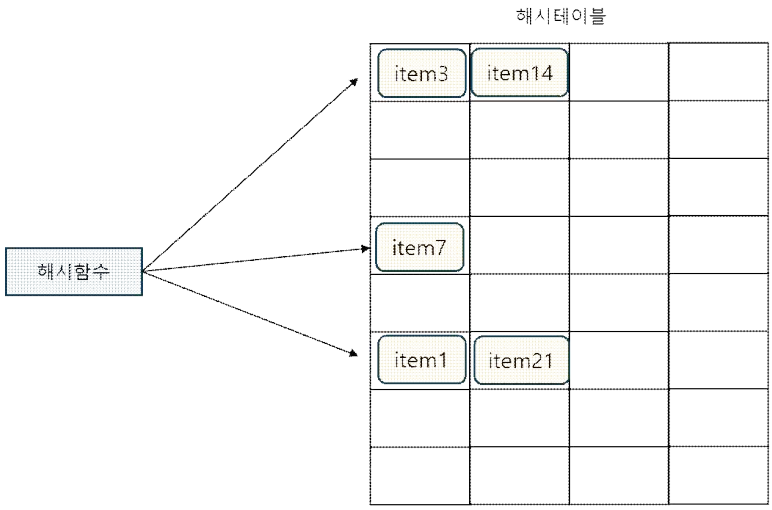
1. 1차 해시 함수를 통해 타겟 인덱스 발견
2. 타겟 인덱스의 원소가 찾는 아이템인지 체크하여 찾는 아이템이면 탐색을 종료하고 발견한 아이템을 반환한다.
3. 아니면 2차 해시함수를 적용하여 해당 인덱스를 타겟 인덱스로 설정하고 2번을 반복한다.
4. 만약 타겟 인덱스가 비어 있거나 처음 인덱스와 동일하다면, 찾는 아이템이 없다는 의미이므로 실패를 반환한다.

마. 체이닝(chaining)

- 해시 테이블의 각 인덱스의 하나의 아이템이 아닌 여러 아이템을 저장하도록 함으로써 충돌에 대처함.

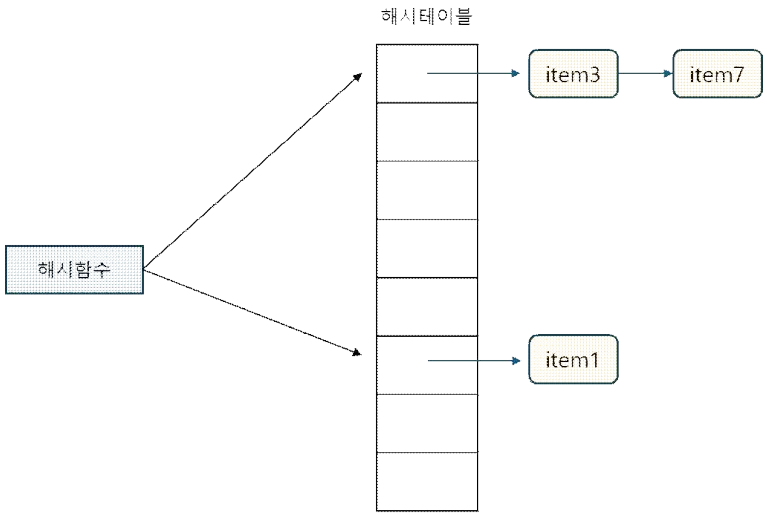
▶ 방법1

- 2차원 배열을 활용하여 여러 아이템을 저장함
- 저장 가능한 아이템의 최대 개수가 배열의 column 값에 의해 결정됨



▶ 방법2

- 1차원 배열을 사용하되, 배열의 각 원소를 연결 리스트의 헤더로 사용.
- 동일한 원소로 해시된 아이템들을 연결 리스트로 연결하여 저장.
- 각 인덱스에 저장할 수 있는 원소의 개수에 제한이 없음.



3. 선형조사 구현

가. 클래스 LinearProbing 정의

- 선형조사 방식의 해시 테이블.
- 멤버 변수
  - tablesize : 해시 테이블의 크기
  - table: 해시 테이블. 리스트이며, 각 원소는 (key, value)의 튜플

- 메서드
  - 생성자
  - hash()
  - add()
  - search()
  - remove()
  - print()

나. 클래스 LinearProbing 생성자 코드

```
1. class LinearProbing:
2.         def __init__(self, size):          # 해시 테이블을 빈 상태로 초기화해야 함
3.                 self.tablesize = size
4.                 self.table = [(None, None)] * size
```

- 2번 라인에서 해시 테이블의 크기를 인자로 전달받는다.
- 4번 라인에서 해시 테이블의 모든 원소를 (None, None)으로 초기화한다. 즉 key, value 모두 지정되지 않은 비어 있는 상태로 초기화한다.

다. 클래스 LinearProbing의 hash() 메서드 코드

- 타겟 인덱스를 결정하는 해시 함수로 % 연산자를 사용한다.

```
        def hash(self, key):
                return key % self.tablesize
```

라. 클래스 LinearProbing의 add() 메서드 코드

- 새로운 아이템을 해시 테이블에 추가 저장하는 메서드이다.

```
1.      def add(self, key, value):
2.          initial_position = self.hash(key)
3.          position = initial_position
4.
5.          while True:
6.              (fkey, fvalue) = self.table[position]
7.              if fvalue == None:              # (1) 빈 버킷 발견 & 추가
8.                  self.table[position] = (key, value)
9.                  return  True
10.             elif fkey == key:               # (2) 동일 아이템 발견 & 값 수정
11.                 self.table[position] = (key, value)
12.                 return  True
13.             position = (position + 1) % self.tablesize
14.             if position == initial_position:            # 추가할 빈 버킷 없음
15.                 return False
```

- 2번 라인에서 initial_position은 해시 함수의 출력인 원래의 타겟 인덱스를 저장한다.
- 아이템을 저장할 빈 인덱스를 발견하거나 추가 작업이 실패할 때까지 while 문을 반복함
- position은 현재의 타겟 인덱스를 의미한다.
- 6번 라인에서 현재의 타겟 인덱스에 저장된 fkey, fvalue를 구한다.
- 7번 라인에서 만약 fvalue가 None이면 해당 인덱스가 비어 있다는 의미이므로 현재의 인덱스에 key와 value를 저장하고 종료한다.
- 10번 라인에서 만약 fvalue가 None이 아니고 저장되어 있는 fkey와 추가하려고 하는 key가 동일하면 fvalue를 value로 바꾸고 종료한다.
- 그 외의 경우는 충돌이 발생한 경우이므로 타겟 인덱스를 다음 인덱스로 설정한다. (13번 라인)
- 이때 새로운 타겟 인덱스가 해시 함수의 출력과 동일하면, 추가할 빈 인덱스가 없다는 의미이므로 실패를 반납하고 종료한다.

마. 클래스 LinearProbing의 print() 메서드 코드

- 해시 테이블에 저장된 모든 아이템들을 출력하는 메서드이다.

```
1.      def print(self):
2.          i = 0
3.          for tuple in self.table :
4.              print(i, tuple)
5.              i += 1
6.          print()
```

- 4번 라인에서 for문을 반복하며 해시 테이블의 모든 원소를 인덱스와 함께 출력한다.
- 6번 라인에서 모든 원소를 출력한 후, 행을 바꾼다.

바. 클래스 LinearProbing의 search() 메서드 코드

- 해시 테이블에서 특정 key에 해당하는 아이템을 찾는 메서드이다.

```
1.          def search(self, key):
2.                  initial_position = self.hash(key)
3.                  position = initial_position
4.
5.                  while True:
6.                          (fkey, fvalue) = self.table[position]
7.                          if fkey == key:
8.                                  return fvalue
9.                          elif fkey == None:
10.                                 return None
11.
12.                         position = (position + 1) % self.tablesize
13.                         if position == initial_position:
14.                                 return None
```

- 2번 라인에서 initial_position은 해시 함수의 출력인 원래의 타겟 인덱스를 저장한다.
- while 문을 반복하며 검색 작업을 수행한다 (5번 라인)
- position은 현재의 타겟 인덱스를 의미한다.
- 6번 라인에서 현재의 타겟 인덱스에 저장된 fkey, fvalue를 구한다.
- 7번 라인에서 저장되어 있는 fkey와 key가 동일하면 아이템을 발견했으므로 fvalue를 반환하고 종료한다.
- 9번 라인에서 fkey가 None이면 해당 인덱스가 비어 있다는 의미이므로 찾는 아이템이 존재하지 않는다는 뜻이다. 따라서 실패의 의미로 None을 반환한다.
- 그 외의 경우는 충돌이 발생했을 수 있으므로 타겟 인덱스를 다음 인덱스로 설정한다. (12번 라인)
- 이때 새로운 타겟 인덱스가 해시 함수의 출력과 동일하면, 모든 가능한 후보 인덱스들을 검사했다는 의미이므로 None을 반납하고 종료한다.

사. add(), print() 테스트 코드 및 실행 결과

```
1. if __name__ == '__main__':
2.          t = LinearProbing(7)
3.          t.add(7, 'grape')
4.          t.add(1, 'apple')
5.          t.add(2, 'banana')
6.          t.print()
7.          t.add(15, 'orange')
8.          t.print()
```

- 2번 라인에서 크기가 7인 빈 해시 테이블을 생성한다.
- key가 7, 1, 2인 아이템들을 각각 저장한다. (충돌이 발생하지 않으므로 0, 1, 2번 인덱스에 각각 저장된다.)
- 이후, key가 15인 아이템을 저장하는데 타겟 인덱스가 1로 충돌이 발생한다. 다음 빈 인덱스가 3이므로 orange는 3에 저장되어야 한다.
- 다음은 이를 실행한 결과 화면이다. 출력 결과가 예상과 동일함을 알 수 있다.

```
0 (7, 'grape')
1 (1, 'apple')
2 (2, 'banana')
3 (None, None)
4 (None, None)
5 (None, None)
6 (None, None)

0 (7, 'grape')
1 (1, 'apple')
2 (2, 'banana')
3 (15, 'orange')
4 (None, None)
5 (None, None)
6 (None, None)
```

아. search() 테스트 코드 및 실행 결과

```
1. if __name__ == '__main__':
2.         t = LinearProbing(7)
3.         t.add(7, 'grape')
4.         t.add(1, 'apple')
5.         t.add(2, 'banana')
6.         t.add(15, 'orange')
7.         print('1의 data = ', t.search(1))
8.         print('15의 data = ', t.search(15))
9.         print('11의 data = ', t.search(11))
```

- 2번 라인에서 크기가 7인 빈 해시 테이블을 생성한다.
- key가 7, 1, 2, 15인 아이템들을 각각 저장한다. 각각, 0, 1, 2, 3번 인덱스에 저장된다.
- 이후, key가 1인 아이템을 검색하면 apple이 발견되어야 한다.
- key가 15인 아이템을 검색하면 orange가 발견되어야 한다.
- key가 11인 아이템을 검색하면 없으므로 None이 출력되어야 한다.
- 다음은 이를 실행한 결과 화면이다. 출력 결과가 예상과 동일함을 알 수 있다.

```
1의 data =  apple
15의 data =  orange
11의 data =  None
```

연습문제

1. 해시 테이블을 사용할 때 충돌의 의미에 대해 설명하시오.

정답 : 해시 함수의 입력으로 사용되는 키(key)가 다름에도 불구하고 해시 함수의 출력이 동일한 경우이다. 이미 데이터가 저장되어 있는 해시 테이블의 인덱스로 매핑되므로 새로운 데이터를 해당 인덱스에 저장할 수 없다.

해설 : 정답 참조

2. 충돌 해결 기법 중 선형 조사에 대해 설명하시오.

정답 : 충돌이 발생하면 순차적으로 다음의 빈 인덱스를 탐색하여 해당 인덱스에 새로운 데이

터를 저장하는 충돌 해결 기법이다.

해설 : 정답 참조

3. 대량의 전화번호를 저장할 때 큐를 사용하여 저장하는 방식 대비하여 해시 테이블을 사용하여 저장할 때의 장점에 대해 설명하시오.

정답 : 큐를 사용하면, 특정 이름의 전화번호를 검색할 시에, 최악의 경우 모든 데이터를 전부 검색해야 한다. 하지만 해시 테이블을 사용하면 해시 함수를 활용하여 검색해야 하는 대상 데이터의 개수를 줄일 수 있으므로 검색의 효율성이 높은 장점이 있다.

해설 : 정답 참조

정리하기

1. 해시 테이블은 검색의 효율성을 높이기 위한 자료구조로서, 해시 함수를 활용하여 검색 공간의 크기를 줄인다.
2. 충돌 문제의 개념 및 해결 방안
3. 선형 조사의 구현

참고자료

- 파이썬과 함께하는 자료구조의 이해, 양성봉, 2021, 생능출판

다음 차시 예고

- 트리에 대해 학습한다.