**자료구조 04**

****

**중앙대학교**

**자료구조 05 분반**

**20165079 김영빈**

**목차**

과제

[**1.** **사전 만들기** 3](#_Toc71428128)

[**2.** **연결 리스트 응용하기** 11](#_Toc71428129)

# **사전 만들기**

* 1. **클래스와 메서드 정의**

class Node:

**def** \_\_init\_\_(*self*, **data**, **link=**None):

*self*.data **=** data

*self*.link **=** link

class LinkedList:

**def** \_\_init\_\_(*self*):

*#더미 노드를 사용하기 위해서 head 변수에 None값이 저장된 Node를 할당했다.*

*self*.\_\_head **=** Node(None)

*self*.\_\_count **=** 0

*#더 빠른 search를 위한 메서드*

*self*.alphabet **=** {}

*self*.\_\_english **=** "abcdefghijklmnopqrstuvwxyz"

*# 구현한 더 빠른 Search 방법*

**def** quick\_search(*self*, **question**):

        in\_dict **=** False

*#사전에 질의하는 단어의 맨 앞글자를 확인한다.*

        alphabet **=** *self*.\_\_english.index(question[0])

        """

        만약 해당 alphabet이 alphabet 인덱스에 존재하지 않는다면

        사전에 존재하지 않는다고 판단하고 in\_dict를 false로 return 한다.

        이때 alphabet이 self.alphabet에 key 값으로 존재하는지의 여부는 파이썬 딕셔너리의 특성상 O(1)이다.

        """

**if** alphabet **not** **in** *self*.alphabet:

**return** in\_dict, 0

**else**:

*#만약 사전에 해당 알파벳으로 시작하는 인덱스가 존재한다면 해당 인덱스의 시작 노드부터 비교를 한다.*

            cur **=** *self*.alphabet[alphabet]

            search\_time **=** time.time()

**while** cur **!=** None **and** question **>=** cur.data[0]:

**if** question **==** cur.data[0]:

                    print(cur.data[1])

                    in\_dict **=** True

**break**

**else**:

                    cur **=** cur.link

            search\_time **=** time.time() **-** search\_time

**return** in\_dict, search\_time

**def** search(*self*, **question**):

        in\_dict **=** False

*#무조건 맨 앞의 head 노드에서부터 search를 진행한다.*

        cur **=** *self*.\_\_head

        search\_time **=** time.time()

**while** cur.link **!=** None **and** question **>=** cur.link.data[0]:

**if** question **==** cur.link.data[0]:

                print(cur.link.data[1])

                in\_dict **=** True

**break**

**else**:

                cur **=** cur.link

        search\_time **=** time.time() **-** search\_time

**return** in\_dict, search\_time

**def** insert(*self*, **question**):

        print("찾을 수 없는 단어입니다. 뜻을 추가하세요(추가하지 않으려면 공백)")

        meaning **=** input("> ")

**if** meaning **==** "":

            print("추가하지 않습니다.")

**else**:

*#적절한 단어와 meaning이 들어온다면 initialize 메서드를 호출하여 적절한 위치를 찾고 사전에 추가되게 한다.*

*self*.initialize([question, meaning])

            print("%s %s 가 추가되었습니다.(총 %d개 단어)" **%** (question, meaning, len(*self*)))

**def** initialize(*self*, **elem**):

        elem[0] **=** elem[0].lower()

        before **=** *self*.\_\_head

**while** before.link **!=** None:

**if** before.link.data[0] **>** elem[0]:

**break**

            before **=** before.link

        before.link **=** Node(elem, before.link)

        """

        더 빠른 Search를 위한 개선 사항이다.

        이미 Node는 추가되어 있는 상태이지만, 나중에 Search를 더 빠르게 해주기 위해서 선언한 self.alphabet을

        필요하다면 Update 시켜줘야 sel.alphabet이 가장 최신의 상태를 유지할 수 있다.

        따라서 self.alphabet에 해당 알파벳으로 시작하는 인덱스가 있는지를 먼저 확인하고 존재하지 않는다면

        새로운 key value 쌍을 등록한다.

        만약 존재한다면 두 글자를 비교하여 업데이트 시킬 필요가 있다면 self.alphabet을 업데이트 시킨다.

        """

        alphabet **=** *self*.\_\_english.index(elem[0][0])

**if** alphabet **not** **in** *self*.alphabet:

*self*.alphabet[alphabet] **=** before.link

**else**:

**if** *self*.alphabet[alphabet].data[0] **>** before.link.data[0]:

*self*.alphabet[alphabet] **=** before.link

*self*.\_\_count **+=** 1

**def** display(*self*):

        node **=** *self*.\_\_head

**while** node.link **!=** None:

            print(node.link.data)

            node **=** node.link

**def** \_\_len\_\_(*self*):

**return** *self*.\_\_count

LinkedList 클래스의 head는 더미 노드를 가리키게 하여 이후에 구현할 메서드를 보다 유연하게 구현할 수 있도록 했다. 또한 얼마나 많은 Node들이 연결되어 있는지를 저장할 count 변수도 선언했다. Alphabet과 english는 기존의 search를 개선한 quick\_search를 위한 변수로 1.3에서 자세하게 다루었다. Node 클래스에서 다음 Node를 가리킬 때 link라는 변수를 사용한 이유는 Python에서 next가 예약어이기 때문이다.

Search는 입력 받은 단어의 의미를 출력하는 메서드이고 quick\_search는 이 search를 더 빠른 방법으로 개선한 메서드이다. 기본적인 동작 방식은 기준점이 되는 Node와 입력 받은 단어를 비교하여 존재 여부를 파악한다. 단어가 존재한다면 단어의 뜻이 출력되고 존재하지 않는다면 false값이 반환된다. Quick\_search와 search의 동작 차이는 1.3에서 더 자세하게 다루었다.

Insert는 main함수에서 in\_dict가 False라면 동작하도록 했다. 해당 메서드는 사용자에게 단어의 의미를 입력 받아 LinkedList에 적절한 곳에 배치하기 위해 구현한 메서드이다. 사용자가 valid한 값을 입력한 경우에만 initialize 메서드를 실행시킨다.

Initialize 메서드는 LinkedList를 순회하면서 파라미터로 전달받은 단어의 정보를 토대로 새로운 Node의 위치를 정하여 Link해주는 역할을 한다. Quick\_search를 구현하면서 initialize 역시 추가적인 구현을 했는데 이 부분도 1.3에서 더 자세하게 다루었다.

마지막으로 display는 LinkedList에 있는 모든 Node들을 출력하도록 구현하였고, \_\_len\_\_을 정의하면서 LinkedList의 count수를 반환하도록 했다.

|  |  |
| --- | --- |
| **클래스 or 메서드** | **내용** |
| Node | Data를 저장할 data와 다음 Node를 가리킬 link로 구성되어 있다. |
| LinkedList | LinkedList의 첫 시작을 가리키는 head 와 얼마나 많은 Node들이 연결되어 있는지를 표시하는 count 변수를 선언했다. 이후에 quick\_search에서 사용될 alphabet dictionary와 english 변수도 함께 선언했다. |
| quick\_search | 보다 빠른 검색을 위해 Search를 개선한 메서드이다. 단어를 받아서 해당 단어의 존재 유무를 판단하고 존재한다면 단어의 뜻을 출력한다. |
| search | 단어를 받아서 해당 단어의 존재 유무를 판단하고 존재한다면 단어의 뜻을 출력한다. |
| insert | 단어를 전달받고 그 뜻을 입력 받는다. 뜻이 Valid하게 입력되었다면 initialize 메서드를 실행시켜 적절한 위치에 단어를 연결하도록 한다. |
| initialize | 파라미터로 전달받는 단어의 정보를 적절한 위치에 삽입하여 연결한다. |
| display | 연결 리스트를 출력한다. |
| len | 연결 리스트에 존재하는 정보의 전체 수를 반환한다. |

* 1. **실행**

*# 사전의 역할을 할 randdict 선언 후 LinkedList인스턴스를 할당했다.*

randdict**=** LinkedList()

**with** open('randdict\_utf8.TXT', 'r', **encoding=**"utf-8") **as** f:

**for** line **in** f:

        line **=** line.strip().split(":")

        line **=** list(map(lambda **x**: x.strip(), line))

*#형식에 맞지 않으면 randdict에 추가하지 않는다.*

**if** line[1] **==** "":

**continue**

**else**:

            randdict.initialize(line)

**while** True:

    question **=** input(">> ")

*#일반적인 search*

    in\_dict, randdict\_time **=** randdict.search(question)

*#개선한 quick\_search*

    in\_dict, randdict\_quick\_search\_time **=** randdict.quick\_search(question)

*#만약 검색 내용이 사전에 없다면 새로 추가함*

**if** **not** in\_dict:

        randdict.insert(question)

*#search와 quick\_search 2개의 시간 비교를 위한 함수*

    print("randdict search time:       %.10f" **%** randdict\_time)

    print("randdict quick search time: %.10f" **%** randdict\_quick\_search\_time, **end=**"\n\n")

제공된 TXT파일의 정보를 사용하기 위해서 randdict\_utf8.TXT 파일을 읽기 모드로 open한 뒤에, 각 line에서 단어와 의미를 분리했다. 이때 뜻이 형식에 맞지 않는 단어에 대해서는 randdict에 추가하지 않았다. Initialize가 끝나면 사용자에게 단어를 입력 받아 해당 단어가 존재한다면 뜻을 출력했고, 존재하지 않는다면 내용을 입력 받아 insert를 실행하게 했다. 이때 사용자가 아무것도 입력하지 않았다면 insert 메서드 내부에서 단어를 추가하지 않도록 구현했으며 적절한 내용이 입력되었다면 initialize 메서드를 통해 적절한 위치에 새로운 Node가 연결되도록 했다. 마지막으로 search와 quick\_search의 성능 차이를 실행 시간을 통해 증명하기 위해 출력하면서 동작이 마무리된다. 이때 main 함수가 아닌 search와 quick\_search 함수 내부에서 time을 측정하여 돌려받는 이유는 main에서 search와 quick\_search를 호출하는 시간을 전체 시간에서 제외하여 보다 정확한 시간을 측정하기 위해서이다.

* 1. **검색 개선하기**

class LinkedList:

**def** \_\_init\_\_(*self*):

*#더미 노드를 사용하기 위해서 head 변수에 None값이 저장된 Node를 할당했다.*

*self*.\_\_head **=** Node(None)

*self*.\_\_count **=** 0

*#더 빠른 search를 위한 메서드*

*self*.alphabet **=** {}

*self*.\_\_english **=** "abcdefghijklmnopqrstuvwxyz"

검색을 개선하기 위해서 LinkedList의 인스턴스 변수에 alphabet과 english 변수를 추가했다. English 변수는 입력으로 들어오는 단어가 몇 번째 알파벳인지 얻는데 필요한 변수이다. alphabet dictionary는 각 alphabet의 가장 앞에 있는 Node를 저장하여 차후에 인덱싱을 할 때 사용하기 위해 선언했다. alphabet dictionary는 key로 정수 값을, value로 Node reference를 받는다.

**def** quick\_search(*self*, **question**):

        in\_dict **=** False

*#사전에 질의하는 단어의 맨 앞글자를 확인한다.*

        alphabet **=** *self*.\_\_english.index(question[0])

        """

        만약 해당 alphabet이 alphabet 인덱스에 존재하지 않는다면

        사전에 존재하지 않는다고 판단하고 in\_dict를 false로 return 한다.

        이때 alphabet이 self.alphabet에 key 값으로 존재하는지의 여부는 파이썬 딕셔너리의 특성상 O(1)이다.

        """

**if** alphabet **not** **in** *self*.alphabet:

**return** in\_dict, 0

**else**:

*#만약 사전에 해당 알파벳으로 시작하는 인덱스가 존재한다면 해당 인덱스의 시작 노드부터 비교를 한다.*

            cur **=** *self*.alphabet[alphabet]

            search\_time **=** time.time()

**while** cur **!=** None **and** question **>=** cur.data[0]:

**if** question **==** cur.data[0]:

                    print(cur.data[1])

                    in\_dict **=** True

**break**

**else**:

                    cur **=** cur.link

            search\_time **=** time.time() **-** search\_time

**return** in\_dict, search\_time

quick\_search는 search의 동작 방식에서 추가로 몇 가지 일을 더한다. 우선 입력 받은 단어의 첫 글자가 몇 번째 알파벳인지를 구한다. 이후에 해당 alphabet이 alphabet dictionary에 존재하지 않는다면 LinkedList에 해당 단어는 존재할 수 없기 때문에 in\_dict를 false로 return 한다. 만약 해당 alphabet이 alphabet dictionary에 존재한다면 그 alphabet으로 시작하는 가장 처음에 배치된 Node부터 link를 따라가면서 검색을 진행한다.

**def** initialize(*self*, **elem**):

        elem[0] **=** elem[0].lower()

        before **=** *self*.\_\_head

**while** before.link **!=** None:

**if** before.link.data[0] **>** elem[0]:

**break**

            before **=** before.link

        before.link **=** Node(elem, before.link)

        """

        더 빠른 Search를 위한 개선 사항이다.

        이미 Node는 추가되어 있는 상태이지만, 나중에 Search를 더 빠르게 해주기 위해서 선언한 self.alphabet을

        필요하다면 Update 시켜줘야 sel.alphabet이 가장 최신의 상태를 유지할 수 있다.

        따라서 self.alphabet에 해당 알파벳으로 시작하는 인덱스가 있는지를 먼저 확인하고 존재하지 않는다면

        새로운 key value 쌍을 등록한다.

        만약 존재한다면 두 글자를 비교하여 업데이트 시킬 필요가 있다면 self.alphabet을 업데이트 시킨다.

        """

        alphabet **=** *self*.\_\_english.index(elem[0][0])

**if** alphabet **not** **in** *self*.alphabet:

*self*.alphabet[alphabet] **=** before.link

**else**:

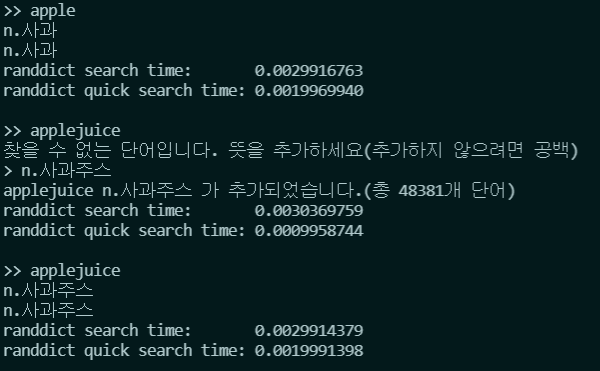
**if** *self*.alphabet[alphabet].data[0] **>** before.link.data[0]:

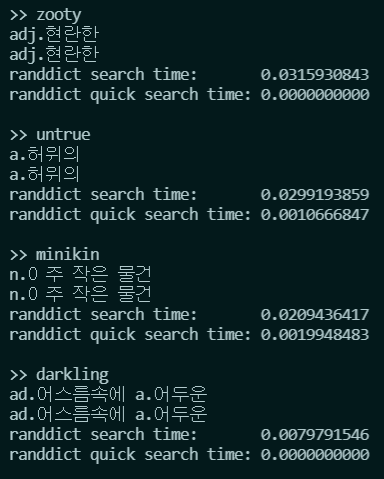
*self*.alphabet[alphabet] **=** before.link

*self*.\_\_count **+=** 1

새로운 Node를 LinkedList의 알맞은 위치에 배치하는 initialize 메서드도 변경이 필요했다. 기존의 head 변수를 이용해서 Node를 적절한 위치에 추가한 다음에 더 빠른 Search를 구현하기 위해 alphabet dictionary를 업데이트 해준다. 우선 english 변수를 이용하여 추가된 단어의 첫 글자가 몇 번째 알파벳인지를 판단한다. 이후에 해당 알파벳(정수값)이 alphabet dictionary에 없다면 새로 추가해준다. 만약 해당 알파벳이 이미 alphabet dictionary에 있었다면 두 Node의 Data를 비교하여 먼저 연결되어야 하는 Node를 판단하여 적절한 조치를 취한다. 이를 통해서 alphabet dictionary를 항상 최신 상태로 유지할 수 있고 이는 이후에 quick\_search를 할 때 더 빠른 검색을 위한 인덱스로 작용하게 된다. 아래의 결과에서 볼 수 있듯이 quick\_search는 단어의 첫 알파벳에 따라서 검색을 시작할 수 있는 위치를 유동적으로 바꾸어 주기 때문에 훨씬 빠른 시간에 의미를 찾아오는 것을 볼 수 있다. 또한 상대적으로 뒤에 위치한 알파벳을 검색할수록 head에서 모든 Node를 순환해서 찾아야하는 search 메서드는 더 많은 시간이 걸렸지만 alphabet dictionary를 사용한 quick\_search의 경우 일정하게 검색이 되는 것을 볼 수 있다. 결과적으로 alphabet dictionary를 저장해야 하는 공간은 추가로 소모되지만 시간의 측면에선 많은 개선이 일어났음을 확인할 수 있다.

* 1. **결과**





# **연결 리스트 응용하기**

* 1. **클래스와 메서드 정의**

class Node:

**def** \_\_init\_\_(*self*, **data**, **link=**None):

*self*.data **=** data

*self*.num\_link **=** link

*self*.name\_link **=** link

class LinkedList:

**def** \_\_init\_\_(*self*):

*#학번을 기준으로 하는 head -> num\_head*

*#이름을 기준으로 하는 head -> name\_head*

*self*.\_\_num\_head **=** Node(None)

*self*.\_\_name\_head **=** Node(None)

**def** insert(*self*, **student**):

*#student의 Node를 먼저 만들고 num\_head와 name\_head를 조절해주면서*

*#결과적으로 하나의 리스트를 2개의 리스트와 같이 사용할 수 있다.*

        student **=** Node(student)

*#학번 순 정렬을 구현한 코드이다.*

        before **=** *self*.\_\_num\_head

**while** before.num\_link **!=** None:

**if** before.num\_link.data[0] **>** student.data[0]:

**break**

            before **=** before.num\_link

        before.num\_link, student.num\_link **=** student, before.num\_link

*#이름 순 정렬을 구현한 코드이다.*

        before **=** *self*.\_\_name\_head

**while** before.name\_link **!=** None:

**if** before.name\_link.data[1] **>** student.data[1]:

**break**

            before **=** before.name\_link

        before.name\_link, student.name\_link **=** student, before.name\_link

*#학번 or 이름을 옵션으로 받아서 입력받은 옵션에 해당하는 head를 기준으로 결과를 출력한다.*

*#결과는 250명당 1번씩 출력했다.*

**def** display(*self*, **op**):

**if** op **==** "1":

            cnt **=** 0

            node **=** *self*.\_\_num\_head

**while** node.num\_link **!=** None:

                cnt **+=** 1

**if** cnt **%** 250 **==** 0:

                    print("%5d번 학생: 학번 -> %s, 이름 -> %s, 전화번호 -> %s" **%** (cnt, node.num\_link.data[0], node.num\_link.data[1], node.num\_link.data[2]))

                node **=** node.num\_link

**elif** op **==** "2":

            cnt **=** 0

            node **=** *self*.\_\_name\_head

**while** node.name\_link **!=** None:

                cnt **+=** 1

**if** cnt **%** 250 **==** 0:

                    print("%5d번 학생: 학번 -> %s, 이름 -> %s, 전화번호 -> %s" **%** (cnt, node.name\_link.data[0], node.name\_link.data[1], node.name\_link.data[2]))

                node **=** node.name\_link

**else**:

            print("잘못된 입력입니다.")

**return**

문제 2번의 핵심은 하나의 연결 리스트를 2개의 기준으로 정렬하는 것에 있다. 이를 해결하기 위해서 Node class에 각 기준에 맞추어 다음 Node를 가리킬 수 있도록 2개의 link 변수를 설정했다. 이후에 LinkedList는 각 기준으로 정렬할 경우 가장 처음에 위치하는 Node를 가리키도록 했으며 초기 값은 None이 아닌 더미 노드를 연결하여 추가로 구현하는 메서드를 보다 유연하게 사용하고자 했다.

Insert 메서드는 새로운 학생의 데이터를 연결리스트의 적절한 부분에 추가하기 위해 구현한 메서드이다. 우선 추가할 학생의 정보를 받은 다음 그 정보로 새로운 Node를 하나 만들고 학번과 이름을 기준으로 해당 Node가 배치될 위치를 순차적으로 찾은 뒤, num\_link와 name\_link를 조절하여 LinkedList를 완성한다. Display메서드는 사용자가 요구한 조건에 맞추어 연결 리스트를 출력하는 기능을 한다.

|  |  |
| --- | --- |
| **클래스 or 메서드** | **내용** |
| Node | Data를 저장할 Data, 학번 순 정렬을 위한 num\_link, 이름 순 정렬을 위한 head인 name\_link 로 구성되어 있다. |
| LinkedList | 학번 순 정렬을 위한 num\_head와 이름 순 정렬을 위한 name\_head로 구성되어 있다. Insert와 Display 메서드가 있다. Num\_head, name\_head는 처음 생성자를 통해 생성될 때 더미 노드와 연결된다. |
| Insert | 파라미터로 전달받는 학생 정보를 적절한 위치에 삽입하여 연결한다. |
| Display | 연결 리스트를 사용자의 요구 조건에 맞추어 출력한다. |

* 1. **무작위로 Data 생성하기**

DATA\_SIZE **=** 10000

student\_data **=** LinkedList()

*#random한 10000개의 학생 데이터를 만든다.*

random\_id **=** random.sample(range(20130000, 20220000), DATA\_SIZE)

**for** r **in** random\_id:

    name\_len **=** random.randint(1, 10)

    name **=** ''.join(random.choice(string.ascii\_uppercase) **for** \_ **in** range(name\_len))

    phone\_number **=** "010" **+** str(format(random.randint(0, 100000000), "08"))

    student\_data.insert([r, name, phone\_number])

무작위로 Data를 생성하는 방법은 이름을 제외하고 Assignment3에서의 방법과 동일하다.

* 학번: 8자리의 숫자로 랜덤하게 생성, 중복 불가
* 이름: 10자 이하의 영문자로 랜덤하게 생성, 중복 가능
* 전화번호: 11자리의 숫자, 010으로 시작, 중복 가능

학번은 중복이 불가능 하므로 중복되지 않게 표본을 만들어주는 random.sample 메서드를 사용했다. 이름과 전화번호는 중복이 가능하므로 각각 random.choice와 random.randint를 사용하였다. 이렇게 생성된 student\_data는 다음과 같은 구조를 가진다.

|  |  |  |
| --- | --- | --- |
| **Index 0** | **Index 1** | **Index 2** |
| 학번 | 이름 | 전화번호 |

* 1. **실행하기**

**while** True:

    print("\n(1) 학번순\n(2) 이름순")

    op **=** input("메뉴 선택(0 : 종료) : ")

**if** op **==** '0':

        print("종료")

**break**

**else**:

        student\_data.display(op)

사용자가 할 수 있는 입력은 크게 3가지가 있다.

* 0번을 눌러 종료한다.
* 1 또는 2번을 눌러 전체 리스트를 본다.
* 완전 잘못된 입력을 한다.

**def** display(*self*, **op**):

**if** op **==** "1":

            cnt **=** 0

            node **=** *self*.\_\_num\_head

**while** node.num\_link **!=** None:

                cnt **+=** 1

**if** cnt **%** 250 **==** 0:

                    print("%5d번 학생: 학번 -> %s, 이름 -> %s, 전화번호 -> %s" **%** (cnt, node.num\_link.data[0], node.num\_link.data[1], node.num\_link.data[2]))

                node **=** node.num\_link

**elif** op **==** "2":

            cnt **=** 0

            node **=** *self*.\_\_name\_head

**while** node.name\_link **!=** None:

                cnt **+=** 1

**if** cnt **%** 250 **==** 0:

                    print("%5d번 학생: 학번 -> %s, 이름 -> %s, 전화번호 -> %s" **%** (cnt, node.name\_link.data[0], node.name\_link.data[1], node.name\_link.data[2]))

                node **=** node.name\_link

**else**:

            print("잘못된 입력입니다.")

**return**

LinkedList의 인스턴스 메서드로 display를 만들었고 여기에 사용자가 입력한 값을 전달해서 취해야 하는 행동을 하도록 했다. 1이 입력된다면 학번순으로 출력을 해야 하므로 node의 시작점을 num\_head로 할당했다. 마찬가지로 2가 입력된다면 이름순으로 출력을 해야 하므로 node의 시작점을 name\_head로 할당했다. 학생은 250명당 1명씩 출력되도록 했고 총 40명의 학생이 기준에 맞추어 출력된 것을 아래 결과에서 볼 수 있다.

* 1. **결과**

