

과제 3

연결리스트 – 사전 만들고 개선하기

20232907 정현승

목차

1. 문제 1, 2: 연결 리스트로 사전 만들기, 정렬 성능 평가(C언어)	3
2. 문제 1, 2 해결 방안	3
3. 문제 1, 2의 결과	4
4. 문제 3, 4: 연결 리스트로 사전 만들기, 성능 개선하기(Python)	5
5. 문제 3, 4 해결 방안	5
6. 문제 3, 4의 결과	7
7. 소스코드	7

1. 문제 1, 2: 연결 리스트로 사전 만들기, 정렬 성능 평가(C언어)

단어 사전 파일에서 단어를 읽어 연결 리스트로 단어사전을 만든다. 이때 단어는 정렬된 상태로 저장되어야 하며, 이를 통해 단어 찾기를 구현한다. 또 연결 리스트로 단어사전을 만들 때와 배열로 사전 데이터를 전부 읽은 뒤 선택정렬 하는 것 사이의 소요 시간을 비교한다.

2. 문제 1, 2 해결 방안

문제 1과 2는 별도의 프로그램으로 만들지 않았다. 하나의 프로그램으로 구성했으며, 배열로 사전 데이터를 읽은 후 정렬할 때도 단어 찾기 기능을 구현했다.

이때 단어를 읽어 연결 리스트의 형태로 만드나 배열의 형태로 만드나 둘 다 사전이다. 본질은 변한 게 없다. 어떠한 형태의 사전이나 함수에 인자로 전달하면, 그게 어떤 형태로 이루어진 사전인지 모르더라도 같은 값을 반환해야 한다고 생각했다. 무엇보다 지금 가지고 있는 사전이 어떤 형태로 구현이 되어 있는지 신경을 쓰고 싶지 않았다. 따라서 **C언어에서 다형성을 구현했다**. 이를 위해 단어의 영어와 뜻이 저장된 문자열을 가리키는 포인터로 이루어진 단어 구조체(Word), 단어 하나와 다음 노드를 가리키는 포인터 값이 저장된 연결 리스트 구조체(WordLinkedList)(이때 연결 리스트는 단순 연결 리스트를 사용했는데, 그 이유는 다른 연결 리스트를 이용할 필요를 못 느끼기 때문이다), 그리고 연결 리스트 시작점 또는 단어 배열 시작점과 문자열 저장소, 단어 개수, 저장된 사전의 종류를 나타내는 enum 멤버로 이루어진 사전 구조체(ReadDict)를 선언했다. 사전 구조체에 단어가 배열로 저장되어 있던 연결 리스트로 저장되어 있건 main 함수에서는 상관하지 않고 특정 기능 사용(단어 찾기 또는 동적할당 해제)을 위해 함수를 호출 시 사전 구조체를 전달하면, 그 함수 내에서 사전의 종류를 파악해 종류에 맞는 동작을 실행해 주는 방식으로 다형성을 구현해 사용했다.

단어를 읽어 사전 구조체를 반환할 때는 파일명은 물론 사전의 데이터 저장 방식도 같이 받아, 파일을 모두 읽어 하나의 문자열로 저장하고, 이를 사전의 데이터 저장 방식에 맞게 문자열을 잘라 저장하였다. 데이터 저장 방식에 맞게 문자열을 잘라 저장한다는 부분을 자세히 설명하자면, 연결 리스트로 저장할 때는 문자열 앞에서부터 단어를 잘라 단어와 뜻으로 나누어 저장하고, 맨 앞에서부터 이번 단어가 들어갈 위치를 찾아 넣는다. 찾아서 넣을 때는 처음 넣을 때와 맨 앞에 넣어야 할 때, 중간이나 끝에 들어갈 때를 분리해 각각 코드를 작성했다. 배열로 저장할 때는 단어 구조체의 배열을 동적 할당받고 앞에서부터 순서대로 단어의 영어와 뜻을 채운 뒤, 요구사항에 따라 영어를 기준으로 선택정렬을 시행하였다. 단어 구조체 배열의 크기가 부족하지 않을 정도로 크게 받아, 배열의 크기가 부족할 때 재할당 기능도 구현했으나 활용하지 않았다. 여기서 각 단어나 뜻 앞뒤로 쓸모없는 공백문자가 들어가 있는데 이는 보기에 좋지 않고 이후 검색에 문제를 일으킬 수 있으니, 단어의 양 공백을 제거해 주는 함수 trim을 만들어 사용했다. 이 함수는 isspace 함수를 사용하지 않고 문자 하나하나를 검사하여 공백인지, 줄 변경 문자인지, 탭인지 확

인하는데, isspace 함수에 한글이 들어가면 에러가 발생하여 해당 함수를 사용하지 못했다. 사전 구조체에 데이터를 저장할 때 연결 리스트 방식으로 저장한 결과와 배열로 저장한 결과를 하나의 구조체에 담을 수 있어야 하므로, 구조체 내에 공용체를 사용했다. 두 방식으로 저장된 사전이 하나의 구조체 변수에 다 들어가지 못하고 둘 중 하나만 들어가므로, 공용체 사용을 선택했다. 따라서 단어가 저장된 위치에 접근할 때 공용체 멤버 변수도 작성해야 해서, 다형성을 구현한 대가로 코드가 약간 복잡해졌다(연결 리스트 방식으로 저장할 때 맨 앞 단어 접근은 result.dictionary.LinkedListDictionaryStart로 해야 한다). 코드는 복잡하지만, 작동에는 문제없으므로 계속 사용하였다.

특정 영어단어를 찾을 때 사전 구조체와 대상이 되는 문자열을 인자로 받게 되는데, 사전의 저장 방식에 따라 처리가 완전히 다르다. 서로 다른 함수라고 보아도 큰 무리가 없는 수준이다. 이 함수를 호출할 때는 사전이 어떤 식으로 저장되어 있든 상관이 없었지만, 단어를 찾는 입장에서는 사전 저장 방식이 매우 중요하기 때문이다. 사전이 연결 리스트 방식으로 저장되었으면 맨 앞 단어부터 하나하나 찾아야 하는 단어와 같은지 확인한다. 배열 방식으로 저장되었으면 전체 단어의 개수를 알고 있고 단어가 정렬도 되어 있으니 이진 탐색을 사용했다. 두 방식 공통으로 단어를 찾았으면 그 단어의 포인터 값을 넘기고, 단어를 찾지 못했으면 NULL을 반환한다.

마지막으로 동적할당을 해제하는 함수에서도 사전 데이터의 할당 해제는 기능별로 분리해 주고, 특히 연결 리스트 동적 할당 해제는 next를 찾아가며 모든 노드에 대한 동적할당 해제를 시행했다.

요구사항에서 2번 문제를 1번 문제에 포함해 하나의 프로그램으로 구성할 수 있으므로, main 함수는 하나만 놓고 먼저 2번 문제에서 요구하는 시간 측정을 먼저 수행했고, 그다음 1번 문제 요구사항인 검색을 구현했다. 이번에는 저번 과제와는 달리 콘솔 입력받을 때 동적으로 할당받은 공간에 저장하는 것이 아니라 길이가 50인 문자열을 선언해 여기에서 입력받도록 했다. 어차피 fgets 함수를 사용하면 정해진 길이를 넘는 문자열이 들어오지 않기도 하고 단어 길이는 그리 길지 않으며(즉 문자열의 크기를 넘는 입력이 들어오면 해당하는 단어가 없다는 점은 확실히 알 수 있다) 이 단어 입력 때문에 저번까지 사용했던 긴 코드를 사용하는 건 낭비라고 판단했기 때문이다. 또한 단어 입력은 콘솔 입력이 EOF를 반환할 때까지 반복하는데, 무한루프는 함부로 만드는 게 아닌 것 같고, 조건을 넣어줄 만한 게 이것밖에 없어서 넣었을 뿐 이 조건은 무시하고 무한루프라고 생각해도 문제없다.

그렇게 문제 1과 2를 해결한 결과는 다음과 같다.

3. 문제 1, 2의 결과

```
Microsoft Visual Studio 디버그 x + v
배열에 담은 후 정렬: 3.6060초 소요
연결리스트에 담으며 정렬: 7.7350초 소요
>> apple
n.사과
>> spring
n.봄
>> springtime
n.봄
>> yes
ad.예
>> of course
해당하는 단어가 없습니다.
>> jfsedjio;ha
해당하는 단어가 없습니다.
>> say
vt.말하다
>> structure
n.구조
>> data
n.자료
>> assignment
n.할당
>> assign
vt.할당하다
>> Adam
아담
>> zymurgy
n.양조학
>> ZZZ
해당하는 단어가 없습니다.
>> ^Z^Z^Z
해당하는 단어가 없습니다.
```

그림 3.1

위 그림 3.1의 앞 두 줄은 2번 문제의 결과이고, 그 아래는 1번 문제의 결과이다. 선택정렬이라 하더라도 배열에 담은 후 정렬하는 것이 약 2배 빠르다는 점을 알 수 있다. 바로 이어지는 1번 문제인 단어 찾기의 결과도 위와 같이 문제없이 동작한다는 점을 알 수 있다. 이 상태에서 프로그램이 종료되었는데 마지막 입력이 Ctrl+Z 3번인 점을 고려하면 넣어줄 거 없어서 넣어준 조건 까지 정상 동작한다는 점도 알 수 있다.

4. 문제 3, 4: 연결 리스트로 사전 만들기, 성능 개선하기(Python)

문제 1과 같은 기능을 하는 코드를 Python으로 구현하고, 성능을 개선할 방법을 찾아 구현한다.

5. 문제 3, 4 해결 방안

문제 3과 4도 별도의 프로그램으로 만들지 않고 하나의 프로그램으로 구성했다.

문제 3은 문제 1을 이중 연결 리스트의 구조로 변경하고 다형성 구현을 삭제한 것 정도가 전부이다. 새로운 노드를 연결 리스트에 연결하는 과정과 단어를 찾는 과정을 클래스로 옮긴 것 이외에는 문제 1과 크게 다른 부분이 없다. 그래서 코드 설명을 자세하게 적어놓지 않았다. 그래도 파이썬과 관련된 문법을 설명하자면 파일을 읽을 때 `readlines` 함수를 이용해 파일 전체를 한 번에 불러왔다. 이는 Python에서 전혀 권장하는 방식이 아닌데, 문제 1의 C 코드에서 파일부터 전부 읽고 작업을 진행한 것과 비슷한 느낌이기도 하고, 이 파일은 용량이 그리 크지 않다 보니 이와 같이 구현하였다. 또 연결 리스트에 단어 하나하나 연결하는 과정에서 헤드의 업데이트는 `return` 값으로 진행하였다. 문제 1에서는 한 함수에서 4만 8천여 개의 연결 리스트를 모두 생성하므로 헤드 업데이트는 단순히 지역 변수 값만 변경하면 되는 사항이지만, 여기서는 함수가 연결

리스트에 단어 하나만 추가해 주므로 return 값을 이용한 헤드 업데이트가 필요하다. 이외에도 이중 연결 리스트를 채택했으므로 연결 리스트에 단어를 넣을 때 꼭 head 노드 없이 중간 지점만 알아도 단어가 들어가도록 코딩했으나 큰 의미가 없는 기능이기에 테스트하지는 않았다. Node 하나를 만들 때도 head 노드를 주어 생성자에서 연결까지 완료하는 방향으로 하려 했으나 예러가 발생해 다른 방법을 이용했다(해당 기능과 관련된 코드는 지우지 않았지만 쓸 일도 없다).

사전 검색의 성능 개선 방법은 정렬된 연결 리스트 중 **중간에 있는 데이터 몇 개를 들고 있는 것**이다. 이렇게 되면 맨 앞에서부터 순차탐색 하기 전 따로 들고 있는 데이터끼리 비교해, 특정 데이터 사이에 값이 있다는 걸 알게 되면 그 사이만 탐색하면 된다. **이번에는 대소문자 앞 글자를 2개 따서** 정렬된 데이터 중 $52 \times 52 = 2,704$ 개의 단어 정보를 연결 리스트에서 들고 와 **2차원 리스트에 넣어 났고**(여전히 연결 리스트에는 저장되어 있다) 단어를 입력받으면 **입력받은 단어의 앞 두 글자를 따서 2차원 리스트의 각 index에 넣으면 사전에 있는 단어 중 입력된 단어와 앞 두 글자가 같으면서도 가장 앞에 있는 단어를 가리키게 된다.** 대소문자 앞 글자를 하나만 따면 너무 비효율적일 것이라 예상하여 2개를 땀다. 이후 그 단어부터 순차 탐색을 시작하면 된다. 순차 탐색 작업은 피할 수 없지만 그 수를 확 줄여주게 된다. 특히 48,406개의 단어 중 마지막 단어를 찾으려 하면 기존 방법은 순차 탐색을 48,406번 해야 하지만, 이 방법은 약 20번 정도만 하면 되어 부담을 크게 줄일 수 있다. 심지어 연결 리스트에서 들고 온 2,704개의 단어 정보는 리스트로 저장되어 있으므로, 입력된 단어와 앞 두 글자가 같은 단어를 찾는 과정은 연산 단 2번이면 충분하다. 특수문자 등으로 2번째 글자가 영어 글자가 아니어도 단어 정보를 찾는 과정은 예외 처리가 되어 있고(입력된 글자가 1글자일 것을 우려해 입력한 단어에 문자열 "AA"를 붙여 단어 정보 검색에 들어가며, chartoidx 함수에서 ascii-code가 영문자가 아니어도 유효한 숫자를 반환하도록 했다), 나중에 단어를 추가해 리스트에 저장된 단어가 앞 두 글자가 같으면서도 가장 앞에 있는 단어라 하더라도 이중 연결 리스트 구조를 사용하였기 때문에 저장된 단어에서 역순으로 올라가며 탐색할 수 있다. 다만 이를 위해 2,704개의 단어를 가져오는 사전 준비 작업은 오래 걸리나 준비시간 없이 단순히 검색 시간만을 보는 상황에서 이는 큰 상관이 없다.

요구사항에서 4번 문제를 3번 문제에 포함해 하나의 프로그램으로 구성할 수 있으므로, 문제 1과 2처럼 4번 문제에 해당하는 성능 비교를 먼저 진행하고 3번 문제에 해당하는 단어 찾기를 시행하였다. 따라서 연결 리스트를 만들 때 성능 비교를 위한 작업도 진행했다. 그리고 무엇보다 데이터 몇 개를 들고 있는 방법은 연결 리스트의 구조를 변경하지 않으므로, 하나의 연결 리스트를 공유해도 상관이 없다. 따라서 연결 리스트는 하나만 만들고, 성능 개선에 해당하는 리스트를 따로 생성해 연결 리스트는 3번 문제가, 리스트는 4번 문제가 중점적으로 사용하였으나 리스트에 저장된 연결 리스트는 3번에서 사용 중인 것을 그대로 가져왔다. 그러니까 문제 1, 2와 달리 두 리스트를 완전히 분리하지 않아 문제 3과 4의 경계를 헛갈릴 수 있으니, 주의를 요한다.

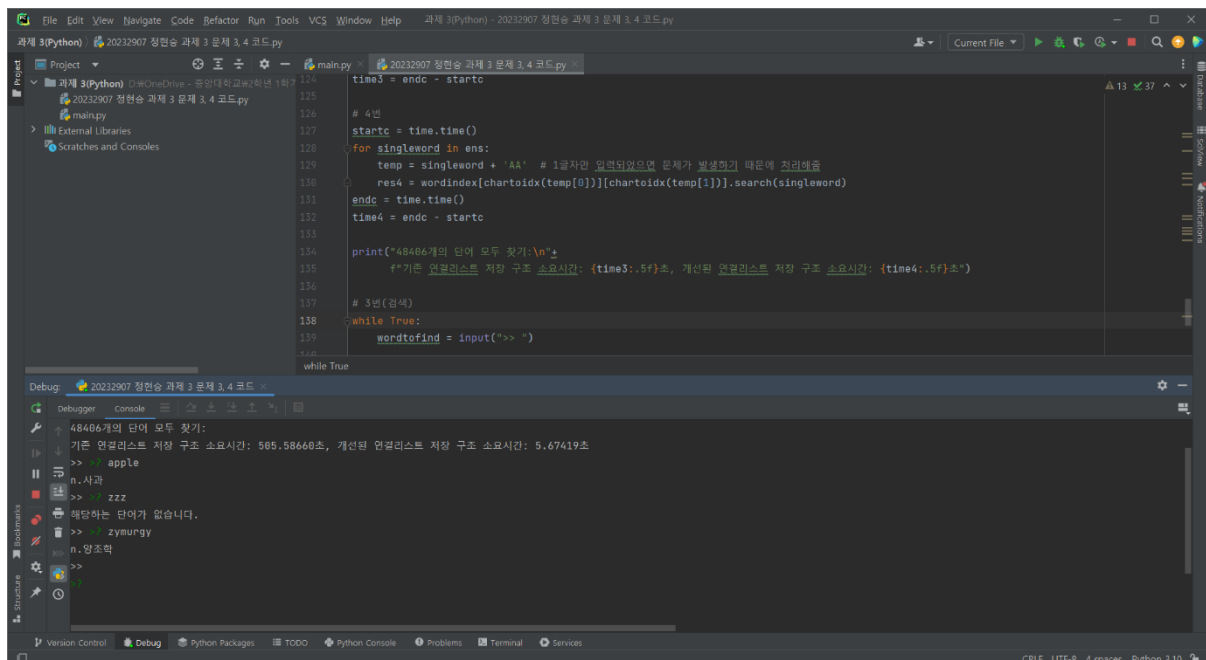
각종 밑 작업으로는 먼저 파일에서 단어를 읽을 때 영어단어만 따로 모아 저장하는 작업을 추가하였고, 4번 문제가 사용하는 리스트(wordindex)와 이를 만들기 위해 추가로 2종의 리스트를 더

만드는 것도 진행하였다. 또 영어단어 입력을 받으면 이를 리스트(wordindex)의 index 형식으로 바꾸어 주는 함수도 추가하였다(위에서 언급한 chartoidx 함수이다). 이 모두 3번 문제만 볼 때는 필요 없는 사항이다. 4번 문제에서 성능이 얼마나 개선되었는지는 앞에서 영어단어만 따로 모아 저장한 리스트의 단어 순서를 섞고, 이 48,406개의 단어를 전부 찾는 작업의 시간을 측정하는 방식으로 진행하였다. 단어 순서를 섞기 위해 random을 import 했고, 시간 측정을 위해 time을 import 했다.

위와 같은 방법으로 문제 3과 4를 해결했다.

6. 문제 3, 4의 결과

이 문제의 코드는 첫 콘솔 출력까지 10분이 넘는 시간이 소요된다. 프로그램이 에러가 발생한 것으로 오해하지 않도록 주의하기를 바란다.



```

File Edit View Navigate Code Refactor Run Tools VCS Window Help 과제 3(Python) - 20232907 정현승 과제 3 문제 3, 4 코드.py
Project 20232907 정현승 과제 3 문제 3, 4 코드.py
main.py
External Libraries
Scratches and Consoles
main.py
time3 = endc - startc
# 4번
startc = time.time()
for singleword in ens:
    temp = singleword + 'AA' # 1글자만 일컫났으면 문제가 발생하기 때문에 처리해줌
    res4 = wordindex[chartoidx(temp[0])][chartoidx(temp[1])].search(singleword)
endc = time.time()
time4 = endc - startc
print("48406개의 단어 모두 찾기:\n")
# 기존 연결리스트 저장 구조 소요시간: {time3:.5f}초, 개선된 연결리스트 저장 구조 소요시간: {time4:.5f}초"
# 3번(검색)
while True:
    wordtofind = input(">> ")
    while True

```

Debug: 20232907 정현승 과제 3 문제 3, 4 코드

Debugger Console

48406개의 단어 모두 찾기:
기존 연결리스트 저장 구조 소요시간: 505.58660초, 개선된 연결리스트 저장 구조 소요시간: 5.67419초

>> apple
n. 사과
>> zzz
>> zymurgy
해당하는 단어가 없습니다.
n. 양조학

그림 6.1

위 결과는 문제 3, 4 코드를 실행한 결과이며, 앞의 두 줄은 기존 연결리스트 저장 방식과 이를 개선한 방식 각각 48,406개의 단어 탐색을 해서 걸리는 시간을 측정한 것이다. 이를 통해 중간의 데이터 몇 개를 들고 시작하는 게 처음부터 단어를 찾는 것보다 **90배** 빨리 찾을 수 있다는 점을 알 수 있다. 즉 90배 효율적이다. 성능 비교 다음은 문제 3에 해당하는 단어 찾기인데 이것도 문제없이 작동한다는 점을 알 수 있다. 콘솔에 보이는 초록색 글자는 디버깅 보드에서 실행해 등장한 것일 뿐 디버깅 없이 실행하면 저 글자는 나타나지 않는다.

7. 소스코드

첨부파일 참조