과제 5

사전 탐색 트리

수식 트리

미로 찾기

지하철 짧은 길 찾기

20232907 정현승

목차

1. 문제 1: 사전 탐색 트리 만들기

사전 파일을 읽어 탐색 트리를 만든다. 이때 단어는 순차적으로 가져와 탐색 트리를 만든다. 트리 높이 최적화는 하지 않으며, 만든 후 트리의 노드 수와 전체 높이를 표시하고, 단어를 검색할 때 해당 단어의 레벨도 표시한다.

2. 문제 1 해결 방안

이 문제는 C언어로 해결하였다. 이 문제에서는 트리 높이 최적화를 하지 않으므로, 사전 파일을 읽고 읽은 순서대로, 가장 먼저 들어온 단어는 루트 노드에 저장되고 다음 단어는 이 루트 노드의 바로 왼쪽 또는 오른쪽으로, 그다음 단어는 이렇게 만들어진 트리에서 알맞은 리프 노드 위치에 저장하는 방식으로 트리를 완성할 수 있다. 이렇게 노드를 만들 때 만들어지는 노드의 수도 세 따로 저장했고, 결과에서도 설명하겠지만 이렇게 만든 노드의 수가 배열로 단어를 저장했을 때의 단어 수와 일치한다는 점도 확인했다. 또한 트리를 만들 때 루트 노드의 단어보다 작은 단어(사전의 앞에 오는 단어)는 왼쪽에, 큰 단어(사전의 뒤에 오는 단어)는 오른쪽에 배치하였으므로, 검색 시에도 입력된 단어와 루트 노드의 단어를 비교해 가며 입력된 단어가 루트 노드의 단어보다 작으면 왼쪽 노드를 루트 노드로 해서, 크면 오른쪽 노드를 루트 노드로 해서 검색을 이어 나갔다. 이때 재귀 함수를 쓰지 않고 단순 반복문을 사용했다. 이는 이 함수를 재귀 함수로 작성하였을 때 이점이 별로 없기 때문이다. 또 파일을 읽는 부분은 저번 과제에서 작성한 코드를 많이 가져왔다. 이유는 저번과 똑같이, 같은 사람이 같은 일을 하는 코드를 한 번 더 짜 봐야 아이디어나 방식은 같으니 코드가 별로 다르지 않기 때문이다.

위와 같은 방법으로 문제 1을 해결하였다. 다만 코드는 문제 2와 통합해서 짰기 때문에, 문제 1의 결과는 문제 2와 같이 보인다.

3. 문제 2: 사전 탐색 트리 개선하기

문제 1처럼 사전 파일을 읽어 탐색 트리를 만들되, 트리의 높이를 최적화한다.

4. 문제 2 해결 방안

이 문제도 C언어로 해결하였다. 탐색 트리의 높이를 줄이는 방법은 노드의 비교 연산을 줄이는 것, 즉 거의 완벽한 이진 탐색을 구현하면 된다. 이 이진 탐색을 위해서는 단순히 데이터를 정렬하고 이진 탐색을 할 시 확인하게 되는 노드 순서대로 트리를 만들면 된다. 따라서 이진 탐색을 하던 방식 그대로, 정렬 결과 중 정중앙에 위치한 단어를 루트 노드로 삼았고, 그 정중앙에 위치한 데이터 기준으로 앞에 있던 단어를 또 다른 단어 정렬 결과로 보고 그 중 정중앙에 위치한 노드를 왼쪽 노드로 삼고, 뒤에 있던 단어도 또 다른 정렬 결과로 보고 그 중 정중앙에 위치한 노

드를 오른쪽 노드로 삼고, 이를 더 이상 노드를 나눌 수 없을 때까지 진행하였다. 여기서 정렬은 과제 4에서 만든 것을 다시 가져왔다. 과제 4에서 내장 정렬보다 직접 만든 게 더 빠르다는 것이 증명되었기 때문이다. 이 과정에는 재귀 함수를 사용하였는데, 이는 다른 이유가 아니라 귀찮아서, 즉 코드를 짧게 쓰고 싶었기 때문이다. 또한 이 과정에서도 노드의 수를 세야 하므로, 함수의 반환값을 루트 노드와 노드의 수를 담은 정수형 변수로 이루어진 구조체로 두었으며, 함수 내에서 구조체를 반환받으면 두 변수를 분리해 루트 노드는 child 노드로 저장하고, 노드의 수는 전체 노드의 수를 세기 위해 두 값을 더해 반환하도록 했다. 또 재귀 함수를 사용해 단어가 3개 남았으면 루트 노드와 좌, 우 child로 단어를 배정하면 되고, 2개 남았으면 큰 단어를 루트 노드로 했고 작은 단어를 왼쪽에 두었다. 이는 재귀 함수 사용 시 중앙 index를 계산해 둘로 나누면, 단어의 수가 짝수인 경우 왼쪽이 오른쪽보다 1개 많은 단어가 들어가기 때문이다. 즉 단어의 수 len=2이면 두 단어의 index는 각각 0, 1이고, len/2=1이므로 index가 1인 큰 단어가 루트가 되고, 작은 단어가 왼쪽에 들어가게 되는 것이다. 굳이 단어의 수가 짝수일 때에 대해 보정해야 할 이유를 느끼지 못해 이렇게 두었다.

나머지 부분은 문제 1과 같다. 위와 같은 방법으로 문제 1과 문제 2를 해결하였다.

5. 문제 1, 2의 결과

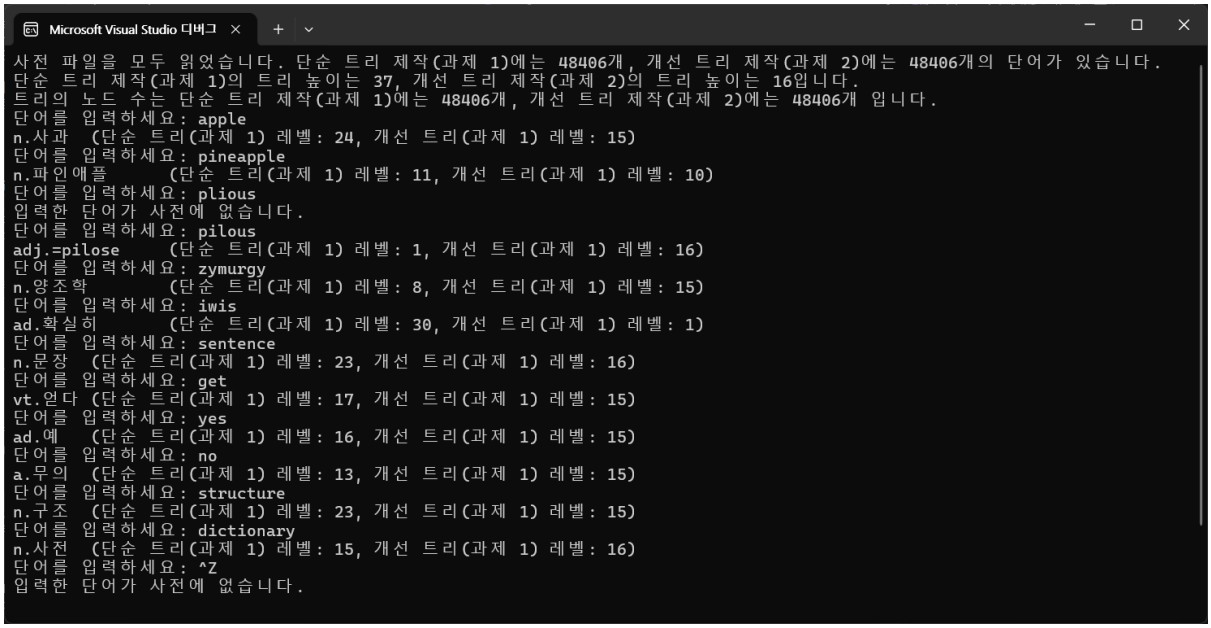

그림 5.1

위 그림 5.1은 두 탐색 트리를 만든 결과와 각 단어당 레벨을 나타낸 것이다. 트리의 노드 수나 개선 트리 높이를 보면 어느 정도 정상적으로 만들어졌음을 확인할 수 있으며, 개선 트리의 레벨이 평균적으로 낮다는 것을 알 수 있다. 다만 개선 트리의 레벨이 절대적으로 낮을 수는 없다. 단순하게 만들었을 때 가장 먼저 읽은 단어는 레벨이 1이기 때문이고(pilous), 읽은 순서에 따라 레

벨은 조금씩 차이 난다. 다만 개선 트리의 목적이 자주 쓰는 단어의 레벨을 낮추는 게 아니라 단순히 높이를 낮추는 것이기 때문에 그 목적은 달성했다고 볼 수 있다. 실제 48406개의 노드를 이진 트리로 구성할 때 필요한 최소한의 높이는 $\log_2 32768 = 15 < \log_2(48406 + 1) < 16 = \log_2 65536$으로 그림 5.1의 결과와 일치한다.

6. 문제 3: 수식 트리 만들기

입력된 중위 표기의 수식을 수식 트리로 만든다. 전위 탐색, 중위 탐색, 후위 탐색, 레벨 순회 결과를 표시하고, 수식 계산 결과를 표시한다. 수식은 +, -, *, /, (,) 6종류이며, 입력되는 수는 실수이다.

7. 문제 3 해결 방안

이 문제는 Python을 이용해 해결하였다. 수식을 읽어 트리로 바꾸는 과정은 과제 2 문제 2의 코드 일부를 활용했다. 스택에 숫자와 연산자를 차례로 넣은 후 우선순위가 낮은 연산자가 등장하면 이전에 스택에 넣어 두었던 연산을 pop 해 트리를 구성하도록 했다. 이 트리에서 숫자는 무조건 leaf 노드이고, 수식은 왼쪽과 오른쪽 자식을 항상 동시에 가지고 있으니, 노드에 저장된 정보의 종류를 파악하는 방법은 노드의 왼쪽 포인터가 None인지 확인하는 방법으로 하였다. 또 숫자 노드 밑에 다른 노드가 있을 수 없으니, 숫자 노드는 스택에 넣을 때부터 트리 노드의 형태로 바꾸어 넣었다. 그런데 생각해 보니 입력되는 숫자는 실수지만 요구사항의 실행 예에는 숫자가 정수로 출력되어 있고, 또 실수 소수점 확인을 위해 소수점이 나왔는지 그렇지 않았는지 확인하는 pointflag 변수를 사용하고 있어서, 입력된 숫자가 정수이면 정수로, 소수점 이하가 존재한다면 실수의 형태로 트리에 저장했다. 저번 과제 2 문제 2에서 여는 괄호가 나오면 우선 스택에 저장한 후 닫는 괄호가 나온 후에야 괄호 안 수식을 정리했다면, 이번에는 재귀 함수를 이용하여, 여는 괄호가 보이자마자 이와 짝이 되는 닫는 괄호를 찾아 그 구간 내의 트리를 즉시 구하도록 했다. 괄호 특성상 내부에 정의된 수식은 밖에서 하나의 숫자로 취급하므로 필요한 작업을 미리 해준 것이다. 따라서 재귀 함수를 호출하는 쪽에서는 괄호 내부 내용은 전부 건너뛰게 된다. 이때 여는 괄호만 연속으로 2번 나오는 문제를 해결하기 위해 별도의 변수를 두어 괄호의 수가 맞는 부분을 떼서 재귀함수로 호출했다. 재귀함수를 사용한 이유는 다른 문제와 같이 스택 구현을 더 하기는 귀찮기 때문이다. 특히 이전 함수의 각종 변수를 저장해 두어야 하는 특성상 함수 호출 전 지역변수를 따로 빼서 저장하는 작업까지 완료해야 해, 스택 구현이 어렵지는 않지만, 아주 귀찮다. 따라서 단순히 재귀함수를 사용하는 쪽을 택했다. 또한 이번 문제의 요구사항에서 빠진 제곱 관련 부분을 삭제하였고, 수식 오류 발견 시 예외를 던지는 방식을 선택하였다. 이번 문제는 저번 과제 2 때와는 달리 예외 처리 요구를 명시하고 있지는 않으나, 작업해 놓은 게 있어서 그대로 활용했다. 다만 저번에는 에러 메시지를 문자열에 담았다면, 이번에는 직접 예외를 생성하고 트리 생성 함수를 호출하는 쪽에서 이를 받아 처리하도록 진행했다. 또 Python에서도 예외에 문

장을 넣고 출력할 수 있다는 것을 알게 되어[i], 이를 활용했다(출처가 블로그이지만 코드가 정상 동작해 그대로 가져왔다). 이외에는 과제 2의 코드와 큰 차이는 없다. 입력된 문자를 하나하나 살피며 숫자인지, 공백인지, 연산자인지 확인 후 연산자이면 연산자 우선순위를 고려해 트리를 만드는 방식이다.

트리 자체는 숫자 하나만을 가지는 leaf 노드를 제외하고는 앞에 오는 수를 왼쪽 노드로, 연산자를 부모 노드로, 뒤에 오는 수를 오른쪽 노드로 하여 만들었고, 가장 나중에 하는 연산을 루트 노드로 하여 만들었다. 계산 시에는 왼쪽 노드와 오른쪽 노드에 대해 부모 노드의 연산자에 해당하는 연산을 시행했으며, 이때 사용하는 왼쪽 노드의 계산 결과는 그 노드의 왼쪽과 오른쪽에 대해 그 부모 노드의 연산자에 해당하는 연산을 해서 구하고, 이를 계속 반복하는 방식으로 진행하였다. 이때도 자식 노드의 연산 결과를 구하기 위해 재귀함수를 사용하였다. 이 과제에서는 수식 트리를 만든 후 전위 순회, 중위 순회, 후위 순회, 레벨 순회도 진행해야 하는데, 레벨 순회는 리스트를 활용했고 나머지 순회 또한 순서만 다를 뿐 재귀함수를 이용했다. 레벨 순회에서는 리스트를 작성했는데 이 이유도 큐를 구현하면 코드가 복잡해질 것을 우려했기 때문으로, 어차피 Python을 사용하는 이상 성능은 버린 것이나 마찬가지이기 때문에, 리스트의 맨 앞에 저장된 노드를 가져오는 작업을 반복했다. 강의자료에 있는 import deque를 사용해도 되나 저것과 리스트가 큰 차이는 나지 않을 것으로 생각해, 새로운 것 쓰다가 괜히 에러 많이 띄우는 것보다는 기존에 쓰던 것을 사용하는 편을 선택했다.

이외에도 트리 구조를 저렇게 짜니 전위 순회나 후위 순회 시 그 순서가 연산식의 전위 표기법과 후위 표기법과 일치하게 되었으며, 중위 순회는 중위 표기법과 순서는 같으나 괄호가 표기되지 않는다는 차이점이 있다.

이와 같은 방법으로 문제 3을 해결하였다. 그 결과는 다음과 같다.

8. 문제 3의 결과

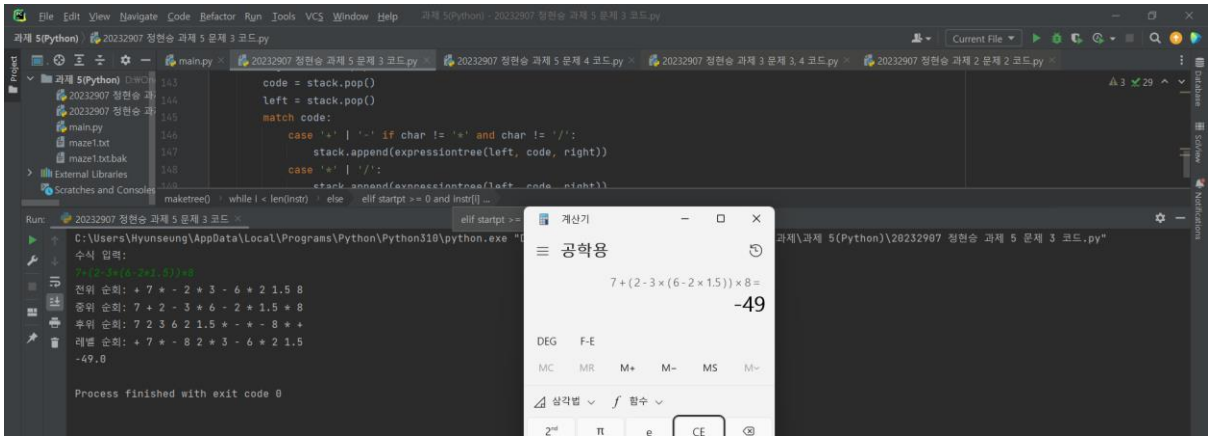

그림 8.1

그림 8.1은 정상 상태 동작을 나타낸 것이다. 괄호가 여러 개여도, 괄호 뒤 우선순위 높은 연산자가 오더라도 트리도 정상적으로 만들어지고 결과도 정상적으로 나온다는 점을 통해 코드가 정상 동작함을 알 수 있다.

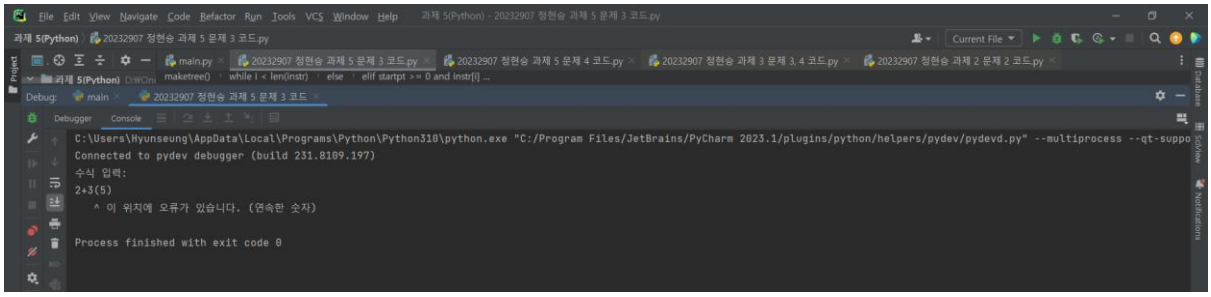

그림 8.2

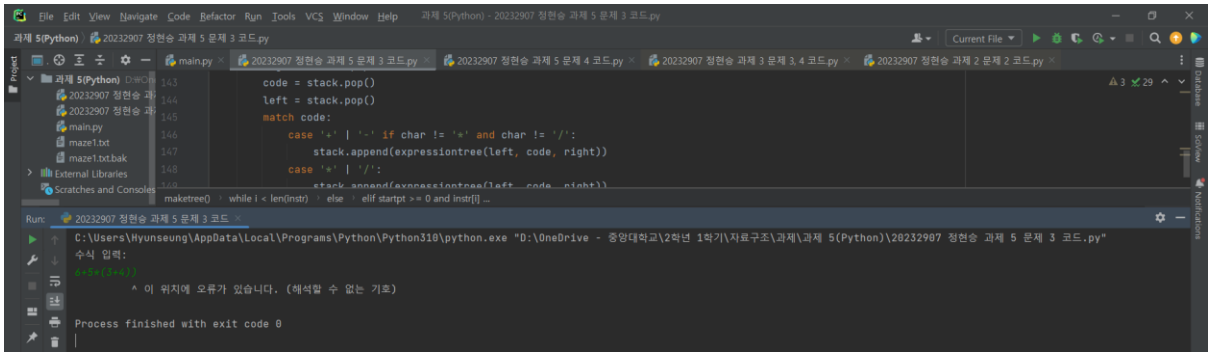

그림 8.3

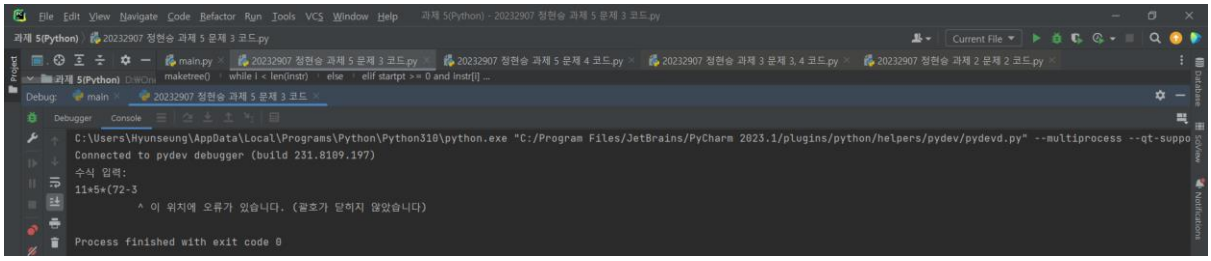

그림 8.4

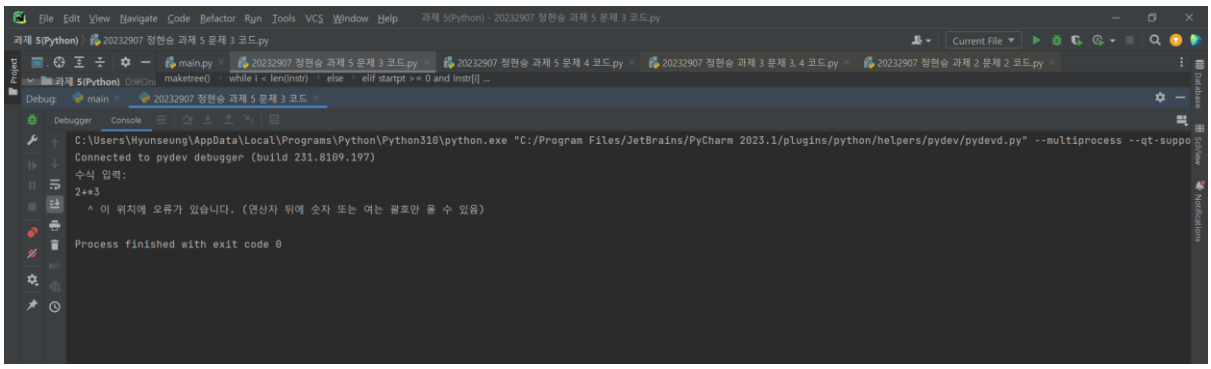

그림 8.5

그림 8.2~그림 8.5는 예외 상황을 나타낸 것이다. 비록 요구사항으로 명시되지는 않았지만 잘

동작함을 알 수 있다.

9. 문제 4: 미로 분석기

과제 1에서 다룬 미로 파일 형식의 파일을 분석해 길이 있는지, 2개 이상 있는지 분석하여 결과를 표시한다. 길이 있다면 그 결과를 화면에 표시한다.

10. 문제 4 해결 방안

이 문제 4는 다른 방법이 아니라 스택을 이용해서 문제를 해결했다. 자료구조도 과제 1에서 사용하던 방식을 그대로 가져왔다. 과제 1에서 쓰던 방식이 한 칸에 각각 오른쪽과 아래쪽에 길이 있는지 저장하고 이 칸의 2차원 배열을 만드는 것인데, 미로를 찾는 데도 문제가 없을 것 같아 이를 활용했다. 또 스택 배울 때 교수님께서 미로가 스택 사용의 예시라고 하셨는데, 이를 참고해 스택(Python의 list)을 이용해 문제를 해결하였다.

먼저 파일을 읽어 2차원 배열을 만들어야 하는데, 파일 구조가 첫 번째 줄에는 미로의 크기가 명시되어 있고, 가로줄과 세로줄 각각 1byte는 칸, 1byte는 벽의 유무를 나타내는 구조로 되어 있다. 따라서 한 열을 읽기 위해서는 파일에서 칸에 해당하는 줄과 아래 벽을 나타내는 줄까지 2개 줄을 읽어야 한다. 또 한 줄에는 칸, 벽, 칸, 벽, …이, 다른 한 줄에는 벽, 의미 없는 공간, 벽, 의미 없는 공간, …이 체스판 형식으로 반복되는 구조이다 보니 파일을 2줄씩, 칸 하나에 2개 문자를 읽어가며 2차원 배열을 만들었다. 첫 번째 칸 위치에 도착하면 그 위치로부터 한 칸 앞과 한 칸 아래에 벽이 있는지 살핀 뒤, 오른쪽으로 2칸을 넘어가 이를 반복하고, 이렇게 1개 열을 읽었으면 1줄이 아닌 2줄을 내려가 이를 반복하는 방식으로 미로 파일을 읽어 2차원 배열로 만들었다.

미로를 찾는 과정은 앞서 언급했듯 스택을 이용했다. 미로를 나타내는 크기와 같은 크기의 2차원 배열을 만들어 이번에는 각 칸이 이미 지나친 칸인지, 다음으로 이동한 칸은 어디인지 저장하도록 했다. 이는 원본 미로 데이터(2차원 배열)을 만지지 않고 미로를 찾은 결과를 미로 데이터와 분리해서 저장하려고 했기 때문이다. 미로를 찾다 보면 여러 갈래가 나올 수 있는데, 미로를 찾는 과정을 미로 데이터에 직접 저장하면 원본 데이터 훼손은 물론 갈래가 나뉘었을 때 공유하는 값이 생겨 치명적인 오류가 발생할 것을 우려했다. 물론 미로를 찾은 결과를 저장하는 배열을 따로 쓰더라도 값 복사가 잘되지 않아 문제가 생길 수 있는데, 이를 차단하기 위해 deepcopy 메서드를 사용하였다[ii]. 이 메서드를 이용해 깊은 복사를 시행하면 배열을 다른 변수에 대입했을 때 그 변수만으로 원본 데이터에 접근하고 조작할 수 있는 것과 다르게 특정 변수가 참조하는 클래스도 전부 새로 만들어지기 때문에 원본 데이터를 조작할 수 없다. 길을 찾다 여러 경로가 하나의 2차원 배열(보드)에 저장되는 일이 벌어지면 문제가 되므로 이 deepcopy 메서드를 활용했다.

스택을 이용해 미로를 찾는 방법은 간단한데, 현재 위치보다 위쪽에 칸이 있고 벽이 없고 이미 간 적이 없다면 한 칸 위로 이동해 이 정보를 리스트에 push 하고, 같은 방식으로 왼쪽, 오른쪽과

아래쪽도 진행한다. 오른쪽과 아래쪽은 이번 이동을 통해 맨 마지막 칸에 도달했으면 스택에 push하는 게 아니라 결과 저장 list에 push 하는 식으로 결과를 저장했다. 이때 칸을 하나 이동할 때마다 deepcopy를 진행했다. 미로에 길이 몇 개 있는지는, 이 과정을 스택이 빌 때까지 반복하고 결과 저장 list에 저장된 경로의 수를 확인하면 된다.

마지막은 이 결과를 화면에 출력하는 것인데, 벽이 끝나거나 교점에는 '-'나 '|'이 아닌 '+'를 표시해야 해 조금 까다로웠다. 이 때문에 화면에 출력할 문자를 한 번에 만드는 게 아니라 한 번 만든 후 만들어진 문자열을 보며 약간씩 수정하는 방식으로 진행했다. 먼저 첫 번째 줄은 테두리이므로 "+-----+"같은 방식으로 둘러주었다. 그리고 이번에도 파일에서 읽을 때처럼 칸과 벽을 따로 표시하는데, 이 때문에 파일을 읽을 때처럼 한 번에 2개 줄을 그리게 되었다. 2개 줄 중 첫 번째 줄은 칸과 벽이 번갈아 나타나므로 처음에 테두리 '|'를 그려주고, 칸은 미로의 최종 경로가 이 칸을 지나면 '0', 아니면 ' '을 띄우고, 벽은 파일을 읽을 때처럼 오른쪽에 벽이 있으면 '|', 아닌 상태에서 오른쪽으로 가거나 오는 게 최종 경로라면 '0', 다 아니면 ' '을 띄우도록 했다. 이 부분은 벽이 끝나거나 만나지 않기 때문에, 이 정도로만 하면 되고 나중에 다시 그릴 일도 없다. 문제는 그다음 줄인데, 바로 위가 벽이 아닌 칸이 오는 위치라면 이 공간에는 벽이 있으면 '-', 최종 경로가 지나가면 '0', 아니면 ' '을 띄우는 것으로 끝난다, 벽과 벽 사이 칸이 문제인데, 일단 왼쪽에 벽이 있으면 '-', 위쪽에 벽이 있으면 '|', 왼쪽에 벽이 있는 상태에서 오른쪽에 벽이 없으면 '+'를 써주었다. 이렇게 한 번 전체적으로 모양을 그린 후 두 번째 줄만 모아 '+'가 들어갈 위치에 '+'를 넣어 주는 작업을 진행했다.

이렇게 얻은 결과를 출력하는 방법으로 문제 4를 해결하였다.

11. 문제 4의 결과

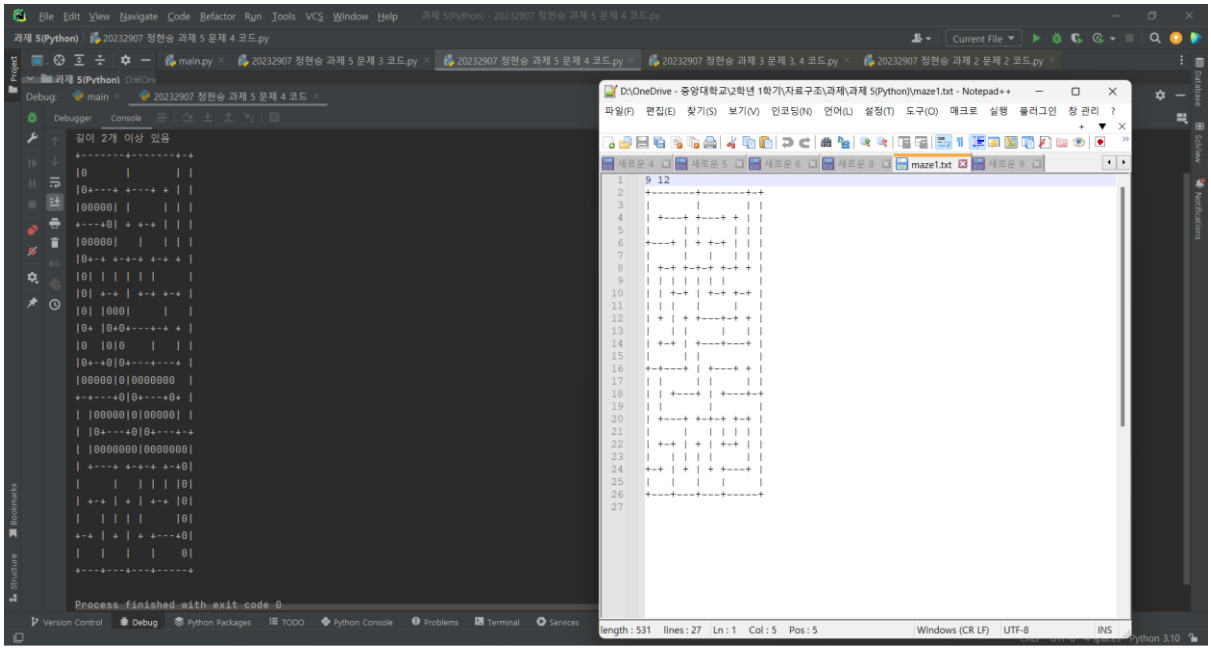

그림 11.1

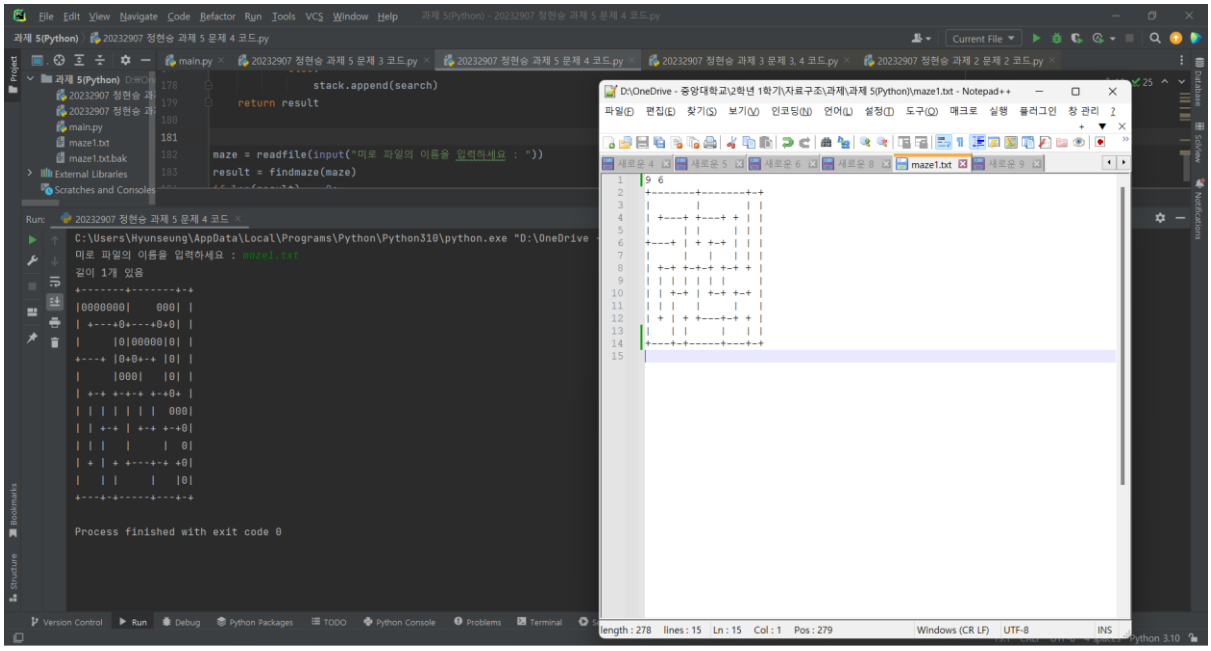

그림 11.2

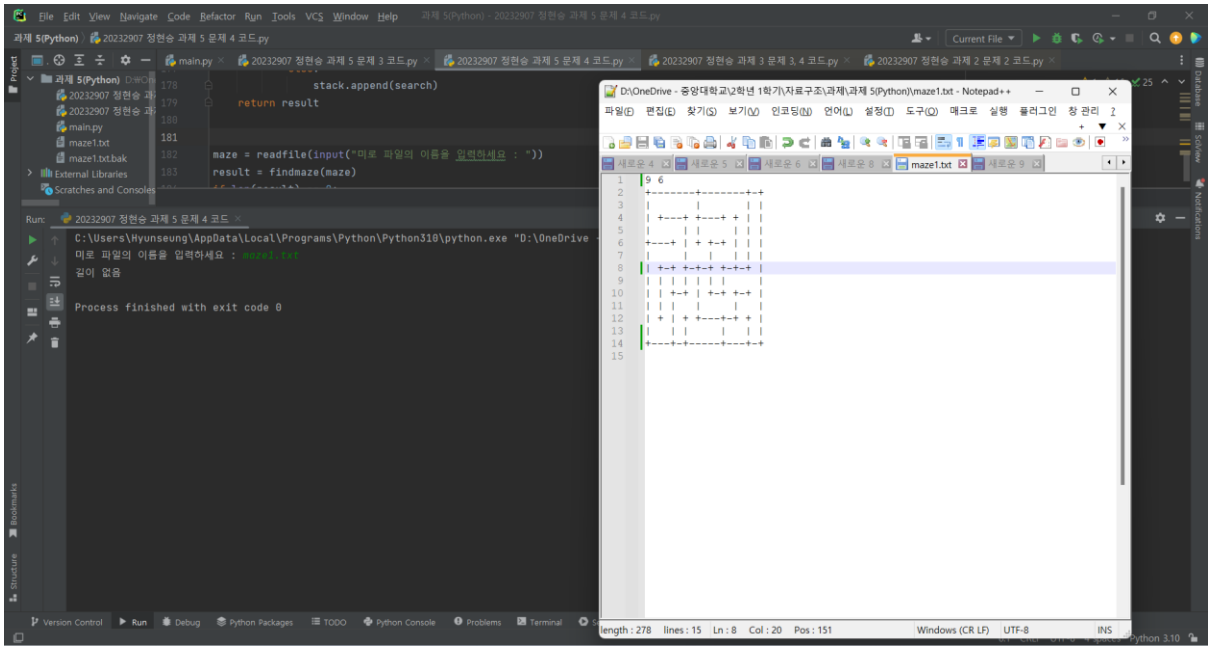

그림 11.3

그림 11.1은 길이 2개 이상 있을 때, 그림 11.2는 길이 1개 있을 때, 그림 11.3은 길이 없을 때는 maze1.txt와 함께 나타낸 것이다. 위 결과를 보아 동작에 문제가 없다는 점을 알 수 있다.

12. 문제 5: 지하철 짧은 길 찾기

사용자가 선택한 2개 역 사이의 짧은 길을 찾아 표시한다. 지하철 노선도는 직접 노선 8개 이상, 환승역 8개 이상으로 골라 40개~50개 정도로 역을 선택한다. 지하철 노선도 데이터는 텍스트 파일로 직접 만든다.

13. 문제 5 해결 방안

이 문제는 복잡해 보이기는 하나 C언어를 통해 해결했다. 우선 지하철역의 범위를 정해야 한다. 마침, 집 근처 전철역에서 노량진까지 이동 시에는 기본요금만 붙지만(최단 거리 10km 이내), 흑석역이나 상도역까지 가면 추가 요금 100원이 붙고(최단 거리 10km 초과, 15km 이하), 집 근처 전철역이 아닌 학교에 더 가까운 역까지 버스로 이동 후 노량진까지 이동 시에도 추가 요금 100원이 붙었다(버스 이동 거리+지하철 최단 거리 10km 초과 15km 이하). 왜 이렇게 되는지 확인을 해 보고 싶었기 때문에, 지하철역의 범위는 집 근처 전철역에서 학교 인근 지하철역까지로, 이 범위 이내 모든 노선과 역을 대상으로 잡았다. 범위는 그림 13.1에서 동그라미 친 부분과 같다.

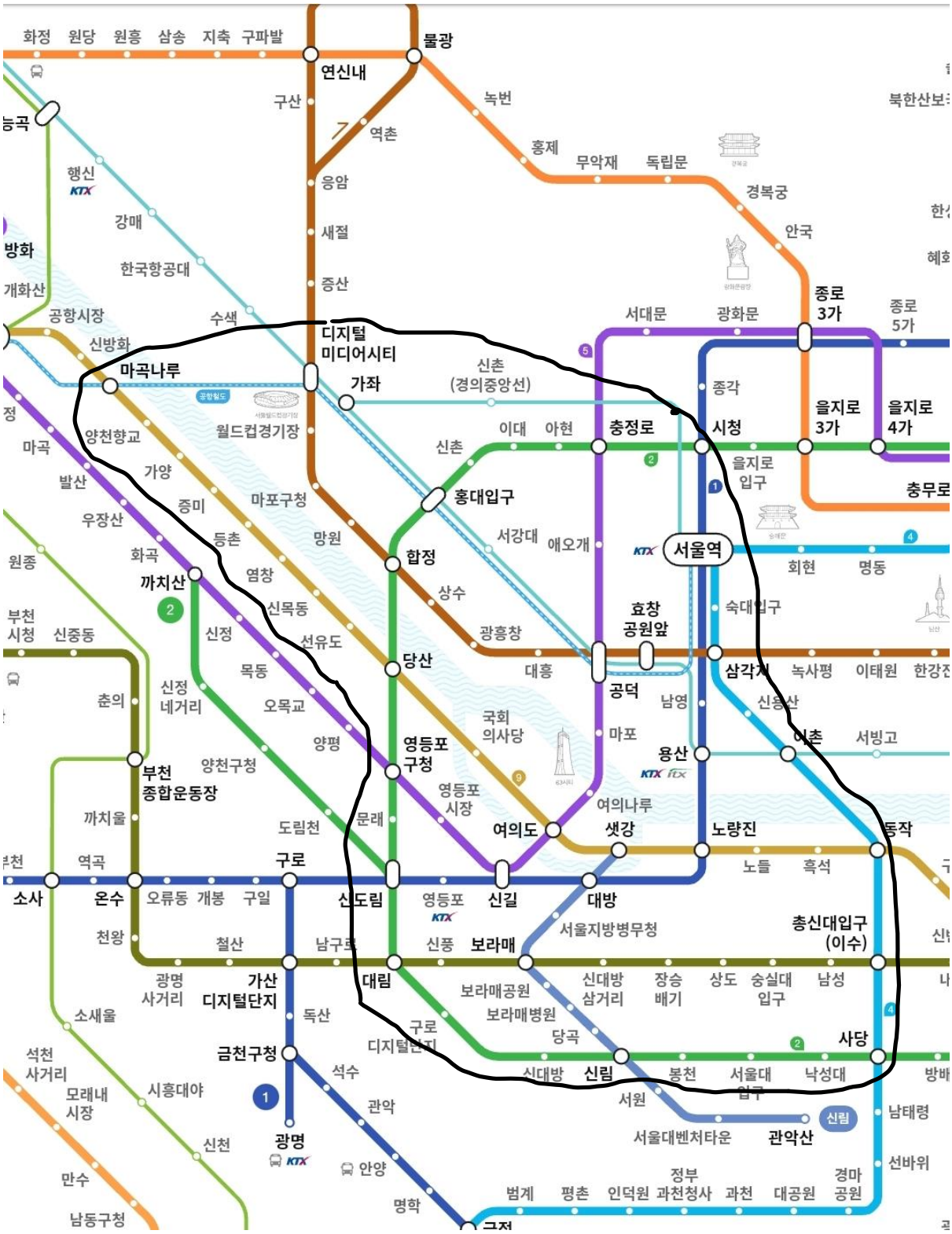

그림 13.1

또한 텍스트 파일의 형식은 다음과 같다. 각 역을 구분하는 구분자(key)로 역 번호를 지정했다. 이 역 번호는 모든 노선의 모든 역이 겹치지 않고 보유하고 있으며, 환승역이면 노선별로 다른

번호가 지정되어 있다. 역 번호는 원래 노선을 나타내는 부분과 역의 순서를 나타내는 번호가 합쳐져 있어 노선을 구분하는 용도로도 사용이 가능하나, 이번에는 노선 구분 용도로는 역 번호를 사용하지 않았다. 다만 그림에서 2호선 신촌역과 경의선 신촌역같이 이름이 같으나 별개인 역이 존재해, 무작정 역명이 같다고 환승역으로 볼 수 없다는 문제가 있어 이 역 번호를 이용해 환승역 정보를 넣어주었다.

이 프로그램이 읽는 텍스트 파일 형식은 다음과 같다. csv 파일 형식으로 만들어 Excel의 형태로 그림 13.2를 담았으며, 노선, 역 번호, 역 이름, 기점으로부터의 누적 거리와 환승역의 역 번호를 전부 작성해 주었다. 환승역 구분은 저것으로 진행한다.

호선	역번호	역명	누계(km)	환승역1	환승역2	환승역3
1	132	시청	67.4	201		
1	133	서울역	68.5	426	P313	A01
1	134	남영	70.2			
1	135	용산	71.7	K110		
1	136	노량진	74.3	917		
1	137	대방	75.8	S402		
1	138	신길	76.6	525		
1	139	영등포	77.6			
1	140	신도림	79.1	234		
2	226	사당	27	433		
2	227	낙성대	28.7			
2	228	서울대입구	29.7			
2	229	봉천	30.7			
2	230	신림	31.8	S408		
2	231	신대방	33.6			
2	232	구로디지털	34.7			
2	233	대림	35.8	744		
2	234	신도림	37.6	140		
2	235	문래	38.8			
2	236	영등포구청	39.7	523		
2	237	당산	40.8	913		

그림 13.2

역 간 거리 정보는 서울 열린데이터 광장[iii]과 공공데이터포털[iv]에서 얻었으며, 여기서 얻을 수 없었던 경의중앙선과 신림선의 역 간 거리 정보는 아무렇게나 지정해도 상관없으니, 나무위키의 것을 가져왔다(비록 신뢰성은 없으나 어차피 자료를 구할 수 없기에 의존할 수 있는 게 여기밖에 없다. 이번에는 자료의 신뢰도도 신경 쓰지 않아도 되니 부담 없이 긁어왔다). 이때 거리 정보는 기점으로부터의 누적 거리로 환산한 값을 사용했고, 누적 거릿값이 작은 게 먼저 오도록 했다. 노선 구별은 맨 앞 노선 정보로 진행하였다. 이때 이런 형식의 데이터는 노선 분기를 담을 수 없다. 바로 위에 적힌 역이 전 역이고 아래 적힌 역이 다음 역으로 인식되기 때문이다. 그림 13.1의 데이터에는 경의중앙선 서울역 지선 분기가 존재한다(가좌~서울역). 따라서 이 지선은 다른 노선으로 취급하도록 해 문제를 해결했다. 따라서 별개의 노선으로 인식하기 위해서는 노선명을 바꾸어 주어야 하므로, 데이터를 보면 서울역 지선 구간은 경의중앙선이 아니라 경의선이라고 쓰여 있다. 또한 당연히 환승역 목록에 서로를 포함했으며, 테스트는 해보지 않았지만, 에러가 날 것을 대비해 경의중앙선 본선 구간의 가좌역과 지선 구간의 가좌역은 역 번호를 다르게 적어주었다(원래 두 역의 역 번호는 K315로 같다). 마지막으로 환승역 정보는 같은 역으로 취급되는(환승할 수 있는) 역의 번호를 있는 대로 앞에서부터 써 주었다.

코드 내 데이터 구조는 역 목록을 담은 노선 구조체와 역 이름, 누적 거리와 환승역 정보가 담긴 역 테이블, 그리고 역을 특정할 수 있게 해 주는 노선 정보와 역정보가 담긴 구조체, 그리고 역과 그곳까지의 최소 거리를 저장하는 구조체 등으로 구성되어 있다. 더 설명하지 않은 구조체는 동적으로 공간을 할당받아 배열을 그리거나, 함수의 반환 값으로만 사용하는 구조체이다. 이때 이웃한 역 구별, 즉 노드는 역 목록에서 이웃하거나 각 역정보에 저장된 환승역 정보를 이용했다. 노선도 형태의 그래프는 정점에 비해 노드가 많지 않고 그것도 대부분이 한 노선상에서 이웃한 역이니, 간선의 수가 적다고 생각해 정점 정보를 역 목록에 담는 방법을 택했다. 다만 이 때문에 역정보를 전달할 때는 역만 전달해서는 안 되고 반드시 노선 정보와 같이 전달해야 간선(이웃한 역 목록)을 확보할 수 있다. 이것이 코드에서 Station 구조체의 존재 이유이고 많이 쓰이는 이유이다. 그리고 환승역 정보는 단순히 역 정보 내 Station 구조체의 배열을 둠으로써 저장했다. 동적 할당을 사용하지 않는 이 방법은 비 환승역에서 의미 없는 공간 낭비를 초래하는 방법이지만, 여기서 동적 할당을 쓰기에는 너무 복잡해지는 것 같아 조금 간단하게 코딩하고자 배열을 사용하는, 단순하지만 공간 낭비는 큰 방식을 이용했다(아마 Python으로 코딩했다면 고민하지 않고 list를 사용하지 않았을까 싶다). 또한 서로가 서로를 참조(StationTable은 아예 Station 구조체 배열을 가짐)하므로 구조체 선언 순서도 난잡하게 되었다.

파일 정보를 읽어오는 건 문제 1, 2에서 작성한 코드를 이용했다. 그리고 이를 '₩n' 문자를 기준으로 잘라 역 하나의 정보를 가져왔다. 각 데이터가 무슨 데이터인지를 나타내는 첫 줄은 읽고 버렸다. 이때 이 역정보에서도 ','를 기준으로 문자열을 분리해야 하는데, '₩n'을 기준으로 문자열을 분리하는 과정에서 strtok 함수를 사용해 ',' 기준 분리에 strtok 함수를 사용할 수 없다. 따라서 ',' 를 기준으로 분리하는 것은 직접 구현했다.

처음으로 문자열을 분리해 얻은 문자열은 노선 정보이다. 첫 입력이거나, 이전 역의 노선명과 다르면 노선 정보를 설정해 주고, 아니면 같은 노선임을 확인만 하고 넘어간다. 이때 노선이 바뀌면 역 목록을 저장하는 배열에 동적 할당을 진행해 주었다. 노선이 같지만 역 목록을 저장하는 배열에 역정보를 더 저장할 수 없게 되면, 재할당을 진행해 주었다. 두 번째와 세 번째로 분리되는 문자열은 역 번호와 역 이름이다. 각각 문자열의 형태로 넣어준다. 다음은 거리 정보인데, Excel로 숫자 정보를 저장하면 자연수로 나누어떨어지는 수는 소수점을 표기하지 않고 자연수의 형태로 표기하고, 아닌 수는 소수점을 표기하고 있어 각 상황에 대한 대비가 필요하다. 따라서 '.' 또는 ','을 기준으로 문자열을 자르고, 자른 문자열이 '.'이면 숫자를 더 읽도록 했다. 그리고 이 거리 정보는 파일에 0.1km 단위로 기록되는바, **프로그램 내 저장도 0.1km 단위로, 정수형으로 진행**하였다. 즉 실제 거리는 저장된 값에서 10을 나누어야 한다. 마지막으로 환승 정보이다. 거리 정보 이후 ','가 연속해서 들어오면 환승역이 아니라는 의미이므로 넘어가고, 환승역의 역 번호를 받으면 지금까지 저장된 노선 중 그 역 번호를 가진 역을 찾는다. 찾았다면 그 역을 Station 구조체의 형태로 만들어 역정보에 넣는다. 또한 그 역의 환승역 정보에도 이 역을 추가한다. 이는 아직 저장되지 않은 역이 환승 정보로 들어올 수 있는데, 이때는 그냥 넘기고, 다음에 그 역이 환승 정보를 가지고 오면 이전에 추가된 역은 환승 정보가 저장되어 있지 않으므로 이때 추가를 해 주는 것이다.

이렇게 모든 역, 모든 노선에 대해 반복하면 역정보 저장은 끝났다. 이제 시작 역과 끝 역을 입력받고, 역 간 거리를 구한다. 역 간 거리를 구하기 전 역 목록에서 입력한 역을 찾는데, 환승역이라 같은 역이 여러 번 등록된 건 상관없지만 신촌역같이 이름은 같지만, 다른 역이 있을 수 있다. 처음에는 이때 사용자에게 원하는 게 무엇인지 선택 창을 띄우려 했으나, 입력을 받는 게 더 어려워 보여 모든 동명이역에 대한 결과를 가져오도록 했다. 따라서 역을 찾을 때 배열의 형태로 가져와야 하고, 이 배열을 넘기기 위해 StationSearchRes 구조체가 존재하게 되었다.

역간 최단 거리를 찾을 때는 기본적으로 Dijkstra의 알고리즘을 사용하였으나, 최적화를 위해 한 쪽에서 가는 게 아니라 시작점과 끝점 양쪽에서 출발해 중간에 만나도록 했다. 또 Dijkstra 알고리즘은 그룹으로 묶는 개념이 존재하는데, 그룹에 넣기는 하나 빼지는 않으므로 스택 개념을 사용했다. 역간 최단 거리를 구할 때는 5가지의 그룹이 필요하다. 각각 시작 지점에서 출발하는 그룹, 이 그룹에 속한 노드와 직접적으로 연결된 노드의 그룹, 끝 지점에서 출발하는 그룹, 이 그룹에 속한 노드와 직접적으로 연결된 노드의 그룹, 새롭게 그룹에 포함된 역 주변의 노드 그룹(임시 변수로 사용)이다. 먼저 마지막을 제외한 4개 그룹을 초기화해 준다. 환승역을 하나의 노드에 담은 게 아니라 각 포인터를 역정보가 들고 있기 때문에, 시작 역이 환승역이면 환승역을 전부 추가해야 한다. 그렇지 않으면 다른 노선의 역은 그룹에 포함되지 않는 별개의 역으로 처리된다. 또 직접적으로 연결된 노드의 그룹은 앞에서 추가한 노드 전부에 대해 조사를 진행한다.

이제 시작 지점에서 출발한다. 시작 지점 출발 그룹과 인접한 그룹을 거리 순으로 정렬하고, 거리가 가장 작은 역을 골라 그룹에 넣는다. 이때 뽑은 역이 끝 시작 그룹에 포함되면, 두 거리의

합을 최종 거리로 하고 연산을 종료한다. 그렇지 않으면 그 역과 인접한 노드와 그 거리를 모두 구해 인접 그룹에 넣는다. 이미 시작 지점 출발 그룹에 있는 역이면 추가를 진행하지 않고, 인접 그룹에 있는 역과 같은 역이 나왔다면 그룹에 넣는 대신 기존 저장 값과 새로 구한 값 중 짧은 값으로 업데이트한다. 이때 이미 그룹에 있는 역인지는, 그 역이 환승역이면 환승역 정보도 전부 찾아봐야 한다. 이렇게 하지 않으면 같은 역이 다른 거리로 그룹에 두 번 들어가는 현상이 일어나고, 이 그룹 내로 들어오면 중복 확인을 하지 않기 때문에 최종 결과가 실제보다 더 커질 수 있다(같은 역, 거리가 더 큰 게 나중에 인접 그룹에서 가장 거리가 작아져 뽑혔는데 마침 끝 시작 그룹에 그 역이 포함된 사례 등) 이를 시작과 끝을 바꾸어 반복한다. 연산 결과가 나오면 동적할당 받은 공간을 해제하고 결과를 반환한다. 이때 거리를 0.1km 단위로 저장했으니 출력 시에도 다시 이를 맞추어준다.

이외에 인접 역의 거리를 구하거나 스택 Push를 하는 함수도 만들었지만 특별한 건 없다. 스택 Push 함수에서 스택의 길이가 부족하면 재할당을 진행하고, Push 대상 객체에 이상이 있으면 과정을 진행하지 않는다는 것 외에는 특이 사항이 없다.

위의 과정으로 문제 5를 해결했으며, 그 결과는 다음과 같다. 경로는 요구사항이 아니므로 출력하지 않았다.

14. 문제 5의 결과

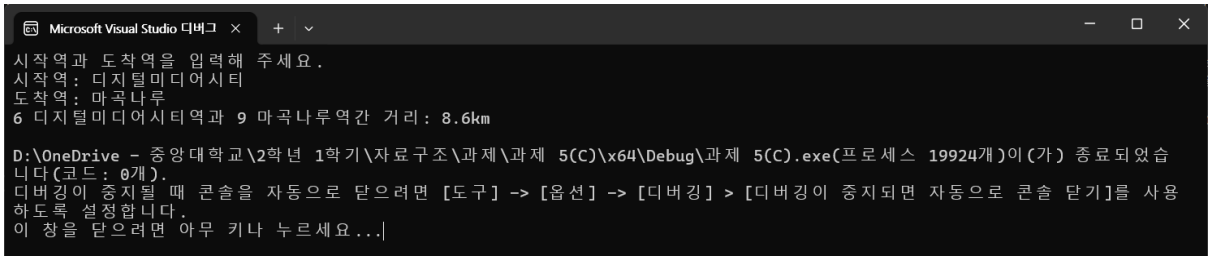

그림 14.1

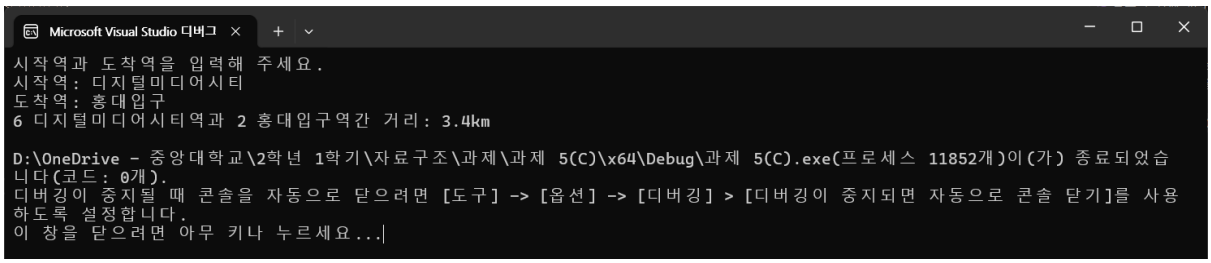

그림 14.2

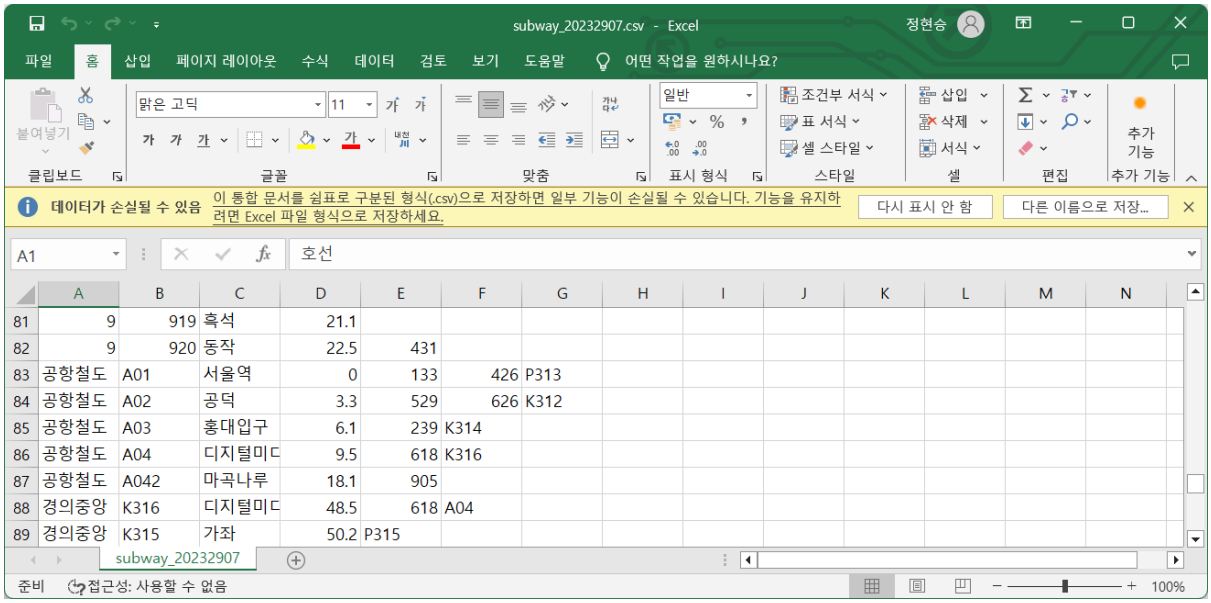

그림 14.3

그림 14.1, 14.2는 단거리에 대해 동작을 확인해 본 결과이다. 그림 14.3이 실제 파일에 저장된 내용이므로 정상 동작함을 알 수 있다.

이번에는 먼 거리의 역 간 거리를 표시해 보았다.

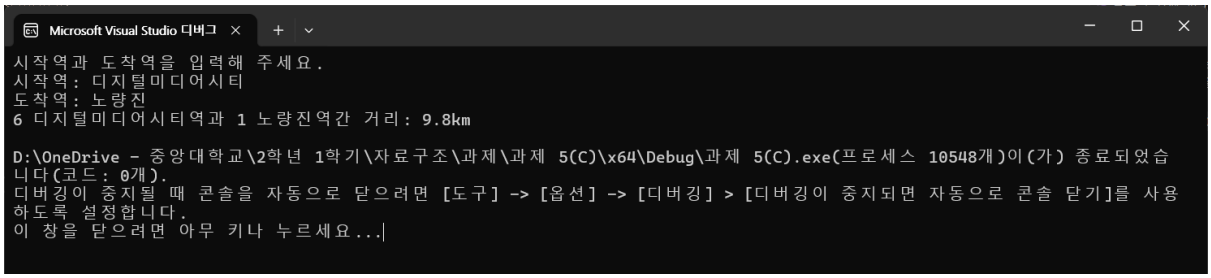

그림 14.4

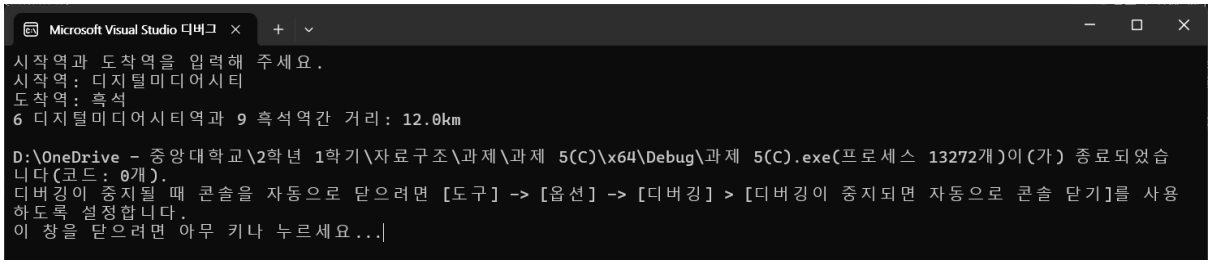

그림 14.5

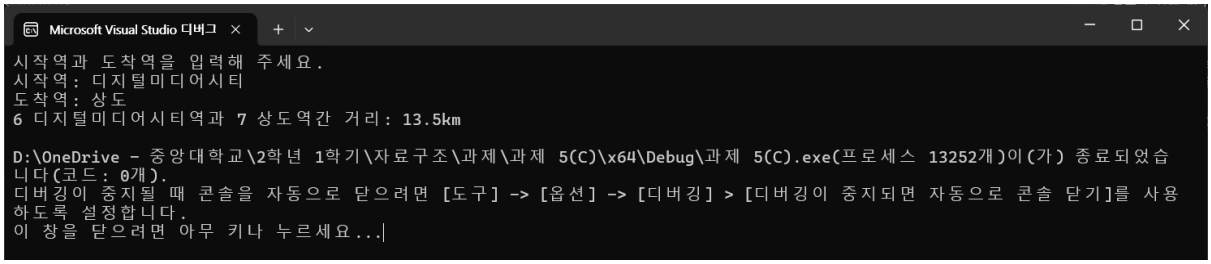

그림 14.6

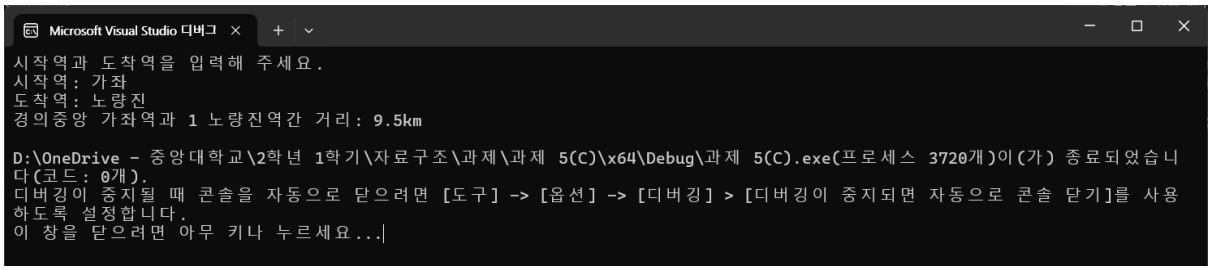

그림 14.7

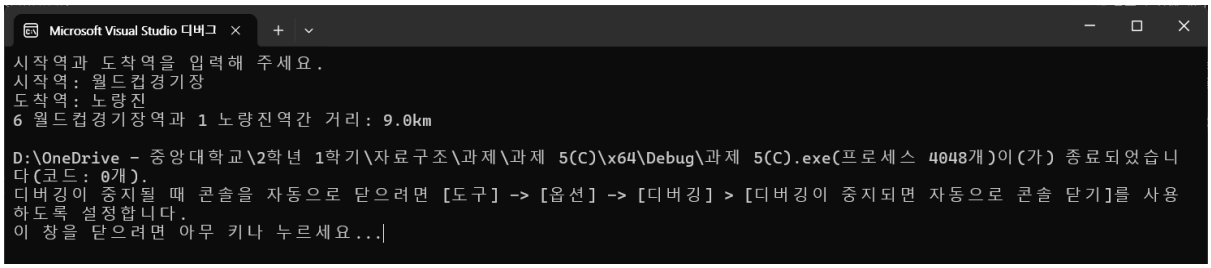

그림 14.8

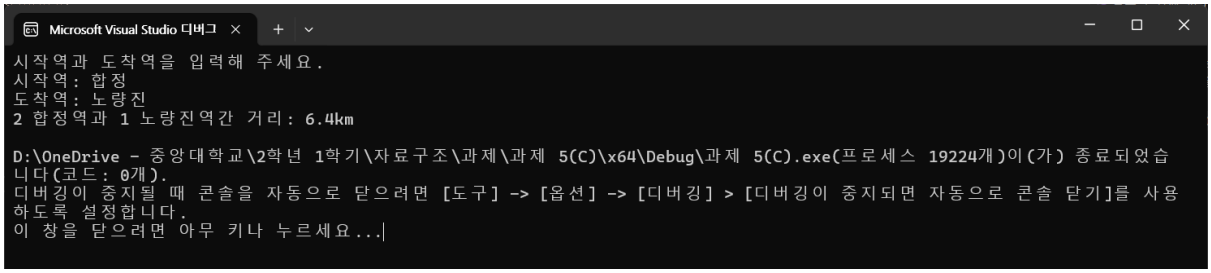

그림 14.9

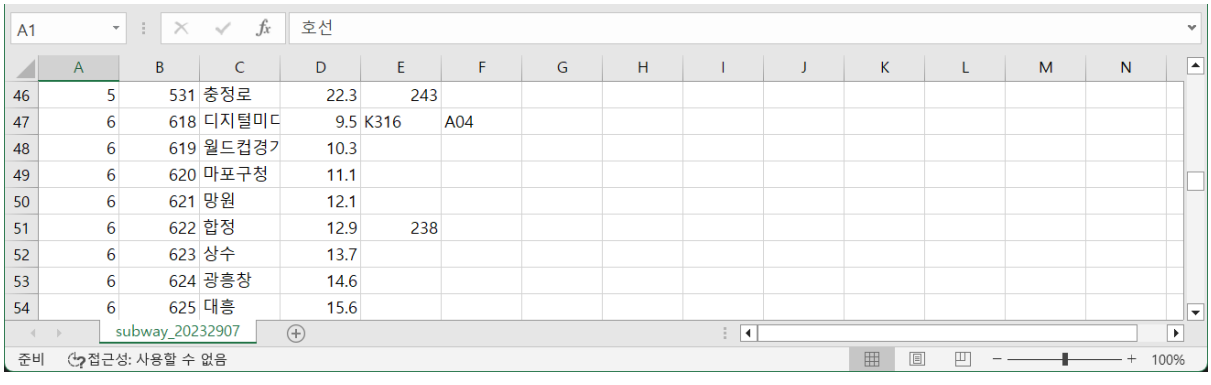

그림 14.10

위 그림 14.4~그림 14.9는 비교적 멀리 떨어져 있는 역 간 거리를 계산한 결과이다. 그림 14.10에서 6호선 역 간 거리를 보면 잘 계산하고 있다는 것을 알 수 있다.

위와 같이 코드가 정상 동작함을 알 수 있다. 또한 그림 14.4, 그림 14.7~14.9의 일관성 있는 결과를 보면 디지털미디어시티역에서 학교(노량진, 흑석, 상도)까지 최단 거리는 6호선 기준으로 계산되므로, 앞으로 학교 쪽으로 나가서 지하철을 탈 때는 가좌역이 아닌 월드컵경기장역으로 가야 한다는 점도 알게 되었다.

15. 소스코드

첨부파일 참조

---

[i] Jun,「[python] 예외처리, 파일처리, 모듈」,『네이버 블로그』, 2020.11.9. (https://m.blog.naver.com/dnjswns2280/222139407143)

[ii] cra4ckerjack,「파이썬 (Python) - 깊은 복사 (Deep Copy)」,『tistory』, 2020.6.26. (https://crackerjacks.tistory.com/14)

[iii] https://data.seoul.go.kr/dataList/OA-12034/S/1/datasetView.do

[iv] https://www.data.go.kr/tcs/dss/selectDataSetList.do?dType=FILE&keyword=%EA%B5%AD%EA%B0%80%EC%B2%A0%EB%8F%84%EA%B3%B5%EB%8B%A8+%EC%97%AD%EA%B0%84%EA%B1%B0%EB%A6%AC&operator=AND&detailKeyword=&publicDataPk=&recmSe=&detailText=&relatedKeyword=&commaNotInData=&commaAndData=&commaOrData=&must_not=&tabId=&dataSetCoreTf=&coreDataNm=&sort=_score&relRadio=&orgFullName=&orgFilter=&org=&orgSearch=&currentPage=1&perPage=5&brm=&instt=&svcType=&kwrdArray=&extsn=&coreDataNmArray=&pblonsipScopeCode=