

|  |
| --- |
| 과제 3 |
|  |
| 4월 16일  소프트웨어대학 소프트웨어학부  작성자: 20206861 이훈의, 20222663 권기영 |

(1) 미로 탐색 하기

|  |
| --- |
| 문제 내용  미로의 길을 찾는 프로그램을 작성해라. push와 pop을 할 때마다 그에 대한 정보를 출력하고, 길을 찾은 경우 길을 출력하고, 길이 없는 경우 길이 없음을 출력해라..  해결 방안 – 1) 미로 저장 방식  과제 1에서 저장한 방식은 메모리 절약 방면에서는 효율적이었으나, 이를 운영하기에는 너무 비효율적이었다. 따라서 메모리에 제한이 없다면, 텍스트 파일에 저장된 미로를 그대로 읽어와 저장하는 것이 가장 운영하기 좋다고 판단하였다. |

해결 방안 – 2) 길 찾기 방식

Available 함수를 만들어 갈 수 있는 방향의 개수를 반환하도록 했고, Available의 반환값이 1 이상이면 동서남북 순으로 이동하고, 현재 위치를 push한다. Available의 반환값이 0이면 길이 없으므로Available이0이 아니게 될 때까지 pop하여 되돌아간다.

또한 길 방문 여부를 저장하는 이차원 배열 visited를 생성하여 방문하지 않았다면 -1, 방문했다면 0을 저장하는 방식으로 길을 방문했는지 여부를 판단한다.

push를 하면 미로를 저장하는 배열 Maze에 길을 표시하고, pop을 하면 표시했던 길을 다시 지움으로써 최종적으로 Maze를 출력하면 길이 출력되도록 하였다.

실행 결과

<길이 있는 경우>

텍스트이(가) 표시된 사진

자동 생성된 설명

<길이 없는 경우>

텍스트이(가) 표시된 사진

자동 생성된 설명

(2) 큐 운영하기

|  |
| --- |
| 문제 내용  원형 큐를 만들어 사용자로부터 문자를 입력 받고, 1초에 문자를 한 개씩 큐에서 가져온다. 입력은 비동기 입력으로 이루어진다.  해결 방안 – 1) 비동기식 입력의 구현  main 함수에서 while 루프 안에 \_kbhit() 함수로 콘솔 입력 버퍼에 입력된 문자가 있는 지의 여부를 반환 받는다. 만약 입력 받은 문자가 있다면 \_getch() 함수를 통해 char형 변수 input으로 문자를 받는다. 그리고 받은 문자 input을 Enqueue 함수를 통해 큐에 저장한다. 이때 소문자를 입력 받더라도 큐에는 대문자로 들어갈 수 있도록 toupper를 통해 큐에 추가한다. 이러한 입력 방식을 채택하여 입력된 문자가 없더라도 프로그램은 계속 실행 될 수 있다. |

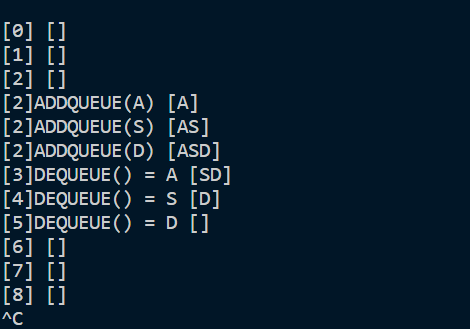
해결 방안 – 2) 1초마다 이루어지는 출력

main 함수에서 1초씩 흐르는 시간을 계산할 int 형 변수 time을 할당한다. Sleep()함수를 통해 스레드를 100ms 동안 일시정지 시킨다. 그리고 그 때마다 time에 100의 값을 더해준다. 1초가 지났을 땐 while 루프가 10번 순회했을 것이고 time 값은 1000이 되었을 것이다. 이때 time 변수에 대한 조건문을 활용하여 큐에 아무것도 없다면 “[시간] [ ]” 의 형태로 큐가 비어있음을 표현하고, 큐에 문자가 존재한다면 Dequeue 함수를 통해 큐에 존재하는 문자를 한 개 제거하고 이를 출력하도록 구현했다.

해결 방안 – 3) 큐의 상태 출력

Enqueue 함수가 실행되며 큐에 문자가 추가 되면, 추가된 데이터를 ADDQUEUE로 화면에 따로 출력하고, for문으로 큐를 front부터 rear까지 훑어서 큐 안의 남아있는 문자들을 모두 출력하도록 했다.

Dequeue 함수가 실행되며 큐에서 문자가 제거 되면, 제거의 대상이 되는 문자가 제거 되기 전에 별도의 char 변수를 가지고 따로 저장해놓은 뒤에 이를 DEQUEUE로 화면에 따로 출력하고, 큐에 남은 문자들을 똑같이 for문을 통해 front의 한 칸 앞부터 큐를 출력하도록 했다.

실행 결과

입력 받은 문자는 ADDQUEUE(%c)의 형태로 출력되고, 제거 되는 문자는 DEQUEUE() = %c의 형태로 출력됨을 확인할 수 있다. 또한 큐에 남은 문자들도 정확하게 출력되고 있다.

(3) 연결 리스트 – 사전 만들기

|  |
| --- |
| 문제 내용  Randdict 텍스트 파일에 존재하는 약 5만개의 단어와 단어 뜻을 연결 리스트를 사용하여 알파벳 순으로 정렬한다. 단어를 검색하는 기능과 검색한 단어가 없다면 뜻을 추가 할 수 있는 기능을 구현해야 한다.  해결방안 – 1) Node 구조체, Dictionary 구조체 선언  문자열 word와 meaning을 멤버로 하는 구조체 Dictionary를 선언하였다. 그리고 Dictionary자료형의 data와 next를 멤버로 가진 자기참조 구조체 Node를 선언하였다.  해결 방안 – 2) load\_data, add\_node 함수를 통해 단어 정렬하기  Randdict.txt 파일에 존재하는 단어와 그 뜻은 “:”로 구분되어 있다. 그래서 strtok 함수를 활용해 “:”를 토큰으로 설정하여 단어와 뜻을 구분하려 받으려고 했으나, : 앞뒤에 공백 문자가 있는 문제로 인하여 fgetc로 문자를 하나씩 문자열에 추가하는 방식으로 구현하였다.  load\_data에서는 텍스트 파일의 데이터들을 한 줄씩 읽어서 string 배열에 저장한다. 그리고 배열에 저장된 문자들을 “:”를 전후로 하여 “:”의 앞은 data.word 배열에 저장하고, “:”의 뒤는 data.meaning 배열에 저장하여 단어와 단어 뜻을 구분했다.  add\_node 에서는 단어를 알파벳순으로 정렬하여 연결 리스트에 삽입하는 함수이다. 연결리스트의 첫 노드부터 마지막 노드까지 strcmp함수를 통해 단어 순서를 비교하고, 적합한 위치에 삽입하도록 구현하였다. |

해결 방안 – 3) print\_meaning - 단어의 뜻 출력하기

print\_meaning 함수는 탐색할 단어를 인자로 받아서 연결리스트의 head 부터 null까지 하나하나 비교해 가면서 탐색한다. 또한 사전에 단어 포함 여부를 판단하는 전역변수 check를 선언하여 단어를 찾으면 그 뜻을 출력하고 check = 0; 탐색이 끝났는데도 단어를 찾지 못한다면 check = 1;로 설정한다.

해결 방안 – 4) 단어의 뜻 추가하기

이는 함수를 따로 만들지 않고 main함수 내에서 구현하였다.

앞서 해결 방안 2)에서 사용자가 찾고자 하는 단어가 연결 리스트에 존재하지 않는다면(check ==1), 새로운 단어의 뜻을 입력 받는다. 사용자로부터 단어의 뜻을 입력 받으면, 앞서 입력 받았던 새로운 단어와 그 뜻을 저장할 new\_data 구조체를 선언하고, new\_data에 단어와 뜻을 저장한다. 그리고 add\_node 함수를 사용해 새로 추가한 단어가 정렬되어 사전에 저장된다.

사용자가 공백 문자를 입력하면 단어의 뜻을 추가하지 않는다.

해결 방안 – 5) 대소문자 문제 해결

사용자가 단어를 입력할 때 대소문자를 신경 쓰지 않고 입력한다는 가정이 있기 때문에, 대소문자에 관한 처리를 따로 해주어야 한다.

정렬을 할 때에는 대문자와 소문자가 구별되어 정렬된다. 즉, A로 시작하는 단어가 가장 앞쪽에 삽입되고, Z로 시작하는 단어 다음에 a로 시작하는 단어가 삽입된다는 뜻이다.

그러나 탐색을 진행할 때는 입력 받은 데이터의 대소문자 구별이 없어야 한다. 따라서 strcasecmp 함수를 활용하여 대소문자의 구별 없이 비교를 진행하였다.

해결 방안 – 6) 배열과 연결 리스트 정렬 시간 비교하기

문자열 word와 meaning을 멤버로 가지는 구조체 배열 data를 만들어서, 이 구조체 배열에 텍스트 파일의 단어와 뜻을 전부 저장하고, 선택 정렬을 통해 정렬을 진행하였다.

그리고 연결 리스트를 통해 구현한 사전의 정렬 시간과 배열을 통해 구현한 사전의 정렬 시간을 비교했더니, 연결 리스트 사전은 약 5.8초, 배열 사전은 약 4.1초 정도 소요되었다.

실행 결과

텍스트이(가) 표시된 사진

자동 생성된 설명

(4) 연결 리스트 – 사전 개선하기

문제 내용

(3)에서 연결리스트로 구현한 사전에 단어를 찾거나 추가할 때, 5만개의 링크를 따라가는 것은 매우 비효율적이다. 어떻게 하면 좀 더 빠르게 원하는 단어를 찾을 수 있을까?

개선 방법 : 첫 번째 알파벳을 기준으로 26개의 연결리스트를 각각 운영한다.

해결 방안 – 1) 초기 선언

26개의 연결리스트의 head를 구조체 배열로 선언한다. 예를 들어, head[0] 은 첫 알파벳이 ‘a’인 단어가 모여 있는 연결리스트의 헤드이다. a~z 로 시작하는 단어를 저장하는 26개의 연결리스트를 운영하기 위하여 head도 26개를 생성하였다.

해결 방안 – 2) load\_data, add\_data 함수를 통해 단어 정렬하기

(3)의 방법과 단어를 입력 받는 방법은 같지만, 이를 삽입하는 방식을 개선했다.

(3)의 방법은 단어를 삽입할 때, 전체 단어가 연결 되어있는 하나의 연결리스트에 삽입했다. 개선된 방식은 단어의 첫 번째 알파벳을 확인하고, 26개의 연결 리스트 중 적합한 연결리스트에 삽입한다. 처음부터 순차적으로 비교해서 정렬하는 방식은 (3)의 방식과 동일하다.

add\_data의 인자로 a~z중 어떤 연결리스트인지를 나타내는 인덱스 값을 전달함으로써 함수 안에서는 26개 중 적합한 연결리스트 하나에만 접근한다.

해결 방안 – 3) print\_meaning – 단어의 뜻 출력하기

print\_meaning 함수도 동작 방식은 (3)과 같으나, 매개변수로 head를 받을 때 head[index], 즉 적합한 연결리스트의 헤드를 받아서 함수 안에서는 적합한 연결리스트에만 접근한다.

해결 방안 – 4) 대소문자 문제 해결

(3)번에서는 정렬할 때는 대소문자를 구별하여 정렬했지만, 개선된 구조에서는 정렬할 때 부터 strcasecmp함수를 활용하여 대소문자를 구별하지 않고 정렬하였다.

그 이유는 단어의 첫 알파벳에 따라 26개의 연결리스트만을 운영할 계획인데, 대소문자가 구별되면 이것이 불가능해지기 때문이다.

또한 첫 번째 알파벳이 대문자이면 word[0] – ‘A’ 를 인덱스로 가지는 헤드에 접근하고, 소문자라면 word[0] – ‘a’ 를 인덱스로 가지는 헤드에 접근하도록 처리하였다.

해결 방안 – 5) 기존 방식과 개선된 방식의 시간 비교

기존 방식은 단어를 입력 받아 정렬하는 데에 약 5.8초가 소요되었고, 개선된 방식은 약 0.34초 정도 소요되었다.

정렬 알고리즘을 바꾸지 않고도 연결리스트의 구조를 바꾼 것 만으로 시간이 많이 줄어든 이유는 다음과 같다.

1) 단어의 개수가 많다.

단어의 개수가 적다면 이런 식으로 나누어서 정렬하는 것이 큰 효과가 없고, 오히려 불필요한 동작만 늘어날 수 있다. 그러나 텍스트 파일에는 약 48000개의 많은 단어가 있었기 때문에 나누어서 정렬하면 한번 삽입할 때 마다 걸리는 시간을 효과적으로 줄일 수 있다.

2) 알파벳별로 단어가 고루 분포되어 있다.

이 방식의 worst case는 모두 똑같은 알파벳으로 시작하는 것이다. 그렇다면 결국에는 26개의 연결리스트 중 하나만 운영하게 될 것이고, 소요시간은 줄어들지 않는다.

best case 는 26개의 연결리스트에 동일한 개수의 단어가 정렬되는 것이다. 그렇게 되면 약 26배 빠르게 수행할 수 있다. 실제로 계산을 해보면 소요 시간이 약 0.22초가 나온다.

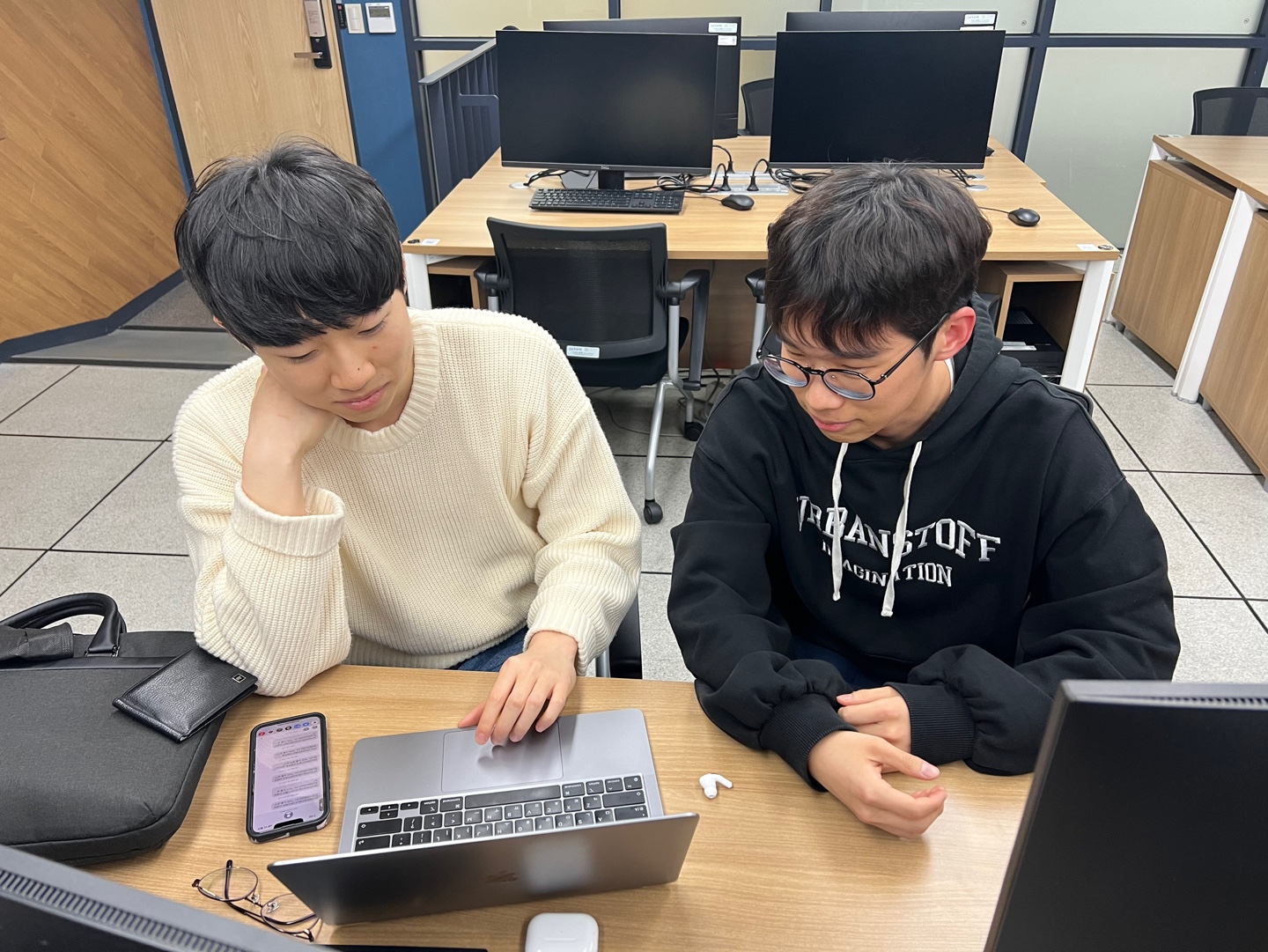
이 사전의 경우에는 소요 시간이 0.34초임을 고려해봤을 때, 알파벳이 한 곳에 치우쳐져 있지 않고 고르게 분포되어 있기 때문에 best case에 근접하게 결과가 나온다.

\*\* 단어를 입력받아 정렬하는 시간만 측정하였고, 단어를 검색하는 시간은 측정하지 않았는데, 그 이유는 단어를 정렬하는 과정과 단어를 검색하는 과정이 거의 비슷하기 떄문이다 두 과정 모두 연결리스트의 head부터 시작해서 원하는 값이 나올 때 까지 이동하는 원리로 작동한다. 그렇기에 단어를 검색하는 소요시간 또한 단어 정렬과 비슷한 효율로 줄어들었을 것이라고 생각한다.

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명



1, 3, 4번의 실행 환경은 mac OS VScode이고, 2번의 실행 환경은 window VScode이다.