HW4 결과보고서

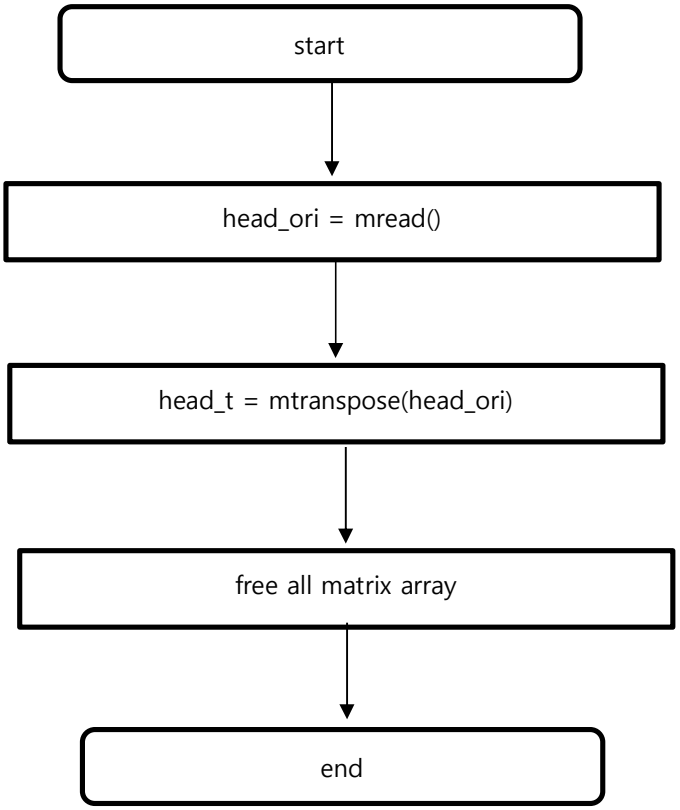
20211522

김정환

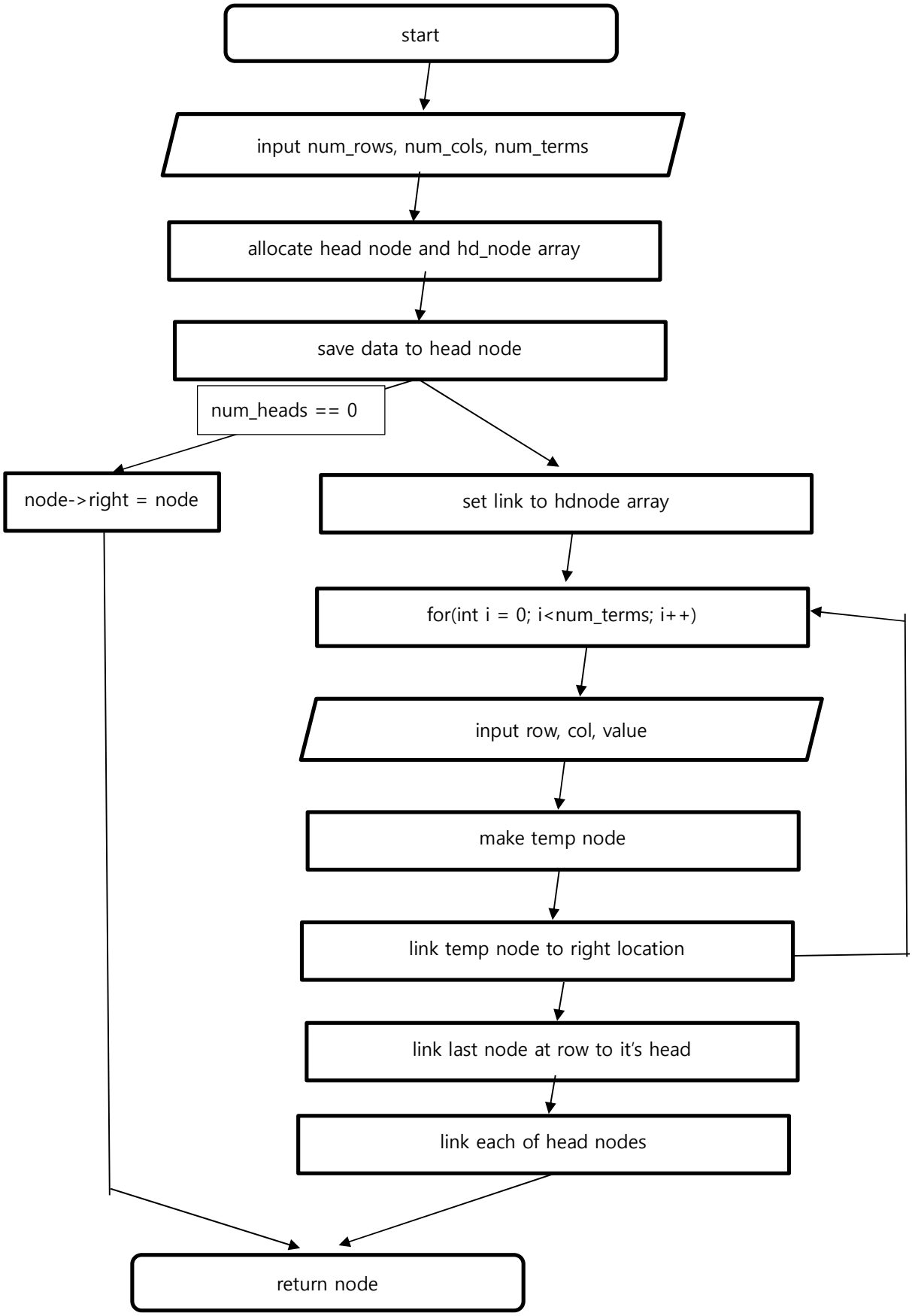
1번 문제

<flow chart>

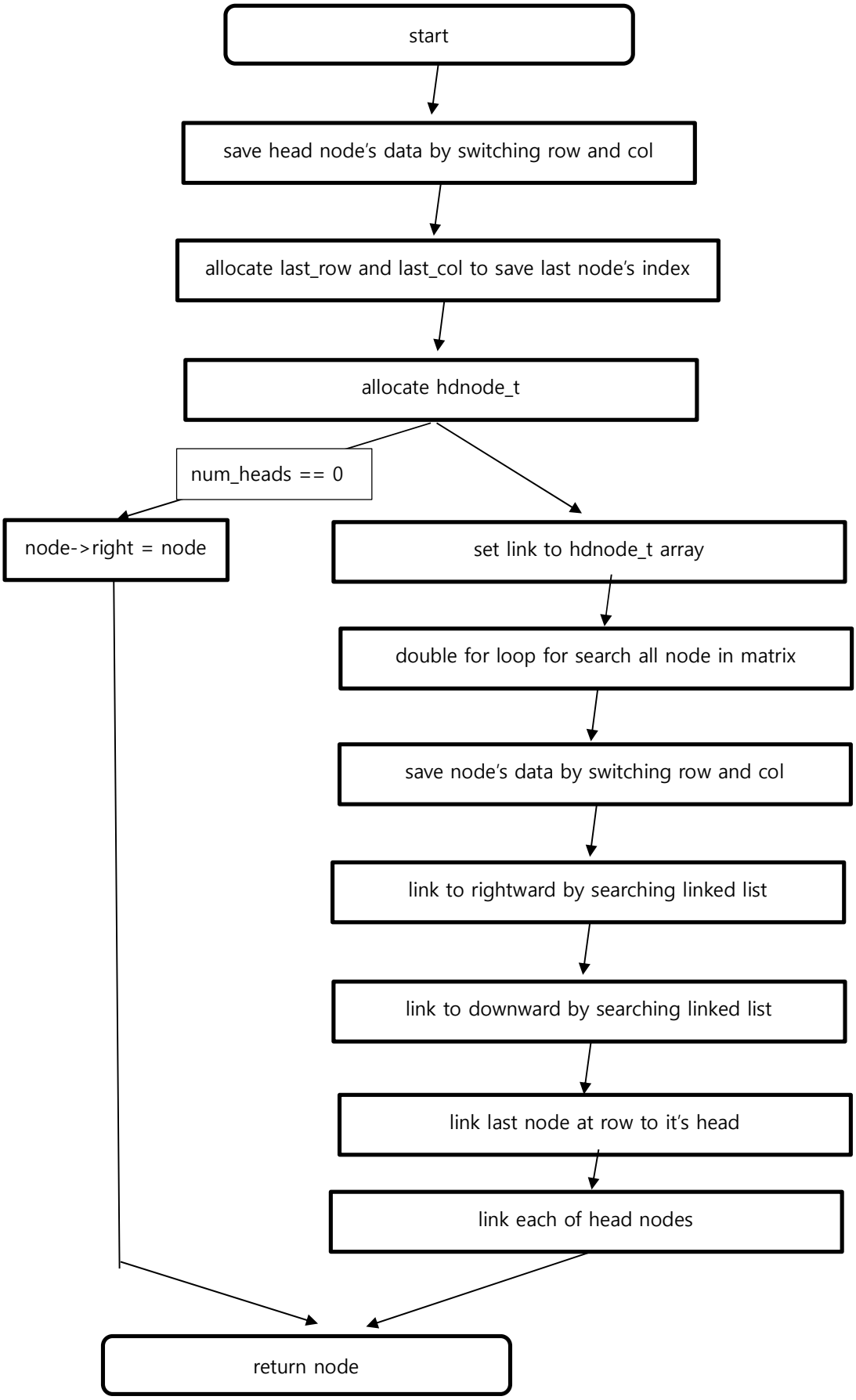
-int main()



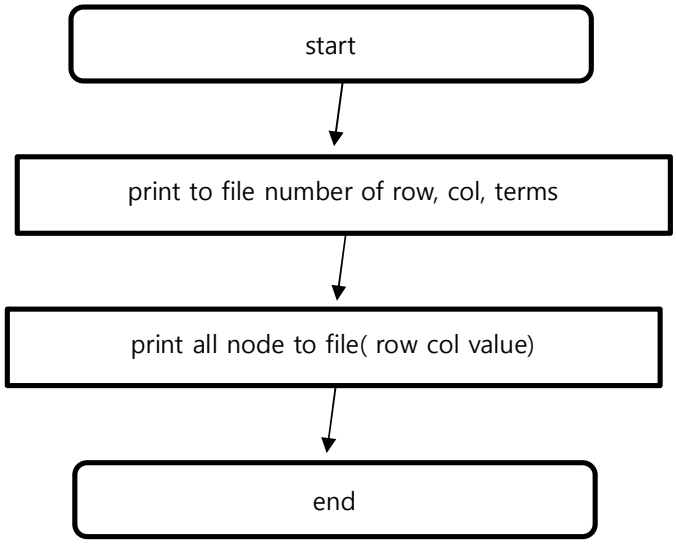
-matrix_pointer m_read()



-matrix_pointer mtranspose(matrix_pointer head1)



-void mwrite(matrix_pointer node)



<구현 설명>

우선 matrix_node의 struct나 node의 struct의 구조, 그리고 mread()와 mwrite()는 강의자료를 참고하여 구현하였다. 강의 자료와 달라진 점으로는 File input과 ouput을 활용한 점에서의 차이만 있다. 구현에 있어서 가장 주요했던 matrix_pointer mtranspose(matrix_pointer head1) function에 대해 자세히 설명하자면 우선 argument로 head1인 이유는 enum으로 인해 variable name이 겹쳐 그것을 방지하고자 사용하였다. function의 시작 부분에서 transpose matrix의 전체 head node인 node variable을 declare하고 해당 node에는 head1에 있는 value들을 넣어주었다. 단 transpose이므로 row와 col의 value는 서로 switch하여 넣었다. mread()와 유사하게 num_heads에는 row와 col중에서 더 큰 value를 넣어주었다. 또한 last_row, last_col variable을 declare하여 transpose 시에 linking을 진행하면서 마지막 node의 index를 저장할 array를 allocate하였다. 이 array들은 모두 value를 -1로 initiating하였다. 그 후 matrix search를 위한 variable ptr_temp와 새로운 node 생성 시 사용할 temp variable을 declare하였다. num_heads가 0인 경우에는 mread()와 같이 node의 right에 node를 link하고 종료한다. 그 외에는 우선 hdnode_t에 각각 head node들을 생성하여 넣어주었다. 그 후 mwrite()에서 search할 때와 똑같이 for loop를 구성해주고, 각각의 node의 경우마다 우선 temp에서 새로운 node를 allocate해주고, value들을 row와 col을 switch하고 data를 넣어준다. 그 후 list에 link하기 위해서 row와 col에 대해서 같은 방법을 이용하는데, 우선 해당 row나 col에서 last_col이나 last_row가 -1인 경우는 아직 아무 node도 연결되지 않은 것이다. 해당 경우에는 각각의 head node의 right 또는 down에 link하고, 새로운 node의 right이나 down에는 head node를 link한다. 그 후 last_col이나 last_row의 value를 update해준다. 다음으로는 새로운

node가 마지막 node가 되어야 하는 case이다. 해당 case에는 cur를 마지막까지 이동시키고, 해당 지점에서 temp의 right 또는 down은 cur의 right 또는 down이 되도록 하고 cur의 right는 temp가 되도록 한 후 last_col이나 last_row를 update 한다. 그 외의 case에는 temp의 col이나 row가 cur보다 작을 때까지 while loop로 search하면서 pre와 cur에 저장한다. 그 후 새로운 node의 right 또는 down에는 cur, pre의 right또는 down에는 temp가 되도록 한다. 이처럼 모든 node를 돈 후에는 마지막 node들의 down에 head node를 link한다. 그 후 hdnode_t의 element들끼리 link한 후 필요 없는 last_row, last_col을 free해준 후 node variable을 return한다. main function에서는 head_ori에 mread()의 결과를, head_t에 mtranspose(head_ori)의 결과를 저장한다. 그 후 mwrite(head_t)를 통해 파일에 print한다. 그 후에 node의 전체 searching을 응용하여 del variable에 이전 node를 저장하여 free로 memory를 해제한다. 그 후 hdnode와 hdnode_t의 memory도 해제하며 main function이 종료된다.

다음으로는 결과에 대한 확인이다.

첫 번째로 주어진 자료에 있는 case에 대해서 확인해 보았다. 결과를 보았을 때 변환도 잘 되었고 txt file에 저장하는 것 역시 잘 진행되었음을 확인할 수 있다.

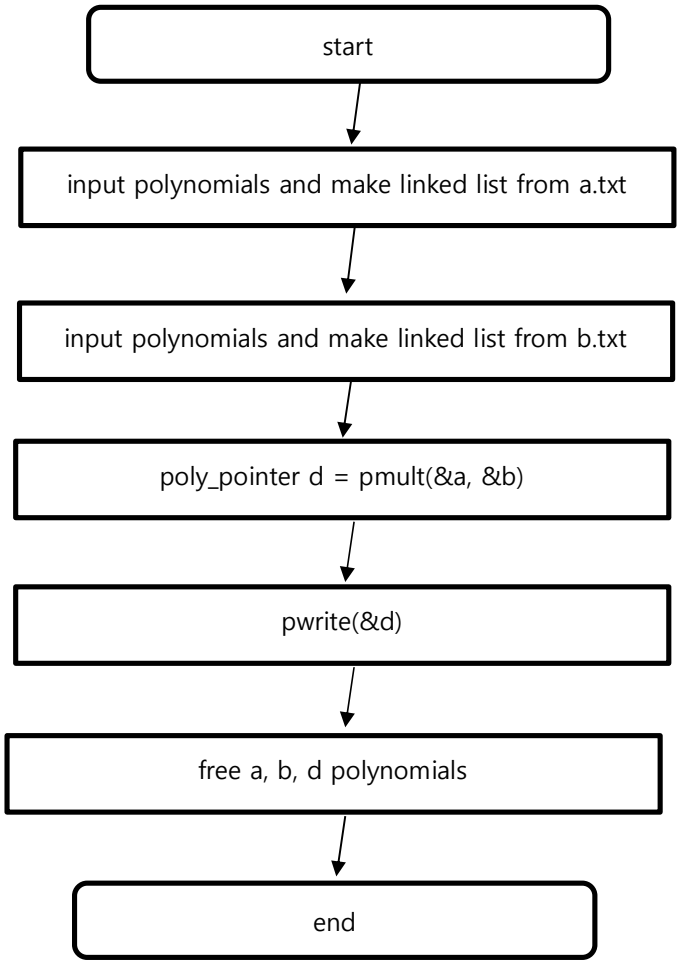
```
cse20211522@cspro:~/HW4$ cat input.txt
6 5 2
0 2 11
3 3 -15
cse20211522@cspro:~/HW4$ ./test1
cse20211522@cspro:~/HW4$ cat output.txt
5 6 2
2 0 11
3 3 -15
cse20211522@cspro:~/HW4$
```

다른 case에 대해서도 test 해본 결과 문제 없이 나오는 것을 확인할 수 있다. algorithm의 efficiency에 있어서는 매 node에 있어서 link할 지점을 일일이 search하므로 효율적이지는 못하다고 평가할 수 있을 것이다. 다른 방법이 있다면 좀 더 효율적으로 개선할 방향의 여지가 남아있다.

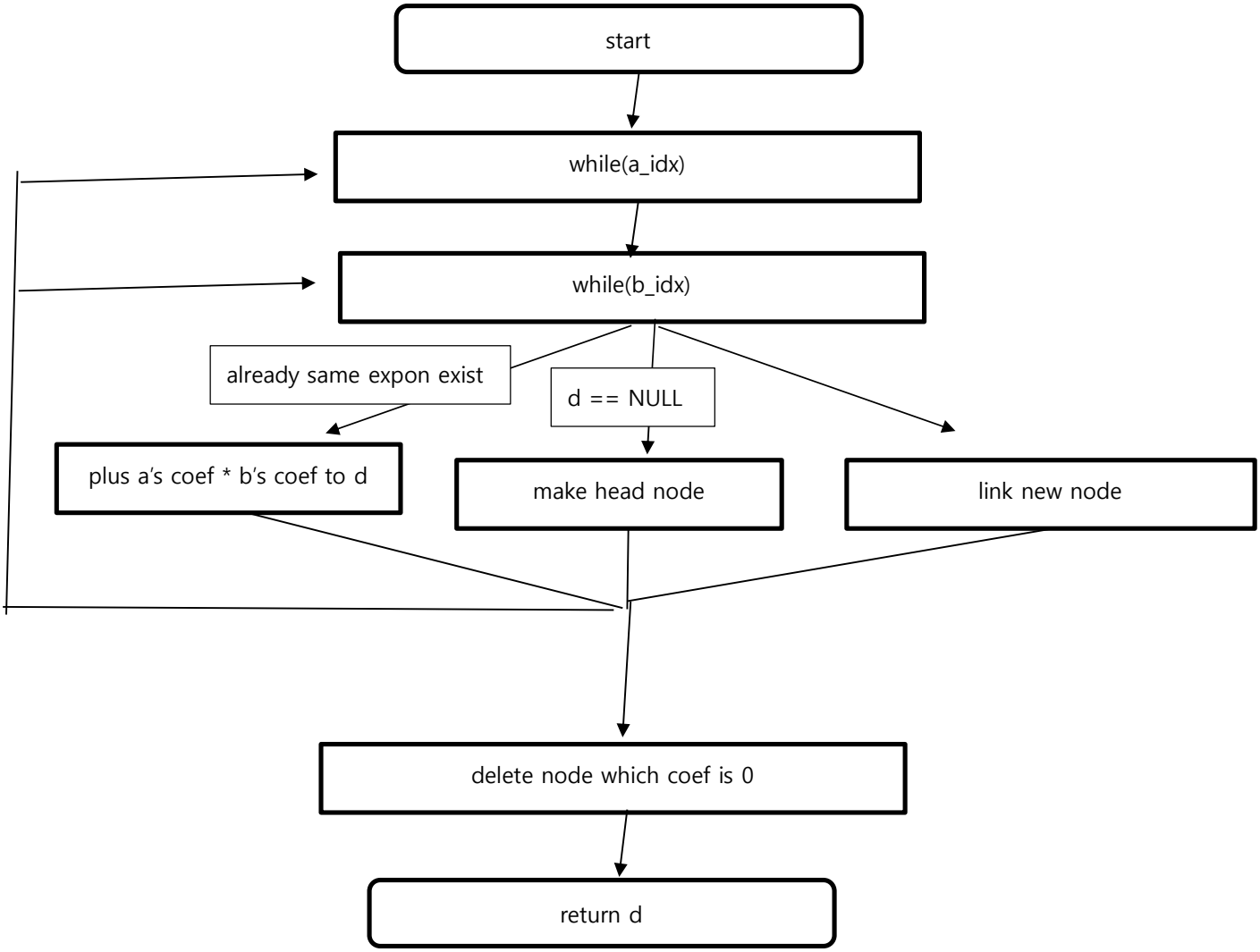
2번 문제

<flow chart>

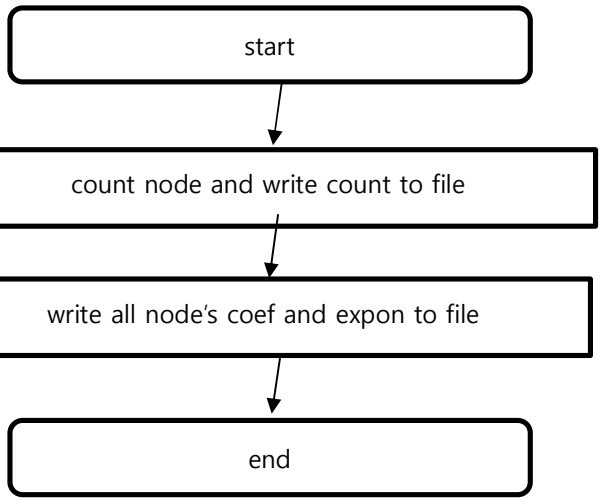
-int main()



-poly_pointer pmult(poly_pointer a, poly_pointer b)



-void pwrite(poly_pointer d)

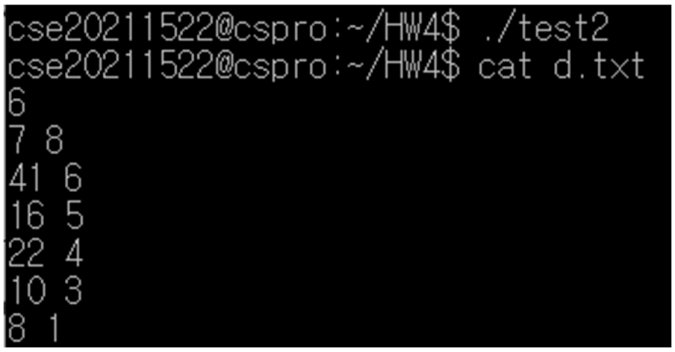


<구현 설명>

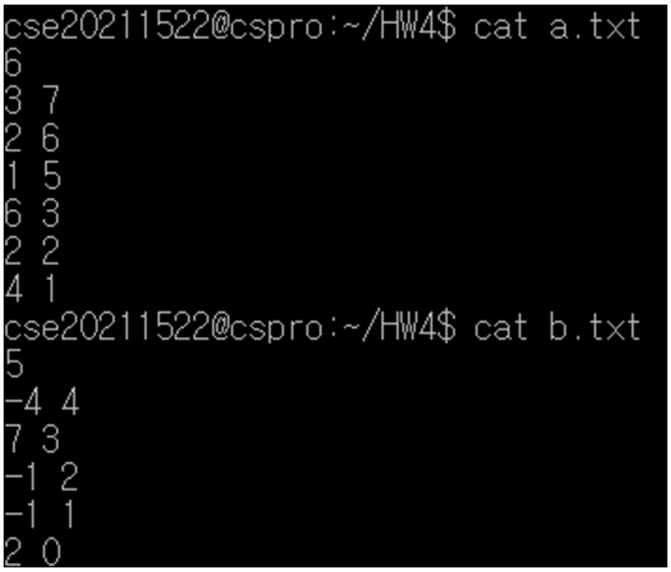
node 구현을 위한 struct는 강의자료를 참고하여 구현하였다. 우선 poly_pointer pmult(poly_pointer a, poly_pointer b) function은 function 내에서 local variable인 poly_pointer type인 a_idx, b_idx를 이용하여 a와 b polynomial을 search하도록 했다. 2중 while loop를 통해서 search하였고, a_idx 쪽의 loop 한 번 당 b_idx는 *b로 initialize 해주어 모든 경우의 수를 search했다. 모든 경우의 수에 대해서 우선 flag를 통해서 이미 같은 expon을 가진 node가 있는지 나타낸다. 같은 expon을 가진 node가 있을 경우, 해당 node에 coef의 multiple을 더해주고 flag를 1로 바꿔 표시한다. 이 경우 밑의 새로운 node를 만드는 부분을 거치지 않는다. 없는 경우에는 d에 첫 node인지 아닌 지로 구분하였다. d의 첫 node인 경우에는 d에 allocate해서 data를 넣어주었다. 아닌 경우에는 temp에 node를 allocate하였고, data를 저장해준 뒤 link는 두 가지 경우를 나누었다. head node인 d보다 expon이 높을 경우에는 temp를 새로운 head로 바꿔준다. 아닌 경우에는 temp의 expon보다 작은 node가 나올 때까지 search 후에 해당 지점에 pre와 d_idx에 저장된 node들을 이용하여 list에 link한다. 모든 경우의 수 search 후에는 list를 돌면서 coef가 0인 node들을 찾아서 삭제한다. head node가 coef가 0인 경우는 head를 바꿔주며 이전 head는 free 처리한다. 이외의 coef가 0인 node들은 pre의 link를 d_idx->link로, d_idx를 del에 저장하며 다음 link로 옮긴다. 그 후 del을 free 처리하며 삭제한다. 이 작업이 끝난 후에 d를 return하면서 function이 종료된다. void pwrite(poly_pointer d) function은 우선 d polynomial의 모든 node의 개수 먼저 counting 한다. 이후 counting한 개수를 file에 output하고, 다시 모든 linked list를 돌면서 file에 각각의 node의 coef와 expon value를 file에 output한다. main function은 a, b polynomial을 input 받는 부분과 function을 call하는 부분, 그리고 모든 list의 memory를 해제하는 부분 3가지로 나눌 수 있다. a, b polynomial을 input 받는 부분은 a.txt와 b.txt로부터 input 받고 이는 pmult에서 d를 생성하는 과정과 유사하게 구현했다. 그 후 poly_pointer d에 pmult의 결과를 저장하고, pwrite(&d)로 d.txt에 결과를 저장한다. 그 후 pre와 ptr variable을 이용하여 a, b, d list에 대해서 search하면서 memory 해제를 진행한다.

다음은 결과에 대한 확인 과정이다. 우선 input으로 주어진 a.txt와 b.txt는

위와 같고, code를 돌린 결과는



위와 같이 잘 계산되고 d.txt에 잘 output된 것을 확인할 수 있다. 다른 예시에 대해서도 결과를 확인해보았다.



위와 같은 예시로 test를 진행했다. 그 결과는 다음과 같다.

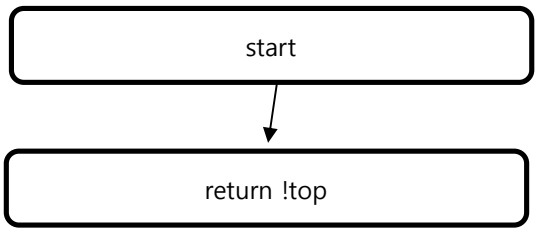
```
cse20211522@cspro:~/HW4$ ./test2
cse20211522@cspro:~/HW4$ cat d.txt
10
-12 11
13 10
7 9
2 8
-21 7
37 6
-6 5
20 4
6 3
8 1
```

위와 같이 계산에 문제 없이 d.txt에 잘 저장된 것을 확인할 수 있다. 구현에 있어서 아쉬운 점으로는 head node에 node의 개수를 저장하는 variable이 존재하지 않아 pwrite에서 일일이 세야 하는 부분이 필요했던 점에 아쉬움이 있다. 그 외에는 pmult는 time complexity가 a의 길이 M, b의 길이가 N이라 하면 O(M * N)이라 할 수 있으므로 비효율적인 부분은 없다고 생각한다.

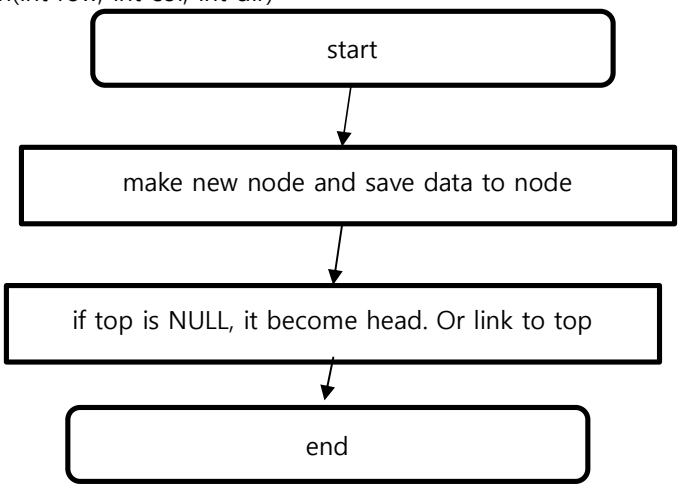
3번 문제

<flow chart>

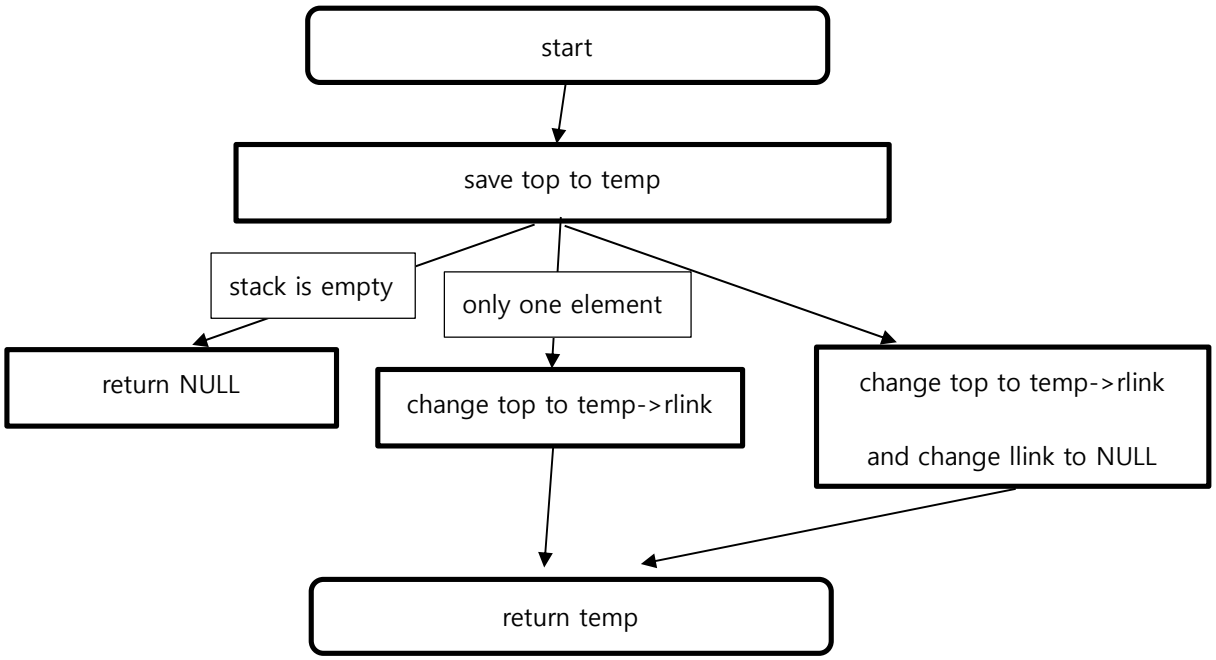
-int isEmpty()



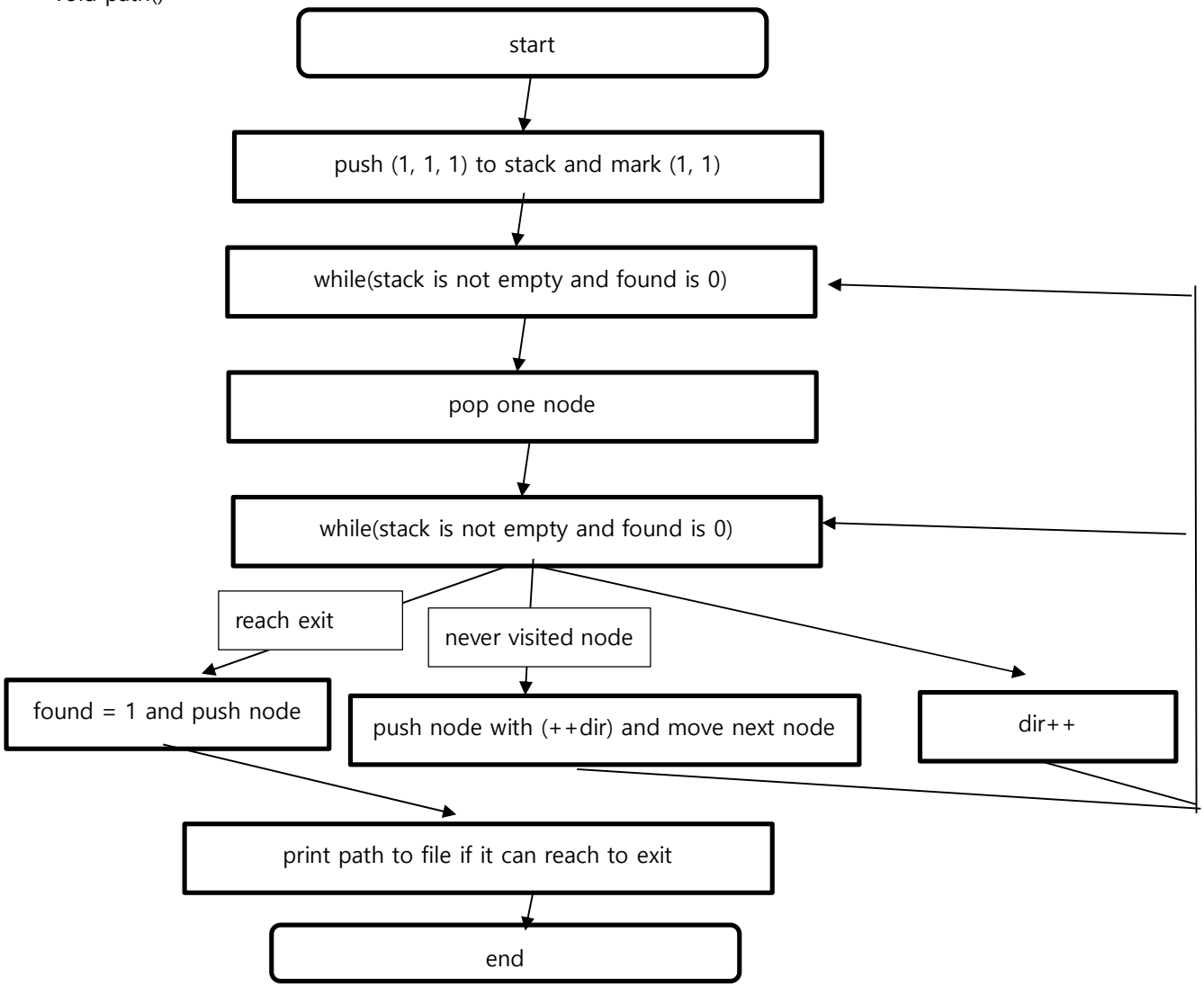
-void push(int row, int col, int dir)



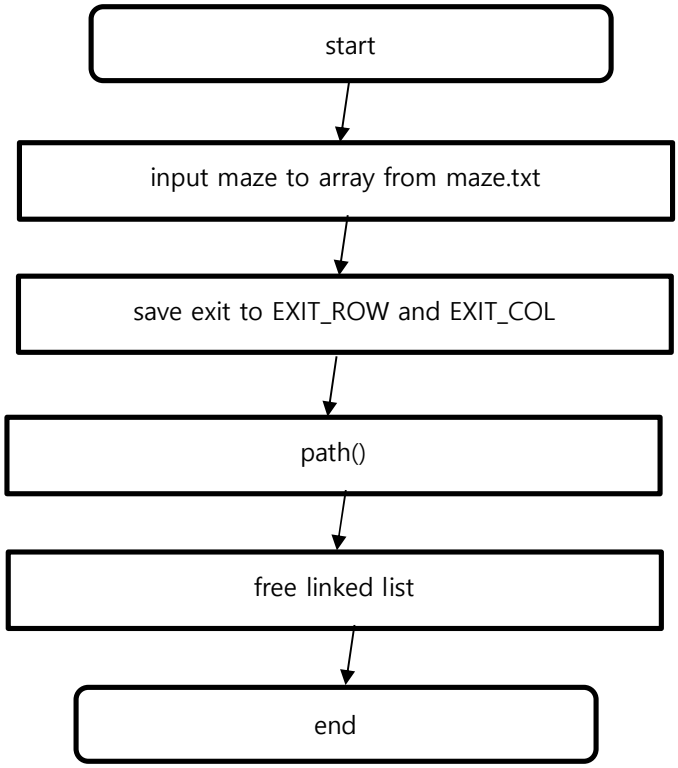
-node_pointer pop()



-void path()



-int main()



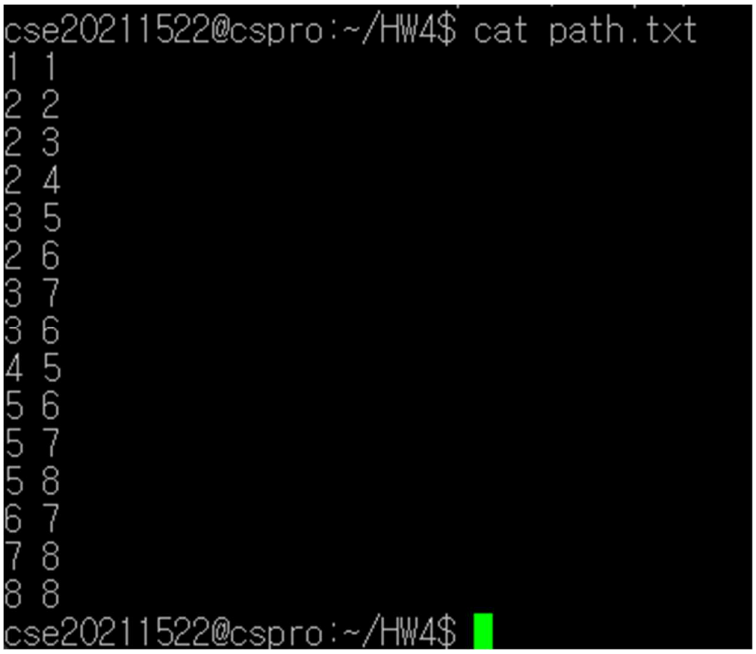
<구현 설명>

주어진 문제는 기존에 강의자료에서 array로 구현한 stack으로 maze problem을 해결하던 부분을 doubly linked list로 저장하여 path의 정보를 저장하도록 하는 문제이다. 우선 node struct는 node_pointer type의 llink와 rlink를 가지고, int type의 row, col, dir을 가지고 있도록 구성했다. 여기서 node_pointer는 node의 pointer type이다. doubly linked list로 구현하는 stack은 global variable인 top에 node를 link하는 방식으로 구현했다. 그 후에 각각 isEmpty()와 push, pop 기능을 구현했다. isFull은 linked list 특성 상 node만 계속 link하면 되므로 제한이 없어 필요가 없기에 구현하지 않았다. int isEmpty()는 !top을 return하는 것으로 구현했다. top이 NULL일 때가 stack이 비어있는 것이기 때문이다. 그 다음으로는 void push(int row, int col, int dir)이다. 이는 function 내에서 temp node를 생성하여 해당 node에 argument로 주어진 data를 넣는다. 그 후 top이 NULL 즉 stack이 비어있을 때는 temp를 top에 넣어 list의 head가 될 수 있도록 했다. 이 경우에 llink와 rlink가 모두 NULL으로 하여 양 쪽이 다 끝임을 알 수 있게 표시했다. 7이 외의 경우에는 기존의 top을 temp의 rlink에 link 후에 top의 llink를 temp로 하고, top을 temp로 바꾸었다. 또한 새롭게 top이 된 temp는 llink를 NULL로 하여 역순으로 search 시에도 끝을 알 수 있도록 했다. node_pointer pop()은 우선 temp에 top을 저장한다. 그 후에 isEmpty()인 경우에는 temp에 NULL을 넣어 temp를 return하고, 이외의 경우에는 element가 오직 하나인 경우와 아닌 경우로 나누었

다. element가 오직 하나인 경우에는 top을 그냥 temp의 rlink로 이동하기만 한다. 아닌 경우에는 top을 temp의 rlink로 이동 후 top의 llink를 NULL로 바꾼다. 이 후에 temp를 return하면서 pop은 종료된다. void path()는 push와 pop을 사용하여 구현된다. 처음에 global variable로 declare한 mark array의 mark[1][1]을 1로 표시하며 push(1, 1, 1)로 첫 value를 stack에 push한다. 그 후 while loop를 통해서 top이 NULL이 아니고 found가 1이 아닐 동안 position variable에 stack에서 pop한 node를 통해 search한다. pop한 node에서 row, col, dir value를 variable에 저장하고, while loop에서 dir이 8보다 작고, found가 1이 아닌 동안 다음 지점을 search한다. dir에 따른 다음 지점이 exit이면 found를 1로 바꾸고, stack에 push한다. 이는 exit을 찾고 난 후에 마지막 node가 stack에 들어가 있지 않은 부분에 대해서 강의자료 code에서 수정한 부분이다. 이외에 map에서 해당 지점이 1이 아니고 mark에도 체크되어 있지 않으면 우선 현재 지점을 dir을 1 증가시킨 후 stack에 push한 후 해당 지점을 이동한다. 이외에는 dir을 1 증가시킨 후 다시 search한다. 한 position에 대한 code가 끝나면 더 이상 node를 사용하지 않으므로 해당 node는 free로 memory를 해제한다. 모든 loop가 종료되면 found가 1인 경우 즉, exit에 도달할 수 있는 경우는 path.txt file에 모든 path를 print 한다. 이 경우에 stack의 역순으로 print해야 하므로 우선 loop를 통해 stack의 제일 아래 node로 이동한다. 그 후에 llink를 따라서 순서대로 모든 node들을 file에 print 한다. 그리고 마지막으로 exit의 row와 col을 file에 print하고 종료한다. exit에 도달하지 못하는 경우에는 찾지 못했다는 메시지만 표시한다. main function에서는 maze.txt로부터 maze를 input받는 부분과 path()의 call, list의 memory 해제의 기능이 구현되어 있다. maze의 input은 '₩n'이 나올 때마다 i를 증가, j를 0으로 initialize, '1' or '0'인 경우 j를 증가하며 map[i][j]에 넣고, EOF인 경우에 break하도록 구현했다. 이 후 global variable인 EXIT_ROW와 EXIT_COL에 exit의 위치를 넣었다. 그 후 path()를 call하고, doubly linked list를 memory 해제하면서 종료된다. 구현 시에 이용한 global variable은 maze를 저장할 int map[10000][10000], 지나간 여부를 체크할 mark[10000][10000], 그리고 int EXIT_ROW, EXIT_COL, 마지막으로 dir마다의 direction을 지정하는 row_dir과 col_dir array를 이용하였다. map의 size는 10000 * 10000 이면 size에 있어 부족함이 없을 것으로 판단하여 넣었다.

다음으로 결과에 대한 test를 통한 확인이다. pdf에 주어진 예제에 대해서 test 해보았다.

위와 같은 maze.txt를 이용하였고 결과는 다음과 같이 나온다.



결과가 예시 답안과 같은 것을 확인할 수 있었고 잘 돌아가는 것을 확인했다. 구현 과정에서 아쉬운 점으로는 maze를 저장할 array나 mark array에 대해서 size를 유연하게 설정할 수 있도록 구현하지 못한 데에 있어 아쉬운 점이 있다. 그렇기에 작은 maze에도 큰 array를 사용하도록 구현되었다. 이 점은 maze의 size를 input 받거나 미리 한 번 maze.txt를 search하여 size를 측정 후 dynamic allocating을 이용하는 방식을 개선할 수 있을 것이라 추측된다.