데이터구조설계

1차 프로젝트

이기훈 교수님

컴퓨터정보공학부

2018202076

이연걸

- Introduction

  이번 프로젝트는 Stack, Queue, Linked-List, BST 등의 자료구조와 Boyer-Moore 등의 알고리즘을 사용해 파일을 업로드, 추가, 수정, 이동, 출력, 검색, 선택, 편집의 작업을 수행하는 것이다. 위의 기능을 구현하기 위해서는 기본적으로 파일 입출력에 대한 이해가 필요했다. 파일을 읽어서 변경하고 새롭게 만들고 하는 부분에서 입출력 함수들의 옵션을 공부할 필요가 있었다.

  LOAD, ADD 명령어를 완성하기 위해서는 연결 리스트의 개념이 필요하다. 더 나아가 2 차원 연결 리스트를 구현해야 하는데 단순히 2 차원 배열을 리스트로 만든 것으로 생각할 수 있다.

  MODIFY 명령어를 완성하기 위해서는 2 차원 연결 리스트의 활용 능력이 필요하다. 쉽게 1 번 리스트, 2 번 리스트로 LOAD 와 ADD 명령어의 수행결과를 정의하면 1, 2 번 리스트에 쓰여 있는 디렉토리 정보를 바탕으로 찾는 파일이 어느 위치에 있는지 확인할 수 있고 리스트 노드에 쓰여 있는 정보를 사용해 찾는 파일이 어느 노드에 있는지 확인할 수 있다.

  MOVE 명령어를 완성하기 위해서는 BST 를 알아야 한다. Binary Search Tree 는 왼쪽, 오른쪽의 서브트리가 모두 BST 이다. 각 노드는 왼쪽, 오른쪽 자식, 값 이렇게 3 개의 필드를 갖고 있는 것을 활용해 class 를 정의할 수 있고 파일들의 인덱스를 비교하면서 BST 에 push 하면 된다.

  PRINT 명령어를 완성하기 위해서는 트리 중위 순회, 즉 Inorder-traversal 을 알아야 한다. 이것은 왼쪽 서브 트리를 방문 후 자신을 방문하고 그 후 오른쪽 서브 트리를 방문하는 알고리즘이다.

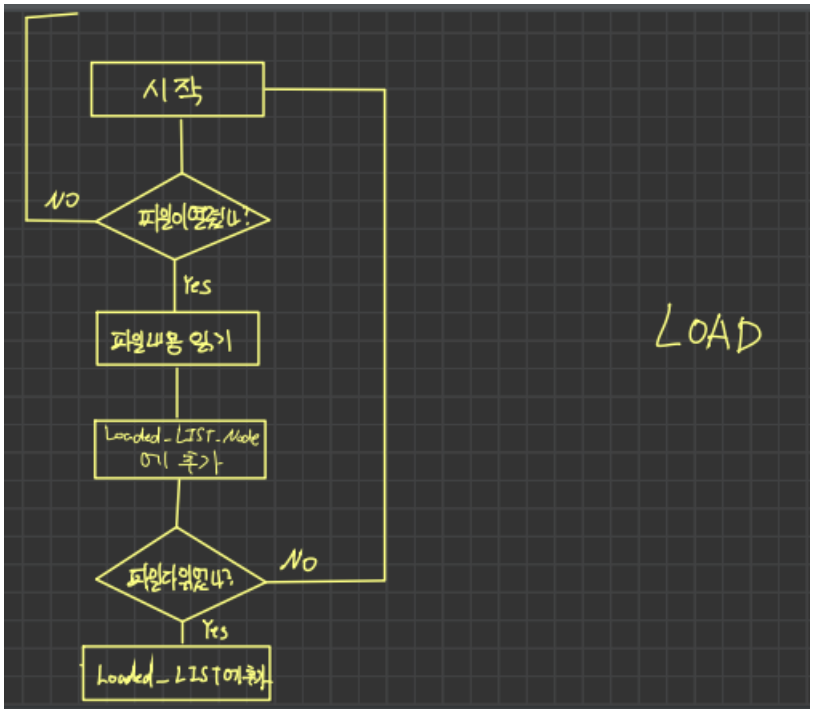
  SEARCH 명령어를 완성하기 위해서는 postorder-traversal 을 알아야 한다. 이것은 왼쪽 서브 트리를 방문 후 오른쪽 서브 트리를 방문하고 마지막으로 자신을 방문하는 알고리즘이다. 이것을 반복문으로 구현하기 위해서는 STACK 자료구조가 필요하다.

  SELECT 명령어를 완성하기 위해서는 preorder-traversal 을 알아야 한다. 이것은 자신을 방문하고 왼쪽 서브 트리를 방문 후 오른쪽 서브 트리를 방문하는 알고리즘이다.

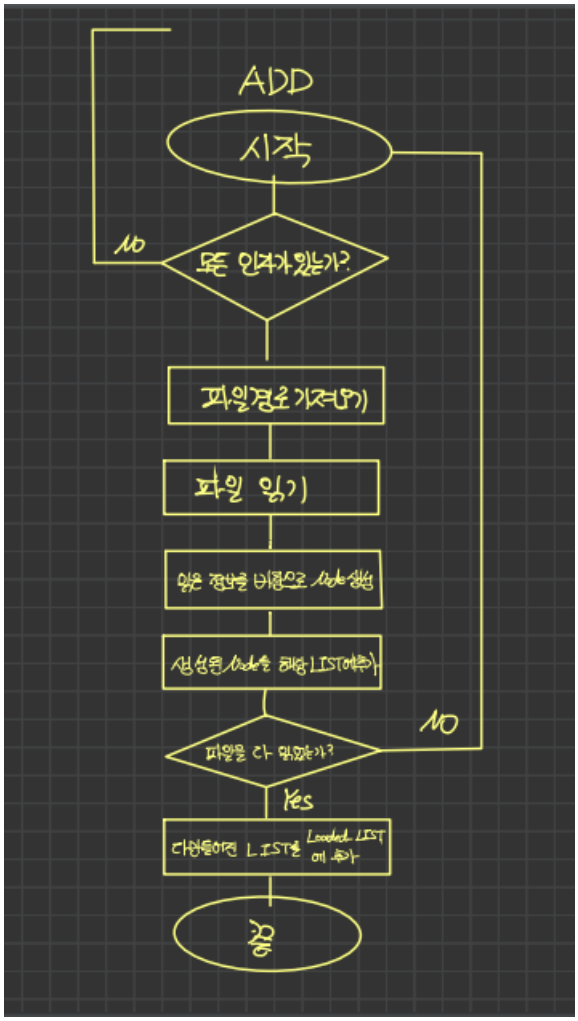
  EDIT 명령어를 완성하기 위해서는 STACK 과 QUEUE 의 활용 능력이 필요하다. 파일을 점대칭하는 flipped 는 STACK 을 사용해서 파일의 데이터를 역순으로 정렬한다. 파일의 밝기를 올리는 adjusted 는 QUEUE 를 사용해서 모든 데이터를 담은 후 하나씩 꺼내서 밝기를 올린다. 밝기를 올리는 작업은 인자로 전달된 수를 더하면 된다. 마지막으로 파일의 크기를 조정하는 resized 는 배열을 사용하면 완성할 수 있다.

  가장 마지막 명령어인 EXIT 는 지금까지 할당 받은 메모리들을 할당 해제하고 메모리 leak 이 없는 상태로 만드는 것이다. Class 는 해당 범위를 벗어나면 자동으로 소멸자가 호출되어 사라지기에 Loaded_LIST 와 Database_BST 를 할당해제 해주고 로그를 출력한 뒤 마무리하였다.

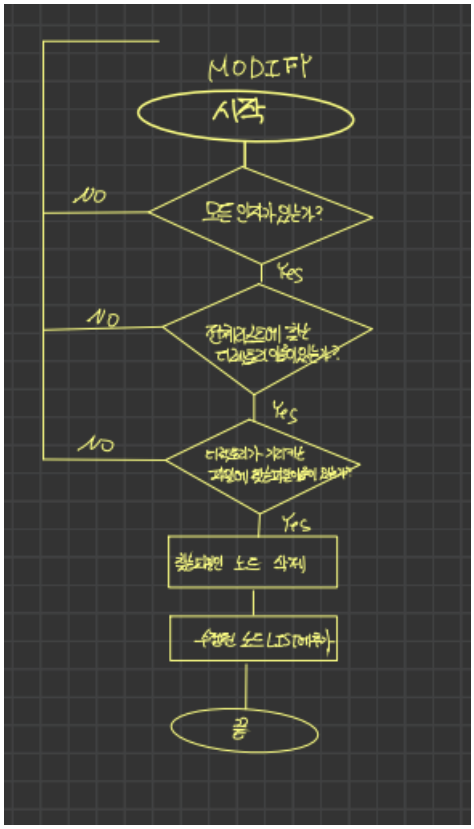
- Flowchart



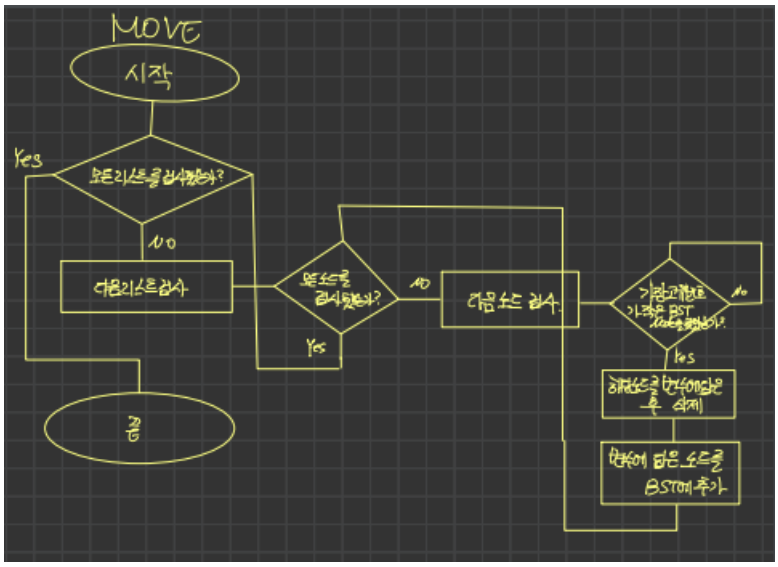
위는 LOAD 의 플로우 차트다. 파일이 열렸는지 확인, 파일 내용을 읽어서 Node 로 만든 뒤 Loaded_LIST_Node 에 추가, 파일을 다 읽지 않았으면 위의 작업을 반복하면 다 읽었다면 Loaded_LIST 에 리스트를 추가하고 종료한다.



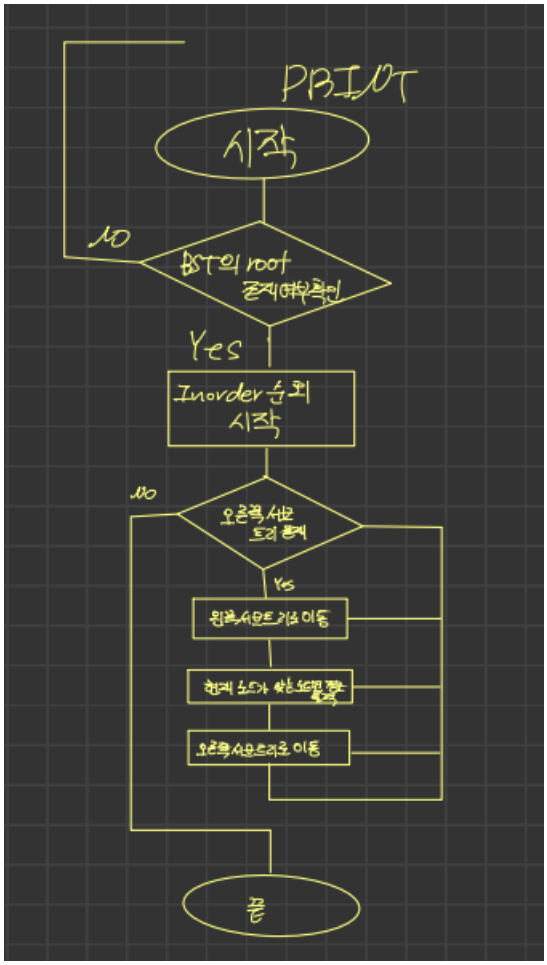
ADD 의 플로우 차트다. 모든 인자가 없다면 전체 while 문으로 돌아간다. 파일 경로를 가져와 파일을 읽고 읽은 정보로 Node 를 만들어 ROW LIST 에 추가하고 파일을 다 읽기 전까지 위의 작업을 반복한다. 파일을 다 읽었다면 새롭게 만들어진 (ROW) LIST 에 추가하고 작업을 마친다.



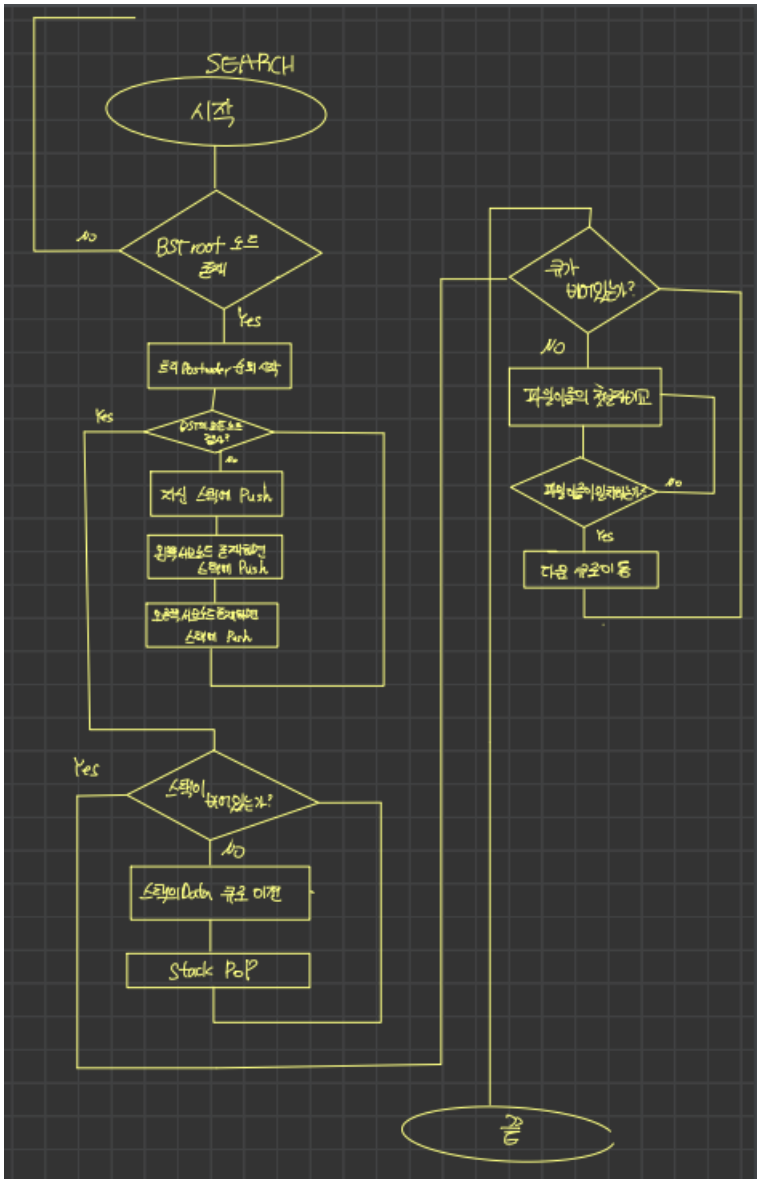
MODIFY 의 플로우 차트다. 모든 인자가 없거나 내가 찾는 디렉토리가 존재하지 않거나 디렉토리 내에 즉 노드 내에 파일이 존재하지 않는다면 본래의 반복문으로 돌아간다. 찾는 파일(노드)이 존재한다면 해당 노드를 삭제하고 수정된 노드를 추가한다.



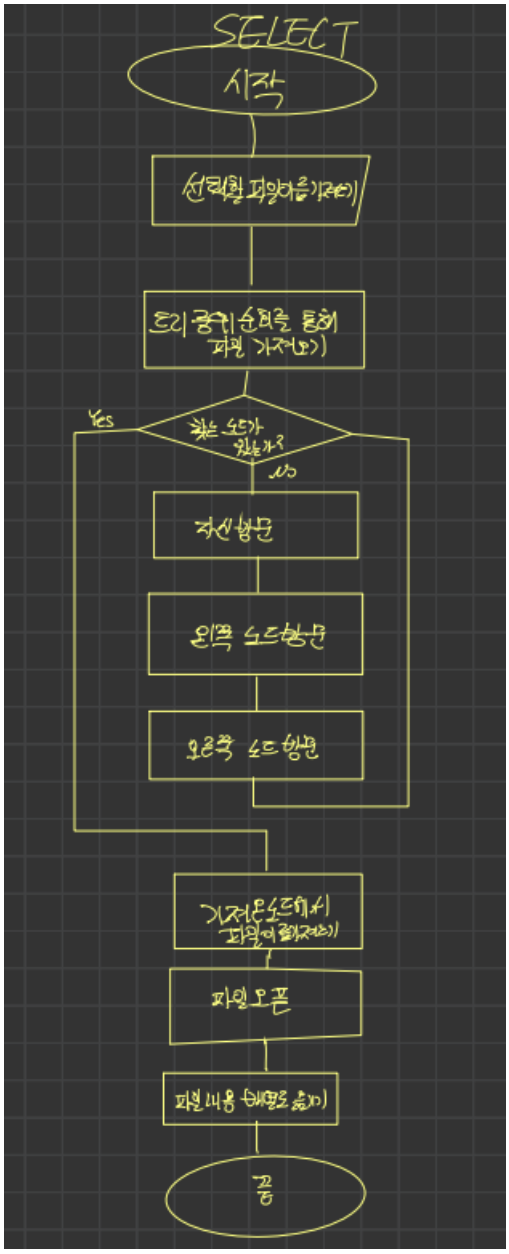
모든 ROW_LIST 와 또 그안의 Loaded_LIST_Node 를 찾는다. 각각의 노드를 삭제하고 BST 에 추가하고 이 작업을 반복해 모든 노드를 순회한다.



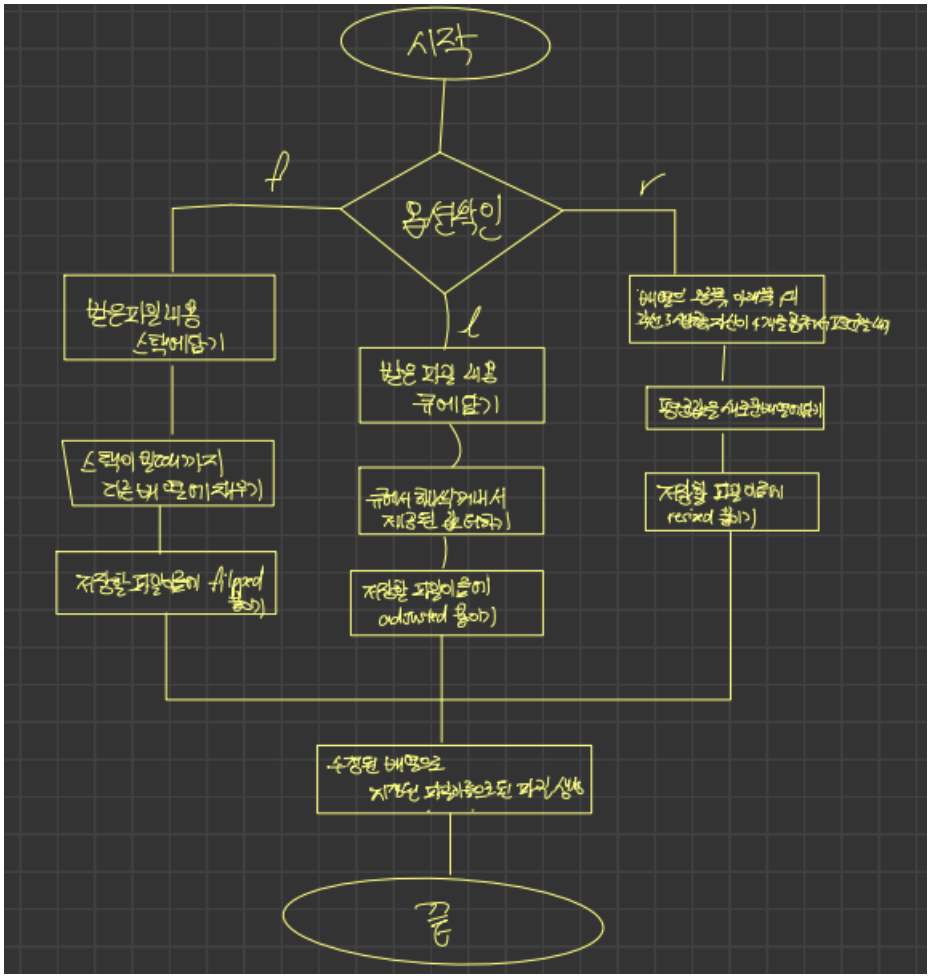
PRINT 의 플로우 차트다. BST 의 root 가 존재한다면 inorder-traversal 을 시작한다. 해당 알고리즘은 위에서 설명했다. 순회를 통해 모든 노드를 검사해 찾으려 한 노드를 찾으면 작업을 마친다.



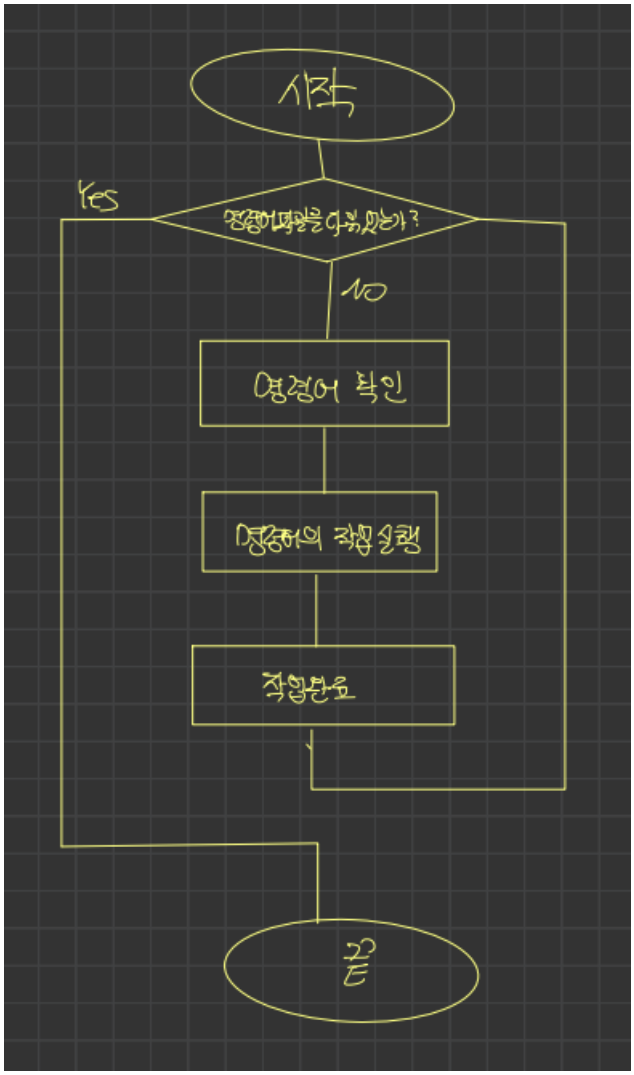
BST root 가 존재하면 post order traversal 을 시작한다. 작동 방식은 위에서 설명했다. 순회가 끝나면 스택이 빌 때 까지 큐로 데이터를 옮긴다. 이제 옮겨진 큐를 pop 해가면서 원하는 노드를 찾는다. 큐가 비었다면 작업을 마친다.



선택할 파일 이름을 가져와 트리 중위 순회를 시작한다. 트리 중위 순회를 마치고 가져온 노드에서 파일 이름을 추출한다. 파일 이름을 사용해 해당 파일을 열고 fread 를 사용해 해당 파일의 내용을 배열에 쓴다.



옵션을 확인해 점대칭인지 밝기 조절인지 파일 압축인지를 확인한다. 각각은 스택+배열, 큐+배열, 배열을 사용하고 수정된 배열을 사용해 수정된 파일을 만든다.



앞서서 설명한 작업들을 위와 같이 반복문을 돌면서 진행한다. 명령어 파일을 다 읽을 때 까지 각종 명령을 실행하며 모든 파일을 다 읽었다면 프로그램을 종료한다.

- Algorithm
  - BST

    BST 는 Binary Search Tree 의 줄임 말이다. 노드 x 에 대해 왼쪽 서브 트리의 노드는 모두 x 보다 작다. 반면 오른쪽 서브 트리의 노드는 모두 x 보다 크다.

```
void insert(Loaded_LIST_Node *node)
{
    int idx = stoi(node->index);
    Database_BST_Node *p = tree_root, *pp = NULL;

    while (p)
    {
        pp = p;
        if (idx < p->tree_data->index)
            p = p->tree_left;
        else if (idx > p->tree_data->index)
            p = p->tree_right;
    }

    ImageNode *img_node = new ImageNode(node->file_name, node->dir_n
    p = new Database_BST_Node(img_node, NULL, NULL);

    if (tree_root != NULL)
    {
        if (idx < pp->tree_data->index)
            pp->tree_left = p;
        else
            pp->tree_right = p;
    }
    else
        tree_root = p;
};
```

BST 의 Insert 기능이다. Root 가 존재하지 않는다면 새롭게 할당한다. 만약 root 가 존재한다면 새롭게 들어온 노드의 인덱스를 비교하면서 위치를 결정해 준다. 해당 노드가 leaf 로 가게 되면 p 는 NULL 이기에 반복문을 탈출하고 밑으로 내려와 부모 노드를 결정해준다.

```
void deletion(int index)
{
    Database_BST_Node *p = tree_root;
    Database_BST_Node *pp = NULL;
    while (p && index != p->tree_data->index)
    {
        pp = p;
        if (index < p->tree_data->index)
            p = p->tree_left;
        else
            p = p->tree_right;
    }
    if (p == NULL)
        return;
    if (p->tree_left == NULL && p->tree_right == NULL)
    {
        if (pp == NULL)
            tree_root = NULL;
        else if (pp->tree_left == p)
            pp->tree_left = NULL;
        else
            pp->tree_right = NULL;
        delete p;
    }
    else if (p->tree_left == NULL)
    {
        if (pp == NULL)
            tree_root = p->tree_right;
        else if (pp->tree_left == p)
            pp->tree_left = p->tree_right;
        else
            pp->tree_right = p->tree_right;
        delete p;
    }
```

```
    else if (p->tree_right == 0)
    {
        if (pp == 0)
            tree_root = p->tree_left;
        else if (pp->tree_left == p)
            pp->tree_left = p->tree_left;
        else
            pp->tree_left = p->tree_left;
        delete p;
    }
    else
    {
        Database_BST_Node *pp = p;
        Database_BST_Node *prev = p->tree_left;
        Database_BST_Node *curr = p->tree_left->tree_right;

        while (curr)
        {
            pp = prev;
            prev = curr;
            curr = curr->tree_right;
        }

        p->tree_data->index = prev->tree_data->index;
        if (pp == p)
            pp->tree_left = prev->tree_left;
        else
            pp->tree_right = prev->tree_left;
        delete prev;
    }
};
```

삭제는 우선 해당 index 를 갖는 노드를 찾는 것부터 이루어진다. 해당 노드(p)가 리프 노드라면 부모 노드인 pp 의 왼쪽, 오른쪽 중 어느곳에 위치하는 가를 확인하고 해당 위치에 NULL 을 넣어주고 삭제한다.

해당 노드가 오른쪽 노드를 갖고 있다면 부모 노드가 루트인지, 부모 노드의 왼쪽, 오른쪽 중 어느곳에 위치하는 가를 확인하고 해당 위치에 현재 노드의 오른쪽 노드를 할당하고 현재 노드를 삭제한다.

해당 노드가 왼쪽 노드를 갖고 있다면 위와 같은 조건을 확인하고 해당 위치에 현재 노드의 왼쪽 노드를 할당하고 현재 노드를 삭제한다.

만약 해당 노드가 왼쪽, 오른쪽 노드를 갖고 있다면 오른쪽 서브트리에서 가장 작은 값을 찾는다. 삭제될 노드가 부모 노드와 같다면 즉, 부모 노드의 왼쪽에 서브 트리를 붙여야 한다면 부모 노드의 왼쪽 서브트리에 삭제될 노드의 왼쪽 서브트리를 붙이고 반대라면 오른쪽 서브트리에 삭제될 노드의 왼쪽 서브트리를 붙이면 된다. 그 후 현재 노드를 삭제한다.

- Boyer_moore

```
void boyer_moore(QUEUE<Database_BST_Node> *Q, ofstream *f_log, string file_name)
{
    while (!Q->empty())
    {
        for (unsigned int i = 0; i < Q->top().tree_data->file_name.length(); i++)
        {
            unsigned int j;
            for (j = file_name.length() - 1; j > 0; j--)
            {
                if (Q->top().tree_data->file_name[i + j] != file_name[j])
                    break;
            }
            if (j == 0)
                *f_log << "\"" << Q->top().tree_data->file_name << "\" / " << Q->top().tree_data->index << "\n";
        }
        Q->pop();
    }
};

#endif
```

보이어 무어 알고리즘은 보통 문자열의 끝에서 불일치가 일어날 확률이 높다는 것을 이용한 문자열 검색 알고리즘이다. 문자열을 하나씩 지나치면서 뒤에서부터 비교한다. 마지막 글자가 같으면 비교를 시작하고 모든 글자가 같으면 해당 파일 이름을 출력하는 간단한 형식으로 이루어져 있다.

- Preorder traversal

```
Database_BST_Node *search_bst_node(Database_BST_Node *t, int index)
{
    if (t->tree_data->index == index)
        return t;
    return (NULL);
}

Database_BST_Node *traversal_preorder(Database_BST_Node *t, ofstream *f_log, int index)
{
    if (t != NULL)
    {
        if (search_bst_node(t, index))
            return (t);
        traversal_preorder(t->tree_left, f_log, index);
        traversal_preorder(t->tree_right, f_log, index);
    }
    return (NULL);
}
```

Preorder traversal 은 노드를 방문할 때 자신->왼쪽->오른쪽 순으로 방문한다. 자신을 방문하고 왼쪽으로 이동하고 왼쪽의 방문이 모두 종료되면 오른쪽을 방문하기 때문에 재귀함수를 사용해 쉽게 구현할 수 있다.

- Inorder traversal

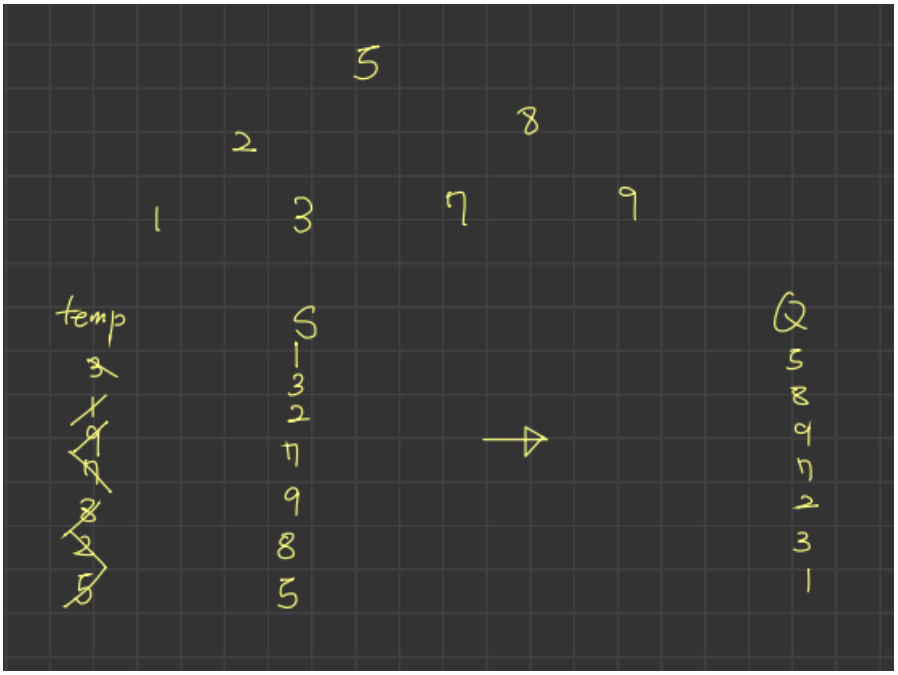
```
void print_bst_node(Database_BST_Node *t, ofstream *f_log)
{
    *f_log << t->tree_data->dir_name << " / \"" << t->tree_data->file_name << "\" / " << t->tree_data->index << "\n";
}

void traversal_inorder(Database_BST_Node *t, ofstream *f_log)
{
    if (t != NULL)
    {
        traversal_inorder(t->tree_left, f_log);
        print_bst_node(t, f_log);
        traversal_inorder(t->tree_right, f_log);
    }
}
```

Inorder traversal 은 노드를 방문할 때 왼쪽->자신->오른쪽 순으로 방문한다. 그렇기 때문에 왼쪽 서브 트리로 계속 들어가야 하고 리프 노드에 도달하면 부모 노드로 올라가 자신을 방문하고 그 후 오른쪽 서브 트리로 들어가는 형태의 재귀함수를 구성하면 쉽게 구현이 가능하다.

- Postorder traversal

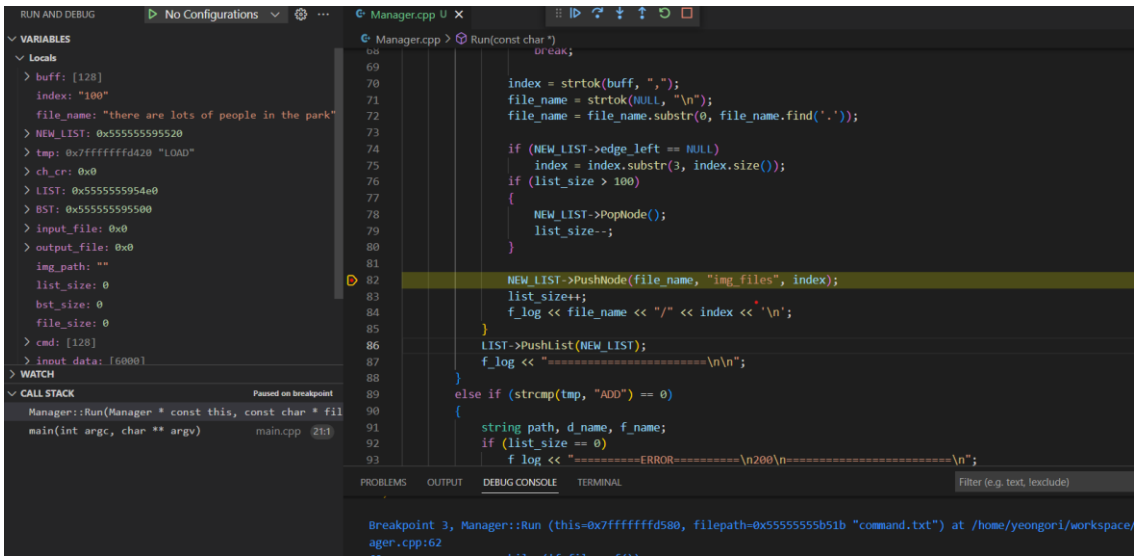
```
while (!temp->empty())
{
    Database_BST_Node curr = temp->top();
    temp->pop();

    S->push(curr);
    if (curr.getLeftNode())
        temp->push(*curr.getLeftNode());
    if (curr.getRightNode())
        temp->push(*curr.getRightNode());
}

while (!S->empty())
{
    Q->push(S->top());
    S->pop();
}
```

Postorder traversal 은 노드를 방문할 때 왼쪽->오른쪽->자신 순으로 방문한다. 그렇기에 왼쪽 서브 트리를 전부 끝내고 돌아올 필요가 있다. 그 결과 stack 자료구조를 사용하였다. 첫번째 반복문의 흐름을 그대로 따라가면 두개의 스택을 사용한다. S 는 실제로 값을 저장하는 스택이며 temp 는 잠시 값을 저장해두는 용도로 사용한다. 좌우로 갈라지는 BST 를 하는 것처럼 좌, 우에 노드가 존재한다면 스택에 담는다. 그리고 다음 반복문으로 들어가 해당 노드를 pop 하고 그곳을 기준으로 다시 탐색을 시작함과 동시에 스택에 값을 담는다. 이것을 그림으로 표현하면 다음과 같다

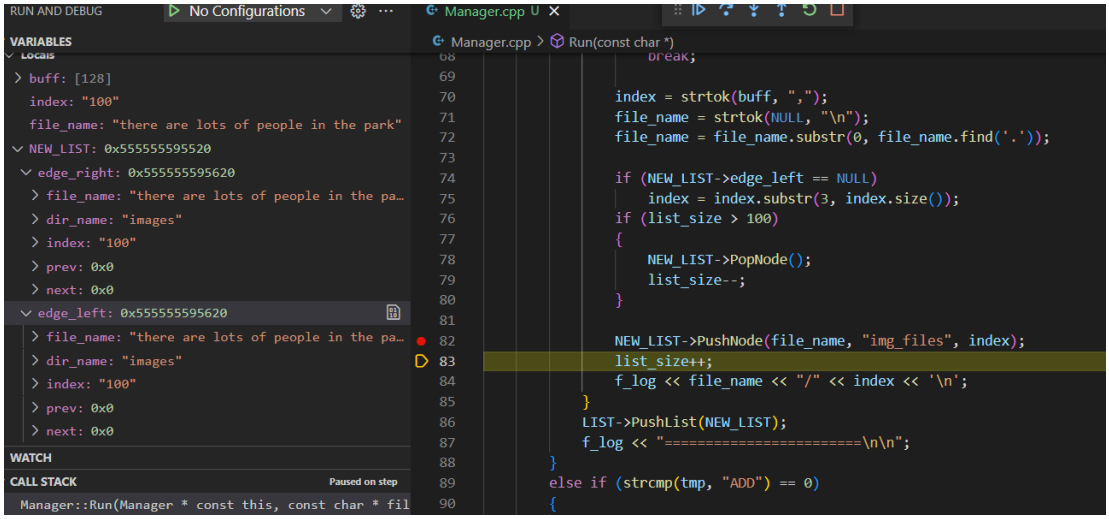


위의 과정을 거쳐서 스택 S 에 쌓인 값을 Q 로 보내면 좌->우->자신의 순서로 방문하는 post-order 순회를 완성할 수 있다.

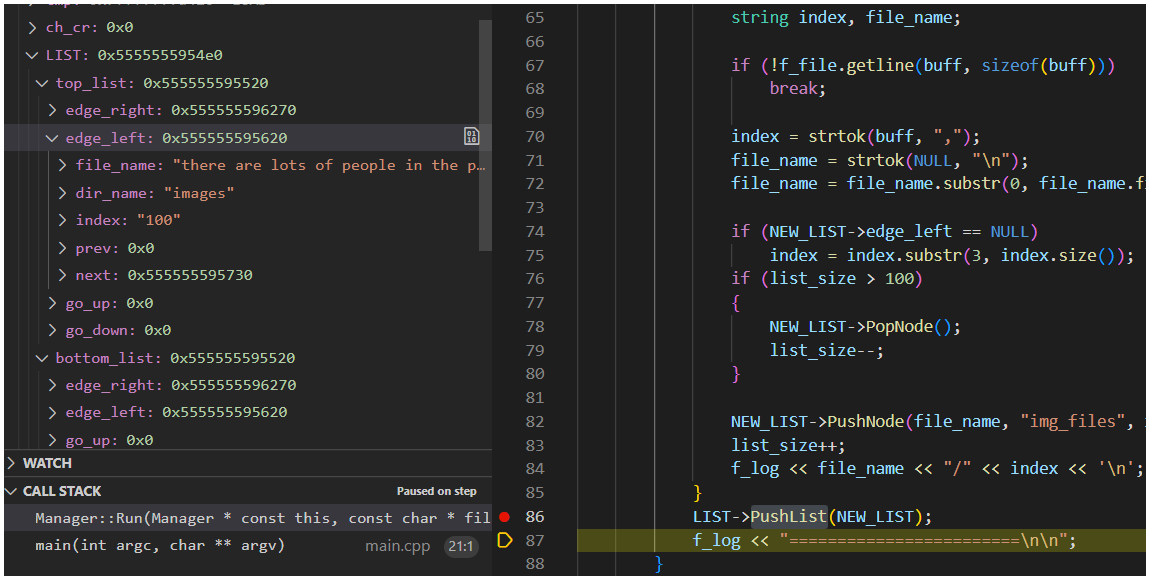
- Result Screen



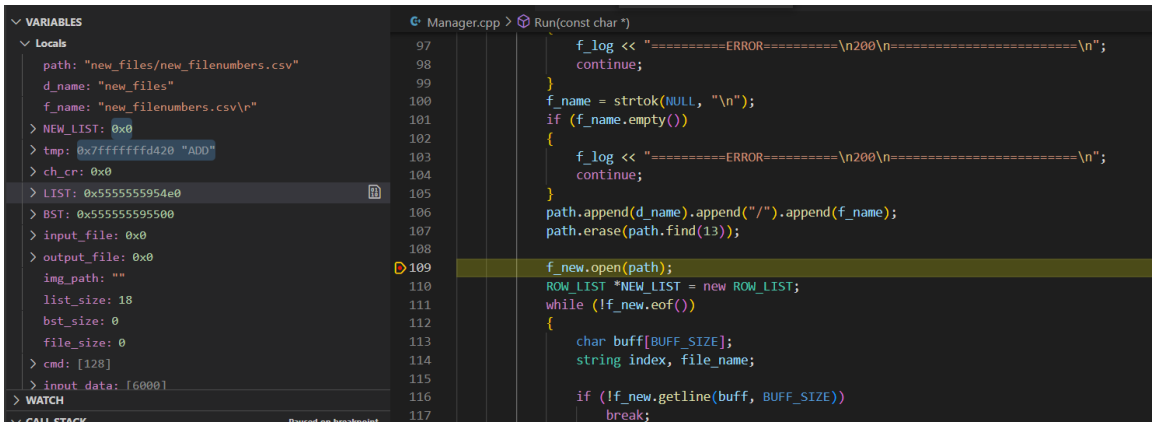
LOAD 명령어의 동작이다. 왼쪽을 보면 파일 내의 파일 이름, 인덱스가 잘 들어온 것을 확인할 수 있다.



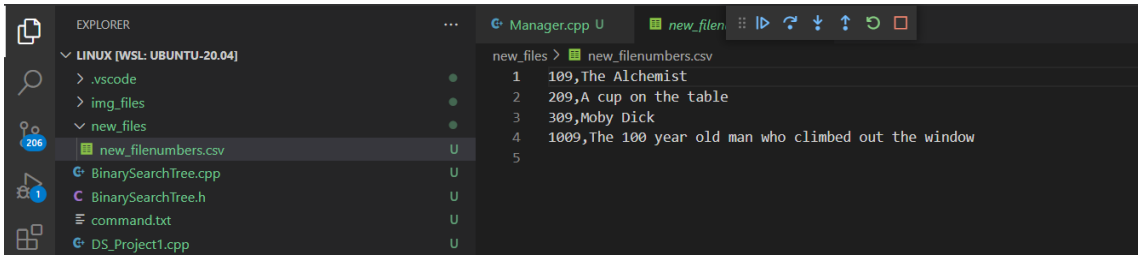
바로 다음 줄에서 NEW_LIST 로 값이 잘 들어온 것을 확인할 수 있다.

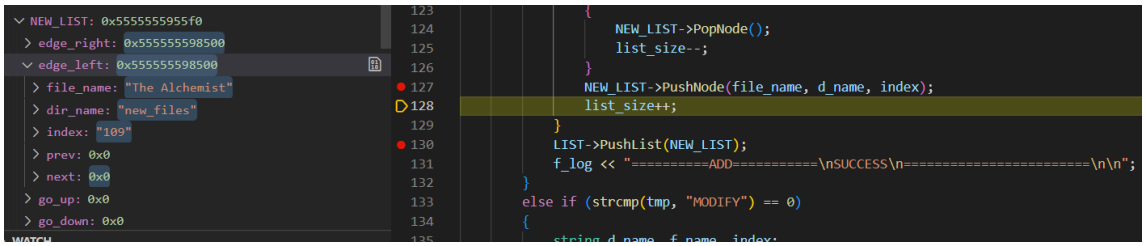


전체 리스트에도 값들이 잘 입력된 것을 확인할 수 있다.

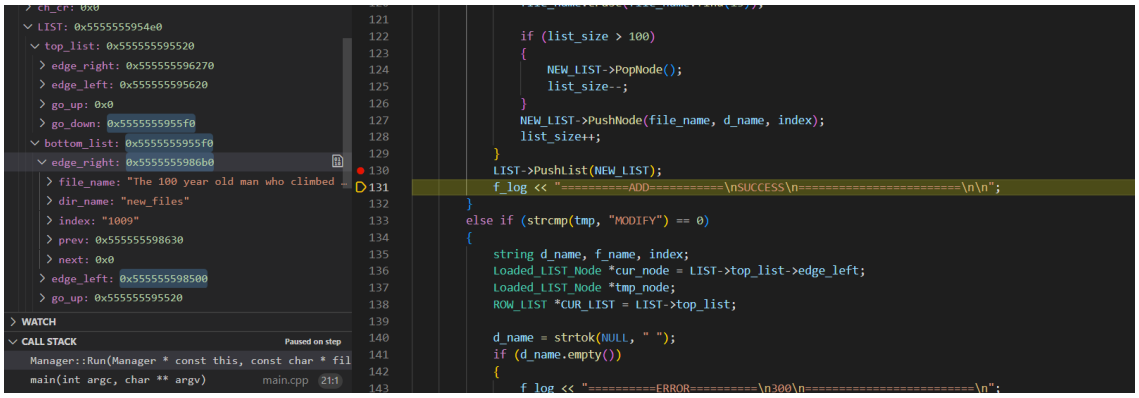


다음은 ADD 명령어다. 왼쪽의 path 변수를 보면 Command.txt 에 저장된 디렉토리와 파일 이름을 잘 가져온 것을 확인 할 수 있다.

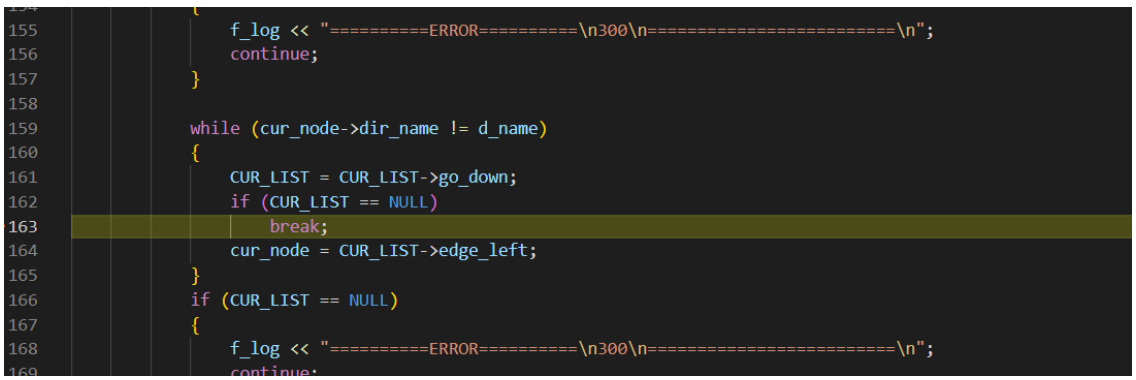




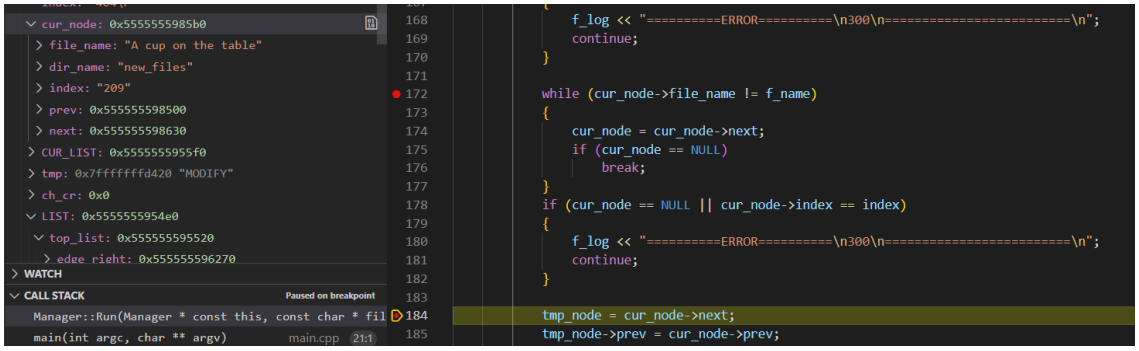
File_filenumbers.csv 파일내에 저장된 첫번째 파일이 NEW_LIST 에 잘 push 된 것을 확인할 수 있다.



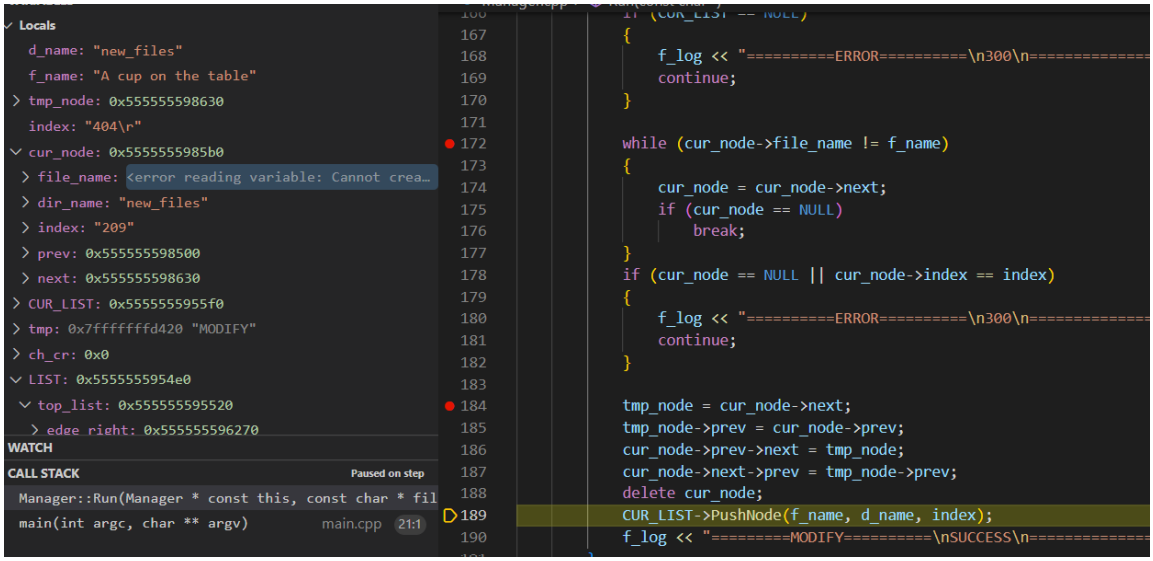
ADD 명령어는 전체 Loaded_LIST 에 추가된 새로운 LIST 다. 그렇기 때문에 LIST 의 bottom_list 에 위치한다. 그리고 new_filenumbers.csv 의 가장 마지막인 The 100 year old.. 파일이 가장 오른쪽에, 즉 과제에서 제시한 조건에 맞게 들어간 것도 확인할 수 있다.



과제에서 제시된 첫번째 MODIFY 명령어는 찾는 디렉토리가 존재하지 않는다. 때문에 에러 처리 문으로 잘 들어간 것을 확인할 수 있다.

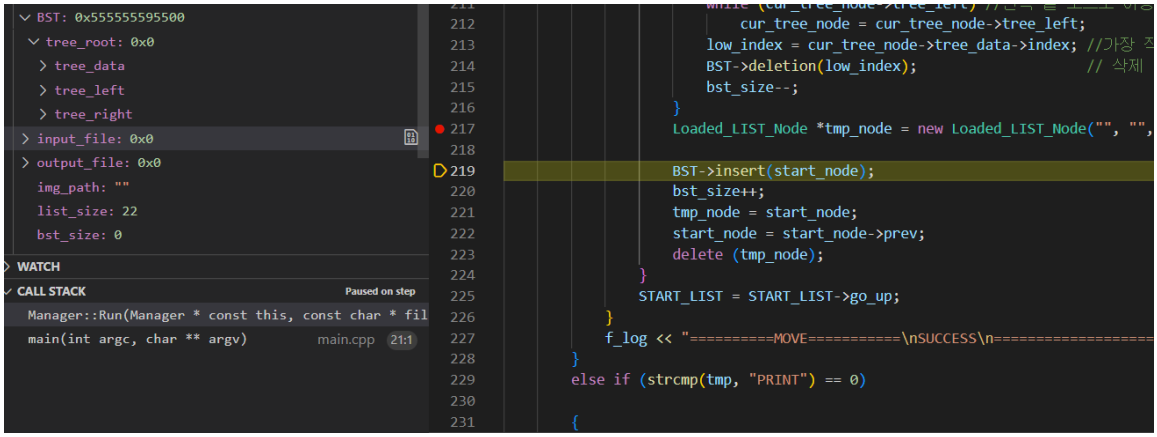


찾고자 하는 node 인 file_name 이 A cup on the table 과 dir_name 이 new_files 인 노드를 찾은 것을 왼쪽 위에서 확인할 수 있다.

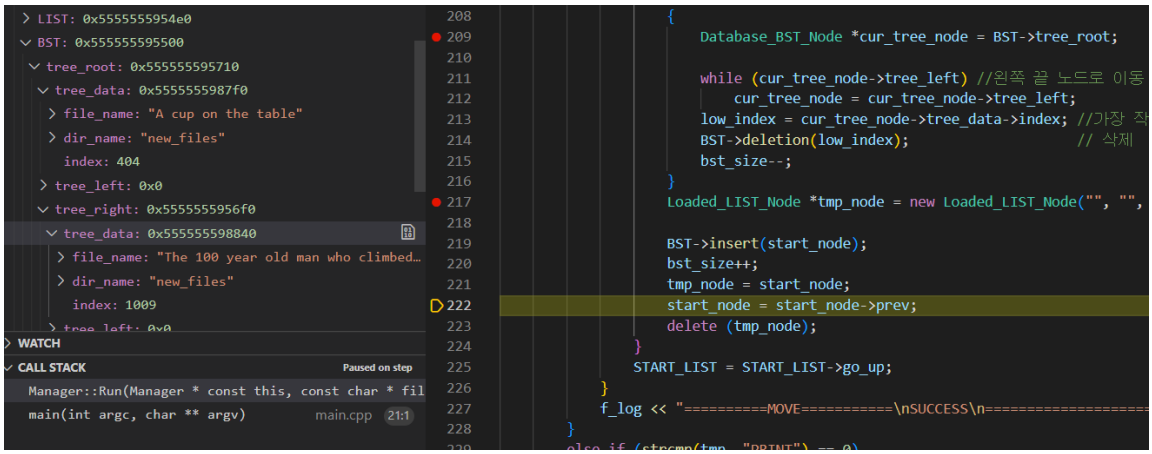


수정에 사용되는 index 를 넣어줌으로써 index 가 209 에서 404 로 바뀐 것을 확인할 수 있다.

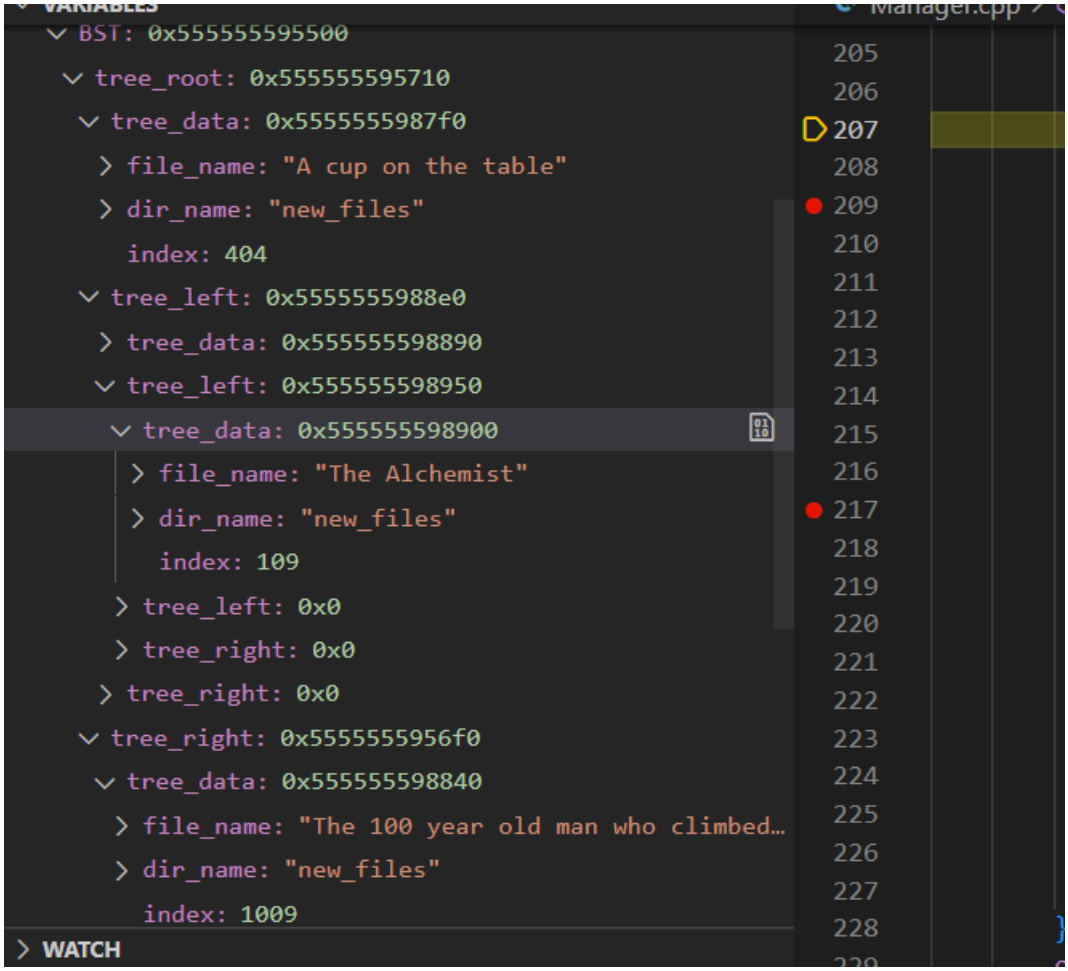
다음은 MOVE 다. 만들어진 모든 LIST 를 순회하기 위해 bottom_list 여기서 사용된 command.txt 에 의하면 new_filenumbers.csv 의 첫번째 노드와 마지막 노드가 순회용 리스트에 잘 들어간 것을 확인할 수 있다.



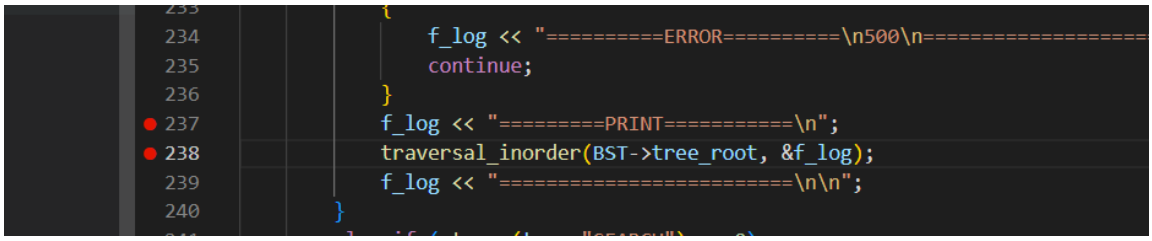
BST 에는 현재 아무런 값도 담겨있지 않은 것을 확인할 수 있다.



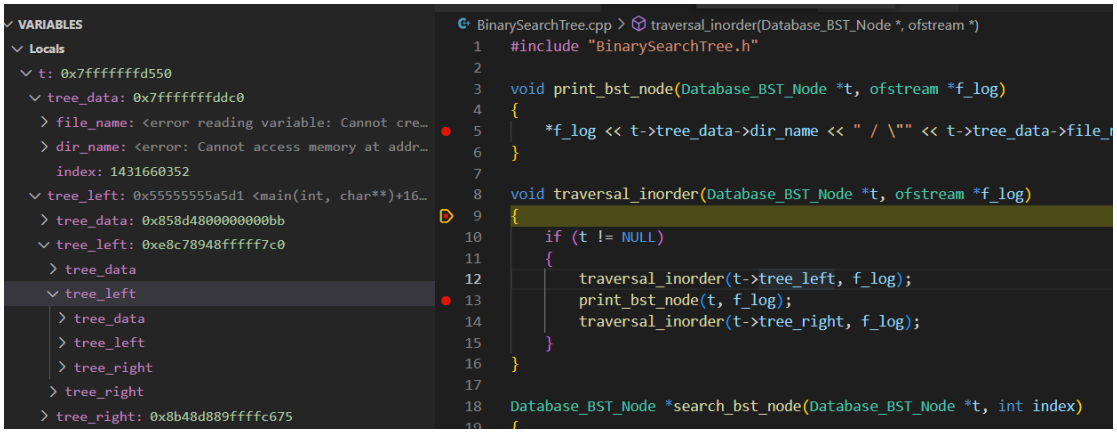
몇번의 반복을 거치면서 tree_root 와 그보다 인덱스가 큰 값이 오른쪽에 담긴 것을 확인할 수 있다.



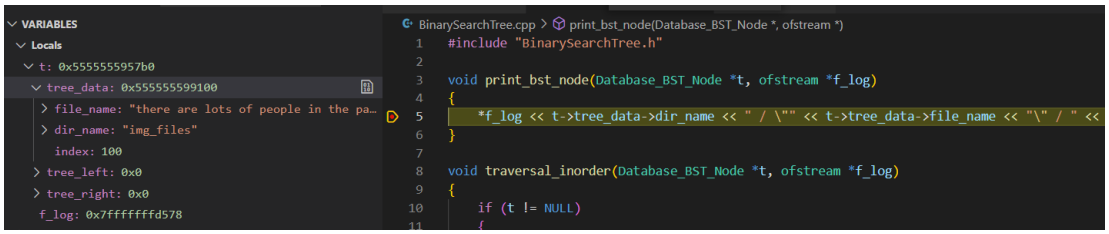
BST 의 node 에 root 의 인덱스보다 작은 값은 왼쪽 큰 값은 오른쪽에 잘 담긴 것을 확인할 수 있다.



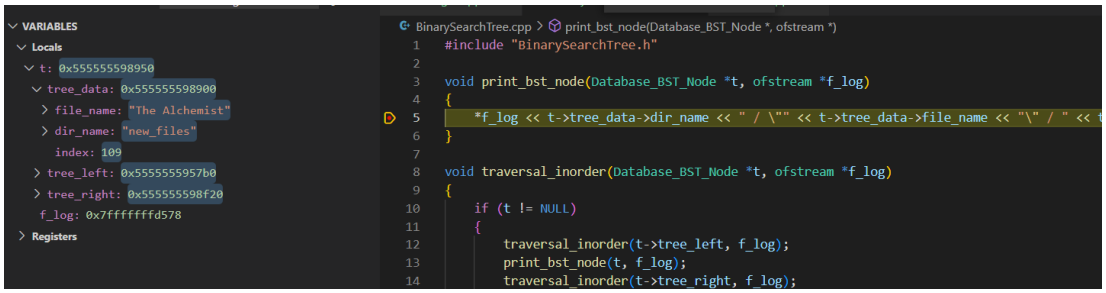
다음은 PRINT 명령어다.



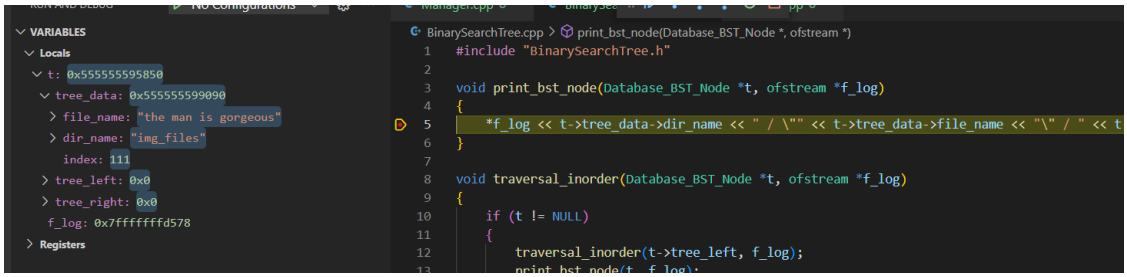
현재 t 값에는 아무것도 담겨있지 않지만



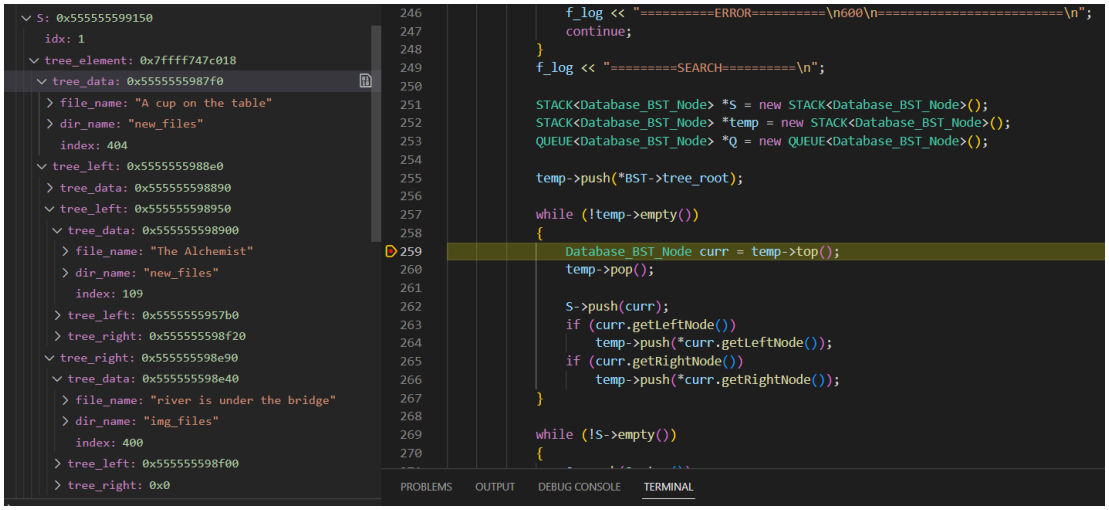
Inorder 순회가 진행되면서 BST 에 있는 값들이 출력되고 있는 것을 볼 수 있다.



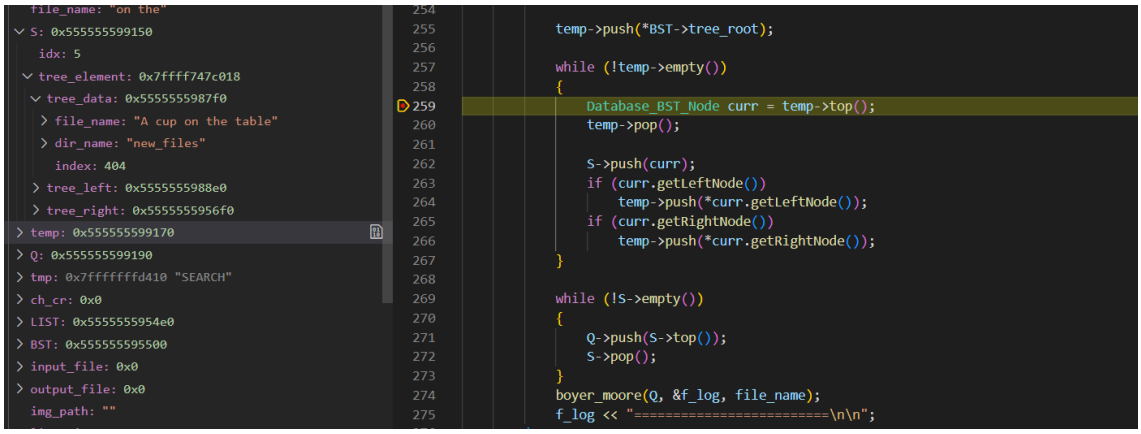
과제에서 제공된 파일의 특성 상 왼쪽 하위 트리에 방문하고 자신을 방문한다.



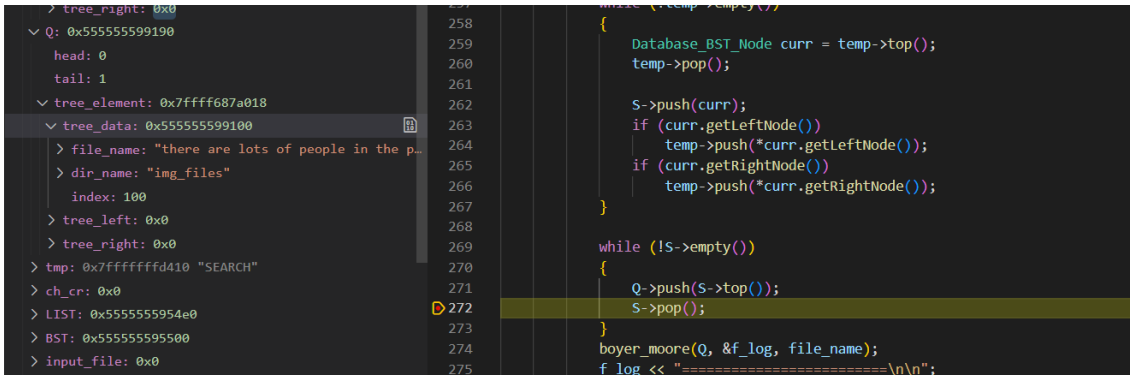
그 후 오른쪽 하위 트리에 방문하므로 index 값이 점점 커지고 있어 inorder 순회가 잘 동작함을 확인할 수 있다.



다음은 SEARCH 명령어다.

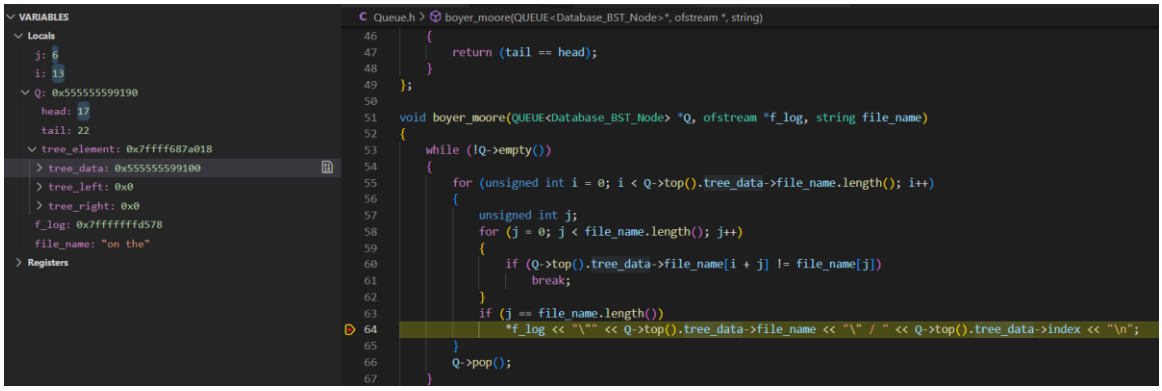


왼쪽의 idx 명령어를 보면 Stack 에 값이 채워지는 것을 확인할 수 있다.

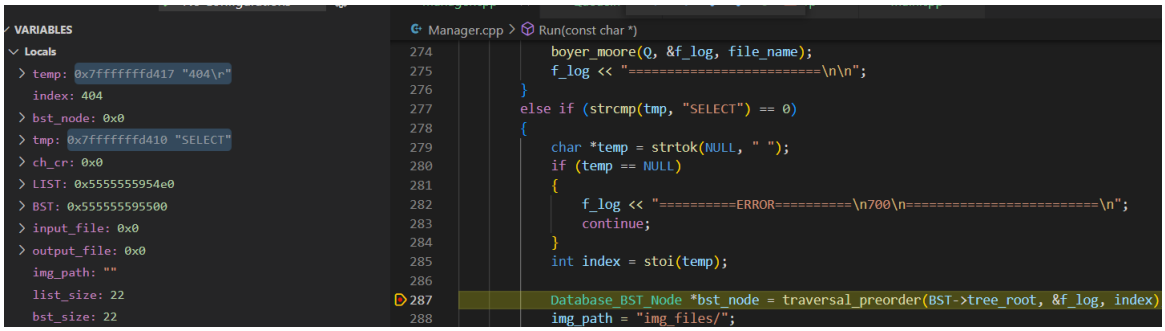


이제 pop 을 하기 시작하는데 앞서서 자신을 먼저 넣고 왼쪽, 오른쪽 노드를 채웠다. 현재 BST 의 root 는 왼쪽에 보이는 것처럼 인덱스가 100 이다. 그렇기 대문에 Q 에서의 첫번재 node 도 index 가 100 이다. 이 과정을 거치면 Q 에 BST 의 값들이 들어간다.

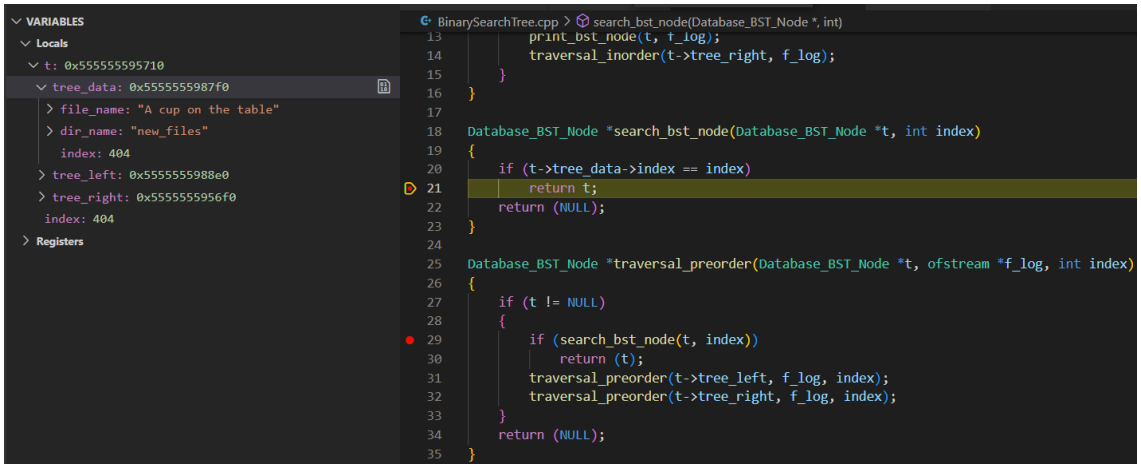
BST 의 값이 Q 에 들어간 것은 여기서 확인할 수 있다. Tail 이 22 개 증가했는데 이것은 filenumbers.csv 와 new_filenumbers.csv 의 파일 개수를 합한 것이다. 여기서는 on the 와 일치하는 문자열이 node 안에 있는지를 확인하는데 첫번째 문자에서 o 를 발견해 들어왔지만 그 다음 문자가 n 이 아닌 f 이기에 반복문을 탈출하는 것을 알 수 있다.



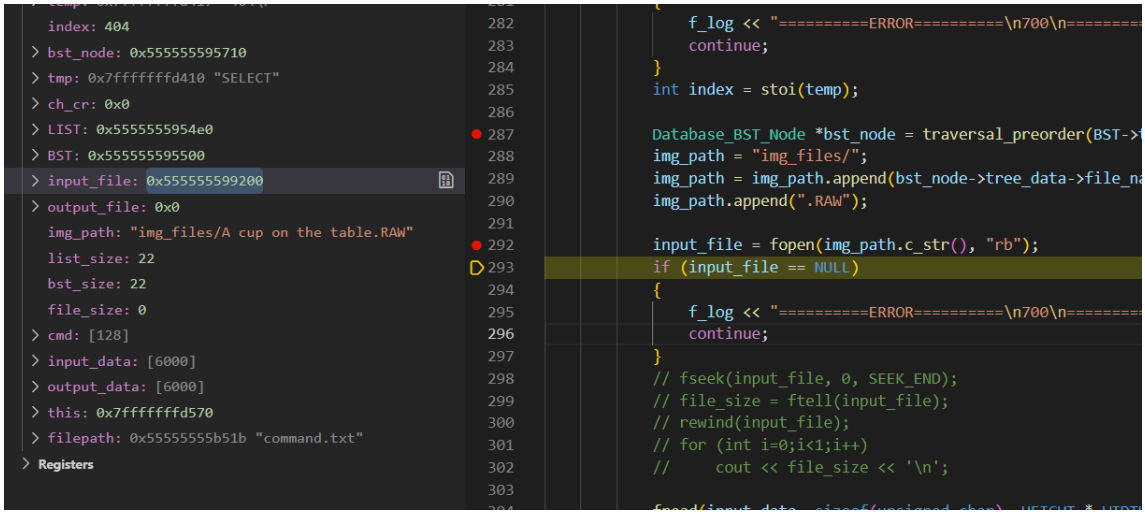
계속해서 검사하다가 file_name 과 6 글자가 동일 한, 즉 file_name 과 일치하는 node 를 찾았다. 그것을 j 값을 통해 확인할 수 있는데 j 가 6 인 것을 확인할 수 있다. 이렇게 보이어 무어 알고리즘이 작동하는 것을 확인할 수 있다.



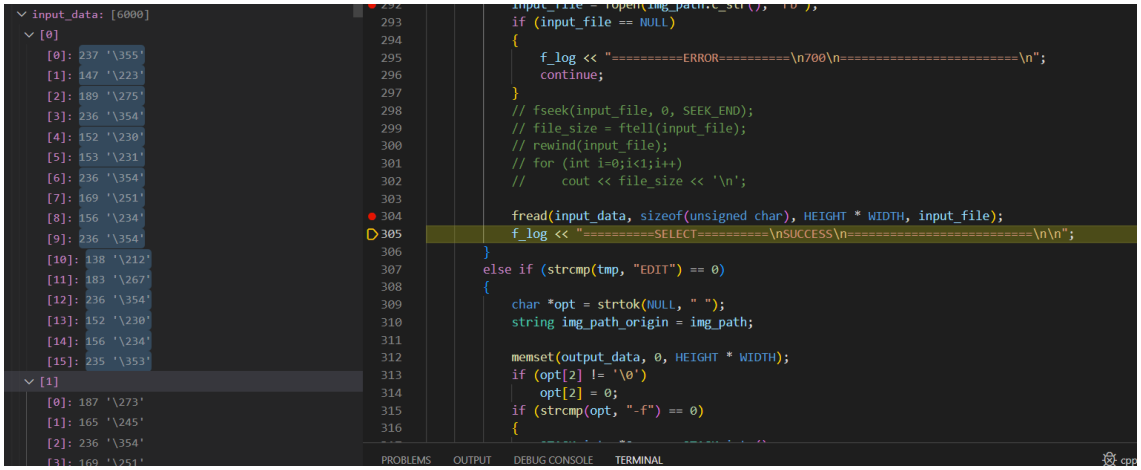
다음은 SELECT 다. Index 에 command.txt 에 정의된 index 를 잘 받아온 것을 확인할 수 있다.



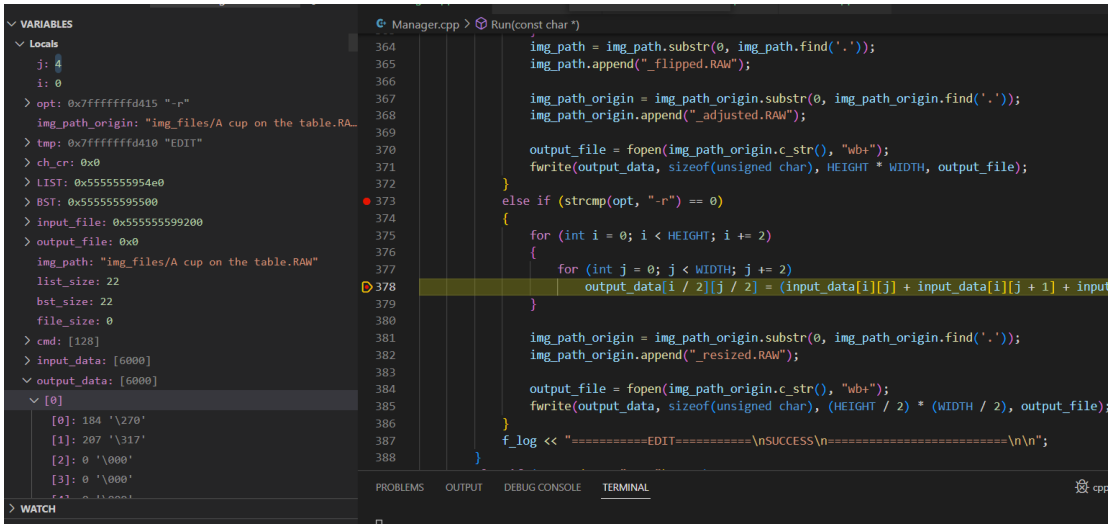
결과 값이 될 t 에 찾는 index 를 갖는 노드를 잘 가져온 것을 확인할 수 있다.

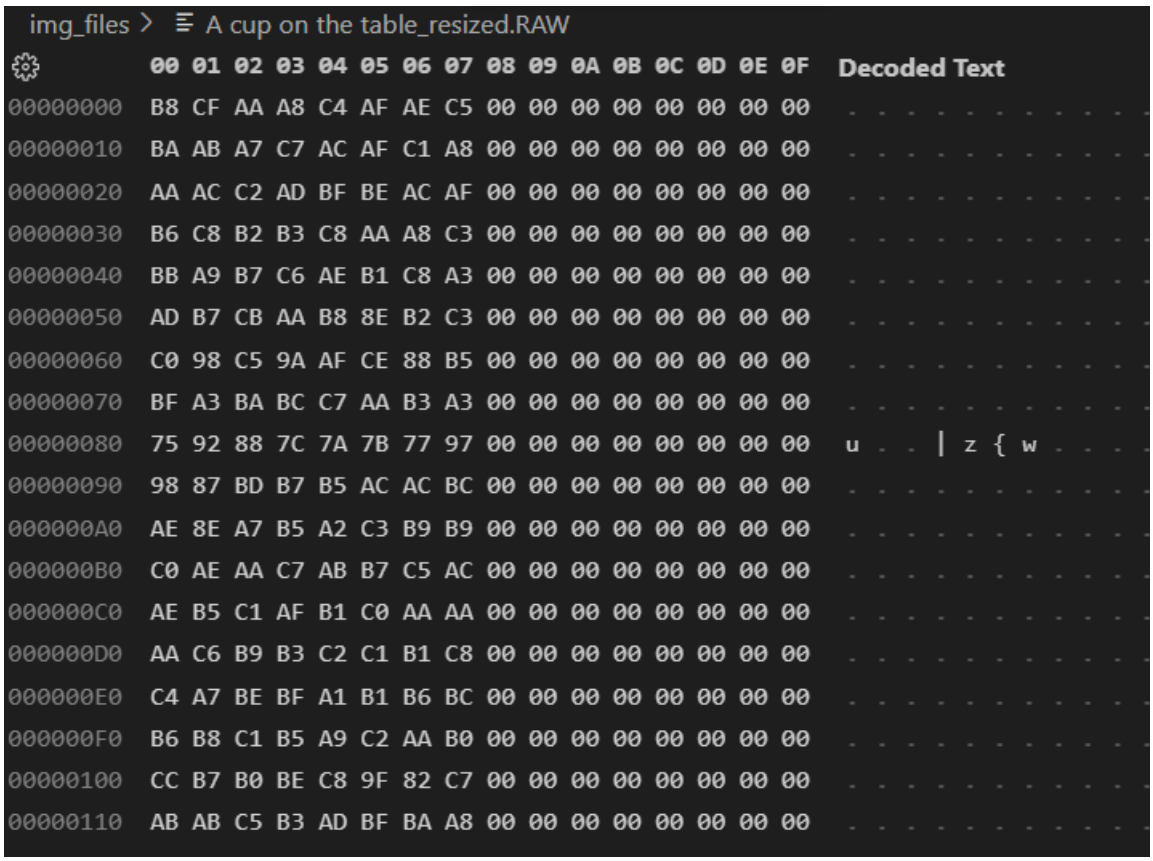


왼쪽을 보면 img_path 도 해당하는 노드의 위치를 잘 가져왔고 open 을 통해 가져온 파일 포인터도 NULL 이 아닌 것을 확인할 수 있다.

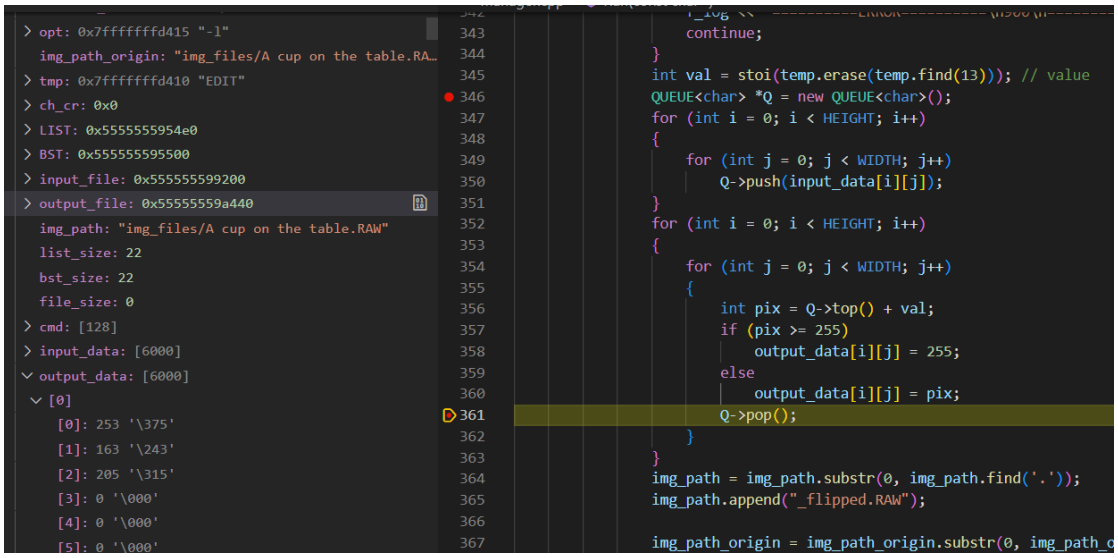


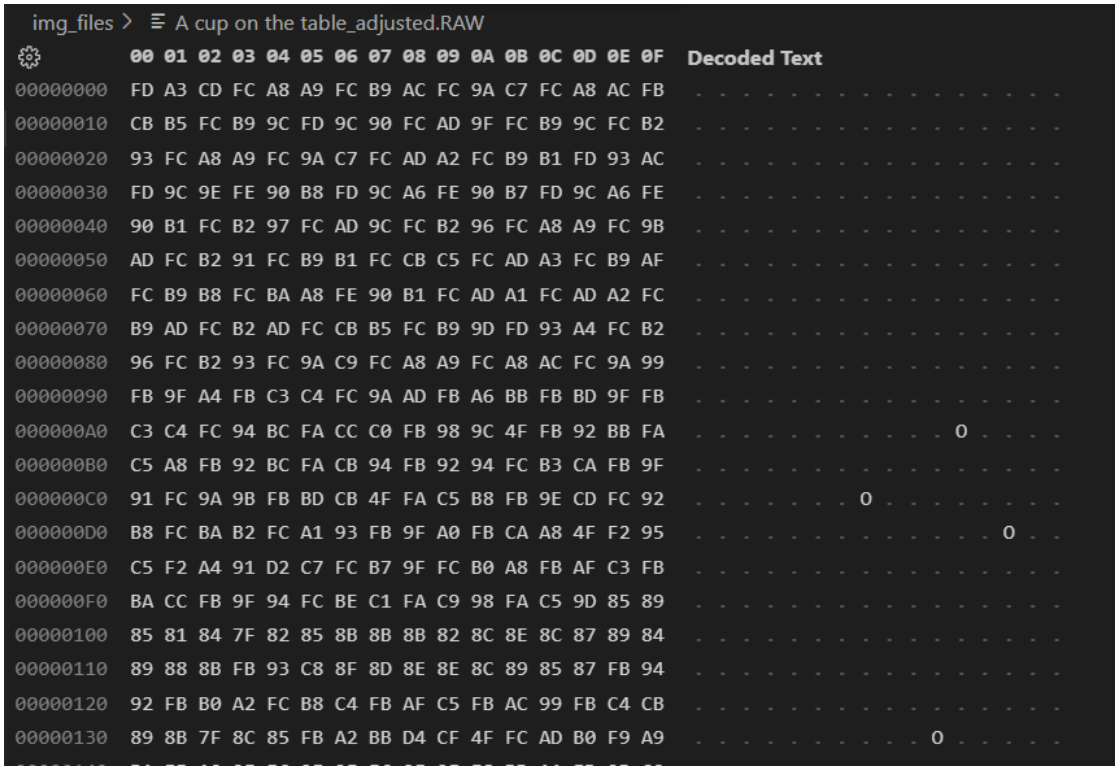
Fread 를 거치며 값이 배열에 잘 들어온 것을 확인할 수 있다.



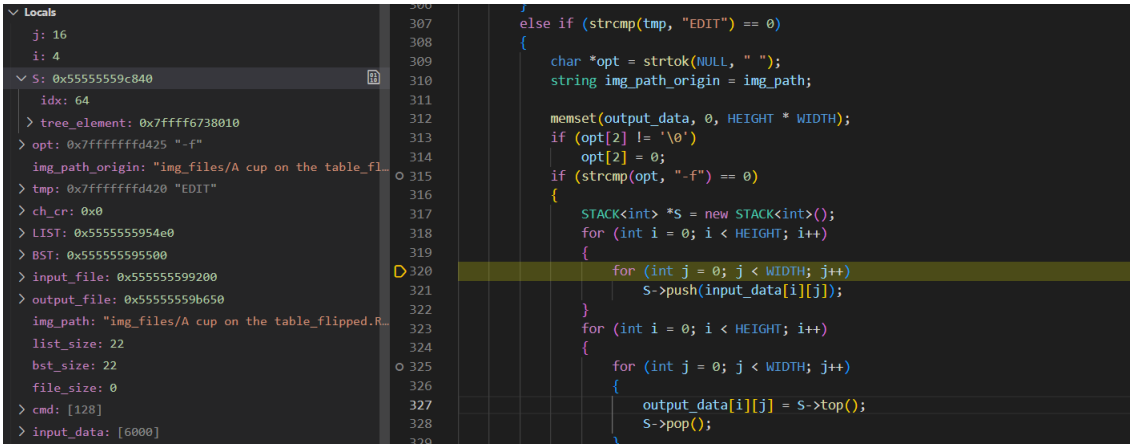


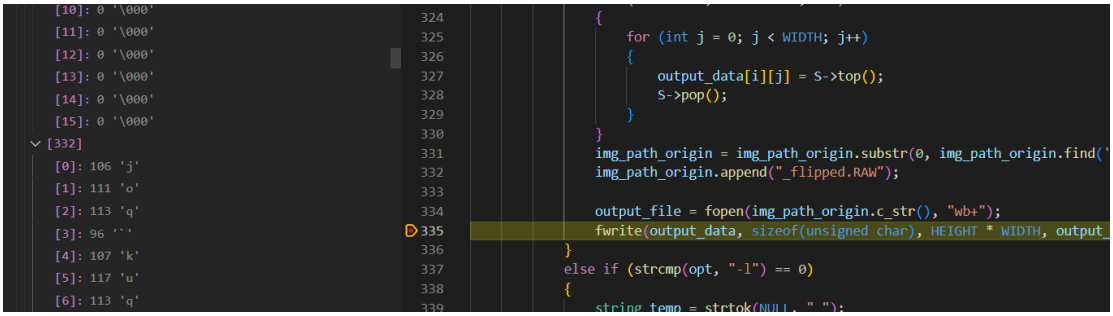
-r 옵션일 때 조건문을 잘 들어왔고 계산된 값, 즉 축소된 값이 배열에 담기는 것 또한 확인할 수 있다.

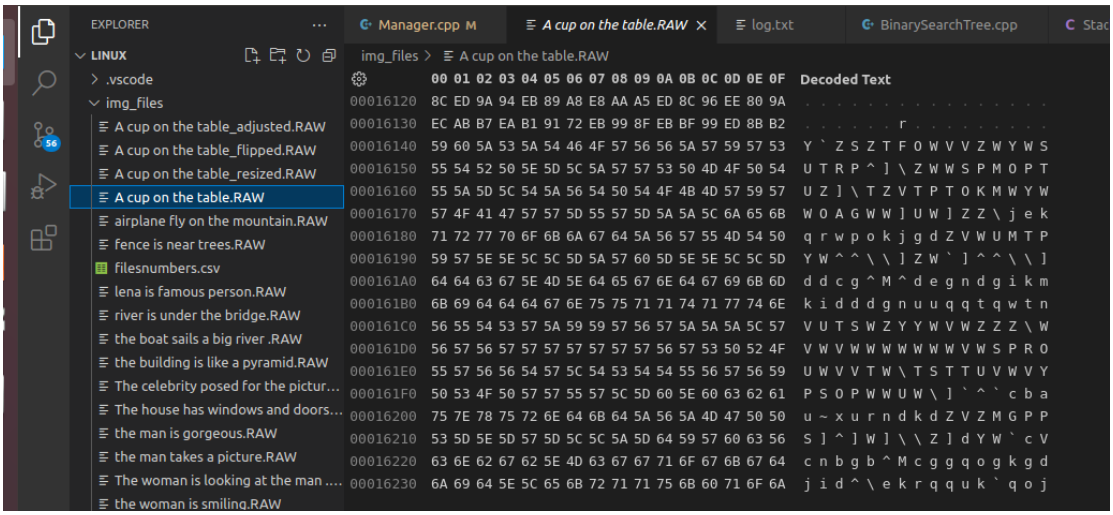


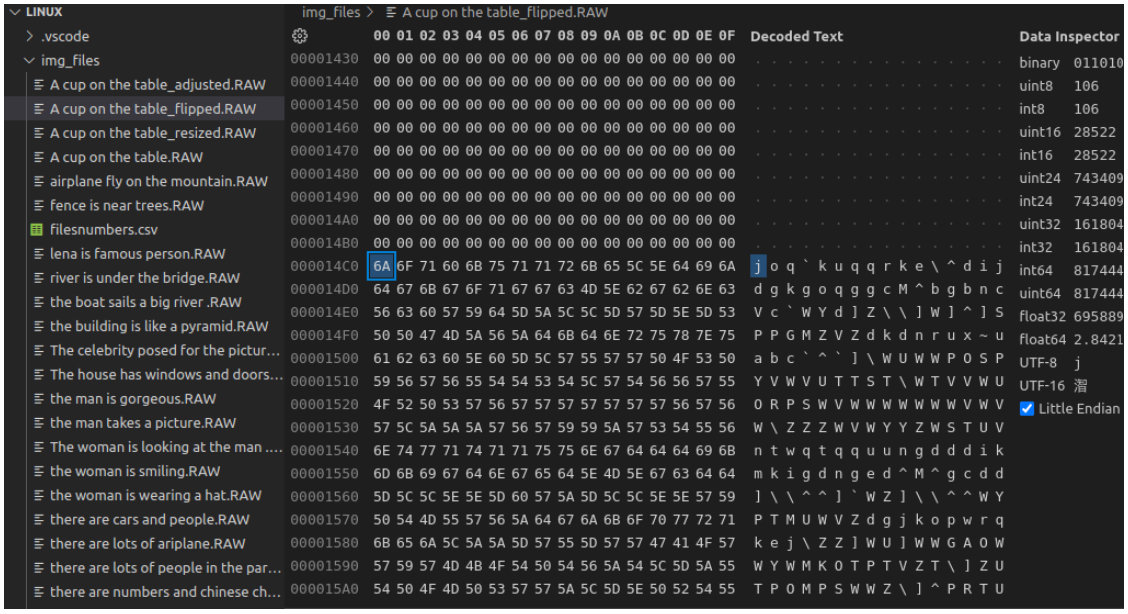


-I 옵션일 때 조건문을 잘 들어왔고 앞선 값에 16 이 더해진 값이 들어간 것을 확인할 수 있다.


Locals
j: 16
i: 4
S: 0x55555559c840
idx: 64
tree_element: 0x7ffff6738010
opt: 0x7fffffffd425 "-f"
img_path_origin: "img_files/A cup on the table_fl...
tmp: 0x7fffffffd420 "EDIT"
ch_cr: 0x0
LIST: 0x5555555954e0
BST: 0x555555595500
input_file: 0x555555599200
output_file: 0x55555559b650
img_path: "img_files/A cup on the table_flipped.R...
list_size: 22
bst_size: 22
file_size: 0
cmd: [128]
input_data: [6000]
else if (strcmp(tmp, "EDIT") == 0)
{
char *opt = strtok(NULL, " ");
string img_path_origin = img_path;
memset(output_data, 0, HEIGHT * WIDTH);
if (opt[2] != '\0')
opt[2] = 0;
if (strcmp(opt, "-f") == 0)
{
STACK<int> *S = new STACK<int>();
for (int i = 0; i < HEIGHT; i++)
{
for (int j = 0; j < WIDTH; j++)
S->push(input_data[i][j]);
}
for (int i = 0; i < HEIGHT; i++)
{
for (int j = 0; j < WIDTH; j++)
{
output_data[i][j] = S->top();
S->pop();


[10]: 0 '\000'
[11]: 0 '\000'
[12]: 0 '\000'
[13]: 0 '\000'
[14]: 0 '\000'
[15]: 0 '\000'
[332]
[0]: 106 'j'
[1]: 111 'o'
[2]: 113 'q'
[3]: 96 '`'
[4]: 107 'k'
[5]: 117 'u'
[6]: 113 'q'
{
for (int j = 0; j < WIDTH; j++)
{
output_data[i][j] = S->top();
S->pop();
}
}
img_path_origin = img_path_origin.substr(0, img_path_origin.find('
img_path_origin.append("_flipped.RAW");
output_file = fopen(img_path_origin.c_str(), "wb+");
fwrite(output_data, sizeof(unsigned char), HEIGHT * WIDTH, output
}
else if (strcmp(opt, "-l") == 0)
{


EXPLORER
LINUX
.vscode
img_files
A cup on the table_adjusted.RAW
A cup on the table_flipped.RAW
A cup on the table_resized.RAW
A cup on the table.RAW
airplane fly on the mountain.RAW
fence is near trees.RAW
filesnumbers.csv
lena is famous person.RAW
river is under the bridge.RAW
the boat sails a big river .RAW
the building is like a pyramid.RAW
The celebrity posed for the pictur...
The house has windows and doors...
the man is gorgeous.RAW
the man takes a picture.RAW
The woman is looking at the man ....
the woman is smiling.RAW
Manager.cpp M
A cup on the table.RAW
log.txt
BinarySearchTree.cpp
img_files > A cup on the table.RAW
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F Decoded Text
00016120 8C ED 9A 94 EB 89 A8 E8 AA A5 ED 8C 96 EE 80 9A
00016130 EC AB B7 EA B1 91 72 EB 99 8F EB BF 99 ED 8B B2
00016140 59 60 5A 53 5A 54 46 4F 57 56 56 5A 57 59 57 53 Y ` Z S Z T F O W V V Z W Y W S
00016150 55 54 52 50 5E 5D 5C 5A 57 57 53 50 4D 4F 50 54 U T R P ^] \ Z W W S P M O P T
00016160 55 5A 5D 5C 54 5A 56 54 50 54 4F 4B 4D 57 59 57 U Z] \ T Z V T P T O K M W Y W
00016170 57 4F 41 47 57 57 5D 55 57 5D 5A 5A 5C 6A 65 6B W O A G W W] U W] Z Z \ j e k
00016180 71 72 77 70 6F 6B 6A 67 64 5A 56 57 55 4D 54 50 q r w p o k j g d Z V W U M T P
00016190 59 57 5E 5E 5C 5C 5D 5A 57 60 5D 5E 5E 5C 5C 5D Y W ^ ^ \ \ ] Z W `] ^ ^ \ \ ]
000161A0 64 64 63 67 5E 4D 5E 64 65 67 6E 64 67 69 6B 6D d d c g ^ M ^ d e g n d g i k m
000161B0 6B 69 64 64 64 67 6E 75 75 71 71 74 71 77 74 6E k i d d d g n u u q q t q w t n
000161C0 56 55 54 53 57 5A 59 59 57 56 57 5A 5A 5A 5C 57 V U T S W Z Y Y W V W Z Z Z \ W
000161D0 56 57 56 57 57 57 57 57 57 57 56 57 53 50 52 4F V W V W W W W W W V W S P R O
000161E0 55 57 56 56 54 57 5C 54 53 54 54 55 56 57 56 59 U W V V T W \ T S T T U V W V Y
000161F0 50 53 4F 50 57 57 55 57 5C 5D 60 5E 60 63 62 61 P S O P W W U W \ ] ` ^ ` c b a
00016200 75 7E 78 75 72 6E 64 6B 64 5A 56 5A 4D 47 50 50 u ~ x u r n d k d Z V Z M G P P
00016210 53 5D 5E 5D 57 5D 5C 5C 5A 5D 64 59 57 60 63 56 S] ^] W] \ \ Z] d Y W ` c V
00016220 63 6E 62 67 62 5E 4D 63 67 67 71 6F 67 6B 67 64 c n b g b ^ M c g g q o g k g d
00016230 6A 69 64 5E 5C 65 6B 72 71 71 75 6B 60 71 6F 6A j i d ^ \ e k r q q u k ` q o j



-I 옵션일 때 기존 데이터의 값들이 거꾸로 담기는 것을 확인할 수 있다.

- Consideration

  프로젝트를 진행하면서 많은 우여곡절과 코드 구조 개선 작업이 있었다.

  첫번째는 strtok 였다. strtok 는 두번째 인자로 전달된 문자를 첫번째 인자로 전달된 문자열에서 찾아 해당 부분을 NULL 로 바꾸고 문자열의 주소 값을 리턴 한다. Strtok 를 여러 번 하게 되면 앞서 tok 된 이후부터 두번째 인자로 전달된 인자의 탐색을 시작하는데 탐색에 성공하면 tok 된 이후 문자의 주소 값을 리턴 한다. 이때 주의 깊게 봐야할 것이 있다.

  strtok 는 주소 값을 리턴할뿐 새로운 문자열을 생성하지는 않는 것이다. 즉 새로운 문자열의 생성이 아닌 원본 문자열의 변형인 것이다. 이부분을 알지 못해 free 함수를 남발했고 double free, segmentation fault 를 다수 마주했지만 vscode 의 디버그 기능으로 한줄씩 실행해가며 해결할 수 있었다.

  두번째는 getline 이었다. Getline 은 파일의 내용을 원하는 만큼 읽어서 배열에 저장할 수 있다. 문제는 파일의 내용을 한줄씩 읽을 때마다. 무작위로 carriage return, 즉 ascii 코드 값 13 의 수가 랜덤으로 붙는 것이었다. strtok 함수로 잘린 명령어와 실제 명령어의 비교, strtok 함수로 잘린 옵션과 실제 옵션의 비교 등에 strcmp 함수를 사용했는데 cout 으로 확인 출력 값은 동일하지만 반환 값이 0 이 나오지 않고 자연수 13 이 나오는 오류가 있었다. 실제로 for 문으로 문자열의 문자 각각을 출력해보니 예상한 문자열 뒤에 ascii code 13 의 값을 갖는 carriage return 이 존재하였다.

  Carriage return 을 지우기 위해 getline 다음 줄마다 for 문을 사용해 마지막 문자를 null 로 바꾸는 작업을 진행했지만 양이 너무 많이 불필요하게 코드의 길이가 늘어나게 되었고 이것을

remove_cr 라는 임의의 함수로 선언했지만 string 헤더 파일을 사용해도 된다는 힌트를 얻어 string 헤더파일 내의 편리한 erase, find 등의 메소드로 carriage return 을 제거하였다.

세번째는 ADD 의 push 기능 구현이었다. 과제에 제시된 조건에 2 차원 연결 리스트를 사용한다가 명시되어 있었다. 처음 생각한 방식은 Loaded_LIST 의 노드 각각에 prev, next, top, down 을 가리키는 포인터를 주고 add 가 들어오면 node 각각을 연결하는 격자무늬의 연결 리스트였다. 구조상의 문제는 없었다. 각 리스트 가장 앞 노드의 top, down 만 신경 써 주면서 push, pop 을 할 때 포인터를 변경해주면 정상적으로 동작한다. 하지만 과제의 "구현 시 반드시 정의해야하는 Class"에 Loaded_LIST, Loaded_LIST_Node 가 분리되어 있는 것을 늦게 확인했고 LIST 를 구성하는 Node 와 Loaded_LIST 를 구성하는 List, 이 2 개로 구조를 분리했고 결과적으로 낭비되는 메모리 공간을 압도적으로 줄일 수 있었다. 노드 한 개당 차지하는 메모리가 절반으로 줄었기 때문이다.

네번째는 class 의 생성자에 관련한 문제다. TreeNode.h 파일에서 과제의 조건에 따라 BST 를 구성하는 Database_BST_Node class 를 만들었다. 이렇게 만든 class 를 Database_BST class 에서 사용할 때 new 로 할당이 안되는 오류가 있었다. 오류의 메시지는 "no default constructor exists for class Database_BST_Node"였다. 당시 class 의 생성자는 member initializer 로 사용하고 있었다. 이것을 생성자 안에서 직접 초기화 하는 방식으로 바꿔도, 멤버 변수들의 타입을 바꿔도 여전히 같은 오류가 떴다. 결국 생성자를 지우니까 사라졌는데 그 이유는 아래와 같았다.

다른 생성자를 정의하지 않고 class 를 사용한다면 compiler 가 자동으로 default constructor 를 생성해주지만 생성자를 정의한다면 자동으로 default constructor 를 만들지 않는다. 또한 C++에서 default constructor 는 사용자의 생성자 정의 여부와는 관계없이 인자를 제공하지 않는 생성자를 뜻한다는 것이다. 따라서 member initializer 로 사용하는 것 이외에 인자도 없고 동작도 정의하지 않은 기본 생성자를 만들어서 해결할 수 있었다.

다섯번째 문제는 코드 리펙토링 도중에 발생했다. 처음에는 STACK, QUEUE 자료구조를 template 으로 만들지 않았다. 필요한 자료구조마다 스택, 큐를 만들었지만 헤더파일의 길이도 길어지고 정확히 동일한 동작을 자료구조만 바꿔서 하기 때문에 동일한 코드를 두 번, 세 번 작성하게 되었다. 이때 스택, 큐에 들어갈 데이터의 최대 크기를 고려하지 않아서 문제가 발생했다. 파일을 1 바이트 씩 읽어오면 제공된 파일의 최소 크기인 6 만 5 천개가 필요했다. 하지만 자료구조의 사이즈는 최대 512 였다. 때문에 로직에 문제는 없지만 스택, 큐에 담아야 할 데이터가 너무 많아서 segmentation fault 가 발생했고 스택, 큐의 최대 사이즈를 조정해 해결할 수 있었다.

여섯번째 문제는 SEARCH 기능 구현 도중 발생했다. 반복문을 사용한 post order traversal 의 개념이 굉장히 낯설었기 때문이다. 큐를 사용해 post order traversal 를 구현하려고 했었다. 하지만 큐의 구조적인 한계 때문에 root 로 되돌아오는 것이 root 왼쪽 오른쪽 이렇게 순회하기 위해서 root, 왼쪽, 오른쪽을 각각 저장하는 연결리스트를 만들었지만 과제에서 제시한 방향과

맞는 것인지 의문이 들었다. 왜냐하면 결국 만들어진 것은 전위 순회가 아니라 큐라는 이름의 연결리스트 3 개를 돌아가면서 각각 원소를 가져오는 것이기 때문이다.

그러나 문제의 조건에서 가져온 element 를 집어넣을 때 사용하는 것이 QUEUE 지 순회도 QUEUE 를 사용해 구현하라는 말이 없어서 다른 방식으로 구현하는 방법을 시도했다. 그 결과 STACK 에 root 를 저장해 내려갔다가 다시 되돌아올 수 있었고 코드의 길이를 대폭 줄일 수 있었다.

일곱번째 문제는 EDIT 기능을 구현할 때 발생했다. 과제에서 rawreader 로 준 예제 코드에서는 512 * 512 배열을 사용한다. 그에 반해 과제에서 제공한 파일의 최대 크기는 아무리 커도 9 만 바이트를 넘지 않았다. Fread 를 통해 파일을 읽게 되면 절반의 배열에는 값이 할당되지만 남은 절반의 배열은 값이 할당되지 않는다. 때문에 -f 옵션 즉, stack 을 사용해 역순으로 정렬할 때 마지막에는 초기화되지 않은 값이 들어갔고 쓰레기 값을 제거하기 위해 전부 0 으로 초기화해도 의미는 없었다. 그래서 RAW 파일에 맞춰서 가로, 세로를 512, 512 로 고정하는 것이 아닌 16, 6000 으로 조정했다. 혹시라도 발생할 수 있는 오류에 대비해 16, 6000 을 하드코딩하지 않고 #define 으로 매크로 상수를 정의해 반복문, 배열 크기 등을 조절할 수 있게 해였다. 그 결과 온통 0 으로 채워진 filpped 를 확인할 수 있게 되었고 resized 는 예상된 값이 출력된 것을 (ex: (ED+93+BB+A5) /4 == B8) 확인할 수 있었다.