<자료구조 7차 과제 – Shortest Path>

소프트웨어학과 20172073601균

1. Data Structure

1. 개요

이번 과제는 6차 과제와 매우 밀접하다. 다만 저번 과제와 다르게 도시의 입력에서 도시간의 cost를 함께 입력해서 도시간의 연결에 cost가 부과된다.
저번 과제와 마찬가지로 adjacency list를 만들고, BST를 만들어서 중복과 잘못된 입력을 체크한다. 이후 shortest path algoritm을 통해 distance와 pi 배열을 만들고, 이를 통해 출발노드부터 도착 노드까지의 path순서와 shortest path의 cost를 출력하는 과제이다.

2. 6차 과제와 동일하게 사용한 데이터 구조

a) **static int** __parse_command(**char** *command, **int** *nr_tokens, **char** *): 도시들이 입력으로 들어올 때 도시의 개수만큼 입력이 들어온다. 이때 도시들은 공백으로 구분되어 들어온다. 이때 공백 문자를 기준으로 도시 문자열의 입력을 파싱하여 저장하기 위해서 __parse_command함수를 선언하였다. 추후 도시간의 연결을 확인할 때에도 이 함수를 사용하여 -문자를 기준으로 ₩0을 삽입하여 문자열을 파싱하여 연결되는 도시들을 분리해내었다.

char command[MAX_COMMAND] = {'\0'}; : 입력되는 도시 문자열을 저장하기 위해서 사용하였다.
char *tokens[MAX_NR_TOKENS] = { **NULL** }; 파싱한 토큰을 저장하기 위해 사용하였다.
int nr_tokens = 0; : 파싱한 토큰의 개수를 저장한다.

b)
```
typedef struct node *nodePointer;
typedef struct node {
    int vertex;
   int rental;
    char cityName[10];
    nodePointer link;
}Node;
Node adjListsArr[MAX_VERTICES];
nodePointer avail = NULL;
```

adjacency list를 만들기 위해서 선언하였다. adjListArr 는 들어오는 순서에 따른 vertex와 이름인 cityName, link field를 가진다. 추후 사용하지 않는 노드를 재사용하기 위해 avail을 사용하여 메모리를 재활용하였다.
따라서 입력에 맞게 파싱된 도시들을 adjListsArr[i].vertex와 adjListsArr[i].cityName에 추가해주고, 입력에 맞게 BST를 만든 후에 도시 연결 입력이 들어오면 연결된 node를 adjListsArr[i]에 추가해주면서 adjacency list를 만들었다.
adjacency list를 만들 때 도시와 도시간의 cost가 추가되어야 한다. 따라서 rental이라는 변수를 추가로 선언하여 cost를 담게 하였다.

c) **typedef struct** treeNode* treePointer;

```
typedef struct treeNode {
    char key;
    int index;
    char cityName[10];
    treePointer leftChild;
    treePointer rightChild;
} treeNode;
treePointer root = NULL;
treePointer insert(treePointer node, char* str, int
index){}
treePointer newNode (char* str, int i){}
treePointer search(treePointer root, char* str){}
```

tree 자료들을 구조체로 선언해주었다. char key;는 BST에서 크기를 비교하기 위해 도시의 첫 글자 알파벳을 저장하기위한 key값이고, int index;는 입력순서에 맞는 index값으로 사용하였다. 또한 도시 이름을 위한 char cityName[10];, child node를 위한 treePointer leftChild; treePointer rightChild;를 선언하였다.
트리의 root를 나타내도록 treePointer root = NULL;을 사용하였고, 입력으로 들어오는 도시들을 insert와 newNode함수를 통해 BST로 만들었다.
처음 도시이 개수만큼 도시의 입력이 들어오면 insert()함수를 통해서 BST를 만들고(이때 알파벳 순서로 크기비교를 한다), 이후 도시간의 연결이 입력으로 들어오면 search()함수를 통해 BST를 search하면서 BST에 입력된 도시가 없으면 잘못된 입력이고, BST에서 찾은 노드들이 이미 adjacency list에 만들어져있는지를 check_dup에서 확인한다.

d) `int check_dup(treePointer ptr0, treePointer ptr1){}:`
중복을 확인하기 위한 함수이다.
우선 도시들의 연결이 입력으로 들어오면 그 연결에 맞는 adjacency list를 만들게 된다. 이후 똑같은 연결이 들어온다면 adjacency list에서 이미 연결이 추가되어있을 것이므로 list를 스캔하며 입력과 똑같은 연결이 있는지 체크하면 된다. 따라서 만약 도시의 연결이 mn-kl인데 똑같은 입력인 mn-kl이 들어온다면 이미 adjacency list에 mn과 kl이 연결된 상황일 것이고, 두번째 입력에 대해서 리스트를 스캔하며 adjacency list의 vertex와 입력된 도시의 index를 비교해서 중복성을 체크한다.

```
while(temp && temp1){
  if(temp->vertex == ptr0->index && temp1->vertex == ptr1->index){
      return TRUE;
  }
  temp1 = temp1->link;
}
```

예를들어 도시의 입력이 mn-kl이라면 temp->vertex는 adjacency list에서 cityname이 mn인 노드의 vertex일 것이고, temp1의 링크를 계속 증가하면서 리스트를 스캔하며 BST에서 찾은 노드의 index와 같은지 비교하였다.

e) `void createAdjList(treePointer ptr0, treePointer ptr1, char* cost){}`
```
void connected(){}
void dfs(int v){}
short int visited[MAX_VERTICES];
```

도시의 연결이 입력으로 들어온다면 그 연결에 맞는 adjacency list를 만들어야 한다. 따라서 createAdjList를 이용하였다.
우선 사용하지 않는 노드의 저장을 위한 avail이 남아있는지의 여부에 따라 메모리를 reuse하게 하였고, list를 추가할 때 맨 앞부분에 추가하는 방식으로 코드를 구성하였다. 따라서 O(1)의 시간이 걸리도록 하였다.

추가로 도시들이 몇 개의 component로 분리되는지 확인하기 위해서 도시의 연결성이 끊기는지 아닌지 확인해야 한다. 따라서 adjacency list에서 dfs를 이용하면 연결이 끊기는지 안닌지 확인할 수 있다. connect와 dfs함수를 통해 그래프가 연결되었는지 아닌지를 visited 배열을 통해 알 수 있다. visited의 값이 0인 인덱스가 있다면 연결이 끊긴것이고 추후 이는 shortest 함수에서 choose함수를 통해 나온 인덱스 u가 visited일때만 TRUE로 바꿔주게 하는 역할을 한다.

3. void shortestPath(int v, int distance[], int n, short int found[], int pi[]);

shortestPath 함수에서

```
nodePointer temp = adjListsArr[v].link;
 for(int i = 0; i < n; i++)
{
   distance[i] = MAX_COST;
   found[i] = FALSE;
   pi[i] = -1;
}
found[v] = TRUE; distance[v] = 0;
while(temp)
{
   distance[temp->vertex] = temp->rental;
   pi[temp->vertex] = v;
   temp = temp->link;
}
```

위의 코드에서 첫번째 for문에서는 shortest path의 cost를 담는 distance array를 초기화 하며 found를 0으로 초기화 하고, pi 즉 predecessor id를 담는 array를 -1로 초기하 한다. 이후 v로 들어오는 즉 처음 입력하는 도시의 index를 가지는 adjacency list를 가리키는 temp의 링크를 스캔하며 처음 입력되는 도시와 adjacent한 도시들의 cost를 distance array에 추가해주고 있다. 추가로 pi array역시 v로 바꿔주는데 이는 인접한 도시들은 처음 출발 노드를 거쳐가기 때문에 predecessor id가 v가 되기 때문이다.

4. choose(int distance[], int n, short int found[]);

choose 함수는 found가 되지 않은 도시들 중에서 가장 cost가 작은 도시의 index를 리턴한다.

5.
```
u = choose(distance, n, found);
 if(visited[u])
      found[u] = TRUE;
 temp1 = adjListsArr[u].link;
 while(temp1){
if(!found[temp1->vertex])
  {
    if(distance[u] + temp1->rental < distance[temp1->vertex]){
       distance[temp1->vertex] = distance[u] + temp1->rental;
       pi[temp1->vertex] = u;
    }
  }
  temp1 = temp1->link;
}
```

위의 코드에서 shortestPath 함수에서 choose함수를 통해 리턴된 인덱스가 v와 connected가 아닐때 TRUE로 found array를 바꿔주고 있다. 또한 u의 adjacency list를 scan하면서 shortest path를 찾지 못한 vertex w에 대해서 distance 값을 보정하고, distance 값을 보정

한 경우, pi[w]=u로 설정하고 있는데 temp1을 u의 adjacency list를 가리키게 함으로써 temp1과 adjacent한 노드 중에서 이미 방문한 노드가 아닌, 즉 과제 참고자료에서 ab-mn-kl의 연결을 봤을 때 mn은 이미 found된 ab를 제외하고서 mn과 adjacent한 노드들 중에 mn을 거쳐가면 만약 거리가 더 짧아지는게 있나 확인해야 한다. 따라서 링크를 증가시키며 u를 거쳐가면 cost가 줄어드는 노드들의 distance 값을 바꿔주고 pi의 값도 바꿔주고 있다.

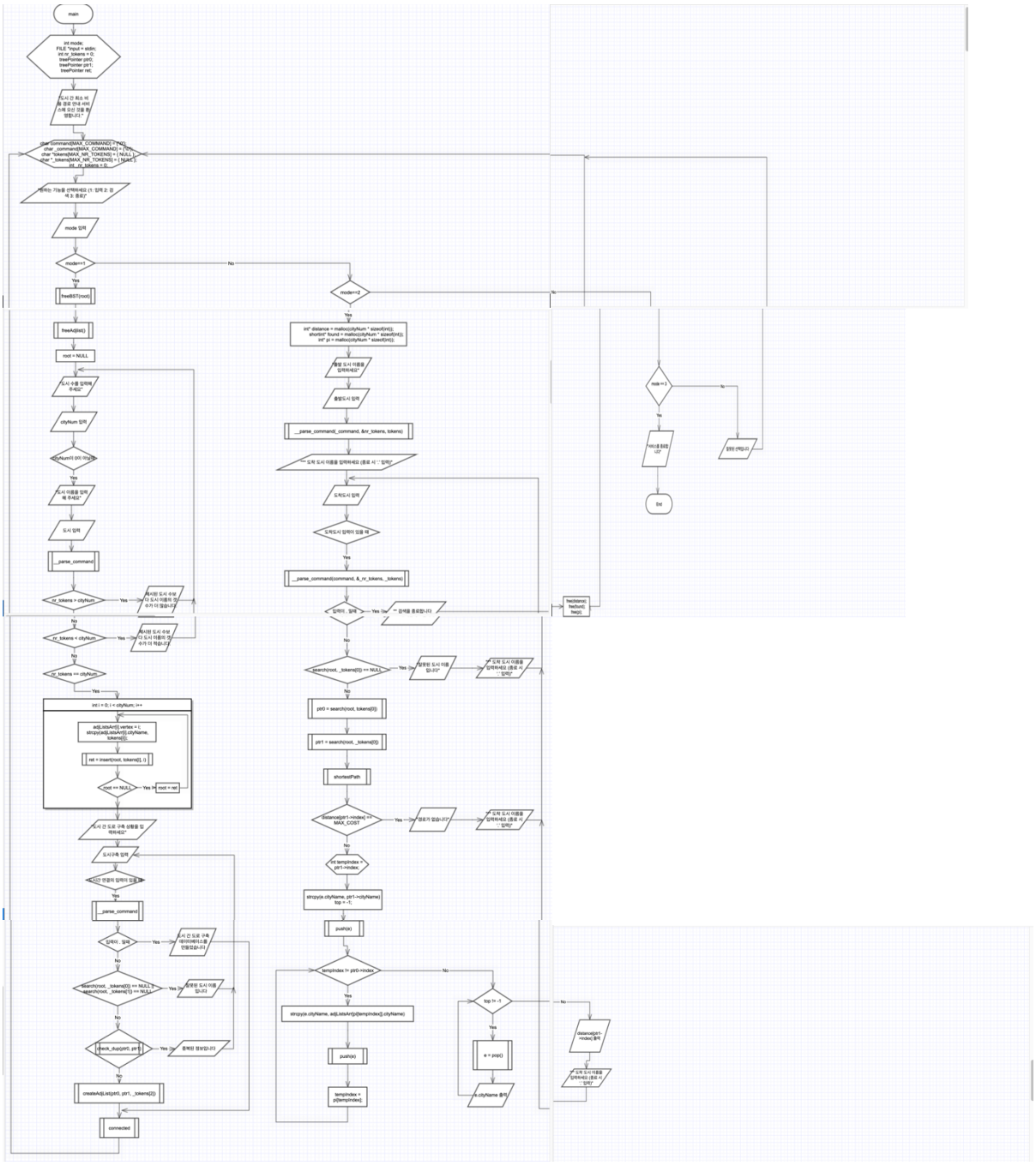
6. int tempIndex = ptr1->index;

shortest path를 출력 시 pi의 index를 역으로 참조하면서 출력을 진행해야 하는데 이때 BST의 index를 임시 저장하기 위해서 tempIndex라는 변수를 선언하였다. 이후 stack에 pi를 역참조하며 push하고, 다시 pop하면서 출력을 하고 있다.

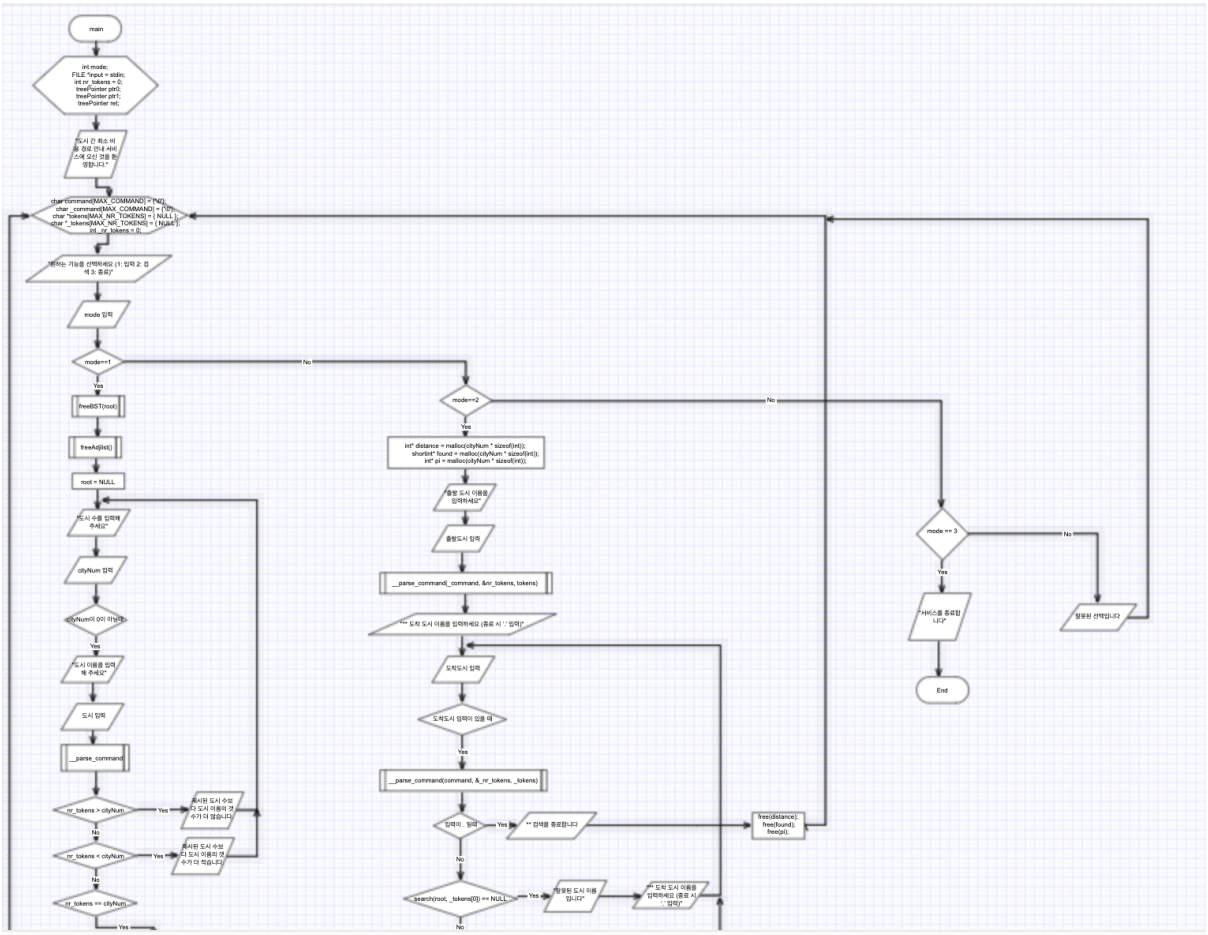
```
element stack[MAX_VERTICES];
element e;
int top = -1;
typedef struct {
    char cityName[MAX_CITYNAME];
}element;
```

다음은 stack을 위한 자료구조이며 element에 도시 이름을 담을수 있게 하였다.

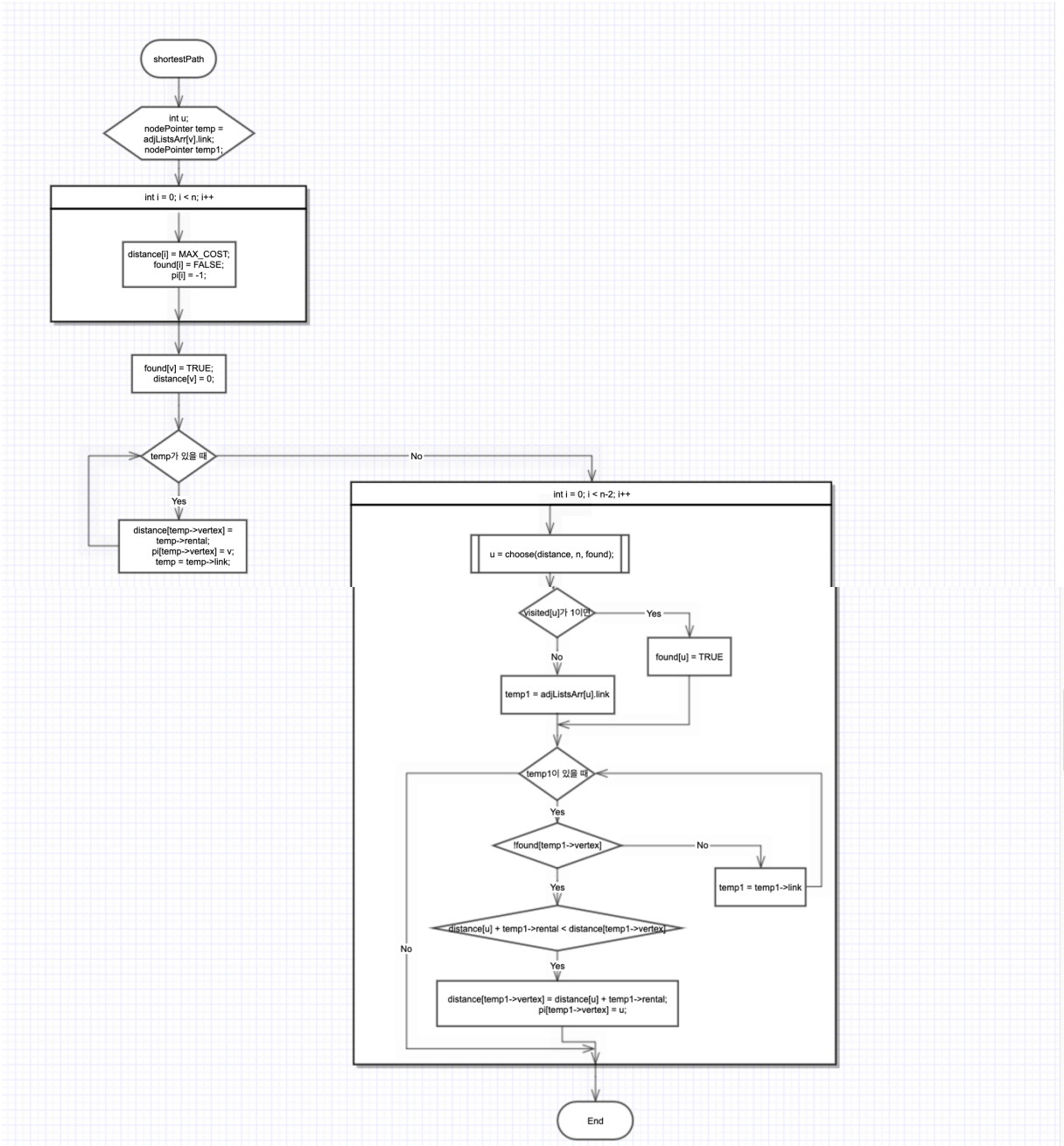
2. Flow chart



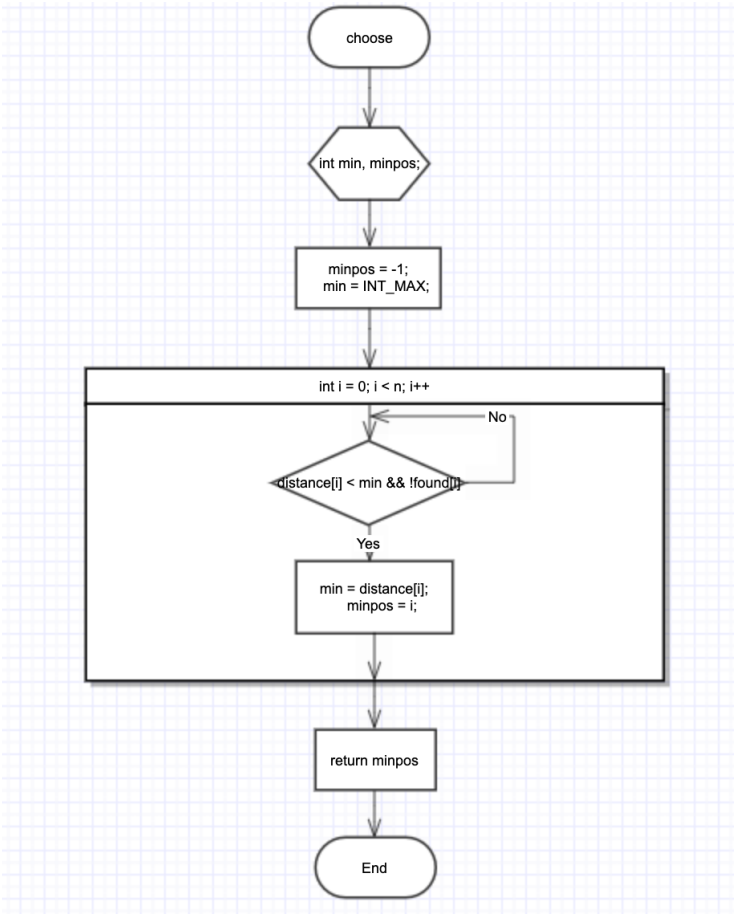
main함수



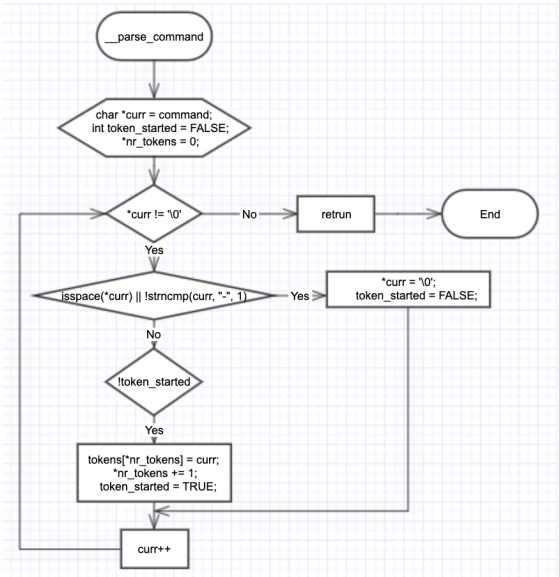
참고 - main함수(화질 안좋음)



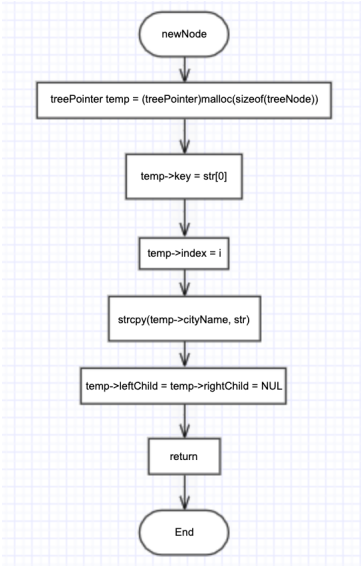
shortestPath함수



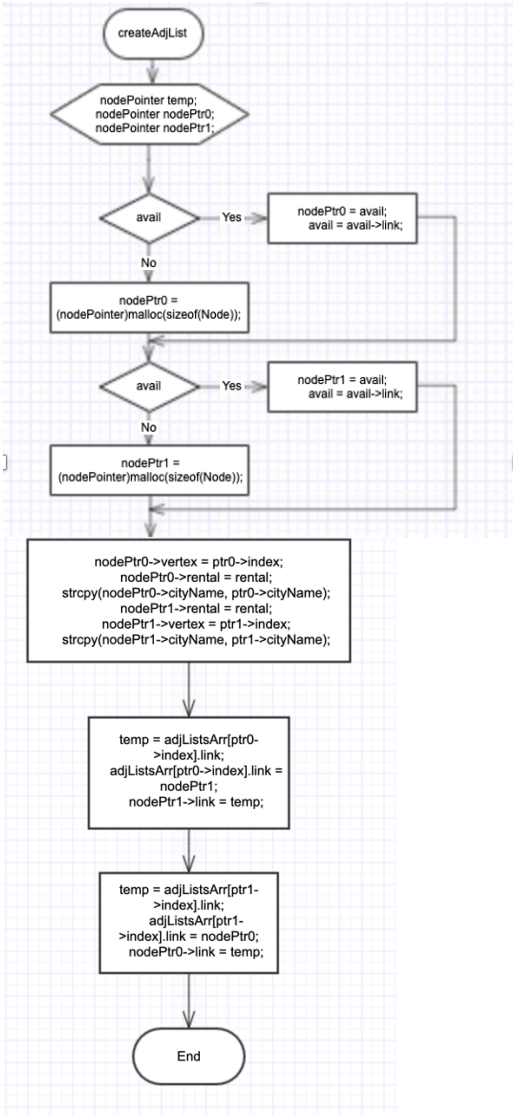
choose함수



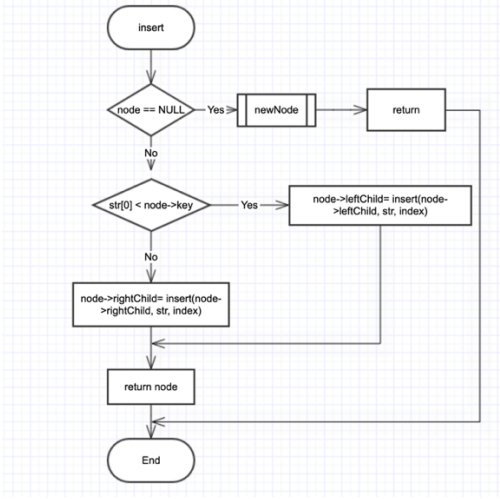
__process_command 함수



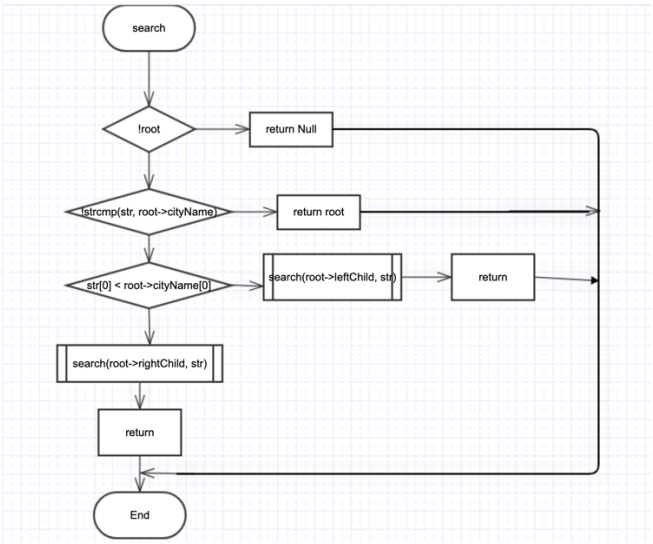
newNode 함수



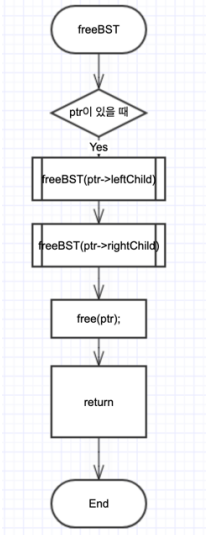
createAdjList함수



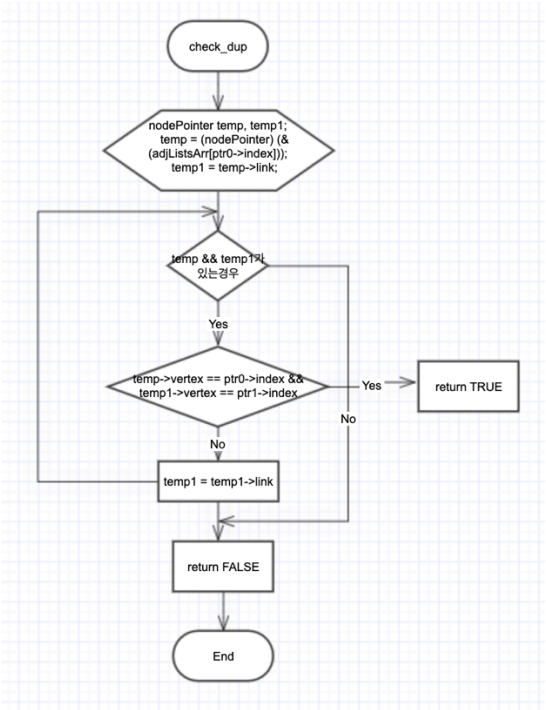
insert 함수



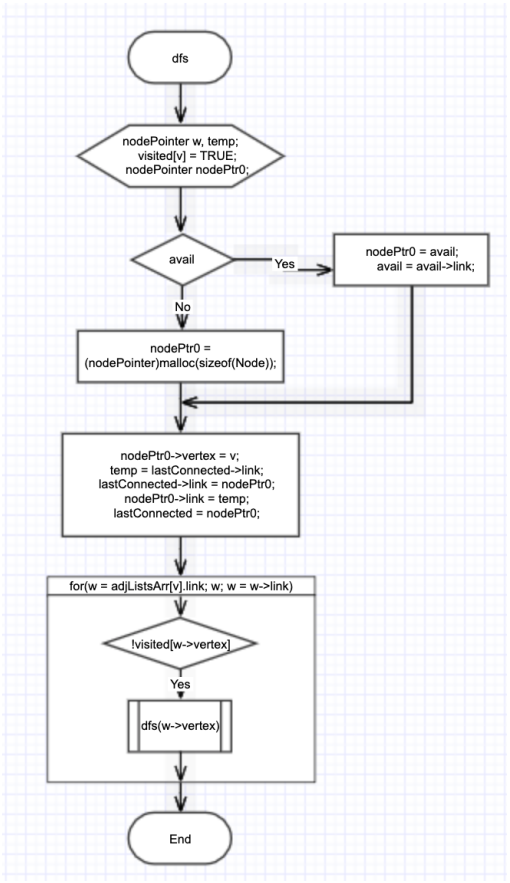
search 함수



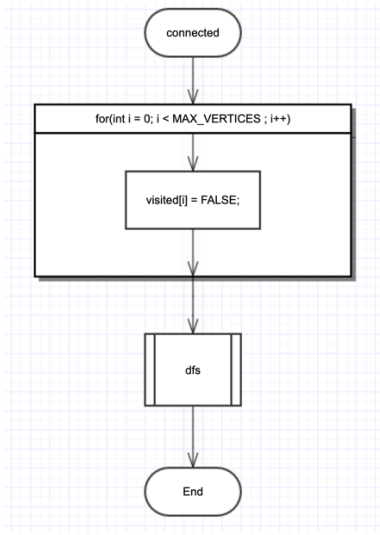
freeBST함수



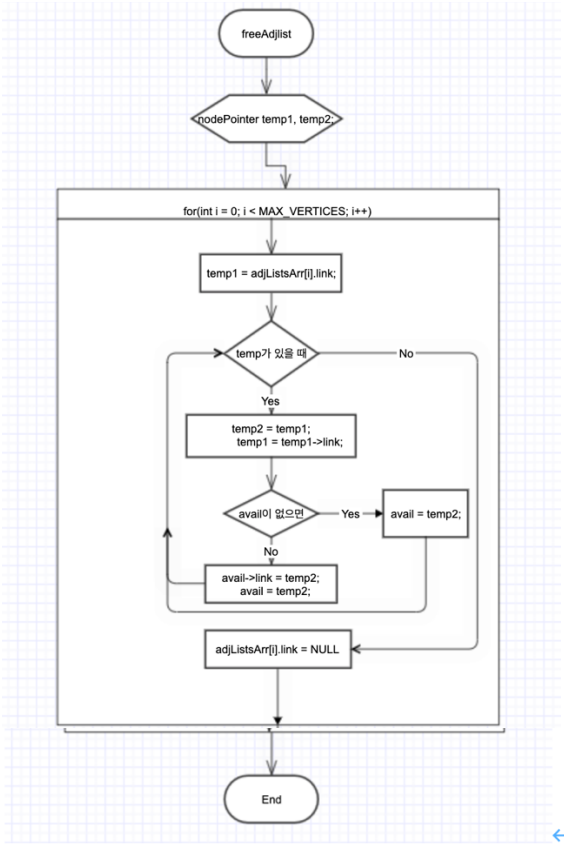
check_dup 함수



dfs 함수



connected 함수



freeAdjlist 함수