2021.10.04

자료구조2 실습 레포트

과목명 | 자료구조2 실습

담당교수 | 홍 민 교수님

학과 | 컴퓨터소프트웨어공학과

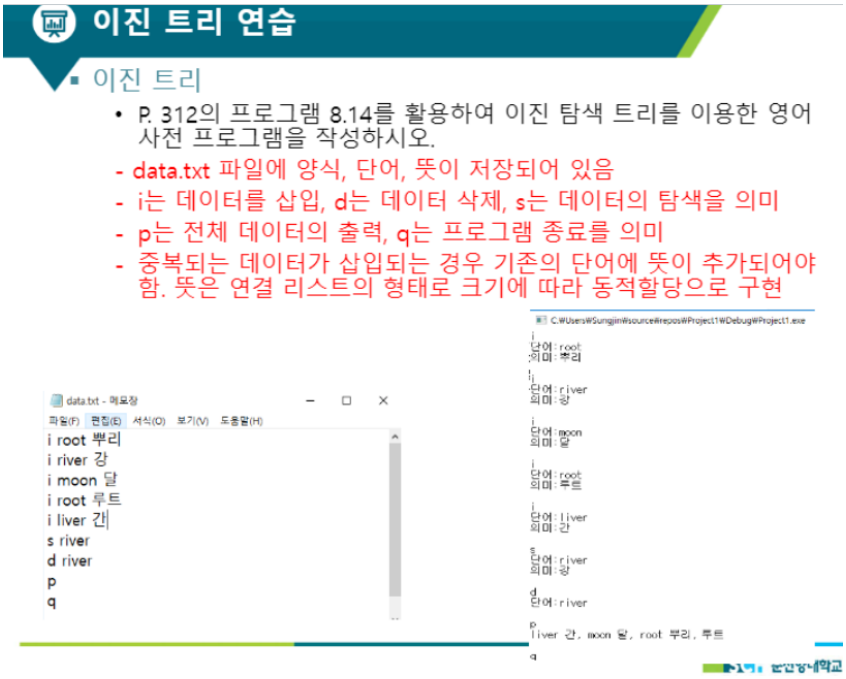
학년 | 2학년

학번 | 20204059

이름 | 이예빈

목 차

1. 이진 트리로 구현한 영한 사전



1.1 문제 분석

이 문제는 이진 탐색 트리를 이용하여 단어와 뜻을 저장하는 영한 사전 프로그램이다. data.txt에 양식에 따라 각 줄에 저장되어 있는 명령어 문자와 단어 데이터들(단어: word, 뜻: meaning)을 읽어, 단어와 뜻을 삽입, 탐색, 삭제, 출력을 하는 프로그램이다.

파일 데이터와 main 함수 동작

삽입(insert)의 경우, 명령어 문자 i와 함께 삽입할 단어와 뜻이 차례대로 저장되어 있다. 만약 이미 사전에 동일한 단어(word)가 저장되어 있는 경우, 연결 리스트를 이용하여 새로운 의미(meaning)을 추가하도록 한다. 탐색(search)의 경우, 명령어 문자 s와 함께 검색하고자 하는 단어(word)가 저장되어 있다. 탐색에 성공한 경우, 해당 단어와 의미를 출력하도록 한다. 삭제(delete)의 경우, 명령어 문자 d와 함께 삭제하고자 하는 단어(word)가 저장되어 있다. 이진 트리에서 탐색을 한 후에 해당 단어와 모든 의미들을 저장한 노드를 삭제하도록 한다. 종료(quit)를 의미하는 명령어 문자 q가 나타나면 사전 프로그램을 종료하도록 한다.

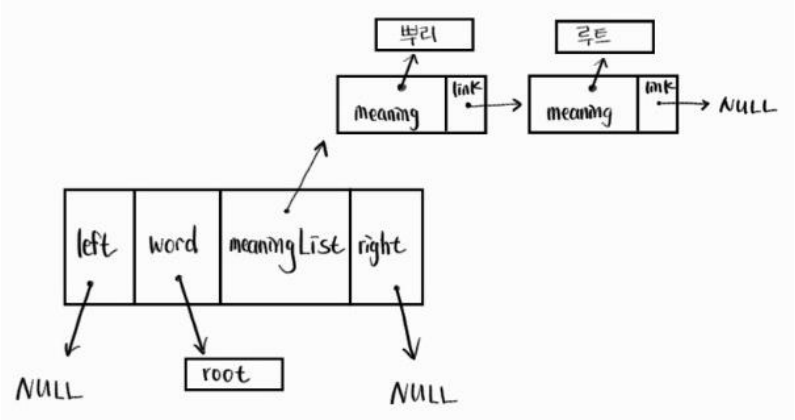
구조체(TreeNode, ListNode)

TreeNode: 하나의 단어를 구성하는 구조체

- word(트리 노드의 키 값): 단어를 저장하는 문자 배열을 가리키는 포인터
- meaningList: 여러 의미(meaning)들을 저장하는 연결 리스트 ListNode를 가리키는 포인터
- left, right: 왼쪽, 오른쪽 자식 노드를 가리키는 포인터

ListNode: 의미를 구성하는 구조체

- meaning(데이터 필드): 의미를 저장하는 문자 배열을 가리키는 포인터
- link(링크 필드): 다음 노드를 가리키는 포인터



한 단어(word)를 트리의 노드 TreeNode로 구현하면, 노드의 구조는 다음과 같다. 단어(word)는 트리 노드의 삽입 위치를 결정하는 키 값이라고 할 수 있다. 이는 문자형 포인터 word 멤버로 나타난다. 여러 뜻(meaning)들을 저장하는 연결 리스트 ListNode를 가리키는 리스트 구조체 포인터 meaningList, 그리고 각 자식 트리 노드를 가리키는 left와 right 포인터 멤버로 구성된다. meaningList가 가리키는 연결 리스트는 뜻(meaning) 문자열을 가리키는 문자형 포인터 meaning과 다음 리스트 노드를 가리키는 link로 구성된다.

단어 삽입

파일에서 읽은 단어(word)와 의미(meaning)을 하나의 노드로 생성하여 트리에 삽입한다. 단어를 트리에 삽입하고자 할 때는, 삽입의 위치를 탐색해야 한다. 삽입하고자 하는 단어(word)와 루트 노드부터 차례대로 각 트리 노드의 단어(word)의 사전 순 크기를 비교하여 왼쪽 혹은 서브 트리를 따라 내려간다. (사전 순 크기를 비교할 때에는 strcmp 함수를 사용하자.) 동일한 단어(word)가 나타나는 경우, 새로운 뜻(meaning)을 데이터 필드로 갖는 노드를 meaningList 리스트에 삽입하

도록 한다. (이 때, 중복되는 뜻은 추가되지 않는 것으로 가정한다.) 마지막 NULL값에 도달하게 되면, 해당 위치가 삽입하게 될 위치이므로 새로운 트리 노드, 그리고 meaningList 리스트 노드를 할당 받아 필요한 데이터들을 포함하여 새로운 노드의 삽입을 완료한다.

단어 탐색

파일에서 읽은 단어(word)를 트리에서 탐색을 한다. 루트 노드부터 차례대로, 단어(word)를 사전순으로 비교하며 탐색을 한다. 동일한 것이 나오는 경우 해당 노드의 주소를 반환한다. NULL값에 도달한다면, 해당 단어를 탐색하지 못한 것이므로 그대로 NULL을 반환하여 탐색에 실패했음을 알린다.

단어 출력

사전의 전체 단어는 사전 순으로 출력한다. 따라서 이진 트리를 중위 순회하며, 각 트리 노드의 단어, 그리고 단어에 대한 의미(들)을 출력하도록 한다. 단어는 트리 노드의 키 값이고, 하나 또는 여러 의미들은 리스트 노드의 데이터 필드에 저장되어 있다.

단어 삭제

파일에서 읽은 단어(word)를 트리에서 탐색하여, 탐색에 성공한 경우 해당 노드를 삭제한다. 총 3 가지의 경우가 존재하는데, 첫번째는 삭제하고자 하는 노드의 왼쪽 자식 노드가 없는 경우, 두번째는 오른쪽 자식 노드가 없는 경우, 그리고 세번째는 왼쪽과 오른쪽 자식 노드 모두 없는 경우이다. 첫번째 또는 두번째의 경우는 삭제하려는 노드의 부모 노드에 상대편의 자식 노드를 연결해주면 된다. 세번째의 경우에는, 삭제하는 노드의 키 값과 가장 비슷한 키 값을 탐색하여 데이터 필드를 바꾸어 주면 된다.

가장 비슷한 값은 값을 갖는 노드는 삭제하려는 노드의 오른쪽 서브 트리에서 가장 왼쪽에 있는 노드, 또는 왼쪽 서브 트리에서 가장 오른쪽에 있는 노드이다. 전자의 경우는 삭제하는 노드보다 큰 모든 값들 중 가장 작은 값을 가지고 있는 노드이고, 후자의 경우는 삭제하는 노드보다 작은 모든 값들 중 가장 큰 값을 가지고 있는 노드이다.

이번 사전 프로그램에서는 트리 노드의 데이터 필드가 구조체의 형식이 아닌, 문자 배열을 가리키는 포인터와 리스트를 가리키는 포인터로 구성되어 있으므로 단순히 가장 비슷한 값을 가진 노드의 데이터 필드를 그대로 대입해서는 안 된다. 조금 복잡한 과정을 요구하므로, 자세한 내용은 *1.3 소스코드 분석*] 의 *delete_node, min_value_node 함수 설명 부분* 에서 살펴보자.

1.2 소스 코드

```
/*
	작성자 : 이예빈(20204059)
	작성일 : 2021.10.03
	프로그램명 : 이진 탐색 트리를 이용한 영한 사전 프로그램

	 	i: 삽입(중복되는 의미 추가), d: 삭제, s: 탐색
	 	p: 전체 출력(단어에 대한 의미들 모두 출력), q:종료
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <memory.h>

typedef struct element {
	char word[200], meaning[200];
}element;

typedef struct TreeNode {
	char* word;
	struct ListNode* meaningList;
	struct TreeNode* left, * right;
}TreeNode;

typedef struct ListNode {
	char* meaning;
	struct ListNode* link;
}ListNode;

void error(char* message);  //오류 함수
int compare(char* e1, char* e2);  //사전 순 문자열 비교
void display_allMeanings(ListNode* head);  //모든 meaning 출력
void display(TreeNode* p);  //사전 전체 출력
TreeNode* search(TreeNode* root, char* word);  //단어 탐색
ListNode* insert_first(ListNode* head, char* meaning);  //리스트의 처음에 삽입(meaning)
TreeNode* new_treeNode(element new_data);  //새로운 트리 노드 생성(word, meaning)
TreeNode* insert_node(TreeNode* root, element temp);  //트리에 노드 삽입
void clear_treeNode(TreeNode* node);  //트리 노드 메모리 해제
TreeNode* delete_node(TreeNode* node, char* word);  //단어 노드 삭제
void clear(TreeNode* root);  //전체 트리 노드 메모리 해제

int main() {
	FILE* fp;
	TreeNode* root = NULL, * p = NULL;
	element temp;
	char word[200];
	char command;

	fp = fopen("data.txt", "rt");
	if (fp == NULL) error("파일 열기 실패");

	while (!feof(fp)) {
		fscanf(fp, " %c", &command);
		printf("%c \n", command);

		if (command == 'i') {  //단어 삽입
			fscanf(fp, "%s %s", temp.word, temp.meaning);
			printf("단어:%s\n의미:%s \n\n", temp.word, temp.meaning);
			root = insert_node(root, temp);
		}

```

```
63
64      else if (command == 's') { //단어 탐색
65          fscanf(fp, "%s", word);
66          p = search(root, word);
67          if (p == NULL)
68              printf("%s 탐색 실패 \n\n", word);
69          else {
70              printf("단어:%s\n의미:", p->word);
71              display_allMeanings(p->meaningList);
72              printf("\b\b  \n\n");
73          }
74      }
75
76      else if (command == 'd') { //단어 삭제
77          fscanf(fp, "%s", word);
78          printf("단어: %s \n\n", word);
79          root = delete_node(root, word);
80      }
81
82      else if (command == 'p') { //단어 전체 출력
83          display(root);
84          printf("\b\b \n\n");
85      }
86
87      else if (command == 'q')    //사전 프로그램 종료
88          break;
89      }
90
91      clear(root);    //트리의 모든 메모리 해제
92      fclose(fp);     //파일 닫기
93      return 0;
94  }
95
96  void error(char* message) {
97      fprintf(stderr, "%s \n", message);
98      exit(1);
99  }
100
101 int compare(char* e1, char* e2) {
102     return strcmp(e1, e2);
103 }
104
105 void display_allMeanings(ListNode* head) {
106     ListNode* p = head;
107     while (p != NULL) {
108         printf("%s, ", p->meaning);
109         p = p->link;
110     }
111 }
112 void display(TreeNode* p) {
113     if (p != NULL) {
114         display(p->left);
115         printf("%s ", p->word);//단어 word 출력
116         display_allMeanings(p->meaningList); //word의 모든 meaning 출력
117         display(p->right);
118     }
119 }
120
```

```
120  //·word의·위치·탐색하여·반환
121 TreeNode*·search(TreeNode*·root,·char*·word)·{
122  →  TreeNode*·p·=·root;
123  →  while·(p·!=·NULL)·{
124  →  →  if·(compare(word,·p->word)·==·0)
125  →  →  →  return·p;
126  →  →  else·if·(compare(word,·p->word)·<·0)
127  →  →  →  p·=·p->left;
128  →  →  else·if·(compare(word,·p->word)·>·0)
129  →  →  →  p·=·p->right;
130  →  }
131  →  return·p;·//·탐색에·실패한·경우·NULL·반환
132  }
133
134  //·새로운·meaning·추가(meaningList·리스트의·처음에·삽입)
135 ListNode*·insert_first(ListNode*·head,·char*·meaning)·{
136  →  ListNode*·new_listNode·=·(ListNode*)malloc(sizeof(ListNode));
137  →  if·(new_listNode·==·NULL)·error("동적·메모리·할당·실패·");
138  →  else·{
139  →  →  new_listNode->meaning·=·(char*)malloc(sizeof(char)·*·(strlen(meaning)·+·1));
140  →  →  strcpy(new_listNode->meaning,·meaning);
141  →  →  new_listNode->link·=·head;
142
143  →  →  head·=·new_listNode;
144  →  →  return·head;
145  →  }
146  }
147
148  //·새로운·트리·노드·생성·(새로운·단어·word·삽입)
149 TreeNode*·new_treeNode(TreeNode*·root,·element·new_data)·{
150  →  TreeNode*·temp·=·(TreeNode*)malloc(sizeof(TreeNode));
151  →  if·(temp·==·NULL)·error("동적·메모리·할당·실패·");
152  →  else·{
153  →  →  temp->meaningList·=·NULL;
154  →  →  temp->left·=·temp->right·=·NULL;
155
156  →  →  temp->word·=·(char*)malloc(sizeof(char)·*·(strlen(new_data.word)·+·1));
157  →  →  strcpy(temp->word,·new_data.word);
158
159  →  →  temp->meaningList·=·insert_first(temp->meaningList,·new_data.meaning);→ //·meaningList에·meaning·추가
160  →  →  return·temp;
161  →  }
162  }
163
164  //·위치·탐색하여·새로운·단어·삽입·(word가·이미·존재하는·경우,·meaning·추가)
165 TreeNode*·insert_node(TreeNode*·root,·element·temp)·{
166  →  if·(root·==·NULL)·return·new_treeNode(root,·temp);
167
168  →  if·(compare(temp.word,·root->word)·<·0)
169  →  →  root->left·=·insert_node(root->left,·temp);
170  →  else·if·(compare(temp.word,·root->word)·>·0)
171  →  →  root->right·=·insert_node(root->right,·temp);
172  →  else→  →  //·word가·동일한·경우에는·해당·treeNode의·meaningList에·새로운·meaning·추가
173  →  →  root->meaningList·=·insert_first(root->meaningList,·temp.meaning);
174
175  →  return·root;
176  }
177  //·meaningList의·노드·메모리·해제
178 void·clear_listNode(ListNode*·head)·{
179  →  ListNode*·q·=·head,·*·next·=·NULL;
180  →  while·(q·!=·NULL)·{
181  →  →  next·=·q->link;
182  →  →  free(q->meaning);
183  →  →  free(q);
184  →  →  q·=·next;
185  →  }
186  }
187
188  //·TreeNode·구조체의·모든·메모리·해제
189 void·clear_treeNode(TreeNode*·node)·{
190  →  clear_listNode(node->meaningList);
191  →  free(node->word);→ //·word·문자열·메모리·해제
192  →  free(node);→→  →  //·TreeNode·구조체·메모리·해제
193  }
194
```

```
195  //·왼쪽·자식·노드를·따라가며·가장·마지막·단말·노드를·반환
196  TreeNode*·min_value_node(TreeNode*·node)·{
197  →   TreeNode*·current·=·node,·*·prev·=·node;
198  →   while·(current->left·!=·NULL)·{
199  →   →   prev·=·current;→→   →   →   //·current의·부모·노드
200  →   →   current·=·current->left;
201  →   }
202  →   //·prev의·왼쪽·자식·노드·위치(기존·current·노드)에·current의·오른쪽·자식·노드·주소를·대입·(NULL값이·아닌·경우·고려)
203  →   prev->left·=·current->right;
204  →   return·current;
205  }
206
207  TreeNode*·delete_node(TreeNode*·node,·char*·word)·{
208  →   TreeNode*·temp·=·NULL;
209  →   if·(node·==·NULL)·return·node;
210
211  →   if·(compare(word,·node->word)·<·0)
212  →   →   node->left·=·delete_node(node->left,·word);
213  →   else·if·(compare(word,·node->word)·>·0)
214  →   →   node->right·=·delete_node(node->right,·word);
215  →   else·if·(compare(word,·node->word)·==·0)·{
216  →   →   if·(node->left·==·NULL)·{
217  →   →   →   temp·=·node->right;
218  →   →   →   clear_treeNode(node);
219  →   →   →   return·temp;
220  →   →   }
221  →   →   else·if·(node->right·==·NULL)·{
222  →   →   →   temp·=·node->left;
223  →   →   →   clear_treeNode(node);
224  →   →   →   return·temp;
225  →   →   }
227
228  →   →   temp·=·min_value_node(node->right);→→   //·temp·=·오른쪽·서브트리에서·가장·왼쪽·단말·노드
229
230  →   →   clear_listNode(node->meaningList);→//·meaningList·전체
231  →   →   free(node->word);→   →   →   →   →   //·word·삭제
232  →   →
233  →   →   node->word·=·temp->word;→   →   →   →   //·temp의·word를·삽입
234  →   →   node->meaningList·=·temp->meaningList;→//·temp의·meaningList를·삽입
235  →   →   free(temp);→→   →   →   →   →   →   →   →   //·temp·트리·노드·구조체의·메모리·해제(word,·meaningList제외)
236  →   }
237  →   return·node;
238  }
239
240  //·후위·순회를·이용하여·모든·노드·메모리·해제
241  void·clear(TreeNode*·root)·{
242  →   if·(root·!=·NULL)·{
243  →   →   clear(root->left);
244  →   →   clear(root->right);
245  →   →   clear_treeNode(root);
246  →   }
247  }
```

1.3 소스 코드 분석

```
/*
    작성자 : 이예빈(20204059)
    작성일 : 2021.10.03
    프로그램명 : 이진 탐색 트리를 이용한 영한 사전 프로그램

        i: 삽입(중복되는 의미 추가), d: 삭제, s: 탐색
        p: 전체 출력(단어에 대한 의미들 모두 출력), q:종료
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <memory.h>
```

1. 주석에 작성자, 작성일, 프로그램명을 작성하고 필요한 헤더 파일들을 추가한다. 동적 메모리 할당을 위한 표준 라이브러리, 메모리 헤더 파일을, 그리고 문자열을 다루기 위한 스트링 헤더 파일을 추가적으로 추가한다.

```
typedef struct element {
    char word[200], meaning[200];
}element;

typedef struct TreeNode {
    char* word;
    struct ListNode* meaningList;
    struct TreeNode* left, * right;
}TreeNode;

typedef struct ListNode {
    char* meaning;
    struct ListNode* link;
}ListNode;
```

2. 프로그램에서 사용되는 자료 구조들의 기본이 되는 구조체들을 정의한다.

- element: 단어(word)와 뜻(meaning) 문자 배열을 멤버로 갖는 구조체

- TreeNode(이진 탐색 트리의 노드): 키 값으로 단어(word) 문자형 포인터, 그리고 의미들이 저장되어 있는 ListNode 리스스트의 포인터인 meaningList, 그리고 왼쪽/오른쪽 자식 노드를 가리키는 left 와 right 포인터를 멤버로 갖는 구조체

- ListNode(연결 리스트의 노드): 의미(meaning) 문자형 포인터와 다음 노드를 가리킬 link 자기 참조 구조체 포인터를 멤버로 갖는 구조체

```
void error(char* message);                    //오류함수
int compare(char* e1, char* e2);              //사전순문자열비교
void display_allMeanings(ListNode* head);     //모든meaning출력
void display(TreeNode* p);                    //사전전체출력
TreeNode* search(TreeNode* root, char* word); //단어탐색
ListNode* insert_first(ListNode* head, char* meaning);  //리스트의처음에삽입(meaning)
TreeNode* new_treeNode(element new_data);     //새로운트리노드생성(word, meaning)
TreeNode* insert_node(TreeNode* root, element temp);    //트리에노드삽입
void clear_treeNode(TreeNode* node);          //트리노드메모리해제
TreeNode* min_value_node(TreeNode* node);     //노드에서가장작은키값가진노드반환
TreeNode* delete_node(TreeNode* node, char* word);      //단어노드삭제
void clear(TreeNode* root);                   //전체트리노드메모리해제
```

3. 프로그램에서 사용되는 함수들의 원형을 정의한다.

main 함수의 동작을 살펴보기 전 아래에서 각 함수들을 먼저 자세히 살펴보도록 하자.

```
void error(char* message) {
    fprintf(stderr, "%s\n", message);
    exit(1);
}
```

4. 파일 입출력 또는 동적 메모리 할당에서 오류가 발생한 경우 호출되는 error 함수이다. 매개변수로 받은 message 를 출력한 후, 전체 프로그램을 종료하도록 한다.

```
void display_allMeanings(ListNode* head) {
    ListNode* p = head;
    while (p != NULL) {
        printf("%s, ", p->meaning);
        p = p->link;
    }
}
```

5. ListNode 연결 리스트의 헤드 포인터를 매개 변수로 받아 모든 meaning 문자열을 출력하는 display_allMeanings 함수이다. ListNode 포인터 p 가 전체 리스트를 순회하며 meaning 문자열을 출력하도록 한다.

```
void display(TreeNode* p) {
    if (p != NULL) {
        display(p->left);
        printf("%s ", p->word);//단어 word 출력
        display_allMeanings(p->meaningList);//word의 모든 meaning 출력
        display(p->right);
    }
}
```

6. 이진 트리에서 순회를 시작하고자 하는 루트 노드 포인터를 매개 변수로 받아, 모든 노드를 중위 순회하며 모든 단어(word)와 각 단어의 모든 의미(meaning)들을 출력하는 display 함수이다. 데이터를 출력할 때에는, 먼저 단어(word)를 출력하고 이후 display_allMeaningList 함수를 출력하여 해당 단어에 대한 모든 의미도 함께 출력하도록 한다.

```
int compare(char* e1, char* e2) {
    return strcmp(e1, e2);
}
```

7. 두 문자열 포인터를 입력 받아 strcmp 함수를 통해 두 문자열의 사전 순 비교를 하여 그 결과 값을 반환하는 compare 함수이다. e1 이 가리키는 문자열이 e2 보다 사전 순으로 앞에 위치하는 경우 음수를, 뒤에 위치하는 경우에는 양수를, 두 문자열이 동일한 경우에는 0 을 반환한다.

```
//word의 위치 탐색하여 반환
TreeNode* search(TreeNode* root, char* word) {
    TreeNode* p = root;
    while (p != NULL) {
```

8. 트리 노드의 루트 노드 포인터와 탐색하고자 하는 word 문자형 포인터를 매개 변수로 전달받아 word 가 트리 내에서 탐색하여, 탐색에 성공한 경우 해당 노드의 주소를 반환하는 search 함수이다. TreeNode 형 포인터 p 가 이진 트리의 루트 노드 주소로 초기화된다.

```
        if (compare(word, p->word) == 0)
            return p;
        else if (compare(word, p->word) < 0)
            p = p->left;
        else if (compare(word, p->word) > 0)
            p = p->right;
    }
    return p;//탐색에 실패한 경우 NULL 반환
}
```

9. compare 함수를 호출하여 각 노드의 word 멤버와 탐색하고자 하는 word 를 사전 순 비교하여 다음 탐색 위치를 결정하도록 한다. 이진 트리의 특성을 이용하여, word 가 노드의 word 보다 사전 순으로 앞에 위치한다면 왼쪽 자식 노드로 이동하여 다시 탐색을 반복하고, 뒤에 위치한다면 오른쪽 자식 노드로, 그리고 동일한 것이 나온다면 해당 노드의 주소를 반환하도록 한다. 마지막 단말 노드 아래 NULL 값까지 도달하게 된다면, 탐색에 실패한 경우이다.

```
//·새로운·meaning·추가(meaningList·리스트의·처음에·삽입)
ListNode*·insert_first(ListNode*·head,·char*·meaning)·{
    ListNode*·new_listNode·=·(ListNode*)malloc(sizeof(ListNode));
    if·(new_listNode·==·NULL)·error("동적·메모리·할당·실패·");
```

10. 새로운 의미(meaning)을 담은 노드를 meaningList 연결 리스트의 처음에 삽입하는 insert_first 함수이다. ListNode 의 헤드 포인터와 meaning 문자형 포인터를 매개 변수로 전달받는다. new_listNode 포인터에 ListNode 구조체의 크기만큼의 메모리를 동적 할당 받는다.

```
    else·{
        new_listNode->meaning·=·(char*)malloc(sizeof(char)·*·(strlen(meaning)·+·1));
        strcpy(new_listNode->meaning,·meaning);
        new_listNode->link·=·head;

        head·=·new_listNode;
        return·head;
    }
}
```

11. 할당에 성공한 경우, meaning 문자 포인터형 멤버(new_listNode->meaning)에 매개 변수로 받은 meaning 의 문자열 길이에 1 을 더한 크기만큼 메모리를 동적 할당한다. (마지막 문자열의 끝의 널문자('₩0')의 공간까지 고려해야 하기 때문이다.) 이후 strcpy 함수를 이용하여 meaning 문자열을 복사하여 저장한다. 링크 필드에는 리스트의 head 주소를 저장하고, 다시 head 주소에는 new_listNode 를 대입하여 리스트의 처음에 삽입하는 것을 완성시킨다. 마지막으로 변경된 헤드 포인터를 반환한다.

```
//·새로운·트리·노드·생성·(새로운·단어·word·삽입)
TreeNode*·new_treeNode(element·new_data)·{
    TreeNode*·temp·=·(TreeNode*)malloc(sizeof(TreeNode));
    if·(temp·==·NULL)·error("동적·메모리·할당·실패·");
    else·{
        temp->meaningList·=·NULL;
        temp->left·=·temp->right·=·NULL;
```

12. 이진 트리의 새로운 단어(word) 노드를 동적 할당 받는 new_treeNode 함수이다. 단어(word)와 의미(meaning)을 멤버로 하는 element 구조체의 new_data 를 매개 변수로

전달받는다. TreeNode 형 temp 에 메모리를 동적 할당하고, 할당에 성공한 경우 우선적으로 menaingList 포인터, 그리고 left 와 right 포인터에 NULL 값을 대입한다.

```
    temp->word = (char*)malloc(sizeof(char) * (strlen(new_data.word) + 1));
    strcpy(temp->word, new_data.word);

    temp->meaningList = insert_first(temp->meaningList, new_data.meaning); // meaningList에 meaning 추가
    return temp;
  }
}
```

13. 이후, word 문자 포인터에는 new_data 의 word 문자 배열 길이보다 1 큰 값만큼의 메모리 공간을 할당하고, 문자열을 복사하도록 한다.

14. meaningList 포인터 멤버에는 아직은 NULL 값인 헤드 포인터 (temp->meaningList_)와 새로운 new_data 의 meaning 문자 배열을 인수로 전달하여 insert_first 함수를 호출하고 새로운 연결 리스트를 생성하도록 한다.

```
// 위치 탐색하여 새로운 단어 삽입 (word가 이미 존재하는 경우, meaning 추가)
TreeNode* insert_node(TreeNode* root, element temp) {
    if (root == NULL) return new_treeNode(root, temp);
```

15. 이진 트리에서 위치를 탐색한 후, 새로운 단어 노드를 삽입하는 insert_node 함수이다. 트리의 루트 노드 포인터와 새로운 단어를 담은 element 구조체 temp 를 매개 변수로 전달받는다. root 가 NULL 값인 경우(아직 이진 트리 생성 전이거나, 혹은 위치를 탐색 성공한 경우), new_treeNode 함수를 호출하여 새로운 노드를 해당 위치에 삽입하도록 한다.

```
    if (compare(temp.word, root->word) < 0)
        root->left = insert_node(root->left, temp);
    else if (compare(temp.word, root->word) > 0)
        root->right = insert_node(root->right, temp);
    else        // word가 동일한 경우에는 해당 treeNode의 meaningList에 새로운 meaning 추가
        root->meaningList = insert_first(root->meaningList, temp.meaning);

    return root;
}
```

16. compare 함수를 통해 새로 삽입하고자 하는 word 와 각 노드의 word 멤버를 사전 순 비교하여 비교한 값에 따라 왼쪽 또는 오른쪽 자식 노드로 이동한다. 만약 word 가 동일한 것이 나온 경우에는, 해당 노드의 meaningList 연결 리스트에 새로운 meaning 노드를 추가하도록 한다. (이 때, meaning 은 중복되는 일이 없는 것을 전제로 하자.)

```
//·meaningList의·노드·메모리·해제
void·clear_listNode(ListNode*·head)·{
→   ListNode*·q·=·head,·*·next·=·NULL;
→   while·(q·!=·NULL)·{
→   →   next·=·q->link;
→   →   free(q->meaning);
→   →   free(q);
→   →   q·=·next;
→   }
}
```

17. meaningList 연결 리스트의 모든 노드 메모리를 해제하는 clear_listNode 함수이다. 리스트의 헤드 포인터를 매개 변수로 전달받는다. 현재 노드를 가리키는 q, 그리고 다음 노드를 가리키는 next 포인터가 리스트를 순회하며 meaning 을 담은 노드들의 메모리를 해제한다. 노드의 메모리를 해제하기 전, meaning 포인터 멤버에 할당된 메모리도 해제한다.

```
//·TreeNode·구조체의·모든·메모리·해제
void·clear_treeNode(TreeNode*·node)·{
→   clear_listNode(node->meaningList);
→   free(node->word);→  //·word·문자열·메모리·해제
→   free(node);→→   →   //·TreeNode·구조체·메모리·해제
}
```

18. 트리의 노드 한 개의 메모리를 해제하는 clear_treeNode 함수이다. 우선 노드를 해제하기 전, clear_listNode 를 호출하여 meaningList 포인터 멤버가 가리키는 연결 리스트를 모두 해제하고, word 멤버에 할당된 메모리 또한 해제한다. 마지막으로 node 구조체의 메모리를 해제하도록 한다.

아래는 트리 노드의 루트 노드 포인터와 삭제하고자 하는 단어(word)를 매개변수로 전달받아 트리에서 word 를 탐색 후, 해당 노드의 메모리를 해제하는 delete_node 함수에 대한 긴 설명이다.

```
TreeNode*·delete_node(TreeNode*·node,·char*·word)·{
→   TreeNode*·temp·=·NULL;
→   if·(node·==·NULL)·return·node;
```

19. 임시로 다른 노드의 주소를 저장할 temp 포인터를 선언한다. 만약 루트 노드가 null 값이라면 그대로 null 을 반환한다.

```
→   if·(compare(word,·node->word)·<·0)
→   →   node->left·=·delete_node(node->left,·word);
→   else·if·(compare(word,·node->word)·>·0)
→   →   node->right·=·delete_node(node->right,·word);
```

20. 삽입, 탐색과 마찬가지로 compare 사전 순으로 word 와 노드의 word 를 비교하며 동일한 문자열을 멤버로 하는 노드를 탐색한다. 사전 순 비교를 통해 왼쪽 또는 오른쪽 자식 노드로 이동한다.

```
else if (compare(word, node->word) == 0) {
    if (node->left == NULL) {
        temp = node->right;
        clear_treeNode(node);
        return temp;
    }
    else if (node->right == NULL) {
        temp = node->left;
        clear_treeNode(node);
        return temp;
    }
```

21. 위는 동일한 word 문자열을 가진 노드가 발견되는 경우이다. 만약, 해당 노드의 왼쪽 자식 노드가 없는 경우라면, 오른쪽 자식 노드의 주소(또는 NULL 값)를 temp 에 저장해 놓은 후, clear_treeNode 함수를 호출하여 현재 노드의 메모리는 해제한다. 이후, temp 를 반환하여 삭제한 노드의 부모 노드와 연결해주면 된다. 오른쪽 자식이 NULL 인 경우도 동일한 방법으로 해결한다.

22. 아래는 왼쪽, 오른쪽 자식 노드 모두 존재하는 경우이다. 이 경우에는, 삭제하고자 하는 현재 노드의 키 값(word)과 가장 비슷한 값을 찾아 해당 노드의 데이터들을 삭제하는 현재 노드의 데이터 필드로 옮겨야 한다. 가장 비슷한 값은 오른쪽 서브 트리에서 가장 왼쪽 단말 노드(삭제하는 노드의 키 값보다는 크지만, 오른쪽 서브 트리의 모든 노드 중에서는 가장 큰 키 값을 가지는 노드), 또는 왼쪽 서브 트리에서 가장 오른쪽 단말 노드의 키 값이다. 현재 프로그램에서는 오른쪽 서브 트리에서 가장 왼쪽 단말 노드의 주소를 반환하는 min_value_node 함수를 사용하자.

```
        temp = min_value_node(node->right);    // temp = 오른쪽 서브트리에서 가장 왼쪽 단말 노드

        clear_listNode(node->meaningList); // meaningList 전체
        free(node->word);                  // word 삭제

        node->word = temp->word;           // temp의 word를 삽입
        node->meaningList = temp->meaningList; // temp의 meaningList를 삽입
        free(temp);                        // temp 트리 노드 구조체의 메모리 해제 (word, meaningList 제외)
    }
    return node;
}
```

23. temp 에 노드의 오른쪽 자식 노드를 인수로 전달하여 min_value_node 에서 얻은 노드의 주소를 대입한다. 이후 현재 삭제하는 노드의 meaningList 리스트와 word 멤버들을 모두 메모리

해제한다. 그리고 다시 현재의 word 포인터에와 meaningList 포인터에는 temp 의 word, meaningList 포인터 값을 대입하도록 한다.

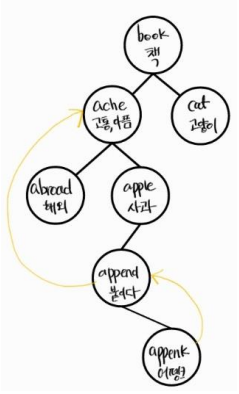
24. 마지막으로 temp 트리 노드 구조체에 할당된 메모리를 해제한다. 이 때, word 와 meaningList 포인터에 할당된 값은 node 의 데이터 필드에 각각 저장되어 있으므로, 이 메모리들을 해제가 되지 않도록 한다.

25. 아래는 매개 변수로 전달받은 node 의 왼쪽 자식 노드를 따라가며 가장 왼쪽에 있는 노드를 반환하는 min_value_node 함수이다. 위의 delete_node 함수에서 삭제하고자 하는 노드의 왼쪽, 자식 노드가 모두 존재하는 경우, 오른쪽 서브 트리를 인수로 전달하여 삭제하고자 하는 노드와 가장 비슷한 노드의 주소를 반환 값으로 받기 위해 호출되는 함수이다.

```
//·왼쪽·자식·노드를·따라가며·가장·마지막·단말·노드를·반환
TreeNode*·min_value_node(TreeNode*·node)·{
→   TreeNode*·current·=·node,·*·prev·=·node;
→   while·(current->left·!=·NULL)·{
→   →   prev·=·current;→→   →   →   //·current의·부모·노드
→   →   current·=·current->left;
→   }
→   //·prev의·왼쪽·자식·노드·위치(기존·current·노드)에·current의·오른쪽·자식·노드·주소를·대입.·(NULL값이·아닌·경우·고려)
→   prev->left·=·current->right;
→   return·current;
}
```

26. TreeNode 포인터인 current 와 prev 를 선언하고 node 의 주소를 대입한다. current 는 노드에서 가장 큰 키 값을 가지고 있는, 가장 왼쪽에 존재하는 노드이고 prev 는 current 의 부모 노드이다. current 의 왼쪽 자식 노드가 NULL 값일 때까지 반복하여 순회하며, 왼쪽 자식 노드를 따라 마지막 노드까지 도달하게 된다.

27. 이후 부모 노드인 prev 에 current 의 오른쪽 자식 노드를 연결한다. (current 노드는 delete_node 함수에서 temp 값에 저장되고, 결국 메모리가 해제되는 노드이다. 이 때, current 의 왼쪽 자식 노드는 항상 NULL 이지만, 오른쪽 자식 노드에는 또다른 노드가 존재할 수 있으므로 오른쪽 서브 트리를 prev 와 연결해주도록 한다.)



(위 그림이 그 예시이다. 루트 노드의 왼쪽 자식 노드인 'ache'를 삭제하는 경우, 오른쪽 서브 트리에서 가장 작은 값인 'append'의 데이터 값들을 'ahce'의 데이터 필드로 연결한 후 해당 노드를 삭제하게 되는 경우, 'append'의 오른쪽 자식 노드였던 'appenk'를 'append'의 부모 노드인 'appenk'의 왼쪽 자식 노드로 연결해주어야 한다.)

마지막으로, main 함수의 동작을 살펴보자.

```
int main() {
    FILE* fp;
    TreeNode* root = NULL, * p = NULL;
    element temp;
    char word[200];
    char command;

    fp = fopen("data.txt", "rt");
    if (fp == NULL) error("파일 열기 실패");
```

28. 파일 포인터 fp 를 선언하고, 이진 트리의 루트 노드인 root 포인터와 search(탐색)할 때 탐색에 성공한 노드의 값을 저장할 포인터 p 를 선언한다. 파일로부터 입력 받는 단어(word)와 의미(meaning)들을 임시로 저장할 element 형 temp 구조체 변수, 그리고 문자 배열 word 임시 변수, 그리고 파일에서 입력 받는 문자를 저장할 command 변수를 선언한다.

29. "data.txt" 파일을 읽기 모드로 연다.

```
    while (!feof(fp)) {
        fscanf(fp, "%c", &command);
        printf("%c \n", command);

        if (command == 'i') {        //단어 삽입
            fscanf(fp, "%s %s", temp.word, temp.meaning);
            printf("단어:%s\n의미:%s \n\n", temp.word, temp.meaning);
            root = insert_node(root, temp);
        }
```

30. 파일을 열어 가장 첫번째에 저장되어 있는 명령어 문자를 command 로 읽는다.

31. 만약 command 가 i 라면, 각각 temp 구조체의 word 와 meaning 멤버로 문자열을 읽는다. 이후, insert_node 에 트리의 루트 노드 포인터 root 와 temp 를 전달하여 호출하고, 변경된 루트 포인터를 다시 root 에 저장한다.

```
    else if (command == 's') { // 단어 탐색
        fscanf(fp, "%s", word);
        p = search(root, word);
        if (p == NULL)
            printf("%s 탐색 실패 \n\n", word);
        else {
            printf("단어:%s\n의미:", p->word);
            display_allMeanings(p->meaningList);
            printf("\b\b  \n\n");
        }
```

32. 명령어 문자가 s 인 경우, 단어를 탐색한다. 먼저 단어 word 를 읽고, search 함수를 호출하여 탐색한 노드의 주소를 p 에 저장한다. 만약 p 가 NULL 값이라면 탐색에 실패했다는 것이고, 그렇지 않다면 해당 노드의 단어와 의미(들)을 출력하도록 한다. 먼저 노드의 word 멤버를 출력하고, display_allMeanings 를 meaningList 포인터를 인수로 전달하여 호출한 뒤, 의미를 모두 출력한다. 백스페이스 키와 공백을 이용하여 마지막에 출력되는 '->' 문자들이 출력되지 않도록 한다.

```
    else if (command == 'd') { // 단어 삭제
        fscanf(fp, "%s", word);
        printf("단어: %s \n\n", word);
        root = delete_node(root, word);
    }

    else if (command == 'p') { // 단어 전체 출력
        display(root);
        printf("\b\b \n\n");
    }

    else if (command == 'q')    // 사전 프로그램 종료
        break;
}
```

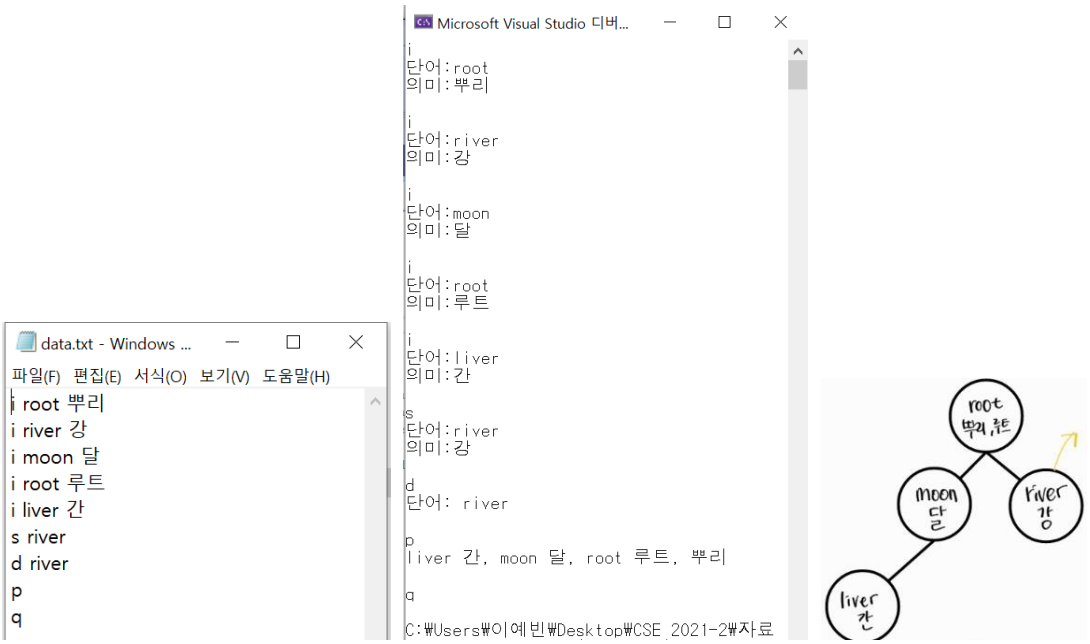
33. 명령어 문자가 d 인 경우, word 를 읽어 delete_node 함수를 호출하여 이진 트리에서 해당 word 를 삭제한다. p 인 경우 display 를 호출하여 이진 트리에 저장된 모든 단어와 의미들을 출력한다. Q 인 경우는 사전 프로그램을 종료한다.

```
clear(root);    // 트리의 모든 메모리 해제
fclose(fp);     // 파일 닫기
return 0;
```

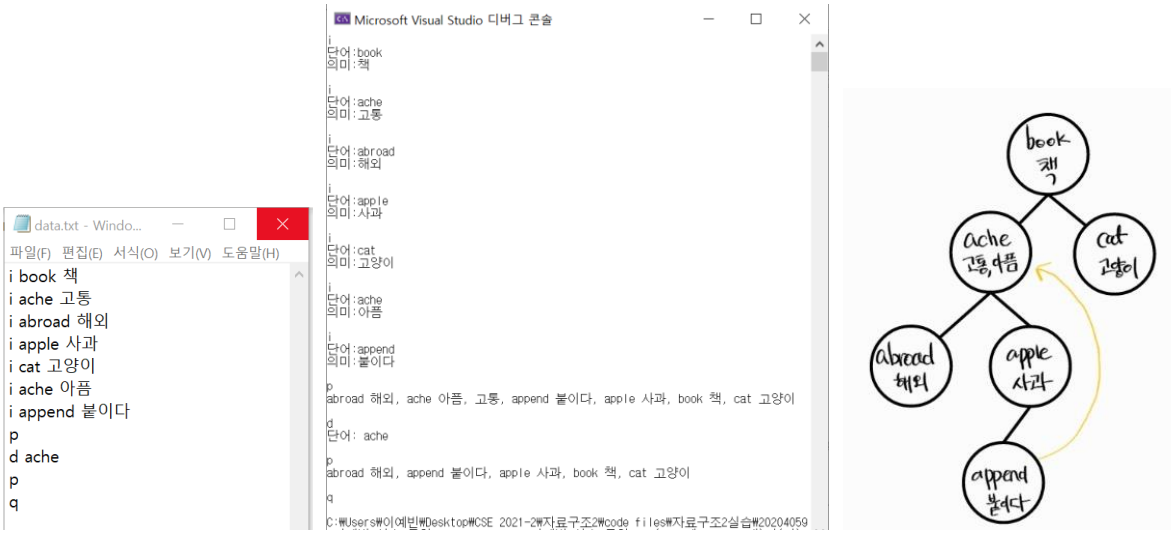
34. 마지막으로 clear 함수를 호출하여 트리의 모든 노드에 대한 메모리를 해제하고, 파일을 닫고 프로그램 전체를 종료하도록 한다.

1.4 실행창

결과 1 (문제의 예시 실행창과 동일)



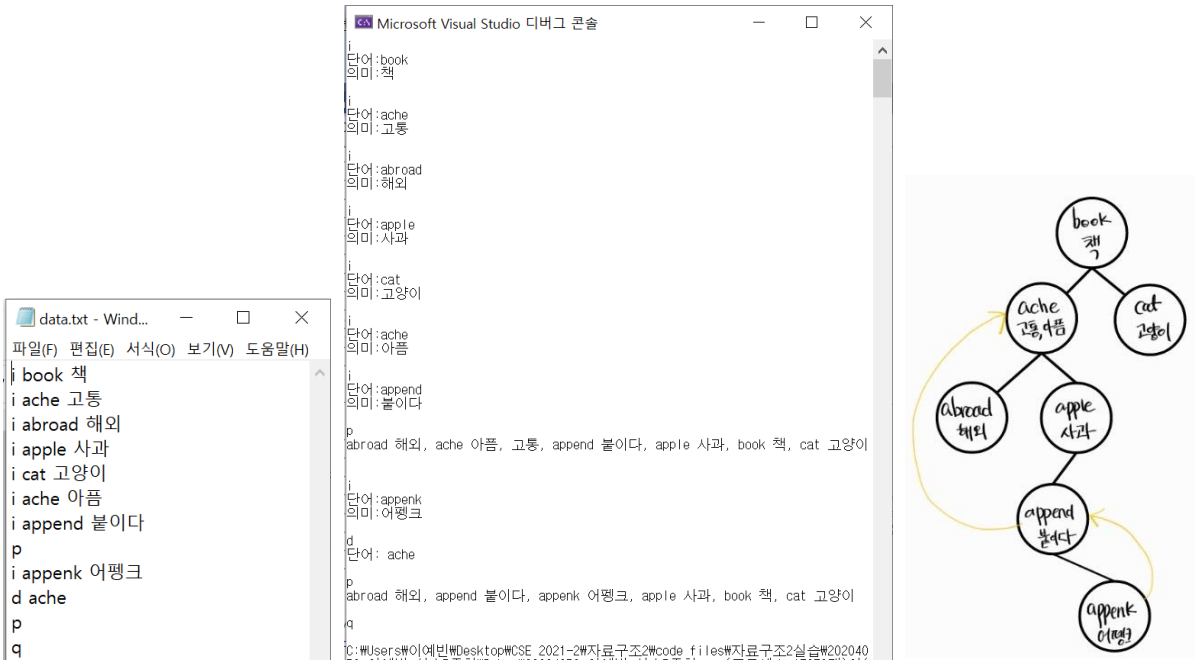
결과 2



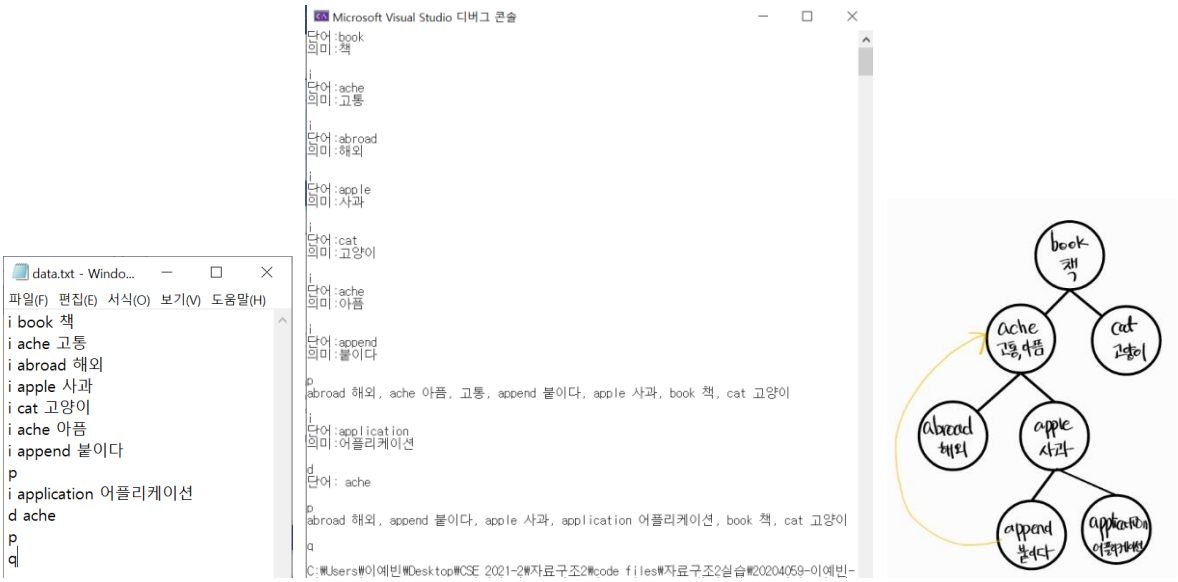
결과 3

(삭제하는 노드의 오른쪽 서브 트리에서

가장 왼쪽 단말 노드에 오른쪽 자식 노드가 존재하는 경우)



결과 4



1.5 느낀점

이번 사전 프로그램 과제를 하면서 그동안 놓쳤던 부분들을 다시 해결할 수 있게 되었고, 또 트리 노드를 구성하는 구조체의 형식이 기존과는 조금 다르기 때문에 교재에 나와 있는 기존의 코드들을 변경하는 과정에서 이진 탐색 트리에 대해 이해도를 이전보다 훨씬 더 높일 수 있는 기회였다.

우선 이전에 놓쳤던 부분은, 문자열에 대한 메모리를 해제하지 않았다는 점이다. 이전 실습 과제 프로그램을 짜는 과정에서 트리 노드의 메모리를 해제하는 부분에 문자형 포인터에 동적 할당했던 문자 배열에 대한 메모리를 해제하는 코드를 작성하지 않았다는 것을 뒤늦게 깨달었다. 그 실수를 깨닫고는, 이번에는 철저하게 포인터에 할당된 모든 메모리를 제대로 해제하고자 하였다.

다음으로는 여러 시도를 해보는 과정에서 트리에 대한 이해도를 높였다. 처음에 트리 노드의 구조체의 형식을 어떻게 구성해야 할지에 대한 고민이 많았다. 고민 끝에, 이진 탐색 트리에서 각 트리 노드의 위치를 지정하는 키 값이 존재해야 한다는 점과 단어에 대한 의미가 여러 개가 있을 수 있다는 점에서, 트리 노드 구조체의 데이터 필드에는 단어 문자 배열을 가리키는 포인터, 그리고 하나의 의미를 하나의 리스트 노드로 구성하여 의미 리스트들을 가리키는 포인터로 구성하는 것이 가장 효율적인 구조라고 생각하게 되었다. 특히 데이터 필드가 각각 서로 다른 자료라는 점에서, 노드를 삭제하는 코드가 교재의 것에서 조금은 다르게 변경하게 되었다. 이 과정이 쉽지만은 않았지만, 이전의 코드를 자세하게 분석하는 데 많은 시간을 투자함으로써 이진 탐색 트리의 여러가지 연산들에 대한 이해도가 많이 높아진 것 같다.