

자료구조2 실습

14주차 과제



제출일	21.12.07	전 공	컴퓨터소프트웨어공학과
과 목	자료구조 2 실습	학 번	20184612
담당교수	홍 민 교수님	이 름	김동민

| 목 차 |

1. 탐색 : 보간 탐색 구현

- 1.1 문제 분석
- 1.2 소스 코드
- 1.3 소스 코드 분석
- 1.4 실행 창
- 1.5 느낀 점

2. AVL트리 구현

- 2.1 문제 분석
- 2.2 소스 코드
- 2.3 소스 코드 분석
- 2.4 실행 창
- 2.5 느낀 점

3. 선형 조사법을 사용한 HashTable

- 3.1 문제 분석
- 3.2 소스 코드
- 3.3 소스 코드 분석
- 3.4 실행 창
- 3.5 느낀 점

4. 느낀 점

1. 탐색

보간 탐색 구현

- data.txt에서 데이터를 읽어와 509페이지의 프로그램 13,7의 보간 탐색 프로그램을 아래와 같이 실행되도록 구현하시오

1.1 문제 분석

조건 1 동적할당을 이용하여 파일에 있는 데이터를 저장

조건 2 보간 탐색 알고리즘을 사용하여 데이터를 찾음

조건 3 데이터의 개수와 찾고자 하는 정수를 직접 입력받음

- 이번 문제는 보간 탐색을 사용하여 원하는 값을 찾고, 알고리즘의 진행 시간을 확인해보는 문제이다. 보간 탐색은 사전이나 전화번호부를 탐색하는 방법과 같이 탐색키가 존재할 위치를 예측하여 탐색한다.

이번 데이터 파일에는 10000000개의 데이터가 존재하기 때문에 이를 순차 탐색으로 값을 찾게 된다면 매우 많은 시간이 걸릴 것이다. 하지만 보간 탐색을 이용하여 값을 찾는다면 시간을 매우 빠르게 줄일 수 있다.

보간 탐색을 구현하기 위해서는 탐색 위치를 찾는 공식을 사용한다. 탐색 위치를 찾는 공식은 $((k - \text{list}[\text{low}]) / (\text{list}[\text{high}] - \text{list}[\text{low}]) * (\text{high} - \text{low})) + \text{low}$ 이다. 여기서 low와 high는 각각 탐색할 범위의 최소, 최대 인덱스 값을 나타낸다. 이 공식을 사용하게 된다면 찾고자 하는 키 값이 있는 곳에 근접하게 갈 수 있는 것이다.

보간 탐색은 데이터가 비교적 균등하게 분포되어 있을 경우 이진 탐색보다 우수한 방법이 될 수 있고 시간복잡도는 $O(\log n)$ 을 가진다.

그리고 이번 문제에서는 보간 탐색이 진행되는 동안 시간을 측정해야 하기 때문에 time.h헤더를 포함하여 탐색이 진행되는 시간을 측정하고, 이 값을 출력하여 확인해보도록 한다.

1.2 소스 코드

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int interpol_search(int list[], int key, int n) { //보간 탐색 함수
    int low, high, j;

    low = 0; //low는 가장 첫 인덱스
    high = n - 1; //high는 가장 끝 인덱스
    while ((list[high] >= key) && (key > list[low])) {
        j = ((float)(key - list[low]) / (list[high] - list[low])) * (high - low) + low; //탐색 위치를 구하는 공식
        if (key > list[j]) high = j + 1;
        else if (key < list[j]) high = j - 1; //key보다 list[j]가 클때 high는 j-1
        else low = j; //key와 list[j]가 같을 때
    }
    if (list[low] == key) return low; //탐색 성공
    else return -1; //탐색 실패
}

int main() {
    FILE *fp;
    int count, find, i = 0; //count는 데이터의 개수, find는 찾을 정수
    int n; //찾을 정수의 위치를 저장
    int *list; //데이터 파일을 저장할 배열
    clock_t start, stop;
    double duration;

    fp = fopen("data.txt", "r"); //data.txt 파일 오픈
    if (!fp) {
        printf("file not open");
        return 0;
    }
    printf("데이터의 개수 : ");
    scanf("%d", &count); //데이터의 개수 입력
    printf("찾고자 하는 정수를 입력하세요 : ");
    scanf("%d", &find); //찾고싶은 정수 입력

    list = (int*)malloc(sizeof(int)*count); //count로 list포인터 동적할당
    for(i=0; i<count; i++){
        fscanf(fp, "%d", &list[i]); //파일 데이터를 list에 삽입
    }

    start = clock(); //시작 시간 저장
    n = interpol_search(list, find, count); //보간 탐색 함수 호출
    stop = clock(); //끝 시간 저장
    duration = (double)(stop - start) / CLOCKS_PER_SEC; //실행 시간 계산
    if (n == -1) {
        printf("찾을 정수가 없음\n");
    }
    else {
        printf("%d 번째에 저장되어 있음\n", n + 1); //위치 출력
        printf("보간 탐색 실행 속도 : %lf", duration);
    }

    free(list); //리스트 동적할당 해제
    fclose(fp); //파일포인터 닫음
    return 0;
}
```

1.3 소스 코드 분석

```
/*
    학번 : 20184612
    학과 : 컴퓨터소프트웨어공학과
    이름 : 김동민
    파일 명: 보간 정렬 프로그램
*/
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

1. 소스코드를 작성한 날짜, 이름, 프로그램명을 작성하고, 필요한 헤더를 포함한다.

2. 이번 코드에서는 보간 탐색의 실행시간을 확인해야 하기 때문에 이와 관련된 헤더인 time.h헤더를 포함시킨다.

```
int interpol_search(int list[], int key, int n) {           //보간 탐색 함수
    int low, high, j;

    low = 0;           //low는 가장 첫 인덱스
    high = n - 1;      //high는 가장 끝 인덱스
    while ((list[high] >= key) && (key > list[low])) {
        j = ((float)(key - list[low]) / (list[high] - list[low])) * (high - low) + low;
        if (key > list[j]) low = j + 1;
        else if (key < list[j]) high = j - 1; //key보다 list[j]가 클때 high는 j-1
        else low = j; //key와 list[j]가 같을 때
    }
    if (list[low] == key) return low; //탐색성공
    else return -1; //탐색 실패
}
```

3. 보간 탐색 함수를 구현한다.

low는 가장 첫 인덱스의 번호를 나타내고 high는 가장 끝 인덱스의 번호를 나타낸다.

4. while문을 이용하여 list[high]값이 키보다 작을 때까지 그리고 key가 list[low]보다 작거나 같아질 때까지 반복한다. 먼저 j에 보간탐색에서 탐색 위치를 찾는 공식을 이용하여 저장한다.

공식은 $((key - list[low]) / (list[high] - list[low]) * (high - low)) + low$ 이다.

위의 식은 탐색 위치를 결정할 때 찾고자 하는 키 값이 있는 곳에 근접하게 되도록 가중치를 준다.

5. 만약 key가 list[j]보다 크다면 low에 j+1을 삽입하고 key가 더 작다면 high에서 j-1을 삽입한다. 그리고 반복문을 계속 돌며 값을 찾게 된다. 만약 list[j]와 key가 같게 된다면 low에 j값을 삽입하고 반복을 종료하게 된다.

6. 마지막으로 탐색이 성공했다면 low인덱스 값을 리턴하고 탐색이 실패했다면 -1을 리턴하게 된다.

```
int main() {
    FILE *fp;
    int count, find, i = 0; //count는 데이터의 개수, find는 찾을 정수
    int n;                  //찾을 정수의 위치를 저장
    int *list;              //데이터 파일을 저장할 배열
    clock_t start, stop;
    double duration;

    fp = fopen("data.txt", "r"); // "data.txt"파일 오픈
    if (!fp) {
        printf("file not open");
        return 0;
    }
    printf("데이터의 개수 : ");
    scanf("%d", &count); //데이터의 개수 입력
    printf("찾고자 하는 정수를 입력하세요: ");
    scanf("%d", &find); //찾고싶은 정수 입력

    list = (int*)malloc(sizeof(int)*count); //count로 list포인터 동적할당
    for(i=0; i<count; i++){
        fscanf(fp, "%d", &list[i]); //파일 데이터를 list에 삽입
    }
    start = clock(); //시작 시간 저장
    n = interpol_search(list, find, count); //보간 탐색 함수 호출
    stop = clock(); //끝 시간 저장
    duration = (double)(stop - start) / CLOCKS_PER_SEC; //실행 시간 계산
    if (n == -1) {
        printf("찾을 정수가 없음\n");
    }
    else {
        printf("%d 번째에 저장되어 있음\n", n + 1); //위치 출력
        printf("보간 탐색 실행 속도 : %f", duration);
    }
    free(list); //리스트 동적할당 해제
    fclose(fp); //파일포인터 닫음
    return 0;
}
```

7. 필요한 변수들을 선언한다.

- fp는 파일포인터, count는 데이터의개수, find는 찾을 정수를 입력받는다.
- n은 보간 탐색 함수에서 반환되는 값을 저장하고 list는 데이터파일을 저장한다. 그리고 시간을 측정할 때 사용하는 start와 stop변수를 선언한다.

8. fp를 fopen함수를 이용하여 data.txt파일을 오픈한다. 만약 파일이 존재하지 않는다면 file not open을 출력하고 프로그램을 종료한다.

9. 사용자로부터 데이터의 개수와 찾을 정수를 입력받는다.
데이터의 개수는 count에 저장하고 찾고자 하는 정수는 find변수에 입력받는다.

10. 입력받은 count개를 이용하여 list포인터를 동적할당하여 선언한다.
그리고 for문을 이용하여 count만큼 반복하며 파일로부터 입력받은 데이터를 list배열에 저장한다.

11. 보간탐색의 시간을 파악하기 위해 start변수에 현재 시간을 저장한다.
그리고 interpol_search함수를 호출하여 보간 탐색을 수행하고, 그 위치를 n값에 반환받는다. 탐색이 끝나면 stop변수에 끝나는 시간을 입력받는다.

12. 실행시간을 계산하기 위해 duration을 계산한다. stop에서 start를 빼고 그 값을 CLOCKS_PER_SEC로 나누면 실행시간을 계산할 수 있다.
그리고 만약 n값이 -1이면 데이터 파일에 없는 값이므로 이를 출력하고 n값이 존재한다면 정수값의 위치를 출력한다.
그리고 실행시간을 나타내는 duration을 출력한다.

13. 마지막으로 free를 이용하여 동적할당한 list배열을 삭제하고 fclose로 파일포인터를 닫은 후 프로그램을 종료한다.

1.4 실행 창

6666666을 찾을 때

데이터의 개수 : 10000000
찾고자 하는 정수를 입력하세요: 6666666
6666667 번째에 저장되어 있음
보간 탐색 실행 속도 : 0.000000
C:\Users\wdmk46\OneDrive\바탕 화면\2학기 수업자료\자료구조2\실습\week14_1_interpol_search.exe(20828 프로세스)이(가) 0 코드로 인해 종료되었습니다.
이 창을 닫으려면 아무 키나 누르세요.

data - Windows 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
3	284	285	286	287	288	289	290	291	292	293	294	295								
9	540	541	542	543	544	545	546	547	548	549	550	551								
5	796	797	798	799	800	801	802	803	804	805	806	807								
0	41	1042	1043	1044	1045	1046	1047	1048	1049	1050										

-1을 찾을 때(데이터 파일에 존재하지 않음)

데이터의 개수 : 10000000
찾고자 하는 정수를 입력하세요: -1
찾을 정수가 없음
C:\Users\wdmk46\OneDrive\바탕 화면\2학기 수업자료\자료구조2\실습\week14_1_interpol_search.exe(1784 프로세스)이(가) 0 코드로 인해 종료되었습니다.
이 창을 닫으려면 아무 키나 누르세요.

1.5 느낀 점

이번 문제는 보간 탐색 알고리즘을 사용하여 원하는 값의 위치를 찾는 문제였습니다. 보간 탐색이라는 알고리즘을 처음 배웠기 때문에 처음에는 어렵게 느껴졌었는데, 막상 프로그래밍을 하고나니 생각보다 쉽게 느껴졌던 것 같습니다.

이번 문제에서 가장 인상깊었다고 생각되는 부분은 보간 탐색 알고리즘을 수행할 때 위치를 찾는 공식이었습니다. 그냥 막 써놓은 것 같아도 직접 이 공식을 사용해보니 거의 그 자리에 값이 존재했다는 것이 신기하게 느껴졌습니다.

이번 기회를 계기로 보간탐색에 대해 자세히 할 수 있었고 앞으로 보간 탐색을 사용할 때가 생긴다면 이번에 했던 것을 이용하여 탐색을 할 수 있도록 노력하겠습니다.

2. 탐색

AVL트리 구현

- 520 페이지에 있는 프로그램 13,11의 AVL트리 프로그램을 구현하시오. 이때 데이터는 data.txt에서 읽어오며 a는 add(삽입) 그리고 s는 search(탐색)이다.

2.1 문제 분석

조건 1 AVL트리 알고리즘을 사용하여 data파일 저장

조건 2 삽입 또는 탐색할 값을 data로부터 입력받음

조건 3 탐색을 할 때 방문하는 노드들을 차례로 출력

- 이 문제는 AVL트리를 이용하여 파일에 있는 데이터를 삽입하고, 탐색하는 문제이다. AVL트리는 왼쪽 서브트리의 높이와 오른쪽 서브트리의 높이 차이가 1이하인 이진 탐색 트리로 트리가 비균형상태가 되면 스스로 노드를 재배치하여 균형으로 만든다. 따라서 AVL트리는 탐색이 $O(\log n)$ 시간안에 끝난다.

이번 문제의 경우 파일에는 삽입과 탐색 연산만 존재한다. 그러므로 이진 탐색트리에서 균형 상태가 깨지는 경우는 삽입만 존재한다. 따라서 삽입 후 노드의 균형인수를 계산하고 그에 맞게 노드들을 회전시키면 될 것이다.

노드의 균형을 맞추기 위해서는 노드를 오른쪽 또는 왼쪽으로 회전시킨다. 만약 노드가 LL타입으로 왼쪽 자식의 왼쪽에 삽입되어있다면 오른쪽회전, RR타입이면 왼쪽 회전이다. 그리고 LR타입이면 왼쪽회전 후 오른쪽 회전을 하고 RL타입이면 오른쪽 회전 후 왼쪽 회전을 수행한다.

2.2 소스 코드

```
#define _CRT_SECURE_NO_WARNINGS

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

typedef struct AVLNode { //AVL트리 정의
    int key;
    struct AVLNode *left;
    struct AVLNode *right;
}AVLNode;

AVLNode *rotate_right(AVLNode *parent); //오른쪽으로 회전시키는 함수
AVLNode *rotate_left(AVLNode *parent); //왼쪽으로 회전시키는 함수

int get_height(AVLNode *node); //트리의 높이를 반환하는 함수
int get_balance(AVLNode *node); //노드의 균형 인수를 반환
AVLNode * create_node(int key); //새로운 노드를 생성하는 함수

AVLNode* insert(AVLNode*node, int key); //AVL트리에 새로운 노드를 추가하는 함수
AVLNode* search(AVLNode*node, int key); //AVL트리에서 노드를 찾는 함수

void Delete(AVLNode*node) { //AVL트리를 삭제하는 함수
    if (node != NULL) {
        Delete(node->left); //왼쪽 노드 내려감
        Delete(node->right); //오른쪽 노드 내려감
        // printf("%d ", node->key);
        free(node); //현재 노드 삭제
    }
}

void preorder(AVLNode*node) { //전위 순회 함수
    if (node != NULL) {
        printf("%d ", node->key); //현재 노드 출력
        preorder(node->left); //왼쪽 노드 순회
        preorder(node->right); //오른쪽 노드 순회
    }
}

int main() {
    FILE *fp;
    AVLNode *root = NULL;
    char c;
    int n;

    fp = fopen("data.txt", "r");
    if (!fp) {
        printf("file not open");
        return 0;
    }
    while (!feof(fp)) {
        fscanf(fp, "%c", &c);
        if (c == 'a') { //란의 입력받은 문자가 a하면 삽입
            fscanf(fp, "%d", &n);
            root = insert(root, n); //insert 삽입 함수 호출
        }
        else if (c == 's') { //입력받은 문자가 s하면 탐색
            AVLNode*tmp;
            fscanf(fp, "%d", &n);
            printf("탐색 : %d\n", n);
            tmp = search(root, n); //search탐색 함수 호출
            if (tmp != NULL) { //tmp가 NULL이 아니면 탐색결과
                printf(" %d\n", tmp->key);
            }
            else { //tmp가 NULL이면 탐색 불가
                printf("탐색 불가\n");
            }
        }
    }
    // preorder(root);

    Delete(root); //AVL트리 삭제
    fclose(fp); //파일포인터 닫음
    return 0;
}
```

```

AVLNode* rotate_right(AVLNode* parent) { // 오른쪽으로 회전시키는 함수
    AVLNode* child = parent->left;
    parent->left = child->right;
    child->right = parent;
    return child;
}

AVLNode* rotate_left(AVLNode* parent) { // 왼쪽으로 회전시키는 함수
    AVLNode* child = parent->right;
    parent->right = child->left;
    child->left = parent;
    return child;
}

int get_height(AVLNode* node) { // 트리의 높이를 구하는 함수
    int left_count, right_count;
    if (node == NULL) return 0; // 만약 node가 NULL이면 0 리턴

    left_count = get_height(node->left); // 왼쪽 트리의 높이 반환
    right_count = get_height(node->right); // 오른쪽 트리의 높이 반환

    if (left_count > right_count) // 만약 왼쪽이 더 높이가 높다면
        return left_count + 1; // 왼쪽 높이+1 반환
    else
        return right_count + 1; // 오른쪽 높이가 더 높다면 오른쪽 +1 반환
}

int get_balance(AVLNode* node) { // 트리의 균형 인수를 반환하는 함수
    if (node == NULL) return 0;
    return get_height(node->left) - get_height(node->right); // 노드의 왼쪽 높이에서 오른쪽 높이를 뺀 값을 반환
    // 왼쪽이 더 높다면 양수, 오른쪽이 더 높다면 음수
}

AVLNode* create_node(int key) { // 새로운 노드를 생성하는 함수
    printf("삽입 : %d\n", key);
    AVLNode* node = (AVLNode*)malloc(sizeof(AVLNode)); // 노드 할당할 공간
    node->key = key;
    node->left = NULL; // left와 right의 초기값을 NULL로 설정
    node->right = NULL;
    return node;
}

AVLNode* insert(AVLNode* node, int key) { //avl 트리에 새로운 노드를 삽입하는 함수
    int balance;
    if (node == NULL) // 만약 노드가 NULL이면
        return create_node(key); // 새로운 노드 리턴

    if (key < node->key) { // 만약 key가 node의 key보다 작다면
        node->left = insert(node->left, key); // 왼쪽으로 내려감
    }
    else if (key > node->key) { // 만약 key가 node의 key보다 크다면
        node->right = insert(node->right, key); // 오른쪽으로 내려감
    }
    else return node; // 중복된 키는 허용x

    balance = get_balance(node); // 노드의 균형인수 계산

    if (balance > 1 && key < node->left->key) { // 새로운 노드가 왼쪽 자식의 왼쪽에 추가 LL
        return rotate_right(node);
    }
    if (balance < -1 && key > node->right->key) { // 새로운 노드가 오른쪽 자식의 오른쪽에 추가 RR
        return rotate_left(node);
    }
    if (balance > 1 && key > node->left->key) { // 새로운 노드가 왼쪽 자식의 오른쪽에 추가 LR
        node->left = rotate_left(node->left);
        return rotate_right(node);
    }
    if (balance < -1 && key < node->right->key) { // 새로운 노드가 오른쪽 자식의 왼쪽에 추가 RL
        node->right = rotate_right(node->right);
        return rotate_left(node);
    }
    return node;
}

AVLNode* search(AVLNode* node, int key) { //avl 트리의 탐색 함수
    printf("탐색 결과 : ");
    while (node != NULL) { // 반복을 이용하여 탐색
        if (key == node->key) return node; // 만약 key와 노드의 key가 같다면 노드 리턴
        else if (key < node->key) { // key가 노드의 key보다 작다면 왼쪽으로 내려감
            printf("%d", node->key);
            node = node->left;
        }
        else { // key가 노드의 key보다 크다면
            printf("%d", node->key);
            node = node->right; // 오른쪽으로 내려감
        }
    }
    return NULL;
}

```

2.3 소스 코드 분석

```
/*
    학번 : 20184612
    학과 : 컴퓨터소프트웨어공학과
    이름 : 김동민
    파일명 : AVL트리 프로그램
*/
#define _CRT_SECURE_NO_WARNINGS
```

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

```
typedef struct AVLNode {    //AVL트리 정의
    int key;
    struct AVLNode *left;
    struct AVLNode *right;
}AVLNode;
```

1. 소스코드를 작성한 날짜, 이름, 프로그램명을 작성하고, 필요한 헤더를 포함한다.

2. AVL트리의 노드를 정의한다. 노드 안에는 데이터를 나타내는 key와 왼쪽 노드를 가리키는 left, 오른쪽 노드를 가리키는 right가 있다.

```
AVLNode *rotate_right(AVLNode *parent); //오른쪽으로 회전시키는 함수
AVLNode *rotate_left(AVLNode *parent);  //왼쪽으로 회전시키는 함수
```

```
int get_height(AVLNode*node);    //트리의 높이를 반환하는 함수
int get_balance(AVLNode *node);  //노드의 균형인수를 반환
AVLNode * create_node(int key);   //새로운 노드를 생성하는 함수
```

```
AVLNode* insert(AVLNode*node, int key); //AVL트리에 새로운 노드를 추가하는 함수
AVLNode* search(AVLNode*node, int key); //AVL트리에서 노드를 찾는 함수
```

3. 필요한 함수들을 선언한다.

- rotate_right함수는 AVL트리를 오른쪽으로 회전시키는 함수이다.
- rotate_left함수는 AVL트리를 왼쪽으로 회전시키는 함수이다.
- get_height함수는 트리의 높이를 반환하는 함수이다.
- get_balance함수는 노드의 균형인수를 반환하는 함수이다.
- create_node함수는 새로운 노드를 생성하여 반환하는 함수이다.
- insert함수는 AVL트리에 새로운 노드를 삽입하는 함수이다.
- seartch함수는 AVL트리에서 탐색을 수행하는 함수이다.

```

void Delete(AVLNode*node) {           //AVL트리를 삭제하는 함수
    if (node != NULL) {
        Delete(node->left);           //왼쪽 노드로 내려감
        Delete(node->right);          //오른쪽 노드로 내려감
        // printf("%d ", node->key);
        free(node);                   //현재 노드 삭제
    }
}

void preorder(AVLNode*node) {         //전위 순회 함수
    if (node != NULL) {
        printf("%d ", node->key);     //현재 노드 출력
        preorder(node->left);         //왼쪽 노드로 순환
        preorder(node->right);        //오른쪽 노드 순환
    }
}

```

3. Delete함수는 AVL트리를 삭제하는 함수이다.

후위 알고리즘을 사용하여 가장 아래에 있는 노드부터 위로 올라가며 노드를 하나씩 삭제한다.

4. preorder함수는 AVL트리를 전위 순회하며 출력하는 함수이다.

현재 노드를 출력한 후, 왼쪽 노드를 먼저 방문하고 그 다음 오른쪽 노드를 방문한다.

```

int main() {
    FILE *fp;
    AVLNode *root = NULL;
    char c;
    int n;

    fp = fopen("data.txt", "r");
    if (!fp) {
        printf("file not open");
        return 0;
    }
    while (!feof(fp)) {
        fscanf(fp, "%c", &c);
        if (c == 'a') { //만약 입력받은 문자가 a라면 삽입
            fscanf(fp, "%d", &n);
            root = insert(root, n); //insert 삽입함수 호출
        }
        else if (c == 's') { //입력받은 문자가 s라면 탐색
            AVLNode*tmp;
            fscanf(fp, "%d", &n);
            printf("탐색 : %d\n", n);
            tmp = search(root, n); //search탐색함수 호출
            if (tmp != NULL) { //tmp가 NULL이 아니면 탐색된것
                printf(" %d\n", tmp->key);
            }
            else { //tmp가 NULL이면 탐색 불가
                printf("탐색 불가\n");
            }
        }
    }
    // preorder(root);

    Delete(root); //AVL트리 삭제
    fclose(fp); //파일포인터 닫음
    return 0;
}

```

5. 필요한 변수들을 선언한다.

- fp는 파일포인터, root는 AVL트리의 루트가 된다.
- c와 n은 파일로부터 입력을 받을 때 데이터를 임시로 입력받는 변수이다.

6. fp를 fopen함수를 이용하여 data.txt파일을 오픈한다. 만약 파일이 존재하지 않는다면 file not open을 출력하고 프로그램을 종료한다.

7. feof를 이용하여 파일 끝까지 데이터를 입력받는다. 먼저 문자형 변수 c에 데이터를 입력받고, 만약 c가 'a'이면 삽입함수를 호출하고 's'이면 탐색함수를 호출한다.

8. 만약 c가 'a'라면 먼저 n에 파일 데이터를 읽어오고 insert함수를 호출하여 AVL트리에 이 값을 삽입한다.

9. 만약 c가 's'라면 n에 파일 데이터를 읽어오고 search함수를 호출하여 입력받은 n값을 AVL트리에서 찾는다. 만약 트리에 값이 존재하지 않는다면 NULL이 반환되므로 tmp가 NULL이라면 탐색 불가 메시지를 출력한다. 반대로 tmp가 NULL이 아니라면 tmp의 key값을 출력한다.

```

AVLNode *rotate_right(AVLNode *parent) {    //오른쪽으로 회전시키는 함수
    AVLNode*child = parent->left;
    parent->left = child->right;
    child->right = parent;
    return child;
}

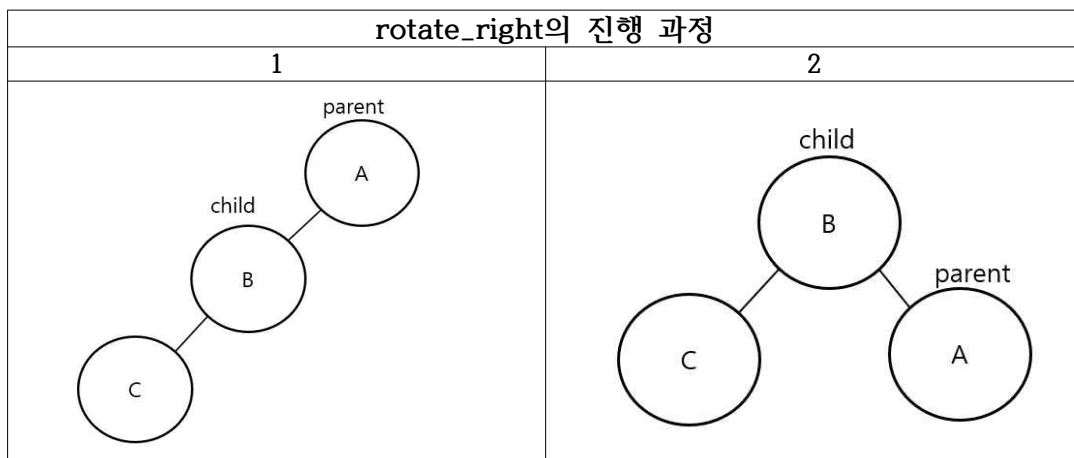
AVLNode *rotate_left(AVLNode *parent) {    //왼쪽으로 회전시키는 함수
    AVLNode*child = parent->right;
    parent->right = child->left;
    child->left = parent;
    return child;
}

```

10. rotate_right함수는 AVL트리를 오른쪽으로 회전시키는 함수이다. child노드의 초기값을 전달받은 parent의 left값으로 선언한다. 그리고 parent의 left값에 child의 right값을 삽입하고 child의 right값이 parent가 된다.

그리고 child를 반환한다.

11. rotate_left함수는 AVL트리를 왼쪽으로 회전시키는 함수이다. child노드의 초기값을 parent의 right값으로 선언하고, parent의 right에 child의 left값을 삽입하고 child의 left값에 parent를 삽입한다.




```

int get_height(AVLNode*node) {           //트리의 높이를 구하는 함수
    int left_count, right_count;
    if (node == NULL) return 0;          //만약 node가 NULL이면 0 리턴

    left_count = get_height(node->left);  //왼쪽 트리의 높이 반환
    right_count = get_height(node->right); //오른쪽 트리의 높이 반환

    if (left_count > right_count)         //만약 왼쪽이 더 높이가 높다면
        return left_count + 1;           //왼쪽 높이+1 반환
    else
        return right_count + 1;          //오른쪽 높이가 더 높다면 오른쪽 +1반환
}

int get_balance(AVLNode *node) {         //노드의 균형 인수를 반환하는 함수
    if (node == NULL) return 0;
    return get_height(node->left) - get_height(node->right); //노드의 왼쪽 높이에서 오른쪽 높이를 뺀 값을 반환
                                                                //왼쪽이 더 높다면 양수, 오른쪽이 더 높다면 음수
}

```

12. get_height함수는 트리의 높이를 구하는 함수이다.

만약 node가 NULL이면 0을 반환한다.

왼쪽 트리의 높이와 오른쪽 트리의 높이를 각각 left_count, right_count 변수에 전달받고 만약 왼쪽 노드의 높이가 더 크다면 left_count에 +1하여 리턴하고 반대로 오른쪽 노드의 높이가 더 크다면 right_count에 +1하여 반환한다.

13. get_balance함수는 노드의 균형 인수를 반환하는 함수이다.

만약 node가 NULL이면 0을 반환하고 아니라면 get_height함수로 구한 트리의 왼쪽 높이에서 오른쪽 높이를 뺀 값을 반환한다.

```

AVLNode * create_node(int key) {         //새로운 노드를 생성하는 함수
    printf("삽입 : %d\n", key);
    AVLNode* node = (AVLNode*)malloc(sizeof(AVLNode)); //노드 동적할당 생성
    node->key = key;
    node->left = NULL;                     //left와 right의 초기값을 NULL로 설정
    node->right = NULL;
    return node;
}

```

14. create_node함수는 새로운 노드를 생성하는 함수이다.

새로운 노드가 생성되면 현재 삽입되는 함수를 출력하고 AVLNode타입으로 새로운 노드를 생성한다.

15. node의 key값에 입력받은 key를 삽입하고 left와 right의 초기값을 NULL로 설정한 후 새로 생성한 노드를 반환한다.

```

AVLNode* insert(AVLNode*node, int key) { //avl트리에 새로운 노드를 삽입하는 함수
    int balance;
    if (node == NULL) //만약 노드가 NULL이면
        return create_node(key); //새로운 노드 리턴

    if (key < node->key) { //만약 key가 node의 key보다 작다면
        node->left = insert(node->left, key); //왼쪽으로 내려가는 순환
    }
    else if (key > node->key) { //만약 key가 node의 key보다 크다면
        node->right = insert(node->right, key); //오른쪽으로 내려감
    }
    else return node; //동일한 키는 허용x

    balance = get_balance(node); //노드들의 균형인수 계산

    if (balance > 1 && key < node->left->key) { //새로운 노드가 왼쪽 자식의 왼쪽에 추가 LL
        return rotate_right(node);
    }
    if (balance < -1 && key > node->right->key) { //새로운 노드가 오른쪽 자식의 오른쪽에 추가 RR
        return rotate_left(node);
    }
    if (balance > 1 && key > node->left->key) { //새로운 노드가 왼쪽 자식의 오른쪽에 추가 LR
        node->left = rotate_left(node->left);
        return rotate_right(node);
    }
    if (balance < -1 && key < node->right->key) { //새로운 노드가 오른쪽 자식의 왼쪽에 추가 RL
        node->right = rotate_right(node->right);
        return rotate_left(node);
    }
    return node;
}

```

16. insert함수는 AVL트리에 새로운 노드를 삽입하는 함수이다.

만약 노드가 NULL이라면 존재하지 않은 노드이므로 create_node함수를 통해 새로운 노드를 생성하여 반환한다.

17. 만약 key가 node의 key값보다 작다면 순환 호출을 이용해 트리의 왼쪽으로 내려가고, 반대로 key값이 node의 key값보다 크다면 트리의 오른쪽으로 내려간다.

만약 같은 key가 트리에 존재한다면 허용하지 않으므로 새로 삽입하지 않는다.

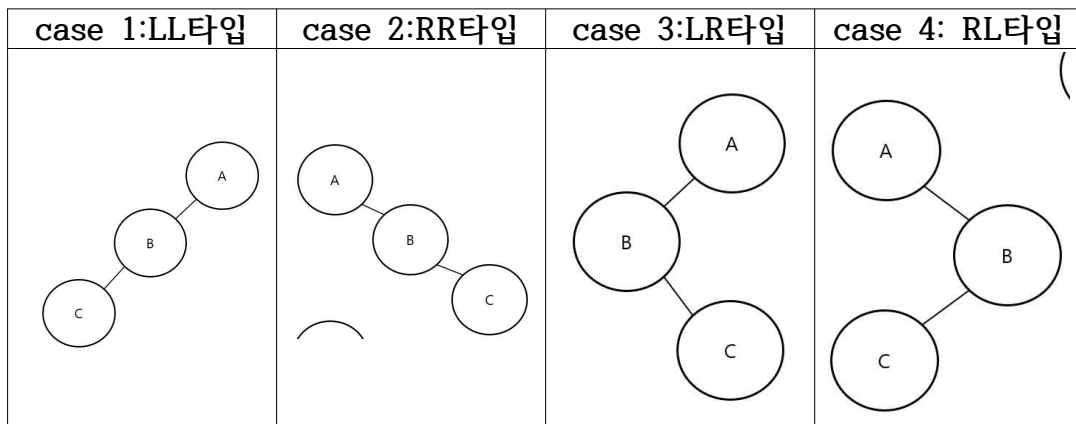
18. balance변수에 get_balance함수를 통해 노드의 균형인수를 반환받는다. 만약 balance가 2이면 왼쪽 높이가 2가 더 높은 것이고 -2이면 오른쪽 높이가 더 높은 것이다.

19. 만약 balance가 1보다 크고 key가 node의 왼쪽의 key보다 작다면 왼쪽으로 차례로 있는 트리어기 때문에 오른쪽으로 회전한다.

20. 만약 balance가 -1보다 작고 key가 node의 오른쪽의 key보다 크다면 오른쪽으로 경사진 트리이기 때문에 왼쪽으로 회전하는 함수인 rotate_left를 호출한다.

21. 만약 balance가 1보다 크고 key가 node의 왼쪽의 key보다 크다면 왼쪽 자식의 오른쪽에 추가가 된 것이므로 LR 타입으로 이중 회전해야 하기 때문에 rotate_left를 호출한 후 rotate_right함수를 차례로 호출한다.

22. 만약 balance가 -1보다 작고 key가 node의 오른쪽보다 작다면 오른쪽 자식의 왼쪽에 추가가 된 것이기 때문에 RL타입으로 이중회전한다. 그렇기 때문에 rotate_right함수를 호출한 후 rotate_left함수를 호출한다.



```
AVLNode* search(AVLNode*node, int key) { //avl트리 탐색 함수
    printf("탐색 결과 : ");
    while (node != NULL) { //반복을 이용하여 탐색
        if (key == node->key) return node; //만약 key와 노드의 key가 같다면 노드 리턴
        else if (key < node->key) { //key가 노드의 key보다 작다면 왼쪽으로 내려감
            printf(" %d", node->key);
            node = node->left;
        }
        else { //key가 노드의 key보다 크다면
            printf(" %d", node->key);
            node = node->right; //오른쪽으로 내려감
        }
    }
    return NULL;
}
```

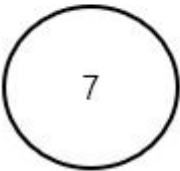
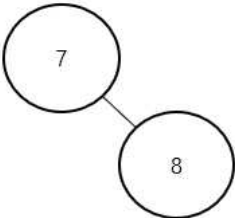
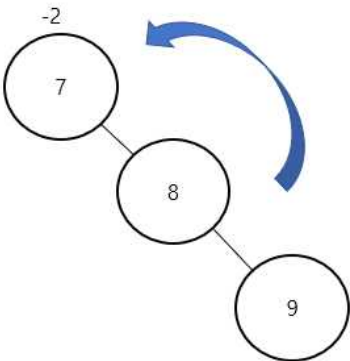
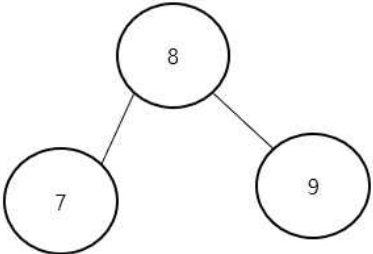
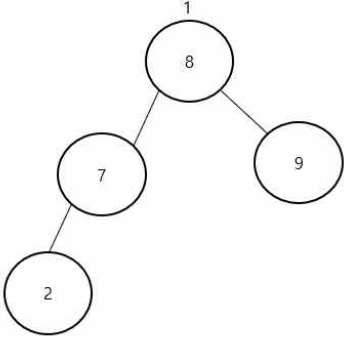
23. search함수는 AVL트리에서 key값을 찾는 함수이다.

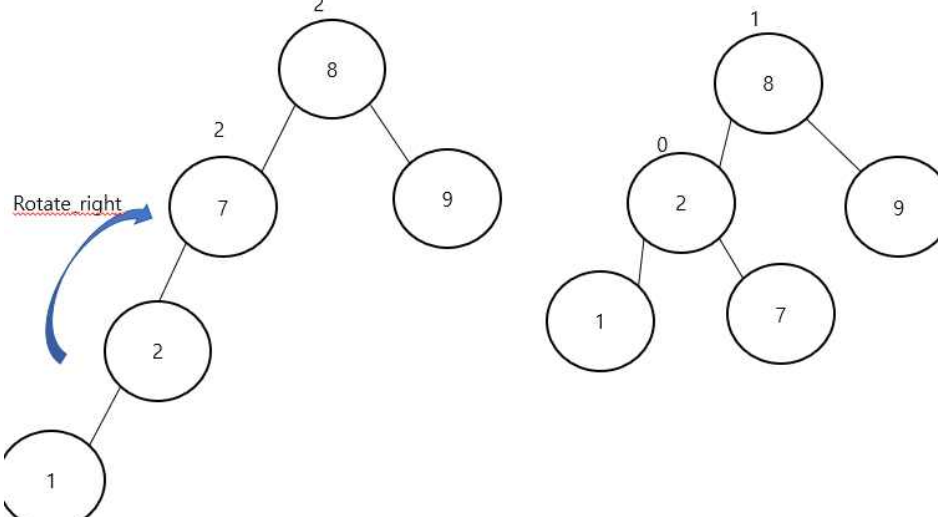
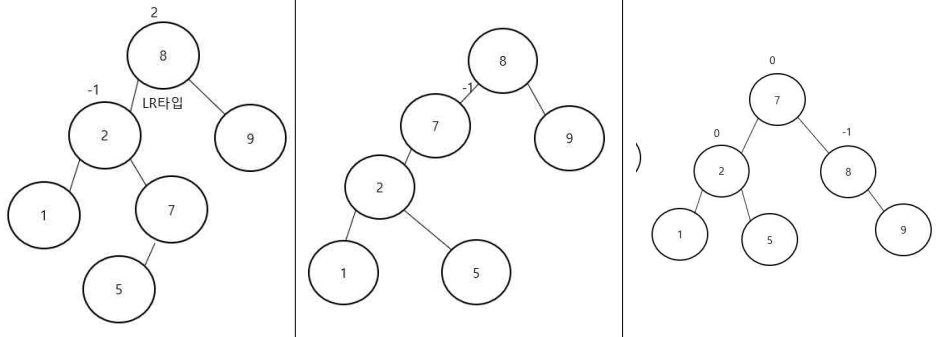
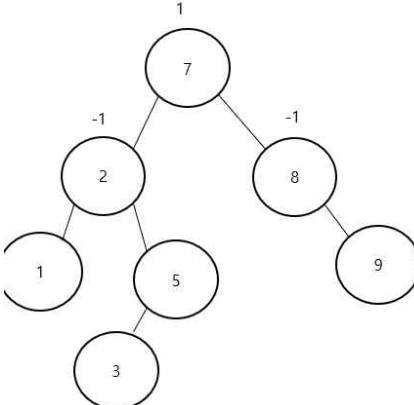
while문을 사용하여 반복을 이용해 탐색한다. 만약 key와 node의 key값이 같다면 현재 node를 반환한다.

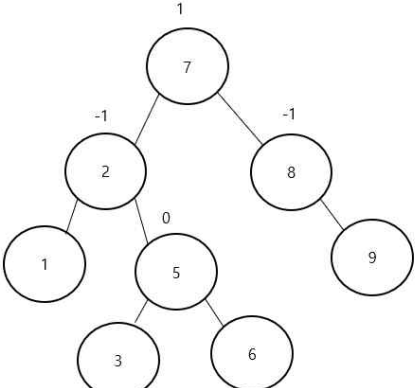
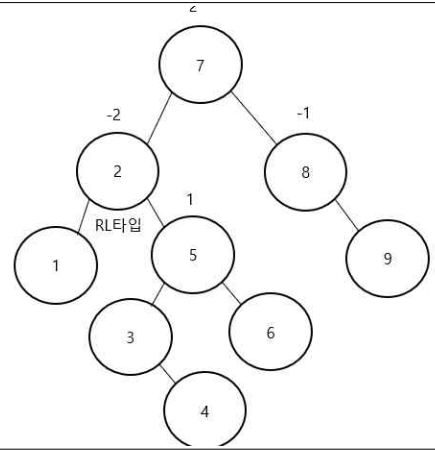
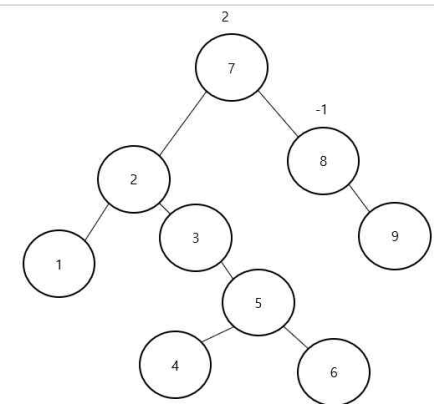
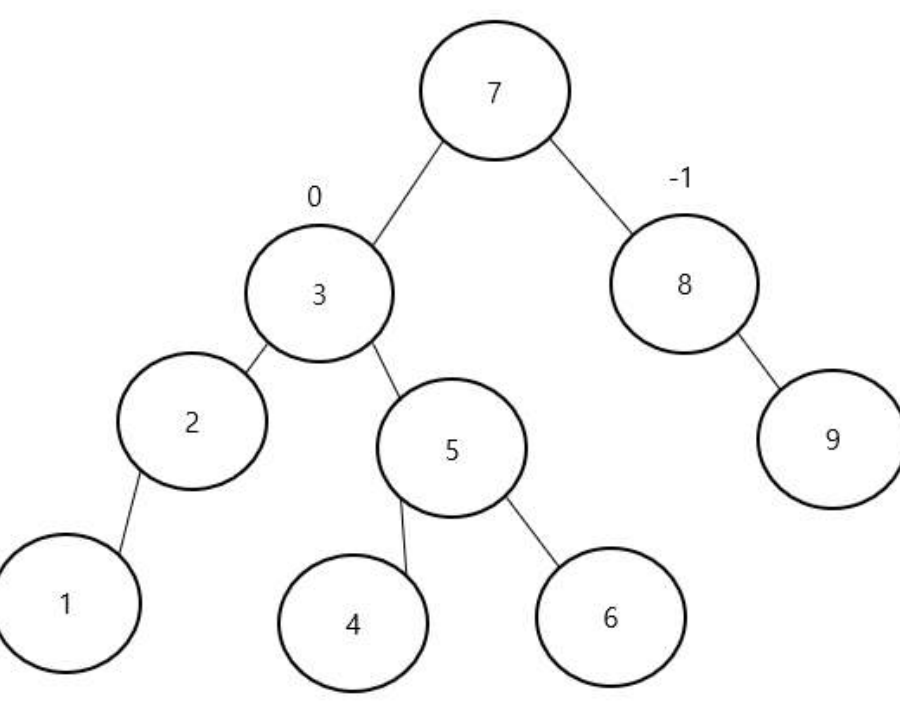
key가 node의 key값보다 작다면 node를 왼쪽으로 내려가고 key값이 더 크다면 node의 오른쪽으로 내려간다.

만약 트리에 값이 존재하지 않는다면 NULL값을 반환한다.

AVL트리 삽입시 변화 과정

7 삽입	 <pre>graph TD; 7((7))</pre>
8 삽입	 <pre>graph TD; 7((7)) --- 8((8))</pre>
9 삽입	<div><pre>graph TD; 7((7)) --- 8((8)) --- 9((9))</pre></div> <div>RRE타입 ->rotate left</div> <div><pre>graph TD; 8((8)) --- 7((7)) --- 9((9))</pre></div>
2 삽입	 <pre>graph TD; 8((8)) --- 7((7)) --- 9((9)) --- 2((2))</pre>

<p>1 삽입</p>	
<p>5 삽입</p>	
<p>3 삽입</p>	

<p>6삽입</p>	
<p>4 삽입</p>	<div data-bbox="427 703 863 1151"></div> <div data-bbox="863 703 1361 1151"></div>
<p>최종 트리</p>	

2.4 실행 창

```
삽입 : 7
삽입 : 8
삽입 : 9
삽입 : 2
삽입 : 1
삽입 : 5
삽입 : 3
삽입 : 6
삽입 : 4
탐색 : 4
탐색 결과 : 7 3 5 4

C:\Users\dmk46\OneDrive\로세스)이(가) 0 코드로 영
이 창을 닫으려면 아무 키
```

data - Windows

파일(F) 편집(E) *

a 7

a 8

a 9

a 2

a 1

a 5

a 3

a 6

a 4

s 4

2.5 느낀 점

이번 문제는 AVL트리를 이용하여 삽입과 탐색을 하는 문제였습니다. 트리에 대해서는 옛날에 배운 적이 있었지만 AVL트리는 기존 트리 알고리즘과 살짝 달라 처음에는 어렵게 느꼈던 것 같습니다. 특히 삽입을 수행할 때 균형인수를 기준으로 왼쪽 또는 오른쪽으로 트리를 회전 시키는 것이 잘 이해가 안되기도 했습니다. 하지만 이번 과제를 계기로 삽입의 과정을 직접 손으로 그려보고 과정을 확인해보며 AVL트리에 대해 이해할 수 있었던 것 같습니다.

아직 복잡한 트리의 LR이나 RL타입에 대해 이해가 잘 안되는 부분이 있었기 때문에 이번 기회를 계기로 여러 데이터들을 이용하여 AVL트리의 변화 과정을 확인해보고 공부하는 데에 도움이 될 수 있도록 할 것입니다.

3. 해싱

선형 조사법을 이용한 HashTable

- data.txt에 저장된 정수들을 선형 조사법을 이용한 HashTable에 저장하라, 저장된 정수들의 앞의 문자에 따라 저장하거나 HashTable에 있는지 검색하여 있으면 HashTable에 저장된 위치를 출력하고 검색이 되지 않는다면 아래와 같은 검색 실패 문구를 출력하라

3.1 문제 분석

조건 1 Mod연산은 7, TableSize는 10으로 설정

조건 2 만약 삽입 수행 중 충돌 시 증가한 index를 함께 출력

조건 3 데이터에서 입력받을 때 I는 삽입연산, s는 탐색 연산을 수행

- 이 문제는 선형 조사법을 이용하여 데이터 파일에 있는 데이터를 해시테이블에 저장하고, 저장이 되는 과정을 출력한다. 또한 s가 입력되면 탐색을 수행하는데, 탐색을 수행할 때는 위치를 함께 출력한다.

선형 조사법은 해시테이블에서 충돌이 발생한다면 그 다음 인덱스가 비었는지 확인한다. 만약 거기도 비어있지 않다면 그 다음 인덱스를 또 확인한다. 이런식으로 비어있는 공간이 나올 때까지 계속 반복하는 것이다. 만약 테이블의 끝에 도달하면 다시 처음으로 가야하기 때문에 mod연산이 필요할 것이다.

해시테이블은 1차원 배열로 구성된다. 이번 문제에서는 테이블의 크기를 10으로 지정했으므로 바로 배열로 선언하고, 데이터의 자료형은 정수이므로 테이블 구조체 내에는 int형 key값을 하나 선언할 것이다.

삽입과 탐색을 구현하기 위해 버킷이 비었는지 확인하는 매크로인 empty함수를 선언할 것이고 두 개의 항목이 동일한지 검사하는 equal함수를 선언해야 할 것이다.

3.2 소스 코드

```
#define _CRT_SECURE_NO_WARNINGS

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

#define TABLE_SIZE 10
#define MOD 7

typedef struct {
    int key;           //해시 값
    element;           //원소
} element;

element hash_table[TABLE_SIZE]; //해시테이블 선언

void init_table(element ht[]); //해시 테이블 초기화 함수
int hash_function(int n); //해시 값을 사소한 하성 함수

#define empty(item) (item.key == 0) //key가 0인지 확인하는 매크로
#define equal(item1, item2) (item1.key == item2.key) //item1과 item2가 같은지 확인하는 매크로

void hash_add(element item, element ht[]); //해시테이블 삽입 함수
void hash_search(element item, element ht[]); //해시테이블 검색 함수

int main() {
    FILE *fp;
    int tmp;
    element e;
    char c;

    init_table(hash_table); //해시 테이블 초기화

    fp = fopen("data.txt", "r"); //data.txt파일 오픈
    if (!fp) {
        printf("File not open");
        return 0;
    }
    while (!feof(fp)) { //파일마지막 줄까지 읽기 반복
        fscanf(fp, "%c", &c); //파일의 c가 (하위)해시값에 삽입연산
        if (c == '\n') { //단락 c가 (하위)해시값에 삽입연산
            fscanf(fp, "%s", &tmp);
            e.key = tmp;
            hash_add(e, hash_table); //hash_add함수를 호출하여 삽입
        }
        if (c == 'a') { //단락 c가 (하위)해시값에 검색 연산
            fscanf(fp, "%s", &tmp);
            e.key = tmp;
            hash_search(e, hash_table); //hash_search함수를 호출하여 검색
        }
    }
    fclose(fp);
    return 0;
}

void init_table(element ht[]) { //해시 테이블 초기화 함수
    int i;
    for (i = 0; i < TABLE_SIZE; (i)++) {
        ht[i].key = 0; //해시 값을 내의 값을 0으로 초기화
    }
}

int hash_function(int n) { //해시 값을 사소한 하성 함수
    return n % MOD;
}

//소셜 조사법을 이용하여 테이블에 리를 삽입하고, 테이블이 가득 찬 경우에는 리를
void hash_add(element item, element ht[]) {
    int i, hash_value;
    hash_value = i = hash_function(item.key);

    while (!empty(ht[i])) {
        if (equal(item, ht[i])) { //단락 값이 리가 중복될 경우
            printf("data = %d 가 이미 HashTable : %d 에서 중복 값이 ", item.key, i); //중복값의 해시값 출력
        }
        i = (i + 1) % MOD; //mod7연산 리의스 증가
        printf("index = %d로 옮겨야합니다.\n", i);
        if (i == hash_value) { //테이블이 가득 찬 경우
            fprintf(stderr, "테이블이 가득 찼습니다.\n");
            exit(1);
        }
    }
    ht[i] = item; //ht[i]에 item삽입
}

//소셜 조사법을 이용하여 테이블에 리를 검색 리를 검색
void hash_search(element item, element ht[]) {
    int i, hash_value;
    hash_value = i = hash_function(item.key);
    while (!empty(ht[i])) {
        if (equal(item, ht[i])) { //해시 테이블에서 key가 일치할 경우
            printf("%d 은 HashTable : %d에서 검색되었습니다.\n", item.key, i);
            return;
        }
        i = (i + 1) % MOD; //mod7연산 리의스 증가
        if (i == hash_value) { //같은 key가 해시테이블에 있는 경우
            printf("일치하지 않 %d은 HashTable에서 검색되지 않았습니다.\n", item.key);
            return;
        }
    }
    printf("일치하지 않 %d은 HashTable에서 검색되지 않았습니다.\n", item.key);
}
```

3.3 소스 코드 분석

```
/*
    학번 : 20184612
    학과 : 컴퓨터소프트웨어공학과
    이름 : 김동민
    파일 명: 해시테이블 프로그램
*/
#define _CRT_SECURE_NO_WARNINGS

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

#define TABLE_SIZE 10
#define MOD 7

typedef struct {
    int key;           //필드 선언
}element;
```

1. 소스코드를 작성한 날짜, 이름, 프로그램명을 작성하고, 필요한 헤더를 포함한다.

2. Table_SIZE와 나머지 연산을 수행할 MOD를 정의한다. 문제에서 mod는 7로하고 테이블의 크기는 10으로 지정했으므로 이를 선언한다. 해시테이블의 필드를 선언한다. 이 문제의 경우는 정수형 변수 하나만 있기 때문에 key값만 구조체서 삽입한다.

```
void init_table(element ht[]); //해시 테이블 초기화 함수
int hash_function(int n); //제산 함수를 사용한 해싱 함수

#define empty(item)(item.key==0) //key가 0인지 확인하는 매크로
#define equal(item1, item2)(item1.key==item2.key) //item1과 item2가 같은지 확인하는 매크로

void hash_lp_add(element item, element ht[]); //해시테이블 삽입함수
void hash_lp_search(element item, element ht[]); //해시테이블 탐색함수
```

4. 필요한 함수들을 선언한다.

- init_table함수는 해시테이블을 초기화하는 함수이다.
- hash_function함수는 키를 mod연산하여 반환하는 함수이다.
- hash_lp_add함수는 해시테이블에 새로운 값을 삽입하는 함수이다.
- hash_lp_search함수는 해시테이블에서 값을 찾는 함수이다.

5. key가 0인지 확인하는 매크로인 empty와 item1과 item2가 같은지 확인하는 equal매크로를 선언한다.

```

int main() {
    FILE *fp;
    int tmp;
    element e;
    char c;
    element hash_table[TABLE_SIZE]; //해시테이블 선언
    init_table(hash_table);        //해시 테이블 초기화

    fp = fopen("data.txt", "r");    //data.txt파일 오픈
    if (!fp) {
        printf("file not open");
        return 0;
    }
    printf("< HashTableSize = [%d]> \n\n", TABLE_SIZE);
    while (!feof(fp)) {            //데이터를 끝까지 입력받음
        fscanf(fp, "%c", &c);
        if (c == 'i') {            //만약 c가 i라면 해시테이블에 삽입연산
            fscanf(fp, "%d", &tmp);
            e.key = tmp;
            hash_ip_add(e, hash_table); //hash_ip_add함수를 호출하여 삽입
        }
    }
    rewind(fp);
    printf("\n< Find Data Location >\n");
    while (!feof(fp)) {
        fscanf(fp, "%c", &c);
        if (c == 's') {
            fscanf(fp, "%d", &tmp);
            e.key = tmp;
            hash_ip_search(e, hash_table);
        }
    }

    fclose(fp);
    return 0;
}

```

6. 필요한 변수들을 선언한다.

- fp는 파일포인터 tmp는 파일 데이터를 임시로 읽어오는 변수이다.
- element는 해시테이블에 값을 삽입하거나 탐색할 때 쓰는 변수이다.

element형으로 hash_table전역변수를 선언한다. 해시테이블의 크기는 10으로 정해졌기 때문에 배열로 선언한다.

7. 먼저 init_table함수를 통해 해시테이블을 초기화한다.

그리고 fp를 fopen함수를 이용하여 data.txt파일을 오픈한다. 만약 파일이 존재하지 않는다면 file not open을 출력하고 프로그램을 종료한다.

8. feof함수를 이용하여 파일 끝까지 데이터를 입력받는다. 먼저 임시변수 c에 문자를 입력받는다. 만약 c가 I라면 해시테이블에 삽입연산을 수행하고 s라면 탐색연산을 수행한다.

9. 만약 I를 입력받았다면 tmp에 정수 값을 입력받은 후 e의 key값에 삽입하고, 이 값을 hash_lp_add함수를 호출하여 해시테이블에 삽입한다.

10. I로 해시테이블을 모두 입력하고 rewind함수로 파일포인터를 다시 처음으로 되돌린 후 다시 입력받는다. 만약 s를 입력받았다면 tmp에 탐색할 정수 값을 입력받은 후 e의 key값에 삽입하고, 이 값을 hash_lp_search함수를 호출하여 해시테이블에서 key값을 탐색한다.

11. 모든 동작이 끝나면 fclose로 파일포인터를 닫고 프로그램을 종료한다.

```
void init_table(element ht[]) { //해시 테이블 초기화 함수
    int i;
    for (i = 0; i < TABLE_SIZE; i++) {
        ht[i].key = 0; //해시 테이블 내의 값을 0으로 초기화
    }
}

int hash_function(int n) { //제산 함수를 사용한 해싱 함수
    return n % MOD;
}
```

12. init_Table은 해시테이블을 초기화하는 함수이다.

for문을 이용하여 처음부터 해시테이블의 크기만큼 반복하며 해시 테이블의 key값을 0으로 설정한다.

13. hash_fuction함수는 제산 함수를 사용한 해싱함수이다.

제산함수는 나머지 연산을 사용하는 함수이다. 입력받은 n값을 MOD값으로 나머지 연산하고 이 값을 반환한다.

```

//선형 조사법을 이용하여 테이블에 키를 삽입하고, 테이블이 가득 찬 경우는 종료
void hash_lp_add(element item, element ht[]) {
    int i, hash_value;
    hash_value = i = hash_function(item.key);

    while (!empty(ht[i])) {
        if (!equal(item, ht[i])) { //만약 탐색 키가 중복될 경우
            printf("data = %d 저장 도중 HashTable : %d 에서 충돌 감지 ", item.key, i); //중복감지 메시지 출력
        }
        i = (i + 1) % MOD; //mod7연산 인덱스 증가
        printf("- index = %d로 증가하였습니다.\n", i);
        if (i == hash_value) { //테이블이 가득 찬 경우
            fprintf(stderr, "테이블이 가득 찼습니다.\n");
            exit(1);
        }
    }
    ht[i] = item; //ht[i]에 item삽입
}

```

14. hash_lp_add함수는 입력받은 item을 해시 테이블에 삽입하는 함수이다. 먼저 hash_value와 I에 hash_function함수를 사용하여 key를 이용하여 얻은 나머지 값을 삽입한다.

15. while문을 이용하여 버킷이 비어있을 때까지 반복한다. equal매크로를 통해 만약 탐색키가 중복된다면 중복이 된다는 메시지를 출력하고 I에 I+1을 MOD연산하여 I를 증가시킨 후 증가된 I값을 출력한다.

16. 그리고 만약에 I와 hash_value가 같아졌다는 것은 테이블이 가득 찼다는 의미이므로 메시지를 출력한 후 프로그램을 종료한다.

17. while문을 빠져나왔다면 해시테이블의 I인덱스에 item을 삽입한다.

```

//선형 조사법을 이용하여 테이블에 지정된 키를 탐색
void hash_lp_search(element item, element ht[]) {
    int i, hash_value;
    hash_value = i = hash_function(item.key);
    while (!empty(ht[i])) {
        if (equal(item, ht[i])) { //해시 테이블에서 key가 검색된 경우
            printf("%d 는 HashTable : %d에서 검색되었습니다.\n", item.key, i);
            return;
        }

        i = (i + 1) % MOD; //mod7연산 인덱스 증가
        if (i == hash_value) { //찾는 key가 해시테이블에 없는 경우
            printf("입력하신 값 %d는 HashTable에서 검색되지 않았습니다.\n", item.key);
            return;
        }
    }
    printf("입력하신 값 %d는 HashTable에서 검색되지 않았습니다.\n", item.key);
}

```

18. hash_lp_search함수는 해시테이블에서 탐색을 수행하는 함수이다. add함수와 마찬가지로 hash_value와 I에 hash_function함수로 반환된 값을 삽입한다.

19. 만약 equal매크로를 통해 item의 key와 해시테이블의 key가 같다면 검색이 된 것이므로 이를 출력한다. 그리고 I를 I+1의 Mod연산한 값으로 증가시킨다.

20. 만약 I와 hash_value값이 같아진다면 해시테이블에 key가 없다는 의미이므로 이를 출력한다.

해시테이블의 삽입 과정

74 삽입 $\text{mod}7 = 4$						
0	1	2	3	4	5	6
				74		

94 삽입 $\text{mod}7 = 3$						
0	1	2	3	4	5	6
			94	74		

64 삽입 $\text{mod}7 = 1$						
0	1	2	3	4	5	6
	64		94	74		

70 삽입 $\text{mod}7 = 0$						
0	1	2	3	4	5	6
70	64		94	74		

12 삽입
mod7 = 5

0	1	2	3	4	5	6
70	64		94	74	12	

91 삽입
mod7 = 0

0	1	2	3	4	5	6
70	64		94	74	12	



충돌

91 삽입
mod7 = 0

0	1	2	3	4	5	6
70	64		94	74	12	



충돌

91 삽입
mod7 = 0

0	1	2	3	4	5	6
70	64	91	94	74	12	



삽입

106 삽입
mod7 = 1

0	1	2	3	4	5	6
70	64	91	94	74	12	



충돌

106 삽입
mod7 = 1

0	1	2	3	4	5	6
70	64	91	94	74	12	



충돌

106 삽입
mod7 = 1

0	1	2	3	4	5	6
70	64	91	94	74	12	



충돌

106 삽입
mod7 = 1

0	1	2	3	4	5	6
70	64	91	94	74	12	



충돌

106 삽입
mod7 = 1

0	1	2	3	4	5	6
70	64	91	94	74	12	



충돌

106 삽입
mod7 = 1

0	1	2	3	4	5	6
70	64	91	94	74	12	106



삽입

3.4 실행 창

```
< HashTableSize = [10]>
data = 91 저장 도중 HashTable : 0 에서 충돌 감지 - index = 1로 증가하였습니다.
data = 91 저장 도중 HashTable : 1 에서 충돌 감지 - index = 2로 증가하였습니다.
data = 106 저장 도중 HashTable : 1 에서 충돌 감지 - index = 2로 증가하였습니다.
data = 106 저장 도중 HashTable : 2 에서 충돌 감지 - index = 3로 증가하였습니다.
data = 106 저장 도중 HashTable : 3 에서 충돌 감지 - index = 4로 증가하였습니다.
data = 106 저장 도중 HashTable : 4 에서 충돌 감지 - index = 5로 증가하였습니다.
data = 106 저장 도중 HashTable : 5 에서 충돌 감지 - index = 6로 증가하였습니다.
< Find Data Location >
74 는 HashTable : 4에서 검색되었습니다.
94 는 HashTable : 3에서 검색되었습니다.
64 는 HashTable : 1에서 검색되었습니다.
입력하신 값 90는 HashTable에서 검색되지 않았습니다.
12 는 HashTable : 5에서 검색되었습니다.
70 는 HashTable : 0에서 검색되었습니다.
106 는 HashTable : 6에서 검색되었습니다.
입력하신 값 51는 HashTable에서 검색되지 않았습니다.
C:\Users\wdmk46\OneDrive\바탕 화면\2학기 수업자료\자료구조2\실습\week14_3_HashTable
로세스)이(가) 0 코드로 인해 종료되었습니다.
이 창을 닫으려면 아무 키나 누르세요.

scanf(fp, "%c", &c);
if (c == 's') {
    fscanf(fp, "%d", &tmp);
    e.key = tmp;
    hash_ip_search(e, hash_table);
}
```

data - Win
파일(F) 편집

i 74
i 94
i 64
i 70
i 12
i 91
i 106
s 74
s 94
s 64
s 90
s 12
s 70
s 106
s 51

3.5 느낀 점

이번 문제는 해시테이블을 이용하여 파일로부터 데이터를 삽입하고, 값을 찾는 문제였습니다. 해시테이블에 대해서는 처음 배웠기 때문에 초반에는 잘 이해가 되지 않던 부분이 많았었는데, 직접 프로그래밍을 해보니 탐색을 수행하는 데에 상당히 유용한 자료구조라고 생각했습니다. 이번시간에는 선형 조사법에 대해서만 해보았는데 앞으로는 이중해싱과 체이닝에 대해서도 프로그래밍해보며 직접 확인해보고 싶다는 생각이 들었습니다.

벌써 기말고사 날이 다가와서 수업시간을 통해 해싱에 대해 더 이상 배울 수는 없지만 겨울방학을 이용하여 해싱에 대해 더 자세히 배워보고 싶다는 생각을 했습니다. 겨울방학을 이용하여 제가 부족하다고 생각했던 부분을 채울 수 있도록 노력하겠습니다.

4. 느낀 점

드디어 마지막 과제가 끝이 났습니다. 처음에는 매주 나오는 과제 양 때문에 힘들게만 느꼈었는데, 막상 문제를 풀고 나니 제게 도움이 많이 되었던 것 같습니다. 수업을 들으며 잘 이해가 되지 않던 문제들이 많았는데, 직접 실습 시간을 통해 문제를 풀어보며 몰랐던 부분도 알게 되고, 새롭게 배울 수 있던 부분도 많이 있었던 것 같습니다.

겨울방학이 되었다고 놀지 않고 자료구조와 알고리즘에 대한 공부를 꾸준히 하여 다음 학기에 도움이 될 수 있도록 노력하겠습니다.
한 학기 동안 고생 많으셨습니다. 감사합니다.