PA1 보고서

2016311821 한승하

```
typedef struct db {
        struct db *left;
        struct db *right;
        int num;
        char* word;
} db_t;

db_t *db_open(int size);
void db_close(db_t *db);
void db_put(db_t *db, char *key, int key_len,
                        char *val, int val_len);
/* Returns NULL if not found. A malloc()ed array otherwise.
 * Stores the length of the array in *vallen */
char *db_get(db_t *db, char *key, int key_len,
                        int *val_len);
void freeing(db_t *db);
```

db.h 헤더파일 입니다.

db_t의 내용으로는 tree 형 구조를 가지게 하여 insert 시간을 줄이기 위해 left, right 두개의 pointer를 추가하였고, 단어가 들어온 횟수를 기록하기 위한 num, 단어를 기록하기위한 char* 포인터로 구성해 주었습니다.

기본 함수 외에 db_close를 위한 freeing함수를 추가하였으며 이는 이후 close 단계에서 설명하겠습니다.

```
#include "db.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int dbsize;
```

db.c의 첫 부분입니다. Malloc을 위해 stdlib.h를, strcmp함수를 사용하기 위해 string.h를 선언하였으며, 초기 db의 size를 기록하기위해 dbsize를 선언해 주었습니다.

```
db_t *db_open(int size)
{
        int i;
        dbsize = size;
        db_t *db = (db_t*)malloc(sizeof(db_t)*size);
        for(i=0;i<size;i++)
        {
                db[i].word = (char*)malloc(sizeof(char));
                *db[i].word = (char)94;
        }
        return db;
}
```

db_open 함수입니다. 이후 과정에서 사용하기 위하여 size는 global 변수에 저장해 두었습니다.

Hash table로 이용하기 위하여 db_t를 size개의 배열로 선언해 주었으며 각각의 db[i]에 대하여 연결된 right node, left node에만 정보를 저장할 것이고, insert시간을 줄이기 위해 대문자는 left node, 소문자는 right node에 저장하기 위하여 초기 비교대상인 각 head에는 아스키 코드상 중간 값인 94를 저장하여 주었습니다.

```
char *db_get(db_t *db, char *key, int key_len,
                        int *val_len)
{
        char *value = NULL;
        int idx = key_len % dbsize;
        db_t *find;
        if(key[0]>db[idx].word[0]) find = db[idx].right;
        else find = db[idx].left;
        while(find != NULL)
        {
                if(strcmp(key,find->word) == 0)
                {
                        value = (char*)malloc(sizeof(int));
                        memcpy((void*)value,(void*)(&find->num),sizeof(int));
                }
                if(key[0]>find->word[0]) find = find->right;
                else find = find->left;
        }
        return value;
}
```

db_get 함수입니다. Hash function은 길이에 따라 dbsize로 나눈 나머지를 fuction으로 사용하여 모든 dbsize에 대해 작동할 수 있게 하였습니다.

첫 글자의 아스키 value를 기준으로 각 Hash Head에 대해 binary tree 를 구성하였기 때문에 binary tree search를 통해 값을 찾아주고 있다면 memory 할당 이후 memcpy로 binary value 그대로를 복사해 주었습니다. 이는 이후 char* 형으로 넘겨준 value를 int형으로 다시 읽기 위하여 memcpy를 사용하였습니다.

```
void db_put(db_t *db, char *key, int key_len,
                        char *val, int val_len)
{
        int i;
        int idx = key_len % dbsize;
        if(*(int*)val == 1)
        {
                db_t *insert = (db_t*)malloc(sizeof(db_t));
                insert->word = (char*)malloc(sizeof(char)*key_len);
                for(i=0;i<key_len;i++)
                {
                        insert->word[i] = key[i];
                }
                insert->num = 1;
                insert->left = NULL;
                insert->right = NULL;
                db_t *find = NULL;
                if(key[0]>db[idx].word[0])
                {
                        if(db[idx].right == NULL)
                        {
                                db[idx].right = insert;
                                return;
                        }
                        else find = db[idx].right;
                }
                else
                {
                        if(db[idx].left == NULL)
                        {
                                db[idx].left = insert;
                                return;
                        }
                        else find = db[idx].left;
                }
                while(1)
                {
                        if(insert->word[0]>find->word[0])
                        {
                                if(find->right != NULL) find = find->right;
                               else{
                                       find->right = insert;
                                       return;
                               }
                        }
                        else
                        {
                                if(find->left != NULL) find = find->left;
                                else{
                                        find->left = insert;
                                        return;
                                }
                        }
                }
        }
        else
        {
                db_t *find;
                if(key[0]>db[idx].word[0]) find = db[idx].right;
                else find = db[idx].left;
                while(find != NULL)
                {
                        if(strcmp(key,find->word) == 0)
                        {
                                find->num++;
                        }
                        if(key[0]>find->word[0]) find = find->right;
                        else find = find->left;
                }
        }
}
```

db_put 함수입니다.

Memcpy로 복사한 val값을 다시 int형으로 변환하여 integer value를 가져왔습니다.

Val == 1 즉 처음 들어오는 단어는 insert라는 임시 node에 저장하고 앞서 설명했듯 단어의 앞 글자를 기준으로 각각의 Hash value 위치에 binary tree insertion을 해주었습니다.

이때 단어의 맨 앞 글자를 기준으로 아스키 코드 값이 클 경우 right으로 작거나 같을 경우 left으로 insert 해주었습니다.

Val 이 1이 아닐 경우에 저장 되어 있는 단어를 binary tree search를 통해 찾아 num을 증가시켜 저장된 횟수를 기록해 주었습니다.

```
void db_close(db_t *db)
{
        int i;
        for(i=0;i<dbsize;i++)
        {
                if(db[i].left != NULL) freeing(db[i].left);
                if(db[i].right != NULL) freeing(db[i].right);
        }
        free(db);
}
```

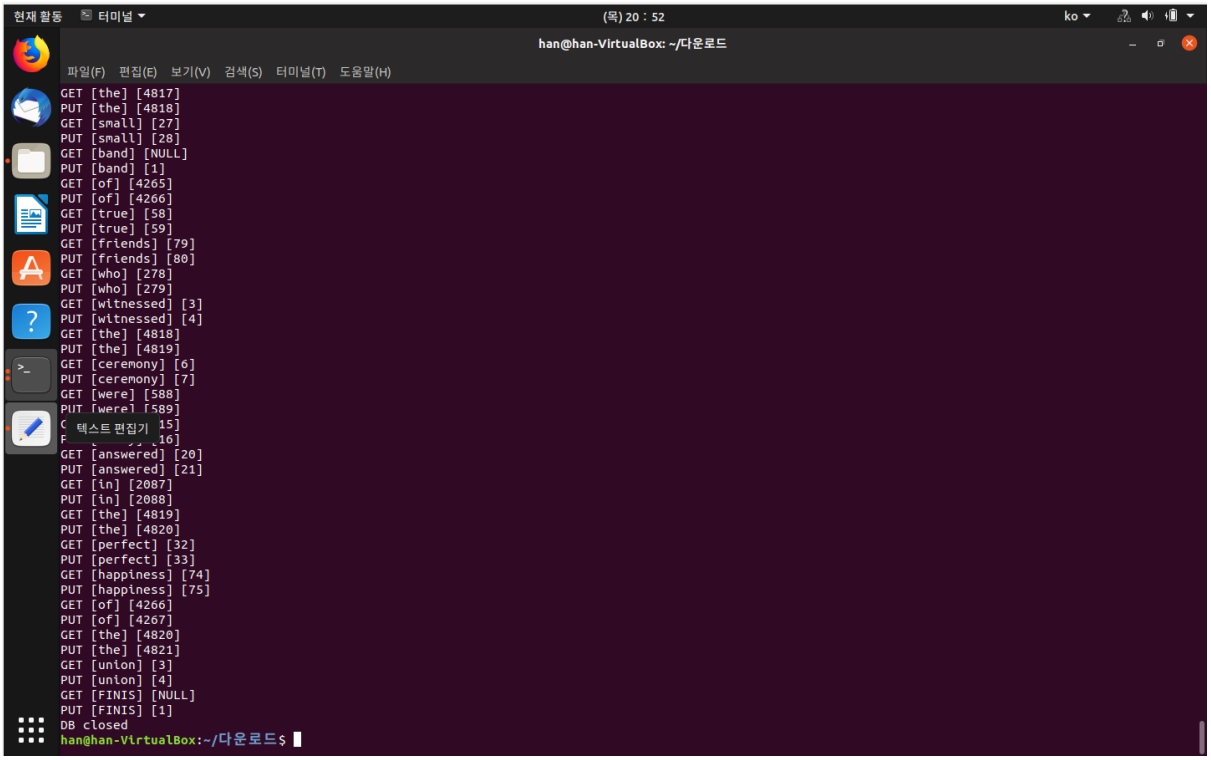
마지막인 db_close입니다.

모든 db[i]에 대해 freeing함수로 각각의 left, right에 달려있는 모든 node를 일일이 free 해주었습니다.

```
void freeing(db_t *db)
{
        if(db->left != NULL) freeing(db->left);
        if(db->right != NULL) freeing(db->right);
        free(db->word);
        free(db);
}
```

이때 freeing함수는 위와 같이 recursion으로 구성되어 존재하는 모든 node를 free할 수 있게 하였습니다.

이후 마지막으로 초기에 선언된 db를 free해줌으로써 모든 db의 메모리 할당을 해지했습니다.



다음은 이번 과제를 실행시켰을 때의 출력화면입니다.