```c
#include <stdio.h>
#include <stdlib.h>         // malloc
#include <string.h>         // strdup
#include <ctype.h>         // isupper, tolower

#define MAX_DEGREE         27 // 'a' ~ 'z' and EOW
#define EOW                          '$' // end of word

// used in the following functions: trieInsert, trieSearch, triePrefixList
#define getIndex(x)                (((x) == EOW) ? MAX_DEGREE-1 : ((x) - 'a'))

// TRIE type definition
typedef struct trieNode {
        int                          index; // -1 (non-word), 0, 1, 2, ...
        struct trieNode        *subtrees[MAX_DEGREE];
} TRIE;

TRIE *trieCreateNode(void);
void trieDestroy( TRIE *root);
int trieInsert( TRIE *root, char *str, int dic_index);
int trieSearch( TRIE *root, char *str);
void trieList( TRIE *root, char *dic[]);
void triePrefixList( TRIE *root, char *str, char *dic[]);
int make_permuterms( char *str, char *permuterms[]);
void clear_permuterms( char *permuterms[], int size);
void trieSearchWildcard( TRIE *root, char *str, char *dic[]);

////////////////////////////////////////////////////////////////////////////////
int main(int argc, char **argv)
{
        TRIE *trie;
        TRIE *permute_trie;
        char *dic[100000];

        int ret;
        char str[100];
        FILE *fp;
        char *permuterms[100];
        int num_p;
        int index = 0;

        if (argc != 2)
        {
                fprintf( stderr, "Usage: %s FILE\n", argv[0]);
                return 1;
        }

        fp = fopen( argv[1], "rt");
        if (fp == NULL)
        {
                fprintf( stderr, "File open error: %s\n", argv[1]);
                return 1;
        }

        trie = trieCreateNode(); // original trie
        permute_trie = trieCreateNode(); // trie for permuterm index

        while (fscanf( fp, "%s", str) != EOF)
        {
                ret = trieInsert( trie, str, index);

                if (ret)
                {
                        num_p = make_permuterms( str, permuterms);

                        for (int i = 0; i < num_p; i++)
                                trieInsert( permute_trie, permuterms[i], index);

                        clear_permuterms( permuterms, num_p);

                        dic[index++] = strdup( str);
                }
        }

        fclose( fp);

        printf( "\nQuery: ");
        while (fscanf( stdin, "%s", str) != EOF)
        {
                // wildcard search term
                if (strchr( str, '*'))
                {
                        trieSearchWildcard( permute_trie, str, dic);
                }
                // keyword search
                else
```

```
                {
                        ret = trieSearch( trie, str);
                        if (ret == -1) printf( "[%s] not found!\n", str);
                        else printf( "[%s] found!\n", dic[ret]);
                }
                printf( "\nQuery: ");
        }

        for (int i = 0; i < index; i++)
                free( dic[i]);

        trieDestroy( trie);
        trieDestroy( permute_trie);

        return 0;
}









/*
typedef struct trieNode {
        int                         index; // -1 (non-word), 0, 1, 2, ...
        struct trieNode        *subtrees[MAX_DEGREE];
} TRIE;
*/


/* Allocates dynamic memory for a trie node and returns its address to caller
        return        node pointer
                        NULL if overflow
*/
TRIE *trieCreateNode(void)
{
        TRIE* newTrie = (TRIE*)malloc(sizeof(TRIE));
        newTrie->index = -1;

        for(int i = 0; i < MAX_DEGREE; i++){
                (newTrie->subtrees)[i] = NULL;
        }


        return newTrie;
}

/* Deletes all data in trie and recycles memory
*/
void trieDestroy( TRIE *root)
{
        for(int i = 0; i < MAX_DEGREE; i++){
                if(root->subtrees[i] != NULL) trieDestroy(root->subtrees[i]);
        }

        free(root);
}

/* Inserts new entry into the trie
        return        1 success
                        0 failure
*/
// 주의! 엔트리를 중복 삽입하지 않도록 체크해야 함
// 대소문자를 소문자로 통일하여 삽입
// 영문자와 EOW 외 문자를 포함하는 문자열은 삽입하지 않음
int trieInsert( TRIE *root, char *str, int dic_index)
{
        int n = strlen(str);
        for(int i = 0; i < n; i++){
                if(isupper(str[i])) str[i] = tolower(str[i]);
                if(('a' > str[i] || 'z' < str[i]) && str[i] != EOW) return 0;
        }

        for(int i = 0; i < n; i++){
                if(root->subtrees[getIndex(str[i])] == NULL) root->subtrees[getIndex(str[i])] = trieCreateNode();
                root = root->subtrees[getIndex(str[i])];
        }
        root->index = dic_index;
        return 1;
}

/* Retrieve trie for the requested key
```

```
        return        index in dictionary (trie) if key found
                        -1 key not found
*/
int trieSearch( TRIE *root, char *str)
{
        int n = strlen(str);
        for(int i = 0; i < n; i++){
                if(root->subtrees[getIndex(str[i])] == NULL) return -1;
                root = root->subtrees[getIndex(str[i])];
        }
        return root->index;
}

/* prints all entries in trie using preorder traversal
*/
void trieList( TRIE *root, char *dic[])
{
        if(root->index != -1){
                printf("%s\n", dic[root->index]);
        }

        for(int i = 0; i < 27; i++){
                if(root->subtrees[i] != NULL){
                        trieList(root->subtrees[i], dic);
                }
        }
}

/* prints all entries starting with str (as prefix) in trie
        ex) "abb" -> "abbas", "abbasid", "abbess", ...
        this function uses trieList function
*/
void triePrefixList( TRIE *root, char *str, char *dic[])
{
        for(int i = 0; i < strlen(str); i++){
                if(root->subtrees[getIndex(str[i])] == NULL) return;
                root = root->subtrees[getIndex(str[i])];
        }

        trieList(root, dic);
}

/* makes permuterms for given str
        ex) "abc" -> "abc$", "bc$a", "c$ab", "$abc"
        return        number of permuterms
*/
int make_permuterms( char *str, char *permuterms[])
{
        char temp;
        int n = strlen(str);
        int num = 0;
        str[n] = EOW;
        str[n + 1] = '\0';
        permuterms[num++] = strdup(str);
        for(int i = 0; i < n; i++){
                temp = str[0];
                for(int j = 0; j < n; j++){
                        str[j] = str[j + 1];
                }
                str[n] = temp;
                permuterms[num++] = strdup(str);
        }

        for(int j = 0; j < n; j++){
                str[j] = str[j + 1];
        }

        str[n] = '\0';
        return num;
}

/* recycles memory for permuterms
*/
void clear_permuterms( char *permuterms[], int size)
{
        for(int i = 0; i < size; i++){
                free(permuterms[i]);
        }
}

/* wildcard search
        ex) "ab*", "*ab", "a*b", "*ab*"
        this function uses triePrefixList function
*/
void trieSearchWildcard( TRIE *root, char *str, char *dic[])
{
        int n = strlen(str);
```

```
	char c;

	if(str[0] == '*' && str[n - 1] == '*'){
		for(int i = 0; i < n - 1; i++){
			str[i] = str[i + 1];
		}
		str[n - 2] = '\0';
		triePrefixList(root, str, dic);
		return;
	}
	str[n] = EOW;
	while(str[n] != '*'){
		c = str[0];
		for(int i = 0; i < n; i++){
			str[i] = str[i + 1];
		}
		str[n] = c;
	}

	str[n] = '\0';

	triePrefixList(root, str, dic);
}
```