アルゴリズムとデータ構造　第 10 回演習　　年　　組　　番 氏名

1. 配列は**ランダムアクセス**が可能なデータ構造であり，整数の添字（インデックス）で要素を参照することができる。この特徴を用いると，整数からデータへの変換表を作ることができる。
以下は，配列を用いて 1 桁の整数を英単語に“翻訳”するプログラムである。if 文の中に ch から n を求める適切な計算式を挿入してプログラムを完成させ，実行して動作を確認せよ。

```
#include <stdio.h>

char *ewords[] ={
    "zero", "one", "two", "three", "four",
    "five", "six", "seven", "eight", "nine"
};

int main(void)
{
    int ch, n;

    printf("number? ");
    ch = getchar();  /* 1 文字入力 */

    if ('0' <= ch && ch <= '9') {


        printf("%d: %s¥n", n, ewords[n]);
    }
    return 0;
}
```

2. **ハッシュ関数**は**キー**と呼ばれるデータを入力とし，それからなるべくバラバラな整数値（**ハッシュ値**）を算出する関数である。以下は文字列のハッシュ関数の例であり，入力された文字列に対してなるべく重複しない整数を生成する。このハッシュ関数によって，次の文字列はどのような数値に変換されたか。

(a) zero _____　(b) one _____　(c) two _____　(d) three _____　(e) four _____

(f) five _____　(g) six _____　(h) seven _____　(i) eight _____　(j) nine _____

```
#include <stdio.h>

#define HASHSIZE 31  /* 素数がよい */

/* ハッシュ関数の例
   任意の文字列を 0〜30 の整数に変換する */
int hash(char *key)
{
    unsigned h = 0;
    int i;

    for (i = 0; key[i] != '¥0'; i++) {
        h = 37 * h + key[i];  /* 別の素数 */
    }
    return (int) (h % HASHSIZE);
}
```

```
int main(void)
{
    char buf[100];
    int h;
    int i;

    for (i = 0; i < 10; i++) {
        printf("文字列？ ");
        scanf("%s", buf);
        /* ハッシュ関数によって整数に変換 */
        h = hash(buf);
        printf("¥"%s¥" -> %d¥n", buf, h);
    }
    return 0;
}
```

3. **ハッシュテーブル**は，データをキーのハッシュ値を添字とする配列要素に格納することによって，探索を高速化する仕組みである。下記はハッシュテーブルの原理を理解するための単純なプログラムである。配列の添字に 2.で求めた整数を埋めて動作を確認せよ。問題点を考察せよ。

```
#include <stdio.h>
#include <stdlib.h>
#define HASHSIZE 31  /* 2.と同じ値にする */

/* 文字列の配列 */
char *hashtable[HASHSIZE];

int hash(char *key)
{
    /* 2.と全く同じ内容 */
}
```

```
int main(void)
{
    int i;
    int h;
    char buf[100];

    /* ハッシュテーブルの初期化 */
    for (i = 0; i < HASHSIZE; i++) {
        hashtable[i] = NULL;
    }
```

```
    /* 初期データの登録 */

    hashtable[    ] = "zero";
    hashtable[    ] = "one";
    hashtable[    ] = "two";
    hashtable[    ] = "three";
    hashtable[    ] = "four";
    hashtable[    ] = "five";
    hashtable[    ] = "six";
    hashtable[    ] = "seven";
    hashtable[    ] = "eight";
    hashtable[    ] = "nine";

    printf("探索文字列？ ");
    scanf("%s", buf);

    h = hash(buf);
    if (hashtable[h] == NULL) {
        printf("見つからなかった¥n");
    } else {
        printf("見つかった？¥n");
        printf("%s¥n", hashtable[h]);
    }
    return 0;
}
```

4. 3.のようなハッシュテーブルでは，同じハッシュ値のデータ（ハッシュ値が**衝突**するデータ）を複数登録することができない。**チェーン法**ではハッシュテーブルの各要素を連結リストへのポインタにすることによって，複数のデータの登録を可能とする。下記は，単語の出現回数を数えるプログラムの一部である。空欄を埋めた後，適当な main 関数を補ってチェーン法によるハッシュテーブルの仕組みを理解せよ。

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

/* 登録見込みの単語数程度の素数にする */
#define HASHSIZE 59

/* 連結リストのノード */
struct node {
    char word[20];      /* 単語（キー） */
    int count;          /* 出現回数 */
    struct node *next;  /* 次のノード */
};

struct node *hashtable[HASHSIZE];

int hash(char *key)
{
     /* 2.と全く同じ内容 */
}

/* 全要素を表示する */
void print_hashtable(void)
{
    int i;
    struct node *p;

    for (i = 0; i < HASHSIZE; i++) {
        printf("%3d:", i);
        for (p = hashtable[i]; p != NULL;
                               p = p->next) {
            printf(" %s(%d)", p->word, p->count);
        }
        putchar('¥n');
    }
}

/* 単語の検索 */
struct node *find_word(char *word)
{
    int h;
    struct node *p;

    h =
    p = hashtable[   ];

    /* リストをたどって探索する */
    while (p != NULL) {
        if (strcmp(word, p->word) == 0)
            break;
        p = p->next;
    }
    return p;
}

/* 新しい単語の登録 */
struct node *put_word(char *word)
{
    int h;
    struct node *p;

    p = (struct node *)
            malloc(sizeof(struct node));
    strcpy(p->word, word);
    p->count = 0;

    /* リストの先頭に挿入する */
    h = hash(word);
    p->next =

    hashtable[   ] = p;
    return p;
}
```

5. 探索速度（計算量）の観点から，線形探索と 2 分探索に対するハッシュテーブルの利点を考察せよ。