簡単なデータ構造

辞書検索

辞書の構造

計算機の処理では、大量のデータの中から目的のデータに素早くたどり着けるような手段を必要とする場合が多いものです。これは計算機に限ったことではなく、古くは電話帳や、語学で単語の意味を調べる時に、大量のデータの中から目的のデータを引くという作業はよくあることです。目的のデータに対して、小さくて扱いやすいデータを検索に使用することで、データのアクセス性を高めるようなデータの持ち方は、**辞書**と呼ばれることがあります。

辞書は**マップ**とも言われ、実データを素早く探索できる索引としてのキーを登録することで、効率的に実データにアクセスできるようなデータ構造です。

キーの探索を高速化する手法はいろいろ考えられます。ある英単語、たとえば、`what` を調べる時、辞書の先頭 `a` から順に探索する人は殆どいないのと同様に、データを先頭から探索するよりも効率の良い方法や、検索する範囲を絞り込む方法をとることが必要です。

一つの方法として、索引をソートしておくことで、探索を効率化する方法が考えられます。ただし、検索対象が大量の場合には、ソートの時間、あるいはそれに替えてポインタを動的に挿入する時間が無視できないほど大きい、という問題が生じる場合もあります。

今回の課題では、**ハッシュ関数**という特殊な辞書構造を扱うことにします。

ハッシュ関数 とは、与えられたデータを、ある範囲の中になるべく一様に分布するような整数に変換するような演算を行う関数です。よく使われるハッシュ関数として、データを除算して余りをハッシュ値とする方法があります。

$$H(k) = k \bmod N$$

N の値として、素数を使うことが推奨されています。

何故ハッシュ値による分割が効率的かというと、たとえば `100` 万件のデータを単純に前から探索する方法だと、最悪 `100` 万回のデータ照合が必要になりますが、`500111` （これは素数です）での剰余でデータが一様に分布されたとすると、平均 `2` 回の照合作業で済むからです。

`int` の値であれば、剰余を求めることは簡単ですが、文字列のハッシュ値を上の方法で求めるためには、多倍長演算を行います。多倍長演算というと難しそうですが、小学生で習う割り算と考え方は同じです。

文字列は `char[]` なので、符号なしの整数で考えて、文字列を `256` 進数の数値とみなすと、`unsigned char` で演算を行うことが出来ます。この場合は、文字列の先頭が数値の上位となるような演算をとることになります。

`"(= "` という文字列の場合、 `40 61 32` という数値の並びとなる

(	=		\0

40	61	32	\0

16進数で 40, 61, 32 はそれぞれ 0x28, 0x3D, 0x20 と表す

筆算形式だと下図のようになります。左側は、16 進表記、右側が 10 進で同じ数値を示したものです。

```
       0x5  0xBF  0x96          5                  191                 150
     __________________      ____________________________________________
7 ) 0x28  0x3D  0x20     7 ) 40           61                   32
    0x23                     35
    __________               ______
    0x 5    3D                5 * 0x100 + 61 = 1341
    0x 5    39                                 1337
          __________                           ____
          0x 4    20                              4 * 0x100 + 32 = 1056
          0x 4    1A                                               1050
          __________                                               ____
                0x 6                                                  6
```

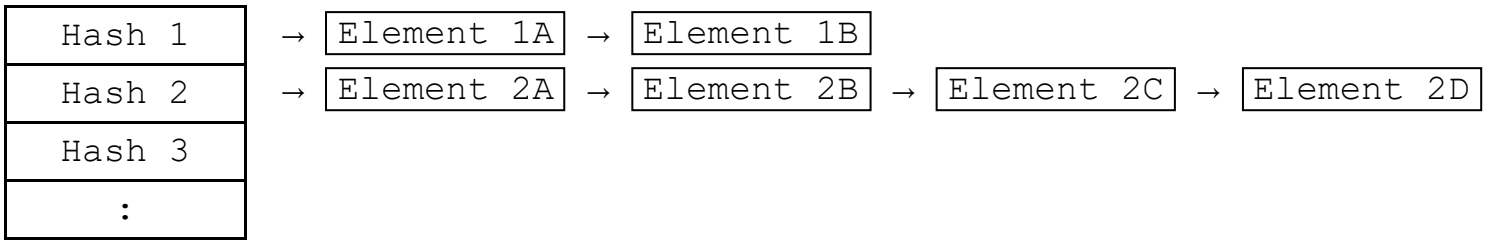
多倍長演算（剰余）の考え方

- 被除数は 40 / 61 / 32 という 3 桁の数として考える
- 最初の桁 40 から始める
- 7 で割って、余り 5

- 16 進表記では、2 桁で 256 進数を表すので、次の 3D と合わせて、 0x53D となる
- 0x53D とするには、余り 5 に 0x100 (=256) を乗じて、 5 * 256 + 61 = 1341 となる
- 1341 は 0x53D の 10進表記。左側と右側の数値は同じものを表していることに注意
- 得られた 1341 に対して繰り返し処理を行う
- ヌル文字 \0 が出現したら、処理は終了する

除数 N の剰余を使うと、ハッシュ値は 0 ～ N - 1 の範囲に収まるため、データを要素数 N の 1 次元配列に格納できます。同じハッシュ値を持つ別のデータが現れた場合、データをリストで扱うことで、可変長データに対応できます。

ハッシュを使った簡単な辞書構造



このとき、辞書自体はデータを格納する構造（リスト）へのポインタの配列、という構造をとります。辞書は、要素数 N のポインタの配列で、NULL で初期化されており、データが追加されるときに、該当するハッシュ値の配列要素が NULL であればその値を更新し、既にリストがあれば、リストの先頭に挿入（最後尾に追加するでも構いませんが、先頭に追加する方が楽でしょう）することになります。

演習課題 5-1

与えられた英単語・訳語データを検索可能なデータ構造として構築する関数と、その構築データからキーに一致する英単語を検索する関数、および、データを振り分けるハッシュ関数を作成して下さい。

- それぞれの関数は `1` つのファイルの中に書いて下さい。ファイル名を `kadai_5_1.c` として下さい。
- ハッシュ関数の仕様は、次の通りです
  - 関数プロトタイプは次の通りです。

```
unsigned int kadai_5_1_hash(char *str);
```

  - 上で説明されている多倍長演算に従い、`127` で割った剰余をハッシュ値として下さい
  - `NULL` が入力されたり、文字列長が `0` だった場合のハッシュ値は `0` とします

- 構築済みの辞書データから、キーに一致する英単語を検索する関数の仕様は、次の通りです
  - 関数プロトタイプは次の通りです。

```
char *kadai_5_1_search(WordList *dictionary[], char
*keyword);
```

  - 単語データは、次の `WordList` 構造体にリスト形式で格納されているものとします

```
typedef struct wordlist {
    char *eng;
    char *jpn;
    struct wordlist *next;
} WordList;
```

- eng メンバーに英単語の文字列が、jpn メンバーにその意味が日本語の文字列で格納されています
- 辞書は、WordList 構造体ポインタの配列で、英単語のハッシュ値が配列インデックスになります
- keyword 文字列で渡された文字列と同じ物を検索し、一致したものが存在する場合、その意味データを返します
- keyword に一致する英単語が存在しない場合、NULL を返します

- 英単語と訳語を受け取り、辞書データに追加する関数の仕様は、次の通りです
  - 関数プロトタイプは次の通りです。

    ```
    void kadai_5_1_append(WordList *dictionary[], char *e_word, char *j_word);
    ```

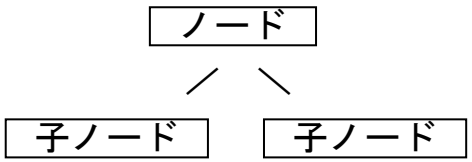
  - 単語データは、WordList 構造体のメモリを割り当てて、リスト形式で格納します
  - eng がキーとなります。キーが既にある場合は、jpn ポインタを書き換えて、重複データがないようにします
  - 単語文字列は静的に格納されています。従って、文字列を扱うときにポインタ値をコピーして単語データを構築しても構いません

- 文字列の比較には `strcmp()` を使ってもよいです。`ctypes.h` で宣言されています。
- `strcmp()` では、文字列が同じ時に、`0` が、小さいときには負の数が、そして、大きいときには `0` より大きい数が返されます。`int strcmp(const char*, const char*)` として宣言されています。

B-Tree

ツリー構造と B-Tree

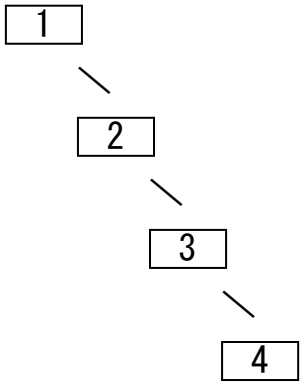
ツリー構造については、前に学習しました。



このツリーは**二分木**といわれるもので、ノードから枝が最大 `2` 本出ていて、ノードの値より小さい枝、または大きい枝、というように、データの値により分岐していくという性質を持つものでした。

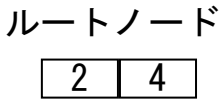
二分木にデータを追加する場合、ノードと比較して小さいか大きいかにより枝を探索していき、進むべき枝がそれ以上無いときにその場所にデータを挿入するという動作を行います。この動作は単純なこともあり、実装することは簡単ですが、データの挿入の順番によっては木が偏ってしまうことが知られています。

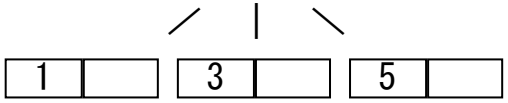
既にソートされているデータから二分木を構築すると、以下のようなツリー構造が作られます。



このようにあらかじめソートされているような最悪の場合は、リストと同じになってしまいます。**B-Tree** はどのような順序でデータを追加しても、ツリー構造がバランスよくなるようなアルゴリズムを持つツリーのひとつです。二分木は **Binary Tree** とも言われますので、B-Tree と混同しないように注意して下さい。

B-Tree では、ノードに N 本の枝と、N - 1 個の要素を含みます。ここでは、要素数が 2 つ、枝が 3 本の B-Tree を考えてみることとします。



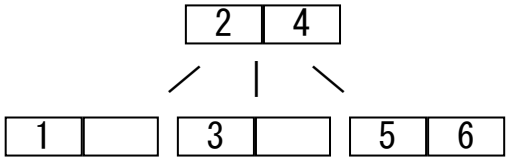


一番上のノードがルート（根）ノードを表しています。ノード内には、要素 2, 4 が昇順に並んでいて、要素の大小関係と枝の分岐が関係しています。つまり、左側の枝は、要素 2 より小さい分岐、中央の枝は、2 より大きくて 4 より小さい分岐、右側の枝は、4 より大きい分岐、ということになります。

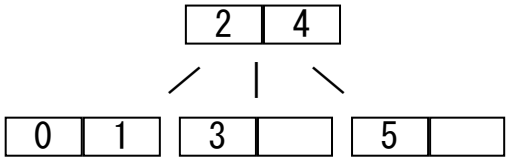
図の状態では、それぞれの子ノードは要素が 1 つで、左要素はありません。このノードに要素が追加される場合を考えてみましょう。

上図の状態で、データ 6 を追加するとします。まず、ルートノードから探索が開始されます。ノード内に 6 があるかどうか探索しますが 6 は見つからないため、この先に進めるかどうか調べます。 6 は 4 より大きいので、右側の枝を調べ、枝が次のノードを指しているので先に進みます。枝が指し示すノードには、6 はありませんので、先に進める枝があるかどうか調べます。先に進む枝がないので、要素 6 はこのツリーに無いということが分かります。

データ 6 をツリーに追加することが決定しましたので、このノードの要素数を調べます。要素が満員でないため、このノードにデータ 6 を格納できることが分かります。6 は 5 より大きいので、右側に 6 が格納されました。



小さい場合の挿入も同様に行われます。下の図は、0 を追加する場合、1 が格納されているノードに追加された様子を示したものです。



続いて、ノードが満員の場に、どのようにデータが追加されるのかを見ていきます。

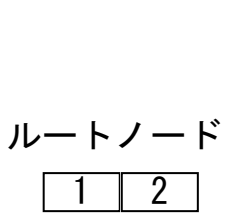


図 1

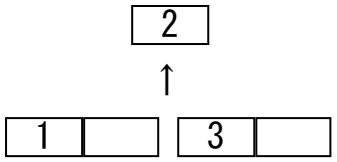


図 2

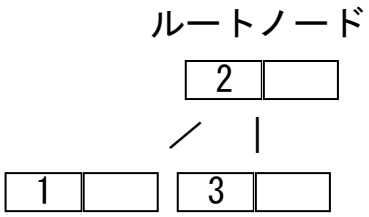


図 3

図 1 は、ルートノードにデータ 1、2 が追加された様子を示しています。ここにデータ 3 が追加されるとき、要素を格納できるように分割が行われます。分割は、中間のデータ 2 を境に、小さい側のノード（ここではデータ 1 ）と大きい側のノード（ここではデータ 3）に分割されます。(図 2)

中間データは、このデータ 1、 2 に対する親ノードとなります。ここでは、元々の親ノードはありませんでしたから、親ノードが作成されて、新しいルートノードとなります。(図 3)

ここで、データ 4 が追加され、図 4 の状態となり、さらに、データ 5 が追加されることを考えます。上で見てきたように、ノードの分割が起こり、中間のデータ 4 がデータ 3, 5 に対する親になろうとします。(図 5)

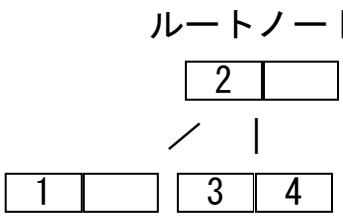


図 4

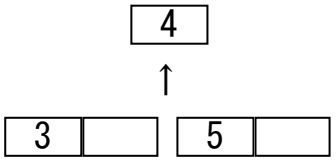


図 5

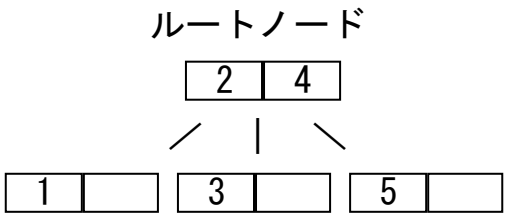


図 6

分割されたノードには、親ノードがあり、要素を格納できる余地があるため、データ 4 が要素としてセットされます。(図 6)

演習課題 5-2

与えられた英単語、訳語から B-Tree データ構造を構築するプログラムを完成させて下さい。

- ファイル名を kadai_5_2.c として下さい。

下にある演習プログラムは、ところどころコードを埋める箇所があります。空欄部分は、

のように、枠で示されています。ただし、枠の大きさはコードの量とは関係がありません。

```
/*
 * B-Tree 演習プログラム
 */

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

/* ノード内の要素の型 */
typedef struct {
    char *key;      /* key string*/
    char *value;    /* data string */
} ElemType;



/* B-Tree ノードの型 */
typedef struct treenode {
    /* ノード内の要素 */
```

```
    ElemType    left;   /* 最初に格納されるデータ */
    ElemType    right;  /* 追加されたデータ (left < right) */

    /* 子孫ノードへのリンク */
    struct treenode *child_left;    /* データ < left */
    struct treenode *child_center;  /* left < データ < right */
    struct treenode *child_right;   /* right < データ */
} TreeNode;



/*
 *  関数プロトタイプ
 */

/* B-Tree のノードをメモリに割り当てる */
TreeNode *createNode();

/* B-Tree に ノードを追加する。
   子孫から繰り上がりのノードがあれば、ノードポインタが返される */
TreeNode *insertNode(TreeNode *tree, char *key, char *value);

/* B-Tree に データを追加する。
   Tree が NULL のとき、新規 Tree を作成する。 */
TreeNode *addTree(TreeNode *tree, char *key, char *value);
```

```
/* B-Tree 表示関数 */
void printTree(TreeNode *p);
void listTree(TreeNode *p, int lv);

/*
 *  マクロ定義
 */

#define SWAP(x,y) ¥
{                 ¥
    void *temp;   ¥
    temp = (x);   ¥
    (x) = (y);    ¥
    (y) = temp;   ¥
}

/*
 *  関数定義
 */

/* B-Tree のノードをメモリに割り当てる */
TreeNode *createNode()
{
    TreeNode *tree = malloc(sizeof (TreeNode));
```

```
    if (tree != NULL) {
        /* メンバーの初期化 */
        tree->left.key = NULL;
        tree->left.value = NULL;
        tree->right.key = NULL;
        tree->right.value = NULL;
        tree->child_left = NULL;
        tree->child_center = NULL;
        tree->child_right = NULL;
    }

    return tree;
}


/* B-Tree に データを追加する。
   Tree が NULL のとき、新規 Tree を作成する。 */
TreeNode *addTree(TreeNode *tree, char *key, char *value)
{
    if (tree == NULL) {
        /* Tree がない場合、新規作成する */
        tree = createNode();
    }

    if (tree != NULL) {
```

```
        /* ノードへのデータ追加処理本体 */
        TreeNode *new_root = insertNode(tree, key, value);
        if (new_root != NULL)
            /* B-Tree がバランスされ、新しいルートノードが設定された */
            tree = new_root;
    }

    return tree;
}


/* B-Tree に ノードを追加する。
   子孫から繰り上がりのノードがあれば、ノードポインタが返される */
TreeNode *insertNode(TreeNode *tree, char *key, char *value)
{
    TreeNode *carry_node = NULL;

    /*
     * まず、ノード内の要素を探索する
     */
    if ((tree->left.key != NULL) && strcmp(key, tree->left.key) == 0) {
        tree->left.value = value;
        return NULL;
    }
    if ((tree->right.key != NULL) && strcmp(key, tree->right.key) == 0) {
```

```
    return NULL;
}

/*
 * ノード内の要素と一致しなかったので、子孫リンクがあれば進む
 */
/* child_left の処理 */
if ((tree->left.key != NULL) && strcmp(key, tree->left.key) < 0) {
    /* left より小さい場合、chid_left があれば進む */
    if (tree->child_left != NULL) {
        carry_node = insertNode(tree->child_left, key, value);
        if (carry_node == NULL)
            /* 子孫から繰り上がりのノードが無ければ終了 */
            return NULL;
    }
}
/* child_right の処理 */
else if ((tree->right.key != NULL) && strcmp(key, tree->right.key) > 0) {
    /* right より大きい場合、chid_right があれば進む */
```

```
}
/* child_center の処理 */
else {
    /* left より大きく right より小さいので、child_center があれば進む */
    if (tree->child_center != NULL) {

    }
}

/*
 * 要素を格納する処理
 *   1) 子孫へのリンクがない場合、left / right にデータを格納する
 *   2) 子孫からの繰り上がりがあった場合、繰り上がりノードのリンクを含めてマージ
 *   3) データが満杯だった場合は、繰り上がりノードを作成して return
 */

/* left / right に格納できるかどうか */
if ((tree->left.key == NULL) || (tree->right.key == NULL)) {
```

```
    /* left / right に格納できる */
    if (carry_node == NULL) {
        /* 繰り上がりではない → 単純にデータをセット */
        if (tree->left.key == NULL) {
            tree->left.key = key;
            tree->left.value = value;
        }
        else {
            if (strcmp(key, tree->left.key) < 0){
                /* データ < left の場合、元の left 位置にデータを挿入 */
                tree->right.key = tree->left.key;
                tree->right.value = tree->left.value;
                tree->left.key = key;
                tree->left.value = value;
            }
            else {
                /* right < データ の場合、right 位置にデータを設定 */
```

```
            }
        }
    }
```

```
        else {
            /* 子孫からの繰り上がり → ノードをマージ */
            if (strcmp(carry_node->left.key, tree->left.key) < 0){
                /* データ < left の場合 */
                tree->right.key = tree->left.key;
                tree->right.value = tree->left.value;
                tree->child_right = tree->child_center;

                tree->left.key = carry_node->left.key;
                tree->left.value = carry_node->left.value;

                tree->child_left = carry_node->child_left;
                tree->child_center = carry_node->child_center;
            }
            else {
                /* left < データ の場合 */
                tree->right.key = carry_node->left.key;
                tree->right.value = carry_node->left.value;

                tree->child_right = carry_node->child_center;
            }
        }
    }
    else {
        /* left / right に格納できない場合、分割・繰り上がりが発生する */
```

```
    /* 分割・繰り上がり用のノードを作成する */
    TreeNode *new_node = createNode();
    TreeNode *sortlist[3];

    /* 挿入対象のデータは、引数データか繰り上がりデータのどちらか */
    if (carry_node == NULL) {
        /* 引数データをノード化する */
        carry_node = createNode();
        carry_node->left.key = key;
        carry_node->left.value = value;
        /* left ノードが最小のデータとなるように調整 */
        if (strcmp(key, tree->left.key) < 0) {
            SWAP(carry_node->left.key, tree->left.key);
            SWAP(carry_node->left.value, tree->left.value);
        }
    }

    /* ノードを分割して、right データを分割ノードにコピー */
    new_node->left.key = tree->right.key;
    new_node->left.value = tree->right.value;

    new_node->child_left = tree->child_center;
    new_node->child_center = tree->child_right;

    tree->right.key = NULL;
```

```
    tree->right.value = NULL;
    tree->child_right = NULL;

    /*
     * ノード内の要素をソートして、中間データを選出する
     */
     /* tree (left) ノードが最小のデータとなるように調整  */
    if (strcmp(carry_node->left.key, tree->left.key) < 0) {
        SWAP(tree->left.key, carry_node->left.key);
        SWAP(tree->left.value, carry_node->left.value);
        SWAP(tree->child_left, carry_node->child_left);
        SWAP(tree->child_center, carry_node->child_center);
    }
    sortlist[0] = tree;
    sortlist[1] = carry_node;
    sortlist[2] = new_node;
    /* tree (left) < new_node (right) なので、
       carry_node の位置に着目してソートする */
    if (strcmp(sortlist[0]->left.key, sortlist[1]->left.key) > 0) {
        /* Swap sortlist[0] <--> sortlist[1] */
        SWAP(sortlist[0], sortlist[1]);
    }
    if (                                                    ) {
        /* Swap sortlist[1] <--> sortlist[2] */
        SWAP(sortlist[1], sortlist[2]);
```

```
        }

        /* right ノードの子孫リンクの調整 */
        sortlist[2]->child_left = sortlist[1]->child_center;
        /* 繰り上げノードに子孫リンク left / right を設定 */
        sortlist[1]->child_left = sortlist[0];
        sortlist[1]->child_center = sortlist[2];

        /* 中間ノードが繰り上がる */
        return
    }

    return NULL;
}



/*************************************************************/


typedef struct {
    char *eng;
    char *jpn;
} WordList;
```

```
#define WORDLISTSIZE(x) (sizeof (x) / sizeof (WordList))

WordList wordlist[] = {
    { "07", "seven" },
    { "15", "fifteen" },
    { "21", "twenty-one" },
    { "04", "four" },
    { "14", "fourteen" },
    { "23", "twenty-three" },
    { "05", "five" },
    { "18", "eighteen" },
    { "22", "twenty-two" },
    { "17", "seventeen" },
    { "01", "one" },
    { "13", "thirteen" },
    { "09", "nine" },
    { "20", "twenty" },
    { "19", "nineteen" },
    { "06", "six" },
    { "08", "eight" },
    { "03", "three" },
    { "10", "ten" },
    { "02", "two" },
    { "12", "twelve" },
    { "11", "eleven" },
```

```
    { "24", "twenty-four" },
    { "16", "sixteen" }
};



void disp_Tree(TreeNode *p)
{
    if (p == NULL) {
        printf(" . ");
        return;
    }
    printf(" (");
    disp_Tree(p->child_left);
    printf("¥"%s¥"", p->left.key);
    disp_Tree(p->child_center);
    printf("¥"%s¥"", p->right.key);
    disp_Tree(p->child_right);
    printf(") ");
}

void printTree(TreeNode *p)
{
    if (p) {
        disp_Tree(p);
```

```
        printf("¥n");
    }
}

void listTree(TreeNode *p, int lv)
{
    int i;
    char spc[64];
    for (i = 0; i < sizeof (spc); i++)
        spc[i] = ' ';
    spc[lv * 2] = '¥0';

    if (p) {
        if (p->child_left)
            listTree(p->child_left, lv + 1);
        if (p->left.key)
            printf("%s|L> %s: %s¥n", spc, p->left.key, p->left.value);
        if (p->child_center)
            listTree(p->child_center, lv + 1);
        if (p->right.key)
            printf("%s|R> %s: %s¥n", spc, p->right.key, p->right.value);
        if (p->child_right)
            listTree(p->child_right, lv + 1);
    }
}
```

```
int main()
{
    int i;
    TreeNode *root = NULL;

    for (i = 0; i < WORDLISTSIZE(wordlist); i++) {
        printf("%d  %s: %s¥n", i, wordlist[i].eng, wordlist[i].jpn);

        root = addTree(root, wordlist[i].eng, wordlist[i].jpn);
        if (root == NULL) {
            perror("malloc error¥n");
            return 1;
        }
        if (i == -1) {
            printf("quit %d...¥n", i);
            break;
        }
        /* printTree(root); */
    }

    printf("¥nlisting:¥n");
    listTree(root, 0);
```

```
    return 0;
}
```

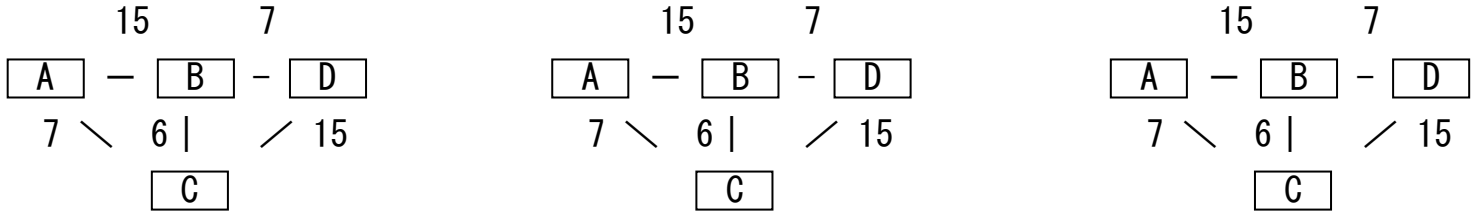
グラフ

最短経路

平面上に何かを表す端点（ノード）がいくつか存在していて、それらのうちある端点同士が線（エッジ）で連結されている状態を考えます。この関係では、ある端点からある端点まで、線を通り別の端点を経由して到達するような「つながり方」に着目する概念をグラフと言います。

ここでは、端点から別の端点までの方向を考えない無向グラフの問題を解くことにします。

以下の図 1 で、A から D までの最短経路を考えます。各ノード間のエッジは経路を表し、数値は距離を表します。



C → B = 6,　A → C → B = 13

A → B = 15,　A → C = 7　　C → D = 15,　A → C → D = 22

図 1　　図 2　　図 3

A を基点に、ここからの距離について考えます。まず、A に隣接するノードの距離について、それぞれの直行距離を割り当てます。A から直接移動できるノードは B と C ですので、そこまでの距離を候補として登録します。つまり、B = 15、 C = 7 となります。(図 2)

次に、A から最も近い距離のノードから探索を続けます。ここでは、A と C が最短ですので C に確定します。具体的には、C の「直前のノード」として、 A が登録されます。最終的には、この「直前のノード」を逆順に辿ることで、最短経路が得られます。

C には、A からの距離が登録されます。次のステップでは、その距離と次のノードまでの距離を加えることで、起点A からの距離が確定されます。C からは、C → A、C → B、C → D という経路がありますが、A は（起点でもありますが）既に確定済みのノードなので、検索対象から除外します。この除外判定を忘れると、無限ループとなりますので注意して下さい。C → B = 6 で、A → B = 13 になり、B に登録されている距離 15 より短いことになります。従って、B への「最短距離」が更新され、B の「直前のノード」が C に更新されます。D には、距離 22 と、「直前のノード」として C が登録されます。(図 3)

これで、目的地 D に到達する経路が一つ発見されましたが、これが最短経路かどうかはまだ分かりませんので、目的地ノードに到達できる他の経路の探索を継続する必要があります。ただし、 D は中継地点ではないので次の探索候補地からは除外されます。

従って、残る B と D のうち、次の探索出発地点は B ということになります。B からは、A、C は確定地点のため、B → D の距離計算が行われ、A → D の距離が、20 となり、D の距離が 22 に更新され、直前の経路のノードが B に確定します。

最短経路は、D から逆順に見ていくことになります。直前の経路のノードをたどると、D → B → C → A となりますので、A から出発すると、A, C, B, D という経路が最短であると分かります。

ノード間の距離計算は、2 次元配列で表すと便利です。2 点 i, j の距離を行列の成分 a_{ij} で扱うことになります。

$$a[n][n] = \begin{vmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \cdots & & & & \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{vmatrix}$$

そうすると、先ほどの A、B、C、D の距離データは、次のように表されます。

$$\begin{vmatrix} 0 & 15 & 7 & 0 \\ 15 & 0 & 6 & 7 \\ 7 & 6 & 0 & 15 \\ 0 & 7 & 15 & 0 \end{vmatrix}$$

対角要素に現れる 0 は同一点を、それ以外の箇所に現れる 0 は到達できないことを表していることに注意して下さい。

演習課題 5-3

ノード間の距離データから、最短経路を求めるプログラムを完成させて下さい。

- ファイル名を `kadai_5_3.c` として下さい。

下にある演習プログラムは、ところどころコードを埋める箇所があります。空欄部分は、

```
```

のように、枠で示されています。ただし、枠の大きさはコードの量とは関係がありません。

```
/*
 * 無向グラフ 演習プログラム
 */

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

```
/* 最大ノード数 */
#define MAXORDER 256


/* グラフを表す構造体 */

typedef struct {
    int ord;                        /* ノード数 */
    int adj[MAXORDER][MAXORDER];    /* 隣接行列 */
} Graph;


/* リストを表す構造体 */
typedef struct {
    int elem[MAXORDER];
} List;


/* 経路情報を表す構造体 */
typedef struct {
    int length;
    List path_list;
} PathData;
```

```
/*
 *  関数プロトタイプ
 */

/* 長さ n、全てのメンバー v のリストを返す */
List initializeList(int val);

/* startNode から goalNode への最短経路を求める */
PathData calcDijkstra(Graph *grp, int start_node, int goal_node);

/* グラフの隣接行列を画面表示する */
void printGraph(Graph *grp);

/* 経路情報リスト（逆順）から、経路を表示する */
void printPath(PathData path, int start_node, int goal_node);



/*
 *  関数定義
 */


/* 長さ n、全てのメンバー v のリストを返す */
```

```
List initializeList(int val)
{
    List t;
    int i;
    for (i = 0; i < MAXORDER; i++)
        t.elem[i] = val;
    return t;
}


/* start_node から goal_node への最短経路を求める */
PathData calcDijkstra(Graph *grp, int start_node, int goal_node)
{
    List temp_list;      /* 作業用距離リスト */
    List dist_list;      /* 確定距離リスト */
    List link_list;      /* 探索ノードに隣接するノードのリスト */
    PathData path;      /* 最短経路情報を保持するリスト */
    int cur_node = start_node;    /* カレント探索ノード */
    int next_node;

    temp_list = initializeList(INT_MAX);
    path.path_list = initializeList(-1);
    dist_list = initializeList(-1);
    dist_list.elem[start_node] = 0;
```

```
do {
    int i;
    int distance;
    int index = 0;
    /*
     * 探索ノードに隣接するノードのリストを作成する
     */
    for (i = 0; i < grp->ord; i++)
        /* 距離未確定地点に到達可能であれば、次候補としてリストアップする */
        if (dist_list.elem[i] == -1 && grp->adj[cur_node][i] > 0)
            link_list.elem[index++] = i;

    /*
     * 先頭ノードから次候補先までの距離を仮計算する （作業用距離リスト）
     */
    for (i = 0; i < index; i++) {
        int next = link_list.elem[i];
        /* 距離計算 （確定部分＋リンク距離） */
        distance = [                                ]
        if (distance < temp_list.elem[next]) {
            /* より短い距離の経路が見つかった場合 */
            [                                      ] = distance;
            /* 隣接ノードでの最短経路情報を登録する */
            path.path_list.elem[next] = cur_node;
        }
```

```
    }
    /*
     * 次探索ノード候補： 現ノードから最短距離のものを探す
     */
    next_node = -1;
    distance = INT_MAX;
    for (i = 0; i < grp->ord; i++)
        if (dist_list.elem[i] == -1)
            if (temp_list.elem[i] < distance) {
                /* より短い距離の次探索ノード候補 */
                distance = [                              ]
                next_node = i;
            }
    if (next_node >= 0) {
        /* 現ノードから最短距離のノード確定、next_node を次の探索ノードに */
        dist_list.elem[[                ]] = temp_list.elem[next_node];
        cur_node = next_node;
    }
} while (next_node >= 0 && cur_node != goal_node);

if (cur_node == goal_node) {
    path.length = dist_list.elem[goal_node];
    return [                              ]
}
```

```
    else {
        /* ゴールまでの経路が見つからなかった場合 */
        path.length = -1;
        return [                                ]
    }
}



/*************************************************************/


/* 探索するグラフのデータ */
Graph graphdata = {10, {
    /*A  B  C  D  E  F  G  H  I  J       */
    { 0, 2, 2, 3, 3, 0, 0, 0, 0, 0 }, /*A*/
    { 2, 0, 2, 0, 0, 0, 0, 0, 4, 0 }, /*B*/
    { 2, 2, 0, 1, 0, 2, 0, 0, 0, 0 }, /*C*/
    { 3, 0, 1, 0, 0, 0, 1, 0, 0, 0 }, /*D*/
    { 3, 0, 0, 0, 0, 0, 1, 0, 0, 0 }, /*E*/
    { 0, 0, 2, 0, 0, 0, 0, 1, 1, 0 }, /*F*/
    { 0, 0, 0, 1, 1, 0, 0, 2, 0, 0 }, /*G*/
    { 0, 0, 0, 0, 0, 1, 2, 0, 0, 2 }, /*H*/
    { 0, 4, 0, 0, 0, 1, 0, 0, 0, 2 }, /*I*/
    { 0, 0, 0, 0, 0, 0, 0, 2, 2, 0 }  /*J*/
}};
```

```
/* グラフの隣接行列を画面表示する */
void printGraph(Graph *grp)
{
    int i, j;
    for (i = 0; i < grp->ord; i++) {
        for (j = 0; j < grp->ord; j++)
            printf("%2d", grp->adj[i][j]);
        printf("¥n");
    }
}

/* 経路情報リスト（逆順）から、経路を表示する */
void printPath(PathData path, int start_node, int goal_node)
{
    int i;
    int idx = 0;
    int pos = goal_node;
    List track_path;    /* 逆順経路 */

    /* 距離を表示 */
    printf("Length: %d¥n", path.length);
```

```
    /* リストに、ゴール地点から逆順で地点データを追加する */
    track_path.elem[idx++] = [          ]
    do {
        /* ゴール地点から、最短隣接経路を辿っていく */
        pos = path.path_list.elem[          ];
        track_path.elem[idx++] = [          ]
    } while (pos != -1 && pos != [          ]);

    printf("Path: ");
    for (i = idx - 1; i >= 0; i--) {
        /* リストを末尾から先頭に向かって表示する */
        printf("%d", track_path.elem[          ]);
        if (i > 0)
            printf(" -> ");
    }
    printf("¥n");
}

/****************************************/


int main()
{
    int start_node = 0;  /* 出発点 */
    int goal_node = 9;   /* 目的地 */
```

```
    /* 最短経路を求める */
    PathData path = calcDijkstra(&graphdata, start_node, goal_node);

    printf("¥nGraph :¥n");
    printGraph(&graphdata);

    if (path.length >= 0) {
        /* 最短経路が正しく計算された */
        printPath(path, start_node, goal_node);
    }
    else {
        printf("Unreachable¥n");
    }

    return 0;
}
```

演習問題索引

演習課題 5-1　演習課題 5-2　演習課題 5-3