データ構造とアルゴリズムII AVL木レポート

報告者: 4I 35番 弘中悠月

提出日: 2021/06/21(月)

設計

木の構造

c : avl_tree.h

```c
typedef struct AVL_tree_node
{
    int value;
    struct AVL_tree_node *left;
    struct AVL_tree_node *right;
} avl_node;
```

構造体を用いたリスト構造でAVL木を実装する。`int value`はデータの値を格納、`struct AVL_tree_node *left, struct AVL_tree_node *right`はそれぞれ左と右へのポインタを保持し、これらを連結することで木構造を表現する。

実装の方針

まず単純な二分探索木の追加機能を実装し、木構造が正しく機能することを確認する。その後追加機能の改造と探索、削除機能を実装する。

プログラムファイルの分割は`AVL_tree.h`に木構造の定義、`AVL_tree.c`にAVL木の機能の定義、`main.c`に今回の課題で行う操作を記述する。

ただし、今回はコンパイル時の作業簡略化のため、`DSaA_AVL.c`の一つのファイルに記述した。本レポートではこのプログラムについて記述していく。

二分木 追加機能/木構造の確認

まず、単純な二分探索木の追加機能のみを実装し、木構造が正しく動作しているかを確かめた。以下のコードの`malloc_node, add_node`が追加機能に当たる関数である。木の表示には`printTree`という関数を定義し使用しているが、これは後のAVL木の実装段階で修正したため解説は省略する。

c : binary_tree.c

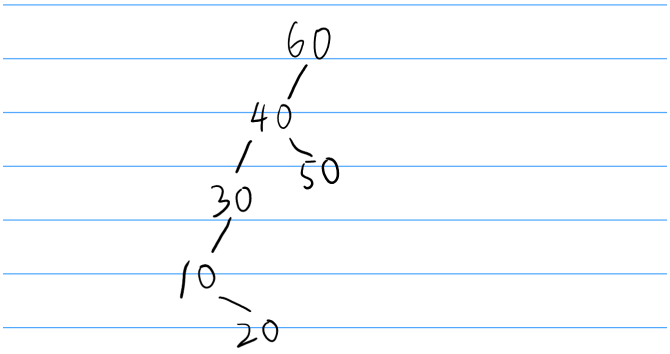
```c
#include <stdio.h>
#include <stdlib.h>
#include "AVL_tree.h"

//新しいノードを作成
struct ALV_tree_node *malloc_node(int value)
{
```

```
    struct ALV_tree_node *new_node = (struct ALV_tree_node *)malloc(sizeof(struct ALV_tree_node));

    new_node->value = value;
    new_node->left = NULL;
    new_node->right = NULL;

    return new_node;
}

//新しいノードを追加
struct ALV_tree_node *add_node(struct ALV_tree_node *root, int value)
{
    struct ALV_tree_node *tmp_node;

    if (root == NULL)
    {
        root = malloc_node(value);
        return root;
    }

    tmp_node = root;
    while (1)
    {
        if (value < tmp_node->value)
        {
            if (tmp_node->left == NULL)
            {
                tmp_node->left = malloc_node(value);
                break;
            }
            tmp_node = tmp_node->left;
        }
        else if (value > tmp_node->value)
        {
            if (tmp_node->right == NULL)
            {
                tmp_node->right = malloc_node(value);
                break;
            }
            tmp_node = tmp_node->right;
        }
        else
        {
            printf("既に存在する値です\n");
            break;
        }
    }
    return root;
}

void printTree(struct ALV_tree_node *root, int depth)
{
    int i;

    if (root == NULL)
    {
        return;
```

```
    }

    /* 右の子孫ノードを表示 */
    printTree(root->right, depth + 1);

    /* 深さをスペースで表現 */
    for (i = 0; i < depth; i++)
    {
        printf("---/", i);
    }

    /* ノードのデータを表示 */
    printf("+%3d\n", root->value);

    /* 左の子孫ノードを表示 */
    printTree(root->left, depth + 1);

    depth++;
}

struct ALV_tree_node *malloc_node(int);
struct ALV_tree_node *add_node(struct ALV_tree_node *, int);

int main()
{
    struct ALV_tree_node *root = NULL;
    root = add_node(root, 60);
    root = add_node(root, 40);
    root = add_node(root, 30);
    root = add_node(root, 10);
    root = add_node(root, 50);
    root = add_node(root, 20);

    printTree(root, 0);
}
```

以下の出力がこのコードの実行結果である。

```
+ 60
---/---/+ 50
---/+ 40
---/---/+ 30
---/---/---/---/+ 20
---/---/---/+ 10
```



図で表すと上図のような出力になっていることがわかり、二分木の構造については問題ないことがわかった。

AVL木 回転/追加

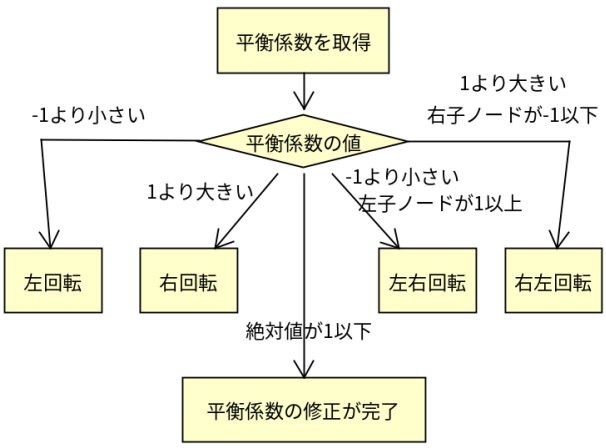
AVL木においては平衡係数の絶対値が1以下である必要があるため、二分木の実装に加えて平衡係数の修正を行う**回転**の動作が必要である。

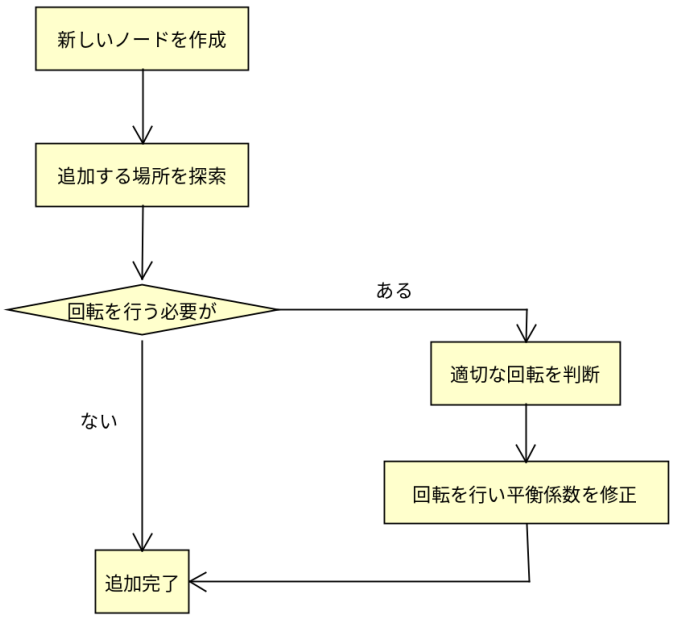
回転動作は4種類実装する必要があり、

1. 平衡係数が-1より小さい場合は部分木を左に回転させる
2. 平衡係数が1より大きい場合は部分木を右に回転させる
3. 平衡係数が-1より小さくてかつ、左の子ノードの平衡係数が1以上の場合は、左の子ノードを根として左回転、元のノードを根として右回転させる
4. 平衡係数が1より大きくてかつ、右の子ノードの平衡係数が-1以下の場合は、右の子ノードを根として右回転、元のノードを根として左回転させる

上記4つの回転が必要である。

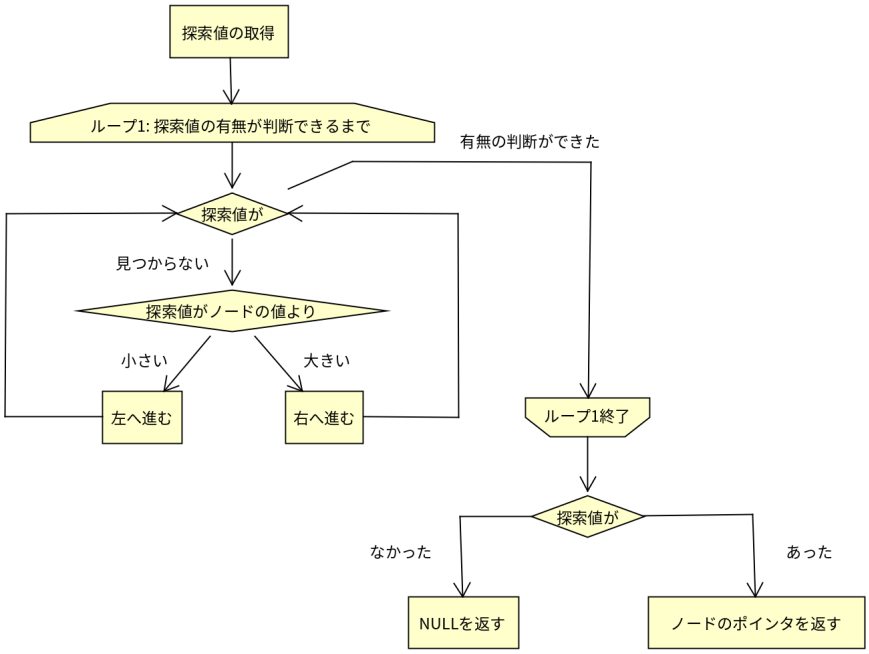
以下に示す図は回転(1枚目)、追加(2枚目)機能のフローチャートである。





AVL木 探索

探索な二分木と同様であり、探索値がノードの値より大きければ右へ、小さければ左へ進めば良い。
以下の図は探索機能のフローチャートである。

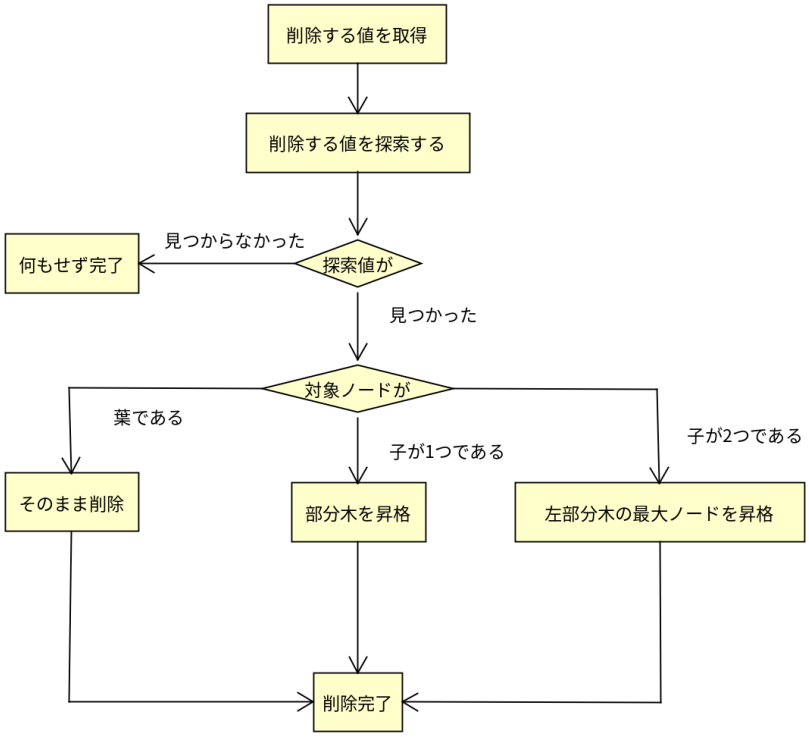


AVL木 削除

削除は3つの場合に分けて実装する。具体的には、

1. 葉の場合は、そのまま削除する
2. 子を1つ持つ場合、部分木を昇格させる
3. 子を2つ持つ場合、左部分木の最大ノードを昇格させる

上記の3つの機能が必要である。以下の図は削除機能のフローチャートである。



実装

コード

前述の通り、プログラムは`DSaA_AVL.c`に結合して作成した。以下のコードがその内容である。

c : dsaa_avl.c

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

#define MAX_HEIGHT 100

#define TREE_LEFT 1
#define TREE_RIGHT 2

typedef struct AVL_tree_node
{
    int value;
    struct AVL_tree_node *right;
    struct AVL_tree_node *left;
} avl_node;

int get_height(avl_node *node)
{
    if (node == NULL)
    {
        return 0;
    }

    int left_height = get_height(node->left);
```

```
    int right_height = get_height(node->right);

    if (left_height > right_height)
    {
        return left_height + 1;
    }
    else
    {
        return right_height + 1;
    }
}

avl_node *left_rotate(avl_node *root, avl_node *node, avl_node *parent, int direction)
{
    avl_node *pivot;
    avl_node *new_root;

    printf("left_rotate:%d\n", node->value);

    pivot = node->right;

    if (pivot != NULL)
    {
        node->right = pivot->left;
        pivot->left = node;
    }

    if (parent == NULL)
    {
        new_root = pivot;
        return new_root;
    }

    if (direction == TREE_LEFT)
    {
        parent->left = pivot;
    }
    else
    {
        parent->right = pivot;
    }
    return root;
}

avl_node *right_rotate(avl_node *root, avl_node *node, avl_node *parent, int direction)
{
    avl_node *pivot;
    avl_node *new_root;

    printf("right_rotate:%d\n", node->value);

    pivot = node->left;

    if (pivot != NULL)
    {
        node->left = pivot->right;
```

```
81          pivot->right = node;
82      }
83
84      if (parent == NULL)
85      {
86          new_root = pivot;
87          return new_root;
88      }
89
90      if (direction == TREE_LEFT)
91      {
92          parent->left = pivot;
93      }
94      else
95      {
96          parent->right = pivot;
97      }
98      return root;
99  }
100
101 avl_node *rightleft_rotate(avl_node *root, avl_node *node, avl_node
    *parent, int direction)
102 {
103     avl_node *new_root;
104     printf("right_left_rotate:%d\n", node->value);
105
106     new_root = right_rotate(root, node->right, node, TREE_RIGHT);
107
108     return left_rotate(new_root, node, parent, direction);
109 }
110
111 avl_node *leftright_rotate(avl_node *root, avl_node *node, avl_node
    *parent, int direction)
112 {
113     avl_node *new_root;
114
115     printf("left_right_rotate:%d\n", node->value);
116
117     new_root = left_rotate(root, node->left, node, TREE_LEFT);
118
119     return right_rotate(new_root, node, parent, direction);
120 }
121
122 avl_node *balancing(avl_node *root, avl_node *node, avl_node *parent, int
    direction, int *branch, int num_branch)
123 {
124     avl_node *next;
125     avl_node *new_root;
126
127     int left_height, right_height;
128     int balance;
129
130     if (node == NULL || root == NULL)
131     {
132         return root;
133     }
134
135     if (num_branch > 0)
```

```
    {
        if (branch[0] == TREE_LEFT)
        {
            next = node->left;
        }
        else
        {
            next = node->right;
        }
        new_root = balancing(root, next, node, branch[0], &branch[1],
num_branch - 1);
    }

    left_height = get_height(node->left);
    right_height = get_height(node->right);
    balance = right_height - left_height;

    if (balance > 1)
    {
        if (get_height(node->right->left) > get_height(node->right->right))
        {
            return rightleft_rotate(new_root, node, parent, direction);
        }
        else
        {
            return left_rotate(new_root, node, parent, direction);
        }
    }
    else if (balance < -1)
    {
        if (get_height(node->left->right) > get_height(node->left->left))
        {
            return leftright_rotate(new_root, node, parent, direction);
        }
        else
        {
            return right_rotate(new_root, node, parent, direction);
        }
    }

    return root;
}

avl_node *malloc_node(int value)
{
    avl_node *new_node;

    new_node = (avl_node *)malloc(sizeof(avl_node));
    if (new_node == NULL)
    {
        return NULL;
    }

    new_node->value = value;
    new_node->left = NULL;
    new_node->right = NULL;

    return new_node;
```

```
}

avl_node *add_node(avl_node *root, int value)
{
    avl_node *node;
    int branch[MAX_HEIGHT] = {0};
    int num_branch = 0;

    if (root == NULL)
    {
        root = malloc_node(value);
        if (root == NULL)
        {
            printf("malloc error\n");
            return NULL;
        }
        return root;
    }

    node = root;
    while (1)
    {
        if (value < node->value)
        {
            if (node->left == NULL)
            {
                node->left = malloc_node(value);
                break;
            }
            branch[num_branch] = TREE_LEFT;
            num_branch++;
            node = node->left;
        }
        else if (value > node->value)
        {
            if (node->right == NULL)
            {
                node->right = malloc_node(value);
                break;
            }
            branch[num_branch] = TREE_RIGHT;
            num_branch++;
            node = node->right;
        }
        else
        {
            printf("%dは既に存在します\n", value);
            break;
        }
    }

    return balancing(root, root, NULL, 0, branch, num_branch);
}

avl_node *search_node(avl_node *root, int value)
{
    avl_node *node;
    node = root;
```

```
    while (node)
    {
        if (value < node->value)
        {
            node = node->left;
        }
        else if (value > node->value)
        {
            node = node->right;
        }
        else
        {
            return node;
        }
    }

    return NULL;
}

avl_node *delete_not_have_child_node(avl_node *root, avl_node *node,
avl_node *parent)
{
    if (parent != NULL)
    {
        if (parent->left == node)
        {
            parent->left = NULL;
        }
        else
        {
            parent->right = NULL;
        }
        free(node);
    }
    else
    {
        free(node);
        root = NULL;
    }
    return root;
}

avl_node *delete_have_one_child_node(avl_node *root, avl_node *node,
avl_node *child)
{
    node->value = child->value;
    node->left = child->left;
    node->right = child->right;

    free(child);

    return root;
}

avl_node *delete_have_two_child_node(avl_node *root, avl_node *node, int
*branch, int *num_branch)
{
    avl_node *max;
```

```
306     avl_node *maxParent;
307 
308     max = node->left;
309     maxParent = node;
310 
311     branch[*num_branch] = TREE_LEFT;
312     (*num_branch)++;
313 
314     while (max->right != NULL)
315     {
316         maxParent = max;
317         max = max->right;
318 
319         branch[*num_branch] = TREE_RIGHT;
320         (*num_branch)++;
321     }
322     printf("max value is %d\n", max->value);
323 
324     node->value = max->value;
325 
326     if (max->left == NULL)
327     {
328         root = delete_not_have_child_node(root, max, maxParent);
329     }
330     else
331     {
332         root = delete_have_one_child_node(root, max, max->left);
333     }
334 
335     return root;
336 }
337 
338 avl_node *delete_node(avl_node *root, int value)
339 {
340     avl_node *node;
341     avl_node *parent;
342     int branch[MAX_HEIGHT] = {0};
343     int num_branch = 0;
344 
345     if (root == NULL)
346     {
347         return NULL;
348     }
349 
350     node = root;
351     parent = NULL;
352 
353     while (node != NULL)
354     {
355         if (value < node->value)
356         {
357             parent = node;
358             node = node->left;
359 
360             branch[num_branch] = TREE_LEFT;
361             num_branch++;
362         }
363         else if (value > node->value)
```

```
        {
            parent = node;
            node = node->right;

            branch[num_branch] = TREE_RIGHT;
            num_branch++;
        }
        else
        {
            break;
        }
    }

    if (node == NULL)
    {
        printf("%dを持つノードが存在しません\n", value);
        return root;
    }

    printf("Delete %d node\n", node->value);

    if (node->left == NULL && node->right == NULL)
    {
        root = delete_not_have_child_node(root, node, parent);
    }
    else if ((node->left != NULL && node->right == NULL) || (node->right != NULL && node->left == NULL))
    {
        if (node->left != NULL)
        {
            root = delete_have_one_child_node(root, node, node->left);
        }
        else
        {
            root = delete_have_one_child_node(root, node, node->right);
        }
    }
    else
    {
        root = delete_have_two_child_node(root, node, branch, &num_branch);
    }

    return balancing(root, root, NULL, 0, branch, num_branch);
}

void print_t(avl_node *root, int h)
{
    if (root != NULL)
    {
        print_t(root->right, h + 1);
        for (int i = 0; i < h; i++)
        {
            printf("|---|");
        }
        printf("%d\n", root->value);
        print_t(root->left, h + 1);
    }
}
```

```
421
422 void deleteTree(avl_node *root)
423 {
424     if (root == NULL)
425     {
426         return;
427     }
428
429     if (root->left != NULL)
430     {
431         deleteTree(root->left);
432     }
433     if (root->right != NULL)
434     {
435         deleteTree(root->right);
436     }
437
438     free(root);
439 }
440
441 int main(void)
442 {
443     avl_node *root = NULL;
444
445     // 1. 60,40,30,10,50,20の順で挿入
446     // ☒ 段階的に探索木を出力
447     printf("1. 追加\n\n");
448     int add_nums[6] = {60, 40, 30, 10, 50, 20};
449     for (int i = 0; i < 6; i++)
450     {
451         root = add_node(root, add_nums[i]);
452         print_t(root, 0);
453         printf("\n-------------\n");
454     }
455     // 2. 30と55を検索
456     printf("\n\n2. 探索\n\n");
457     int search_num[2] = {30, 55};
458     for (int i = 0; i < 2; i++)
459     {
460         avl_node *result = search_node(root, search_num[i]);
461         if (result == NULL)
462         {
463             printf("%dは見つかりませんでした\n", search_num[i]);
464         }
465         else
466         {
467             printf("%dを発見しました\n", search_num[i]);
468         }
469     }
470
471     // 3. 20,40,60,30,50,10の順で削除
472     // ☒ 段階的に探索木を出力
473     printf("\n\n3. 削除\n\n");
474     int delete_nums[6] = {20, 40, 60, 30, 50, 10};
475     for (int i = 0; i < 6; i++)
476     {
477         root = delete_node(root, delete_nums[i]);
478         print_t(root, 0);
```

```
        printf("\n-------------\n");
    }

    deleteTree(root);

    return 0;
}
```

実行結果

実行内容

1. 60,40,30,10,50,20の順で挿入
   段階的に探索木を出力
2. 30と55を検索
3. 20,40,60,30,50,10の順で削除
   段階的に探索木を出力

実行内容は`main関数`に記載している。以下に示す出力が`DSaA_AVL.c`の実行結果である。

```
1. 追加

60

-------------
60
|---|40

-------------
right_rotate:60
|---|60
40
|---|30

-------------
|---|60
40
|---|30
|---||---|10

-------------
|---|60
|---||---|50
40
|---|30
|---||---|10

-------------
left_right_rotate:30
left_rotate:10
right_rotate:30
|---|60
|---||---|50
40
|---||---|30
```

```
|---|20
|---||---|10

-------------


2. 探索

30を発見しました
55は見つかりませんでした


3. 削除

Delete 20 node
max value is 10
|---|60
|---||---|50
40
|---||---|30
|---|10

-------------
Delete 40 node
max value is 30
|---|60
|---||---|50
30
|---|10

-------------
Delete 60 node
|---|50
30
|---|10

-------------
Delete 30 node
max value is 10
|---|50
10

-------------
Delete 50 node
10

-------------
Delete 10 node

-------------
```

出力の`|---|`は木の高さを示しており、また、高さ0のノードを基準として下の値が左、上の値が右というように対応している。

参考文献

コードの設計・作成にあたり以下の文献を参照した。

- https://daeudaeu.com/bintree/
- https://daeudaeu.com/avl_tree/#i-3
- http://www.nct9.ne.jp/m_hiroi/linux/clang13.html

以下に参照したコードを示す。

単純二分探索木

c : https://daeudaeu.com/bintree/

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

#define MAX_NAME_LEN 256

/* 二分探索木のノードを表す構造体 */
struct node_t {
  int number;
  char name[MAX_NAME_LEN];
  struct node_t *left;
  struct node_t *right;
};


/* deleteTree:二分探索木のノード全てを削除する
   引数１ root : 根ノードのアドレス
   返却値 : なし */
void deleteTree(struct node_t *root){
  if(root == NULL){
    return;
  }

  deleteTree(root->left);
  deleteTree(root->right);

  printf("free:%d(%s)\n", root->number, root->name);
  free(root);

}

/* mallocNode:ノードの構造体のメモリを確保し、データを設定
   引数１ number : 追加する会員番号
   引数２ name : 追加する会員の名前
   返却値 : 追加したノードのアドレス */
struct node_t *mallocNode(int number, char *name){
  struct node_t *add;

  add = (struct node_t*)malloc(sizeof(struct node_t));
  if(add == NULL){
    return NULL;
  }
```

```c

  add->left = NULL;
  add->right = NULL;
  add->number = number;
  strcpy(add->name, name);

  return add;
}

/* addNode:指定されたnumberとname持つノードを追加する
   引数１ root : 根ノードのアドレス
   引数２ number : 追加する会員番号
   引数３ name : 追加する会員の名前
   返却値 : 根ルートのアドレス */
struct node_t *addNode(struct node_t *root, int number, char *name){
  struct node_t *node;

  /* まだノードが一つもない場合 */
  if(root == NULL){
    /* 根ノードとしてノードを追加 */
    root = mallocNode(number, name);
    if(root == NULL){
      printf("malloc error\n");
      return NULL;
      }
      return root;
   }

   /* 根ノードから順に追加する場所を探索 */
   node = root;
   while(1) {
     if(number < node->number){
       /* 追加する値がノードの値よりも小さい場合 */

       if(node->left == NULL){
         /* そのノードの左の子が無い場合（もう辿るべきノードが無い場合）*/

         /* その左の子の位置にノードを追加 */
         node->left = mallocNode(number, name);

         /* 追加完了したので処理終了 */
         break;
       }

       /* 左の子がある場合は左の子を新たな注目ノードに設定 */
       node = node->left;

     } else if(number > node->number){
       /* 追加する値がノードの値よりも大きい場合 */

       if(node->right == NULL){
         /* そのノードの右の子が無い場合（もう辿るべきノードが無い場合）*/

         /* その右の子の位置にノードを追加 */
         node->right = mallocNode(number, name);

         /* 追加完了したので処理終了 */
         break;
```

```
101         }
102
103         /* 右の子がある場合は右の子を新たな注目ノードに設定 */
104         node = node->right;
105       } else {
106         /* 追加する値とノードの値が同じ場合 */
107
108         printf("%d already exist\n", number);
109         break;
110       }
111     }
112
113     return root;
114 }
115
116 /* searchNode:指定されたnumberを持つノードを探索する
117     引数１ root ： 探索を開始するノードのアドレス
118     引数２ number ： 探索する会員番号
119     返却値 ： number を持つノードのアドレス（存在しない場合は NULL）*/
120 struct node_t *searchNode(struct node_t *root, int number){
121   struct node_t *node;
122
123   node = root;
124
125   /* 探索を行うループ（注目ノードがNULLになったら終了 */
126   while(node){
127     if(number < node->number){
128       /* 探索値がノードの値よりも小さい場合 */
129
130       /* 注目ノードを左の子ノードに設定 */
131       node = node->left;
132     } else if(number > node->number){
133       /* 探索値がノードの値よりも大きい場合 */
134
135       /* 注目ノードを右の子ノードに設定 */
136       node = node->right;
137     } else {
138       /* 探索値 = ノードの値の場合 */
139       return node;
140     }
141   }
142
143   /* 探索値を持つノードが見つからなかった場合 */
144   return NULL;
145 }
146
147 /* deleteNoChildeNode:指定された子の無いノードを削除する
148     引数１ root ： 木の根ノードのアドレス
149     引数２ node ： 削除するノードのアドレス
150     引数３ parent：削除するノードの親ノードのアドレス
151     返却値 ： 根ノードのアドレス */
152 struct node_t *deleteNoChildNode(struct node_t *root, struct node_t *node,
    struct node_t *parent){
153
154   if(parent != NULL){
155     /* 親がいる場合（根ノード以外の場合）は
156     削除対象ノードを指すポインタをNULLに設定 */
157     if(parent->left ==  node){
```

```
158         /* 削除対象ノードが親ノードから見て左の子の場合 */
159          parent->left = NULL;
160       } else {
161         /* 削除対象ノードが親ノードから見て右の子の場合 */
162         parent->right = NULL;
163       }
164       free(node);
165     }  else {
166       /* 削除対象ノードが根ノードの場合 */
167       free(node);
168 
169       /* 根ノードを指すポインタをNULLに設定 */
170       root = NULL;
171     }
172 
173     return root;
174 }
175 
176 /* deleteOneChildeNode:指定された子が一つのノードを削除する
177    引数１ root : 木の根ノードのアドレス
178    引数２ node : 削除するノードのアドレス
179    引数３ child : 削除するノードの子ノードのアドレス
180    返却値 : 根ノードのアドレス */
181 struct node_t *deleteOneChildNode(struct node_t *root, struct node_t *node,
    struct node_t * child){
182 
183   /* 削除対象ノードにその子ノードのデータとポインタをコピー */
184   node->number = child->number;
185   strcpy(node->name, child->name);
186   node->left = child->left;
187   node->right = child->right;
188 
189   /* コピー元のノードを削除 */
190   free(child);
191 
192   return root;
193 }
194 
195 /* deleteTwoChildeNode:指定された子が二つのノードを削除する
196    引数１ root : 木の根ノードのアドレス
197    引数２ node : 削除するノードのアドレス
198    返却値 : 根ノードのアドレス */
199 struct node_t *deleteTwoChildNode(struct node_t *root, struct node_t *node)
    {
200 
201   struct node_t *max;
202   struct node_t *maxParent;
203 
204   /* 左の子から一番大きい値を持つノードを探索 */
205   max = node->left;
206   maxParent = node;
207 
208   while(max->right != NULL){
209     maxParent = max;
210     max = max->right;
211   }
212   printf("max number is %d\n", max->number);
213 
```

```
  /* 最大ノードのデータのみ削除対象ノードにコピー */
  node->number = max->number;
  strcpy(node->name, max->name);

  /* 最大ノードを削除 */

  /* maxは最大ノードなので必ずmax->rightはNULLになる */
  if(max->left == NULL){
    /* 最大ノードに子がいない場合 */
    root = deleteNoChildNode(root, max, maxParent);
    } else {
      /* 最大ノードに子供が一つついる場合 */
      root = deleteOneChildNode(root, max, max->left);
    }

    return root;
}



/* deleteNode:指定されたnumberを持つノードを削除する
   引数１ root : 木の根ノードのアドレス
   引数２ number : 削除する会員番号
   返却値 : 根ノードのアドレス */
struct node_t *deleteNode(struct node_t *root, int number){
  struct node_t *node;
  struct node_t *parent;

  if(root == NULL){
    return NULL;
  }

  /* 削除対象ノードを指すノードを探索 */
  node = root;
  parent = NULL;

  while(node !=  NULL){
    if(number < node->number){
      parent = node;
      node = node->left;
    } else if(number > node->number){
      parent = node;
      node = node->right;
    } else {
      break;
    }
  }

  /* 指定されたnumberを値として持つノードが存在しない場合は何もせず終了 */
  if(node == NULL){
    printf("%d を持つノードが存在しません\n", number);
    return root;
  }

  printf("Delete %d(%s) node\n", node->number, node->name);

  if(node->left == NULL && node->right == NULL){
    /* 子がいないノードの削除 */
```

```
272       root = deleteNoChildNode(root, node, parent);
273     } else if((node->left != NULL && node->right == NULL) ||
274       (node->right != NULL && node->left == NULL)){
275       /* 子が一つしかない場合 */
276 
277       if(node->left != NULL){
278         root = deleteOneChildNode(root, node, node->left);
279       } else {
280         root = deleteOneChildNode(root, node, node->right);
281       }
282     } else {
283       /* 左の子と右の子両方がいるノードの削除 */
284       root = deleteTwoChildNode(root, node);
285     }
286 
287     return root;
288 }
289 
290 /* printTree:rootを根ノードとする二分探索木をの全ノードを表示する
291    引数１ root : 木の根ノードのアドレス
292    引数２ depth: 関数呼び出しの深さ
293    返却値 : なし */
294 void printTree(struct node_t *root, int depth){
295     int i;
296 
297     if(root == NULL){
298       return ;
299     }
300 
301     /* 右の子孫ノードを表示 */
302     printTree(root->right, depth+1);
303 
304     /* 深さをスペースで表現 */
305     for(i = 0; i < depth; i++){
306       printf("  ");
307     }
308 
309     /* ノードのデータを表示 */
310     printf("+%3d(%s)\n", root->number, root->name);
311 
312     /* 左の子孫ノードを表示 */
313     printTree(root->left, depth+1);
314 
315     depth++;
316 }
317 
318 int main(void){
319     struct node_t *root, *node;
320     int input;
321     int number;
322     char name[MAX_NAME_LEN];
323     int loop;
324 
325     /* まだ木がないのでrootをNULLにセット */
326     root = NULL;
327 
328     /* 最初にてきとうにノードを追加しておく */
329     root = addNode(root, 100, "100");
```

```
  root = addNode(root, 200, "200");
  root = addNode(root, 300, "300");
  root = addNode(root, 50, "50");
  root = addNode(root, 150, "150");
  root = addNode(root, 250, "250");
  root = addNode(root, 10, "1");
  root = addNode(root, 125, "125");
  root = addNode(root, 5, "5");
  root = addNode(root, 25, "25");
  root = addNode(root, 500, "500");
  root = addNode(root, 175, "175");

  loop = 1;
  while(loop){
    printf("処理を選択(1:add, 2:delete, 3:search, 4:exit)");
    scanf("%d", &input);

    switch(input){
    case 1:
      printf("会員番号(1 - 999):");
      scanf("%d", &number);
      if(number < 1 || number > 999){
        printf("値が範囲外です\n");
        continue;
      }

      printf("名前:");
      scanf("%s", name);

      root = addNode(root, number, name);
      break;
    case 2:
      printf("会員番号(1 - 999):");
      scanf("%d", &number);
      if(number < 1 || number > 999){
        printf("値が範囲外です\n");
        continue;
      }

      root = deleteNode(root, number);

      break;
    case 3:
      printf("会員番号(1 - 999):");
      scanf("%d", &number);
      if(number < 1 || number > 999){
        printf("値が範囲外です\n");
        continue;
      }

      node = searchNode(root, number);
      if(node == NULL){
        printf("number %d is not found\n", number);
      } else {
        printf("number %d : %s\n", number, node->name);
      }
      break;
    default:
```

```
      loop = 0;
      break;
    }
    printTree(root, 0);
  }

  deleteTree(root);

  return 0;
}
```

AVL木

c : https://daeudaeu.com/avl_tree/#i-3

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

#define MAX_NAME_LEN 256
#define MAX_HEIGHT 100

#define TREE_LEFT 1
#define TREE_RIGHT 2

/* 二分探索木のノードを表す構造体 */
struct node_t {
  int number;
  char name[MAX_NAME_LEN];
  struct node_t *left;
  struct node_t *right;
};

/* getHeight:二分探索木のノード全てを削除する
   引数1 node : 根ノードのアドレス
   返却値 : nodeを根とした木の高さ */
int getHeight(struct node_t *node) {

  int left_height;
  int right_height;
  int tree_height;

  if (node == NULL) {
    /* nodeが無いなら高さは0 */
    return 0;
  }

  /* 左右の子を根とした木の高さを取得 */
  left_height = getHeight(node->left);
  right_height = getHeight(node->right);

  /* 大きい方に+1したものを木の高さとして返却 */
  if (left_height > right_height) {
    tree_height = left_height;
  } else {
    tree_height = right_height;
```

```
  }

  return tree_height + 1;
}

/* leftRotate:nodeを根とする部分木を回転（左）
   引数１ root : 根のノードを指すアドレス
   引数２ node : 回転する部分木の根ノードを指すアドレス
   引数３ parent : nodeの親ノードを指すアドレス
   引数４ direction : parentから見たnodeのある方向
   返却値 : 根のノードを指すアドレス */
struct node_t *leftRotate(struct node_t *root, struct node_t *node, struct
node_t *parent, int direction) {
  /* nodeを根として左回転を行う */

  struct node_t *pivot;
  struct node_t *new_root;

  printf("left_rotate:%d\n", node->number);

  /* 新しい根とするノードをpivotとして設定 */
  pivot = node->right;

  /* 左回転 */
  if (pivot != NULL) {
    node->right = pivot->left;
    pivot->left = node;
  }

  /* parentもしくはrootに新しい根ノードを参照させる */
  if (parent == NULL) {
    new_root = pivot;
    return new_root;
  }

  /* どちらの子に設定するかはdirectionから判断 */
  if (direction == TREE_LEFT) {
    parent->left = pivot;
  } else {
    parent->right = pivot;
  }
  return root;
}

/* rightRotate:nodeを根とする部分木を回転（右）
   引数１ root : 根のノードを指すアドレス
   引数２ node : 回転する部分木の根ノードを指すアドレス
   引数３ parent : nodeの親ノードを指すアドレス
   引数４ direction : parentから見たnodeのある方向
   返却値 : 根のノードを指すアドレス */
struct node_t * rightRotate(struct node_t *root, struct node_t *node,
struct node_t *parent, int direction) {

  struct node_t *pivot;
  struct node_t *new_root;

  printf("right_rotate:%d\n", node->number);

```

```c
  /* 新しい根とするノードをpivotとして設定 */
  pivot = node->left;

  /* 右回転 */
  if (pivot != NULL) {
    node->left = pivot->right;
    pivot->right = node;
  }

  /* parentもしくはrootに新しい根ノードを参照させる */
  if (parent == NULL) {
    new_root = pivot;
    return new_root;
  }

  /* どちらの子に設定するかはdirectionから判断 */
  if (direction == TREE_LEFT) {
    parent->left = pivot;
  } else {
    parent->right = pivot;
  }

  return root;
}

/* leftRightRotate:nodeを根とする部分木を二重回転（右->左）
   引数１ root : 根のノードを指すアドレス
   引数２ node : 回転する部分木の根ノードを指すアドレス
   引数３ parent : nodeの親ノードを指すアドレス
   引数４ direction : parentから見たnodeのある方向
   返却値 : 根のノードを指すアドレス */
struct node_t *rightLeftRotate(struct node_t *root, struct node_t *node,
struct node_t *parent, int direction) {
  /* ２重回転（Right Left Case）を行う */

  struct node_t *new_root;
  printf("right_left_rotate:%d\n", node->number);

  /* nodeの右の子ノードを根として右回転 */
  new_root = rightRotate(root, node->right, node, TREE_RIGHT);

  /* nodeを根として左回転 */
  return leftRotate(new_root, node, parent, direction);
}

/* leftRightRotate:nodeを根する部分木を二重回転（左->右）
   引数１ root : 根のノードを指すアドレス
   引数２ node : 回転する部分木の根ノードを指すアドレス
   引数３ parent : nodeの親ノードを指すアドレス
   引数４ direction : parentから見たnodeのある方向
   返却値 : 根のノードを指すアドレス */
struct node_t * leftRightRotate(struct node_t *root, struct node_t *node,
struct node_t *parent, int direction) {
  /* ２重回転（Left Right Case）を行う */

  struct node_t *new_root;

  printf("left_right_rotate:%d\n", node->number);
```

```
154
155   /* nodeの左の子ノードを根として左回転 */
156   new_root = leftRotate(root, node->left, node, TREE_LEFT);
157
158   /* nodeを根として右回転 */
159   return rightRotate(new_root, node, parent, direction);
160 }
161
162 /* balancing:nodeからbranchで辿ったノードを平衡にする
163    引数１ root : 根のノードを指すアドレス
164    引数２ node : 平衡にするノードを指すアドレス
165    引数３ parent : nodeの親ノードを指すアドレス
166    引数４ direction : parentから見たnodeのある方向
167    引数５ branch : 平衡化を行うノードへの経路
168    引数６ num_branch : branchに格納された経路の数
169    返却値 : 根のノードを指すアドレス */
170 struct node_t * balancing(struct node_t *root, struct node_t *node, struct
    node_t *parent, int direction, int *branch, int num_branch) {
171
172   struct node_t *next;
173   struct node_t *new_root;
174
175   int left_height, right_height;
176   int balance;
177
178   if (node == NULL || root == NULL) {
179     return root;
180   }
181
182   if (num_branch > 0) {
183     /* 辿れる場合はまず目的のノードまで辿る */
184
185     /* 辿る子ノードを設定 */
186     if (branch[0] == TREE_LEFT) {
187       next = node->left;
188     } else {
189       next = node->right;
190     }
191
192     /* 子ノードを辿る */
193     new_root = balancing(root, next, node, branch[0], &branch[1],
    num_branch - 1);
194   }
195
196   /* 平衡係数を計算 */
197   left_height = getHeight(node->left);
198   right_height = getHeight(node->right);
199   balance = right_height - left_height;
200
201   if (balance > 1) {
202     /* 右の部分木が高くて並行状態でない場合 */
203
204     /* ２重回転が必要かどうかを判断 */
205     if (getHeight(node->right->left) > getHeight(node->right->right)) {
206       /* ２重回転（Right Left Case） */
207       return rightLeftRotate(new_root, node, parent, direction);
208
209     } else {
```

```
      /*１重回転（左回転）*/
      return leftRotate(new_root, node, parent, direction);
    }

  } else if (balance < -1) {
    /* 左の部分木が高くて並行状態でない場合 */

    /* ２重回転が必要かどうかを判断 */
    if (getHeight(node->left->right) > getHeight(node->left->left)) {
      /* ２重回転（Left Right Case）*/
      return leftRightRotate(new_root, node, parent, direction);
    } else {
      /* １重回転（右回転）*/
      return rightRotate(new_root, node, parent, direction);
    }
  }

  return root;
}

/* deleteTree:二分探索木のノード全てを削除する
   引数１ root : 根ノードのアドレス
   返却値 : なし */
void deleteTree(struct node_t *root){
  if(root == NULL){
    return;
  }

  if(root->left != NULL){
    deleteTree(root->left);
  }
  if(root->right != NULL){
    deleteTree(root->right);
  }

  printf("free:%d(%s)\n", root->number, root->name);
  free(root);

}

/* mallocNode:ノードの構造体のメモリを確保し、データを設定
   引数１ number : 追加する会員番号
   引数２ name : 追加する会員の名前
   返却値 : 追加したノードのアドレス */
struct node_t *mallocNode(int number, char *name){
  struct node_t *add;

  add = (struct node_t*)malloc(sizeof(struct node_t));
  if(add == NULL){
    return NULL;
  }

  add->left = NULL;
  add->right = NULL;
  add->number = number;
  strcpy(add->name, name);

  return add;
```

```c
}

/* addNode:指定されたnumberとname持つノードを追加する
   引数１ root ： 根ノードのアドレス
   引数２ number ： 追加する会員番号
   引数３ name ： 追加する会員の名前
   返却値 ： 根ルートのアドレス */
struct node_t *addNode(struct node_t *root, int number, char *name){
  struct node_t *node;
  int branch[MAX_HEIGHT] = {0};
  int num_branch = 0;

  /* まだノードが一つもない場合 */
  if(root == NULL){
    /* 根ノードとしてノードを追加 */
    root = mallocNode(number, name);
    if(root == NULL){
      printf("malloc error\n");
      return NULL;
    }
    return root;
  }

  /* 根ノードから順に追加する場所を探索 */
  node = root;
  while(1) {
    if(number < node->number){
      /* 追加する値がノードの値よりも小さい場合 */

      if(node->left == NULL){
        /* そのノードの左の子が無い場合（もう辿るべきノードが無い場合）*/

        /* その左の子の位置にノードを追加 */
        node->left = mallocNode(number, name);

        /* 追加完了したので処理終了 */
        break;
      }

              /* 左ノードを辿ったことを覚えておく */
      branch[num_branch] = TREE_LEFT;
      num_branch++;

      /* 左の子がある場合は左の子を新たな注目ノードに設定 */
      node = node->left;

    } else if(number > node->number){
      /* 追加する値がノードの値よりも大きい場合 */

      if(node->right == NULL){
        /* そのノードの右の子が無い場合（もう辿るべきノードが無い場合）*/

        /* その右の子の位置にノードを追加 */
        node->right = mallocNode(number, name);

        /* 追加完了したので処理終了 */
        break;
      }
```

```
326
327       /* 右ノードを辿ったことを覚えておく */
328       branch[num_branch] = TREE_RIGHT;
329       num_branch++;
330
331       /* 右の子がある場合は右の子を新たな注目ノードに設定 */
332       node = node->right;
333     } else {
334       /* 追加する値とノードの値が同じ場合 */
335
336       printf("%d already exist\n", number);
337       break;
338     }
339   }
340
341   return balancing(root, root, NULL, 0, branch, num_branch);
342 }
343
344 /* searchNode:指定されたnumberを持つノードを探索する
345    引数１ root ： 探索を開始するノードのアドレス
346    引数２ number ： 探索する会員番号
347    返却値 ： number を持つノードのアドレス（存在しない場合は NULL）*/
348 struct node_t *searchNode(struct node_t *root, int number){
349   struct node_t *node;
350
351   node = root;
352
353   /* 探索を行うループ（注目ノードがNULLになったら終了 */
354   while(node){
355     if(number < node->number){
356       /* 探索値がノードの値よりも小さい場合 */
357
358       /* 注目ノードを左の子ノードに設定 */
359       node = node->left;
360     } else if(number > node->number){
361       /* 探索値がノードの値よりも大きい場合 */
362
363       /* 注目ノードを右の子ノードに設定 */
364       node = node->right;
365     } else {
366       /* 探索値 = ノードの値の場合 */
367       return node;
368     }
369   }
370
371   /* 探索値を持つノードが見つからなかった場合 */
372   return NULL;
373 }
374
375 /* deleteNoChildeNode:指定された子の無いノードを削除する
376    引数１ root ： 木の根ノードのアドレス
377    引数２ node ： 削除するノードのアドレス
378    引数３ parent：削除するノードの親ノードのアドレス
379    返却値 ： 根ノードのアドレス */
380 struct node_t *deleteNoChildNode(struct node_t *root, struct node_t *node,
    struct node_t *parent){
381
382   if(parent != NULL){
```

```
    /* 親がいる場合（根ノード以外の場合）は
    削除対象ノードを指すポインタをNULLに設定 */
    if(parent->left ==  node){
       /* 削除対象ノードが親ノードから見て左の子の場合 */
        parent->left = NULL;
    } else {
      /* 削除対象ノードが親ノードから見て右の子の場合 */
      parent->right = NULL;
    }
    free(node);
  }  else {
    /* 削除対象ノードが根ノードの場合 */
    free(node);

    /* 根ノードを指すポインタをNULLに設定 */
    root = NULL;
  }

  return root;
}

/* deleteOneChildeNode:指定された子が一つのノードを削除する
   引数１ root : 木の根ノードのアドレス
   引数２ node : 削除するノードのアドレス
   引数３ child : 削除するノードの子ノードのアドレス
   返却値 : 根ノードのアドレス */
struct node_t *deleteOneChildNode(struct node_t *root, struct node_t *node,
struct node_t * child){

  /* 削除対象ノードにその子ノードのデータとポインタをコピー */
  node->number = child->number;
  strcpy(node->name, child->name);
  node->left = child->left;
  node->right = child->right;

  /* コピー元のノードを削除 */
  free(child);

  return root;
}

/* deleteTwoChildeNode:指定された子が二つのノードを削除する
   引数１ root : 木の根ノードのアドレス
   引数２ node : 削除するノードのアドレス
   返却値 : 根ノードのアドレス */
struct node_t *deleteTwoChildNode(struct node_t *root, struct node_t *node,
int *branch, int *num_branch){

  struct node_t *max;
  struct node_t *maxParent;

  /* 左の子から一番大きい値を持つノードを探索 */
  max = node->left;
  maxParent = node;

  /* 左の子ノードを辿ったことを覚えておく */
  branch[*num_branch] = TREE_LEFT;
  (*num_branch)++;
```

```
439
440   while(max->right != NULL){
441     maxParent = max;
442     max = max->right;
443
444     /* 右の子ノードを辿ったことを覚えておく */
445     branch[*num_branch] = TREE_RIGHT;
446     (*num_branch)++;
447   }
448   printf("max number is %d\n", max->number);
449
450   /* 最大ノードのデータのみ削除対象ノードにコピー */
451   node->number = max->number;
452   strcpy(node->name, max->name);
453
454   /* 最大ノードを削除 */
455
456   /* maxは最大ノードなので必ずmax->rightはNULLになる */
457   if(max->left == NULL){
458     /* 最大ノードに子がいない場合 */
459     root = deleteNoChildNode(root, max, maxParent);
460     } else {
461       /* 最大ノードに子供が一ついる場合 */
462       root = deleteOneChildNode(root, max, max->left);
463     }
464
465     return root;
466 }
467
468
469
470 /* deleteNode:指定されたnumberを持つノードを削除する
471    引数１ root : 木の根ノードのアドレス
472    引数２ number : 削除する会員番号
473    返却値 : 根ノードのアドレス */
474 struct node_t *deleteNode(struct node_t *root, int number){
475   struct node_t *node;
476   struct node_t *parent;
477   int branch[MAX_HEIGHT] = {0};
478   int num_branch = 0;
479
480   if(root == NULL){
481     return NULL;
482   }
483
484   /* 削除対象ノードを指すノードを探索 */
485   node = root;
486   parent = NULL;
487
488   while(node !=  NULL){
489     if(number < node->number){
490       parent = node;
491       node = node->left;
492
493       /* 左の子ノードを辿ったことを覚えておく */
494       branch[num_branch] = TREE_LEFT;
495       num_branch++;
496     } else if(number > node->number){
```

```
      parent = node;
      node = node->right;

      /* 右の子ノードを辿ったことを覚えておく */
      branch[num_branch] = TREE_RIGHT;
      num_branch++;
    } else {
      break;
    }
  }

  /* 指定されたnumberを値として持つノードが存在しない場合は何もせず終了 */
  if(node == NULL){
    printf("%d を持つノードが存在しません\n", number);
    return root;
  }

  printf("Delete %d(%s) node\n", node->number, node->name);

  if(node->left == NULL && node->right == NULL){
    /* 子がいないノードの削除 */
    root = deleteNoChildNode(root, node, parent);
  } else if((node->left != NULL && node->right == NULL) ||
    (node->right != NULL && node->left == NULL)){
    /* 子が一つしかない場合 */

    if(node->left != NULL){
      root = deleteOneChildNode(root, node, node->left);
    } else {
      root = deleteOneChildNode(root, node, node->right);
    }
  } else {
    /* 左の子と右の子両方がいるノードの削除 */
    root = deleteTwoChildNode(root, node, branch, &num_branch);
  }

  return balancing(root, root, NULL, 0, branch, num_branch);
}

/* printTree:rootを根ノードとする二分探索木をの全ノードを表示する
   引数1 root : 木の根ノードのアドレス
   引数2 depth: 関数呼び出しの深さ
   返却値 : なし */
void printTree(struct node_t *root, int depth){
  int i;

  if(root == NULL){
    return ;
  }

  /* 右の子孫ノードを表示 */
  printTree(root->right, depth+1);

  /* 深さをスペースで表現 */
  for(i = 0; i < depth; i++){
    printf("  ");
  }

```

```c
  /* ノードのデータを表示 */
  printf("+%3d(%s)\n", root->number, root->name);

  /* 左の子孫ノードを表示 */
  printTree(root->left, depth+1);

  depth++;
}

int main(void){
  struct node_t *root, *node;
  int input;
  int number;
  char name[MAX_NAME_LEN];
  int loop;

  /* まだ木がないのでrootをNULLにセット */
  root = NULL;

  /* 最初にてきとうにノードを追加しておく */
  root = addNode(root, 100, "100");
  root = addNode(root, 200, "200");
  root = addNode(root, 300, "300");
  root = addNode(root, 50, "50");
  root = addNode(root, 150, "150");
  root = addNode(root, 250, "250");
  root = addNode(root, 10, "1");
  root = addNode(root, 125, "125");
  root = addNode(root, 5, "5");
  root = addNode(root, 25, "25");
  root = addNode(root, 500, "500");
  root = addNode(root, 175, "175");
  root = addNode(root, 501, "501");
  root = addNode(root, 502, "502");
  root = addNode(root, 503, "503");
  root = addNode(root, 504, "504");
  root = addNode(root, 505, "505");
  root = addNode(root, 506, "506");
  root = addNode(root, 507, "507");
  root = addNode(root, 508, "508");
  root = addNode(root, 509, "509");
  root = addNode(root, 510, "510");

  loop = 1;
  while(loop){
    printf("処理を選択(1:add, 2:delete, 3:search, 4:exit)");
    scanf("%d", &input);

    switch(input){
    case 1:
      printf("会員番号(1 - 999):");
      scanf("%d", &number);
      if(number < 1 || number > 999){
        printf("値が範囲外です\n");
        continue;
      }

      printf("名前:");
```

```
613         scanf("%s", name);
614
615         root = addNode(root, number, name);
616         break;
617       case 2:
618         printf("会員番号(1 - 999):");
619         scanf("%d", &number);
620         if(number < 1 || number > 999){
621           printf("値が範囲外です\n");
622           continue;
623         }
624
625         root = deleteNode(root, number);
626
627         break;
628       case 3:
629         printf("会員番号(1 - 999):");
630         scanf("%d", &number);
631         if(number < 1 || number > 999){
632           printf("値が範囲外です\n");
633           continue;
634         }
635
636         node = searchNode(root, number);
637         if(node == NULL){
638           printf("number %d is not found\n", number);
639         } else {
640           printf("number %d : %s\n", number, node->name);
641         }
642         break;
643       default:
644         loop = 0;
645         break;
646       }
647       printTree(root, 0);
648     }
649
650     deleteTree(root);
651
652     return 0;
653 }
```

木の巡回

c : http://www.nct9.ne.jp/m_hiroi/linux/clang13.html

```
1 static void foreach_node(void (*func)(double), Node *node)
2 {
3   if (node != NULL) {
4     foreach_node(func, node->left);
5     func(node->item);
6     foreach_node(func, node->right);
7   }
8 }
```