

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
1  typedef struct AVL_tree_node
2  {
3      int value;
4      struct AVL_tree_node *left;
5      struct AVL_tree_node *right;
6  } 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認した。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
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "AVL_tree.h"
4
5  //新しいノードを作成
6  struct AVL_tree_node *malloc_node(int value)
7  {
```

```

8      struct ALV_tree_node *new_node = (struct ALV_tree_node
*)malloc(sizeof(struct ALV_tree_node));
9
10     new_node->value = value;
11     new_node->left = NULL;
12     new_node->right = NULL;
13
14     return new_node;
15 }
16
17 //新しいノードを追加
18 struct ALV_tree_node *add_node(struct ALV_tree_node *root, int value)
19 {
20     struct ALV_tree_node *tmp_node;
21
22     if (root == NULL)
23     {
24         root = malloc_node(value);
25         return root;
26     }
27
28     tmp_node = root;
29     while (1)
30     {
31         if (value < tmp_node->value)
32         {
33             if (tmp_node->left == NULL)
34             {
35                 tmp_node->left = malloc_node(value);
36                 break;
37             }
38             tmp_node = tmp_node->left;
39         }
40         else if (value > tmp_node->value)
41         {
42             if (tmp_node->right == NULL)
43             {
44                 tmp_node->right = malloc_node(value);
45                 break;
46             }
47             tmp_node = tmp_node->right;
48         }
49         else
50         {
51             printf("既に存在する値です\n");
52             break;
53         }
54     }
55     return root;
56 }
57
58 void printTree(struct ALV_tree_node *root, int depth)
59 {
60     int i;
61
62     if (root == NULL)
63     {
64         return;

```

```

65     }
66
67     /* 右の子孫ノードを表示 */
68     printTree(root->right, depth + 1);
69
70     /* 深さをスペースで表現 */
71     for (i = 0; i < depth; i++)
72     {
73         printf("---/", i);
74     }
75
76     /* ノードのデータを表示 */
77     printf("+%3d\n", root->value);
78
79     /* 左の子孫ノードを表示 */
80     printTree(root->left, depth + 1);
81
82     depth++;
83 }
84
85 struct ALV_tree_node *malloc_node(int);
86 struct ALV_tree_node *add_node(struct ALV_tree_node *, int);
87
88 int main()
89 {
90     struct ALV_tree_node *root = NULL;
91     root = add_node(root, 60);
92     root = add_node(root, 40);
93     root = add_node(root, 30);
94     root = add_node(root, 10);
95     root = add_node(root, 50);
96     root = add_node(root, 20);
97
98     printTree(root, 0);
99 }

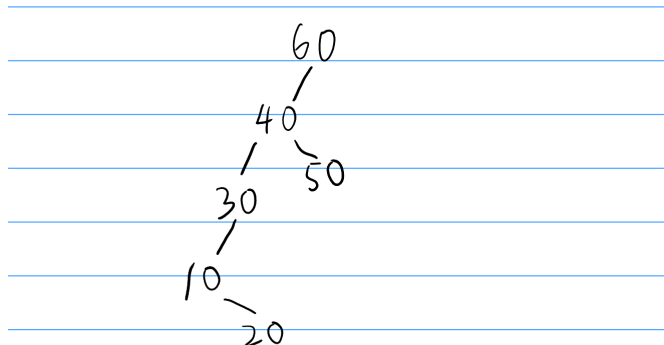
```

以下の出力がこのコードの実行結果である。

```

1  + 60
2  ---/---/+ 50
3  ---/+ 40
4  ---/---/+ 30
5  ---/---/---/---/+ 20
6  ---/---/---/+ 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

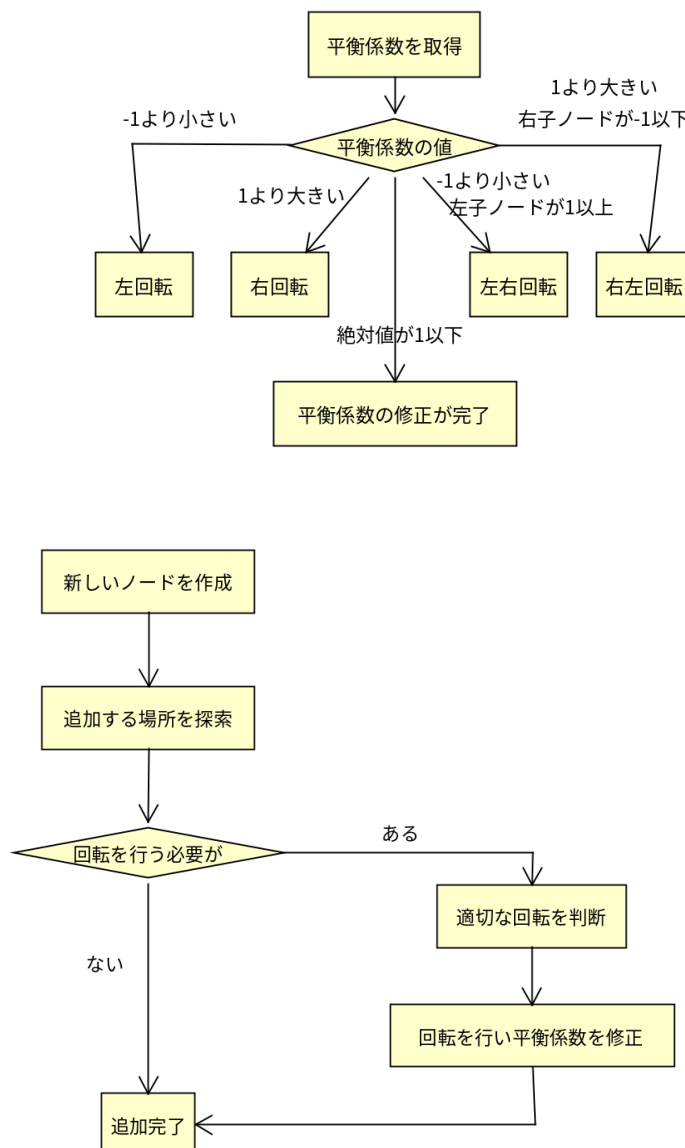
AVL木においては平衡係数の絶対値が1以下である必要があるため、二分木の実装に加えて平衡係数の修正を行う**回転**の動作が必要である。

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

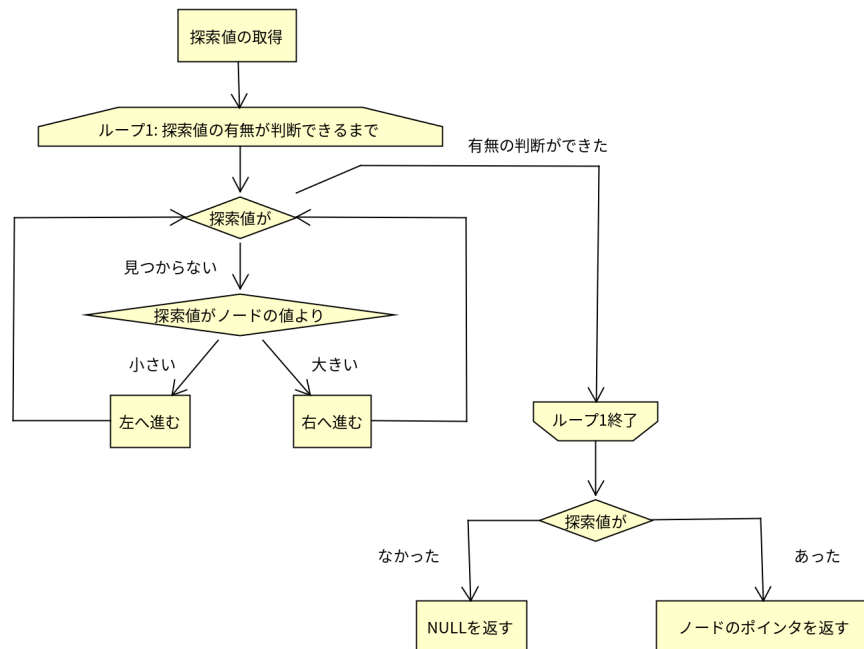
上記4つの回転が必要である。

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。

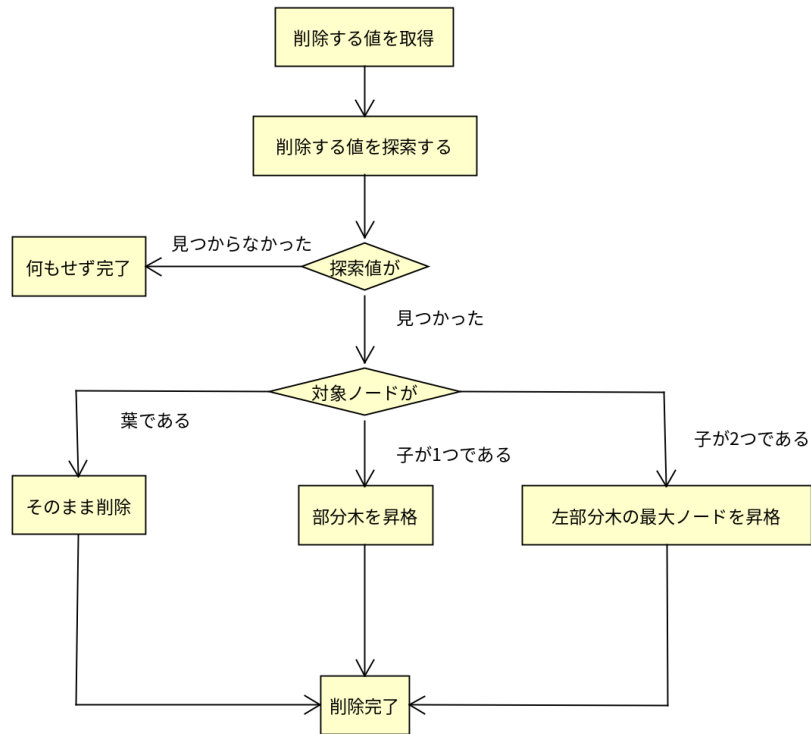


AVL木 削除

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

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  #define MAX_HEIGHT 100
6
7  #define TREE_LEFT 1
8  #define TREE_RIGHT 2
9
10 typedef struct AVL_tree_node
11 {
12     int value;
13     struct AVL_tree_node *right;
14     struct AVL_tree_node *left;
15 } avl_node;
16
17 int get_height(avl_node *node)
18 {
19     if (node == NULL)
20     {
21         return 0;
22     }
23
24     int left_height = get_height(node->left);
```

```

25     int right_height = get_height(node->right);
26
27     if (left_height > right_height)
28     {
29         return left_height + 1;
30     }
31     else
32     {
33         return right_height + 1;
34     }
35 }
36
37 avl_node *left_rotate(avl_node *root, avl_node *node, avl_node *parent, int
direction)
38 {
39     avl_node *pivot;
40     avl_node *new_root;
41
42     printf("left_rotate:%d\n", node->value);
43
44     pivot = node->right;
45
46     if (pivot != NULL)
47     {
48         node->right = pivot->left;
49         pivot->left = node;
50     }
51
52     if (parent == NULL)
53     {
54         new_root = pivot;
55         return new_root;
56     }
57
58     if (direction == TREE_LEFT)
59     {
60         parent->left = pivot;
61     }
62     else
63     {
64         parent->right = pivot;
65     }
66     return root;
67 }
68
69 avl_node *right_rotate(avl_node *root, avl_node *node, avl_node *parent,
int direction)
70 {
71     avl_node *pivot;
72     avl_node *new_root;
73
74     printf("right_rotate:%d\n", node->value);
75
76     pivot = node->left;
77
78     if (pivot != NULL)
79     {
80         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)

```



```

136     {
137         if (branch[0] == TREE_LEFT)
138         {
139             next = node->left;
140         }
141         else
142         {
143             next = node->right;
144         }
145         new_root = balancing(root, next, node, branch[0], &branch[1],
num_branch - 1);
146     }
147
148     left_height = get_height(node->left);
149     right_height = get_height(node->right);
150     balance = right_height - left_height;
151
152     if (balance > 1)
153     {
154         if (get_height(node->right->left) > get_height(node->right->right))
155         {
156             return rightright_rotate(new_root, node, parent, direction);
157         }
158         else
159         {
160             return left_rotate(new_root, node, parent, direction);
161         }
162     }
163     else if (balance < -1)
164     {
165         if (get_height(node->left->right) > get_height(node->left->left))
166         {
167             return leftleft_rotate(new_root, node, parent, direction);
168         }
169         else
170         {
171             return right_rotate(new_root, node, parent, direction);
172         }
173     }
174
175     return root;
176 }
177
178 avl_node *malloc_node(int value)
179 {
180     avl_node *new_node;
181
182     new_node = (avl_node *)malloc(sizeof(avl_node));
183     if (new_node == NULL)
184     {
185         return NULL;
186     }
187
188     new_node->value = value;
189     new_node->left = NULL;
190     new_node->right = NULL;
191
192     return new_node;

```

```

193 }
194
195 avl_node *add_node(avl_node *root, int value)
196 {
197     avl_node *node;
198     int branch[MAX_HEIGHT] = {0};
199     int num_branch = 0;
200
201     if (root == NULL)
202     {
203         root = malloc_node(value);
204         if (root == NULL)
205         {
206             printf("malloc error\n");
207             return NULL;
208         }
209         return root;
210     }
211
212     node = root;
213     while (1)
214     {
215         if (value < node->value)
216         {
217             if (node->left == NULL)
218             {
219                 node->left = malloc_node(value);
220                 break;
221             }
222             branch[num_branch] = TREE_LEFT;
223             num_branch++;
224             node = node->left;
225         }
226         else if (value > node->value)
227         {
228             if (node->right == NULL)
229             {
230                 node->right = malloc_node(value);
231                 break;
232             }
233             branch[num_branch] = TREE_RIGHT;
234             num_branch++;
235             node = node->right;
236         }
237         else
238         {
239             printf("%dは既に存在します\n", value);
240             break;
241         }
242     }
243
244     return balancing(root, root, NULL, 0, branch, num_branch);
245 }
246
247 avl_node *search_node(avl_node *root, int value)
248 {
249     avl_node *node;
250     node = root;

```

```

251     while (node)
252     {
253         if (value < node->value)
254         {
255             node = node->left;
256         }
257         else if (value > node->value)
258         {
259             node = node->right;
260         }
261         else
262         {
263             return node;
264         }
265     }
266
267     return NULL;
268 }
269
270 avl_node *delete_not_have_child_node(avl_node *root, avl_node *node,
271 avl_node *parent)
272 {
273     if (parent != NULL)
274     {
275         if (parent->left == node)
276         {
277             parent->left = NULL;
278         }
279         else
280         {
281             parent->right = NULL;
282         }
283         free(node);
284     }
285     else
286     {
287         free(node);
288         root = NULL;
289     }
290     return root;
291 }
292
293 avl_node *delete_have_one_child_node(avl_node *root, avl_node *node,
294 avl_node *child)
295 {
296     node->value = child->value;
297     node->left = child->left;
298     node->right = child->right;
299
300     free(child);
301
302     return root;
303 }
304
305 avl_node *delete_have_two_child_node(avl_node *root, avl_node *node, int
306 *branch, int *num_branch)
307 {
308     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)

```

```

364         {
365             parent = node;
366             node = node->right;
367
368             branch[num_branch] = TREE_RIGHT;
369             num_branch++;
370         }
371     else
372     {
373         break;
374     }
375 }
376
377 if (node == NULL)
378 {
379     printf("%dを持つノードが存在しません\n", value);
380     return root;
381 }
382
383 printf("Delete %d node\n", node->value);
384
385 if (node->left == NULL && node->right == NULL)
386 {
387     root = delete_not_have_child_node(root, node, parent);
388 }
389 else if ((node->left != NULL && node->right == NULL) || (node->right !=
NULL && node->left == NULL))
390 {
391     if (node->left != NULL)
392     {
393         root = delete_have_one_child_node(root, node, node->left);
394     }
395     else
396     {
397         root = delete_have_one_child_node(root, node, node->right);
398     }
399 }
400 else
401 {
402     root = delete_have_two_child_node(root, node, branch, &num_branch);
403 }
404
405 return balancing(root, root, NULL, 0, branch, num_branch);
406 }
407
408 void print_t(avl_node *root, int h)
409 {
410     if (root != NULL)
411     {
412         print_t(root->right, h + 1);
413         for (int i = 0; i < h; i++)
414         {
415             printf("|---|");
416         }
417         printf("%d\n", root->value);
418         print_t(root->left, h + 1);
419     }
420 }

```

```

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);

```

```

479         printf("\n-----\n");
480     }
481
482     deleteTree(root);
483
484     return 0;
485 }

```

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  1. 追加
2
3  60
4
5  -----
6  60
7  |---|40
8
9  -----
10 right_rotate:60
11 |---|60
12 40
13 |---|30
14
15 -----
16 |---|60
17 40
18 |---|30
19 |---||---|10
20
21 -----
22 |---|60
23 |---||---|50
24 40
25 |---|30
26 |---||---|10
27
28 -----
29 left_right_rotate:30
30 left_rotate:10
31 right_rotate:30
32 |---|60
33 |---||---|50
34 40
35 |---||---|30

```

```

36 |---|20
37 |---||---|10
38
39 -----
40
41
42 2. 探索
43
44 30を発見しました
45 55は見つかりませんでした
46
47
48 3. 削除
49
50 Delete 20 node
51 max value is 10
52 |---|60
53 |---||---|50
54 40
55 |---||---|30
56 |---|10
57
58 -----
59 Delete 40 node
60 max value is 30
61 |---|60
62 |---||---|50
63 30
64 |---|10
65
66 -----
67 Delete 60 node
68 |---|50
69 30
70 |---|10
71
72 -----
73 Delete 30 node
74 max value is 10
75 |---|50
76 10
77
78 -----
79 Delete 50 node
80 10
81
82 -----
83 Delete 10 node
84
85 -----

```

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/>

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  #define MAX_NAME_LEN 256
6
7  /* 二分探索木のノードを表す構造体 */
8  struct node_t {
9      int number;
10     char name[MAX_NAME_LEN];
11     struct node_t *left;
12     struct node_t *right;
13 };
14
15
16 /* deleteTree:二分探索木のノード全てを削除する
17     引数1 root : 根ノードのアドレス
18     返却値 : なし */
19 void deleteTree(struct node_t *root){
20     if(root == NULL){
21         return;
22     }
23
24     deleteTree(root->left);
25     deleteTree(root->right);
26
27     printf("free:%d(%s)\n", root->number, root->name);
28     free(root);
29
30 }
31
32 /* mallocNode:ノードの構造体のメモリを確保し、データを設定
33     引数1 number : 追加する会員番号
34     引数2 name : 追加する会員の名前
35     返却値 : 追加したノードのアドレス */
36 struct node_t *mallocNode(int number, char *name){
37     struct node_t *add;
38
39     add = (struct node_t*)malloc(sizeof(struct node_t));
40     if(add == NULL){
41         return NULL;
42     }
```

```

43
44     add->left = NULL;
45     add->right = NULL;
46     add->number = number;
47     strcpy(add->name, name);
48
49     return add;
50 }
51
52 /* addNode:指定されたnumberとname持つノードを追加する
53     引数1 root : 根ノードのアドレス
54     引数2 number : 追加する会員番号
55     引数3 name : 追加する会員の名前
56     返却値 : 根ルートのアドレス */
57 struct node_t *addNode(struct node_t *root, int number, char *name){
58     struct node_t *node;
59
60     /* まだノードが一つもない場合 */
61     if(root == NULL){
62         /* 根ノードとしてノードを追加 */
63         root = mallocNode(number, name);
64         if(root == NULL){
65             printf("malloc error\n");
66             return NULL;
67         }
68         return root;
69     }
70
71     /* 根ノードから順に追加する場所を探索 */
72     node = root;
73     while(1) {
74         if(number < node->number){
75             /* 追加する値がノードの値よりも小さい場合 */
76
77             if(node->left == NULL){
78                 /* 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） */
79
80                 /* その左の子の位置にノードを追加 */
81                 node->left = mallocNode(number, name);
82
83                 /* 追加完了したので処理終了 */
84                 break;
85             }
86
87             /* 左の子がある場合は左の子を新たな注目ノードに設定 */
88             node = node->left;
89
90         } else if(number > node->number){
91             /* 追加する値がノードの値よりも大きい場合 */
92
93             if(node->right == NULL){
94                 /* 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） */
95
96                 /* その右の子の位置にノードを追加 */
97                 node->right = mallocNode(number, name);
98
99                 /* 追加完了したので処理終了 */
100                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 /* deleteNoChildNode: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,
153 struct node_t *parent){
154
155     if(parent != NULL){
156         /* 親がいる場合 (根ノード以外の場合) は
157         削除対象ノードを指すポインタをNULLに設定 */
158         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 /* deleteOneChildNode: 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,
182 struct node_t * child){
183
184     /* 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 */
185     node->number = child->number;
186     strcpy(node->name, child->name);
187     node->left = child->left;
188     node->right = child->right;
189
190     /* コピー元のノードを削除 */
191     free(child);
192
193     return root;
194 }
195
196 /* deleteTwoChildNode: 指定された子が二つのノードを削除する
197  引数1 root : 木の根ノードのアドレス
198  引数2 node : 削除するノードのアドレス
199  返却値 : 根ノードのアドレス */
200 struct node_t *deleteTwoChildNode(struct node_t *root, struct node_t *node)
201 {
202     struct node_t *max;
203     struct node_t *maxParent;
204
205     /* 左の子から一番大きい値を持つノードを探索 */
206     max = node->left;
207     maxParent = node;
208
209     while(max->right != NULL){
210         maxParent = max;
211         max = max->right;
212     }
213     printf("max number is %d\n", max->number);

```

```

214  /* 最大ノードのデータのみ削除対象ノードにコピー */
215  node->number = max->number;
216  strcpy(node->name, max->name);
217
218  /* 最大ノードを削除 */
219
220  /* maxは最大ノードなので必ずmax->rightはNULLになる */
221  if(max->left == NULL){
222      /* 最大ノードに子がない場合 */
223      root = deleteNoChildNode(root, max, maxParent);
224  } else {
225      /* 最大ノードに子供が一ついる場合 */
226      root = deleteOneChildNode(root, max, max->left);
227  }
228
229  return root;
230 }
231
232
233
234  /* deleteNode:指定されたnumberを持つノードを削除する
235     引数1 root : 木の根ノードのアドレス
236     引数2 number : 削除する会員番号
237     返却値 : 根ノードのアドレス */
238  struct node_t *deleteNode(struct node_t *root, int number){
239      struct node_t *node;
240      struct node_t *parent;
241
242      if(root == NULL){
243          return NULL;
244      }
245
246      /* 削除対象ノードを指すノードを探索 */
247      node = root;
248      parent = NULL;
249
250      while(node != NULL){
251          if(number < node->number){
252              parent = node;
253              node = node->left;
254          } else if(number > node->number){
255              parent = node;
256              node = node->right;
257          } else {
258              break;
259          }
260      }
261
262      /* 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 */
263      if(node == NULL){
264          printf("%d を持つノードが存在しません\n", number);
265          return root;
266      }
267
268      printf("Delete %d(%s) node\n", node->number, node->name);
269
270      if(node->left == NULL && node->right == NULL){
271          /* 子がないノードの削除 */

```

```

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");

```

```

330 root = addNode(root, 200, "200");
331 root = addNode(root, 300, "300");
332 root = addNode(root, 50, "50");
333 root = addNode(root, 150, "150");
334 root = addNode(root, 250, "250");
335 root = addNode(root, 10, "1");
336 root = addNode(root, 125, "125");
337 root = addNode(root, 5, "5");
338 root = addNode(root, 25, "25");
339 root = addNode(root, 500, "500");
340 root = addNode(root, 175, "175");
341
342 loop = 1;
343 while(loop){
344     printf("処理を選択(1:add, 2:delete, 3:search, 4:exit)");
345     scanf("%d", &input);
346
347     switch(input){
348     case 1:
349         printf("会員番号(1 - 999):");
350         scanf("%d", &number);
351         if(number < 1 || number > 999){
352             printf("値が範囲外です\n");
353             continue;
354         }
355
356         printf("名前:");
357         scanf("%s", name);
358
359         root = addNode(root, number, name);
360         break;
361     case 2:
362         printf("会員番号(1 - 999):");
363         scanf("%d", &number);
364         if(number < 1 || number > 999){
365             printf("値が範囲外です\n");
366             continue;
367         }
368
369         root = deleteNode(root, number);
370
371         break;
372     case 3:
373         printf("会員番号(1 - 999):");
374         scanf("%d", &number);
375         if(number < 1 || number > 999){
376             printf("値が範囲外です\n");
377             continue;
378         }
379
380         node = searchNode(root, number);
381         if(node == NULL){
382             printf("number %d is not found\n", number);
383         } else {
384             printf("number %d : %s\n", number, node->name);
385         }
386         break;
387     default:

```

```

388     loop = 0;
389     break;
390 }
391 printTree(root, 0);
392 }
393
394 deleteTree(root);
395
396 return 0;
397 }

```

AVL木

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

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  #define MAX_NAME_LEN 256
6  #define MAX_HEIGHT 100
7
8  #define TREE_LEFT 1
9  #define TREE_RIGHT 2
10
11 /* 二分探索木のノードを表す構造体 */
12 struct node_t {
13     int number;
14     char name[MAX_NAME_LEN];
15     struct node_t *left;
16     struct node_t *right;
17 };
18
19 /* getHeight:二分探索木のノード全てを削除する
20    引数1 node : 根ノードのアドレス
21    返却値 : nodeを根とした木の高さ */
22 int getHeight(struct node_t *node) {
23
24     int left_height;
25     int right_height;
26     int tree_height;
27
28     if (node == NULL) {
29         /* nodeが無いなら高さは0 */
30         return 0;
31     }
32
33     /* 左右の子を根とした木の高さを取得 */
34     left_height = getHeight(node->left);
35     right_height = getHeight(node->right);
36
37     /* 大きい方に+1したものを木の高さとして返却 */
38     if (left_height > right_height) {
39         tree_height = left_height;
40     } else {
41         tree_height = right_height;

```



```

42     }
43
44     return tree_height + 1;
45 }
46
47 /* leftRotate:nodeを根とする部分木を回転（左）
48     引数1 root : 根のノードを指すアドレス
49     引数2 node : 回転する部分木の根ノードを指すアドレス
50     引数3 parent : nodeの親ノードを指すアドレス
51     引数4 direction : parentから見たnodeのある方向
52     返却値 : 根のノードを指すアドレス */
53 struct node_t *leftRotate(struct node_t *root, struct node_t *node, struct
54     node_t *parent, int direction) {
55     /* nodeを根として左回転を行う */
56
57     struct node_t *pivot;
58     struct node_t *new_root;
59
60     printf("left_rotate:%d\n", node->number);
61
62     /* 新しい根とするノードをpivotとして設定 */
63     pivot = node->right;
64
65     /* 左回転 */
66     if (pivot != NULL) {
67         node->right = pivot->left;
68         pivot->left = node;
69     }
70
71     /* parentもしくはrootに新しい根ノードを参照させる */
72     if (parent == NULL) {
73         new_root = pivot;
74         return new_root;
75     }
76
77     /* どちらの子に設定するかはdirectionから判断 */
78     if (direction == TREE_LEFT) {
79         parent->left = pivot;
80     } else {
81         parent->right = pivot;
82     }
83     return root;
84 }
85
86 /* rightRotate:nodeを根とする部分木を回転（右）
87     引数1 root : 根のノードを指すアドレス
88     引数2 node : 回転する部分木の根ノードを指すアドレス
89     引数3 parent : nodeの親ノードを指すアドレス
90     引数4 direction : parentから見たnodeのある方向
91     返却値 : 根のノードを指すアドレス */
92 struct node_t *rightRotate(struct node_t *root, struct node_t *node,
93     struct node_t *parent, int direction) {
94
95     struct node_t *pivot;
96     struct node_t *new_root;
97
98     printf("right_rotate:%d\n", node->number);
99

```

```

98  /* 新しい根とするノードをpivotとして設定 */
99  pivot = node->left;
100
101  /* 右回転 */
102  if (pivot != NULL) {
103      node->left = pivot->right;
104      pivot->right = node;
105  }
106
107  /* parentもしくはrootに新しい根ノードを参照させる */
108  if (parent == NULL) {
109      new_root = pivot;
110      return new_root;
111  }
112
113  /* どちらの子に設定するかはdirectionから判断 */
114  if (direction == TREE_LEFT) {
115      parent->left = pivot;
116  } else {
117      parent->right = pivot;
118  }
119
120  return root;
121 }
122
123 /* leftRightRotate:nodeを根とする部分木を二重回転 (右->左)
124  引数1 root : 根のノードを指すアドレス
125  引数2 node : 回転する部分木の根ノードを指すアドレス
126  引数3 parent : nodeの親ノードを指すアドレス
127  引数4 direction : parentから見たnodeのある方向
128  返却値 : 根のノードを指すアドレス */
129 struct node_t *rightLeftRotate(struct node_t *root, struct node_t *node,
130 struct node_t *parent, int direction) {
131
132     /* 2重回転 (Right Left Case) を行う */
133
134     struct node_t *new_root;
135     printf("right_left_rotate:%d\n", node->number);
136
137     /* nodeの右の子ノードを根として右回転 */
138     new_root = rightRotate(root, node->right, node, TREE_RIGHT);
139
140     /* nodeを根として左回転 */
141     return leftRotate(new_root, node, parent, direction);
142 }
143
144 /* leftRightRotate:nodeを根とする部分木を二重回転 (左->右)
145  引数1 root : 根のノードを指すアドレス
146  引数2 node : 回転する部分木の根ノードを指すアドレス
147  引数3 parent : nodeの親ノードを指すアドレス
148  引数4 direction : parentから見たnodeのある方向
149  返却値 : 根のノードを指すアドレス */
150 struct node_t *leftRightRotate(struct node_t *root, struct node_t *node,
151 struct node_t *parent, int direction) {
152
153     /* 2重回転 (Left Right Case) を行う */
154
155     struct node_t *new_root;
156
157     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         } else {
209

```

```

210     /* 1重回転 (左回転) */
211     return leftRotate(new_root, node, parent, direction);
212 }
213
214 } else if (balance < -1) {
215     /* 左の部分木が高くて並行状態でない場合 */
216
217     /* 2重回転が必要かどうかを判断 */
218     if (getHeight(node->left->right) > getHeight(node->left->left)) {
219         /* 2重回転 (Left Right Case) */
220         return leftRightRotate(new_root, node, parent, direction);
221     } else {
222         /* 1重回転 (右回転) */
223         return rightRotate(new_root, node, parent, direction);
224     }
225 }
226
227 return root;
228 }
229
230 /* deleteTree:二分探索木のノード全てを削除する
231 引数1 root : 根ノードのアドレス
232 返却値 : なし */
233 void deleteTree(struct node_t *root){
234     if(root == NULL){
235         return;
236     }
237
238     if(root->left != NULL){
239         deleteTree(root->left);
240     }
241     if(root->right != NULL){
242         deleteTree(root->right);
243     }
244
245     printf("free:%d(%s)\n", root->number, root->name);
246     free(root);
247
248 }
249
250 /* mallocNode:ノードの構造体のメモリを確保し、データを設定
251 引数1 number : 追加する会員番号
252 引数2 name : 追加する会員の名前
253 返却値 : 追加したノードのアドレス */
254 struct node_t *mallocNode(int number, char *name){
255     struct node_t *add;
256
257     add = (struct node_t*)malloc(sizeof(struct node_t));
258     if(add == NULL){
259         return NULL;
260     }
261
262     add->left = NULL;
263     add->right = NULL;
264     add->number = number;
265     strcpy(add->name, name);
266
267     return add;

```

```

268 }
269
270 /* addNode:指定されたnumberとname持つノードを追加する
271 引数1 root : 根ノードのアドレス
272 引数2 number : 追加する会員番号
273 引数3 name : 追加する会員の名前
274 返却値 : 根ルートのアドレス */
275 struct node_t *addNode(struct node_t *root, int number, char *name){
276     struct node_t *node;
277     int branch[MAX_HEIGHT] = {0};
278     int num_branch = 0;
279
280     /* まだノードが一つもない場合 */
281     if(root == NULL){
282         /* 根ノードとしてノードを追加 */
283         root = mallocNode(number, name);
284         if(root == NULL){
285             printf("malloc error\n");
286             return NULL;
287         }
288         return root;
289     }
290
291     /* 根ノードから順に追加する場所を探索 */
292     node = root;
293     while(1) {
294         if(number < node->number){
295             /* 追加する値がノードの値よりも小さい場合 */
296
297             if(node->left == NULL){
298                 /* 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） */
299
300                 /* その左の子の位置にノードを追加 */
301                 node->left = mallocNode(number, name);
302
303                 /* 追加完了したので処理終了 */
304                 break;
305             }
306
307             /* 左ノードを辿ったことを覚えておく */
308             branch[num_branch] = TREE_LEFT;
309             num_branch++;
310
311             /* 左の子がある場合は左の子を新たな注目ノードに設定 */
312             node = node->left;
313
314         } else if(number > node->number){
315             /* 追加する値がノードの値よりも大きい場合 */
316
317             if(node->right == NULL){
318                 /* 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） */
319
320                 /* その右の子の位置にノードを追加 */
321                 node->right = mallocNode(number, name);
322
323                 /* 追加完了したので処理終了 */
324                 break;
325             }

```

```

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 /* deleteNoChildNode: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,
381 struct node_t *parent){
382     if(parent != NULL){

```

```

383     /* 親がいる場合（根ノード以外の場合）は
384     削除対象ノードを指すポインタをNULLに設定 */
385     if(parent->left == node){
386         /* 削除対象ノードが親ノードから見て左の子の場合 */
387         parent->left = NULL;
388     } else {
389         /* 削除対象ノードが親ノードから見て右の子の場合 */
390         parent->right = NULL;
391     }
392     free(node);
393 } else {
394     /* 削除対象ノードが根ノードの場合 */
395     free(node);
396
397     /* 根ノードを指すポインタをNULLに設定 */
398     root = NULL;
399 }
400
401 return root;
402 }
403
404 /* deleteOneChildNode:指定された子が一つのノードを削除する
405     引数1 root : 木の根ノードのアドレス
406     引数2 node : 削除するノードのアドレス
407     引数3 child : 削除するノードの子ノードのアドレス
408     返却値 : 根ノードのアドレス */
409 struct node_t *deleteOneChildNode(struct node_t *root, struct node_t *node,
410 struct node_t * child){
411
412     /* 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 */
413     node->number = child->number;
414     strcpy(node->name, child->name);
415     node->left = child->left;
416     node->right = child->right;
417
418     /* コピー元のノードを削除 */
419     free(child);
420
421     return root;
422 }
423
424 /* deleteTwoChildNode:指定された子が二つのノードを削除する
425     引数1 root : 木の根ノードのアドレス
426     引数2 node : 削除するノードのアドレス
427     返却値 : 根ノードのアドレス */
428 struct node_t *deleteTwoChildNode(struct node_t *root, struct node_t *node,
429 int *branch, int *num_branch){
430
431     struct node_t *max;
432     struct node_t *maxParent;
433
434     /* 左の子から一番大きい値を持つノードを探索 */
435     max = node->left;
436     maxParent = node;
437
438     /* 左の子ノードを辿ったことを覚えておく */
439     branch[*num_branch] = TREE_LEFT;
440     (*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){

```



```

497     parent = node;
498     node = node->right;
499
500     /* 右の子ノードを辿ったことを覚えておく */
501     branch[num_branch] = TREE_RIGHT;
502     num_branch++;
503 } else {
504     break;
505 }
506 }
507
508 /* 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 */
509 if(node == NULL){
510     printf("%d を持つノードが存在しません\n", number);
511     return root;
512 }
513
514 printf("Delete %d(%s) node\n", node->number, node->name);
515
516 if(node->left == NULL && node->right == NULL){
517     /* 子がないノードの削除 */
518     root = deleteNoChildNode(root, node, parent);
519 } else if((node->left != NULL && node->right == NULL) ||
520 (node->right != NULL && node->left == NULL)){
521     /* 子が一つしかない場合 */
522
523     if(node->left != NULL){
524         root = deleteOneChildNode(root, node, node->left);
525     } else {
526         root = deleteOneChildNode(root, node, node->right);
527     }
528 } else {
529     /* 左の子と右の子両方がいるノードの削除 */
530     root = deleteTwoChildNode(root, node, branch, &num_branch);
531 }
532
533 return balancing(root, root, NULL, 0, branch, num_branch);
534 }
535
536 /* 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
537  引数1 root : 木の根ノードのアドレス
538  引数2 depth: 関数呼び出しの深さ
539  返却値 : なし */
540 void printTree(struct node_t *root, int depth){
541     int i;
542
543     if(root == NULL){
544         return ;
545     }
546
547     /* 右の子孫ノードを表示 */
548     printTree(root->right, depth+1);
549
550     /* 深さをスペースで表現 */
551     for(i = 0; i < depth; i++){
552         printf(" ");
553     }
554

```

```

555     /* ノードのデータを表示 */
556     printf("%+3d(%s)\n", root->number, root->name);
557
558     /* 左の子孫ノードを表示 */
559     printTree(root->left, depth+1);
560
561     depth++;
562 }
563
564 int main(void){
565     struct node_t *root, *node;
566     int input;
567     int number;
568     char name[MAX_NAME_LEN];
569     int loop;
570
571     /* まだ木がないのでrootをNULLにセット */
572     root = NULL;
573
574     /* 最初にてきとうにノードを追加しておく */
575     root = addNode(root, 100, "100");
576     root = addNode(root, 200, "200");
577     root = addNode(root, 300, "300");
578     root = addNode(root, 50, "50");
579     root = addNode(root, 150, "150");
580     root = addNode(root, 250, "250");
581     root = addNode(root, 10, "1");
582     root = addNode(root, 125, "125");
583     root = addNode(root, 5, "5");
584     root = addNode(root, 25, "25");
585     root = addNode(root, 500, "500");
586     root = addNode(root, 175, "175");
587     root = addNode(root, 501, "501");
588     root = addNode(root, 502, "502");
589     root = addNode(root, 503, "503");
590     root = addNode(root, 504, "504");
591     root = addNode(root, 505, "505");
592     root = addNode(root, 506, "506");
593     root = addNode(root, 507, "507");
594     root = addNode(root, 508, "508");
595     root = addNode(root, 509, "509");
596     root = addNode(root, 510, "510");
597
598     loop = 1;
599     while(loop){
600         printf("処理を選択(1:add, 2:delete, 3:search, 4:exit)");
601         scanf("%d", &input);
602
603         switch(input){
604             case 1:
605                 printf("会員番号(1 - 999):");
606                 scanf("%d", &number);
607                 if(number < 1 || number > 999){
608                     printf("値が範囲外です\n");
609                     continue;
610                 }
611
612                 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 }

```

