# 平成 27 年度情報工学実験第一・第二のための春休み課題（その 2）

笹倉　万里子 †　渡邊誠也 ††　佐藤将也 †††

この課題は，平成 27 年度情報工学実験第一・第二の準備として，実験受講予定者のみなさんに春休み中に解答していただくものです．平成 27 年度の第一回目の実験の時間にこの課題の解答を報告書にまとめて持参してください．報告書にはそれぞれのプログラムの中の空白 (1)〜(13) に入るコードおよびそれらのコードにした根拠等を記述してください．なお，報告書は手書きでも可とします．課題（その 1）と（その 2）は提出先が異なりますので，2 つの報告書を綴じないでください．

## 1. （課題 I）スタック

スタックは，最後に格納したデータが最初に取り出される (LIFO: last-in first-out) 仕組みのデータ構造である．通常，スタックにデータを格納すること（これを「データを積む」と呼ぶこともある）をプッシュ (push), データを取り出すことをポップ (pop) と呼ぶ．

### 1.1 配列を用いたスタックの実現

整数を格納する 2 つのスタックを用いるプログラムの例を**図 1** に示す．図 1 において，2 つのスタック `stack_1` と `stack_2` は整数の配列を用いて大域変数として宣言されている (10, 11 行目)．2 つのスタックにおいて，スタックの一番上がどこかを示すスタックポインタはそれぞれ変数 `counter_1` と `counter_2` である．これらの変数を配列の添字として使うことでスタックポインタを実現している．このスタックポインタは，次にスタックにデータを入れる時，配列のどの場所にデータをいれればよいのかを示すようになっている．このことは変数 `counter_1` と `counter_2` の初期値が 0 であることからわかる．これらも大域変数として宣言されている (13, 14 行目)．

スタックの動きを決めるプッシュ操作とポップ操作は，それぞれ `push`, `pop` という関数で実現されている．図 1 のプログラムでは 2 つのスタックを扱っているので，どちらのスタックに対してプッシュ操作あるいはポップ操作を行うかを，各操作に対応するそれぞれの関数の第 1 引数 (`int stackid`) で指定する．具体的には，`stackid` の値が 1 の場合は `stack_1` に対して，2 の場合は `stack_2` に対してプッシュ操作ある

```
1   /*
2    * 春課題 I : スタック配列版
3    */
4
5   #include <stdio.h>
6   #include <stdlib.h>
7
8   #define STACKMAX 50
9
10  int stack_1[STACKMAX];
11  int stack_2[STACKMAX];
12
13  int counter_1 = 0;
14  int counter_2 = 0;
15
16  void
17  push(int stackid, int n) {
18    if (stackid == 1) {
19      if (counter_1 == STACKMAX) {
20        printf("Stack_1 overflow.\n");
21      } else {
22        stack_1[counter_1++] = n;
23      }
24    } else if (stackid == 2) {
25      if (counter_2 == STACKMAX) {
26        printf("Stack_2 overflow.\n");
27      } else {
28        stack_2[      (1)      ] = n;
29      }
30    }
31  }
32
33  int
34  pop(int stackid) {
35    if (stackid == 1) {
36      if (counter_1 < 1) {
37        printf("Stack_1 underflow.\n");
38        exit(1);
39      } else {
40        return stack_1[--counter_1];
41      }
42    } else if (stackid == 2) {
43      if (counter_2 < 1) {
44        printf("Stack_2 underflow.\n");
45        exit(1);
46      } else {
47        return stack_2[      (2)      ];
48      }
49    }
50  }
51
52  int
53  main(int argc, char *argv[]) {
54    push(1,1);
55    push(2,2);
56
57    push(1,3);
58    push(2,4);
59
60    push(1,5);
61
62    printf("From Stack1: %d\n", pop(1));
63    printf("From Stack1: %d\n", pop(1));
64    printf("From Stack1: %d\n", pop(1));
65
66    printf("From Stack2: %d\n", pop(2));
67    printf("From Stack2: %d\n", pop(2));
68    printf("From Stack2: %d\n", pop(2));
69
70  }
```

**図 1**　スタック操作（配列版）のプログラムリスト (StackA.c)

† E-mail: sasakura@momo.cs.okayama-u.ac.jp
†† E-mail: nobuya@cs.okayama-u.ac.jp
††† E-mail: sato@cs.okayama-u.ac.jp
本課題に関する問い合わせは，下記のアドレス宛へ
E-mail: jikken-haru-kadai@arc.cs.okayama-u.ac.jp
追加情報等は，下記 URL を参照
http://www.arc.cs.okayama-u.ac.jp/~nobuya/jikken_kadai/

いはポップ操作を行う．プッシュ操作の際，スタックの大きさを超えてデータを格納しようとしたときは，“Stack_$n$ overflow.” というメッセージを画面に表示し，データは格納せずにプログラムの実行を続ける．ここで $n$ は stackid の値（1 または 2）である．一方，ポップ操作の際，スタックにデータがないのにポップをしようとしたときには，“Stack_$n$ underflow.” というメッセージを画面に表示し，実行を終了するようになっている．

図 1 に示すプログラムでは，stack_1 に対し，順に 1, 3, 5 を，stack_2 に対しては 2, 4 をプッシュした後，それぞれのスタックに 1 回ずつポップ操作を行う．stack_1 では，5, 3, 1 の順にデータがポップされて表示されるが，stack_2 では，4, 2 とポップされた後，“Stack_2 underflow.” のメッセージが画面に表示され実行を終了する．

図 1 のプログラムは配列を使って素直にスタックを実現しているが，次の点で美しいプログラムとは言えない．

(1) stack_1 のスタックポインタが counter_1 であるということが明確でない．つまり，ソースの細かい部分を読まないとわからない．
(2) プッシュ操作，ポップ操作を行う関数においてどちらのスタックに対してその操作を行うかを条件分岐で分けている．つまり，もし，スタックの数や名前を変更すれば，プッシュ操作，ポップ操作の関数の中身も書き換えなければならない．
(3) スタックやスタックポインタが大域変数として宣言されている．

以下では，これらを改良したプログラムの実現について説明する．

### 1.2 アドレス参照を用いたスタックの実現

図 1 のプログラムとほぼ同じスタックの動きを，アドレス参照を用いて実現するプログラムを**図 2** に示す．図 2 のプログラムは，図 1 に示したプログラムにおいて前述した「美しくない点」を以下のように改善したものである．

(1) 構造体を使うことで，スタック本体とスタックポインタの関係を明示的にしている．
(2) スタック自身をプッシュ，ポップの引数とすることで，スタックの個数や名前に依存しないプッシュ操作，ポップ操作を実現している．
(3) メイン関数の中でスタックを動的に確保し，プッシュ，ポップの引数としてそれを渡すことで，スタックを大域変数にしないで実現している．

また，このスタックの構造体では，スタック本体をポ

```
1   /*
2    * 春課題 I' : スタックアドレス参照版
3    */
4
5   #include <stdio.h>
6   #include <stdlib.h>
7   #include <string.h>
8
9   #define STACKMAX 50
10  #define STACK_NAME_MAX 16
11
12  typedef struct stack {
13    int size;
14    char name[STACK_NAME_MAX];
15    int *pointer;
16    int *limit;
17    int data[1];
18  } Stack;
19
20  Stack *
21  newStack(char *name, int size) {
22    Stack *stack = NULL;
23
24    if (size > 0) {
25      stack = (Stack *) malloc(sizeof(Stack) +
26                               sizeof(int) * size);
27      if (stack != NULL) {
28        stack->size = size;
29        strncpy(stack->name, name, STACK_NAME_MAX);
30        stack->pointer = &stack->data[0];
31        stack->limit = stack->data + sizeof(int) * size;
32      }
33    }
34    return stack;
35  } /* newStack */
36
37  void
38  push(Stack *stack, int value) {
39    if (stack->pointer < stack->limit) {
40      (3) = value;
41      stack->pointer++;
42    } else {
43      printf("%s: stack overflow.\n", stack->name);
44    }
45  } /* push */
46
47  int
48  pop(Stack *stack) {
49    if (stack->pointer > (4)) {
50      return *( (5) );
51    } else {
52      printf("%s: stack underflow.\n", stack->name);
53      exit(1);
54    }
55  } /* pop */
56
57  void
58  printStack(Stack *stack) {
59    int *p;
60
61    printf("%15s:", stack->name);
62    for (p = stack->data;
63         p < stack->limit && p < stack->pointer; p++) {
64      printf(" %d", *p);
65    }
66    printf("\n");
67  } /* printStack */
68
69  int
70  main(int argc, char *argv[]) {
71    int i;
72    Stack *stack1 = newStack("Stack1", STACKMAX);
73    Stack *stack2 = newStack("Stack2", STACKMAX);
74
75    push(stack1, 1);
76    push(stack2, 2);
77
78    push(stack1, 3);
79    push(stack2, 4);
80
81    push(stack1, 5);
82
83    printStack(stack1);
84    printStack(stack2);
85
86    for (i = 0; i < 3; i++)
87      printf("From %s: %d\n", stack1->name, pop(stack1));
88
89    for (i = 0; i < 3; i++)
90      printf("From %s: %d\n", stack2->name, pop(stack2));
91
92  } /* main */
```

**図 2** スタック操作（アドレス参照版）のプログラムリスト (StackB.c)

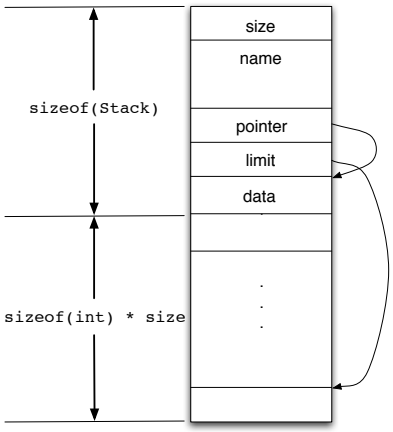


**図 3** Stack オブジェクトのメモリ上でのレイアウト

インタによるアドレス参照で実現しているので，スタックのサイズを呼び出し側で（このプログラムで言えば関数 `main` で）制御することが可能である．

まず，スタックの構造体について解説する（12〜18 行目）．構造体 `stack` は以下の 5 つのメンバを持っている．

`int size` このスタックに格納できるデータの最大数．

`char name[]` このスタックの名前文字列を格納する領域．

`int *pointer` このスタックのスタックポインタ．

`int *limit` このスタックに格納されるデータ数の上限を示すポインタ．

`int data[]` データを格納する領域（の先頭）．これ以降の連続したメモリ領域を確保し，この領域にデータが格納される．

この構造体 `stack` は typedef により別名が付けられ，この宣言以降は，`Stack` 型として取り扱うことが可能となる．

関数 `newStack`（20〜35 行目）は動的にスタックを生成するための関数である．引数としてスタックの名前（名前文字列が格納されている先頭アドレス）とスタック領域のサイズ（スタックに格納可能なデータの個数）を受け取り，結果として生成したスタックオブジェクトの先頭アドレスを返す．このプログラムの場合は，スタックに格納される値は整数であるため，生成されるスタックオブジェクトの大きさは，`Stack` 型のサイズ（`sizeof(Stack)`）と，データを格納するスタック領域のサイズ（`sizeof(int) * size`）の合計となる．生成されるスタックオブジェクトのメモリ上でのレイアウトイメージを**図 3** に示す．

`pointer` の初期値はスタック本体である `data` の先頭アドレスが格納される．`limit` にはスタック本体の最後のアドレスが格納され，スタックオーバフローの検知に使われる．スタックへのデータのプッシュとポップは `pointer` を進めたり減らしたりすることによって行われる（関数 `push`，`pop` を参照）．

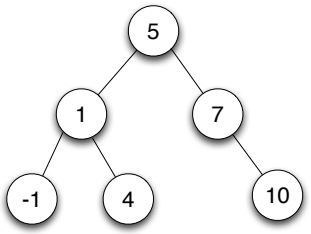


**図 4** 二分探索木の例

## 2. （課題 II）二分探索木

二分探索木とは，二分木で，かつ，すべてのノード $n$ について，

**二分探索木の条件** $V(n_{\mathrm{l}}) \leq V(n) \leq V(n_{\mathrm{r}})$

を満たす二分木のことである．ただし，ここで，$n_{\mathrm{l}}$ は $n$ の左側の子供，$n_{\mathrm{r}}$ は $n$ の右側の子供とし，$V(n)$ はノード $n$ の値を示すものとする．例えば，**図 4** は二分探索木の一例である．

本課題では，この二分探索木へのノードの追加と削除を扱う．ただし，簡単のために，本課題での二分探索木では，各ノードに対し $V(n_{\mathrm{l}}) < V(n) < V(n_{\mathrm{r}})$ が満たされるものとする．つまり，**同じ値をもつノードは存在しない二分探索木**を扱う．また，ノードの値は整数（int 型）とする．

このような二分探索木のノードの構造体を作り，そのノードをポインタでつなげていくことで木を作ることを考える．ここで定義するノードの構造体は，自分自身の値を格納する変数と，自分の左側の子供へのポインタ，および右側の子供へのポインタの 3 つのメンバからなる (8〜12 行目)．

### 2.1 ノードの追加

ノード $n$ の追加は，二分探索木の条件を満たすように適切な位置に新しいノードを作成する．**図 5** にノードを追加するアルゴリズムを示す．

### 2.2 ノードの削除

ノード $n$ の削除の場合は，$n$ に子供がいるかいないかで以下のように動作が異なる．

**ノード $n$ に子供がいない場合，** 単に $n$ を削除する．

**ノード $n$ の子供が左右どちらか 1 つしかいない場合，** その子供を $n$ の位置に持ってくる．

**ノード $n$ に左右どちらの子供もいる場合，** 左右どちらかの子供を $n$ の位置に持ってき，その子供の

(1) もし，この二分探索木に一つもノードがなかったら，ノード $n$ をルートノードとする．
(2) ルートノード $r$ があったなら，
　(a) もし $V(n) < V(r)$ ならば，$r$ の左側の子供をルートノードとする部分木に対して （1） から繰り返す．
　(b) もし $V(r) < V(n)$ ならば，$r$ の右側の子供をルートノードとする部分木に対して （1） から繰り返す．
　(c) そうでない場合，何もしないで終了する．

**図 5**　ノードを追加するアルゴリズム

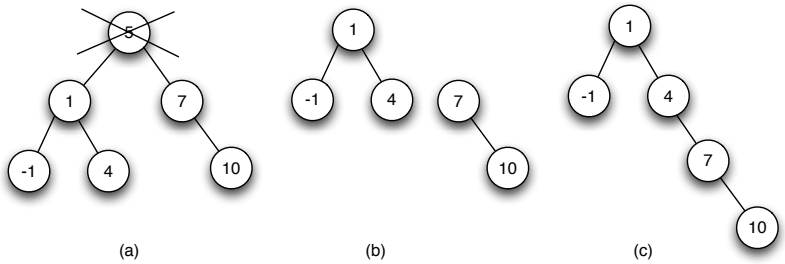


**図 6**　ノードの削除:(a) の図のような木の場合，ノード 5 を削除すると，ノード 5 の左の子供であるノード 1 がノード 5 の場所に来て，ノード 5 の右の子供であるノード 7 がノード 4 の右の子供となり (c) の図の木となる．

下の適切な場所にそれ以外の方の子供をつける．このプログラムでは，左右どちらの子供もいる場合，左側の子供を $n$ の位置にもってきているので，右側の子供を左側の子供の右側の子孫の右につける（**図 6** を参照）．

以上を整理し，ノード $n$ を削除する際のアルゴリズムを**図 7** に示す．

### 2.3　再帰を用いた二分探索木プログラムの実現

再帰を用いてノードの追加および削除を実現するプログラムを**図 10** に示す．また，図 10 に示すプログラムで用いている主要な関数とその説明を**図 8** に示す．

### 2.4　ループを用いた二分探索木プログラムの実現

二分探索木に対するノードの追加および削除を，再帰ではなくループ（繰り返し）を用いて実現したプログラムを**図 11** に示す．また，図 11 に示すプログラムで用いている主要な関数とその説明を**図 9** に示す．

一般に，再帰を用いる方がプログラムの可読性は高いが，ループを用いるほうがプログラムの実行速度が速い．そのため，特に大量のデータを高速で扱う必要がある場合にはループで実現したプログラムが用いられる．

`Node *newNode(int n)`　`n` で与えられた値をもつ新しいノードオブジェクトを生成し，そのアドレスを返す．
`Node *addNode(Node *obj, int n)`　`obj` で与えられたノードをルートノードとする木に対し，`n` を値としてもつ新しいノードを追加し，その木のルートノードのアドレスを返す．再帰的に自分自身を呼び出し，適切な位置に新しいノードを追加する．
`Node *appendRightEnd(Node *obj, Node *right)`　`obj` で与えられたノードの一番右端に `right` で与えられたノードを追加する．ノードの削除の後，削除されたノードの右側の子供を適切な位置に追加するために使用する．
`Node *deleteNode(Node *obj, int n)`　`n` で与えられた値をもつノードをノード `obj` をルートノードとする木から削除する．再帰的に自分自身を呼び出して値が `n` の値と一致するノードを探し出し，その後必要ならば関数 `appendRightEnd` を呼び出す．
`void printNodes(Node *obj)`　木の表示用の関数．

**図 8**　図 10 のプログラムで用いている主要な関数とその説明

`Node *newNode(int n)`　`n` で与えられた値をもつ新しいノードのオブジェクトを生成し，そのアドレスを返す．
`Node *addNode(Node *obj, int n)`　`obj` で与えられたノードをルートノードとする木に対し，`n` を値としてもつ新しいノードを追加し，その木のルートノードのアドレスを返す．ループを使って木をたどりながら，適切な位置に新しいノードを追加する．
`Node *deleteThisNode(Node *obj)`　`obj` で指定されたノード $n$ を削除する．ノード $n$ の右側の子供を適切な位置に追加し，ノード $n$ の代わりとなるノードのアドレスを返す．ノード $n$ の代わりとなるノードが無い場合は NULL を返す．
`Node *deleteNode(Node *obj, int n)`　`obj` で指定されたノードをルートノードとする木から，`n` で与えられた値をもつノードを削除する．ループを使って木をたどり，削除すべきノードを探す．削除すべきノードがあれば関数 `deleteThisNode` を呼び出す．
`void printNodes(Node *obj)`　木の表示用の関数．

**図 9**　図 11 のプログラムで用いている主要な関数とその説明

(1) もし，二分探索木にノードが 1 つもなかったら何もしないで終了する．
(2) ルートノード $r$ があったら，
　(a) もし $V(n) < V(r)$ ならば，$r$ の左側の子供をルートノードとする部分木に対して（１）から繰り返す．
　(b) もし $V(r) < V(n)$ ならば，$r$ の右側の子供をルートノードとする部分木に対して（１）から繰り返す．
　(c) そうでない場合，すなわち，$V(r) = V(n)$ の場合，
　　(i) もしノード $r$ に子供がいなければ，ノード $r$ を消去して終了．
　　(ii) もしノード $r$ の左側の子供がいなければ，ノード $r$ の場所にノード $r$ の右側の子を持ってきて終了．
　　(iii) もしノード $r$ に左側の子供がいてかつノード $r$ の右側の子供がいなければノード $r$ の場所にノード $r$ の左側の子を持ってきて終了．
　　(iv) そうでない場合，すなわち，ノード $r$ に左側の子供ノード $r_l$ と右側の子供ノード $r_r$ がある場合，
　　(A) もしノード $r_l$ の右側の子供ノードがなければ ノード $r_r$ を ノード $r_l$ の右側の子供として終了する．
　　(B) そうでなければ，ノード $r_l$ の右側の子供ノードに対して（Ａ）からを繰り返す．

**図 7**　ノードを削除するアルゴリズム

```
/*
 * 春課題 II : 二分探索木
 */

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

typedef struct node {
  int value;
  struct node *left;
  struct node *right;
} Node;

/* Node の作成 */
Node *
newNode(int n) {
  Node *obj;
  obj = (Node *)malloc(sizeof(Node));
  obj->value = n;
  obj->left = NULL;
  obj->right = NULL;
  return obj;
}


/* Node の追加 */
Node *
addNode(Node *obj, int n) {
  /* obj が NULL の場合 */
  if (obj == NULL) {
    return newNode(n);
  }

  /* 追加する場所を探して追加 */
  if (obj->value == n) {
    /* do nothing */ /* 追加しない */
  } else if(obj->value < n) {
    obj->right = addNode(obj->right, n);
  } else { /* obj->value > n の時 */
    obj->left = (6);
  }

  return obj;
} /* end of addNode */

/* Nodes を一番右に追加 */
Node *
appendRightEnd(Node *obj, Node *right) {
  if (obj != NULL) {
    if (obj->right == NULL) {
      obj->right = right;
    } else {
      obj->right = (7);
    }
  }
  return obj;
}

/* Node の削除 */
Node *
deleteNode(Node *obj, int n) {
  Node *newobj;

  if (obj != NULL) {
    if (obj->value == n) {

      if (obj->left == NULL) {
        newobj = obj->right;
        free(obj);
        return newobj;
      } else if(obj->right == NULL) {
        (8);
        free(obj);
        return newobj;
      } else {
        /* 右の子供たちのつけかえ */
        obj->left = appendRightEnd(obj->left, obj->right);
        newobj = obj->left;
        free(obj);
        return newobj;
      }
    } else {
      if (obj->value < n)
        obj->right = deleteNode(obj->right, n);
      else
        obj->left = (9);
    }
  }
  return obj;
}


/* Node の表示 */
void
printNodes(Node *obj){
  if (obj != NULL) {
    if (obj->left != NULL) {
      printNodes(obj->left);
    }

    printf("%d\t", obj->value);

    if (obj->right != NULL) {
      printNodes(obj->right);
    }
  }

}

int
main(int argc, char *argv[]){
  Node *topnode = NULL;

  topnode = addNode(topnode, 5);
  topnode = addNode(topnode, 7);
  topnode = addNode(topnode, 1);
  topnode = addNode(topnode, -1);
  topnode = addNode(topnode, 4);
  topnode = addNode(topnode, 10);

  printNodes(topnode);
  printf("\n");

  topnode = deleteNode(topnode, 5);

  printNodes(topnode);

}
```

**図 10**　二分探索木（再帰版）のプログラムリスト (BinaryTree.c)

```
1   /*
2    * 春課題 II' : 二分探索木ループ版
3    */
4
5   #include <stdio.h>
6   #include <stdlib.h>
7
8   typedef struct node{
9     int value;
10    struct node *left;
11    struct node *right;
12  } Node;
13
14  /* Node の作成 */
15  Node *
16  newNode(int n) {
17    Node* obj;
18    obj = (Node *)malloc(sizeof(Node));
19    obj->value = n;
20    obj->left = NULL;
21    obj->right = NULL;
22    return obj;
23  }
24
25
26  /* Node の追加 */
27  Node *
28  addNode(Node* obj, int n) {
29    Node *iobj;
30
31    /* obj が NULL の場合 */
32    if (obj == NULL) {
33      return newNode(n);
34    }
35
36    iobj = obj;
37
38    while (1) {
39      /* 追加する場所を探して追加 */
40      if (iobj->value == n) {
41        break;        /* 追加しない */
42      } else if (iobj->value < n) {
43        if(iobj->right == NULL){
44          iobj->right = (10);
45          break;
46        } else {
47          iobj = iobj->right;
48        }
49      } else { /* obj->value > n の時 */
50        if (iobj->left == NULL) {
51          iobj->left = newNode(n);
52          break;
53        } else {
54          iobj = iobj->left;
55        }
56      }
57    }
58    return obj;
59  } /* end of addNode */
60
61
62  /* Node の削除（指定された Node を削除）*/
63  Node *
64  deleteThisNode(Node *obj) {
65    Node *tmpobj;
66    Node *iobj;
67
68    if (obj == NULL) return NULL;
69    if (obj->left == NULL) {
70      tmpobj = obj->right;
71      free(obj);
72    } else if (obj->right == NULL) {
73      tmpobj = obj->left;
74      free(obj);
75    } else {
76      /* 右の子供たちのつけかえ */
77      tmpobj = obj->left;
78
79      iobj = tmpobj;
80      while (iobj->right != NULL) {
81        iobj = iobj->right;
82      }
83      iobj->right = (11);
84      free(obj);
85    }
86    return tmpobj;
87  }
88
89  /* Node の削除 */
90  Node *
91  deleteNode(Node *obj, int n) {
92    Node *pobj;
93
94    if (obj->value == n) {/* obj が該当 Node だった時 */
95      obj = deleteThisNode(obj);
96      return obj;
97    } else {
98
99      pobj = obj;
100     while (1) {
101       if (pobj->value < n) {
102         if (pobj->right == NULL) {
103           break; /* 該当なし */
104         } else {
105           if (pobj->right->value == n) {
106             pobj->right = (12);
107             break;
108           } else {
109             pobj = pobj->right;
110           }
111         }
112       } else { /* n < pobj->value */
113
114         if (pobj->left == NULL) {
115           break; /* 該当なし */
116         } else {
117           if (pobj->left->value == n) {
118             pobj->left = deleteThisNode(pobj->left);
119             break;
120           } else {
121             pobj = (13);
122           }
123         }
124       } /* end of n < pobj->vlaue */
125     }
126
127   }
128 }
129
130
131 /* Node の表示 */
132 void
133 printNodes(Node *obj) {
134   if (obj != NULL) {
135     if (obj->left != NULL) {
136       printNodes(obj->left);
137     }
138
139     printf("%d\t", obj->value);
140
141     if (obj->right != NULL) {
142       printNodes(obj->right);
143     }
144   }
145
146 }
147
148 int
149 main(int argc, char *argv[]){
150   Node *topnode = NULL;
151
152   topnode = addNode(topnode, 5);
153   topnode = addNode(topnode, 7);
154   topnode = addNode(topnode, 1);
155   topnode = addNode(topnode, -1);
156   topnode = addNode(topnode, 4);
157   topnode = addNode(topnode, 10);
158
159   printNodes(topnode);
160   printf("\n");
161
162   topnode = deleteNode(topnode, 5);
163
164   printNodes(topnode);
165
166 }
```

**図 11**　二分探索木（ループ版）のプログラムリスト (BinaryTreeL.c)