これまでの復習 (学年末試験に向けて)

山本昌志*

2006 年 2 月 27 日

1　試験範囲

これまでの，学習をまとめる．学年末試験には，このプリントに書いてある内容を理解して臨むこと。授業内容の分かっていない者は，少なくとも 10 時間は集中して勉強すること．どうしても理解できない場合は，成績の良い者に聞くなり，オフィスアワーを利用して，私に質問すること．

試験範囲は，

- 第 23 回の講義 (スタックとキュー)〜第 30 回の講義 (二分法) までである．ただし，第 27 回の講義で配布したプリント「マップとハッシュ」は試験範囲外とする．
- 教科書 [1] は，p.97〜252 までが範囲である．ただし，第 7 章の「マップとハッシュ」は範囲外とする．

である．教科書の内容のうち，説明を省いた部分も範囲外となるが，それを細かく指定することもできない．そこで，このプリントに書かれていることが試験範囲と思って欲しい．この程度のことが理解できていれば，十分合格点をとれる．

2　スタックとキュー

2.1　スタック

2.1.1　スタックのイメージ

図 1 のようなデータ構造である．データ構造であるから，データを蓄えることと，それを取り出すことができる．スタックの特徴は，最後に入れたデータが一番最初に取り出されることにある．取り出されるデータは，格納されている最新のデータで，最後に入れられたものが最初に取り出されることから，LIFO(last in first out, 後入れ先出し) と呼ばれる．スタックの途中のデータを取り出すことは許されない．スタックにデータを積むことをプッシュ(push) と，スタックからデータを取り出すことをポップ (pup) と呼ぶ．

*国立秋田工業高等専門学校　電気情報工学科

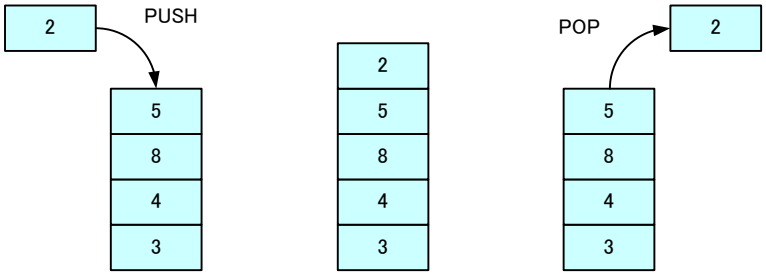


図 1: スタック

2.1.2 スタックの実装

スタックを実装する関数をリスト 1 に示す．これは，教科書の List 4-1(p.100) のプログラムである．スタックを実装するために必要な記憶領域は，スタックそのものの記憶領域とスタックの頂上を表す記憶領域である．スタックのための記憶領域として配列 `stack[]` を使っている．また，スタックの頂上を表すために，変数 `stack_top` を使っている[1]．

スタックのために必要な記憶領域が確保されたならば，それを操作しなくてはならない．スタックに必要な操作は 2 つで，データを積む (push) ことと，データを取り出す (pop) ことである．リスト 1 では，次の 2 つの関数でそれを行っている．

`void stack_push(double val)` スタックへデータ `val` をプッシュする関数である．データを積み重ねたならば，スタックの頂上を表す変数 `stack_top` を 1 加算している．スタックがいっぱいで，さらにデータをプッシュしようとするとプログラムは終了するようになっている．

`double stack_pop(void)` スタックからデータを取り出す関数で，戻り値が取り出されたデータである．データを取り出したならば，スタックの頂上を表す変数 `stack_top` を 1 減算している．スタックが空の状態でデータをポップしようとするとプログラムは終了するようになっている．

リスト 1: スタックの実装

```
1  #define STACK_MAX 10
2
3  /* ※この例では，double型のデータを格納するスタックを作成 */
4  double stack[STACK_MAX];
5  /* スタック頂上の位置（最下部からのオフセット） */
6  int stack_top=0;
7
8  /* スタックへpush */
9  void stack_push(double val)
10 {
11     if(stack_top==STACK_MAX)
12     {
13         /* スタックが満杯になっている */
14         printf("エラー:スタックが満杯です(Stack overflow)\n");
```

[1] ここでは単なる変数を使ったが，実際のプログラムではポインターが使われることが多い．そのため，頂上を表す変数をスタックポインターと呼ぶ．

```
15            exit(EXIT_FAILURE);
16        }
17        else
18        {
19            /* 渡された値をスタックに積む */
20            stack[stack_top]=val;
21            ++stack_top;
22        }
23    }
24
25    /* スタックからデータをpop */
26    double stack_pop(void)
27    {
28        if(stack_top==0)
29        {
30            /* スタックには何もない */
31            printf("エラー:スタックが空なのにpopが呼ばれました"
32                                                "(Stack underflow)\n");
33            exit(EXIT_FAILURE);
34            return 0;
35        }
36        else
37        {
38            /* いちばん上の値を返す */
39            --stack_top;
40            return stack[stack_top];
41        }
42    }
```

2.2 キュー

2.2.1 キューのイメージ

待ち行列と呼ばれるキュー (queue) と呼ばれるデータ構造がある．これは，queue とは窓口に並ぶ順番待ちの行列の意味で，図 2 のような構造である．スタックではデータの挿入と取り出しが列の一方からのみであったの対して，キューは列の両端から行う．一方がデータの追加で一方がデータの取り出しとして使われる．キューでは，最初に入れたデータが一番最初に取り出されることにある．取り出されるデータは格納されている最古のデータで，最初に入れられたものが最初に取り出されることから，FIFO(first in first out, 先入れ先出し) と呼ばれる．スタック同様，スタックの途中のデータを取り出すことは許されない．

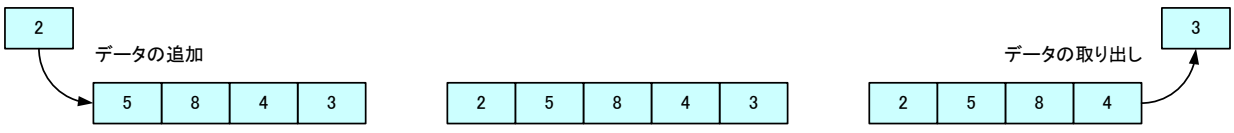


図 2: キュー

キューはスタックに比べて少しばかり，構造が複雑になっている．実際，それを直線的なイメージのメモリーにデータを追加しようとすると，以下のような問題が生じる．

- FIFO を続けると，いずれはメモリーの端に到達して，データの追加が出来なくなる．

- データを追加したり取り出したりする毎に，データの列を移動させることも考えられる．こうすると計算量が増加して不経済である．

これを防ぐために，図 3 のようなリングバッファと言うものが考えられた．

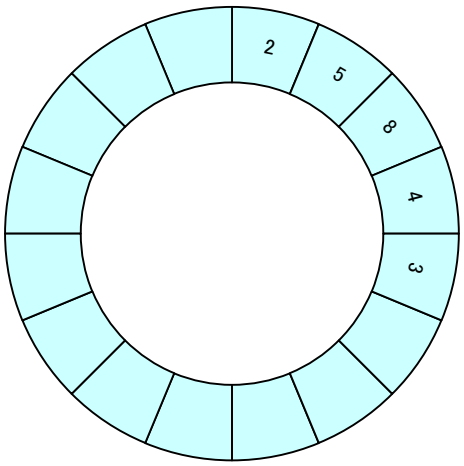


図 3: リングバッファー

2.2.2 キューの実装

キューを実装する関数をリスト 2 に示す．これは，教科書の List 4-4(p.120-121) と同じである．

このプログラムでは，配列 queue[] を使ってキューを実現している．配列 queue[] とキューの先頭を表す変数 queue_first と末尾を表す queue_last はグローバル変数と宣言されているので，以下のキューを操作する関数で直にアクセスすることができる．

キューを実装するための記憶領域が確保されたならば，それを操作しなくてはならない．キューの操作は 2 つで，データを追加することと，データを取り出すことである．リスト 2 では，次の 2 つの関数でそれを行っている．

`void enqueue(int val)` キューへデータ val を追加する関数である．データを追加したならば，キューの末尾を表す変数 queue_last を 1 移動 (加算) している．キューがいっぱいで，さらにデータを追加しようとするとプログラムはメッセージを出すようになっている．

`int dequeue(void)` キューからデータを取り出す関数で，戻り値が取り出したデータである．データを取り出したならば，キューの先頭を表す変数 queue_first を 1 移動 (加算) している．キューが空の状態でデータを取り出そうとするとプログラムは QUEUE_EMPTY を返すようになっている．

リスト 2: キューの実装

```
1 #define QUEUE_MAX 5+1    /* キューのサイズ+1 */
2 #define QUEUE_EMPTY -1
3
```

```
4  /* 配列によるキュー構造 */
5  int queue[QUEUE_MAX];
6  /* キューの先頭位置（配列先頭からのオフセット） */
7  int queue_first=0;
8  /* キューの末尾位置（配列先頭からのオフセット） */
9  int queue_last=0;
10
11 /* キューにデータを追加する */
12 void enqueue(int val)
13 {
14     /* lastの次がfirstならば */
15     if((queue_last+1)%QUEUE_MAX ==queue_first)
16     {
17         /* 現在配列の中身は，すべてキューの要素で埋まっている */
18         printf("ジョブが満杯です\n");
19     }
20     else
21     {
22         /* キューに新しい値を入れる */
23         queue[queue_last]=val;
24         /* queue_lastを1つ後ろにずらす。
25            もし，いちばん後ろの場合は，先頭にもってくる */
26         queue_last=(queue_last+1)%QUEUE_MAX;
27     }
28 }
29
30 /* キューからデータを取り出す */
31 int dequeue(void)
32 {
33     int ret;
34
35     if(queue_first==queue_last)
36     {
37         /* 現在キューは1つもない */
38         return QUEUE_EMPTY;
39     }
40     else
41     {
42         /* いちばん先頭のキューを返す準備 */
43         ret=queue[queue_first];
44         /* キューの先頭を次に移動する */
45         queue_first=(queue_first+1)%QUEUE_MAX;
46         return ret;
47     }
48 }
```

2.3 練習問題

[問 1] スタックと呼ばれるデータ構造を説明せよ．

[問 2] LIFO とは何か?

[問 3] スタックについて，以下の操作を定義する．

PUSH(n) :スタックにデータ n をプッシュする．戻り値は無し．

POP() :スタックからデータをポップする．戻り値は，取り出した値．

空のスタックに対して，以下の操作を行ったとき，データ構造の様子を図で示せ．

PUSH(3)→PUSH(5)→POP()→PUSH(2)→PUSH(1)→POP()→POP()→PUSH(1)→POP()→PUSH(7)

[**問 4**] 前問の POP() で取り出される値を示せ.

[**問 5**] キューと呼ばれるデータ構造を説明せよ.

[**問 6**] FIFO とは何か?

[**問 7**] キューにリングバッファーが使われる理由を説明せよ.

[**問 8**] キューについて，以下の操作を定義する.

ENQ(n) :キューにデータ n を追加する．戻り値はなし.
DEQ() :キューからデータを取り出す．戻り値は取り出した値.

空のスタックやキューに対して，以下の操作を行ったとき，データ構造の様子を図で示せ.

ENQ(6)→ENQ(2)→DEQ()→ENQ(7)→DEQ()→ENQ(3)→ENQ(1)→ENQ(2)→DEQ()→DEQ()

[**問 9**] 前問の DEQ() で取り出される値を示せ.

3 再帰呼び出し

3.1 再帰呼び出しとは

関数の中で，自分自身の関数を呼び出すことを再帰呼び出し (recursive call) と言う．このアルゴリズムを使うと，ある種の問題に対して，手続きが非常に簡素に表現できることがある.

これは，実際にプログラムを示した方がよく分かるので，簡単な例を示す．数学で，演算子！で表される階乗という演算にしばしばお目にかかる．たとえば，5 の階乗は

$$\begin{aligned} 5! &= 5 \times 4 \times 3 \times 2 \times 1 \\ &= 120 \end{aligned} \tag{1}$$

である．一般の整数 n では，

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1 \tag{2}$$

となる．また，$0! = 1$ となっている．これらのことから，漸化式は

$$n! = n \times (n-1)! \tag{3}$$

$$0! = 1 \tag{4}$$

と表すことができる.

ここで，整数を入力して，その階乗を計算するプログラムを作ってみよう．通常は，式 (2) の通りにプログラムを書くだろう．この式の通りにプログラムを書くと，リスト 3 のようになる.

同じ計算を，式 (3) と (4) に着目してプログラムを作ってみよう．すると，リスト 4 のようになる．このプログラムを見ると，漸化式をそのままプログラムしたことが分かるだろう.

リスト 4 のプログラムは，リスト 3 とは異なり，関数内で自分自身を呼びだしている．このように，自分自身を呼び出すことを再帰呼び出し (recursive call) と言う．C 言語ではこのように自分自身を呼び出すことができる．

これまでの例で，再帰呼び出しは繰り返し文と似ていることが分かっただろう．繰り返し文では，無限ループにならないように，必ず終了条件が必要であった．同じように，再帰呼び出しでも，呼び出しの終了条件が絶対に必要である．そうしないと，無限に関数を呼び出すことになり，いずれはプログラムがクラッシュするであろう．

もうひとつ重要なことは，再帰呼び出しが出きるようなアルゴリズムを考えなくてはならない．一つの考え方は，漸化式を書いてみることである．数学の漸化式のような考え方ができれば，それを忠実にプログラムすれば，再帰呼び出しができるであろう．

リスト 3: 繰り返し文を使った階乗を計算するプログラム

```
1  #include <stdio.h>
2
3  /*=============================================================*/
4  /*  階乗を計算する関数                                         */
5  /*=============================================================*/
6  int kaijyo(int n){
7    int i, value;
8
9    value=1;
10
11   for(i=n; i>=1; i--){
12     value*=i;
13   }
14
15   return value;
16
17 }
18
19 /*=============================================================*/
20 /*  メイン関数                                                 */
21 /*=============================================================*/
22 int main(){
23   int n;
24
25   printf("階乗を計算します．値を入れてください\n");
26   scanf("%d",&n);
27
28   printf("%d!=%d\n",n,kaijyo(n));
29
30   return 0;
31 }
```

リスト 4: 再帰呼び出しを使った階乗を計算するプログラム

```
1  #include <stdio.h>
2
3  /*=============================================================*/
4  /*  階乗を計算する関数                                         */
5  /*=============================================================*/
6  int kaijyo(int n){
7
8    if(n==0){
9      return 1;
```

```
10    }else{
11      return n*kaijyo(n-1);
12    }
13
14  }
15
16  /*═══════════════════════════════════════════════*/
17  /*   メイン関数                                  */
18  /*═══════════════════════════════════════════════*/
19  int main(){
20    int n;
21
22    printf("階乗を計算します．値を入れてください\n");
23    scanf("%d",&n);
24
25    printf("%d!=%d\n",n,kaijyo(n));
26
27    return 0;
28  }
```

3.2 練習問題

[**問 1**] 次に示す数列 (フィボナッチ数列) の F_7 の値は，いくつか?．値のみならず，計算過程も示せ．

$$F_k = F_{k-1} + F_{k-2} \qquad F_0 = 0 \qquad F_1 = 1$$

[**問 2**] フィボナッチ数列を計算するための，再帰呼び出しを使った関数を作成せよ．

[**問 3**] n の階乗[2]を再帰呼び出しで計算するための関数 $F(n)$ を以下に示すが， $\boxed{}$ の内容を選択肢から選べ．

$$F(0) = 1 \qquad F(n) = \boxed{}$$

選択肢

$$F(n) \times F(n-1) \qquad F(n-1) \times F(n-2)$$
$$n \times F(n-1) \qquad (n-1) \times F(n)$$

[**問 4**] n の階乗を計算するための，再帰呼び出しを使った関数を作成せよ．

4 ツリー構造

4.1 ツリー構造とは

同じような複数のデータを表す場合，配列やリスト，スタック，キューのようなデータ構造を使うことを学習してきた．これらのデータ構造では，階層をもつデータの集まりを表すことは，困難である．このよ

[2] n は非負の整数で，n の階乗を $n!$ と書く．$n! = n(n-1)(n-2)\cdots 2 \times 1$ である．

うに階層構造をもつデータを処理するときに威力を発揮するのがツリー構造である．ツリーとは，tree(木)のことである．

ツリー構造にはいろいろ名称があり，それを表 1 と図 4 に示す．なぜ，図 6 のようなデータ構造をツリー構造と呼ぶか?．それは，この図を反対にしてみるのである．すると，根が一番下にきて，枝や葉が上にあることが分かるであろう．まさに，木である．

表 1: ツリー構造の名称

構成要素	内容
ルート (root:根)	最上位のノード
ノード (node:節)	枝が分かれるところ
リーフ (leaf:葉)	子がないノード
ブランチ (branch:枝)	親と子を結ぶ線
親 (parent)	上のノード
子 (child)	下のノード
兄弟 (brother)	同じ親を持つノード

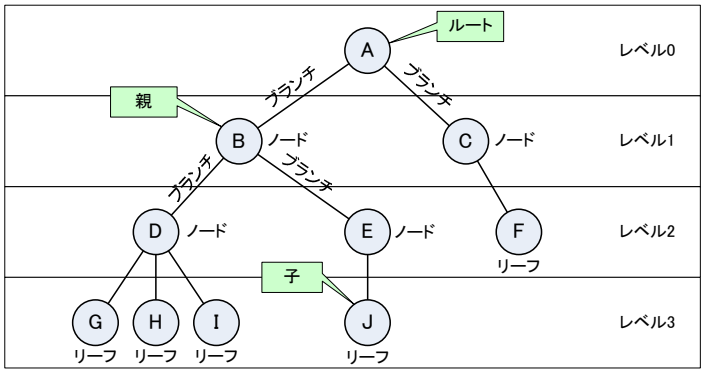


図 4: ツリー構造．図中の親子関係はノード E を基準にしている．

4.2 2 分木

木構造のうち，各ノードの子どものノードが最大 2 つのようなものを 2 分木と言う．そして，ある規準に従ってその 2 分木が作られている場合，2 分探索木 (binary search tree) と呼ばれる．この構造のあるノードの左側と右側はそれぞれの子を頂点とする木構造になる．この子孫に対しても，同じ規準が適用されれるので，再帰による処理が可能となる．

特に有用な 2 分探索木は，大小関係を表すもので，

- 左側の子孫は，自分より必ず小さい．

- 右側に子孫は，自分より必ず大きい．

である．図??は，2 分探索木になっている．

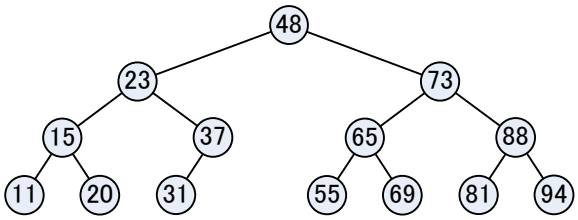


図 5: ツリー構造 (2 分探索木)

4.3 ノードを表す構造体

図 6 が 2 分木のツリー構造である．一つのノードは，データとその 2 つの子を表すポインターからなる．このように複数の異なる型のデータから成る情報を表すためには，構造体を用いる．その構造体をリスト 5 に示す．

このツリー構造を使うためには，型宣言として以下のものが必ず必要である．

ノードを表す構造体 データと子を示すポインターから構成され，ノードを表す．

ルートを表すポインター ツリー構造のデータをたどるために，ルートを示すポインターが必ず必要である．

リスト 5: 2 分木のノードを表す構造体 (教科書 List 6-1)

```
1 typedef struct _tag_tree_node
2 {
3     /* このノードが保持する値 */
4     int value;
5     /* 自分より小さい値の node：図では左側のノード */
6     struct _tag_tree_node *left;
7     /* 自分より大きい値の node：図では右側のノード */
8     struct _tag_tree_node *right;
9 } tree_node;
```

ツリーはノードのみならず，ルートを表すポインターが必要である．教科書の List 6-4(p.176) のように，

```
tree_node *tree_root=NULL;
```

とそのポインターを宣言しておく．もちろん，これはノードを表す構造体のよりも後で宣言する必要がある．先に宣言すると，tree_node という型がコンパイラーが分からないからである．また，初期値は意味のないポインター (NULL) を入れておく．最初はルートがないため，それを表すポインターは意味が無いためである．

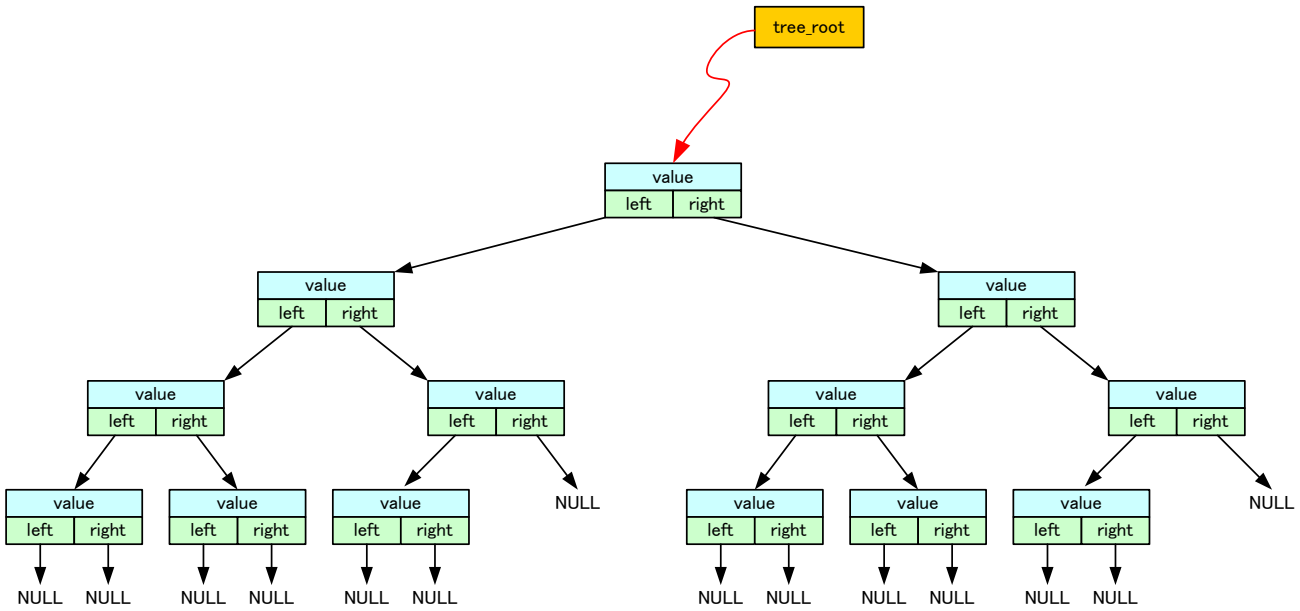
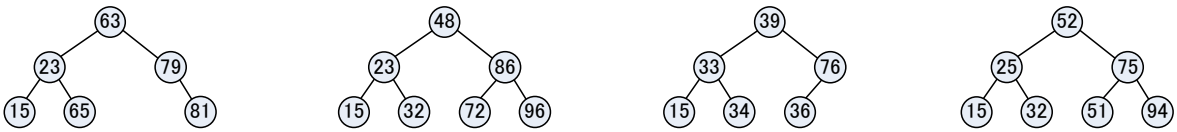


図 6: 教科書 List6-1〜List 6-4 のツリー構造

4.4 練習問題

[**問 1**] 図の中で 2 分探索木になっているものはどれか?



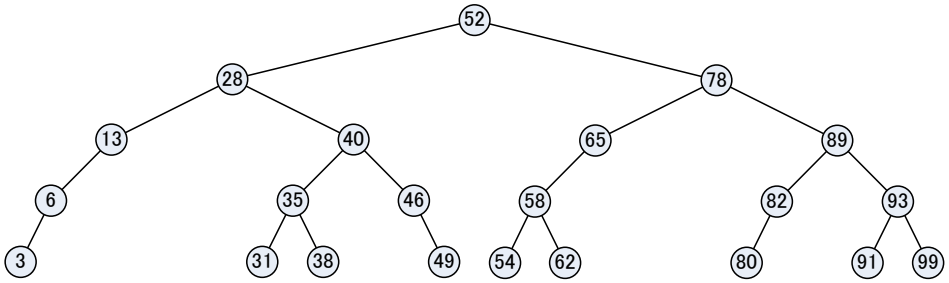
(ア) (イ) (ウ) (エ)

[**問 2**] 最初はデータが無く，以下の並びでデータが追加される．2 分探索木を作成せよ．

69,26,36,45,89,65,11,12,14,23,44

[**問 3**] 図の 2 分探索木にデータを追加する．以下の問いに答えよ．

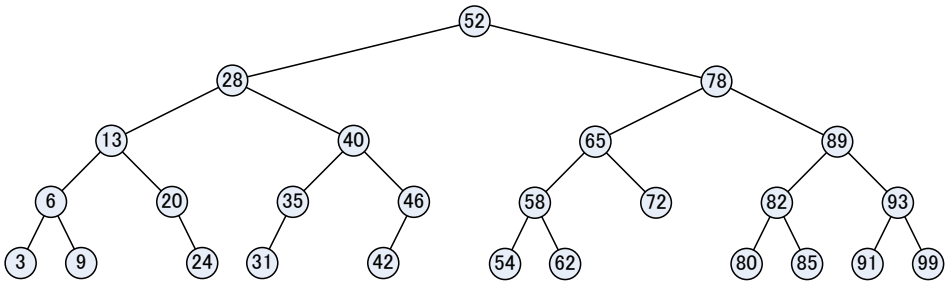
70,73,67,75,77,66 と順に追加する．



データを追加する 2 分木

[**問 4**] 図の 2 分探索木にデータを削除する．以下の問いに答えよ．

`52,46,54` と順に削除する．



データを削除する 2 分木

[**問 5**] 2 分木のノードを示す構造体を書け．

5　浮動小数点と数値計算

5.1　実数演算の誤差

実数では有限のビット数で表現されるため，2 進数での桁数が無限に必要な場合はメモリー中のデータには誤差が含まれる．この誤差を**丸め誤差** (rounding error) と言う．

C 言語の倍精度実数型の精度は，およそ 10^{-16} である．この精度よりも大きさの差 (絶対値) が小さい 2 つの実数の和と差の演算はできない．加算や減算を行っても，その小さい値は無視される．これを**情報落ち**と言う．

三角関数を計算する場合，以下のような級数を使う．

$$\begin{aligned}\sin x &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!} + \cdots \\ &= \sum_{n=1}^{\infty} (-1)^{n+1} \frac{x^{2n-1}}{(2n-1)!} \end{aligned} \tag{5}$$

コンピューターでは無限級数を計算することはできない．これをきちんと計算するためには，無限界の計算回数が必要で，無限の時間がかかる．それに，精度が有限であるため，ある程度以下の計算は無意味である．そのため，精度内に計算が収まったら，計算を止めるようになっている．途中で計算を打ち切るので，この誤差を**打ち切り誤差**と言う．

5.2 二分法

二分法の考え方は単純であるが，非常に強力な方程式の近似解を求める方法である．考え方の基本は，閉区間 $[a, b]$ で連続な関数 $f(x)$ の値が，

$$f(a)f(b) < 0 \tag{6}$$

ならば，$f(\alpha) = 0$ となる α が区間 $[a, b]$ にある．これは，中間値の定理から保証される．こんなことを言わないまでもあたりまえである．ただ，連続な区間で適用できることを忘れてはならない．

実際の数値計算は，$f(a)f(b) < 0$ であるような 2 点 $a, b(a < b)$ から出発する．そして，区間 $[a, b]$ を 2 分する点 $c = (a + b)/2$ に対して，$f(c)$ を計算を行う．$f(c)f(a) < 0$ ならば b を c と置き換え，$f(c)f(a) > 0$ ならば a を c と置き換える．絶えず，区間 $[a, b]$ の間に解があるようにするのである．この操作を繰り返して，区間の幅 $|b - a|$ が与えられた値 ε よりも小さくなったならば，計算を終了する．解へ収束は収束率 1/2 の一次収束という．

リスト 6 に実際のプログラム例を示す．このプログラムと先の説明で用いた変数との対応は，次の通りである．

$a \rightarrow$ `left`
$b \rightarrow$ `riht`
$c \rightarrow$ `mid`
$\varepsilon \rightarrow$ `epsilon`

リスト 6: 二分法のプログラム

```
1  #include <stdio.h>
2  #include <math.h>
3  #include <stdlib.h>
4
5  double func(double x)
6  {
7      return x*cos(x)+sin(x)+exp(-x*x)+x*x-x-3;
8  }
9
10
11 double BinarySearch(void)
12 {
13     double left,mid,right,epsilon;
14
15     epsilon=0.00001;
16
17     left=-2.0;
18     right=2.0;
19
20     while(fabs(right-left)>epsilon)
```

```
21      {
22          mid=(left+right)/2.0;
23
24          if(func(left)*func(mid)>=0.0)
25              left=mid;
26          else
27              right=mid;
28      }
29      return left;
30  }
31
32  int main(void)
33  {
34      double d;
35      d=BinarySearch();
36      printf("方程式の解は%lf, "
37              "そのときのfunc(x)は%lfです。\n",d,func(d));
38      return EXIT_SUCCESS;
39  }
```

5.3 練習問題

[問 1] C 言語の実数型を使った演算で生じる 3 つの誤差の名前を書け.

[問 2] 2 分法 (バイナリーサーチ) を用いて，方程式の近似解を求める原理を説明せよ.

参考文献

[1] 紀平拓男, 春日伸弥. プログラミングの宝箱　アルゴリズムとデータ構造. ソフトバンクパブリッシング (株), 2004 年.