プログラミング通論（Ｉ類３）2020

久野　靖
電気通信大学

2020.2.21 改訂

目 次

＃1　C言語の基本機能

今回は初回ですが、次のことが目標となります。

- C 言語の基本的な機能について復習し、関数・変数・代入・制御構造を用いたプログラミングができることを確認する。

ただしその前にガイダンスから始めて、その後本題に入ります。

1.1　ガイダンス

1.1.1　本科目の主題・目標

本科目の主題ならびに達成目標は次のようになっています。

主題: プログラミングの初歩は学習したという前提で、再帰的手続き、データ構造の初歩、および、基本的アルゴリズムについて学習する。

達成目標: 再帰的手続き、基本的なデータ構造、基本的なアルゴリズムを理解し、それらを用いた C 言語のプログラムを読むことと、書くことができる。

本学では授業 1 単位について 45 時間の学修を必要とすることとなっています。本科目は 2 単位ですから 90 時間となります。これを 15 週で割ると週あたり 6 時間となります。授業そのものは 90 分 (1.5 時間) であるので、時間外に 4.5 時間の学修が必要です。課題等もこのことを前提に用意されていますので、留意してください。

1.1.2　本科目の運用

本科目の運用ですが、各時間の内容は前もってテキストを公開しますので、予習してくるようにお願いします。授業時間にはその要点だけ説明し、あとは演習をしていただきます。

各回とも、出席相当の課題として「プログラム 1 つを作成し報告する当日一杯締切の課題 (A 課題)」と、プログラム 2 つ以上を作成し報告する翌週授業前日一杯締切の課題 (B 課題)」を課します。これは、演習してプログラムを書かない限りプログラミングは身につかないからです。[1]

各回の課題レポートは CED システム上の提出プログラムで提出していただきます (学外からでも sol を経由して CED システムに入って作業できますし、提出だけなら sol でもできます)。レポートの内容は互いに見られるように学内限定で公開します。従って、レポートには公開されたくないもの (個人の情報など) は書かないでください。名前を書かないわけにはいかないので、行頭が「`@@@`」となっている行は除外して公開します。氏名等を記入した行は「`@@@`」で隠すことを勧めます。[2]

すべての課題レポートは× (未提出ないしそれと同等)、△ (要件を満たしていないか遅刻)、○ (通常点=要件を満たしたレポートである)、◎ (特に見るべきところがある) の 4 段階で評価します。全レポートが○のときがレポート点の満点であり、◎は上積み (または減点の相殺) となります。50 点を超えた部分は係数 α ($0 < \alpha \leq 1$、期末時に調整) を掛けて、59 点を上限としてから試験点と加算します。

[1] このほかにも、プログラミングが身に付くようにするためのさまざまな工夫を本科目中に盛り込んでいます。
[2] 必要な箇所のみ隠すこと。すべて隠そうとして山のように「`@@@`」をつけた場合は「`@@@`」を全て削除します。

レポートは必ず個人単位で出して頂き、個人単位で採点します (試験も)。複数人から同一内容ないしコピーと思われるレポートが出て来た場合、両方とも減点します (後述するペアでプログラムが同一なのは許容します)。

試験は期末試験のみで、プログラムを書く試験 (並べ替え式を含む) を課す予定です。試験点も他のクラスと SABCD の比率が乖離しないよう調整してから加算します。テキスト内容のうち表題に option と記されている節は試験範囲となりません。あと、テキストに載っていなくてもプログラミングの基本部分はいずれにせよ含まれるものと考えてください。

1.1.3　ペアプログラミング

本科目では「ペアプログラミング」を推奨します。これは次のようなものです。

- 1 つの画面の前に 2 人で座り、一人がキーボードを持ちプログラムを打つ。もう 1 人はそれを一緒に眺めて意見やコメントや考えを述べる。キーボードの担当者は適宜交替してもよい。

このような方法がよい理由としては、次のものがあげられます。

- プログラムを作って動かすのには多くの緻密で細かい注意が必要ですが、1 人でやるより 2 人でやる方がこれらの点が行き届き、無用なトラブルによる時間の空費が避けられます。
- プログラミングではたまたま「簡単な知識」が足りなくて、それを調べて使うまでにすごく時間が掛かることがありますが、2 人いればそのような知識を「どちらかは知っている」可能性が高まり、時間の無駄が省けます。
- プログラミング的な考え方を身に付けるには、さまざまな方面から思考したり、それを身体的な活動と結びつけることが有効です。2 人で互いに議論することで、思考が活発になり、多方面にわたるアイデアが出やすくなるため、上記のことがらに貢献します。

課題提出に際してはもちろん、2 人で作ったものですから、その 2 人については同一のプログラムを出して頂いて構いません。ただし次の条件があります。

- 提出するレポートにおいて、互いに「誰がペアであるか (相手の学籍番号)」を明示する。個人的な好みや都合よりペアを組まずに作業することも認めますので、そのときは「個人作業」と記してください。
- 「当日課題 (A 課題)」と「翌週までの課題 (B 課題)」でペアを変更したり、1 人で作業との間で変更しても構わない。ただし課題の「途中で」は変更しないこと。たとえば B 課題をペアでやる場合は、時間外もプログラミングについてはすべて 2 人で時間を合わせて作業すること。

ペアで複数のプログラムを作った場合、どれを提出するかは各自で選んで構いません。

1.1.4　レビュー課題

A 課題 (授業当日の課題) レポートにおいて、自分 (達) が作成したプログラムの 1 つを「ペア以外の誰か」に見せて説明し、ひとこと (1 行、30 文字程度) コメントをもらってください。それを提出して頂きます (レビュー課題)。授業時にクラスメートに見せてコメントをもらうのが簡単だと思いますが、事後に先輩、後輩、家族など誰にもらうのでも構いません。ただし「すごいねえ」「がんばったね」などのプログラムの中身と関係ない (または抽象的な) コメントではなく、「プログラムのどこのところが、どうである (よい、悪い、このように工夫されているなど)」という形のコメントであること。

この課題を課す意図ですが、プログラムを他人に説明するすることは自分でプログラムを客観化して見る大変よい練習になるからです。面倒だと思うでしょうが、有用なので必ずお願いします。

1.1.5 担当教員との連絡

資料や皆様のレポートを掲載する授業サイトは次の場所となります。

http://www.edu.cc.uec.ac.jp/~ka002689/prog20/

ここに掲示も出しますので、定期的に (授業期間中は毎日 1 回以上) チェックしてください。掲示を確認しなかったことによる不利益は救済しません。久野のメールアドレスは y-kuno@uec.ac.jp ですので、個別の質問等はこちらにお願いします。では、よろしくお願いします。

1.2 C 言語の基本機能

1.2.1 プログラムの構造

初回はウォームアップとして、C 言語の基本機能をざっと整理し、それらに基づいた演習をやりましょう。具体的には、プログラムの構造、型、関数、変数、計算式、代入、制御構造、入出力などの内容について整理することから始めます。

まず確認ですが、皆様は初心者ではないはずなので、次のようなプログラムは読み書きできるものとして扱います。[3]

```
// triarea --- area of a triangle
#include <stdio.h>
double triarea(double w, double h) {
  double s = (w * h) / 2.0;
  return s;
}
int main(void) {
  double w, h;
  printf("w> "); scanf("%lf", &w);
  printf("h> "); scanf("%lf", &h);
  double x = triarea(w, h);
  printf("area of triangle = %g\n", x);
  return 0;
}
```

ここでまず、C 言語のプログラムは「関数の集まり」であるということを指摘します。**関数** (function) は実行の単位であり、「(必要ならパラメタを付して) 呼び出され、内部で計算を実行し、終わったら呼び出された箇所に戻る (値を返すこともできる)」のでした。そして、プログラムはまず main が呼び出されてそこから始まり、main の実行が戻ると終了します (図 1.1)。

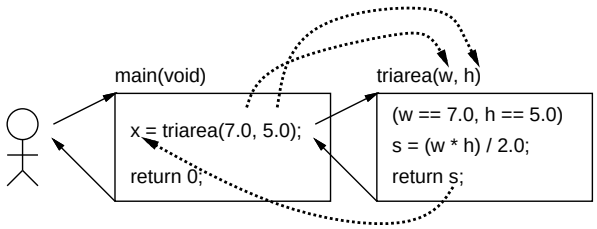


図 1.1: 関数の呼び出しと戻り

このほか、次のことも重要です。

[3] もしできない場合は、できる状態になるまでは、各自で自習してきてください。情報部会のサイト https://joho.g-edu.uec.ac.jp/joho/ に基礎プログラミングおよび演習のテキストとビデオがあります。

- 呼び出し時に渡した値の並び (**実引数**、actual arguments) は、関数の定義冒頭に書かれた変数の並び (**仮引数**、formal argument、パラメタ) に順に対応させられる。図 1.1 の場合、`triarea` に入ったときにそのパラメタ `w` の値は渡された 1 番目の値 7.0 に、そして `h` の値は渡された 2 番目の値 5.0 になっている。
- **return** 文「`return 式;`」が実行されると、直ちにその関数の実行を終わって読んだ箇所に戻る。式の値は、関数の値となるので、それを計算に使ったり変数に代入できる。返値の型が `void` と指定された関数の場合は値を返さないので式も指定しないが、直ちに実行を終わることは同じである。なお、`main` は呼び出し元に整数を返すことになっており、正常終了のときは 0 を返しそうでない解きは 0 以外を返す約束になっている。[4]
- コンパイラは引数の個数と型、および返値の型が合っていることをチェックするが、C 言語ではその際「それまでに処理したソースコードの情報だけを使う」という制約がある。このため、上のように「関数定義が先にあり、その後で呼び出す箇所がある」場合は大丈夫だが、関数 `main` を先に書くなら、その前に次のような `triarea` の**プロトタイプ宣言** (prototype declaration) を置く必要がある。

```
double triarea(double w, double h);
```

プロトタイプ宣言は、関数の本体部分「{ … }」を削除し、代わりに「;」を置いたものである。

1.2.2　基本概念の復習

例題に出て来るそのほかの概念をひととおり説明します。

- C 言語で扱う**基本型** (primitive type) として `char`(1 バイト==8 ビットの整数)、`int`(整数)、`long`(ビット数の多い整数)、`float`(実数)、`double`(倍精度実数) がある。整数は符号つき (2 の補数表現) だが、**unsigned** という修飾を前につけると符号なし整数にできる。
- 値を返さない関数は **void** 型として定義/宣言する。関数のパラメタが 0 個の場合はパラメタ部に `void` と書く (上の例の `main` がそう)。
- 変数はすべて**宣言** (declaration) してから使う必要がある。宣言は「型指定 変数,…;」の形をしている。それぞれの変数の後に「=式」をつけることで、宣言と同時に**初期値** (initial value) を入れてもよい。
- 「**式** (expression)」は値を計算することを表す。「`3.14`」などの**定数** (literal — その定数の表す値を意味する)、「`x`」などの**変数** (variable — 変数に格納されている値を表す)、および先に述べた「関数呼び出し」(関数が返す値を表す) をもとに、**演算子** (operator) でさまざまな演算をする。演算子には「`+, -, *, /, %`」(加減乗除と剰余)、「`++, --`」(整数変数を 1 増やす/減らす)、「`==, !=, >, >=, <, <=`」(比較)、「`&&, ||, !`」(かつ、または、〜でない)、「`&, |, ~`」(整数をビット列とみなしての and, or, not) などがある。そして「`=`」(**代入**、assignment) も演算子であり、結果として代入した値を返すので、「`x = y = 0`」などとも書ける。
- 関数本体には**文** (statement) の並びを書く。式を文として書くときは「式;」のように「;」をつける必要がある (**式文**、expression statement)。関数呼び出しだけ書く場合も式文になる。上の例では変数宣言と return 文以外はすべて式文である。
- 出力関数「`printf("書式文字列", 式,…)`」は、基本的に**書式文字列** (format string) をそのまま出力するが、その中に「`%`○」という形のもの (書式指定) があると、そこに後ろに指定した式の値を埋め込む。主要な書式指定に「`%d`」(整数を十進で出力)、「`%x`」(16 進で出力)、「`%f`」(実数を小数点つき形式で出力)、「`%e`」(指数形式で出力)、「`%g`」(おまかせで出力)、「`%c`」(文字を出力)、「`%s`」(文字列を出力) がある。「`%8d`」のように出力の幅を指定することもできる。

[4] 実際はこの `main` が返した値を使うことはあまり多くありませんが。

- 入力関数「scanf("書式文字列", 式,…)」は、書式指定に従って入力をおこなう。主要な書式指定に「%d」(整数入力)、「%c」(文字入力)、「%f」(float 入力)、「%lf」(double 入力) がある。対応する式は「変数のアドレス」でなければならない。変数のアドレスを取るのには、**アドレス演算子** (addressing operator)「&」を用いて「&変数名」でできる (そのほかポインタ値やポインタ演算などでもできる)。

1.2.3 論理値の扱い

C 言語には**論理型** (boolean type、はい/いいえを表す型) が無く、int で代用し、「0」が「いいえ」、「1」が「はい」を表します。しかし今日の多くの言語では論理型として「bool」、値として「false」「true」を使うようになっているので、次のような定義をする C プログラマも多数います。

```
#define bool int
#define true 1
#define false 0
```

毎回そうするのも面倒なので、#include <stdbool.h>を書いておくことで上記と同じに使えるようになります。以下でもそのようにします (論理値と整数を分けて考えておく方が間違えにくいので)。

1.2.4 制御構造の復習と文法

関数定義の { … } の中には文の並びが入ります。具体的にどのようなものが書けるかについては、プログラミング言語の**文法** (syntax specification) によって定められています。ここでは文法を読む練習として、「文」の定義を見ましょう (分かりやすさのため簡略化しています)。なお、「$A|B$」は「A または B」、「$[A]$」は「A があってもなくてもよい」を意味します。

```
文 ::= 変数定義 | ; | 式 ; | if 文 | while 文 | for 文
     | return [ 式 ] ; | break ; | continue ; | { 文… }
if 文 ::= if( 式 ) 文 [ else 文 ]
while 文 ::= while( 式 ) 文
for 文 ::= for( [ 変数定義 | 式 ] ; [ 式 ] ; [ 式 ] ) 文
```

「;」だけの文がありますが、これは**空文** (empty statement) といい、何も動作をしない文です。なんでこれが必要かというと、たとえば **if** 文で次のような場合のためです。

```
if(x <= 10)
  ;   // x は 10 以下なので何もしない
else
  printf("x too large: %d\n", x);
```

文法を見ると「if と else の間には 1 つ文がある」となっているので、何も書かないわけにはいかないので、それで空文を入れています。しかしこれまでそんなものは使わないで済んでいたが…そうでしょう。つまり、常に「{ … }」を使えば、それ自体がちょうど 1 つの文ですし、その中には文が 0 個でもよいので、上のようなことを気にしないでよいのです。

```
if(x <= 10) {
  // x は 10 以下なので何もしない
} else {
  printf("x too large: %d\n", x);
}
```

なので、こちらのスタイルを使うことを勧めます。あと、if-else if の連鎖が文法に無いですね? それは「else の直後にすぐ次の if 文を書いた」だけで、それで上の文法に合致しています。

```
if(x > 0) {
  printf("positive.\n");
} else if(x < 0) {
  printf("negative.\n");
} else {
  printf("zero.\n");
}
```

次にループですが、C 言語では **for** 文が **while** 文と同様、任意の条件を指定する文であることが特徴です。たとえば次の 2 つは同じ動作です。

```
i = 0;
while(i < length) { b[i] = a[i]; ++i; }

for(int i = 0; i < length; ++i) { b[i] = a[i]; }
```

1 つだけ小さい違いがあります。それは、ループの中で **continue** 文 (次の周回に進めという命令) を使ったとき、for 文の場合は「`++i`」のところに飛びますが、while 文であれば次の条件テストにすぐ進む、というところです。なお、ループを脱出する **break** 文の場合は両者の差はありません。

あと、for 文の初期化部分には (C99 以降では)「変数定義」が書けます。上の断片でもそうなっています。ただし、ここで定義した変数のスコープはこの for 文の中だけなので注意 (ループを出た後で `i` の値を参照することはできなくなるという意味です)。

では演習に進む前に、2 番目の例題として「`n` を入力し、`n` が素数か否かを出力する」というものを示しましょう。

数学ライブラリを使うので `math.h` を取り込むこととコンパイル時に「`gcc8 prime.c -lm`」のように追加指定が必要です。[5]

```
// prime.c --- see if an integer is a prime.
#include <stdio.h>
#include <math.h>      // sqrt を使う場合必要
#include <stdbool.h>   // true, false, bool を使う場合必要
bool isprime(int n) {
  int limit = (int)sqrt(n);
  for(int i = 2; i <= limit; ++i) {
    if(n % i == 0) { return false; }
  }
  return true;
}
int main(void) {
  int n;
  printf("n> "); scanf("%d", &n);
  if(isprime(n)) { printf("%d is a prime.\n", n); }
  else           { printf("%d is not a prime.\n", n); }
  return 0;
}
```

ある数が素数かどうか調べるには、$2 \sim \sqrt{n}$ までの数で割ってみればよいわけです。**平方根** (square root)「`sqrt(n)`」は当然実数を返すので、実数値を切り捨てて整数に変換するためにキャスト演算「`(int)`」を使用して、結果を変数 `limit` に入れています。

[5] C99 対応の GCC バージョン 8 を入れてもらいましたので、それを使います。コマンドは「`gcc8`」です。

ループでは 2 から初めてその limit まで割り切れるかを調べています。%は剰余演算なので、剰余が 0 なら割り切れたことになります。return 文は実行すると直ちに値を返してその関数を終わるので、ループの途中でやめる (その先を調べないで済ませる) ことができるわけです。最後まで割り切れなければ素数なので「はい」を返します。

演習 1 まず「三角形の面積」「素数判定」を打ち込み動かせ。動いたら次のようなプログラムを作れ。

a. 直角三角形の直角をはさむ 2 辺の長さを入力し、斜辺の長さを出力する。
b. 整数 n を入力し、2^n を出力する。$n \geq 0$ のときは整数、そうでないときは実数形式で出力すること。
c. 整数 n ($n > 1$) を入力し、n の素因数分解を表示する。たとえば 60 を入力したら「2 2 3 5」を出力する (出力の順番や形式は任意)。
d. 整数 n を入力し、n 以下の素数を出力する (出力の順番や形式は任意)。
e. 整数 n を入力し、何らかの数列 (選択は任意) の最初の n 項を出力する。

1.3 PostScript を用いた描画 option

1.3.1 PostScript を用いた描画ライブラリ

数値ばかり扱うのも飽きるので、後半は **PostScript** 言語 (以下 PS) を用いた描画のためのライブラリと API を題材に取り上げます。PS については「コンピュータリテラシ」において取り上げていますので、学んでいない人は詳しくはそちらの資料を見てください。必要最低限のことは以下で説明します。

以下で使い方を説明するライブラリは、BoundingBox 指定を持った PS 形式 (**EPS**、Encapsulated PostScript) を出力するためのものです。PS そのものが図形を記述する機能を用意していますが、そのままだと使い慣れないとやりにくいので、その上に C 言語むけ API をかぶせて使いやすくしています (細かいことは直接 PostScript の命令を使って指定します)。

API を使うためのヘッダファイル eps.h を示します (実装部分 eps.c は付録に掲載しました。またその中身は読めば読めると思うので細かくは説明していません)。

```
// eps.h --- eps library API
void eps_open(char *fname, int w, int h);
void eps_close(void);
void eps_cmd(char *cmd);
void eps_num(double val);
void eps_drawline(double x0, double y0, double x1, double y1);
void eps_drawrect(double x, double y, double w, double h);
void eps_fillrect(double x, double y, double w, double h);
void eps_drawcircle(double x, double y, double r);
void eps_fillcircle(double x, double y, double r);
int eps_newfont(char *font, double size);
void eps_puts(int id, double x, double y, char *s);
```

API に含まれる関数の機能は次の通りです。

- eps_open — 出力するファイル名と描画面の大きさを指定する。単位は pt (1pt = $\frac{1}{72}$inch)。A4 版の紙に収まるには「600 × 850」程度までにするのがよい。
- eps_close — 描画を終わる (ファイルを完成させる)。
- eps_cmd — 任意の PS コマンドをそのまま出力。PS の細かい機能を使いたいときは基本的にこれを使う。

- eps_num — PS コマンドの中に C 言語で計算した数値を混ぜたいとき、その数値を出力するのに使う。
- eps_drawline、eps_drawrect、eps_fillrect、eps_drawcirlce、eps_fillcircle — 指定した座標間の線分を描く、または指定した位置と大きさの長方形/ 円を描く/塗りつぶす。長方形の指定は「左下隅の XY 座標、幅、高さ」、円の指定は「中心の XY 座標、半径」でおこなう。
- eps_newfont — 文字を描画するのに使うフォントを準備する。フォント名とポイント数を指定する。PS で標準で (必ず) 使えるフォントは Times-Roman, Times-Italic, Times-Bold, Times-BoldItalic, Helvetica, Helbetica-Oblique, Helvetica-Bold, Helvetica-BoldOblique, Courier, Courier-Oblique, Courier-Bold, Courier-BoldOblique がある (Italic、Oblique は斜体を表す)。結果として「フォント番号」が返され、その番号を eps_puts で使う。
- eps_puts — フォント番号、XY 座標、文字列を指定することで、指定位置に指定した文字を指定したフォントで描画する。

1.3.2　PostScript の主要なコマンド

参照に便利なように、直接使いそうな PS のコマンドを示しておきます。

- gsave、grestore — グラフィクスの状態を保存/ 復元する。以下で説明する座標変換、色や線幅の変更を行なったあと、元に戻すには、gsave しておいてから変更を行ない、戻したいところで grestore する。
- $R\ G\ B$ setrgbcolor、$H\ S\ S$ sethsbcolor — 描画に使う色を RGB または HSB で指定。これらの値は 0.0-1.0 までの実数で、RGB の場合は赤/緑/青の強さ、HSB の場合は色相/ 彩度 (鮮やかさ)/明度 (明るさ) を意味する。
- B setgray — 描画に使う色 (白黒) を明るさで指定。B は 0.0-1.0 までの実数で、真っ黒から真っ白までの間の灰色を指定する。
- $T_x\ T_y$ translate、$S_x\ S_y$ scale、D sotate — 描画の座標の変換。原点を (T_x, T_y) に移動、X および Y 方向に S_x および S_y 倍、原点を中心に反時計回りに D 度回転がおこなえる
- W setlinewidth — 線幅を変更。単位は pt。

1.3.3　eps ライブラリの使用例

それでは実際にライブラリを使った例を示しましょう。このプログラムで描画した結果は図 1.2 なので、照合しながら見てください。

```
// testeps.c --- demonstration of eps library.
#include <stdio.h>
#include "eps.h"

int main(void) {
  eps_open("out.ps", 480, 480);
  eps_cmd("240 240 translate");
  eps_drawline(-200, 0, 200, 0);
  eps_drawline(0, 200, 0, -200);
  for(int i = 1; i <= 8; ++i) {
    eps_num(i*0.1); eps_cmd("setgray");
    eps_cmd("4 setlinewidth");
    eps_drawrect(i*20, i*20, 30, 30);
    eps_num(i*0.1); eps_cmd("1.0 1.0 sethsbcolor");
```

```
    eps_fillcircle(-i*20, -i*20, 15);
  }
  int f1 = eps_newfont("Courier", 20);
  eps_puts(f1, -180, 50, "This is a pen.");
  int f2 = eps_newfont("Helvetica", 30);
  eps_puts(f2, 20, -50, "How are you?");
  eps_close();
  return 0;
}
```

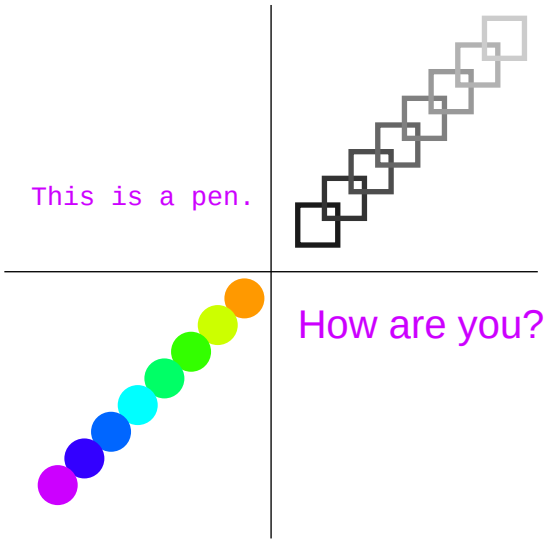


図 1.2: eps 例題プログラムの出力

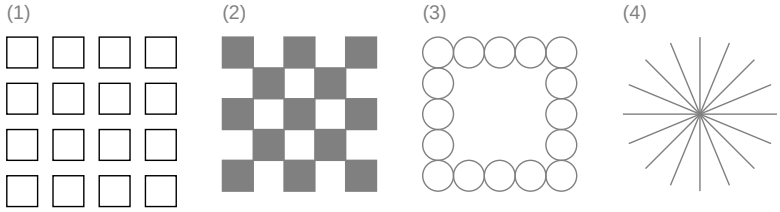
コメントで書いた通り、さまざまな機能をひととおり呼んでいます。色の設定と線幅の設定は PS のコマンドを直接出力しますが、そのときのパラメタを C の変数で設定する箇所は eps_num を用いて指定しています。

これを動かすには、このプログラムのファイル (たとえば testeps.c) に加えて、eps.h と eps.c が必要です。動かすコマンドは次のようになります。

```
% gcc8 testeps.c eps.c      ←必要に応じて -lm なども指定
% ./a.out                   ←このプログラムは入力なし
% gv out.ps                 ←画面で直接見る場合。または
% convert out.ps out.pdf    ← PDF やその他の形式に変換し利用
```

演習 2 上の例題をそのまま動かせ。動いたら色をや図形の位置、個数などを変更して結果を確認すること。それができたら、以下のプログラムを作れ。

a. 次の図のような絵を作成する。色や線幅は自由にしてよい。



b. 数学関数 (sin、cos、tan など)、多項式による関数 ($y = x^2 - 4$、$y = \frac{1}{3}(x^3 - x)$ など)、その他好きな関数を 1 つ選びグラフを描画する (細かい折れ線で近似する)。
c. お天気サイト等から自分の居住地 (または出身地) に関する何らかのデータ (月毎平均気温、月毎降水量、月毎の晴れ日数など) を取得し、分かりやすいグラフにする。
d. その他 eps ライブラリを利用して面白い描画をおこなう。

1.4　トレーニング課題

2019 年度の実施で、極めて基本的な C 言語プログラムでの練習が必要な学生さん「も」いるということが明らかになったので、今年度から「トレーニング課題」を追加します。トレーニング課題は基本練習の易しい課題で、「トレーニング課題 10 個で」通常の課題 1 個の代わりとして提出することができます。B 課題は通常の課題 2 個でも、通常 1 個トレーニング 10 個でも、トレーニング 20 個でも構いません。A 課題をトレーニング課題のみで出す場合、レビュー課題は 10 個見てもらってください。

演習 T3 正の整数 n を入力し、1〜n までの整数を出力する例題を示す。

```
#include <stdio.h>
int main(void) {
  int n;
  printf("n> "); scanf("%d", &n);
  for(int i = 1; i <= n; ++i) { printf(" %d", i); }
  printf("\n");
  return 0;
}
```

これを参考に (しなくてもよいが)、次のプログラムを作成せよ。

a. 正の整数 n を入力し、0 から n-1 までの整数を出力。
b. 正の整数 n を入力し、1, 3, 5, ... と n 未満の整数を出力。
c. 正の整数 n を入力し、0, 2, 4, ... と n 未満の偶数を出力。
d. 正の整数 n を入力し、0, 3, 6, ... と n 以下の 3 の倍数を出力。
e. 正の整数 n を入力し、n 〜 0 の範囲の整数を大きい順に出力。
f. 正の整数 n を入力し、-n 〜 n の範囲の整数を小さい順に出力。
g. 正の整数 n を入力し、n 〜 -n の範囲の整数を大きい順に出力。

演習 T4 正の実数 x を入力し、x〜10x を出力する例題を示す。

```
#include <stdio.h>
int main(void) {
  double x;
  printf("x> "); scanf("%lf", &x);
  for(int i = 1; i <= 10; ++i) { printf("%8.2f\n", i*x); }
  return 0;
}
```

これを参考に (しなくてもよいが)、次のプログラムを作成せよ。

a. 正の実数 x を入力し、x^1〜x^{10} を出力。

b. 正の実数 x を入力し、x^0〜x^9 を出力。
c. 正の実数 x と正の整数 n を入力し、x^0〜x^n を出力。
d. 正の実数 x と正の整数 n を入力し、x^{-1}〜x^{-n} を出力。
e. 正の実数 x と正の整数 n を入力し、$\frac{x}{1}$〜$\frac{x}{n}$ を出力。
f. 正の実数 x と正の整数 n を入力し、x, $2x$, ..., nx を出力。
g. 正の実数 x と正の整数 n を入力し、x, $x+1$, ..., $x+n$ を出力。
h. 正の実数 x と正の整数 n を入力し、x, $1-x$, ..., $n-x$ を出力。

演習 T5 正の整数 n を入力し、1〜n を出力するが 3 の倍数のときは代わりに fizz と出力する例を示す。

```
#include <stdio.h>
int main(void) {
  int n;
  printf("n> "); scanf("%d", &n);
  for(int i = 1; i <= n; ++i) {
    if(i % 3 == 0) { print(" fizz"); } else { printf(" %d", i); }
  }
  printf("\n");
  return 0;
}
```

これを参考に (しなくてもよいが)、次のプログラムを作成せよ。

a. 正の整数 n を入力し、1〜n を出力するが 5 の倍数のときは代わりに buzz と出力。
b. 正の整数 n を入力し、1〜n を出力するが 3 の倍数のときは fizz、5 の倍数のときは buzz、3 と 5 の公倍数では fizzbuzz と代わりに出力。
c. 正の整数 n を入力し、1〜n のうち 2 の倍数でも 3 の倍数でもないものだけを出力。
d. 正の整数 n を入力し、1〜n を出力するが偶数はマイナス符号をつけて出力。
e. 正の整数 n を入力し、1〜n を出力するが奇数は同じものを 2 個出力。
f. 正の整数 n を入力し、1〜n を出力するが 3 の倍数は同じものを 3 個出力。

演習 T6 正の整数 n を入力し、1〜n を、i 番目の数は i 個出力する例を示す。

```
#include <stdio.h>
int main(void) {
  int n;
  printf("n> "); scanf("%d", &n);
  for(int i = 1; i <= n; ++i) {
    for(int j = 1; j <=i; ++j) { printf(" %d", i); }
  }
  printf("\n");
  return 0;
}
```

これを参考に (しなくてもよいが)、次のプログラムを作成せよ。

a. 正の整数 n を入力し、n〜1 を数 i は i 回繰り返し出力。

b. 正の整数 n を入力し、n〜1 を i 番目の数は i 回繰り返し出力。
c. 正の整数 n を入力し、i〜n を奇数は 1 回偶数はその数の回数繰り返し出力。
d. 正の整数 n を入力し、1 は n 回、2 は n-1 回、…、n は 1 回繰り返し出力。
e. 正の整数 n を入力し、1〜n をすべて n 回ずつ出力。
f. 正の整数 n を入力し、1 行目に-1…1、2 行目に-2…2, ...n 行目に-n…n を出力。

本日の課題 1a

「演習 1」「演習 2」で動かしたプログラム 1 つを含むレポートを本日中 (授業日の 23:59 まで) に提出してください。

1. sol または CED 環境で「`/home3/staff/ka002689/prog20upload 1a` ファイル名」で以下の内容を提出。文字コード UTF8。「ファイル名」の代わりに「`-show`」と指定すると提出内容が確認できる。以後毎回同様。
2. 学籍番号、氏名、ペアの学籍番号 (または「個人作業」)、提出日時。名前の行は先頭に「@@@」を付けることを勧める。
3. プログラムどれか 1 つのソースと「簡単な」説明。
4. レビュー課題。提出プログラムに対する他人 (ペア以外) からの簡単な (ただしプログラムの内容に関する) コメント。
5. 以下のアンケートの回答。

   Q1. C 言語のプログラミングは好き/嫌いどちら? 理由は?
   Q2. eps ライブラリについてどのように思いましたか。
   Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

次回までの課題 1b

「演習 1」「演習 2」(ただし 1a で提出したものは除外、以後も同様) の小課題全体から選択して 2 つ以上プログラムを作り、レポートを提出しなさい (図形やグラフなどを 2 つ作って 2 個ということでもよい)。できるだけ演習 2 からも選ぶこと。レポートは次回授業前日 23:59 を期限とします。

1. sol または CED 環境で「`/home3/staff/ka002689/prog20upload 1b` ファイル名」で以下の内容を提出。
2. 学籍番号、氏名、ペアの学籍番号 (または「個人作業」)、提出日時。名前の行は先頭に「@@@」を付けることを勧める。
3. 1 つ目の課題の再掲 (どの課題か分かればよい)、プログラムのソースと「丁寧な」説明、および考察 (課題をやってみて分かったこと、分析、疑問点など)。
4. 2 つ目の課題についても同様。
5. 以下のアンケートの回答。

   Q1. プログラムを作るという課題はどれくらい大変でしたか?
   Q2. eps ライブラリを使ってみてどのように感じましたか。
   Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

1.5 付録: eps ライブラリの実装

eps ライブラリの実装はそれほど大したものではないので、掲載しておきます。PostScript の細かい機能を使っていますが、それはここでは説明しません。適宜調べてください。C 言語関係でまだ学んでいない機能としては、任意のファイルへの書き出しを使います。その概略は次の通り。

```
FILE *fd = fopen("ファイル名", "wb"); ←ファイルを書き出しオープン
...
fprintf(fd, "書式文字列", 引数…);  ← printf と類似。何回も使う
...
fclose(fd);                    ←後しまつ。最後に必ず呼ぶ必要あり
```

この API は実は stdio.h で宣言ずみです。fopen はファイルを「開く」(読み書きの準備をすることをそう呼ぶ) 操作で、ここでは書き出しモードとします (wb は write binary)。返値はファイル型のポインタ値で、その中身は知らなくてもよいです (fprintf や fclose にそのまま渡すだけなので)。

fprintf は printf そっくりですが、ただし最初の引数としてファイル型のポインタ値を受け取ることで、その対応するファイルへの書き出しを行なうところが違います。fclose はファイルを「閉じる」(処理を終了する) もので、これによりファイルが完成します。

```
// eps.c --- eps library implementation
#include <stdio.h>
#include "eps.h"
static FILE *fd = NULL;
static int fontid = 0;

void eps_open(char *fname, int w, int h){
  fd = fopen(fname, "wb");
  fprintf(fd, "%%!PS-Adobe-2.0\n%%%%BoundingBox: 0 0 %d %d\n", w, h);
}
void eps_close(void) {
  fprintf(fd, "showpage\n"); fclose(fd); fd = NULL;
}
void eps_cmd(char *cmd) {
  fprintf(fd, "%s\n", cmd);
}
void eps_num(double val) {
  fprintf(fd, "%.2f ", val);
}
void eps_drawline(double x0, double y0, double x1, double y1) {
  fprintf(fd, "newpath %.2f %.2f moveto %.2f %.2f lineto stroke\n",
          x0, y0, x1, y1);
}
static void rect(double x, double y, double w, double h) {
  fprintf(fd, "newpath %.2f %.2f moveto %.2f 0 rlineto\n", x, y, w);
  fprintf(fd, " 0 %.2f rlineto %.2f 0 rlineto closepath\n", h, -w, -h);
}
void eps_drawrect(double x, double y, double w, double h) {
  rect(x, y, w, h); fprintf(fd, "stroke\n");
```

```
}
void eps_fillrect(double x, double y, double w, double h) {
  rect(x, y, w, h); fprintf(fd, "fill\n");
}
static void circle(double x, double y, double r) {
  fprintf(fd, "%.2f %.2f %.2f 0 360 arc closepath\n", x, y, r);
}
void eps_drawcircle(double x, double y, double r) {
  circle(x, y, r); fprintf(fd, "stroke\n");
}
void eps_fillcircle(double x, double y, double r) {
  circle(x, y, r); fprintf(fd, "fill\n");
}
int eps_newfont(char *font, double size) {
  fprintf(fd, "/%s findfont %.2f scalefont /font%d exch def\n",
          font, size, ++fontid);
  return fontid;
}
void eps_puts(int id, double x, double y, char *s) {
  fprintf(fd, "font%d setfont %.2f %.2f moveto (%s) show\n",
          id, x, y, s);
}
```

＃2　アドレスとポインタ

今回は次のことが目標となります。

- アドレスとポインタの概念、および C 言語におけるそれらの扱いを理解する。
- 配列や構造体とポインタの関係について理解し扱えるようになる。

2.1　アドレスとポインタの概念

2.1.1　アドレスと左辺値・右辺値

コンピュータがプログラムを実行するときの主要な構成要素は、データを格納する**主記憶** (memory、メモリ) と、命令を実行し演算をおこなう **CPU**(central processing unit) です。そして図 2.1 のように、メモリには**番地** (address、アドレス) が割り振られていて、番地を指定することでデータの取り出しや格納が行なえます。この図では番地は 4 飛びに (16 進で) 表記してありますが、これは整数 1 個が 4 バイト (32 ビット) で、番地は 1 バイト単位でつけてある CPU が多いのでそれにならっています。

アセンブリ言語や高水準言語でプログラムするときは、番地を直接書いていたら繁雑ですから、適当な名前 (A とか B とか) をつけて扱っています。この図の例では 3 つのアセンブリ言語命令がありますが、それは次のような意味になります (番地は適当な例示で 16 進表記です)。

- A の番地 (1C04) から整数値を取り出し、レジスタ eax に転送する。
- B の番地 (1C08) から整数値を取り出し、レジスタ eax に現在入っている値と加算する (レジスタの値がその加算の結果になる)。
- レジスタ eax に入っている値を、C の番地 (1C0C) に格納する。

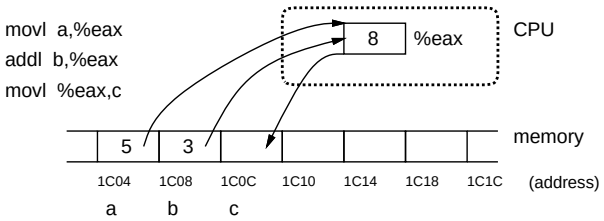


図 2.1: CPU 命令とメモリ番地

C 言語の科目なので C に戻るとして、これを C では次のように書くわけです。

```
c = a + b;
```

ここで良く見て欲しいのですが、「=」の左と右では変数の意味するところが違います。右では「a に入っている値」「b に入っている値」を意味しますね。これを**右辺値** (right value、rvalue) と呼びます。しかし、左では…「c = ...」というのは「c の 番地」つまり図でいうと 1C0C 番地に (右辺で計算した) 値を格納する、という意味になります。つまり、代入の左に書いた場合はそれは番地 (アドレス) なのです。これを**左辺値** (left value、lvalue) とも呼びます。

2.1.2　アドレス取得演算子・間接参照演算子

普段はこのようにアドレスと値 (右辺値) を使い分けているのですが、C 言語の特徴として、任意の変数のアドレスを取得して値として扱える、ということがあります (ややこしくなる原因でもあります)。たとえば次のコードを見てみましょう。

```
int a, b = 10, *p;
p = &a; *p = b;
```

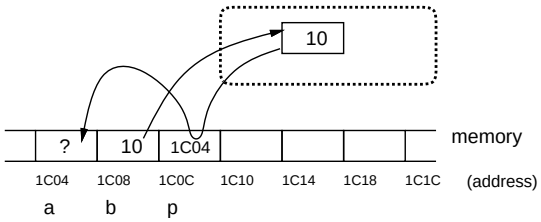


図 2.2: アドレスの取得と間接参照

これは、1 行目の宣言/初期化で、変数 a と b は整数型、p は「整数を指すポインタ型」と宣言しています。ポインタ型とは要するに「アドレスを表す型」のことです。さらに、b には初期値 10 を入ています。次に 2 行目で、p に「a のアドレスを」入れています (「&」は「アドレス取得演算子」です)。そして 3 行目は b に入っている値を取り出し、「p に入っているアドレスに」格納します。p には変数 a のアドレスが入っているわけですから、結果として変数 a に 10 が入ることになります (図 2.2)。[1]

「*p = b;」の「*」は「参照たどり演算子」ないし「**間接参照演算子** (indirect reference operator)」と呼ばれ、ポインタ型の値にだけ使えます。その意味は、左辺値として使う場合は「そのポインタ値が表すアドレスに格納」、右辺値として使う場合は「そのポインタ値が表すアドレスに格納されている値」となります。上記は左辺値に現れる例でしたが、たとえば続いて「b = *p + 1;」のように書くと、こんどは「a に入っている値に 1 を足して b のアドレスにに格納」になります。

なぜ「間接」かというと、「a = b」や「b = a」のように書いた場合は「a の番地に代入」「a の値を取り出し」のように「直接」指定しているのに対し、ポインタ変数 p を経由している場合は「いちど p に入っている番地を取り出し、その番地への代入/番地に格納されている値を参照」となるからです。面倒なだけに思えるかも知れませんが、このように「間接」にすることで、p に格納する番地を変更することでさまざまな場所にある値を扱えるという柔軟性が得られるのです。

ここで、C 言語の宣言の書き方と型の書き方について説明しておきます。「int i;」と書いた場合、変数 i は整数型、というのは普通に使ってきました。では「int *p;」は? これは「変数 p に間接参照演算子を適用した*p が整数である」ことを意味します。ということは、p そのものは「整数へのポインタ型」となります。たとえば「int **q;」であれば、2 回間接参照すると整数になるので、q は「整数へのポインタのポインタ型」です。

そして、**キャスト演算** (cast operation)「(型指定) 式」のために型指定を書くときは、変数宣言から宣言される変数名を取り除いたものを書きます。ですから、(int) は整数型へのキャスト、(int*) は整数のポインタ型へのキャスト、(int**) は整数のポインタへのポインタ型へのキャストになります。C 言語のここの構文は大変わかりづらいのですが、そういうことになっています。

2.1.3　配列とポインタ演算

次は**配列** (array) です。C 言語では配列を確保するということは、単にその配列の各要素を格納するのに必要な領域を用意することと同じです。そして、変数宣言で配列を確保した場合、その宣言した名前は… 「領域の先頭のアドレス」になります。次を見てください。

[1] 今日のシステムではアドレスは 64 ビット長が多いのですが、図が繁雑になるので図ではアドレスも整数と同じ 32 ビットの長さとして描いてあります。さらに、32 ビットのアドレスは 16 進表記で 8 桁になりますが、見にくいので 16 進 4 桁で例示しています。

```
int a[5] = {1,3,5,7,9}, i, *p; p = a;
```

このようにすると、メモリ上にはまず整数 5 個ぶんの領域が用意され、配列 a となります。そして整数変数 i、ポインタ変数 p の領域が取られます。次の「p = a;」は? 配列の名前 a は、配列の先頭のアドレスつまり 1C04 を意味するので、それが p に入ります。

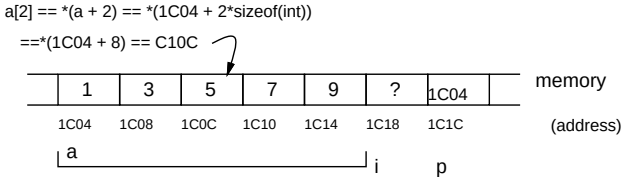


図 2.3: 配列の領域とポインタの関係

そして、配列は添字をつけてアクセスしますが、C 言語では「a[i]」は「*(a + i)」と同じ意味である、と定められています。この足し算は「**ポインタ演算** (pointer calculation)」と呼ばれ、片方がポインタ値、もう片方が整数である必要があります。

そして、アドレスに直接足すのでなく、ポインタが指す要素のバイト数 (整数なら 32 ビットなので 4) を掛けて足すと定められています。これはつまり、「ポインタが指している要素 (例: 整数) の i 個ぶん先の要素のアドレス」となります。そして、外側の*演算子があるので、そのアドレスに入っている値を参照したり (右辺値の場合)、そのアドレスに格納したり (左辺値の場合) できるわけです。

上の例では配列名 a に対してポインタ演算していましたが、ポインタ変数 p を使っても同じようにできます。そして上の例では p には配列 a のアドレスが入っていますから、どちらでも同じことです。たとえば「a[2]」でも「p[2]」でもどちらも「5」の入っている場所がアクセスできます。

2.1.4 配列の入力・出力・比較

それでは実践に進むことにして、まず配列を読み込む/出力する関数 (これらは既習だと思いますが)、そして 2 つの配列が等しいかどうか調べる関数を作ってみます。これの関数の名前やパラメタを決める必要があるので、それらのプロトタイプ宣言を示します。

- `void iarray_read(int *a, int n);` — 配列 a に n 個の値を読み込む。
- `void iarray_print(int *a, int n);` — 配列 a の n 個の値を出力。
- `bool iarray_equal(int *a, int *b, int n);` — 二つの配列 a、b の先頭から n 個の値が等しいなら true、そうでなければ false を返す。

ではこれらを使うコードを見てみましょう。

```
// iarray_demo.c --- array input/output and equality demo.
#include <stdio.h>
#include <stdbool.h>
void iarray_read(int *a, int n) {
  for(int i = 0; i < n; ++i) {
    printf("%d> ", i+1); scanf("%d", a+i);
  }
}
void iarray_print(int *a, int n) {
  for(int i = 0; i < n; ++i) { printf(" %2d", a[i]); }
  printf("\n");
}
```

```
bool iarray_equal(int *a, int *b, int n) {
  for(int i = 0; i <= n; ++i) {
    if(a[i] != b[i]) { return false; }
  }
  return true;
}
char *bool2str(bool b) { return b ? "true" : "false"; }
int main(void) {
  int a[4], b[4];
  iarray_read(a, 4); iarray_print(a, 4);
  iarray_read(b, 4); iarray_print(b, 4);
  printf("equal: %s\n", bool2str(iarray_equal(a, b, 4)));
  return 0;
}
```

bool2str とは? C 言語では bool 型といっても整数なので、普通に出力すると 0 か 1 になりますが、true、false と出力したいので「論理値を"true"、"false"の文字列 (C 言語では文字ポインタ) に変換」します。実行例は次の通り。

```
% ./a.out
1> 4
2> 3
3> 2
4> 1
  4  3  2  1
1> 4
2> 3
3> 2
4> 1
  4  3  2  1
equal: false
%
```

あれれ、同じ内容を打ち込んだのに「等しくない」という結果になっていますね? iarray_equal にバグがあるようです。ここですぐ iarray_equal を熟読してバグを取ろうとする方、ちょっと待ってください。このようにバグがあると分かればそれを探して除去できますが、たとえば上の実行例で「2 つの配列が違う場合」だけ試してしまうと、この例のように「等しいのに等しくないと答える」バグはあると気づかず見過ごされます。ではどうするのがよいのでしょう?

2.1.5　単体テストとテストケース

array_equal が正しいかどうかを確認するのに、上のように「気分で打ち込んで」様子を見るのはよい方法ではありません。もっと系統的に「確認するためのデータ」を用意して調べるべきです。調べる対象である単純な機能に対してその正しさをチェックするテストは**単体テスト** (unit test)、テストに含まれる「試行データと想定正解の組」を**テストケース** (test cases) と呼びます。具体例を見てみましょう。

```
// test_array_equal.c --- unit test of array_equal
#include <stdio.h>
#include <stdbool.h>
```

```
(iarray_read, iarray_print, iarray_equal, bool2str here)
void expect_bool(bool b1, bool b2, char *msg) {
  printf("%s %s:%s %s\n", (b1==b2)?"OK":"NG",
         bool2str(b1), bool2str(b2), msg);
}
int main(void) {
  int a[] = {0, 1, 2, 3, 4, 5}, b[] = {9, 1, 2, 3, 4, 6};
  expect_bool(iarray_equal(a, b, 5), false, "01234 : 91234");
  expect_bool(iarray_equal(a+1, b+1, 5), false, "12345 : 12346");
  expect_bool(iarray_equal(a+1, b+1, 4), true, "1234 : 1234");
  expect_bool(iarray_equal(a+1, b+1, 3), true, "123 : 123");
  expect_bool(iarray_equal(a, b, 0), true, "[] : []");
  return 0;
}
```

新たに作った関数 expect_bool というのは、テストしようとする関数の結果型が bool のとき使うもので、結果値 b1 と想定される結果 b2 が等しければ OK、そうでなければ NG と表示し、さらに 2 つの値とメッセージ (どのテストケースかが分かるような文字列) を表示します。ついでなので、整数用の expect_int と実数用の exptct_double も示しておきます。

```
void expect_int(int i1, int i2, char *msg) {
  printf("%s %d:%d %s\n", (i1==i2)?"OK":"NG", i1, i2, msg);
}
void expect_double(double d1, double d2, char *msg) {
  printf("%s %g:%g %s\n", (d1==d2)?"OK":"NG", d1, d2, msg);
}
```

話題を戻すと、ここでは先頭と最後の値が違う 2 つの配列を用意して、先頭や末尾を含んだり含まなかったりするさまざまな範囲で iarray_equal を呼び、結果を想定正解と比べてチェックします。実行例を見ましょう。

```
% ./a.out
OK false:false 01234 : 91234
OK false:false 12345 : 12346
NG false:true 1234 : 1234
OK true:true 123 : 123
NG false:true [] : []
%
```

こうして見るとやはり、「等しいはずなのに等しくないという結果」があります。「1234 : 1234」は等しくなくて、「123 : 123」は等しいというのは、指定された範囲の先まで比べているからかもと思えます。長さ 0 の「[] : []」も等しくないという結果なのでその予想は正しそうです。そこでコードを熟読すると、iarray_equal の中の for 文の条件「i <= n」は正しくなく、「i < n」であるべきだとわかります。

どうでしょう? そんな面倒なことをしなくてもよく見たらバグは見付かる? この程度ならそうでしょうけれど、複雑なコードの中のどこかで iarray_equal を利用しているとしたら、そのコード全体からバグを探すのは大変です。だから単体テストを用意してそれぞれの部品をチェックしておくことは大切なのです。さらに、どの部品も将来的に機能を追加・修正する可能性があります。そのときに「壊れて」いないかどうか確認するために、このテストケースをとっておいて再度実行するべきです。これを**回帰テスト** (regression test) と呼びます。

この後の演習で配列を結果とするものに対して単体テストを書くため、expect_iarray も示します。2 つの配列とサイズとメッセージを受け取り、中では (もちろんバグを修正した)iarray_equal を下請けに使います。

```
bool iarray_equal(int *a, int *b, int n) {
  for(int i = 0; i < n; ++i) {
    if(a[i] != b[i]) { return false; }
  }
  return true;
}
void expect_iarray(int *a, int *b, int n, char *msg) {
  printf("%s %s\n", iarray_equal(a, b, n)?"OK":"NG", msg);
  iarray_print(a, n); iarray_print(b, n);
}
```

演習 1 array_equal.c、test_array_equal.c の例題を動かして動作を確認しなさい。納得したら以下のものを作ってみなさい。必ず単体テストを動かし、またテストケースは増やしてみること。

a. 配列の最大値を求める関数 `int iarray_max(int *a, int n)` を作成する。

```
int a[] = {9,0,0,1,2,3}, b[] = {-1,-3,-2,-4,-1};
expect_int(iarray_max(a, 6), 9, "9 0 0 1 2 3");
expect_int(iarray_max(a+1, 5), 3, "0 0 1 2 3");
expect_int(iarray_max(b, 5), -1, "-1 -3 -2 -4 -1");
```

b. 配列の並び順を逆順にする関数 `void iarray_revese(int *a, int n)` を作成する。

```
int a[] = {8,5,2,4,1}, b[] = {1,4,2,5,8};
iarray_reverse(a, 5); expect_iarray(a, b, 5, "85241 -> 14258");
```

c. 何らかの整列アルゴリズムで配列を昇順に整列する関数 `void iarray_sort(int *a, int n)` を作成する。

```
int a[] = {8,5,2,4,1}, b[] = {1,2,4,5,8};
iarray_sort(a, 5); expect_iarray(a, b, 5, "85241 -> 12458");
```

d. 2 つの配列 (長さは同じ) を受け取り、2 番目の各要素の値を 1 番目の配列の各要素に足し込む関数 `void iarray_add(int *a, int *b, int n)` を作成する。

```
int a[] = {8,5,2,4,1}, b[] = {1,1,2,2,3}, c[] = {9,6,4,6,4};
iarray_add(a, b, 5); expect_iarray(a, c, 5, "85241+11223 -> 96464");
```

e. 配列に加えて整数 2 つを受け取り 1 つを返す関数へのポインタを受け取り、それを用いて配列の値を集約した結果を返す関数 `int iarray_inject(int *a, int n, int (*fp)(int, int))` を作成する。さらにそれを用いて、「配列の合計値」「配列の最大値」を求める。[2]

```
int a[] = {8,5,2,4,1};
expect_int(iarray_inject(a, 5, iadd), 20, "8+5+2+4+1");
expect_int(iarray_inject(a, 5, imax), 8, "max(8,5,2,4,1)");
```

最後の設問には新しい概念が出て来ているので説明しておきます。C 言語では変数のポインタのほかに**関数のポインタ** (function pointer) も扱うことができます。たとえば次のような感じです。

[2]inject(注入) とは、たとえば列「1,2,3」に対して「+」を注入するといった場合、個々の要素の間に「+」を挿入するような意味です。その結果は「1+2+3」となり、合計を意味します。

```
int iadd(int x, int y) { return x + y; }
int imax(int x, int y) { return (x > y) ? x : y; }
int (*fp)(int,int);
fp = iadd;  // or fp = imax;
```

変数 fp の型は何でしょうか。先に説明した宣言の読み方を援用すると、「fp を間接参照して、さらに整数を 2 つパラメタとして渡して呼び出すと、整数になる」型、つまり「整数を 2 つ受け取り 1 つ返す関数へのポインタ」型となります。呼び出す時は「(*fp)(1, 2)」のように間接参照が先に実行されるようかっこで囲む必要があります。あと、関数ポインタの変数への代入時には「&」が不要なことに注意してください。単独の関数名はそれ自体が「関数へのポインタ」を表します。

2.2 動的メモリ管理と構造体の情報隠蔽

2.2.1 malloc と free による動的メモリ管理

ここまでに学んだ内容だと、配列の大きさは配列を宣言する時に定数で指定し、実行時には変更できません。たいていのプログラムでは扱うデータの量が分かるのでそれにいくらか余裕を持たせた大きさで宣言すればよいのですが、配列を多数使うような場合、全部に余裕を持たせていると「使わない無駄」が多かったりして好ましくありません。

そこで別の方法として、「使う時にサイズを指定してメモリを割り当てる」「使い終わったら返却する」という機能を活用するやり方があります。これを「**動的メモリ管理** (dynamic memory management)」と呼び、C の標準ライブラリにはこのために、次の 2 つの関数が含まれています。これらは (関連する型も含めて)stdlib.h で宣言されています。

- void *malloc(size_t size); — 領域の**割り当て** (allocation)
- void free(void *ptr); — 領域の**返却** (deallocaiton)

見慣れない型が出てきますが、まず **void*** というのは「何らかのポインタ」を表す型で、実際に使う時は int*など必要なポインタ型にキャストして使います。次に **size_t** というのは「メモリのサイズを扱う時に使う整数型」で、システムの必要に応じて unsigned int や unsigned long に対応させられますが、使う時は要するに整数を渡すと思っておけばよいです。

たとえば、1000 要素の実数配列を使いたい時には次のようにすればよいわけです。

```
double *a = (double*)malloc(1000 * sizeof(double)); // 割り当て
... a[i] を使用する ...
free(a);  // 返却
```

sizeof は任意のデータ型の 1 要素のバイト数を指定するのに使えます。要素数はここでは 1000 としていますが、実行時にいくつ必要か計算して、その値で割り当てるのが実際の使い方なわけです。

ところで、free で返却しなかったらどうなるの、という疑問があるかも知れません。free で返却した領域は、後で別の malloc 呼び出しがあったときに再度利用されます。ですから、長時間動くプログラムに「free し損ない」が含まれていると、徐々に使わないメモリ領域がふくらんできて性能低下する原因になります。これを「**メモリリーク** (memory leak)」と呼びます。

ただし、プログラムがもう終わるというところでは malloc ももう使わないわけですから、free しない、という流儀もあり得ます。また、今日のシステムではメモリは潤沢にあるので、長時間動くプログラムでなく、あまり大量のデータを使わないなら、途中で一切 free しないという方針でも動くでしょう。

そして、込み入ったプログラムだとどれとどれを使っていてどこで使わなくなるのかを把握するのが難しいこともあります。その結果、間違って (つまり実はまだ使うのに)free してしまう) と、その領域を他の部分で書き換えることになり、原因の分かりにくいバグになります。

このように free には多くの問題があるので、現在は使わなくなった領域を自動的に把握して回収する「**ごみ集め** (garbage colleciton)」機能を搭載し、free を使わない言語が主流です。

今回は、行儀よく「自分で割り当てたものは自分で解法する」流儀で記述していますが、次回からは簡単のため「プログラムが終了する際には開放は省略する」方針を採ります。ただし、ずっと動き続けるようなプログラムで割り当てを繰り返す場合は、解放も必要である (そうしないとメモリが不足する可能性がある) ことを覚えておいてください。

2.2.2　可変長配列

malloc を使って実行時に自由な長さの領域を割り当てられるのはいいのですが、割り当てた領域の長さは自分で把握している必要があります。その繁雑さを避けるのに、長さも一緒に記録しておく方法があります。とくに整数の配列なら、図 2.4 のように「先頭にデータが何個入っているか記録しておく」ことができます。そして、長さ 2 と 3 の列をくっつけて長さ 5 の列を作り出す、みたいなことができるようにするわけです。

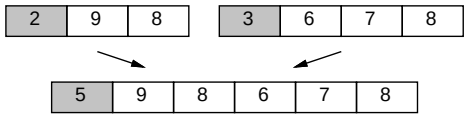


図 2.4: 先頭に長さを入れた整数の列

実際にやってみましょう。指定した長さの列を作る ivec_new、列の内容を読み込む ivec_read、内容を打ち出す ivec_print、そして 2 つの列をくっつける ivec_concat を作成し、main からこれらを呼び出します。

```
// ivec_demo.c --- int vector demonstration.
#include <stdio.h>
#include <stdlib.h>
void iarray_read(int *a, int n) {
  for(int i = 0; i < n; ++i) {
    printf("%d> ", i+1); scanf("%d", a+i);
  }
}
void iarray_print(int *a, int n) {
  for(int i = 0; i < n; ++i) { printf(" %2d", a[i]); }
  printf("\n");
}
int *ivec_new(int size) {
  int *a = (int*)malloc((size+1) * sizeof(int));
  a[0] = size; return a;
}
void ivec_read(int *a) { iarray_read(a+1, a[0]); }
void ivec_print(int *a) { iarray_print(a+1, a[0]); }
int *ivec_concat(int *a, int *b) {
  int *c = ivec_new(a[0]+b[0]);
  for(int i = 1; i <= a[0]; ++i) { c[i] = a[i]; }
  for(int i = 1; i <= b[0]; ++i) { c[i + a[0]] = b[i]; }
  return c;
}
```

```
int main(void) {
  int *a, *b, *c;
  a = ivec_new(3); ivec_read(a);
  b = ivec_new(2); ivec_read(b);
  c = ivec_concat(b, a); ivec_print(c);
  free(a); free(b); free(c);
  return 0;
}
```

参考までに、ivec_concat の単体テストも示しておきます。

```
// test_ivec_concat.c --- unit test for ivec_concat.
#include <stdio.h>
#include <stdbool.h>
(ivec_new, ivec_concat, iarray_equal, iarray_print, expect_iarray)
int main(void) {
  int a[] = {3,1,2,3}, b[] = {2,4,5}, c[] = {5,1,2,3,4,5};
  int *p = ivec_concat(a, b);
  expect_iarray(p, c, 6, "[1,2,3]+[4,5]=[1,2,3,4,5]");
  return 0;
}
```

演習 2 上の例題をそのまま動かせ。長さを変更して動かしてみてもよい。その後、次の関数を作ってみよ。単体テストも作ること (どのようにテストするかは自分で決めてよい)。

a. 2 つの列を受け取り、両方の列の内容を交互に並べた列を返す (例: 「1, 2, 3」「4, 5, 6」→「1, 4, 2, 5, 3, 6」。長さが異なる場合の扱いは好きに決めてよい)。
b. 1 つの列を受け取り、その内容を逆順にした列を返す (例: 「1, 2, 3」→「3, 2, 1」)。
c. 1 つの列を受け取り、その内容を昇順に整列した列を返す (例: 「3, 1, 4」→「1, 3, 4」)。
d. 2 つの昇順に整列ずみの列を受け取り、両方の列をマージした昇順の列を返す (例: 「3, 5, 9」「1, 4, 6」→「1, 3, 4, 5, 6, 9」)。
e. その他自分が面白いと思う列の操作を行なう。

2.2.3 構造体による情報隠蔽

情報隠蔽 (information hiding、encapsulation) とは、ひとまとまりの機能を実装するデータ構造を実装するコードだけから参照可能にし、それ以外からは見えなくすることを言います。

Ruby などクラス機能を持つ言語ではこれはクラスを使うことで自然に実現できますが、C 言語では工夫して実現する必要があります。前回の eps.c のように、ファイルを分けてファイル内だけの変数 (**static** 変数) を使うことも 1 つの方法ですが、その変数群が 1 セットしか用意できないという弱点があります。ここではその弱点がない方法として、次の方法を使います。

(1) データ構造を表す変数群一式が 1 つの構造体 (レコード) のフィールドになるような構造体を定義し、それを用いて実装する。データ一式ごとの領域を動的に割り付ける。
(2) 外部から API を呼び出すためのヘッダファイルには構造体定義は書かず、API の関数群には構造体のポインタを渡す。
(3) 外部のファイルはヘッダファイルを取り込み、API の関数を呼び出すことで機能を使う。

具体例がないと分からないと思うので、「等差数列の項を次々に取り出して来る」機能を作ってみましょう。API を定義するヘッダファイルを次のようにします (cdseq.h)。

```
// cdseq.h --- constant difference sequence API.
struct cdseq *cdseq_new(int s, int d);
int cdseq_get(struct cdseq *r);
void cdseq_free(struct cdseq *r);
```

構造体の定義が書かれていなくても、構造体のポインタ型は自由に使うことができます。それは、ポインタのビット数は指す先がどのような型であっても同じだからです (メモリアドレスなので当然ですが)。

これを用いて「1 から始まる階差 2 の等差数列と、0 から始まる階差 3 の数列を 2:1 で混ぜて 15 個出力する」プログラムを作ってみます。

```
// cdseq_demo.c -- cdseq demonstration.
#include <stdio.h>
#include "cdseq.h"
int main(void) {
  struct cdseq *s1 = cdseq_new(1, 2);
  struct cdseq *s2 = cdseq_new(0, 3);
  int i;
  for(i = 0; i < 6; ++i) {
    printf(" %2d", cdseq_get(s1));
    printf(" %2d", cdseq_get(s1));
    printf(" %2d", cdseq_get(s2));
  }
  printf("\n"); cdseq_free(s1); cdseq_free(s2);
  return 0;
}
```

動かしたようすは次の通り。

```
% ./a.out
1  3  0  5  7  3  9 11  6 13 15  9 17 19 12 21 23 15
```

では、実装部分 cdseq.c を見ていただきます。

```
// cdseq.c -- cdseq implementation.
#include <stdlib.h>
#include "cdseq.h"
struct cdseq { int value, diff; };
struct cdseq *cdseq_new(int s, int d) {
  struct cdseq *r = (struct cdseq*)malloc(sizeof(struct cdseq));
  r->value = s; r->diff = d; return r;
}
int cdseq_get(struct cdseq *r) {
  int v = r->value; r->value += r->diff; return v;
}
void cdseq_free(struct cdseq *r) {
  free(r);
}
```

最初に構造体の定義があります。等差数列なので、初項と公差を覚えることとしています。

cdseq_new では、まず構造体の領域を割り当てます。本当は返されたポインタ値が NULL だったらメモリ不足のエラーなのですが、まず起きないので当面省略します。つぎに、返された構造体の領域の初項と公差に値を入れ、その後ポインタ値を返します。ここで使っているアロー演算子について復習しておきます。

```
struct cdseq s; s.value = 1; s.diff = 3;  // 通常変数
struct cdseq *r = &s;
  (*r).value = 1; (*r).diff = 3;  // ポインタ
  r->value = 1; r->diff = 3; // アロー演算子
```

上の 1 行目のように、構造体のフィールドは「s.value」のように「.」に続けてフィールド名を指定しますが、もし通常変数ではなくポインタだったとすると、2 行目のように「(*r).value」と、まず間接参照する必要があります。C 言語ではこれをよく書くため、もっと見た目がよいように同じことを「r->value」と書いてもよくなっています。

さて cdseq_get ですが、「次の値」は value フィールドにあるのでそれを変数 v に保存します。そして、value フィールドは diff だけ増やすことで、次の値を用意しておきます。最後に保存してあった v を返します。最初に return v; としたくなりますが、return するとその後の文は一切実行されないので、このように一旦覚えておき return は最後にする必要があります。

最後の cdseq_free ですが、これはレコードのポインタを free するだけです。だったら呼ぶ側で直接 free を呼べばよい? この場合はそれでもよいですが、レコードの中にさらに動的メモリ割り付けした結果のポインタを保持したいときは、それも返却しなければなりませんから、このようにそれぞれの構造ごとに後始末の関数を用意するほうがよいのです。

こちらも課題の前に単体テストの例を示します。今度は複数回 get を呼ぶごとに値をそれぞれテストしていることに注意。

```
// test_cdseq_1.c --- unit test for cdseq.
#include <stdio.h>
#include "cdseq.h"
void expect_int(int i1, int i2, char *msg) {
  printf("%s %d:%d %s\n", (i1==i2)?"OK":"NG", i1, i2, msg);
}
int main(void) {
  struct cdseq *s = cdseq_new(2, 3);
  expect_int(cdseq_get(s), 2, "2+3*0 = 2");
  expect_int(cdseq_get(s), 5, "2+3*1 = 5");
  expect_int(cdseq_get(s), 8, "2+3*2 = 8");
  return 0;
}
```

演習 3 上の等差数列生成 API の例題をそのまま動かせ。動いたら次のような機能を同様のやりかたで実現してみよ。いずれも単体テストを作ること (単体テストのやりかたは自分で決めてよい)。

a. 上の cdseq.c に、等差数列を初項に戻す機能 cdseq_reset を追加してみよ。

b. 上の例では get を呼ぶたびに次の値が出て来たが、get だけでは値が進まず、cdseq_fwd を呼ぶと次の値に進むように変更してみよ。さらに、現在が何番目 (最初が 0 とする) の項かを返す機能 cdseq_num を追加してみよ。

c. 初項と公比を与えて等比数列 (実数値) を生成するような機能を実現してみよ。詳細は好きに決めてよい。

d. 複数の値を put でき、いつの時点でもそれまでの数値のうちの最大値と最小値が get できるような機能を実現してみよ。詳細は好きに決めてよい。
e. その他、構造体と情報隠蔽のしくみを使って、何か面白いと思う機能を実現してみよ。

2.3　トレーニング課題

2019 年度の実施で、極めて基本的な C 言語プログラムでの練習が必要な学生さん「も」いるということが明らかになったので、今年度から「トレーニング課題」を追加します。トレーニング課題は基本練習の易しい課題で、「トレーニング課題 10 個で」通常の課題 1 個の代わりとして提出することができます。B 課題は通常の課題 2 個でも、通常 1 個トレーニング 10 個でも、トレーニング 20 個でも構いません。A 課題のレビュー課題はトレーニング課題以外を用いてください。もしトレーニング課題のみで出す場合は、レビュー課題は 10 個見てもらってください。

また、トレーニング課題であっても、各々に単体テストは必須ですので注意してください。

演習 T4 整数配列 a とそのサイズ n、値 v を受け取り、先頭から n 要素を v に設定する関数 `void iarray_fill(int a[], int n, int v)` を作り単体テストする例題を示す。

```
#include <stdio.h>
void iarray_fill{int a[], int v, int n) {
  for(int i = 0; i < n; ++i) { a[i] = v; }
}
(iarray_print, expect_iarray here)
int main(void) {
  int a[] = { 0,1,2,3,4,5 }, b[] = { 6,6,6,3,4,5 };
  iarray_fill(a, 3, 6);
  iarray_print(a, 6);
  expect_iarray(a, b, 6, "[0,1,2,3,4,5] -> [6,6,6,3,4,5]");
  iarray_fill(a+3, 0, 7);
  iarray_print(a+3, 3);
  expect_iarray(a+3, b+3, 3, "[3,4,5] -> [3,4,5]");
  return 0;
}
```

これを参考に (しなくてもよいが)、次の関数を作り単体テストせよ。関数の名前は好きに決めてよい。パラメタ a, n, v は例題と同じとする。

a. 先頭から n 要素を v 増やす。
b. 先頭から n 要素について、値が 0 だったときそれを v に変更する。
c. 先頭から n 要素について、値が v に等しかったきそれを 0 に変更する。
d. 先頭から n 要素について、値が v より大きかったらそれを v に変更する。
e. 先頭から n 要素について、値が v より小さかったらそれを 1 減らす。
f. 先頭から n 要素について、絶対値が v より大きかったらそれを 0 にする。
g. 先頭から n 要素について、絶対値が v より小さかったら符号を反転する。

演習 T5 整数配列 a, b とサイズ n を受け取り、a の内容を反転して (逆順にして)b にコピーする関数 `void iarray_revcopy(int a[], int b[], int n)` を作り単体テストする例題を示す。

```
#include <stdio.h>
void iarray_revcopy{int a[], int b[], int n) {
  for(int i = 0; i < n; ++i) { b[i] = a[n-i-1]; }
}
(iarray_print, expect_iarray here)
int main(void) {
  int a[] = { 0,1,2,3,4,5 }, b[6], c[] = { 5,4,3,2,1,0 };
  iarray_rshift(a, b, 6);
  iarray_print(b, 6);
  expect_iarray(a, c, 6, "[0,1,2,3,4,5] -> [5,4,3,2,1,0]");
  iarray_rshift(a, b, 2);
  iarray_print(b, 2);
  expect_iarray(a, c, 2, "[0,1] -> [1,0]");
  return 0;
}
```

これを参考に (しなくてもよいが)、次の関数を作り単体テストせよ。関数の名前は好きに決めてよい。パラメタ a, b, n は例題と同じとする。指定されていない b の内容は変更されないままにすること。n は偶数であるものとしてよい。

a. a の前半を b の後半、a の後半を b の前半にコピー。
b. a の前半を b の後半に逆順でコピー。
c. a の後半を b の後半に逆順でコピー。
d. a の前半を b の 0、2、4、…番に 1 つおきにコピー。
e. a の前半を b の 1、3、5…番、後半を 0、2、4…番にコピー。
f. a の 0 番目と 1 番目の和、2 番目と 3 番目の和…を b の後半にコピー。

本日の課題 2a

「演習 1」〜「演習 3」で動かしたプログラム 1 つを含むレポートを本日中 (授業日の 23:59 まで) に提出してください。

1. 学籍番号、氏名、ペアの学籍番号 (または「個人作業」)、提出日時。名前の行は先頭に「@@@」を付けることを勧める。
2. プログラムどれか 1 つのソースと「簡単な」説明。単体テストと実行例を必ず含めること。
3. レビュー課題。提出プログラムに対する他人 (ペア以外) からの簡単な (ただしプログラムの内容に関する) コメント。
4. 以下のアンケートの回答。

   Q1. アドレス、ポインタについて納得しましたか。
   Q2. 「構造体を用いた情報隠蔽」について納得しましたか。
   Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

次回までの課題 2b

「演習 1」〜「演習 3」(ただし 2a で提出したものは除外、以後も同様) の小課題全体から選択して 2 つ以上プログラムを作り、レポートを提出しなさい。できるだけ演習 2 からも選ぶこと。レポートは次回授業前日 23:59 を期限とします。

1. 学籍番号、氏名、ペアの学籍番号 (または「個人作業」)、提出日時。名前の行は先頭に「@@@」を付けることを勧める。
2. 1 つ目の課題の再掲 (どの課題をやったか分かればよい)、プログラムのソースと「丁寧な」説明、および考察 (課題をやってみて分かったこと、分析、疑問点など)。単体テストと実行例を必ず含めること。
3. 2 つ目の課題についても同様。
4. 以下のアンケートの回答。

   Q1. アドレス、ポインタを使うプログラムで注意すべきことは何だと思いますか。
   Q2. ここまでのところで、プログラムを作るときに重要だが自分で身に付いていないと思うことは何ですか。
   Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

＃3　スタックとその利用

今回は次のことが目標となります。

- CS における基本的なデータ構造であるスタックの機能と実装について知る。
- スタックを用いた基本的なアルゴリズムについて知る。

その前にコマンド引数の話題から始め、以後プログラムへの入力にコマンド引数を併用します。

3.1　コマンド引数

コマンド引数 (comand line arguments) とはもともと Unix の用語で、コマンドやプログラムを起動するときに「`./a.out aa bbb ccc`」のようにコマンド名 (プログラム名) より後ろに指定する文字列のことです。C 言語では `main` の引数を `void` と指定するかわりに、次のように指定することでこの情報を受け取れます。

```
int main(int argc, char *argv[]) {
  ...
```

`argc`、`argv` は引数なので名前は任意ですが、これらの名前を使うことが慣例となっています。これらに渡される内容ですが、`argc` はコマンド引数の語の数 (コマンド名を含む)、`argv` はその各要素がそれぞれの語の文字列 (C 言語なので `char` へのポインタ) となっています (図 3.1)。

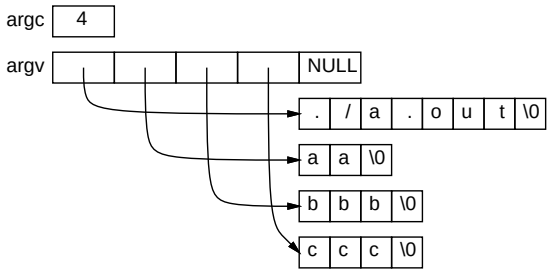


図 3.1: コマンド引数の構造

ではこれらを文字列として打ち出して見ましょう。

```
// argdemo1.c --- print argc/argv as string
#include <stdio.h>
int main(int argc, char *argv[]) {
  for(int i = 0; i < argc; ++i) { printf("%s\n", argv[i]); }
  return 0;
}
```

実行のようすは次の通り。「0 番目」はコマンド名になること、「'...'」や「"..."」で囲んだものは 1 つの文字列として渡されることに注意 (C では文字列は「"..."」のみですが、シェルや Ruby では「'...'」でもよいのでしたね)。

```
% ./a.out aa 'bbb ccc'
./a.out
aa
bbb ccc
```

ついでにもう 1 つ、数値を受け取りたい場合には文字列から整数や実数に変換する次の関数が使えます (stdlib.h で定義されています)。

- `int atoi(char *s)` — 文字列をそれが表す整数に変換。
- `double atof(char *s)` — 文字列をそれが表す実数値に変換。

これも例を示しましょう。コマンド名は数でないので、今度は 1 番目から順に処理します。

```
// argdemo2.c --- sum of argument strings (as int)
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
  int sum = 0;
  for(int i = 1; i < argc; ++i) { sum += atof(argv[i]); }
  printf("sum = %d\n", sum);
  return 0;
}

% ./a.out 3 4 5
sum = 12
```

この先では `scanf` による入力に加えて、このコマンド行引数による入力も活用していきます。

3.2　スタック

3.2.1　スタックの概念

スタック (stack) とは「積み上げたもの」を意味しますが、コンピュータサイエンスでは **LIFO**(last-in, first-out) すなわち「後に入れたものほど先に出て来る」記憶領域のことを指します。図 3.2 左のように、「1」「2」「3」がこの順でやってきたとき、記憶領域に「積み上げて」保管すると、取り出した時に逆順になっていますよね。つまり「後から入れたものほど先に出て来る」からそうなるわけです。

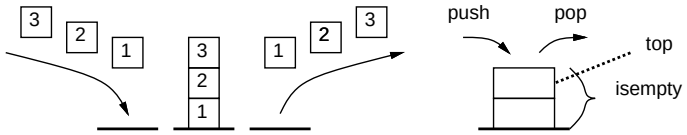


図 3.2: スタックの概念

スタックをプログラム上で実現する場合、その基本操作は要素を追加する **push**、そして取り出す **pop** です (日本語ではしばしば「積む」「降ろす」とも言います)。また、空っぽになったのにさらに取り出すわけには行きませんから、空っぽかどうかを調べる操作 `isempty` が必要です。

あと、先頭 (いちばん上) にある要素を降ろさずに見たいことがあるので、そのための操作 `top` を用意することもあります (`push` と `pop` があれば作れますが)。

ではスタックは何の役に立つのでしょう。それをこれから色々見ていきますが、まずは上の例のように「列を逆順にする」のに使えます。そんな面倒なことをしなくても、配列に入れて逆から取り出

せばよい、とか思いましたか? そうなのですが、そのコードは結構面倒ですよね。それと比較して、「順に push して、それから順に pop する」方が簡単なのです。大した簡単さではないと思うかも知れませんが、このような単純化 (抽象化) が役に立つことは多くあるのです。

3.2.2 配列を使ったスタックの実装

スタックを実装する 1 つの方法は、配列を使うことです。配列 arr と、次に入れる要素の位置を示す整数 ptr を用いた場合、push と pop は次のように書けます (ptr の初期値は 0 にします)。

```
arr[ptr++] = value;  // same as: arr[ptr] = value; ++ptr;
value = arr[--ptr];  // same as: --ptr; value = arr[ptr];
```

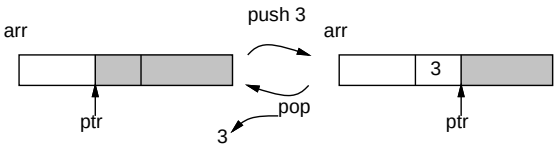


図 3.3: 配列を使ったスタックの実装

つまり、push は「空いている位置に値を入れ、位置は 1 つ進める」、pop は「位置を 1 つ戻し、その位置の値を取り出す」なわけです (図 3.3)。ちょっと違う版として、ptr は空いている位置ではなく「最後に入れた値の位置」にする方法もありますが、その場合 ptr の初期値は -1 にする必要があります。

では次に、これを前回やったレコード型を使ってカプセル化しましょう。さらに、上では push と pop だけでしたが、そのほかの機能も追加します。まずヘッダファイルから。

```
// istack.h --- int type stack interface
#include <stdbool.h>
struct istack;
typedef struct istack *istackp;
istackp istack_new(int size); // allocate new stack
bool istack_isempty(istackp p); // test if the stack is empty
void istack_push(istackp p, int v); // push a value
int istack_pop(istackp p);      // pop a value and return it
int istack_top(istackp p);      // peek the topmost value
```

typedef という見慣れない行がありますが、この行は istackp という名前を「struct istack*」という型を表す型名として定義しています。以後、この名前を使うことで記述が簡潔になります。[1]

そして実装本体は次のようになります。作成時には十分余裕のある要素数を渡すものとします。先の説明で単独変数のように書いていたものはすべて、構造体のフィールドになり、アロー記法でアクセスします。top は ptr の指す 1 つ手前の位置になることに注意。

```
// istack.c --- int type stack impl. with array
#include <stdlib.h>
#include "istack.h"
struct istack { int ptr; int *arr; };
istackp istack_new(int size) {
  istackp p = (istackp)malloc(sizeof(struct istack));
```

[1] たとえば「typedef int *intp」により intp という名前を整数へのポインタを表す型として定義することもできますが、大して読みやすくはならないので、こちらはあまりやりません。

```
  p->ptr = 0; p->arr = (int*)malloc(size * sizeof(int));
  return p;
}
bool istack_isempty(istackp p) { return p->ptr <= 0; }
void istack_push(istackp p, int v) { p->arr[p->ptr++] = v; }
int istack_pop(istackp p) { return p->arr[--(p->ptr)]; }
int istack_top(istackp p) { return p->arr[p->ptr - 1]; }
```

ではこれを使ってみることにして、文字列をさかさまにして出力するという例題を作ってみましょう。文字列はコマンド行から与えます。

```
// reversestr.c --- print argv[1] in reverse order
#include <stdio.h>
#include <stdlib.h>
#include "istack.h"
int main(int argc, char *argv[]) {
  istackp s = istack_new(200);
  char *t = argv[1];
  for(int i = 0; t[i] != '\0'; ++i) { istack_push(s, t[i]); }
  while(!istack_isempty(s)) { putchar(istack_pop(s)); }
  putchar('\n');
  return 0;
}
```

呼び方は「./a.out '文字列'」のようにするので、文字列は argv[1] で渡されますが、それを変数 t に入れます。そしてその各文字をスタックに積んで行きます (文字列はナル文字'\0' で終わりになることに注意)。積み終ったらスタックから各文字を降ろして出力します。putchar(文字) は文字を 1 文字出力する関数です。実行例は次の通り。

```
% ./a.out 'this is a pen.'
.nep a si siht
```

スタックの単体テストをしておきましょう。取り出した整数が正しいかチェックするので、expect_int を使っています。

```
// test_istack.c --- unit test for istack.
#include <stdio.h>
#include <stdlib.h>
#include "istack.h"
void expect_int(int i1, int i2, char *msg) {
  printf("%s %d:%d %s\n", (i1==i2)?"OK":"NG", i1, i2, msg);
}
int main(int argc, char *argv[]) {
  struct istack *s = istack_new(200);
  istack_push(s, 1); istack_push(s, 2); istack_push(s, 3);
  expect_int(istack_pop(s), 3, "push 1 2 3 ; pop");
  expect_int(istack_pop(s), 2, "pop");
  istack_push(s, 4);
  expect_int(istack_pop(s), 4, "push 4; pop");
  expect_int(istack_pop(s), 1, "pop");
```

```
    return 0;
}
```

実行例は次の通り。

```
% ./a.out
OK 3:3 push 1 2 3 ; pop
OK 2:2 pop
OK 4:4 push 4; pop
OK 1:1 pop
```

3.3 数式の扱い

3.3.1 括弧の対応

それではいよいよ、スタックが役に立つ例を見ていただくことにします。スタックは基本的に「入れ子構造」の処理に適しています。たとえば、数式などで括弧を使う場合、(1) 開き括弧と閉じ括弧が対応していて、(2) 閉じ括弧が先行することは無く、(3) 開きと閉じの対応関係がクロスしない、という条件が必要です (図 3.4)。これをチェックするアルゴリズムを考えてみましょう。

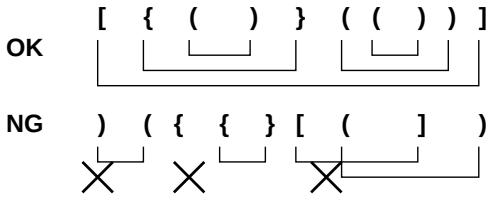


図 3.4: 括弧入れ子構造

とりあえず簡単のため、丸括弧だけで考えます。図 3.5 では、左側にスタックを横向き (右に伸びる) に描き、右側に入力文字列を描いています (最後の$は入力終わりの印のつもり)。↑は入力の位置です。

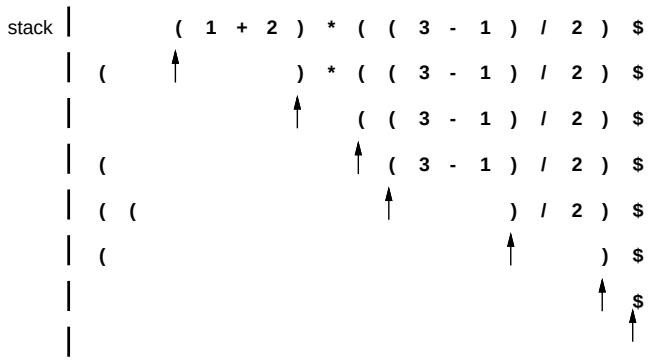


図 3.5: スタックを用いた括弧の対応チェック

開き括弧が来たら、それをスタックに積みます。閉じ括弧が来たら、スタックに対応する括弧が載っていることを確認して、それを降ろします。問題なく最後まで来た時にちょうどスタックが空なら、対応が取れていたと分かります。途中で空のスタックから降ろそうとしたり、閉じ括弧とスタック上の開き括弧の種類が違っていたら、間違いがあったと分かります (ここでは丸括弧しか出ていませんが)。

では上に説明したアルゴリズムを C プログラムにしたものを示します。文字列を受け取って対応を調べ成否を返す関数が `balance1` です。その中でスタックを作り、文字列の各文字をループで順番に

変数 c に取り出します。その文字が「'('」ならスタックに積み、「')'」であればスタックから降ろします。積んでいるのは「'('」だけなので、降ろすときにその文字が何であるかチェックする必要はありません。降ろそうとして空なら、対応があっていない (「')'」が余分) なので「いいえ」を返します。文字列の最後まで来たときはスタックが空なら対応が合っている (余分な「'('」が無い) ので、「空か否か」の論理値を返します。

main はコマンド引数の文字列 1 つずつを balance1 に渡し、結果を文字列「OK」「NG」で表示するだけです。

```
// barance1.c --- see if parentheses are balanced in input
#include <stdio.h>
#include <stdlib.h>
#include "istack.h"
bool balance1(char *t) {
  istackp s = istack_new(200);
  for(int i = 0; t[i] != '\0'; ++i) {
    char c = t[i];
    if(c == '(') {
      istack_push(s, c);
    } else if(c == ')') {
      if(istack_isempty(s)) { return false; }
      istack_pop(s);
    }
  }
  return istack_isempty(s);
}
int main(int argc, char *argv[]) {
  for(int i = 1; i < argc; ++i) {
    printf("%s : %s\n", argv[i], balance1(argv[i])?"OK":"NG");
  }
  return 0;
}
```

実行例を示します。

```
% ./a.out '((a))' '(a))'
((a)) : OK
(a)) : NG
```

これも単体テストを作成してみました。bool2str、expect_bool は前回と同じです。

```
// test_barance1.c --- unit test for balance1
#include <stdio.h>
#include <stdlib.h>
#include "istack.h"
char *bool2str(bool b) { return b ? "true" : "false"; }
void expect_bool(bool b1, bool b2, char *msg) {
  printf("%s %s:%s %s\n", (b1==b2)?"OK":"NG",
         bool2str(b1), bool2str(b2), msg);
}
```

```
(balance1 here)
int main(void) {
  expect_bool(balance1("((a)())"), true, "((a)())");
  expect_bool(balance1(")(a)()("), false, ")(a)()(");
  expect_bool(balance1("((a)()"), false, "((a)()");
  expect_bool(balance1("(a)())"), false, "(a)())");
  expect_bool(balance1("(((()))))"), true, "(((()))))");
  return 0;
}
```

実行例は次の通り。

```
% ./a.out
OK true:true ((a)())
OK false:false )(a)()(
OK false:false ((a)()
OK false:false (a)())
OK true:true ((((()))))
```

演習 1 括弧の対応プログラムをそのまま動かせ。正しい入力、正しくない入力、不自然だけど正しい入力などをそれぞれ複数試してみること。その後、次の課題をやってみよ。必ず単体テストを作成すること。

a. 括弧の種類として「`()`」に加え、「`[]`」と「`{}`」を扱えるようにしてみよ。ただし、どの開きかっことどの閉じかっこでも対応できるとしてよい (たとえば「`[(]}`」でも OK とする)。
b. 上と同じだが、同じ種類の開きかっこと閉じかっこが対応していなければいけないようにする (たとえば「`[(]}`」はだめで「`[()]`」は OK とする)。
c. 上と似ているが、それぞれのかっこは独立に対応をチェックする (かっこ同士が互い違いでもよい) ようにする (たとえば「`[(])`」は OK で「`[{)]`」はだめとする)。
ヒント: スタックを 3 本使う。
d. 上のどれでかで間違いが発見された際、その位置が分かるようにしてみよ。(ヒント: 簡単には間違い発見時にその位置が何文字目か表示したり、次の行にその文字数-1 個の空白の後印を出力する。ただしこの方法だと、括弧の対応が合わないときに該当の開き括弧がどれかは示せない。対応策としては、スタックをもう 1 つ用意して、開き括弧を積むときに何文字目かをの数値を並行して積むなどする。)
e. その他、括弧の対応プログラムに好きな改良を施してみよ。

3.3.2 中置記法の変換

先のアルゴリズムは括弧だけ扱って式のほうは無視していたので、今度は式を扱いましょう。私達が普段使う数式は、演算子を値 (部分式) の間に書きます。これを**中置記法** (infix notation) と呼びます。

```
3 * 2 + 4
3 + 2 * 4
```

中置記法は「演算子の結び付きが強いものを先に計算する」という慣習があったりして複雑です。たとえば上の例では「*」を先に計算しますね。ここで**後置記法** (postfix notation) を導入します。後置記法は、値の順番は中置記法と同じですが、演算子を「後に」書きます。その「後に」の規則は「その演算子に最も近い 2 つの値を計算して結果に置き換えることで、求める式が計算できる」ようにします。上の 2 つの例を後置記法にしてみましょう (後置記法は日本語に置き換えると読みやすいという説もあるのでそれもやっています)。

```
3 2 * 4 +  →  6 4 + → 10   「3と2を掛けたものに、4を足す」
3 2 4 * +  →  3 8 + → 11   「3と、2に4を掛けたものを、足す」
```

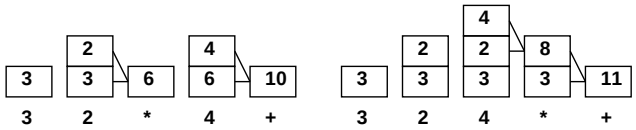


図 3.6: スタックを用いた後置記法の計算

そして、後置記法になった式はスタックを使って簡単に計算できます。計算のしかたは、(1) 数値はそのままスタックに積み、(2) 演算子は 2 つの値を降ろして来てその演算を行ない結果を積む、という動作を順にやるだけです (図 3.6)。

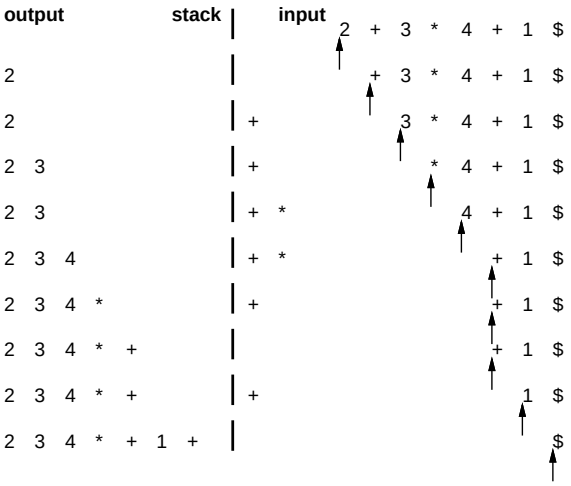


図 3.7: スタックを用いた中置後置変換

では、この変換を行なう方法を説明しましょう。まず、数値は順番が変わらないのでそのまま出力に送ります。演算子の場合の処理は次のようにします。

- スタックが空か、またはスタック先頭の演算子よりも入力先頭の演算子の方が結び付きが強いなら、入力先頭の演算子をスタックに積む。
- そうでない場合は、上記の条件が成り立たなくなるまで、スタックの演算子を降ろして出力に送る。

入力が終わりになったときも上記の 2 番目同様、残っている演算子は降ろして出力に送ります。これを行なう C のプログラムを示しましょう。演算子として「+」「-」を強さ 1、「*」「/」「%」を強さ 2、そしてべき乗のつもりの「^」を強さ 3 としました (実際の C 言語にはべき乗演算子はなく、「^」はビット毎排他的論理和演算子なので注意)。

また、ここまでは「出力に送る」と書いて来ましたが、実際には変換をおこなう関数 `postfix1` は入力の文字列と出力文字列を格納する文字配列 (実際には文字へのポインタ) を受け取ります。「送る」というのはですから、出力文字ポインタの位置に文字を書き込み、文字ポインタは次の位置に進める、ということです。これは C 言語では「`*u++ = 文字;`」という書き方で実現できます。文字列として扱えるようにするために、最後にナル文字を書き込む必要があります。

```
// postfix1.c -- convert infix to postfix (w/o parentheses)
#include <ctype.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#include "istack.h"
int operprec(int c) {
  switch(c) {
    case '+': case '-': return 1;
    case '*': case '/': case '%': return 2;
    case '^': return 3;
    default: return 0;
  }
}
void postfix1(char *t, char *u) {
  istackp s = istack_new(200);
  for(int i = 0; t[i] != '\0'; ++i) {
    char c = t[i];
    if(isdigit(c)) { *u++ = c; continue; }
    while(!istack_isempty(s) &&
          operprec(istack_top(s)) >= operprec(c)) {
      *u++ = istack_pop(s);
    }
    istack_push(s, c);
  }
  while(!istack_isempty(s)) { *u++ = istack_pop(s); }
  *u++ = '\0';
}
int main(int argc, char *argv[]) {
  char buf[200];
  for(int i = 1; i < argc; ++i) {
    postfix1(argv[i], buf);
    printf("%s => %s\n", argv[i], buf);
  }
  return 0;
}
```

なお、isdigit というのは ctype.h を include すると使えるようになる標準関数で、文字が数字 (0〜9) であるなら「はい」を返します。上のコードでは数字ならそれをコピーするだけでもう終わりなので、continue 文で次の周回に進んでいます。それ以外は演算子の場合で、そのロジックは上に説明した通りです。実行例を示します。

```
% ./a.out '1+2*3+4'
1+2*3+4 => 123*+4+
```

それでは単体テストを作ります。今回はそもそも、文字列が結果なので、expect_str を作るところからやります。文字列の比較にはライブラリの strcmp を使うので、string.h の include が必要です。strcmp は「等しいと 0 を返す」ので、文字列 OK と NG の位置がこれまでと反対です。

```
// test_postfix1.c --- unit test of postfix1
#include <ctype.h>
#include <stdio.h>
#include <string.h>
```

```
#include "istack.h"
(operprec, postfix1 here)
void expect_str(char *s1, char *s2, char *msg) {
  printf("%s '%s':'%s' %s\n", strcmp(s1, s2)?"NG":"OK", s1, s2, msg);
}
int main(void) {
  char buf[200];
  postfix1("1+2*3", buf); expect_str(buf, "123*+", "1+2*3 => 123*+");
  postfix1("2*3+1", buf); expect_str(buf, "23*1+", "2*3+1 => 23*1+");
  return 0;
}
```

演習 2 上の中置後置変換の例題をそのまま動かせ。さまざまな式を変換して動作を確認すること。終わったら次のことをやってみよ。単体テストすること。

a. 「2^3^4」のようにべき乗演算が連続している場合、現在は $(2^3)^4$ のように左から計算されるが、慣習としては $2^{(3^4)}$ のように右から計算されるべきである。そのように修正してみよ (その結果、他の演算も並んでいるときに右から計算されるようになったとしても、まあよいことにする)。

b. 上の中置後置変換プログラムは括弧に対応していない。括弧が使えるように修正してみよ。(ヒント: 括弧対応プログラムのように開き括弧を積み、閉じ括弧のところで対応する開き括弧が来るまで降ろして出力する。)

c. 後置記法に変換するだけでなく、その変換結果をもとにスタックで値を計算して式の値を求めるようにせよ。入力に現れる数は 1 桁だが、結果は 2 桁以上であってよい。

d. 計算機能に加えて変数への代入機能と変数参照機能を追加してみよ。この場合、入力は複数行にわたってよく、空行を入力すると終了するようにするのがよい (詳細は任せる)。

e. その他、中置後置変換プログラムに好きな改良を施してみよ。

3.4　文字列の扱い

3.4.1　ファイルの上下逆転

ここまでは整数を積むスタックを扱って来ましたが、たとえば実数値など、他の種類のデータを扱うこともちろんできます。ここでは文字列を扱う例を見てみましょう。文字列は文字ポインタですから、まずポインタ値を扱える版のスタックを定義します。ヘッダファイルは次の通り。

```
// pstack.h --- pointer type stack interface
#include <stdbool.h>
struct pstack;
typedef struct pstack *pstackp;
pstackp pstack_new(int size);
bool pstack_isempty(pstackp p);
bool pstack_isfull(pstackp p);
void pstack_push(pstackp p, void *v);
void *pstack_pop(pstackp p);
void *pstack_top(pstackp p);
```

扱う値が void*型になっただけですね。なお、実際には様々なポインタを扱うわけですが、それは後で見るように、void*との間でキャストして扱います。実装の方も示しましょう。

```
// pstack.c --- pointer type stack impl. with array
#include <stdlib.h>
#include "pstack.h"
struct pstack { int ptr, lim; void **arr; };
pstackp pstack_new(int size) {
  pstackp p = (pstackp)malloc(sizeof(struct pstack));
  p->ptr = 0; p->lim = size; p->arr = (void**)malloc(size * sizeof(void*));
  return p;
}
bool pstack_isempty(pstackp p) { return p->ptr <= 0; }
bool pstack_isfull(pstackp p) { return p->ptr >= p->lim; }
void pstack_push(pstackp p, void *v) { p->arr[p->ptr++] = v; }
void *pstack_pop(pstackp p) { return p->arr[--(p->ptr)]; }
void *pstack_top(pstackp p) { return p->arr[p->ptr - 1]; }
```

値が void*なので、それを並べて格納する配列は型としては void**になることに注意してください。では、これを使ってファイルの上下逆転をやってみます。

```
// reversefile.c --- input a flie and output upside-down
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "pstack.h"
bool getl(char s[], int lim) {
  int c, i = 0;
  for(c = getchar(); c != EOF && c != '\n'; c = getchar()) {
    s[i++] = c; if(i+1 >= lim) { break; }
  }
  s[i] = '\0'; return c != EOF;
}
int main(void) {
  char buf[200];
  pstackp s = pstack_new(200);
  while(getl(buf, 200)) {
    char *t = (char*)malloc(strlen(buf)+1);
    strcpy(t, buf); pstack_push(s, t);
  }
  while(!pstack_isempty(s)) {
    char *t = (char*)pstack_pop(s); printf("%s\n", t); free(t);
  }
  return 0;
}
```

getl は 1 行読み込み、文字列の最後にナル文字を入れます。そして最後に EOF(ファイルの終わり) でないかどうかを返します。[2]main の方ではスタックを用意し、getl で読めたらそれを積みます…が、buf は次の行を読み込むのに使う領域ですから、文字列の長さ+1(ナル文字のぶん) の領域を割り当て、そこにコピーして、その領域のポインタを積みます。終わりまで積み終わったら、今度は順

[2]なぜ for の先頭での変数宣言をしていないかというと、変数 c も i も for が終わった後でも参照するからです。

次降ろしながら出力しますが、出力したらその行はもう使わないので free しています。動作例を示します。

```
% ./a.out
this is a pen.
that is a dog.
how are you?
^D   ← Ctrl-D で終わりの印
how are you?
that is a dog.
this is a pen.
```

演習 3 スタックに「現在格納している値の個数」を調べる機能と、「満杯であるかどうか調べる機能」を追加してみよ。単体テストを作成して確認すること。

3.4.2　行バッファの実装

エディタなどでの内部では、行の並びを扱い、任意位置での挿入や削除ができるようなデータ構造が必要とされます。1 つの選択肢は連結リストを使うことですが、スタックを 2 つ「向かい合わせで」使うことで、そのようなデータ構造が作れます。その概要は次の通りです。

- 「現在位置」は 2 つのスタックの「間」にあるものと考え、2 つのスタックはそれぞれ現在位置より上にあるものと、下にあるものが入っていると考える。
- 現在位置を 1 つ下や上に動かすことは、下のスタックから 1 つ降ろして上のスタックに積んだり、その逆をすることで行なえる。
- 現在位置の上に行を挿入するには、単にその行の内容を上のスタックに積めばよい。
- 現在位置の直後の行を削除するには、単に下のスタックから 1 行降ろせばよい。

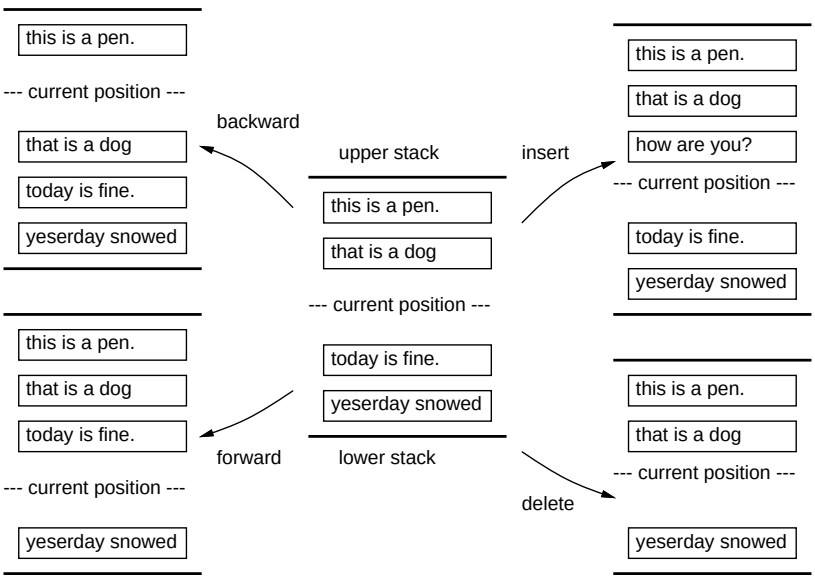


図 3.8: 2 つのスタックを用いた編集バッファ

演習 4 スタックを 2 つ使って行バッファの機能を実現してみよ。行バッファもレコード型を使ってカプセル化できた方がよい。単体テストを作成して確認すること。

演習 5 前問の行バッファを利用してテキストエディタを作成してみよ。機能は自由に設計してよい。

演習 6 スタックを使って何か面白いプログラムを作れ。

本日の課題 3A

「演習 1」〜「演習 6」で動かしたプログラム 1 つを含むレポートを本日中 (授業日の 23:59 まで) に提出してください。

1. 学籍番号、氏名、ペアの学籍番号 (または「個人作業」)、提出日時。名前の行は先頭に「@@@」を付けることを勧める。
2. プログラムどれか 1 つのソースと「簡単な」説明。
3. レビュー課題。提出プログラムに対する他人 (ペア以外) からの簡単な (ただしプログラムの内容に関する) コメント。
4. 以下のアンケートの回答。

   Q1. スタックとその働きについて理解しましたか。
   Q2. 「かっこの対応検査」や「後置記法への変換」について納得しましたか。
   Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

次回までの課題 3B

「演習 1」〜「演習 6」(ただし 3A で提出したものは除外、以後も同様) の (小) 課題から選択して 2 つ以上プログラムを作り、レポートを提出しなさい。できるだけ複数の演習から選ぶこと。レポートは次回授業前日 23:59 を期限とします。

1. 学籍番号、氏名、ペアの学籍番号 (または「個人作業」)、提出日時。名前の行は先頭に「@@@」を付けることを勧める。
2. 1 つ目の課題の再掲 (どの課題をやったか分かればよい)、プログラムのソースと「丁寧な」説明、および考察 (課題をやってみて分かったこと、分析、疑問点など)。
3. 2 つ目の課題についても同様。
4. 以下のアンケートの回答。

   Q1. スタックがさまざまなことに役立つことを納得しましたか。
   Q2. 文字列の取り扱いはどれくらいできるようになりましたか。
   Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

＃4　スタック・キューと探索

今回は次のことが目標となります。

- CS における基本的なデータ構造であるキュー、デックについて知る。
- スタックとキューを用いた探索アルゴリズムについて知る。

4.1　キューとその実装

4.1.1　キューの概念

スタックと対をなす重要なデータ構造として、**キュー** (queue) があります。スタックが LIFO(last-in, first-out) の記憶領域なのに対して、キューは **FIFO**(first-in, first-out) の記憶領域です。実は私達の日常では、キューの方が一般的に見られます。何かのサービスのために並ぶときは、最初に来た人が最初にサービスを受けますね。これが FIFO ということになります (図 4.1)。

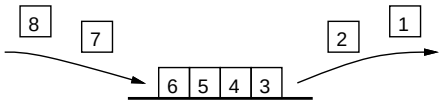


図 4.1: キューの概念

キューも配列を使って実装することができます。ただし、スタックは「入れる」のと「出す」のが同じ側だったので出し入れ位置を 1 つ覚えておけば済んだのですが、キューでは「入れ」と「出し」で位置が違うので、2 つの位置を用いる必要があります。ここでは入れる場所を指す変数は `ip`(input pointer のつもり)、出す場所を指す変数は `op`(output pointer のつもり) という名前にしています (図 4.2)。

さらに、配列を使った場合、出し入れが進むとデータの入っている位置が配列の最後まで来てしまうので、そのときは先頭に戻って続ける必要があります。これを、「最後と先頭が (論理的には) くっついているものとして扱う」ことから、**リングバッファ**(ring buffer、環状バッファ) とも呼びます。

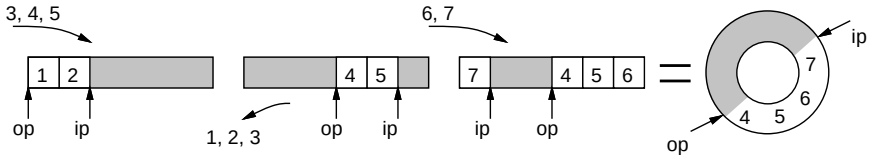


図 4.2: 配列を用いたキューの実装

4.1.2　配列を使ったキューの実装

では、配列を使ったキューの実装を見てみましょう。キューではデータを追加するのが enqueue、取り出すのが dequeue ですが、長いので enq、deq としています。今度はスタックと違い、空っぽのほかに満杯もチェックできるようにします。

```
// iqueue.h --- int type queue interface
#include <stdbool.h>
```

```
struct iqueue;
typedef struct iqueue *iqueuep;
iqueuep iqueue_new(int size);
bool iqueue_isempty(iqueuep p);
bool iqueue_isfull(iqueuep p);
void iqueue_enq(iqueuep p, int v);
int iqueue_deq(iqueuep p);
```

実装を見てみましょう。配列の大きさを size フィールドに覚えておき、size になったら 0 に戻すようにすれば「論理的に端がつながる」ようにできます。

```
// iqueue.c --- int type queue impl. with array
#include <stdlib.h>
#include "iqueue.h"
struct iqueue { int ip, op, size; int *arr; };
iqueuep iqueue_new(int size) {
  iqueuep p = (iqueuep)malloc(sizeof(struct iqueue));
  p->ip = p->op = 0; p->size = size;
  p->arr = (int*)malloc(size * sizeof(int)); return p;
}
bool iqueue_isempty(iqueuep p) { return p->ip == p->op; }
bool iqueue_isfull(iqueuep p) { return (p->ip+1)%p->size == p->op; }
void iqueue_enq(iqueuep p, int v) {
  if(iqueue_isfull(p)) { return; }
  p->arr[p->ip++] = v; if(p->ip >= p->size) { p->ip = 0; }
}
int iqueue_deq(iqueuep p) {
  if(iqueue_isempty(p)) { return 0; }
  int v = p->arr[p->op++]; if(p->op >= p->size) { p->op = 0; }
  return v;
}
```

空っぽと満杯のチェックですが、空っぽは ip と op が一致しているときが相当します。満杯は ip が op に追い付きそう (1 つ増やしたら一致) なときです (size で剰余を取ることで端まできたら 0 に戻るようにしています)。

なお、この方法だと、必ず 1 個は場所をあけておく必要があります。全部入れてしまうと ip と op が一致するので、すべて空っぽのときと区別がつきません。別の方法として、「何個入っているか」を数えておくなら、全部入れるようにできます。

iqueue_enq は満杯のときには何もしないで戻ります。また、iquque_deq は空っぽのときは何も操作をせず 0 を返します。本来なら enq、deq を呼ぶ前に呼ぶ側で満杯や空っぽのチェックをするべきですが、それを怠った場合におかしな状態にならないように上記のようにしています。

ではこれを使ってみることにして、3 行入力してそれをくっつけて出力、というのをやってみましょう。

```
// queuetest -- demonstration of iqueue
#include <stdio.h>
#include <stdlib.h>
#include "iqueue.h"
```

```
void readenq(iqueuep q) {
  int c;
  printf("s> ");
  for(c = getchar(); c != '\n' && c != EOF; c = getchar()) {
    iqueue_enq(q, c);
  }
}
int main(void) {
  int c;
  iqueuep q = iqueue_new(200);
  readenq(q); readenq(q); readenq(q);
  while(!iqueue_isempty(q)) { putchar(iqueue_deq(q)); }
  putchar('\n');
  return 0;
}
```

1 行入力する関数を作って、それを 3 回呼ぶことで 3 行ぶん入力しています。動かしたようすは次の通り。

```
% gcc8 queuetest.c iqueue.c
% ./a.out
s> this is
s>  a
s>  pen.
this is a pen.
```

単体テストも作っておきます。満杯でいれると入らないとか、空っぽだと 0 が出て来るとかも試すようにしました。

```
// test_iqueue.c --- unit test for iqueue
#include <stdbool.h>
#include <stdio.h>
#include "iqueue.h"
(bool2str, expect_bool, expect_int here)
int main(void) {
  iqueuep q = iqueue_new(4);
  iqueue_enq(q, 1); iqueue_enq(q, 2); iqueue_enq(q, 3);
  expect_bool(iqueue_isfull(q), true, "size=4 queue full");
  iqueue_enq(q, 4);
  expect_int(iqueue_deq(q), 1, "1st => 1");
  iqueue_enq(q, 5);
  expect_int(iqueue_deq(q), 2, "2nd => 2");
  expect_int(iqueue_deq(q), 3, "3rd => 3");
  expect_bool(iqueue_isempty(q), false, "queue not empty");
  expect_int(iqueue_deq(q), 5, "4th => 5");
  expect_bool(iqueue_isempty(q), true, "queue emptied");
  expect_int(iqueue_deq(q), 0, "0 returned for empty");
  return 0;
}
```

演習 1 上の例題をそのまま動かして動作を確認しなさい。動いたら、次のような変更をしてみなさい。いずれも単体テストを作成すること。いくつかの課題はキューのサイズを小さくした方が確認しやすい。

a. キューに「今入っている個数」を調べる機能を追加しなさい。またそれを利用して、配列のサイズ一杯まで格納できるようにしなさい
b. キューに「次に取り出されるものを取り出さずにのぞき見る」機能を追加しなさい。
c. 上の例ではキューが満杯のとき入れないようになっていたが、代わりに「一番古いものを捨てて新しいものを入れる」動作に変更しなさい。
d. キューが満杯のとき入れようとしたら「配列を大きいものに取り換えてそちらにコピーして続ける」ようにしなさい。

4.1.3　デック

ここまでに学んだスタックとキューについて整理すると、スタックとは先頭側だけで追加と取り出しができる列、キューは先頭側だけで追加でき、末尾側だけで取り出しができる列でした。しかし、用途によってはもっと自由にしてもよい場合があります。そのような場合にデックを使います。**デック** (DEQ、double-ended queue) とは、先頭側でも末尾側でも追加、取り出しが可能な列です (図 4.3)。

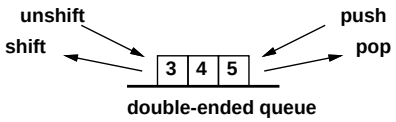


図 4.3: DEQ の概念

ここでは先頭側の追加と取り出しを `push` と `pop`、末尾側での追加と取り出しを `unshift` と `shift` と名付けています (Ruby の配列もこれらのメソッドを持っているのでそれに合わせました)。`push` と `pop` を使えばスタック、`push` と `shift` を使えばキューとして動作させられます。

演習 2 デックの実装を作成しなさい (先の例題のキューを元にするのがよい)。単体テストを実施すること。

4.2　スタック・キューと探索

4.2.1　グラフの表現

スタックやキューを使うアルゴリズムの例として、グラフの探索を取り上げます。ここでは**グラフ** (graph) とは、ノード (node、節) とノード間を結ぶ辺 (edge) から成るような図形です。グラフの例として、図 4.4 にとある鉄道路線図を示します (架空のものです)。

この上で、「横浜から赤羽に行くのにどのような経路で行けるか」という問題を解いてみましょう。そのためにはまず、このグラフをプログラム上でデータ構造として表す必要があります。ここでは、それぞれの節を構造体で表し、構造体の配列でグラフを表します。そうすれば、配列の何番目かという整数でノードを指定できます (その番号は図にも記入してあります)。では、そのデータで初期化した構造体の配列を用意する部分を見てみましょう (ヘッダファイルとして取り込むつもりです)。

```
// railmap.h -- a railroad map
struct node { char *name; int num, edge[5], dist; };
struct node map[13] = {
  { "yokohama", 3, { 1, 3, 4 }, -1 },    // 0
  { "kawasaki", 3, { 0, 2, 5 }, -1 },    // 1
```

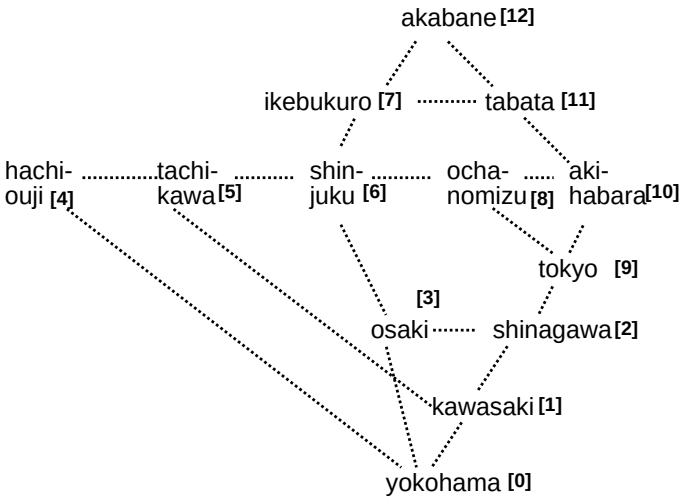


図 4.4: とある鉄道路線図

```
  { "shinagawa", 3, { 1, 3, 9 }, -1 },   // 2
  { "osaki", 3, { 0, 2, 6 }, -1 },       // 3
  { "hachiouji", 2, { 0, 5 }, -1 },      // 4
  { "tachikawa", 3, { 1, 4, 6 }, -1 },   // 5
  { "shinjuku", 4, { 3, 5, 8, 7 }, -1 }, // 6
  { "ikebukuro", 3, { 6, 11, 12 }, -1 }, // 7
  { "ochanomizu", 3, { 6, 9, 10 }, -1 }, // 8
  { "tokyo", 3, { 2, 8, 10 } , -1 },     // 9
  { "akihabara", 3, { 8, 9, 11 }, -1 },  // 10
  { "tabata", 3, { 7, 10, 12 }, -1 },    // 11
  { "akabane", 2, { 7, 11 }, -1 },       // 12
};
```

ノード (駅に対応) の構造体には、駅名の文字列、駅から出ている辺 (路線) の数、それぞれの辺がつながる先の駅番号の並び (配列)、そして、その駅が出発点から何の距離 (ここでは通る辺の数とします) で来たかの番号を入れるものとします。そのような構造体定義がまずあり、つづいて変数 `map` が構造体の配列で、その 1 つずつの要素としてそれぞれの駅のデータを初期値として書いています。出発点からの距離は最初は不明なので -1 です。

4.2.2 探索アルゴリズム

グラフのデータを参照しながら出発地から目的地までの経路を探索するコードですが、アルゴリズムとしては (スタックを使う場合) 次のようになります。

- 出発点を距離 0 としてスタックに積む。
- スタックが空でない間繰り返し、
- ノードを 1 つスタックから取り出す。
- ノードが目的地なら、成功して終了。
- そのノードから 1 ステップで行ける各ノード n について繰り返し、
- n が未訪問 (距離が -1) なら、
- n の距離を現在の節までの距離+1 とし、n をスタックに積む。
- 枝分かれ終わり。
- 繰り返し終わり。

- 繰り返し終わり。
- 目的地まで到達する経路は無かったとして終了。

このアルゴリズムでスタックを使う意味は、「積んだものはいつかは取り出されて処理される」ところにあります。ある節から 1 ステップで行ける節は複数あるわけですが、それを一辺に処理するのは (その節の先にさらに節があることを考えれば) 困難です。なので、それらの節をスタックに積んでしまい、1 つずつ取り出して処理することを繰り返すようにしています。ただしそれだけだと、同じ節を何回も積んでしまって終わらなくなりますから、一度処理した節 (今回の場合は距離が初期値 −1 でなくなっているもの) は積まないようにします。そうすることで、到達可能な節をすべて処理対象にできるわけです。

では、上のアルゴリズムをプログラムにしたものを示します。出発値と目的地の番号をコマンド引数で与えます。

```
// railmap.c -- search railroad paths
#include <stdio.h>
#include <stdlib.h>
#include "istack.h"
#include "railmap.h"

int main(int argc, char *argv[]) {
  int start = atoi(argv[1]), goal = atoi(argv[2]);
  istackp s = istack_new(100);
  map[start].dist = 0; istack_push(s, start);
  while(!istack_isempty(s)) {
    int i, n = istack_pop(s);
    struct node *p = map + n;
    printf("%d: %s, %d\n", n, p->name, p->dist);
    if(n == goal) { printf("GOAL.\n"); break; }
    for(i = 0; i < p->num; ++i) {
      int k = p->edge[i];
      if(map[k].dist < 0) {
        map[k].dist = p->dist+1; istack_push(s, k);
      }
    }
  }
  return 0;
}
```

では動かしてみましょう。確かに横浜から赤羽まで到達しています (最短の経路ではないですが)。

```
% gcc8 railmap.c istack.c
% ./a.out 0 12
0: yokohama, 0
4: hachiouji, 1
5: tachikawa, 2
6: shinjuku, 3
8: ochanomizu, 4
10: akihabara, 5
11: tabata, 6
```

```
12: akabane, 7
GOAL.
```

演習 3 上の例題をそのまま動かせ (ファイルは railmap.c、railmap.h、istack.h、istack.c の 4 つになる)。動いたら別の出発地と目的地で動かしてみよ。さらに、路線を追加してみよ。例の場合は「行き止まり」駅は無かったが、行き止まりがあったらどうなるか? まず予測し、次に動かしてみて確認し、そのようになる理由を検討すること。

4.2.3 深さ優先と幅優先

グラフや (ループのないグラフである) 木構造をたどるときに、前節のようにスタックを使うと、まずどんどん行けるところまで行って、行き止まりになったら最も直近の枝分かれまで戻って別の道に行って、というたどり方になります。これを**深さ優先** (depth first) のたどり、と呼びます (図 4.5)。これはスタックが LIFO つまり「最も直近に積んだものが出て来る」記憶領域であるためにそうなるのです。

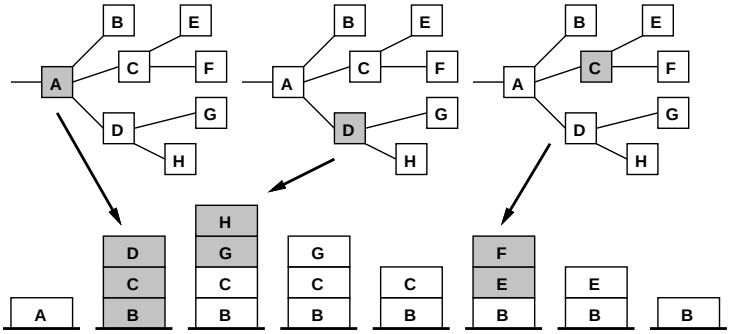


図 4.5: スタックと深さ優先のたどり

ここで、アルゴリズムの骨子は同じままで、FIFO の領域すなわちキューを使うと、まず出発点から 1 ステップで行けるところを全部たどり、次に 2 ステップで行けるところを全部たどり、という風に進んで行きます (図 4.6)。これを**幅優先** (breadeth first) のたどり、と呼びます。

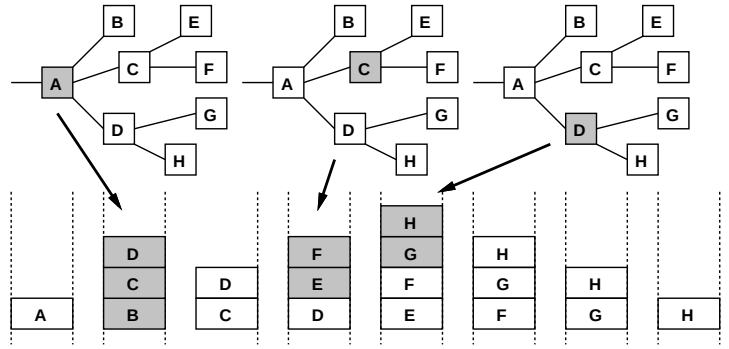


図 4.6: キューと幅優先のたどり

グラフの探索の場合、一般に深さ優先の方が「どんどん先の方まで行ってみる」ため、目的ノードが早く見つかる傾向があります。これに対し、幅優先は「レベル 1、レベル 2、レベル 3…」と網羅的に探すため、探すノードの個数は多くなりますが、見つかった経路が一番短い経路であることが保証されます (図 4.7)。

先の鉄道路線図の問題で、スタックをキューに置き換えただけのバージョンを用意してみます。

```
// railmap2.c -- search railroad paths (queue version)
```

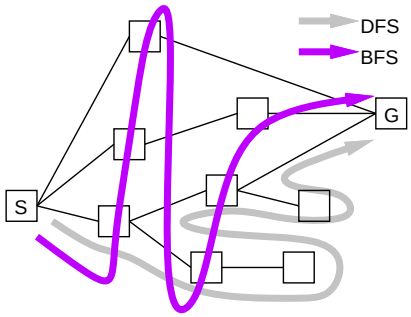


図 4.7: 深さ優先と幅優先の対比

```
#include <stdio.h>
#include <stdlib.h>
#include "iqueue.h"
#include "railmap.h"

int main(int argc, char *argv[]) {
  int start = atoi(argv[1]), goal = atoi(argv[2]);
  iqueuep s = iqueue_new(100);
  map[start].dist = 0; iqueue_enq(s, start);
  while(!iqueue_isempty(s)) {
    int i, n = iqueue_deq(s);
    struct node *p = map + n;
    printf("%d: %s, %d\n", n, p->name, p->dist);
    if(n == goal) { printf("GOAL.\n"); break; }
    for(i = 0; i < p->num; ++i) {
      int k = p->edge[i];
      if(map[k].dist < 0) {
        map[k].dist = p->dist+1; iqueue_enq(s, k);
      }
    }
  }
  return 0;
}
```

これを動かしてみると、確かに前よりも多数の駅を調べていますが、経路の長さは 2 つ短い「5」となっています (そしてこれより短い経路はありません)。[1]

```
% gcc8 railmap2.c iqueue.c
% ./a.out 0 12
1: kawasaki, 0
0: yokohama, 1
2: shinagawa, 1
5: tachikawa, 1
3: osaki, 2
4: hachiouji, 2
```

[1]なお、ここでの長い短いは「通過した辺の数」で言っています。実際の地図や路線図では、所要時間や道のりが基準になるので、辺の数では済まないのが普通です。

```
9: tokyo, 2
6: shinjuku, 2
8: ochanomizu, 3
10: akihabara, 3
11: tabata, 4
7: ikebukuro, 5
12: akabane, 5
GOAL.
```

演習 4 上の例題をそのまま動かせ (ファイルは `railmap2.c`、`railmap.h`、`iqueue.h`、`iqueue.c` の 4 つになる)。動いたら別の出発地と目的地で動かしてみよ。さらに、路線を追加してみよ。「行き止まり」の有無は動作に関係するか検討せよ。

演習 5 幅優先の例題では (そして深さ優先でも行き止まりがある場合は)、たどった経路をそのまま出力するだけでは最終的に見つかった経路がどのような経路かは分からない。経路が見つかったらその経路を表示するような改良を施してみなさい (幅優先でも深さ優先でもどちらでもよい)。

演習 6 スタックまたはキューを用いて探索を行う興味深いプログラムを作成せよ。題材は自由に選んでよい。

本日の課題 4A

「演習 1」〜「演習 5」で動かしたプログラム 1 つを含むレポートを本日中 (授業日の 23:59 まで) に提出してください。

1. 学籍番号、氏名、ペアの学籍番号 (または「個人作業」)、提出日時。名前の行は先頭に「@@@」を付けることを勧める。
2. プログラムどれか 1 つのソースと「簡単な」説明。
3. レビュー課題。提出プログラムに対する他人 (ペア以外) からの簡単な (ただしプログラムの内容に関する) コメント。
4. 以下のアンケートの回答。

   Q1. キューやデックがどのようなものか納得しましたか。
   Q2. 路線図の情報を C 言語で表現するやり方は理解しましたか。
   Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

次回までの課題 4B

「演習 1」〜「演習 6」(ただし 4A で提出したものは除外、以後も同様) の (小) 課題から選択して 2 つ以上プログラムを作り、レポートを提出しなさい。できるだけ複数の演習から選ぶこと。レポートは次回授業前日 23:59 を期限とします。

1. 学籍番号、氏名、ペアの学籍番号 (または「個人作業」)、提出日時。名前の行は先頭に「@@@」を付けることを勧める。
2. 1 つ目の課題の再掲 (どの課題をやったか分かればよい)、プログラムのソースと「丁寧な」説明、および考察 (課題をやってみて分かったこと、分析、疑問点など)。
3. 2 つ目の課題についても同様。
4. 以下のアンケートの回答。

   Q1. スタックやキューを使ってグラフ上の経路を探索するやり方を理解しましたか。

Q2. 深さ優先と幅優先の違いについて納得しましたか。
Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

＃5　再帰呼び出しとその実装

今回は次のことが目標となります。

- 変数の可視範囲/存在期間と引数渡しについて理解する。
- 再帰呼び出しとその実装方法を理解する。

5.1　変数とその実現

5.1.1　変数の可視範囲

変数の**可視範囲** (**スコープ**、scope) とは、その変数の名前を指定して変数にアクセスすることができるコード上の範囲を言います。例として、図 5.1 を見てください。

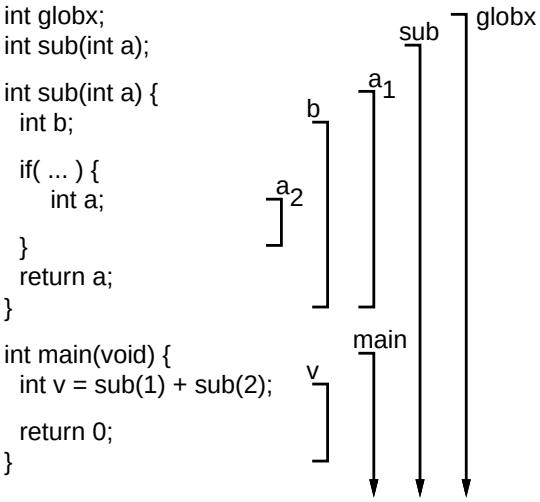


図 5.1: スコープの図解

C 言語ではこれまで見てきたように、変数や関数は宣言/定義した箇所から後ろで使えます。ですから、たとえばグローバル変数 `globx` がファイルの先頭で宣言されていれば、それはその先ファイルの終わりまでがずっと使える範囲つまりスコープです。関数 `sub` はプロトタイプ宣言があるので、それ以降がスコープです。`main` は定義しかないので、定義の先頭以降がスコープです。

では、ローカル変数はどうでしょうか。ローカル変数は関数内で使えるものですから、ローカル変数 `b` のスコープは関数 `sub` の中になります。また、パラメタ `a` も関数の先頭で宣言されているわけで、同様です。

この先がややこしいのですが、C 言語ではブロック (「{…}」で囲まれた範囲) の中で宣言された変数のスコープはそのブロックの終わりまでとなっています。ですから、内側の if のブロック中にある `a`(区別できるように添字をつけて a_2 としました) はそのブロックの末尾までがスコープです。ということは、パラメタ a_1 のスコープの中でこの範囲だけは「穴」があいているわけです。これを (内側の変数が外側の変数を「隠す」ことから) **シャドウ** (shadow) する、と言います。

関数末尾の return のところでは、内側の a_2 のスコープは終わっていますから、ここで返す値は a_1 の方ということになります。

`main` 内のローカル変数 `i` については、そこから関数の終わりまでがスコープになります。

5.1.2　変数の存在期間

変数の**存在期間** (**エクステント**、extent) とは、その変数が存在している時間的な範囲のことを指します。グローバル変数の存在期間はプログラムが始まった時点から終了する時点までずっとです。スコープから外れている間も (たとえばローカル変数で `globx` という名前のものを定義すれば、そのスコープでは外側のグローバル変数はシャドウされてスコープ外になります)、変数自体は存在し続けていることに注意。

そして、ローカル変数やパラメタは宣言された箇所 (パラメタでは関数の先頭) を実行する時点から、そのブロック (関数本体のブロックも含む) を出るところまでが、エクステントとなります。エクステントを出るところでは変数が無くなるのですから、そのあと再度エクステントに入った場合に、前の値が残っているというわけには行きません。

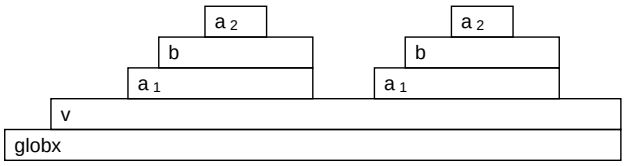


図 5.2: エクステントの図解

図 5.2 に先のコードのエクステントの推移を図示しました。仮に `main` から `sub` を 2 回呼び出したものとしています。そのため、`sub` のパラメタやローカル変数は 2 回に分けて現れます。

なお、a_2 については「if 文の条件が成り立って中を実行した時にはじめて」存在するようになることに注意。このように、エクステントは動的な (実行してみて始めて分かる) 性質を持ちます。一方、スコープは「どの箇所ではどの変数が見えているか」ということなので静的な (実行しなくても/コンパイル時に分かる) 性質を持ちます。

ところで図 5.2 を見ると、内側の変数ほど後で現れ、先に無くなるので、ちょうどスタックに変数を積んだり降ろしたりしているように見えます。これはブロックが入れ子構造になっていることから自然にそうなるのです。そして実際、変数の領域の管理は言語処理系の内部ではスタックを用いて行なうことが普通です。

5.1.3　実行時スタックと戻り番地

前節末で述べたように、言語処理系の内部では変数等をスタックで管理します。このスタックを**実行時スタック** (runtime stack) と呼びます。その様子をもう少し詳しく見て見ましょう。

図 5.3 のように、プログラムが実行を開始したところでは、`main` の変数 `i` だけが実行時スタックに割り当てられています (そのほかに空白や灰色の箇所もありますが、これは制御情報として使われています)。そして、そこから `sub` が呼ばれると、実行時スタック上に `sub` が使う 3 つの変数 (パラメタを含む) の領域が取られます。そして、`sub` から戻るとまた最初と同じになり、再度 `sub` が呼ばれるとまた 3 つの変数の領域が取られます。

なんだ、いつも同じ場所では、と思ったかも知れませんがそうではないのです。仮にそのあとで `main` が `sub2` という関数を呼び、その関数が `sub` を呼んだとすると、`sub2` の変数 (ここでは x、y としています) が取られ、その上に `sub` の 3 つの変数が取られます。スタックなので、常に最後に割り当てられた領域が最初に開放されるという形で進んで行きます。

この、1 つの関数実行に対応するスタック上の領域のことを**スタックフレーム** (stack frame) と呼びます。スタックフレームには、その関数のローカル変数と制御情報が含まれます。しかし制御情報って何でしょうか? いくつかありますが、ここでは最も重要な**戻り番地** (return address) だけ説明しておきます。

プログラムが CPU 命令の列に翻訳されて実行されるとき、命令はメモリ上に順番に並べられていて、それを CPU が上から順に実行していきます。分岐や反復のところはまた別ですが、ともかく 1

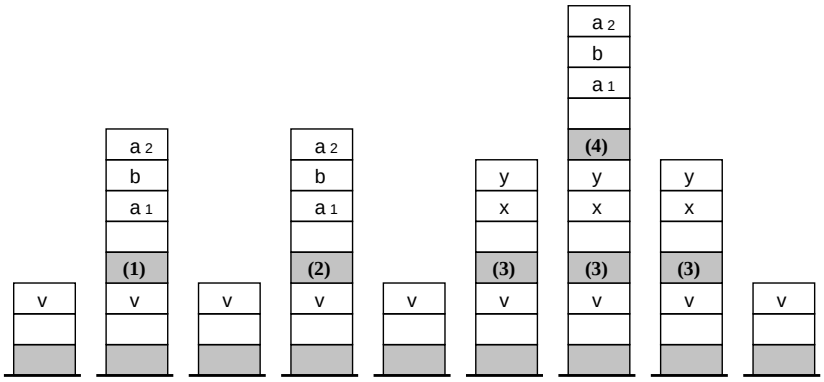


図 5.3: 実行時スタックの推移

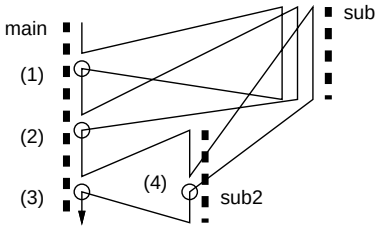


図 5.4: 実行の一筆描きと戻り番地

命令ずつ「一筆描き」のようにして実行が進みます。

図 5.4 では 3 つの関数の命令列を太い点線で示しています。ここで main から実行開始し、sub を呼ぶと実行は sub の先頭に移り、その先頭から順に実行が進みます。そして sub の最後まで来ると… 今度は、main 中のさっき sub を呼んだ命令の「次の」命令に戻り、そこから実行を続けます。もう 1 回 sub を呼んだ時も同様ですが、戻って来るのはその呼んだ命令の次ですから、さっき戻った位置とは違います。

そして次に sub2 を呼び、sub2 から sub を呼ぶと、今度は戻って来るのは sub2 の中の「呼んだ次の」命令です。そして sub2 から戻る位置は main 中の—tt sub2 を呼んだ次の命令です。

ということは、呼んだ時にこれらの「次の」命令の位置 (図では○で表しています)、というかメモリ番地を覚えておく必要があるわけです。これが戻り番地です。そしてそれをどこに覚えておくかというと、実行時スタックに覚えておくわけです。それが先の図では灰色で表された箇所になります (それぞれの位置に対応した番号が記入してあります)。[1]

5.1.4 引数と引数渡し

ここで**引数** (parameter ないし argument) についてもう少し見ておきます。関数定義の先頭には**仮引数** (formal parameter) のリストがあり、ここで引数の個数と型、およびそれぞれの引数の名前が定義されています。そして、実際に関数を呼び出すところでは、それぞれの仮引数に渡す式のリストを与えます (図 5.5 左)。これを**実引数** (actual parameter) と呼びます。

C 言語をはじめ多くの言語では、実引数のそれぞれの式の結果 (値) が、対応する仮引数の初期値として渡されます。関数の中では、仮引数は初期値の格納されたローカル変数としてふるまいます。これを**値渡し** (call by value) の**引数機構** (parameter mechanism) と呼びます。

これがどのように実装されるかを見てみましょう (図 5.5 右)。main の中で実引数の値を計算し、スタックに置きます。続いて sub3 を呼ぶと、sub3 の中では戻り番地よりも下にある引数を他のローカル変数と同じように読み書きして計算を進めます。つまり引数は関数を呼ぶ側のスタックフレームと

[1] main にも対応する戻り番地があることに気がついた方がいるかも知れません。これは、OS がプログラムを起動して main を呼び出すコード中の番地で、そこへ戻ると単にプログラムを終了させます。

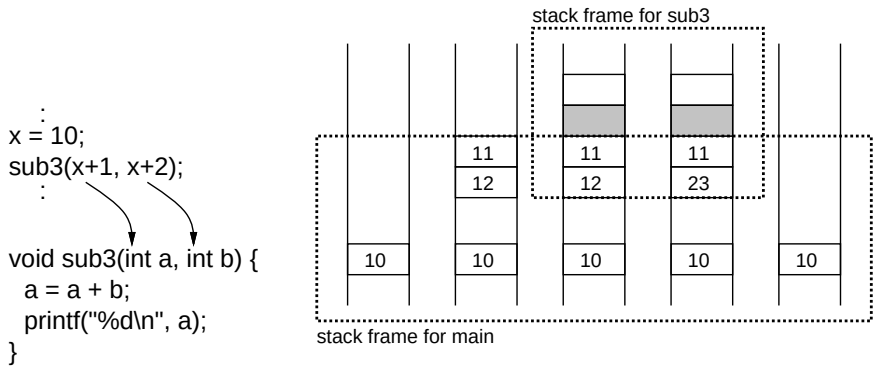


図 5.5: 引数と値渡しの実装

呼ばれた関数のスタックフレームが共有する部分になるわけです。このような、関数呼び出し時に何をどこに置いてどのように渡すかの取り決めを**呼び出し規約** (calling convention) と呼びます。呼び出し規約には**返値** (return value) の受け渡し方も含まれますが、返値は 1 つだけなので特定の CPU レジスタに入れて戻るやり方が普通です)。[2]

C++や Pascal など 一部の言語では引数機構として**参照渡し** (call by reference) を使うこともできます。参照渡しでは、実引数として変数を書いた場合、そのアドレスが渡され、関数側で仮引数に代入すると、対応する実引数の変数が変更されるようになります。関数から複数の値を返したい場合にはこのような機構が便利です。しかし C 言語には値渡ししかないので、ポインタの「値」を渡して間接参照で代入するわけです (それを自動的にやってくれるのが参照渡しだと癒えます)。

たとえば、2 つの実数変数の値を交換する関数 `dswap(double *x, double *y)` を書くことを考えます。変数の値を書き換えるには左辺値が必要ですから、ポインタを受け取りって間接参照で書き換えるわけです。

```
// dswaptest.c --- demonstration of dswap.
#include <stdio.h>
#include <stdlib.h>
void dswap(double *x, double *y) {
  double z = *x; *x = *y; *y = z;
}
int main(int argc, char *argv[]) {
  double a = atof(argv[1]), b = atof(argv[2]);
  printf("a = %g, b = %g\n", a, b);
  dswap(&a, &b);
  printf("a = %g, b = %g\n", a, b);
  return 0;
}
```

実行例は次の通り。

```
% gcc8 dswaptest.c
% ./a.out 8 3
a = 8, b = 3
a = 3, b = 8
```

[2]スタックを介して渡とメモリの出し入れが必要となり速度が出ないので、最近のシステムでは引数も多数ある CPU レジスタに入れて渡す呼び出し規約が使われます。その場合、それをメモリに格納する位置 (つまり局所変数としての場所) は戻り番地より上に用意します。

テストケースも示しましょう。実数は一般に計算誤差があるため、「等しい」で比べるのではなく「差 (の絶対値) がいくつ未満」でチェックするようにしています。ここでは許容誤差は「1.0×10^{-10}」としました。

```
// test_dswap.c --- unit test for dswap.
#include <math.h>
#include <stdio.h>
(dswap here)
void expect_double(double a, double b, double d, char *msg) {
  printf("%s %.10g:%.10g %s\n", (fabs(a-b)<d)?"OK":"NG" , a, b, msg);
}
int main(void) {
  double a = 3.14, b = 2.71; dswap(&a, &b);
  expect_double(a, 2.71, 1e-10, "a should be 2.71");
  expect_double(b, 3.14, 1e-10, "b should be 3.14");
}
```

実行例は次の通り。

```
% gcc8 test_dswap.c
% ./a.out
OK 2.71:2.71 a should be 2.71
OK 3.14:3.14 b should be 3.14
```

演習 1 ポインタと間接参照を使って、次のような関数を作れ。単体テストも作成すること。

a. `void rotate3(double *a, double *b, double *c)` — 3 つの実数変数のアドレスを受け取り、1 番目の値を 2 番目に、2 番目の値を 3 番目に、3 番目の値を 1 番目にと「順繰りに」転送する。

b. `void topolar(double x, double y, double *rad, double *theta)` — 2 次元直交座標の位置 (x, y) を受け取り、極座標形式に変換した結果をアドレスの渡された 2 つの実数変数に格納する。(ヒント: 「`man atan2`」はやってみた方がよい。)

c. `void normalize(double *x, double *y, double *norm)` — 2 次元ベクトル (x, y) を受け取り、正規化する (ノルム $x^2 + y^2$ を 1 にする)。元のベクトルのノルムを 3 番目のパラメタで渡された変数に返す。

d. `void divide(int a, int b, int *q, int *r)` — 整数 a を整数 b で割った時の商 q と余り r を返す。$a = b \times q + r$ であり、なおかつ a の符号 (正負) と q の符号 (正負) は一致すること (これは C 言語の除算、剰余とは違っている)。b が 0 のときは q も r も 0 とすること。

5.2 再帰呼び出しとその特性

5.2.1 再帰呼び出し

再帰呼び出し (recursive call) とは、ある関数が直接または間接に自分自身を呼び出すことをいいます。数学ではしばしば、再帰的な定義が使われますが、これをそのままコードに直すと再帰呼び出しになります。以下は数学の定義に読めますね (ただし x、y は正の整数とします)。

$$gcd(x,y) = \begin{cases} x & (x = y) \\ gcd(x-y,\ y) & (x > y) \\ gcd(x,\ y-x) & (x < y) \end{cases}$$

それをそのまま C 言語のプログラムにするわけです。

```
int gcd(int x, int y) {
  if(x == y) {
    return x;
  } else if(x > y) {
    return gcd(x-y, y);
  } else {
    return gcd(x, y-x);
  }
}
```

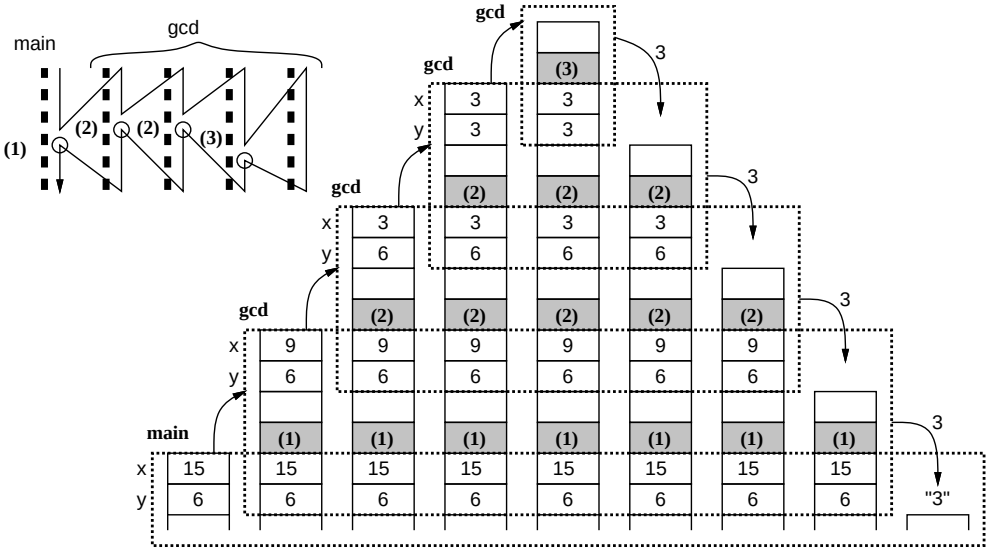


図 5.6: GCD の呼び出しと実行スタックのようす

それでは、再帰呼び出しはどのようにして実装されているのでしょう。実は、先に説明した実行時スタックを用いれば、そのままで再帰呼び出しは実装できます。例として、先の gcd を main から「gcd(15, 6)」のように呼び出した場合の実行時スタックの変化を図 5.6 に示します。

gcd のコードは実際には 1 つだけですが、ここでは分かりやすいように必要な個数コピーして次々に呼び出しているように描いてあります。gcd はローカル変数を持たず、パラメタ x と y は戻り番地より下に積まれて渡されてくることに注意。そして、戻り番地としては (1)main に戻るところ、(2)x が大きかった場合の再帰呼び出しから戻る、(3)y が大きかった場合の再帰呼び出しから戻る、の 3 通りがあることにも注意してください。

5.2.2　再帰の考え方

再帰呼び出しを数学の再帰的定義から考えることもできますが、もっと直接的な考え方を紹介しましょう。それは、「自分がやるべき仕事を少し簡単化して、自分の分身に頼む」というものです。そうやって簡単化していくと、最後は「頼まなくてもすぐできる」ようになるので、それは直接やります。

例題として**ハノイの塔** (tower of Hanoi) を取り上げましょう。これは図 5.7 のように棒の立った台座が 3 つ (A、B、C) と、何枚かの大きさの異なる円盤 (棒が入る穴つき) から成るパズルです (図では 3 枚)。円盤は小さいものから順に 1〜N の番号がついていて、最初は左上のように、A の台座に下から大きい順に積んであります。

それでパズルですが、円盤をいちどに 1 枚だけずつ他の台座に動かすことを繰り返して、最終的に右上のように B の台座にそっくり移してください。ただし、途中のどの段階でも「小さい円盤の上に

より大きい円盤を載せることはできません」。この 3 枚の場合であれば、実線の矢印のように順に動かして行けばできます。

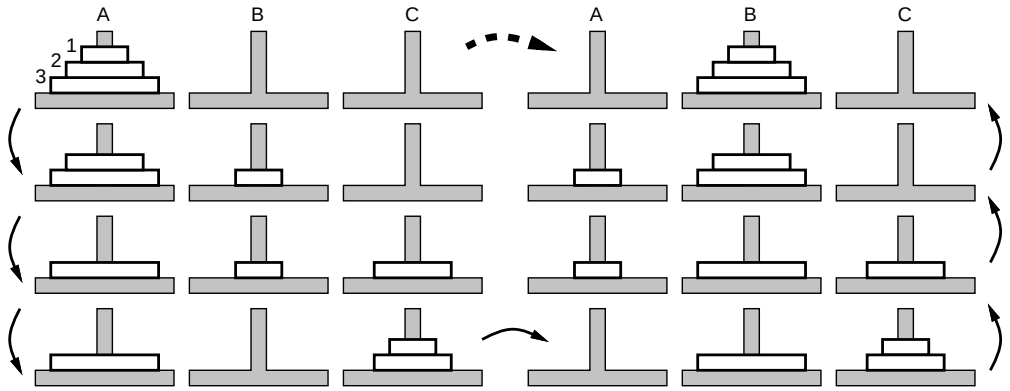


図 5.7: ハノイの塔

この動かし方を出力するプログラムを動かした様子を見ましょう。コマンド引数で円盤の枚数を与えます。

```
% ./a.out 3
move disc 1 from A to B.
move disc 2 from A to C.
move disc 1 from B to C.
move disc 3 from A to B.
move disc 1 from C to A.
move disc 2 from C to B.
move disc 1 from A to B.
```

これはどのように再帰を使って表せるでしょうか。次のように考えてください。

- C を作業場所として使いつつ A から B に K 枚の円盤を移すには、
- B を作業場所として使いつつまず A から C に K-1 枚の円盤を移し (※)、
- K の円盤を A から B に移し、
- B を作業場所として使いつつ C から A に K-1 枚の円盤を移せば (※) よい。

ここで (※) の 2 箇所は自分の分身に丸投げしていますが、問題が少し簡単になっているので (円盤の枚数が 1 枚だけ少ない)、これで大丈夫です。ポイントは、K の円盤が一番大きいのですから、それ以外の円盤はその上に載せて大丈夫なので、(※) の中で K の載っている A や B を作業場所に使ってよいというところです。あと、このままだと再帰が堂々めぐりですが、K が 1 になったらその上には円盤は載っていないので、直接行き先に移せます。

では、これをプログラムにしたものを見てみましょう。先の考え方をそのままコードにしていることが分かります (実行例は上に示しました)。

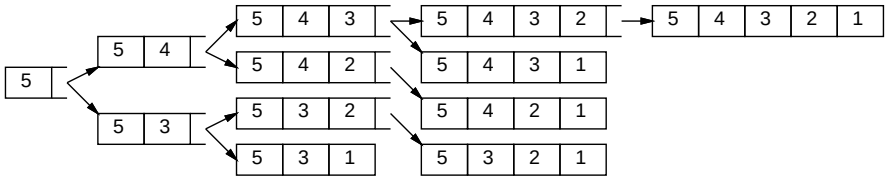
```
// hanoi.c --- tower of hanoi
#include <stdio.h>
#include <stdlib.h>

void hanoi(int k, int x, int y, int z) {
  if(k == 1) {
```

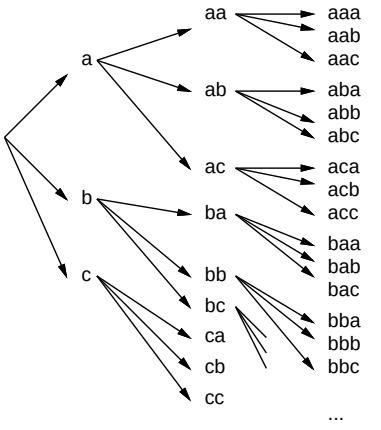
```
    printf("move disc %d from %c to %c.\n", k, x, y);
  } else {
    hanoi(k-1, x, z, y);
    printf("move disc %d from %c to %c.\n", k, x, y);
    hanoi(k-1, z, y, x);
  }
}
int main(int argc, char *argv[]) {
  hanoi(atoi(argv[1]), 'A', 'B', 'C'); return 0;
}
```

演習 2　「正の整数 n から始まる 1-2 減少列」とは、「最初が n、最後が 1 の減少数列で、隣り合う要素の差が 1 または 2 であるようなもの」である。たとえば 5 から始まる 1-2 減少列は「5, 4, 3, 2, 1」「5, 4, 3, 1」「5, 4, 2, 1」「5, 3, 2, 1」「5, 3, 1」の 5 つある。n を与えて n から始まる 1-2 減少列をすべて表示するプログラムを作れ。

ヒント: 配列 a に n から始まる 2-1 減少列を作るには、まず a の先頭に n を入れる。次に、a+1(a の次の要素から始まる配列) に n-1 から始まる 1-2 減少列と n-2 から始まる 1-2 減少列を作ってそれぞれ打ち出せばよい。再帰の終わりは n が 1 だったとき打ち出して戻ればよいが (0 なら行きすぎなので何もしない)、そのとき「全体を打ち出す」ためには、一番最初の配列の先頭が分かる必要がある (グローバル変数にするか、これもパラメタで受け渡す)。そして、先頭から現在の位置まで、ないし 1 を入れた位置まで打ち出す。



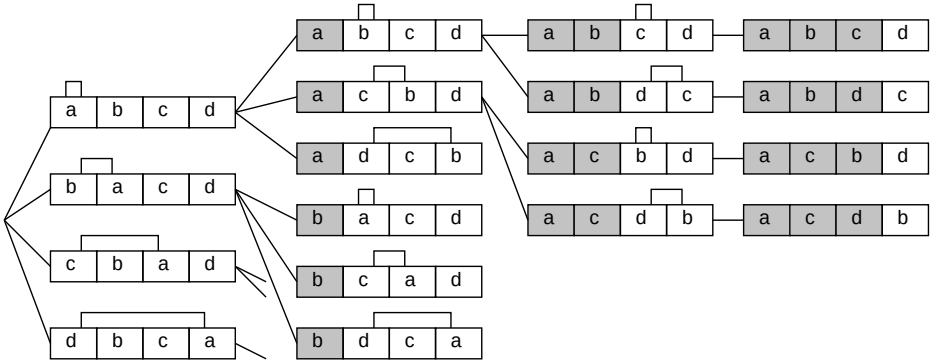
演習 3　文字の配列とその要素数、および生成する文字列の長さを与えて、指定された文字のあらゆる組み合わせで指定された長さの文字列を表示するプログラムを作れ。たとえば「abc」で長さ 3 なら、$3^3 = 27$ 通り、長さ 4 なら $3^4 = 81$ 通りの文字列を打ち出すことになる。



演習 4　文字の配列とその要素数を与えて、現れる文字の全ての順列を打ち出すプログラムを作れ。たとえば「abc」であれば「abc, acb, bac, bca, cab, cba」の 6 つが打ち出される。

ヒント: たとえば「abcd」であれば、まず最初が a の場合、b の場合、c の場合、d の場合を配列に用意する。それには、1 番目と 1、2、3、4 番目をそれぞれ交換すればできる。そのあと、配列の次の位置と、長さとして 1 減らした値を渡して自分自身を呼び出せば、それぞれの場合

についてすべての順列を生成できる。この場合も最後に長さ 1 になったところで打ち出すためには、元の配列の先頭が必要となる。あと、再帰から戻ったところで交換した値を再度交換して元に戻さないとごちゃごちゃになる。必ず「元の状態に戻してから終わる」必要があるのに注意。



演習 5 順列生成プログラムを利用して自分の名前のアナグラムを生成せよ。ただし、ローマ字として読めるものだけを打ち出すこと。

ヒント: 順列プログラムは変更せず、最後に打ち出すところで「文字列がローマ字として正しいか」チェックするのがよい。[3]

演習 6 再帰を活用した自分の好きなプログラムを作れ。

5.2.3 再帰を使った深さ優先探索

再帰呼び出しはスタックによって実装されているので、スタックと同じ操作を行うことができます。具体的には「積む」ところで自分自身を呼び、「降ろして処理する」ところで普通に処理をして戻ればよいのです (処理した結果「積む」場合は再帰呼び出しします)。具体例として、前回やった図 5.8 の探索を再帰でやってみましょう。

`railmap.h` は今回は構造体のフィールドを変更しています。`prev` は、この節まで来たときの「1 つ前」の節番号を入れておき、あとで経路をたどれるようにします。`visit` は「この節を既に通ったか否か」を示すのに使います。[4]

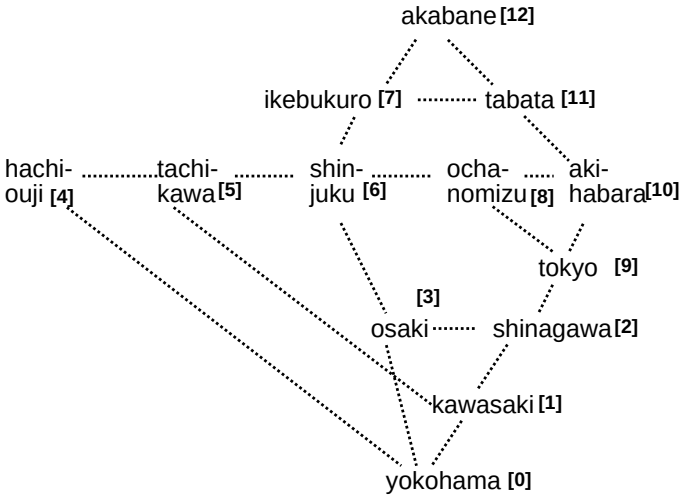


図 5.8: とある鉄道路線図 (再掲)

[3] このように、候補を生成してみて OK かどうかチェックして使用するようなプログラムのことを generate-test 型のプログラムと呼びます。

[4] 前回はフィールド `dist` に距離と通ったか否かの役割を兼ねさせていましたが、今回は距離は節に格納しません。

```
// railmap2.h -- a railroad map (fields added)
#include <stdbool.h>
struct node { char *name; int num, edge[5], prev; bool visit; };
struct node map[] = {
  { "yokohama", 3, { 1, 3, 4 }, -1, false },    // 0
  { "kawasaki", 3, { 0, 2, 5 }, -1, false },    // 1
  { "shinagawa", 3, { 1, 3, 9 }, -1, false },   // 2
  { "osaki", 3, { 0, 2, 6 }, -1, false },       // 3
  { "hachiouji", 2, { 0, 5 }, -1, false },      // 4
  { "tachikawa", 3, { 1, 4, 6 }, -1, false },   // 5
  { "shinjuku", 4, { 3, 5, 8, 7 }, -1, false }, // 6
  { "ikebukuro", 3, { 6, 11, 12 }, -1, false }, // 7
  { "ochanomizu", 3, { 6, 9, 10 }, -1, false }, // 8
  { "tokyo", 3, { 2, 8, 10 } , -1, false },     // 9
  { "akihabara", 3, { 8, 9, 11 }, -1, false },  // 10
  { "tabata", 3, { 7, 10, 12 }, -1, false },    // 11
  { "akabane", 2, { 7, 11 }, -1, false },       // 12
};
```

では、コードの方を示しましょう。重要なポイントですが、1 つ辺を進むごとに「どこから来たか」が分かるように prev に現在の節の番号を入れてから再帰呼び出しをしますが、戻って来たら prev の値を元に戻します。そのため、入れる前に変数 p に元の値を保管してあります。このように、再帰を使いながらデータ構造を書き換えるときには、「終わったら元に戻す」のが原則です。

```
// searchrec.c --- search path with recursion
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "railmap2.h"

void search(int cur, int goal, int len) {
  if(map[cur].visit) { return; }
  if(cur == goal) { printf("len = %d\n", len); return; }
  map[cur].visit = true;
  for(int i = 0; i < map[cur].num; ++i) {
    int k = map[cur].edge[i], p = map[k].prev;
    map[k].prev = cur; search(k, goal, len+1); map[k].prev = p;
  }
  map[cur].visit = false;
}
int main(int argc, char *argv[]) {
  int from = atoi(argv[1]), to = atoi(argv[2]);
  search(from, to, 0); return 0;
}
```

では、試しに品川から新宿まで何ステップで行けるかを調べます (あらゆる経路を調べることに注意)。

```
% ./a.out 2 6
```

```
len = 4
len = 5
len = 6
len = 3
len = 5
len = 5
len = 2
len = 3
len = 6
len = 7
len = 4
len = 5
len = 6
```

距離しか表示されないので分かりにくいですが、番号の若い節から順に試すので図と照合すればどこを通っているか分かります。最初の「4」は「品川→川崎→横浜→大崎→新宿」で、次の「5」は「品川→川崎→立川→横浜→八王子→立川→新宿」のようですね。

演習 7 上のプログラムをそのまま動かして動作を確認し、OK なら次のことをやってみなさい。

a. 距離 (通った辺の数) だけが表示されるのでは分かりにくいので、実際に通っている経路を表示するように改良する。(ヒント: `prev` にどこを通って現在の節まで来たかが入っているので、これを順に出発点までたどって行けばよい。)

b. 全部の経路を表示するのでなく、最短の (または最長の) 経路 1 つだけを表示するように変更する。(ヒント: 1 つ見つかるごとに打ち出す代わりに、経路を配列などに保管しておき、最短や最長が更新されたら新しいものに入れ換えるようにして、最後に打ち出す。)

c. 現在の「距離」は単に通った辺の数だが、その代わりに実際の距離 (または所用時間) を用いて最短、最長などを調べられるように変更する。(ヒント: 各辺ごとに距離や所用時間のデータを保持するようにデータ構造を変更する。)

d. その他、自分が面白いと思う改良をおこなう。

本日の課題 5A

「演習 1」〜「演習 4」で動かしたプログラム 1 つを含むレポートを本日中 (授業日の 23:59 まで) に提出してください。

1. 学籍番号、氏名、ペアの学籍番号 (または「個人作業」)、提出日時。名前の行は先頭に「@@@」を付けることを勧める。
2. プログラムどれか 1 つのソースと「簡単な」説明。
3. レビュー課題。提出プログラムに対する他人 (ペア以外) からの簡単な (ただしプログラムの内容に関する) コメント。
4. 以下のアンケートの回答。

   Q1. 引数の渡し方と手続き呼び出しの実現方法について理解しましたか。

   Q2. 再帰呼び出しのプログラムがどのように動いているか理解しましたか。

   Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

次回までの課題 5B

「演習 1」〜「演習 7」(ただし 5A で提出したものは除外、以後も同様) の (小) 課題から選択して 2 つ以上プログラムを作り、レポートを提出しなさい。できるだけ複数の演習から選ぶこと。レポートは次回授業前日 23:59 を期限とします。

1. 学籍番号、氏名、ペアの学籍番号 (または「個人作業」)、提出日時。名前の行は先頭に「@@@」を付けることを勧める。
2. 1 つ目の課題の再掲 (どの課題をやったか分かればよい)、プログラムのソースと「丁寧な」説明、および考察 (課題をやってみて分かったこと、分析、疑問点など)。
3. 2 つ目の課題についても同様。
4. 以下のアンケートの回答。

   Q1. 再帰呼び出しのプログラムが書けるようになりましたか。
   Q2. 再帰呼び出しでグラフの探索ができることを納得しましたか。
   Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

＃6　分割統治と再帰の除去

今回は次のことが目標となります。

- 分割統治アルゴリズムの考え方について知る。
- 再帰呼び出しを持つプログラムから再帰を除去する手法について知る。

6.1　分割統治アルゴリズム

6.1.1　分割統治アルゴリズムとは

前回から再帰を用いたコードを扱っていますが、再帰アルゴリズムの作り方には多様なものがあります。ここでは**分割統治** (divide and conquer) と呼ばれる手法について取り上げます。[1] 分割統治アルゴリズムの考え方は次のようなものです。

- 与えられた問題に直接取り組む代わりに、問題を 2 つとかさらに多くの「部分問題」に分割する。
- それぞれの部分問題は自分自身を再帰呼び出しして解く。
- その解いた結果を組み合わせることで元の問題の解を得る。

たとえばハノイの塔の問題を振り返ってみましょう (図 6.1)。元の問題は K 段の塔を (1 つずつ円盤を動かしながら)A から B にそっくり移動するというものですが、結構複雑です。

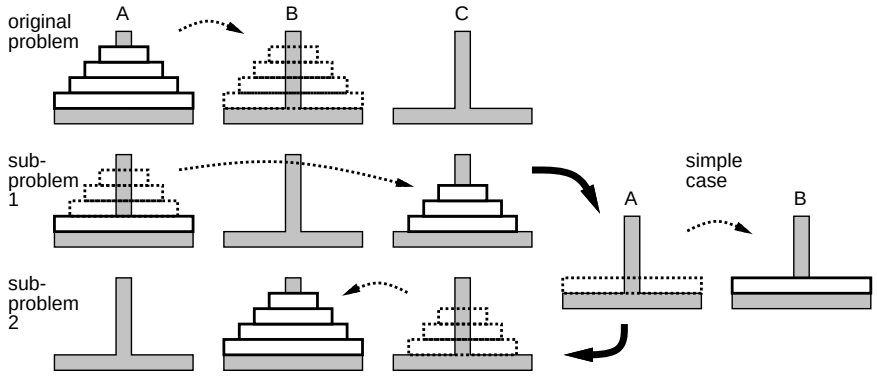


図 6.1: 分割統治手法としてのハノイの塔

そこで自分は円盤 K だけ担当することにして、1〜K-1 をまず C に移し (※)、その後上に何もなくなった K を A から B に移し、最後に C に移してあった 1〜K を B に移す (※) ことにするわけです。ここで※印の 2 つが分割された部分問題にあたります。そして全体の解を組み立てるとき、その 2 つと間の「K を A から B に移す」という簡単な作業を合わせればよいわけです。「部分問題に分割」の意味がお分かりになったでしょうか。

[1] 分割統治という言葉は、ローマ帝国が各地を統治するときに「統治される側をばらばらに分断したりその上で互いに反目させるなどしてうまく御した」という故事から来ていますが、ここではその意味からこの言葉を援用しています。

6.1.2 再帰による空間分割

分割のしかたは問題により様々です。次の例として、長方形 (や正方形) の領域内で曲線や直線で囲まれた図形の面積を求める問題を取り上げましょう。たとえば、2x2 の領域に図 6.2 のように四分の一円が配置されていたとして、その面積を求めれば π になるはずですね。

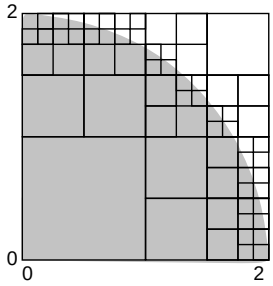


図 6.2: 空間分割により面積を求める

ただし、その問題は簡単ではありません。そこで、自分の領域を 4 分割してそれぞれについて自分自身を再帰呼び出しして、それぞれの領域で図形に囲まれた部分の面積を求めます。これが部分問題です。部分問題から自分の問題の解を作り出すのは簡単で、面積ですから単に足せばよいわけです。

しかしそれでは、どんどん領域が細かくなるだけで止まらなくなりますね? そこで (1) 領域の 4 隅がともに図形に含まれている/いないなら、領域全体の面積ないし 0 を返します。あるいはそうでない場合でも、(2) 分割が十分細かければ…たとえば領域の幅が δ 未満であれば、適当に近似した値を返します。これが「簡単な場合」ですね。このように、2 次元や 3 次元の領域を必要に応じて荒く/細かく分割して処理してゆく手法を**空間分割**と呼びます。

ではプログラムを見てみましょう。最初の関数 `inside_ciecle` は、渡された XY 座標が円の内部にあれば 1、無ければ 0 を返すものとします (それなら `bool` でもよさそうですが、結果を足し算したいので `int` としています)。そして面積を計算する関数 `area` ですが、パラメタとして X の最小と最大、Y の最小と最大を受け取ります。この長方形 (または正方形) の領域内で、さらに渡された図形の中にある面積を求めるわけです。

ですが、最後の「`int (*f)(double,double)`」とは? これは関数ポインタ型で、「実数を 2 つ受け取り整数を返す」関数へのポインタを受け取ります。

```
// area.c -- calcurate area of 2-D object
#include <stdio.h>
#define DELTA 0.001

int inside_circle(double x, double y) {
  return x*x+y*y <= 4.0;
}
double area(double x1, double x2, double y1, double y2,
            int (*f)(double,double)) {
 int b1 = (*f)(x1,y1), b2 = (*f)(x1,y2);
 int b3 = (*f)(x2,y1), b4 = (*f)(x2,y2), bn = b1+b2+b3+b4;
 if(x2-x1 < DELTA || bn == 0 || bn == 4) {
   return bn*0.25*(x2-x1)*(y2-y1);
 } else {
   double x3 = 0.5*(x1+x2), y3 = 0.5*(y1+y2);
   return area(x1, x3, y1, y3, f) + area(x1, x3, y3, y2, f) +
          area(x3, x2, y1, y3, f) + area(x3, x2, y3, y2, f);
```

```
    }
  }
  int main(void) {
    printf("%8.6f\n", area(0, 2, 0, 2, inside_circle));
    return 0;
  }
```

そして area に入ってすぐのところで、領域の 4 隅について渡された関数を呼び出し、図形の内部か否かの 1/0 を求めます。その合計が 4 または 0 なら「完全に入っている/いない」単純な場合です。または、X 方向の幅が δ 未満のときも、これ以上細かく分けずに単純な場合として処理します。具体的に図形内部に入っている頂点の数が bn だとしたとき、領域の面積を s として $s \times \frac{bn}{4}$ で面積を近似します (全て入っている/いない場合もこれで対応できます)。一方、単純でない場合は X および Y の範囲の中間を求めて領域を 4 分割し、それぞれの領域内での面積を再帰で求めてから合計します。

main は簡単で、領域の範囲と inside_ciecle を渡して area を呼び出し、結果を表示するだけです。実行のようすを見ましょう。小数点以下 4 桁まで正しく求まっているようです。

```
% ./a.out
3.141577
```

演習 1 面積の計算を実際に動かして確認しなさい。OK なら次のことをやってみなさい。

a. もっと別の図形で面積を計算してみる。
b. DLETA の値を変化させることで、どれくらい精度が変わるか、試して検討する。できれば複数の図形の面積で試してみるとなおよい。
c. 実際にいくつの長方形 (または正方形) を計算しているのか、それは DELTA の値を変化させるとどのように変わるのかを試してみて検討する。
d. 上の例は 2 次元だったが、3 次元でも同じように考えて体積を空間分割で求めることができる。作成してみよ (たとえば八分の一球の体積を求めてみるなど)。
e. 空間分割を使った面白いプログラムを何か作ってみよ。

6.2 再帰の除去

6.2.1 再帰の除去とその必要性

再帰はさまざまな計算をコンパクトに書けるようにする有力な手法ですが、一方で再帰の数が多くなると実行時スタック領域を多く消費するという弱点も持っています。また、言語や環境によっては再帰呼び出しが使えなかったり使いづらいこともあります。

一般に、再帰を使って書かれたプログラムはすべて、再帰を使わないように書き換えることが可能です。これを**再帰の除去** (resursion elimination) と呼びます。除去のやりやすさは、プログラムの形によって違っています。以下で主要な手法について見て行きます。

6.2.2 末尾再帰の除去

末尾再帰 (tail recursion) とは、再帰呼び出しの形が「return 再帰呼び出し;」となっているもの、すなわち再帰呼び出しで返された値がそのまま自分の返値になるようなものを言います。たとえば前に取り上げた GCD の関数を見てみましょう。

```
int gcd(int x, int y) {
  if(x == y) {
    return x;
  } else if(x > y) {
```

```
    return gcd(x-y, y);
  } else {
    return gcd(x, y-x);
  }
}
```

見て分かるように、2 箇所ある再帰呼び出しはいずれも上記の形すなわち末尾再帰になっています。なぜ末尾再帰を問題にしているかというと、末尾再帰では再帰を呼ぶところでもう呼び側の関数のフレームは不要になっているからです (戻って来たら直ちに return するのでそれ以上変数やパラメタなどを使う余地がない)。ということは、再帰呼び出しをして新しいフレームを作ったりしなくても、単にパラメタを渡そうとしている実引数の値で書き換えて先頭に戻るだけでよいのです。実際に上のコードをそのように変更してみましょう。

```
int gcd2(int x, int y) {
  while(true) {
    if(x == y) {
      return x;
    } else if(x > y) {
      x = x - y;
    } else {
      y = y - x;
    }
  }
}
```

「先頭に戻る」ことを実現するため、本体全体を無限ループで囲みました。そして、再帰呼び出しの箇所ではパラメタ x や y に渡そうとしている式の値を代入しています (このコードでは片方は元の値のままなので何もしなくて済みます)。

このように末尾再帰をループに置き換えることは、呼び/戻りの手間も余分なスタックフレームの消費も削減でき、利点だらけです。そのため、言語処理系によっては、このような変形を自動的にやってくれます (残念ながら C コンパイラではほとんどないですが)。

演習 2　上の再帰除去版の GCD を動かして動作を確認しなさい。OK なら次の末尾再帰のみから成る関数でも同様に再帰除去してみなさい。元の版と両方動かして検討すること。

a. 非負整数を受け取り偶数か否かを返す関数 iseven。

```
bool iseven(int n) {
  if(n == 0) {
    return true;
  } else if(n == 1) {
    return false;
  } else {
    return iseven(n - 2);
  }
}
```

b. 2 つの非負整数を受け取りその和を返す関数 sum。

```
int sum(int a, int b) {
```

```
  if(a == 0) {
    return b;
  } else {
    return sum(a-1, b+1);
  }
}
```

c. 文字列を「abc」「bc」「c」「」のように 1 文字ずつ削りながら打ち出す関数 strtriangle。この例では打ち出すのが目的なので返値がなく return がないが、関数の末尾に来たらそこで return することになるのでこれまでと同様に考えられる。

```
void strtriangle(char *s) {
  printf("%s\n", s);
  if(*s == '\0') {
    // do nothing
  } else {
    strtriangle(s+1);
  }
}
```

d. その他自分の好きな末尾再帰のみから成る関数。

6.2.3 末尾再帰への変形

末尾再帰の除去は分かりましたが、現実には末尾再帰でない再帰呼び出しも多くあります。それらのうちで、自分自身を 1 回しか呼ばない再帰は末尾再帰に変形できます。たとえば階乗を見てみます。

```
int fact(int n) {
  if(n < 1) {
    return 1;
  } else {
    return n * fact(n-1);
  }
}
```

これは再帰呼び出しから返された値にさらに n を掛けて自分の値とするので、「そのままの値を返す」末尾再帰にはなっていません。しかしこれを、次のように変形したらどうでしょうか。

```
int fact1(int n, int r) {
  if(n < 1) {
    return r;
  } else {
    return fact1(n-1, r*n);
  }
}
int fact(int n) { return fact1(n, 1); }
```

再帰関数そのものは 1 つパラメタを追加し、呼び出し方が変わるのでそれを呼び出すための fact を別に用意しました。追加したパラメタ r は「最終結果を累積していくための」パラメタで、**累積引数** (accumulation parameter) と呼びます。

たとえば 5 の階乗であれば fact(5) → fact1(5, 1) → fact1(4, 5) → fact1(3, 20) → fact1(2, 60) → fact1(1, 120) → fact1(0, 120) → 120、のように累積引数を使って結果が計算されていきます。一般

に累積引数は、適切な初期値 (変換前の関数で単純なケースの値) を与えて呼び出し、そこに加算や乗算を行って最終結果ができあがり、最後に単純なケースでその値を返します。これで末尾再帰のみに変形できたので、あとは前と同じようにして再帰を除去できます。

演習 3 階乗の例題をひととおり動かして確認しなさい。OK なら、以下の末尾再帰でない 1 次元の (=自分を 1 回しか呼ばない) 再帰を累積引数を使って末尾再帰に変形し、さらに再帰を除去してみなさい。元の関数と変換した関数の両方を実行し確認すること。

a. 2 つの正の整数を受け取りその積を返す関数 `mul`。

```
int mul(int a, int b) {
  if(b == 0) {
    return 0;
  } else {
    return a + mul(a, b-1);
  }
}
```

b. 実数 x と非負整数 n を受け取り x^b を返す関数 `powx`。

```
int powx(double x, int n) {
  if(n < 1) {
    return 1.0;
  } else {
    return x * powx(x, n-1);
  }
}
```

c. 実数 x と非負整数 n を受け取り x^b を返す関数 `powx`(高速版)。

```
double powx(double x, int n) {
  if(n < 1) {
    return 1.0;
  } else if(n % 2 == 1) {
    return x * powx(x, n-1);
  } else {
    double y = powx(x, n / 2); return y*y;
  }
```

d. 実数 x と非負整数 n を受け取り $\sum_{i=0}^{n} \frac{1}{x+i}$ を返す関数 `calc`。

```
int calc(double x, int n) {
  if(n < 0) {
    return 0.0;
  } else {
    return 1/(x+i) + calc(x, n-1);
  }
}
```

e. その他自分の好きな 1 次元再帰の関数。

6.2.4 スタックを使ったコードへの書き換え

末尾再帰 (や末尾再帰への変形) により再帰が除去できるのは、1 次元の (1 つの呼び出しにつき自分を 1 回しか呼ばない) 再帰に限られます。そうでない場合はどうしたら良いのでしょうか。もともと言語処理系は再帰を実行時スタックを用いて実現しています。ですから、言語処理系の代わりに自分で「同じように」スタックを操作することで、再帰と同じ動作が「必ず」実現できます。とはいえ、あまり分かりやすくはないことが多いですが…

再びハノイの塔を取り上げましょう。再帰版のコードを再掲します。

```
void hanoi(int k, int x, int y, int z) { // (1)
  if(k == 1) {
    printf("move disc %d from %c to %c.\n", k, x, y);
  } else {
    hanoi(k-1, x, z, y); // (2)
    printf("move disc %d from %c to %c.\n", k, x, y);
    hanoi(k-1, z, y, x);
  }
}
```

(1)、(2) とコメントがついていますが、これは「どこから実行するか」が 2 通りあることに対応しています。そして、スタックには引数である k, x, y, z のほかにその「どこから実行」を区別する値 cont も積みます (ローカル変数があればそれも積むのですが、この例題ではローカル変数はありません)。上記の 5 つのフィールドを持つ構造体を要素とするスタックを使うのが自然ですが、ここでは既に作った istack を再利用することにして、5 つの値を逆順に積む下請け関数 p5 というのを作りました。[2] ではスタック版のコードを見てみましょう。

```
// hanoistack.c --- hanoi with stack
#include <stdio.h>
#include <stdlib.h>
#include "istack.h"

void p5(istackp s, int a, int b, int c, int d, int e) {
  istack_push(s,e); istack_push(s,d);
  istack_push(s,c); istack_push(s,b); istack_push(s,a);
}
void hanoiloop(int k, int x, int y, int z) {
  istackp s = istack_new(100); p5(s, 1, k, x, y, z);
  while(!istack_isempty(s)) {
    int cont = istack_pop(s); k = istack_pop(s);
    x = istack_pop(s); y = istack_pop(s); z = istack_pop(s);
//  printf("%d %d %c %c %c\n", cont, k, x, y, z);
    if(cont == 1) {
      if(k == 1) {
        printf("move disk %d from %c to %c.\n", k, x, y);
      } else {
        p5(s, 2, k, x, y, z); p5(s, 1, k-1, x, z, y);
      }
```

[2] なぜ逆順かというと、スタックでは最後に積んだものが最初に出て来るため、降ろす時に自然な順番で取り出したいからです。

```
      } else { // cont == 2
        printf("move disk %d from %c to %c.\n", k, x, y);
        p5(s, 1, k-1, z, y, x);
      }
    }
  }
int main(int argc, char *argv[]) {
    hanoiloop(atoi(argv[1]), 'A', 'B', 'C'); return 0;
  }
```

hanoiloop がその変換した関数ですが、最初にスタックを用意し、実行箇所 1 と 4 つの引数を積みます。その後はスタックが空になるまで繰り返しです。

繰り返しの先頭で 5 つの値を取り出し変数に入れます (実行位置 cont 以外はパラメタの変数を再利用しています)。それから、実行位置によって 2 つに枝分かれします。まず (1) の方ですが、k が 1 のときはこれまで通り出力します。そうでないときが再帰呼び出しですが、再帰版を見るとまず hanoi(k-1, x, z, y) を読んで、戻って来たら出力ですね。変換後も同じに動作するためには、スタックの下にまず「戻って来たら (2) から実行」を積んで、それから「先頭から k-1, x, z, y のパラメタで実行」を積みます (後から積んだ方が先に出て来るので)。そして仕事はこれで終わりです。残りは (2) からの続きの実行ですが、それは出力したあと今度は「先頭から k-1, z, y, x」で実行ですね。

main はコマンド引数から円盤の数を取り出して hanoiloop を呼ぶだけです。実行のようすを示します (printf につけてあるコメントを外して毎回スタックから出て来る値を表示するようにしています)。図 6.3 にスタックの変化の様子を示しました。

```
% gcc8 hanoiloop.c istack.c
% ./a.out 3
1 3 A B C
1 2 A C B
1 1 A B C
move disk 1 from A to B.
2 2 A C B
move disk 2 from A to C.
1 1 B C A
move disk 1 from B to C.
2 3 A B C
move disk 3 from A to B.
1 2 C B A
1 1 C A B
move disk 1 from C to A.
2 2 C B A
move disk 2 from C to B.
1 1 A B C
move disk 1 from A to B.
```

6.2.5 返値がある場合のスタック変換

ハノイの塔では関数に返値がありませんでしたが、返値がある場合はどうしたらいいでしょうか。ここでは分かりやすさのため、返値を積むスタックを別に用意する方法で説明します。例題は組み合わせの数の再帰版です。

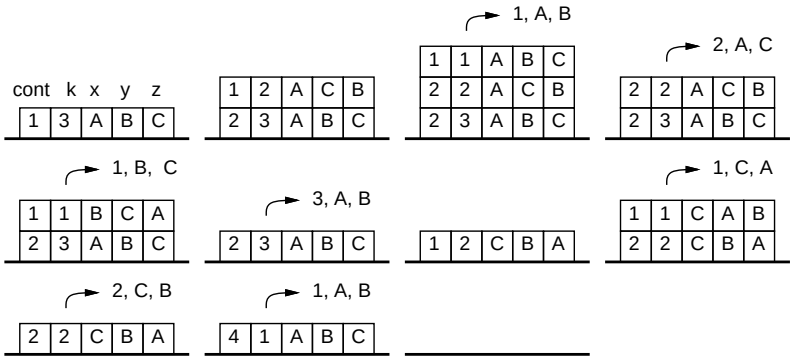


図 6.3: ハノイの塔のスタック版でのスタックの動作

```
int combr(int n, int r) { // (1)
  if(r == 0 || r == n) {
    return 1;
  } else {
    return combr(n-1, r) + combr(n-1, r-1); // (2)
  }
}
```

上述のように返値のスタック v を追加し、return ではそこに値を積むようにします。変数を積むスタックは実行位置も含め 3 つずつ値を積めばよいです。

```
// combstack.c --- combination with stack
#include <stdio.h>
#include <stdlib.h>
#include "istack.h"

void p3(istackp s, int a, int b, int c) {
  istack_push(s,c); istack_push(s,b); istack_push(s,a);
}
int combloop(int n, int r) {
  int x, ret;
  istackp v = istack_new(100);
  istackp s = istack_new(100); p3(s, 1, n, r);
  while(!istack_isempty(s)) {
    int cont = istack_pop(s);
    n = istack_pop(s); r = istack_pop(s);
//  printf("%d %d %d\n", cont, n, r);
    if(cont == 1) {
      if(r == n || r == 0) {
        istack_push(v, 1);
      } else {
        p3(s, 2, 0, 0); p3(s, 1, n-1, r-1); p3(s, 1, n-1, r);
      }
    } else { // cont == 2
      istack_push(v, istack_pop(v)+istack_pop(v));
    }
```

```
    }
    return istack_pop(v);
  }
int main(int argc, char *argv[]) {
    printf("%d\n", combloop(atoi(argv[1]), atoi(argv[2]))); return 0;
  }
```

まず先頭 (1) から実行する場合ですが、単純なケースでは return だけですからスタック v に 1 を積むだけです。そうでない場合は、2 つの再帰を実行したあとでそれらの結果を足し算しますから、(2) からの実行を先に、2 つの再帰呼び出しぶんを後に積みます。再帰から戻った (2) ですが、スタック v から 2 つ値を取り出して足し、それを再度 v に積みます。そしてループが終わった時には結果がスタック v に載っているので、それを返します。

演習 4 hanoiloop と combloop を実行し、動作を確認しなさい。また両方についてそれぞれ、複数の (資料に載っているものとは別の) 実行例のスタックの変化の様子をまず自分で書き出し、次に実行してみて合っているか確認しなさい (printf につけているコメントを外して実行する)。

演習 5 次のような再帰を使った (返値を持たない) プログラムの再帰をスタックを用いて削除せよ。なお、再帰呼び出しごとに変化しないパラメタはスタックに載せないでよいことに注意。

a. 12 減少列のプログラム。参考までに再帰版をつける。

```
// decr12.c --- usage: ./a.out INTEGER
#include <stdio.h>
#include <stdlib.h>

void decr12(int n, int k, int *a) {
  if(n < 1) {
    // do nothing
  } else if(n == 1) {
    a[k] = 1;
    int i;
    for(i = 0; i <= k; ++i) { printf(" %d", a[i]); }
    printf("\n");
  } else {
    a[k] = n; decr12(n-1, k+1, a); decr12(n-2, k+1, a);
  }
}
int main(int argc, char *argv[]) {
  int buf[100]; decr12(atoi(argv[1]), 0, buf); return 0;
}
```

b. すべての文字の組み合わせ。参考までに再帰版をつける。

```
// allstr.c --- usage: ./a.out STRING LENGTH
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void allstr(int n, char *s, int len, int k, char *a) {
```

```
  if(len == 0) {
    a[k] = '\0'; printf("%s\n", a);
  } else {
    int i;
    for(i = 0; i < n; ++i) {
      a[k] = s[i]; allstr(n, s, len-1, k+1, a);
    }
  }
}
int main(int argc, char *argv[]) {
  char str[100];
  allstr(strlen(argv[1]), argv[1], atoi(argv[2]), 0, str);
  return 0;
}
```

c. 文字列のすべての順列。参考までに再帰版をつける。

```
// perm.c --- usage: ./a.out STRING
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void cswap(char *a, int i, int j) {
  char c = a[i]; a[i] = a[j]; a[j] = c;
}
void perm(int len, int k, char *a) {
  if(k == len) {
    printf("%s\n", a);
  } else {
    int i;
    for(i = k; i < len; ++i) {
      cswap(a, k, i); perm(len, k+1, a); cswap(a, k, i);
    }
  }
}
int main(int argc, char *argv[]) {
  char str[100];
  strcpy(str, argv[1]); perm(strlen(str), 0, str); return 0;
}
```

d. その他自分の好きな再帰プログラム。

演習 6 自分の好きな返値を持つ再帰プログラムのスタックを用いた再帰削除版を作成しなさい。

本日の課題 6A

「演習 1」〜「演習 6」で動かしたプログラム 1 つを含むレポートを本日中 (授業日の 23:59 まで) に提出してください。

1. sol または CED 環境で「`/home3/staff/ka002689/prog19upload 6a ファイル名`」で以下の内容を提出。

2. 学籍番号、氏名、ペアの学籍番号 (または「個人作業」)、提出日時。名前の行は先頭に「@@@」を付けることを勧める。
3. プログラムどれか 1 つのソースと「簡単な」説明。
4. レビュー課題。提出プログラムに対する他人 (ペア以外) からの簡単な (ただしプログラムの内容に関する) コメント。
5. 以下のアンケートの回答。

   Q1. 分割統治という概念について納得しましたか。
   Q2. 末尾再帰とはどういうものか理解しましたか。
   Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

次回までの課題 6B

「演習 1」〜「演習 6」(ただし 6A で提出したものは除外、以後も同様) の (小) 課題から選択して 2 つ以上プログラムを作り、レポートを提出しなさい。できるだけ複数の演習から選ぶこと。レポートは次回授業前日 23:69 を期限とします。

1. sol または CED 環境で「`/home3/staff/ka002689/prog19upload 6b ファイル名`」で以下の内容を提出。
2. 学籍番号、氏名、ペアの学籍番号 (または「個人作業」)、提出日時。名前の行は先頭に「@@@」を付けることを勧める。
3. 1 つ目の課題の再掲 (どの課題をやったか分かればよい)、プログラムのソースと「丁寧な」説明、および考察 (課題をやってみて分かったこと、分析、疑問点など)。
4. 2 つ目の課題についても同様。
5. 以下のアンケートの回答。

   Q1. 末尾再帰に対する再帰除去ができるようになりましたか。
   Q2. スタックを用いた再帰除去ができるようになりましたか。
   Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

＃7　単連結リスト

今回は次のことが目標となります。

- 動的データ構造の考え方を知る。
- 単連結リストを様々に操作できるようになる。

7.1　動的データ構造と単連結リスト

7.1.1　動的データ構造と領域の管理

データ構造 (data structure) とは、プログラム (やアルゴリズム) が扱うデータの「かたち」を指す言葉です。これまでのコードで配列、構造体、構造体の配列などを扱い、またそれらを組み合わせてスタック、キューなどを作って来ましたが、これらはすべてデータ構造の例です。

動的データ構造 (dynamic data structure) とは、ポインタを使って構造どうしを結び合わせることにより作り出されているデータ構造のことを言います。動的データ構造の特徴は、データ構造の「かたち」が大きく変化し得ることです。[1]

動的データ構造を作り出したり操作するときには、`malloc` による動的なメモリ割り当てが必要になります。C 言語では本来、使わなくなった領域を `free` により解放する必要がありますが、どの領域を使わなくなったか正確に把握することは繁雑なので、ここでは省略しています。

ここで示す例題程度のプログラムでは、解放を省略してもメモリが不足する心配はありません。そして今日の C 以外のプログラミング言語の多くでは最初から、使わなくなった領域を自動回収する**ごみ集め** (garbage collection、GC) 機能が搭載されています。また C 言語でも、長時間動き続けるプログラムの場合、「保守的 GC(conservative GC)」と呼ばれるライブラリを使うことで、ごみ集め機能を利用できます。[2]

7.1.2　単連結リスト

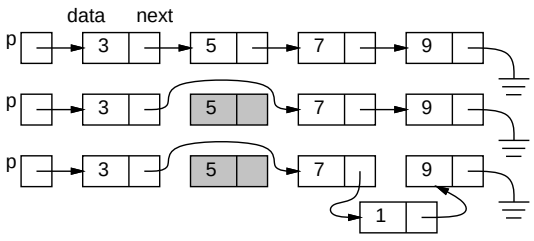


図 7.1: 単連結リストの例

動的データ構造の例として、**単連結リスト** (single linked list、単リストとも呼ぶ) を取り上げましょう。単リストは図 7.1 のように、複数の**節** (node) ないし**セル** (cell) がポインタによって直線的に結ばれた構造です (C 言語では通常、節には動的メモリ割り当てした構造体を使います)。

そして図の上から中のように、また中から下のように、ポインタを付け替えることでデータの並びを変化させられます (その過程で灰色の節のように使わなくなった節が生じることもあります)。配列

[1] これと対比して、これまで扱ったような、プログラムの冒頭で構造を作り出した後、その形が基本的に同じままのものを静的データ構造 (static data structure) と呼ぶこともあります。

[2] 必要になったら「conservative GC」で検索して探してください。

などではデータの並びを変化させるときにはデータを移動させる必要がありましたが、単リストではポインタの付け替えだけで自由に順番が入れ換えられます。これが動的データ構造の利点です。なお、最後のアース印はポインタ値として NULL(終わりを表す特別な値で stdlib.h で定義、多くのシステムでは 0) が入っていることを表します。

それでは最初の例題として、コマンド引数で渡した文字列をそれぞれ実数値に変換してリストにし、続いて 2 番目を削除し、(元の)3 番目の次に「1.0」を挿入するというものを見てみましょう。まずノードの構造体を定義し、データと次の要素へのポインタ値 (または NULL) を渡して新しいセルを作りそのポインタを返す関数 node_new を定義します。

plist は各セルのデータを順に出力して最後に改行します。それには、まず p のデータを出力し、「p = p->next;」で p を次のセルに進め、ということを p が NULL でない限り繰り返せばよいわけです。

```
// slistdemo1.c --- single-linked list demonstration
#include <stdio.h>
#include <stdlib.h>
struct node { double data; struct node *next; };
typedef struct node *nodep;

nodep node_new(double d, nodep n) {
  nodep p = (nodep)malloc(sizeof(struct node));
  p->data = d; p->next = n; return p;
}
void plist(nodep p) {
  while(p != NULL) { printf(" %g", p->data); p = p->next; }
  printf("\n");
}
nodep mklist(int n, char *a[]) {
  nodep p = NULL;
  for(int i = n-1; i >= 0; --i) { p = node_new(atof(a[i]), p); }
  return p;
}
int main(int argc, char *argv[]) {
  nodep p = mklist(argc-1, argv+1); plist(p);
  p->next->next = p->next->next->next; plist(p);
  p->next->next = node_new(1.0, p->next->next); plist(p);
  return 0;
}
```

それではコマンド引数のような文字列の配列 (ただしデータは先頭から全部入っているものとします) をリストに変換する mklist を見てみましょう。リストは後ろから順番に作るのが作りやすいです。最初 p に NULL を入れ、そして「p = node_new(データ, p);」でこれまでのリストの先頭に新しいデータのセルをつけ加えることを繰り返します。

最後に main です。argv の 1 番目以降を渡して (データの個数は argc-1) リストを作り、次に 2 番目のセルをバイパスします。そして次に、(現在の)2 番目のセルの次として新たなセルを指させます。その新たなセルの後ろは (現在の)3 番目のセルにしておけばよいわけです。最初の状態、削除後、そして挿入後の状態を plist で表示します。実行例を見てみましょう。

```
% ./a.out 3 5 7 9
 3 5 7 9
```

```
 3 5 9
 3 5 1 9
```

ときに、上の mklist と plist はループ版でしたが、再帰版も例示しておきます。

```
void plist(nodep p) {
  if(p == NULL) { printf("\n"); return; }
  printf(" %g", p->data); plist(p->next);
}
nodep mklist(int n, char *a[]) {
  if(n == 0) { return NULL; }
  return node_new(atof(*a), mklist(n-1, a+1));
}
```

plist は NULL なら改行するだけ。そうでなければ自分のデータを出力したあと「残りを打ち出して改行」するために自分自身を呼びます。mklist は「残っている個数が 0 なら NULL、そうでないなら自分を除いた残りのリストの前に自分のデータを持つセルをくっつけてそれを返す」わけです。

演習 1 上の例題をそのまま動かし、動作を確認しなさい。うまく動いたら、次のように変更してみよ。

a. 最後の要素を先頭につなぎ直す (その 1 つ前が最後になる)。
b. 先頭の要素を最後につなぎ直す (元の 2 番目が先頭になる)。
c. 2 番目と 3 番目のセルを入れ換える (データだけ入れ換えるのでなく、あくまでもつなぎ替えてセルの並びを入れ換えること)。
d. リストが何であれ、同じもの 2 つずつのリストに変更する (「1 2 3」→「1 1 2 2 3 3」のように)。
e. その他、好きなリストのつなぎ替えをおこなう。

7.1.3 例題: 並びエディタ

前節の例題ではデータは実行時に指定していましたが、操作はプログラムに書き込んであって固定でした。もっとその場で色々操作できた方が楽しそうなので、「並びエディタ」を作ってみます。これは、コマンドと (必要ならパラメタ) を 1 行打ち込むごとに指示された動作をします。当面次のコマンドがあります。

- e 値 値 … — (enter) 並びを設定する (これを「現在の並び」と呼ぶ)。
- a 値 値 … — (append) 指定した並びを現在の並びの後ろに追加。
- add 値 値 … — (add) 指定した並びの値を現在の並びに各々加算。
- d 番号 — (delete) 指定した番号の値を削除。
- p — (print) 現在の並びを表示する。
- q — (quit) プログラムを終了する。

実行のようすを示します。

```
% ./a.out
> e 1 2 3     ←現在の並び
> a 4 5 6     ←後ろに追加
> p           ←表示
1 2 3 4 5 6
```

```
> add 1 1 1 1  ←先頭の４つについて１を足す
> p
2 3 4 5 5 6
> d 2          ←２番を削除（先頭は０番）
> p
2 3 5 5 6
> q
%
```

ではコードを見てみましょう。だいぶ長くなったので、小分けにして見ていきます。まず先頭から plist まではこれまでと代わりません (include は増えていますが)。

```
// listeditor.c --- single-linked list editor
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
struct node { double data; struct node *next; };
typedef struct node *nodep;

nodep node_new(double d, nodep n) {
  nodep p = (nodep)malloc(sizeof(struct node));
  p->data = d; p->next = n; return p;
}
nodep mklist(int n, char *a[]) {
  if(n == 0) { return NULL; }
  return node_new(atof(*a), mklist(n-1, a+1));
}
void plist(nodep p) {
  if(p == NULL) { printf("\n"); return; }
  printf(" %g", p->data); plist(p->next);
}
```

次の nconc はリスト p の後ろに q をくっつけます。それには、p の最後のセルまで行き、その next フィールドを q にすればよいのです。

```
void nconc(nodep p, nodep q) {
  while(p != NULL && p->next != NULL) { p = p->next; }
  if(p != NULL) { p->next = q; }
}
```

削除は、数えながら削除するセルの 1 つ手前まで行きます。そして 1 つ手前のセルの次を削除する次のセルにすればよいわけです。

```
void delnode(nodep p, int n) {
  while(--n > 0 && p != NULL) { p = p->next; }
  if(p != NULL && p->next != NULL) { p->next = p->next->next; }
}
```

足し算ですが、2 つのリストの長い方に長さが揃うことにして、再帰を使っています。単純なケースですが、片方が NULL ならそれ以上足し算は必要でないので、他方が結果でよいわけです。残って

いるのは両方とも null でない場合なので、両方のリストの残りを再帰呼び出しにより足し算して、その前に自分の担当する 2 つのデータを足した値を持つセルを作ります。

```
nodep addlist(nodep x, nodep y) {
  if(x == NULL) { return y; }
  if(y == NULL) { return x; }
  return node_new(x->data+y->data, addlist(x->next, y->next));
}
```

ここから先は単リストと関係ありません。まず getl は指定された文字配列に 1 行ぶん読み込むもので、前にも使いました。getchar で 1 文字ずつ読みながら、読んだ文字が改行でも終わりの印でもなければそれをバッファに追加し、バッファの書き込み位置は 1 つ進めます (後置の++の働き)。最後に文字列の末尾にナル文字を入れますが、できた文字列の長さが 0 でファイル終わりなら false、それ以外は true を返します。

```
bool getl(char s[], int lim) {
  int c, i = 0;
  for(c = getchar(); c != EOF && c != '\n'; c = getchar()) {
    s[i++] = c; if(i+1 >= lim) { break; }
  }
  s[i] = '\0'; return c != EOF;
}
```

次に parse は、1 行入力したものを空白で分けてそれぞれの文字列の先頭を別の配列 (後の並びの配列) に入れます。空白はナル文字に書き換えることで、それぞれの文字列の終わりがあるようにします。文字を順番に調べながら、空白をナル文字に書き換えます。次に現在位置がナル文字であり、かつ「それが先頭文字か 1 つ手前の文字がナル文字なら」単語の先頭だということでその現在位置のポインタを語の並びの配列に入れ、入れる位置は 1 つ進めます。最後に語の数を返します。

```
int parse(char *a[], char *s) {
  int i, k = 0, len = strlen(s);
  for(i = 0; i < len; ++i) {
    if(s[i] == ' ') { s[i] = '\0'; }
    if(s[i] != '\0' && (i == 0 || s[i-1] == '\0')) { a[k++] = s+i; }
  }
  return k;
}
```

それでは main です。まず 1 行入力のバッファと語のリストの領域があり、並びはノードのポインタです。無限ループに入り、先頭で 1 行入力しますが入力が読めなければループを抜け出ます。読めた場合は語に分割し、先頭の語が何であるかによって各コマンドの動作に分岐します。「q」はループを抜けるだけ。「e」はコマンドより後部分を mklist でリストにしてそれを list に入れます。「a」は同様ですが、list に入れるかわりにその末尾に nconc でくつつけます。「add」も同様ですが、元のリストと指定したリストを addlist で足し算し、結果を list に入れ直します。「d」は dellist を読んで指定した番号を削除します。そして「p」は…上記以外の何であっても plist で表示するようにしています。

```
int main(int argc, char *argv[]) {
  char buf[200], *cmd[20];
  nodep list = NULL;
```

```
    while(true) {
      printf("> "); if(!getl(buf, 200)) { break; }
      int k = parse(cmd, buf);
      if(k > 0 && strcmp(cmd[0], "q") == 0) {
        break;
      } else if(k > 0 && strcmp(cmd[0], "e") == 0) { // enter list
        list = mklist(k-1, cmd+1);
      } else if(k > 0 && strcmp(cmd[0], "a") == 0) { // append list
        nconc(list, mklist(k-1, cmd+1));
      } else if(k > 1 && strcmp(cmd[0], "add") == 0) { // add list
        list = addlist(list, mklist(k-1, cmd+1));
      } else if(k > 1 && strcmp(cmd[0], "d") == 0) { // delete item
        delnode(list, atoi(cmd[1]));
      } else {
        plist(list);
      }
    }
    return 0;
  }
```

演習 2 上の例題をそのまま動かし、さまざまなリスト操作をやってみよ。納得したら、次のような機能追加をおこなえ (コマンド名や指定方法は好きに決めてよい)。

a. 足し算のかわりに掛け算をおこなう機能。
b. 並びを逆向きに変更する機能。
c. 並びを右巡回 (末尾の要素を先頭に移し残りを繰り下げる)。
d. 並びを左順解 (先頭の要素を末尾に移し残りを繰り上げる)。
e. 指定位置に要素を挿入する機能 (1 つだけでも並びでもよい)。
f. その他面白いと思う機能。

演習 3 上の並びエディタは「先頭の値は消せない」「先頭に要素を挿入できない」という欠陥がある。下の説明を読んで修正してみよ。

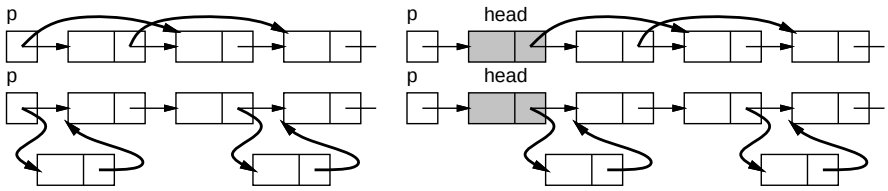


図 7.2: 普通の単リスト (左) と頭を持つ単リスト (右)

演習 3 のような問題が起きるのは、リストの先頭はセルではなく単独の変数で「形が違う」ところにあります。これを修正する 1 つの方法は、リストの先頭としてダミーのセルを常に置くことにして、その次からをデータとして扱うことです。このダミーのセルを「**頭** (head)」と呼びます。演習 3 は、リストが常に頭を持つように変更することで解決可能です (別の方法もありますが)。図 7.2 の右側のように、頭を置くことで先頭への追加/削除が途中への追加/削除と同じ形の操作にできるわけです。

7.2 単連結リストを使ったスタックとキューの実装

7.2.1 単連結リスト版と配列版の比較

単連結リストはさまざまな用途に使えますが、その代表例としてスタックやキューの実装に使うというものがあります。以前にスタックやキューを扱った時は実装に配列を使っていましたが、配列版と対比して単連結リストを使う版には次のような得失があります。

○ データ量の上限を設けなくて済む。

△ データ 1 個あたりの領域オーバヘッド (データ本体以外に必要とする余分な記憶領域) は多くなる。

前者ですが、配列は最初にサイズを決めて割り当てるので、そのサイズまで使ったら「満杯」になります。一方、単連結リストでは上限はなく、動的メモリ割り当てができる限りは要素を増やして行けます。ただし、配列を使う実装でも「満杯になったらより大きい配列を取り直して内容をコピーする」方法を使えば上限の問題を回避できます。

後者については、配列では格納するデータの値が配列要素として並んでいるので、余分に必要な場所はレコード領域など少量の、しかも決まった大きさの領域だけです。しかし単連結リストでは、データを 1 個格納するたびに「次の要素のポインタ」が必要なので、データ量に比例して余分に必要な場所が増えます。

7.2.2 単連結リストを使ったスタックの実装

スタックは片方の端だけでデータを出し入れするデータ構造であるので、単連結リストとの相性はとても良いです。図 7.3 のように、セル以外に必要な領域は top のポインタだけになります。

push するときは新しいセルを作ってこれまでの先頭を next で指した上で、top には新しいセルを指させます (top が NULL だった場合もコードは同じままで大丈夫です)。pop するときはその逆で、top が指しているセルのデータを返し、top はそのセルの next に書き換えます。

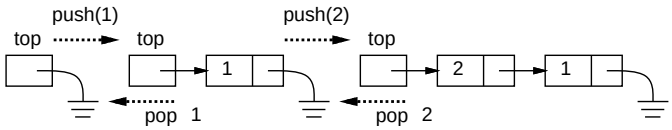


図 7.3: 単リストを使ったスタックの実装

以下に istack.h を再掲します。インタフェースはこれと同じままで、内部では図 7.3 の構造を用いるようにすればよいわけです。

```
// istack.h --- int type stack interface
#include <stdbool.h>
struct istack;
typedef struct istack *istackp;
istackp istack_new(int size); // allocate new stack
bool istack_isempty(istackp p); // test if the stack is empty
void istack_push(istackp p, int v); // push a value
int istack_pop(istackp p);      // pop a value and return it
int istack_top(istackp p);      // peek the topmost value
```

1 つだけ、作るときに size(上限サイズ) を渡していましたが、それはどうしましょうか? いくつか可能性があります。どれも一定の合理性があります。

- 単連結リスト版では不要なので、引数 size を削除する。
- インタフェースは変えたくないので、size は削除しないが単に使わないことにする。
- 現在いくつデータが入っているかも数えるようにして、size も保持して満杯かどうかの管理をする。

演習 4 連結リスト版のスタックの実装を作れ。これまでに作ったスタックを利用するプログラムと組み合わせて動かし動作を確認すること。size の扱いは好きに決めてよい。

7.2.3　単連結リストを使ったキューの実装

キューの場合は、データを入れる場所と取り出す場所が違っているので、top と last という 2 つのポインタが必要で、スタックより実装は複雑です (配列版でもそうでした)。

コードを簡潔にするには、先に説明した頭 (head) を持つリストを使うのがおすすめです。図 7.4 にそのような実装を示しました。空っぽのときは頭のセル (灰色にしてあります) だけがあり、top も last もそこを指しています。

データを追加するときは、last の指しているセルの next に新しいセルをくっつけてデータを入れ、last はその新しいセルに変更します。データを取り出すときは、top の指しているセル…は頭ですから、その next が指しているセルからデータを取り、next はそのまた next で置き換えます。

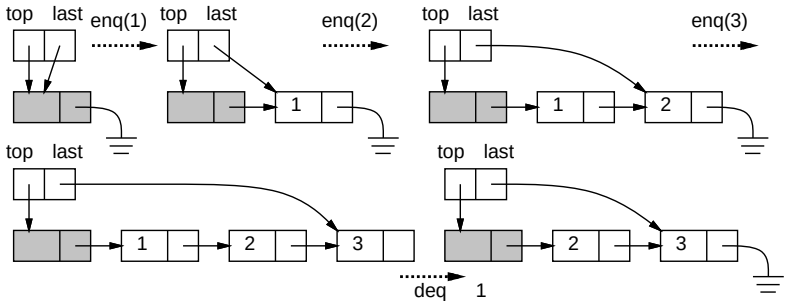


図 7.4: 単リストを使ったキューの実装 (頭つき)

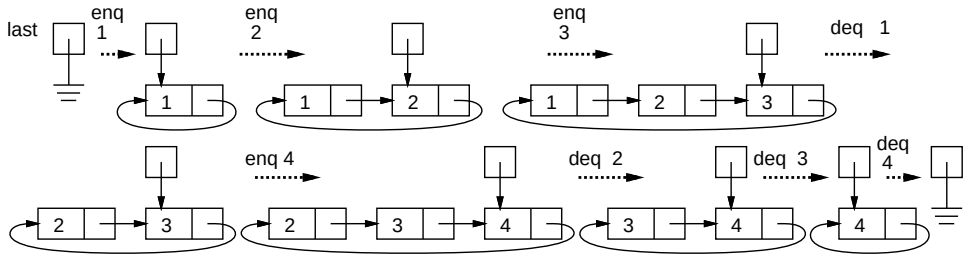
以下に iqueue.h を再掲します。これもサイズの扱いは先に述べたような選択肢があります。さらに、isfull があるのですが、これはデータ数を管理しない場合は「決して満杯にならない」ことにしてもよいですし、動的メモリ割り当てが失敗したときが満杯ということにしてもよいです。

```
// iqueue.h --- int type queue interface
#include <stdbool.h>
struct iqueue;
typedef struct iqueue *iqueuep;
iqueuep iqueue_new(int size);
bool iqueue_isempty(iqueuep p);
bool iqueue_isfull(iqueuep p);
void iqueue_enq(iqueuep p, int v);
int iqueue_deq(iqueuep p);
```

演習 5 連結リスト版のキューの実装 (頭つき版) を作れ。size の扱い、isfull の扱いは好きに決めてよい。単体テストを実施すること (#3 の単体テストが参考になる)。

演習 6 頭なしでもキューの実装は可能である (空のときを特別扱いする必要が生じるだけ)。そのような版の実装を作れ。size の扱い、isfull の扱いは好きに決めてよい。単体テストを実施すること (#4 の単体テストが参考になるが、満杯の扱いが違うと思われるのでそこは違うはず)。

演習 7 循環 (環状) リストとは下図のように、リストの最後の次が先頭を指すようにして循環した形のリストをいう。循環リストを使うことで、last ポインタだけでキューを実現できる (循環リストでは last の次は先頭なので先頭を別に覚えなくてよい)。ただし要素が 1 個もないときは別扱いとなる。循環リストを用いたキューの実装を作成せよ。size の扱い、isfull の扱いは好きに決めてよい。これも単体テストを実施すること。



本日の課題 7A

「演習 1」〜「演習 6」で動かしたプログラム 1 つを含むレポートを本日中 (授業日の 23:59 まで) に提出してください。

1. sol または CED 環境で「`/home3/staff/ka002689/prog19upload 7a` ファイル名」で以下の内容を提出。
2. 学籍番号、氏名、ペアの学籍番号 (または「個人作業」)、提出日時。名前の行は先頭に「@@@」を付けることを勧める。
3. プログラムどれか 1 つのソースと「簡単な」説明。
4. レビュー課題。提出プログラムに対する他人 (ペア以外) からの簡単な (ただしプログラムの内容に関する) コメント。
5. 以下のアンケートの回答。

   Q1. 単連結リストの概念を理解しましたか。
   Q2. 並びエディタのようなコマンドに応答するプログラムの作り方を納得しましたか。
   Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

次回までの課題 7B

「演習 1」〜「演習 6」(ただし 7A で提出したものは除外、以後も同様) の (小) 課題から選択して 2 つ以上プログラムを作り、レポートを提出しなさい。できるだけ複数の演習から選ぶこと。レポートは次回授業前日 23:69 を期限とします。

1. sol または CED 環境で「`/home3/staff/ka002689/prog19upload 7b` ファイル名」で以下の内容を提出。
2. 学籍番号、氏名、ペアの学籍番号 (または「個人作業」)、提出日時。名前の行は先頭に「@@@」を付けることを勧める。
3. 1 つ目の課題の再掲 (どの課題をやったか分かればよい)、プログラムのソースと「丁寧な」説明、および考察 (課題をやってみて分かったこと、分析、疑問点など)。
4. 2 つ目の課題についても同様。
5. 以下のアンケートの回答。

   Q1. 単連結リストの操作が自由にできるようになりましたか。
   Q2. スタックやキューの単連結リスト実装について理解しましたか。
   Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

7.3　付録: デバッガ gdb

皆様はプログラムが動かないとき、どのようにデバッグしますか? 「コードをよく見る」「要所で値を printf 出力」が普通だと思いますが、別の方法として「デバッガ (debugger)」というツールがあります。デバッガでは、コマンドを使って「ここで止める」「値を表示する」「少しずつ動かす」ができるので、効率よくデバッグできる…「こともあり」ます。

ただし、少しずつ実行することは、とても時間を取る (遅いという意味ではなく人間がそれを見るので手間取る) ので、結果として効率が悪く、結局プログラムをよく見る方が良かった (それなら修正すべき箇所が直接わかる)、ということも起こります。なので万能ではありませんよ、という警告はした上で、**gdb** というデバッガを紹介します。

gdb は gcc と一緒のプロジェクトで開発されてきたデバッガであり、gcc が生成するデバッグ情報を使ってさまざまな表示をしてくれます。デバッグ情報を生成させるには、gcc に「`-g`」オプションを指定する必要があります。そのため、使うときは次のようにます。

```
% gcc8 -g その他の指定はこれまで同様
% gdb8 a.out    ← gcc が生成した実行形式ファイルを指定
... メッセージ多数 ...
(gdb)          ← gdb のプロンプト
```

gdb で最低限覚えて欲しいコマンドは次のものがあります (良く使うものなので大抵は 1 文字の省略形が使えます。「bt」のみ 2 文字ですが)。

- break (b) — 中断点を設定する。中断点を設定しないで動かすと、一気に全部動いてしまい途中の様子が調べられないので、通常はまず中断点を指定する。次の 3 つの書き方を覚えるとよい。
  - 「`b` 関数名」 — 指定した関数の入口で停止。
  - 「`b` 行番号」 — 現在のファイルの指定した行で停止。
  - 「`b` ファイル名:行番号」 — 指定ファイルの指定行で停止 (複数ファイルから成るプログラムで使用)。
- run (r) — プログラムを実行開始する。コマンド引数をここで指定する。たとえば「`./a.out 1 2`」のように動かすプログラムであれば、gdb の中では「`r 1 2`」で実行開始させる。とくにコマンド引数を指定しないプログラムなら「`r`」でよい。
- quit (q) — gdb を終了する。

r コマンドで実行開始したあと、中断点に到達すると実行が中断され、gdb プロンプトが表示されます (また、bus error などのエラー終了が起きた場合は中断点を設定していなくてもそこで中断が起きます)。中断した箇所で実行するコマンドに次のものがあります。

- backtreace (bt) — 「どの関数が何行目でどの関数を呼び…」という呼び出の連鎖を表示。
- list (l) — 中断点付近のソースコードを表示する。
- print (p) — 変数や式の値を表示する。「`p i`」のように単独変数の値を表示するほか、「`p a[i]`」「`p q->data`」のように配列、レコード、ポインタを扱うこともできる。
- continue (c) — 実行を再開する。次にまた中断点に到達するまで一気に実行される。
- next (n) — 1 行ぶん実行する。次の行に来たときにそこで止まるので、中断点を設定していなくても 1 行ずつ実行の流れを調べ、そこでの値を (p コマンドで) 調べることができる。なお、関数呼び出しがあった場合はその中は一気に実行して次の行に進む。
- step (s) — next と同様だが、関数呼び出しがあった時にはその関数内の次の行へ行く。

では実際に使ってみるとして、サンプルコードに「再帰版とループ版の階乗」を用意しました。

```
// factdemo.c --- two fact function for gdb exercise.
#include <stdio.h>
#include <stdlib.h>
int fact1(int n) {
  int i, r = 1;
  for(i = 1; i <= n; ++i) {
    r *= i;
  }
  return r;
}
int fact2(int n) {
  if(n <= 0) { return 1; }
  int r = fact2(n-1);
  return n * r;
}
int main(int argc, char *argv[]) {
  int n = atoi(argv[1]);
  printf("fact1(%d) == %d\n", n, fact1(n));
  printf("fact2(%d) == %d\n", n, fact2(n));
  return 0;
}
```

最初にループ版、次に再帰版の階乗計算の関数が呼ばれるので、それぞれを中断点に指定して動かすことにします。中断したあと、ループ版ではステップ実行しつつ、時々変数の値を表示して検討します。再帰版では再帰呼び出しのたびに中断点に来るので、「c」を使って継続していけばよいですが、最後の再帰呼び出しのあと戻ってきたときの計算を調べるにはステップ実行を使っています (または行番号指定で中断点を設定しておく方がよかったかも知れません)。

```
% gcc8 -g factdemo.c
% gdb a.out
Copyright (C) 2018 ...
Reading symbols from a.out...done.
(gdb) break fact1    ←fact1の入口で中断
Breakpoint 1 at 0x40056d: file factdemo.c, line 5.
(gdb) break fact2    ←fact2の入口で中断
Breakpoint 2 at 0x4005a3: file factdemo.c, line 12.
(gdb) r 5
Starting program: /home3/staff/ka002689/a.out 5
Breakpoint 1, fact1 (n=5) at factdemo.c:5 ←中断設定
5         int i, r = 1;
(gdb) s                                   ←ステップ実行
6         for(i = 1; i <= n; ++i) {
(gdb) s
7           r *= i;
(gdb) s
6         for(i = 1; i <= n; ++i) {
```

```
(gdb) s
7           r *= i;
(gdb) s
6         for(i = 1; i <= n; ++i) {
(gdb) s
7           r *= i;
(gdb) p r                                   ←r を表示
$1 = 2
(gdb) p i                                   ←i を表示
$2 = 3
(gdb) c                                     ←一気に実行
Continuing.
fact1(5) == 120                  ←プログラムによる表示
Breakpoint 2, fact2 (n=5) at factdemo.c:12 ←中断設定
12        if(n <= 0) { return 1; }
(gdb) c                                     ←再開
Continuing.
Breakpoint 2, fact2 (n=4) at factdemo.c:12 ←中断
12        if(n <= 0) { return 1; }
(gdb) c
Continuing.
Breakpoint 2, fact2 (n=3) at factdemo.c:12
12        if(n <= 0) { return 1; }
(gdb) c
Continuing.
Breakpoint 2, fact2 (n=2) at factdemo.c:12
12        if(n <= 0) { return 1; }
(gdb) c
Continuing.
Breakpoint 2, fact2 (n=1) at factdemo.c:12
12        if(n <= 0) { return 1; }
(gdb) c
Continuing.
Breakpoint 2, fact2 (n=0) at factdemo.c:12
12        if(n <= 0) { return 1; }
(gdb) bt                         ←呼び出し系列を表示
#0  fact2 (n=0) at factdemo.c:12
#1  0x00000000004005bd in fact2 (n=1) at factdemo.c:13
#2  0x00000000004005bd in fact2 (n=2) at factdemo.c:13
#3  0x00000000004005bd in fact2 (n=3) at factdemo.c:13
#4  0x00000000004005bd in fact2 (n=4) at factdemo.c:13
#5  0x00000000004005bd in fact2 (n=5) at factdemo.c:13
#6  0x0000000000400618 in main (argc=3, argv=0x7fffffffd7f8) at factdemo.c:19
(gdb) s                                  ←ステップ実行
15      }
(gdb) s
14        return n * r;
```

```
(gdb) p r                                    ←r を表示
$3 = 1
(gdb) p n                                    ←n を表示
$4 = 1
(gdb) s
15      }
(gdb) s
14        return n * r;
(gdb) s
15      }
(gdb) s
14        return n * r;
(gdb) p r                                    ←r を表示
$5 = 2
(gdb) p n                                    ←n を表示
$6 = 3
(gdb) c                                      ←再開
Continuing.
fact2(5) == 120            ←プログラムからの出力
[Inferior 1 (process 44205) exited normally]
(gdb) q                                     ←終了
```

このように、デバッガを使うと「途中でどんな値になってるのか不明」「どの経路を通っているのか不明」なときに、そのことが調べられるという点では便利です。ただ、ダメなプログラムを長時間掛けて調べるよりは、分かりにくいと思ったらきれいに書き直す方が早道でお得ということもいえるので、うまく考えて使ってください。

＃8　双連結リスト

今回は次のことが目標となります。

- 双連結リストの考え方と操作方法を理解する。
- 連結リストを使ったエディタアプリケーションについて知る。

8.1　双連結リスト

8.1.1　連結リストのバリエーション

連結リスト (linked list) はセルが直線的に並んだ動的データ構造です。前回までに扱った単連結リスト (単リスト) では、セルはデータに加えて次のセルを指すポインタ値のフィールド next を持ち、これによって片方向にセルを連結していました (図 8.1)。先頭や末尾へのセルの挿入・削除の操作をしやすくするために、頭 (head) や尻尾 (tail) と呼ばれるダミーセル (実際のデータは入れないセル) を常に置く方法があります。

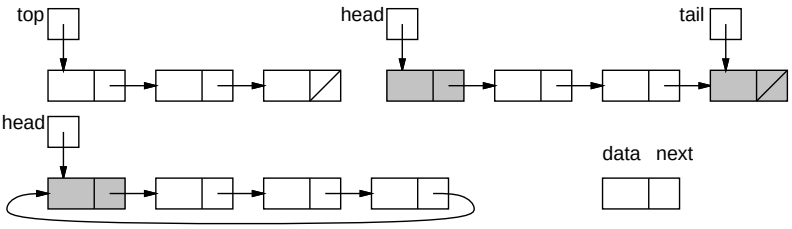


図 8.1: 単連結リストのバリエーション

リストの終わりのポインタには NULL または nil と呼ばれる特別な値を入れます (前回までアース記号で表していましたが、ここからは簡単のため斜線で表すようにしました)。最後のセルが先頭のセルを指すようにしたものを循環リストと呼び、この場合は NULL を使わなくて済みます (間違って NULL をたどるとプログラムが死ぬのでそれを避けられるという利点があります)。循環リストでも頭を持たせることができます。

単連結リストの弱点は、途中のセルをポインタで指している場合、次のセルをたどるのは容易だが、前のセルをたどることは手間が掛かる (先頭のポインタから繰り返し次をたどり、現在のセルの 1 つ手前まで来る必要がある) ことです。

このため、現在のセルの「次に」新しいセルを挿入したり、現在のセルの「次の」セルを削除することは簡単ですが、現在のセルの「前に」新しいセルを挿入したり、現在のセル「そのものを」削除するのは手間が掛かります。

これらの弱点を解消できる連結リストの方式に、**双連結リスト** (double linked list、双リスト、双方向リスト) があります (図 8.2)。

双連結リストでは、1 つ先のセルへのポインタ next に加えて、1 つ前のセルへのポインタ prev も持つため、これを使って前方向へリストをたどれます。

双連結リストの弱点は、1 つのセル当たりポインタのフィールドが 1 つ増えるので領域が余分に掛かることですが、今日ではメモリは潤沢にあるのであまり問題ありません。ポインタの付け変えの手間も単連結リストより多く (倍に) なりますが、それよりも操作しやすさによる利点の方が大きいと考えます。

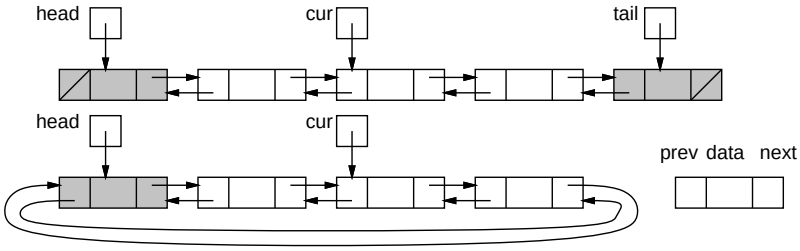


図 8.2: 双連結リスト (双方向リスト)

双連結リストでも頭や尻尾を持たせることがあります。また、双連結リストの循環リストも多く使われます。

8.1.2 双連結リストを使ったエディタバッファ

それでは、双連結リストを使ったエディタバッファを作ってみましょう。エディタバッファというのは何かというと要するに、行単位で文字列を格納し、それぞれの行に移動したりその内容を書き換えたりできるような機能を指すものとします。例によって構造体で情報隠蔽します。

今回は簡単のため、それぞれのノードに 100 バイト (100 文字) の文字配列を含め、そこに 1 行ぶんの文字列を格納します。なので、その 100 という長さも `MAXSTR` という名前でヘッダに定義します。ヘッダファイルは次の通り。

```
// ebuf.h --- editor buffer API.
#include <stdbool.h>
#define MAXSTR 100
struct ebuf;
typedef struct ebuf *ebufp;
ebufp ebuf_new();             // create ebuf
bool ebuf_iseof(ebufp e);     // see if current line is EOF
bool ebuf_forward(ebufp e);   // forward 1 line
bool ebuf_backward(ebufp e); // backward 1 line
void ebuf_top(ebufp e);       // go to top
char *ebuf_str(ebufp e);     // obtain current line string
void ebuf_insert(ebufp e, char *s); // insert a line
```

バッファの最後には「EOF(end of file のつもり)」と書かれた特別な行 (EOF 行) があり、バッファが空のときは現在行は EOF 行です。`ebuf_iseof` で EOF 行にいるかどうか調べられます。また、`ebuf_top` で先頭に行き、`ebuf_forward` と `ebuf_backward` で 1 行先/前へ動けますが、EOF 行より先や先頭より前は行けないのでそのような場合は `false` を返します。`ebuf_str` は現在行の文字列 (先頭文字へのポインタ) を返します。そして `ebuf_insert` は現在行の上に指定した文字列を持つ行を挿入します。

これを使った例をまず見ましょう。バッファを作り、3 行ぶん文字列を挿入し、先頭から順に表示して、そのあとまた前に戻りながら各行を表示します。

```
// ebufdemo.c --- demonstration of ebuf.
#include <stdio.h>
#include "ebuf.h"

int main(void) {
  ebufp e = ebuf_new();
```

```
  ebuf_insert(e, "abc");
  ebuf_insert(e, "def");
  ebuf_insert(e, "ghi");
  ebuf_top(e);
  while(!ebuf_iseof(e)) {
    printf("%s\n", ebuf_str(e)); ebuf_forward(e);
  }
  while(ebuf_backward(e)) { printf("%s\n", ebuf_str(e)); }
  return 0;
}
```

実行例は次の通り。

```
% gcc8 ebufdemo.c ebuf.c
% ./a.out
abc
def
ghi
ghi
def
abc
```

では実装を見てみましょう。内部では双連結循環リストを使いますから、構造体 ebuf は頭のセルへのポインタと現在セルへのポインタがあればよいです。セルそのものは構造体 line で表し、そのフィールドといては前後セルへのポインタと char の配列が含まれます。

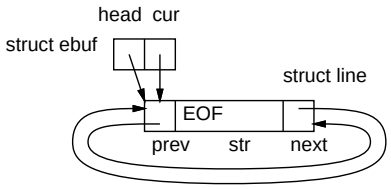


図 8.3: エディタバッファの初期状態

新しいバッファを作るときは (図 8.3)、まず ebuf の構造体を割り当て、head フィールドに EOF 行のセルを指させます。そのあと、cur フィールドにも EOF 行の prev と next にもそのセルへのポインタを指させます。これで循環リストが初期化できました。最後に EOF 行には文字列「EOF」を格納し、構造体 ebuf のポインタを返します。

```
// ebuf.c --- editor buffer implementation
#include <stdlib.h>
#include <string.h>
#include "ebuf.h"
struct line {
  struct line *prev, *next; char str[MAXSTR];
};
struct ebuf { struct line *head, *cur; };
ebufp ebuf_new() {
  ebufp r = (ebufp)malloc(sizeof(struct ebuf));
```

```
  r->head = (struct line*)malloc(sizeof(struct line));
  r->cur = r->head->next = r->head->prev = r->head;
  strcpy(r->head->str, "EOF"); return r;
}
bool ebuf_iseof(ebufp e) { return e->cur == e->head; }
bool ebuf_forward(ebufp e) {
  if(e->cur == e->head) { return false; }
  e->cur = e->cur->next; return true;
}
bool ebuf_backward(ebufp e) {
  if(e->cur->prev == e->head) { return false; }
  e->cur = e->cur->prev; return true;
}
void ebuf_top(ebufp e) { e->cur = e->head->next; }
char *ebuf_str(ebufp e) { return e->cur->str; }

void ebuf_insert(ebufp e, char *s) {
  struct line *p = (struct line*)malloc(sizeof(struct line));
  strncpy(p->str, s, MAXSTR); p->str[MAXSTR-1] = '\0';
  p->prev = e->cur->prev; p->next = e->cur;
  e->cur->prev->next = p; e->cur->prev = p;
}
```

head は常に EOF 行を指しているので、EOF 行かどうかは cur が head と等しいかどうか見れば分かります。1 行進む場合は cur を cur->next に変更すればいいのですが、その前に EOF 行を超えて進まないようにチェックしています。1 行戻る場合も同様です。先頭行は (循環リストなので)EOF 行の次に cur を設定するだけです。文字列を返すのは cur->str を返すだけです。

最後の挿入が最もややこしいですが、まず新しい行のセルを割り当て、パラメタの文字列を strncpy でコピーし、万一最後のナル文字がないと (100 文字以上あったとき) トラブルになりがちなので最後にナル文字を格納します。その後が連結リストの操作で、まず新しいセルの次を現在行、前を現在行の 1 つ前とします。続いて、現在行の 1 つ前の次と現在行の 1 つ前をいずれも新しいセルとします (図 8.4)。cur->prev を先に書き換えてしまうとまずいので注意が必要です。

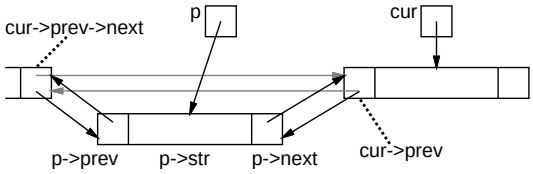


図 8.4: 双連結リストへの挿入

きちんと動くことを確認するため、単体テストを作りましょう。バッファのデータは文字列なので、#03 で作った expect_str ををを使います。

```
// test_ebuf.c --- unit test for cbuf.
#include <stdio.h>
#include <string.h>
#include "ebuf.h"
void expect_str(char *s1, char *s2, char *msg) {
```

```
  printf("%s '%s':'%s' %s\n", strcmp(s1, s2)?"NG":"OK", s1, s2, msg);
}
int main(void) {
  ebufp e = ebuf_new(); ebuf_insert(e, "abc"); ebuf_insert(e, "def");
  ebuf_top(e); expect_str(ebuf_str(e), "abc", "line 1: abc");
  ebuf_forward(e); expect_str(ebuf_str(e), "def", "line 2: def");
  ebuf_insert(e, "ghi"); ebuf_top(e);
  ebuf_forward(e); expect_str(ebuf_str(e), "ghi", "new line 2: ghi");
  ebuf_forward(e); expect_str(ebuf_str(e), "def", "new line 3: def");
  return 0;
}
```

動かしたようすは次の通り。

```
% ./a.out
OK 'abc':'abc' line 1: abc
OK 'def':'def' line 2: def
OK 'ghi':'ghi' new line 2: ghi
OK 'def':'def' new line 3: def
```

演習 1 エディタバッファの例題をそのまま動かせ。単体テストも動かしてみること。OK なら、次のことをやってみよ。追加した機能が正しく動くことを確認できるように単体テストを作成すること。

a. ebuf に現在行の内容を指定した文字列に取り換える命令 ebuf_replace を追加する。
b. 削除命令 ebuf_delete を追加する。EOF 行は消さないようにすること。
c. 上と同様たが、ただし「別の」リストを用意しておき、図削除したセルがそちらにつながるようにする。そして、その別のリストからセルを外して現在位置の上に挿入する ebuf_yank を作る (図 8.5 のように、消してから移動して別の場所で戻すことで行を移動できるようになる)。

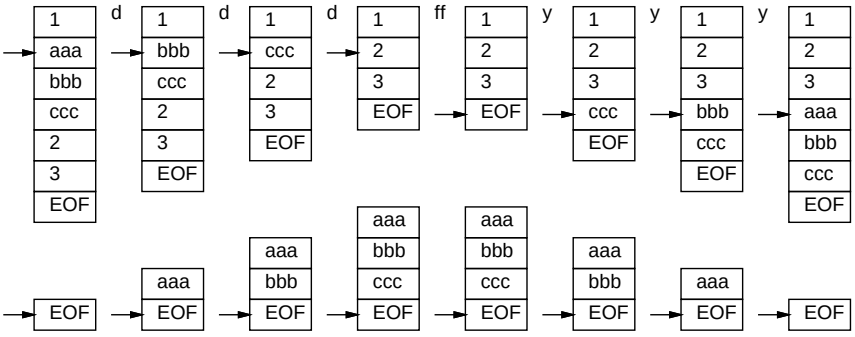


図 8.5: カットバッファの実装

d. 上記に加えて、「削除はせずに現在行のコピーを別の場所に挿入する」 命令 ebuf_copy を作る (現在行は 1 つ下に移る)。これにより、複数行をコピーして別の場所に挿入できるようになる。
e. 図 8.5 を見ると、「別の場所」も本体のバッファと同じ構造でよさそうである。そこで実際にそのようにして、「本体の場所と別の場所を交換する」 命令 ebuf_swap を作る (交換したあとの現在位置は EOF 行でよい)。これにより、数行だけ必要なときは必要な行だけ消してから交換すればよくなる。

f. バッファの内容を上下ひっくり返す (先頭行が最後になる) 機能を追加する。

g. そのほか、エディタバッファにあると便利と思う機能を追加する。

8.2　行エディタを作る

8.2.1　基本的な行エディタ

行エディタ (line editor) とは、行が編集できるテキストエディタ…ではなく (普通のテキストエディタはどれも行は編集できます)、行単位で現在位置を移動したり内容を編集できるような、**画面エディタ** (screen editor) ではないエディタを言います。

画面エディタとは画面上に編集バッファの内容が常に表示されていて、それを見ながら挿入や削除ができるもので、今日の普通のエディタはすべてそうです。しかし、現在のような画面上の表示ができるようになる前に、タイプライタ端末 (表示はタイプライタのように紙に印字することで行う) しか無かった時代があり、そのときは画面エディタが使えず、行エディタが多く使われていました (今でも Unix には ed など行エディタが標準で備わっています)。

ここでは簡単な行エディタを作ってみせます。コマンドは次のものです。

- `q` — エディタを終わる
- `p` — 現在行を表示
- `t` — 先頭行に行く
- `f` — 1 行下に行く
- `b` — 1 行上に行く
- `i` 文字列 — 文字列を行として現在行の直前に挿入
- 空行 (その他任意の行) — `f` と `p` の動作を実行

実際に動かす様子を見てみましょう。

```
% gcc8 editor.c ebuf.c
% ./a.out
> iThis is a pen.  ←挿入
> iThat is a dog.  ←挿入
> t                ←先頭へ
> p                ←表示
   This is a pen.
> f                ← 1 行下へ
> iHow are you?    ←挿入
> t                ←先頭へ
> p                ←表示
   This is a pen.
>                   ←次の行へ行き表示
   How are you?
>                   ←次の行へ行き表示
   That is a dog.
>                   ←次の行へ行き表示
   EOF
> q                ←終了
```

まあ、使いやすくはないかもですが、一応編集はできそうです (削除は後で追加するとして)。ではコードを見てみましょう。`getl` は前から使っているものです。

```
// editor.c --- a simple line editor.
#include <stdio.h>
#include "ebuf.h"

bool getl(char s[], int lim) {
  int c, i = 0;
  for(c = getchar(); c != EOF && c != '\n'; c = getchar()) {
    s[i++] = c; if(i+1 >= lim) { break; }
  }
  s[i] = '\0'; return c != EOF;
}
int main(void) {
  char buf[200];
  ebufp e = ebuf_new();
  printf("> ");
  while(getl(buf, 200)) {
    if(buf[0] == 'q') { // quit
      break;
    } else if(buf[0] == 'p') { // print
      printf("   %s\n", ebuf_str(e));
    } else if(buf[0] == 't') { // top
      ebuf_top(e);
    } else if(buf[0] == 'f') { // fwd
      ebuf_forward(e);
    } else if(buf[0] == 'b') { // back
      ebuf_backward(e);
    } else if(buf[0] == 'i') { // insert
      ebuf_insert(e, buf+1);
    } else {  // other --- fwd and print
      ebuf_forward(e); printf("   %s\n", ebuf_str(e));
    }
    printf("> ");
  }
  return 0;
}
```

要するに、1 行読み込み、先頭の文字に応じてそれぞれのコマンドの動作を実行すればよいわけです。

8.2.2 ファイルの読み書き

ファイルが読み書きできないと実際の編集に使えないので、とりあえず最低限を説明します。読むときも書くときも、次の手順によります。

- 「FILE *f = foepn(ファイル名, モード);」でストリームを開き、結果を FILE*型として受け取ります。ストリームとは、読み書きする対象の文字をやりとりする「通路」だと思ってください。何か問題があれば (例: ファイルが無い、ディレクトリ間違い等)、NULL が返されます。
- 書くときは printf の類似版「fprintf(ストリーム, 書式文字列, 値)」を使うことで任意のものが出力できます。

- 読むときは 1 行単位で読むのが簡単ですが、それには「fgets(文字配列，長さ上限，ストリーム)」を使い、文字配列に 1 行ぶん読み込みます。ただしこの場合、改行文字が最後についています。fgets はこれ以上読めない場合 (ファイルの終わり等) は NULL を返します。
- いずれにせよ、最後は「fclose(ストリーム)」で閉じる (後始末する) 必要があります。

では、読む方から見ましょう。fopen に失敗したらすぐ戻ります。そうでない場合は fgets を繰り返し呼び (NULL が返されたら終わる)、読み込んだ文字列の最後の文字 (改行文字のはず) をナル文字に書き換えてから ebuf に挿入します。

```
bool readfile(ebufp e, char *fname) {
  char str[200];
  FILE *f = fopen(fname, "r");
  if(f == NULL) { return false; }
  while(fgets(str, 200, f) != NULL) {
    int len = strlen(str);
    if(len > 0) { str[len-1] = '\0'; }
    ebuf_insert(e, str);
  }
  fclose(f); return true;
}
```

書く方がどちらかといえば簡単です。fopen に失敗したらすぐ戻ります。そうでない場合は、まず先頭に行き、EOF 行でない間、現在行を fprintf で出力してから次の行に進みます。

```
bool writefile(ebufp e, char *fname) {
  FILE *f = fopen(fname, "w");
  if(f == NULL) { return false; }
  ebuf_top(e);
  while(!ebuf_iseof(e)) {
    fprintf(f, "%s\n", ebuf_str(e)); ebuf_forward(e);
  }
  fclose(f); return true;
}
```

呼び出し方ですが、先の main で i コマンドと同様、readfile(e, buf+1)、writefile(e, buf+1) のように呼び出せばよいですが、読み書きが失敗だったとき (false が返されたとき) は何かその旨を出力した方がよいでしょう。

演習 2 前節にある簡単な行エディタを動かし、動作を確認しなさい。OK なら、以下のことをやってみなさい。実際に簡単なプログラムを改良したエディタで作成/編集してみて体験を報告すること。

a. ファイル読み書きコマンド「r ファイル名」「w ファイル名」を追加する。
b. 移動コマンドを「f 行数」「b 行数」のように何行移動するかを指定もできるようにする (他のコマンドや以下で追加するコマンドも必要に応じて同様に)。
b. 行削除コマンド「d」を追加する。
c. 行の内容を取り換えるコマンド「s 文字列」を追加する。
d. 演習 1c のように削除した結果を戻せるコマンド「y」を追加する。
e. 演習 1d のようにカットバッファと現在のバッファを交換するコマンド「x」を追加する。バッファをもっと多数持てるようにしてもよい。
f. その他あったらよいと思う機能を追加する。

8.3 画面エディタ option

8.3.1 ncurses とその機能

エディタが作れるようになりましたが、やはり行エディタよりは画面エディタの方がいいですね。画面エディタを作るには、画面の任意の位置で文字を表示したり消したりできる必要があります。その方法は色々ありますが、ここでは Unix に備わっている、端末 (ターミナル) 上で画面制御を行うライブラリ ncurses を使います。いきなり例題を見てみましょう。

```
// ncursesdemo.c --- show usage of ncurses.
#include <ncurses.h>
#include <stdlib.h>

int main(void) {
  initscr(); noecho(); cbreak(); system("stty raw"); clear();
  move(10, 10); addstr("press any key"); refresh();
  int ch = getch(); addch('a'); addch('b'); refresh();
  ch = getch(); move(10, 15); insch('a'); insch('b'); refresh();
  ch = getch(); delch(); move(10, 20); clrtoeol(); refresh();
  ch = getch(); endwin(); return 0;
}
```

ヘッダファイルは ncurses.h と stdlib.h が必要です。冒頭部分はだいたい固定で、initscr() で ncurses の初期化をおこない、noecho でキー入力を画面にそのまま表示しないモードに変更し (エディタなので表示は自前で制御したい)、cbreak で 1 文字入力したらすぐそれが読めるモードにし (通常は改行文字が来るまでプログラムには渡さない)、さらに system コマンドは Unix コマンドを実行する関数ですが、それを使って「^C など制御文字の扱いを止める」ようにします (^C を打ったらエディタが強制終了してしまうのでは悲しいですから)。これらと対になる後始末は endwin で、これで ncurses が各種状態を復元してくれます。

プログラム中で繰り返し使う個別の機能は箇条書で説明しましょう。refresh が分かりづらいと思いますが、エディタでは「getch で 1 文字読む直前に refresh を読んで画面を更新する」と思っていればよいです。

- move(Y, X) — 上から Y 行目の X 文字目にカーソルを移動する。
- addch(C)、addstr(S) — カーソル位置に 1 文字または文字列を表示し、カーソルは表示したぶんだけ進める。
- insch(C) — カーソル位置に文字を挿入し、以後の文字を 1 文字右にずらす。カーソル位置は変化しない。
- delch() — カーソル位置の文字をし、右側の文字を文字左に詰める。
- clear()、cleartoeol() — 画面全体をクリア、カーソル位置から右側をクリア。
- refresh() — ここまでに呼んだ機能を実際に適用して画面を更新する (これを呼ぶまでは内部で保留している)。
- getch() — キーボードから 1 文字入力して文字を返す。

コンパイル時のオプションとして「-lncurses」が必要です。

```
% gcc8 ncursesdemo.c -lncurses
% ./a.out
```

画面制御を行うプログラムは例示がしにくいですが、起動するととりあえず画面の 10 行目の 10 文字目以降に次のようにメッセージが表示されます。

```
press any key_
```

ここで何かキーを打つと追加の文字列が表示されます。

```
press any keyab_
```

再度キーを打つと今度は行の途中に挿入がなされます。

```
pressba any keyab
```

再度キーを打つとさっきのカーソル位置の文字と 20 文字目以降が消えます。

```
pressa any_
```

演習 3 ncurses を使って何か「画面上でお絵描きする」プログラムを作ってみなさい。

8.3.2　1 行ウィンドウの画面エディタ

ではなるべく簡単な例として、「ウィンドウサイズが 1 行だけの (カーソルのある行だけが見えている)」画面エディタを作ります。ファイルの冒頭部分を示します (あと `readfile`、`writefile` も必要。

```
// sedit.c --- very primitive screen editor w/ ncurses.
#include <stdio.h>
#include <stdbool.h>
#include <ncurses.h>
#include <stdlib.h>
#include <string.h>
#include "ebuf.h"
// readfile, writefile here
```

次に、画面エディタでは行中の任意の位置 `p` で文字を挿入したり消したりするので、その処理を行う下請け関数を用意します。挿入のときは行の最後から前に向かって 1 文字ずつずらして場所をあけます。いずれも文字列の長さは渡すこととします。

```
void inschar(char *s, char ch, int p, int len) {
  int i;
  for(i = len + 1; i > p; --i) { s[i] = s[i-1]; }
  s[p] = ch;
}
void delchar(char *s, int p, int len) {
  int i;
  for(i = p; i < len; ++i) { s[i] = s[i+1]; }
}
```

これらを呼び出しながら、また ncurses の関数を呼び出しながら、文字の挿入や削除およびカーソル移動を行う関数 `handleline` を示します。自分が取り扱わないコマンド (1 文字ですが) が来たときは `false` を返し、取り扱えた場合は `true` を返します。また、この関数の中でカーソル位置や文字列の長さを変化させるので、この 2 つについては整数へのポインタを渡してもらい、間接参照を使ってもとの (呼び出し側の) 変数を変更します。

```
bool handleline(int c, char *s, int *pos, int *len) {
  if(c >= ' ' && c <= '~') {
    if(*len >= MAXSTR-1) { return false; }
    insch(c); inschar(s, c, *pos, (*len)++); move(10, ++(*pos));
  } else if(c == 'B'-'@') {
    if(*pos > 0) { move(10, --(*pos)); }
  } else if(c == 'F'-'@') {
    if(*pos < *len) { move(10, ++(*pos)); }
  } else if(c == 'H'-'@') {
    if(*pos <= 0) { return false; }
    move(10, *pos - 1); delch(); delchar(s, --(*pos), (*len)--);
  } else {
    return false;
  }
  return true;
}
```

まず文字がスペースからチルダまでの範囲であれば普通の文字なので、その文字を挿入しますが、その前にもし文字列の長さが MAXSTR-1 以上ならこれ以上増やせないのでだめですといって帰ります。OK の場合はカーソル位置に文字を挿入し、また文字列の方もその位置に文字を挿入し、長さを 1 増やします。カーソル位置は 1 進めます (insch はカーソル位置を変更しないので move で変更する必要があります)。

その先、コマンドが^F や^B の場合は (コントロール文字の文字コードは「その文字の大文字のコードから@のコードを引いた値」になっています)、カーソル位置を増減しますが、0 より手前や行長より先には行けないようにしています。

BS 文字 (文字コードでは^H と同じ) の場合、行頭でなければカーソル位置の 1 つ手前の文字を消し、長さも 1 減らします。これら以外の文字は false を返します。

では main を見てみましょう。まず ebuf を作り、readfile でファイルを読み込みますが、そのファイル名はコマンド引数で渡したファイルということにしました。そして先頭行に行き、各種変数を用意しますが、変数 show は現在の 1 行を表示し直すかどうかを制御するフラグです (最初および行が別の行に変化した時に「true」にします)。次に ncurses の初期設定を行い、無限ループに入ります。ループの冒頭で show が「はい」なら文字列、長さ、カーソル位置を変数にいれ、画面に表示し、カーソル位置は行の先頭にし、show は false に変更します。その後、refresh を呼んでからキーボード入力を待ちます。

```
int main(int argc, char *argv[]) {
  ebufp e = ebuf_new();
  if(!readfile(e, argv[1])) { printf("?\n"); return 1; }
  ebuf_top(e);
  int len, pos, ch; char *str;
  bool show = true;
  initscr(); noecho(); cbreak(); system("stty raw"); clear();
  while(true) {
    if(show) {
      str = ebuf_str(e); len = strlen(str); pos = 0;
      move(10, 0); addstr(str); clrtoeol(); move(10, 0); show = false;
    }
    refresh(); ch = getch();
```

```
      if(handleline(ch, str, &pos, &len)) {
        // do nothing
      } else if(ch == 'J'-'@') {
        ebuf_forward(e); ebuf_insert(e, "");
        ebuf_backward(e); show = true;
      } else if(ch == 'N'-'@') {
        ebuf_forward(e); show = true;
      } else if(ch == 'P'-'@') {
        ebuf_backward(e); show = true;
      } else if(ch == 'Z'-'@') {
        break;
      }
    }
    endwin(); writefile(e, argv[1]); return 0;
  }
```

キー入力があったら、まず handleline で行内編集コマンドとしての処理を試み、OK ならそれで終わりで次の周回へ行きます。OK でないなら、他のコマンドですが、ここでは改行 (コードは^J — 1 行下へ行き、直前に空行を挿入し、そこへ行く)、上下の行への移動 (^N と^P) があります。これらでは行が変わるので show を「はい」にする必要があります。そして終了は^Z で、このときは curses を後始末して ebuf の内容を元のファイルに書き戻して終わります。実際に動かしてみると、1 行しか表示されないわりには、それなりに編集できる感じです。試してみてください。

演習 4 1 行ウィンドウの画面エディタを動かしてみなさい。適当なプログラムのファイルを編集してみること。動いたら、次のような改良をしてみなさい。

a. 1 行ウィンドウはつらいので、上の方や下の方も一定行数 (たとえば 5 行とか) 表示されるようにする。現在位置は 10 行目で固定でよいです。

b. emacs のようにカーソル位置が画面内で自由に動かせ、画面から外に出そうになったらスクロールするようにする。

c. ^J のとき空行を挿入するのでなく、カーソル位置で現在の行を 2 つに分けるようにする。

d. 行の先頭で^H のときに無視するのでなく、前の行と今の行がくっつくようにする。カーソル位置もくっついた位置になっているとなおよい。

e. ^F や^B で行末や行頭を超えようとしたときに無視するのでなく次の行や前の行に移るようにする。(ヒント: 現在は handleline でこれらの文字は常に「はい」が返るようになっていので、これ以上行けないときは「いいえ」を返すように修正する必要があります。)

f. 演習 1 や演習 2 で扱ったような機能を使えるようにする。

g. ファイルの読み書きをコマンド引数で指定したファイル固定でなく、画面からファイル名を入力して行えるようにする。

h. その他、あったらいいと思う機能を追加する。

なお、課題のいくつかのためには画面の幅や高さを知りたいですね? その場合「initscr()」の後で「getmaxyx(stdscr, height, width);」を呼びます。ここで height や width は別の名前でもよく、とにかく整数変数です。[1]

[1] なぜ「&」をつけなくても値が変化させられるか疑問に思うでしょうけれど、これはマクロ機能を使っているためです…が、マクロについては後の回で余裕があったら取り上げますのでそれまで保留で。

本日の課題 8A

「演習 1」〜「演習 6」で動かしたプログラム 1 つを含むレポートを本日中 (授業日の 23:59 まで) に提出してください。

1. sol または CED 環境で「`/home3/staff/ka002689/prog19upload 8a` ファイル名」で以下の内容を提出。
2. 学籍番号、氏名、ペアの学籍番号 (または「個人作業」)、提出日時。名前の行は先頭に「@@@」を付けることを勧める。
3. プログラムどれか 1 つのソースと「簡単な」説明。
4. レビュー課題。提出プログラムに対する他人 (ペア以外) からの簡単な (ただしプログラムの内容に関する) コメント。
5. 以下のアンケートの回答。

   Q1. 双連結リストの概念を理解しましたか。
   Q2. 行エディタのプログラムを理解し直せるようになりましたか。
   Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

次回までの課題 8B

「演習 1」〜「演習 6」(ただし 8A で提出したものは除く) の (小) 課題から選択して 2 つ以上プログラムを作り、レポートを提出しなさい。できるだけ複数の演習から選ぶこと。レポートは次回授業前日 23:69 を期限として提出すること。

1. sol または CED 環境で「`/home3/staff/ka002689/prog19upload 8b` ファイル名」で以下の内容を提出。
2. 学籍番号、氏名、ペアの学籍番号 (または「個人作業」)、提出日時。名前の行は先頭に「@@@」を付けることを勧める。
3. 1 つ目の課題の再掲 (どの課題をやったか分かればよい)、プログラムのソースと「丁寧な」説明、および考察 (課題をやってみて分かったこと、分析、疑問点など)。
4. 2 つ目の課題についても同様。
5. 以下のアンケートの回答。

   Q1. 双連結リストの操作が自由にできるようになりましたか。
   Q2. 画面エディタについてどのように理解しましたか。
   Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

＃9　抽象データ型

今回は次のことが目標となります。

- 抽象データ型の考え方を理解する
- 機能は同一でも時間計算量が異なる場合があることを理解する

9.1　抽象データ型

9.1.1　抽象データ型とその必要性

ここまで散々使ってきた「型」ないし「データ型」とは、データの種類を表す言葉でした。たとえば整数型であれば 32 ビットの符号つき 2 進表現で、構造体型であれば構造体定義に書かれたフィールドの集まり、という具合です。そしてその構造と機能 (整数なら整数演算ができ、構造体なら持っているフィールドの値が読み書きできる) とは一体でした。

これに対し、**抽象データ型** (abstract data type, ADT) とは、「機能は定めるが、内部の構造や機能の実装は隠されている (隠蔽されている)」ような型を言います。

抽象化 (abstraction) とは「不必要な細部を見ないで重要なことだけを考える」という意味ですが、プログラムでは機能が動作しさえすれば役に立つので、まさに機能が「重要なこと」であり、中がどのようにできているかは「見なくてよい細部」なのです。

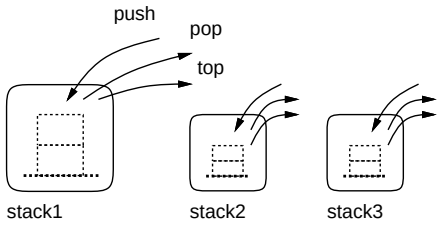


図 9.1: 抽象データ型としてのスタック

実はこれまでに、抽象データ型の考え方は繰り返し使っています。具体的には、構造体のポインタ型を使った構造の隠蔽を使っているときがそうです (等差数列、スタック、キュー、エディタバッファなど多数ありました)。これらでは、ヘッダファイルを取り込んで使う側は機能 (関数呼び出し) だけしか使わず、内部がどうなっているかは一切分かりません。それでいながら、複数の等差数列やスタックを自由に作れるので、確かに「データの種類」つまり「型」なわけです (図 9.1)。

それでは、抽象データ型の何がよいのでしょう? 実は、プログラムは大きさが 2 倍になると、作る手間は 2 倍よりずっと大きくなります。たとえば、A というプログラムと B というプログラムを組み合わせた (サイズ A+B の) プログラムの場合、A を作る手間と B を作る手間だけでは済みません。A から B の一部を使ったり、B から A の一部を使ったりする「組み合せの」手間があるからです。ですから、プログラムが大きくなると、開発の手間は指数関数的に増大してしまいます。

それを避けるには、A の部分と B の部分を「できるだけ独立にして」「組合せの部分をなくす」ことです。そうすれば、プログラムの大きさに対してほぼ比例の手間で済むようになります。その最たるものが抽象データ型です。なにしろ、B は A(抽象データ型の部分) に対して「機能を呼び出す」ことしかできず、中のデータがどうであるかはそもそも分からないからです (A は最初から使われるだけの側で、B の機能を呼んだりしません)。

そしてこの独立性のおかげで、A の中身 (実装) はプログラムの他の部分に影響されることなく、自由に変更できます (スタックも「配列を使った実装」「単連結リストを使った実装」の 2 種類あることを見てきました)。これは、プログラムの改良などに威力を発揮します。

9.1.2　例題: 整数の集合

今回は抽象データ型の例として「0 以上の整数の集合」を扱います。ヘッダファイルを示しましょう。

```
// iset.h --- set of non-negative integers.
#include <stdbool.h>
struct iset;
typedef struct iset *isetp;
isetp iset_new();                  // create empty set
void iset_free(isetp s);           // free memory
bool iset_isempty(isetp s);        // test emptiness
bool iset_isin(isetp s, int e);    // e included in s?
int iset_max(isetp s);             // return max value
void iset_addelt(isetp s, int e); // add e to s
void iset_subelt(isetp s, int e); // remove e from s
isetp iset_union(isetp s, isetp q); // set union
```

今回は領域を大量に使うので、使い終わったら解放するようにします。ということで操作としては、集合を作る、解放する、空集合かどうか調べる、整数を指定して含まれるかどうかを調べる、最大値 (空集合のときは 0 とします)、整数を追加する、取り除く、そして 2 つの集合から和集合を作る、まで用意しました。

ではデモとして、2 つ集合を読み込んで和集合を作って表示する、というのをやってみましょう。読み込みと出力を関数として作っているので、その分量が少し多くなります。読み込む時はまず空の終業を用意し、負の値がくるまで次々に整数を読み込んで集合に追加していき、負の値が来たらそこでやめて集合を返します。打ち出す時はまず最大を調べ、0 から最大まで順に調べて集合に入っている値だったら出力します。

```
// isetdemo1.c --- iset demonstration.
#include <stdio.h>
#include <stdbool.h>
#include "iset.h"

isetp readiset(void) {
  isetp s = iset_new();
  while(true) {
    int i; printf("i> "); scanf("%d", &i);
    if(i < 0) { return s; } else { iset_addelt(s, i); }
  }
}
void printiset(isetp s) {
  printf("{");
  for(int max = iset_max(s), i = 0; i <= max; ++i) {
    if(iset_isin(s, i)) { printf(" %d", i); }
  }
```

```
  printf(" }\n");
}
int main(void) {
  isetp s = readiset(); printiset(s);
  isetp t = readiset(); printiset(t);
  isetp u = iset_union(s, t); printiset(u);
  iset_free(s); iset_free(t); iset_free(u);
  return 0;
}
```

main 本体では、2 つ読み込みそれぞれ表示したあと、和集合を生成してそれも印刷する、というだけです。最後に領域を解放しています。実行例を示しましょう。

```
% gcc8 isetdemo.c iset_1.c
% ./a.out
i> 1
i> 3
i> 5
i> -1
{ 1 3 5 }
i> 2
i> 3
i> 4
i> -1
{ 2 3 4 }
{ 1 2 3 4 5 }
```

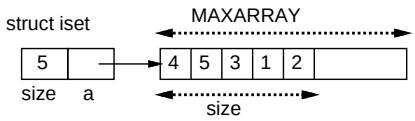


図 9.2: 整数の集合の実装

実装のファイルを iset_1.c としているのは、これからいくつも別の実装を作るからです。とりあえず、最初の実装を見てみましょう。この実装は次の方針で作ってあります (図 9.2)。

- 固定サイズの配列をレコードと別に持ち、そこに集合に含まれる整数を保持する。
- 整数の格納順は任意とする。

2 番目についてですが、新しい整数は末尾に追加していきますが、iset_subelt で要素を取り除いた時はその箇所に末尾にあった値を移してきて埋めるので、常に入れた順になっているとは限りません。ソースを示します。

```
// iset_1.c --- iset impl w/ unsorted array of fixed size.
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "iset.h"
#define MAXARRAY 10000
```

```
struct iset { int size, *a; };

isetp iset_new() {
  isetp s = (isetp)malloc(sizeof(struct iset));
  s->size = 0; s->a = (int*)malloc(sizeof(int)*MAXARRAY);
  return s;
}
void iset_free(isetp s) { free(s->a); free(s); }
static int isin1(isetp s, int e) {
  for(int i = 0; i < s->size; ++i) {
    if(s->a[i] == e) { return i; }
  }
  return -1;
}
bool iset_isempty(isetp s) { return s->size == 0; }
bool iset_isin(isetp s, int e) { return isin1(s, e) >= 0; }
static int max2(int a, int b) { return (a > b) ? a : b; }
int iset_max(isetp s) {
  int max = 0;
  for(int i = 0; i < s->size; ++i) { max = max2(max, s->a[i]); }
  return max;
}
void iset_addelt(isetp s, int e) {
  if(iset_isin(s, e) || s->size >= MAXARRAY-1) { return; }
  s->a[(s->size)++] = e;
}
void iset_subelt(isetp s, int e) {
  int i = isin1(s, e); if(i < 0) { return; }
  s->a[i] = s->a[--(s->size)];
}
isetp iset_union(isetp s, isetp t) {
  isetp u = iset_new();
  for(int i = 0; i < s->size; ++i) { iset_addelt(u, s->a[i]); }
  for(int i = 0; i < t->size; ++i) {
    if(!iset_isin(s, t->a[i])) { iset_addelt(u, t->a[i]); }
  }
  return u;
}
```

static 指定の関数はファイルの外からは参照できず、ファイル内での下請け専用です。

では単体テストについて検討しましょう。個別の整数を取り出したりしてチェックするのだと使いづらそうなので、集合全体をまとめてチェックする expect_iset を作ることにします。パラメタは「集合、テストする値の最大値、含まれているべき要素の個数、含まれている各要素を昇順に並べた配列、メッセージ」となっています。含まれているべき要素を配列 a で受け取り、その中の次に見るべき位置を変数 p で覚え、i は 0〜m で変化させて行きますが、i が a[p] と等しいときは「含まれるべき値」、それ以外は「含まれるべきでない値」となります (等しいのが見つかったら p は 1 つ先に進める)。

```
void expect_iset(isetp s, int m, int n, int a[], char *msg) {
  bool ok = true; int p = 0;
  for(int i = 0; i <= n; ++i) {
    if(p < n && i == a[p]) {
      ++p;
      if(!iset_isin(s, i)) { printf(" NG: %d not in set.\n", i); ok = false; }
    } else {
      if(iset_isin(s, i)) { printf(" NG: %d in set.\n", i); ok = false; }
    }
  }
  printf("%s %s\n", ok?"OK":"NG", msg);
}
```

これを使った単体テストの例です。a はテストデータを入れた配列ですが、expect_iset に渡す時にどこを先頭にするか、何個を指定するかを変えることで 3 通りに使っています。

```
// test_iset.c --- unit test for iset.
#include <stdbool.h>
#include <stdio.h>
#include "iset.h"
(expect_iset here)
int main(void) {
  int a[] = { 1, 3, 5, 7 };
  isetp s = iset_new();
  iset_addelt(s, 1); iset_addelt(s, 3); iset_addelt(s, 5);
  expect_iset(s, 9, 3, a, "initial: { 1 3 5 }"); iset_subelt(s, 1);
  expect_iset(s, 9, 2, a+1, "- { 1 }: { 3 5 }");
  isetp q = iset_new(); iset_addelt(s, 7); iset_addelt(s, 5);
  isetp r = iset_union(s, q);
  expect_iset(r, 9, 3, a+1, "+ { 7 5 }: { 3 5 7 }");
  iset_free(s); iset_free(q); iset_free(r);
  return 0;
}
```

動かしたようすは次の通り。

```
% gcc8 test_iset.c iset_1.c
% ./a.out
OK initial: { 1 3 5 }
OK - { 1 }: { 3 5}
OK + { 7 5 }: { 3 5 7 }
```

演習 1 上の例題をそのまま動かせ。打ち込む値は変えて試してみること。OK だったら、次の機能を追加してみなさい。単体テストを作成し実行すること。

a. 積集合 (intersection) の演算を作れ。
b. 差集合 (difference) の演算を作れ。
c. 排他的和集合 (exclusive or、どちらか片方のみにある値だけを集めた集合) の演算を作れ。
d. その他あるとよいと思う機能を追加してみよ。

演習 2 上で見た実装は、配列のサイズが MAXARRAY に固定になっていて、少しの値しか扱わなければ領域が無駄だし、多くの値を扱おうとすると入り切らなくなった分が無視される。これは不便なので、次の考え方に基づいて実装を修正してみよ。単体テストを作成し実行すること。

1. 構造体のフィールドに配列の現在のサイズ limit を追加する。
2. 最初は小さめの (たとえば大きさ 10) 配列から始めて、大きさを増やそうとして入り切らなくなったら (size が limit より大きくなりそうになったら)、新しい配列を limiit+1 の大きさで割り当て、内容をコピーしてからこちらの配列を構造体に入れる (前の配列は解放する)。

演習 2 のように、情報隠蔽がなされていることで、内部の構造を修正しても外からの使い方は一切変化しないで済みます。これが抽象データ型の利点ということです。

9.2　データ構造の選択と計算量の関係

9.2.1　疑似乱数の生成

これからデータ構造を変化させて計算量を検討しつつ時間計測を行いますが、その準備として疑似乱数の扱いから見ていきましょう。C 言語の標準ライブラリでは疑似乱数の生成には rand を使います。この関数は stdlib.h で宣言されていて、引数は 0 個で、0 から RAND_MAX(これも同じヘッダファイルで定義されています) までの範囲の整数を返します。

ただし、乱数の「種」で初期化しないと毎回同じ列が返ってくるので、通常は最初に「srad(time(NULL))」を呼び出して種を設定します。srand は種を設定する関数、time(宣言は time.h) は 1970.1.1 0:00 からの経過秒数を返します (引数として領域の番地を渡すとそこにも値を入れますが、NULL を渡した場合は値のみ返します)。秒数は絶えず変わって行くので、乱数の種に適切だというわけです。

rand の結果を、ある範囲の整数乱数として使いたければ適当な値で剰余を取ります。また $[0,1)$ の実数にするには RAND_MAX で実数除算します。サンプルを見てみましょう。

```
// randdemo.c --- demonstration of rand.
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char *argv[]) {
  int i, n = atoi(argv[1]);
  srand(time(NULL));
  for(i = 0; i < n; ++i) { printf(" %d", rand() % 1000); }
  printf("\n");
  for(i = 0; i < n; ++i) { printf(" %.3f", rand()/(double)RAND_MAX); }
  printf("\n");
  return 0;
}
```

まったく上で説明した通りで、まず種を設定し、指定された個数の整数乱数 (0〜999) と、実数の乱数 (区間 $[0,1)$) を出力しています。動かしたようすは次の通り。

```
% ./a.out 10
 774 837 508 259 653 449 448 585 551 172
 0.289 0.868 0.881 0.543 0.394 0.571 0.068 0.873 0.531 0.970
```

演習 3 疑似乱数を使った次のようなプログラムを作れ。

a. サイコロ (実際には 0〜5 の整数乱数に 1 を足せばよい) を 6000 回振ってそれぞれの目が何回出たかを表示する実験プログラム。
b. 100 ますのすごろく (後戻り等なし、ゴールから行きすぎても OK) でサイコロを何回振ってゴールできるかを、1000 回やってみて平均回数を求める。
c. X 座標 $[0,1)$、Y 座標 $[0,1)$ の範囲にランダムに点を打ち、中心 0、半径 1 の円内に入った点の比率を求めて 4 倍することで π を求めてみる。何個の点を打つかは実行時に指定する。
d. 0.6 の確率で表が出る偏ったコインで、「10 回連続して表になる」までに何回トスすればできるかを、100 回やってみて平均回数を求める。
e. その他、疑似乱数を使った好きなプログラム。

9.2.2 所要時間の計測

次に、所要時間の計測について知っておきましょう。C ライブラリの中で時間を返すものとして、前述の `time` がありますが、秒単位だと時間計測には荒すぎて不向きです。

ここでは `clock_gettime` というものを使います。その呼び出しの第 1 パラメタは取得する時間の種類で、今回のような計測では `CLOCK_REALTIME`(実時間) または `CLOCK_VIRTUAL`(CPU 消費時間) を指定します。そして第 2 パラメタには構造体 `struct timespec` (上記の呼び出しや定数とともに `time.h` で宣言) のアドレスを渡し、そこに計測値が入って戻って来ます。この構造体の内容は次のようになっています。

```
struct timespec {
  time_t  tv_sec;  /* seconds */
  long    tv_nsec; /* and nanoseconds */
};
```

`time_t` は秒数を入れるのに適切な整数型を `typedef` したものなので、要するにどちらも適当な長さの整数で、秒以上の部分と秒未満の部分 (ナノ秒単位) を分けて扱っています。

では、3^n を遅い再帰を使って計算し、その時間を測ってみましょう。`pow3n` がその遅い再帰の関数で、次の方法で計算をしています。

$$pow3n(n) = \begin{cases} 1 & (n = 0) \\ pow3n(n-1) + pow3n(n-1) + pow3n(n-1) & (otherwise) \end{cases}$$

1 つ少ない n に進むごとに 3 回ずつ自分を呼ぶので、時間計算量は $O(3^n)$ になるはずです (そもそも 1 ずつ足して 3^n になるのだからその意味でも $O(3^n)$ です)。

```
// timedemo.c --- demonstration of mesuring time.
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "iset.h"

int pow3n(int n) {
  if(n < 1) { return 1; }
  else { return pow3n(n-1)+pow3n(n-1)+pow3n(n-1); }
}
int main(int argc, char *argv[]) {
  int i, n = atoi(argv[1]);
  struct timespec tm1, tm2;
  clock_gettime(CLOCK_REALTIME, &tm1);
```

```
  int v = pow3n(n);
  clock_gettime(CLOCK_REALTIME, &tm2);
  double dt = (tm2.tv_sec-tm1.tv_sec) + 1e-9*(tm2.tv_nsec-tm1.tv_nsec);
  printf("pow3n(%d) = %d; elapsed-time = %.4f\n", n, v, dt);
  return 0;
}
```

main の方では、2 回 clock_gettime を呼んで時刻を取得し、その間で pow3n を呼びます (この部分が時間計測の対象)。終わったら、秒単位の実数に直したいので、秒の差はそのまま、ナノ秒の差は 10^{-9} を掛けて加えたものを dt とします。最後にこれらを表示しています。では動かしてみましょう。

```
% ./a.out 4
pow3n(4) = 81; elapsed-time = 0.0000
% ./a.out 10
pow3n(10) = 59049; elapsed-time = 0.0002
% ./a.out 15
pow3n(15) = 14348907; elapsed-time = 0.0560
% ./a.out 16
pow3n(16) = 43046721; elapsed-time = 0.1745
% ./a.out 17
pow3n(17) = 129140163; elapsed-time = 0.5069
```

3^4 とか 3^{10} では速すぎですが、3^{15} で 0.056 秒となり、n をもう 1 つ増やすと 3 倍、さらにもう 1 つ増やすと 3 倍で、確かに $O(3^n)$ となっているようです (注意: 個人使用のマシンであればこのように取得する時刻の種別は CLOCK_REALTIME でよいが、共用マシンだと他人も CPU を使っているので CLOCK_VIRTUAL を使う方がよいかも)。

演習 4 n を指定して動かしたときに、例題では時間計算量が $O(3^n)$ だったが、(a) $O(2^n)$、(b) $O(n^3)$、(c) $O(n^2)$、(d) $O(n)$ などとなる関数を作成して同様に計測し、実際にそのような時間になることを確認してみなさい (どれか 1 つ以上でよい)。

9.2.3 2 分探索

整数の集合に話を戻しますが、先に示した実装では要素の整数は配列内に「おおむね入れた順で」並んでいました (ランダム順と言っていいでしょう)。この方法だと、ある指定した整数 e が含まれているかを調べるのに、先頭から順に調べて行く必要があるため、集合の要素数が n のとき、iset_isin(e) の時間計算量が $O(n)$ になります。

別の方法として、要素を整列して昇順に並べておくとしましょう。そのときは 2 分探索が使えます。次のコードを見てください関数 bsearch は、配列 a の添字 i〜j の範囲で、値 e があればその添字、無ければ-1 を返します。配列 a は昇順に整列されているものとします。

```
int bsearch(int *a, int e, int i, int j) {
  if(i > j) { return -1; }
  int k = (i + j) / 2;
  if(a[k] == e) {
    return k;
  } else if(a[k] > e) {
    return bsearch(a, e, i, k-1);
  } else {
```

```
      return bsearch(a, e, k+1, j);
    }
  }
```

まず、i が j より大きければ範囲が空なので、-1 を返します。そうでないときは、i と j の中間の位置 k を計算します。a[k] が e に等しければ、ちょうど見付かったので、k を返します。そうでない場合は、a[k] と e の大小関係に応じて、e がある範囲が i〜k-1 か k+1〜j のどちらかになりますから (昇順に整列されているため)、それに応じて自分自身を再帰呼び出しします。

このアルゴリズムは「繰り返し半分に分けて扱う」ことから分割統治アルゴリズムに分類できます。また、コードを見ると再帰は末尾再帰のみなので、簡単に再帰を除去してループのみのコードに変換できます。それでは、このアルゴリズムの時間計算量はどうでしょうか。長さ n の配列に対して、半分ずつにしていって長さ 0 になれば終わります。ということは、半分にする回数は $\log_2 n$ なので、時間計算量は $O(\log n)$ です。

ただしもちろん、常に配列を整列された状態に保つためのコストが必要です。前の実装では、集合にまだ無い要素を追加するのは、最後に入れるだけだったので $O(1)$ です。しかし今度は、追加する値がちょうどいい位置になるように、後ろの方を 1 つずつずらす必要があります。平均して n 個の要素のうち半分ずらすだろうと期待できますから、$O(n)$ になりますね。また、要素を削除する場合も、前の実装では空いた場所に最後の要素を移して来るだけでしたので $O(1)$ でしたが、整列を維持する場合は空いた要素の後ろを前に詰めることになり、やはり $O(n)$ になります。

ただし! 上の議論は「要素の位置が分かった後」の話ですよね。そもそもランダム版では位置を探すのに $O(n)$ 掛かるので、その後が $O(1)$ でも結局全体としては $O(n)$ になります。これらを整理して表 9.1 に示しました (3 番目の方法は後述)。整列してあれば最大値はそもそも端っこの値を取るだけなので $O(1)$ になります。このように、機能によって得意不得意がかなり異なることが分かります。

表 9.1: 整数集合のランダム順版と昇順版の操作の比較

操作	ランダム	昇順	ビットマップ
iset_isin	$O(n)$	$O(\log n)$	$O(1)$
iset_addelt	$O(n)$	$O(n)$	$O(1)$
iset_subelt	$O(n)$	$O(n)$	$O(1)$
iset_max	$O(n)$	$O(1)$	$O(1)$
iset_union	$O(n^2)$	$O(n)$	$O(1)$

演習 5 2 分探索を大きな配列に対して実行して時間計測し、大きさ n に対する時間計算量が $O(\log n)$ になっているか確認しなさい。線形探索 (端から順に探す) と比較するとなおよい。(ヒント: 1 回の探索はあっという間に終わってしまうので、同じ探索を 100 回とか 1000 回ループ実行してその時間を計測するのがよい。)

演習 6 整数の集合を要素が昇順に並んでいて 2 分探索を使用できるように修正してみなさい。単体テストを作成し実行すること。

9.2.4 整数集合の文字配列表現

実は、時間計算量がもっと小さく簡単な方法があります。それは…配列 a を用意し、整数 i が集合に含まれていれば a[i] を 1、そうでなければ 0 にしておく、という方法です。簡単でしょう? しかも、配列のどこかをアクセスする時間は一定なので、基本的な操作は全部 $O(1)$(定数時間) です (これは次に出て来るビットマップでも同様)。

ただ、この方法だとメモリが沢山必要になります。できるだけ節約するとして、1 要素のビット数が一番小さいデータ型は 1 要素が 1 バイト (8 ビット) の文字型ですから、char の配列を使うとして

も、0/1 を入れるだけなら 1 ビットで必要なところ、7 ビットの無駄な領域がついて来ます。それでも、最大の値があまり大きくないなら、十分役に立ちます。

演習 7 上記の考え方に基づく整数集合の実装を作れ。最大整数は固定でもいいが、集合に入れられる最大値に応じて大きさが変化できると使いやすい。単体テストを作成し実行すること。

9.2.5　整数集合のビットマップ表現

上で出て来たように、0/1 を使って集合に入っているかどうかを表すのなら、ビットをそのまま使うのが自然な方法だと言えます。整数型でビット数が多いのは `unsigned long` なので (64 ビット。符号つきの `long` でも同じなのですが、マイナスが出てこない方がよさそうです)、これを使って 0〜63 の整数集合を表現できます。この方法を、ビットの有無の「地図」で表す、という意味で集合の**ビットマップ** (bitmap) 表現と言います。

64 ビット中の `e` ビット目を調べるにはどうしたらいいのでしょうか。それには「`1L << e`」という式を使います。これは、64 ビットの一番下のビットが 1 である語 (整数定数の後に `L` がついていると `long` として扱います) を、左シフト演算`<<`によって `e` ビット左にずらすと、右から `e` ビット目が 1 の語ができます。それと集合を表す 64 ビットとを`&`演算 (ビット毎 AND) すると、集合の右から `e` ビット目が 0 であれば全体として 0 になり (図 9.3 上左)、0 でなければそのビットだけが立った (0 ではない) 語になりますから (図 9.3 上右)、それで判別できます (図では見やすさのため 8 ビットにしています)。

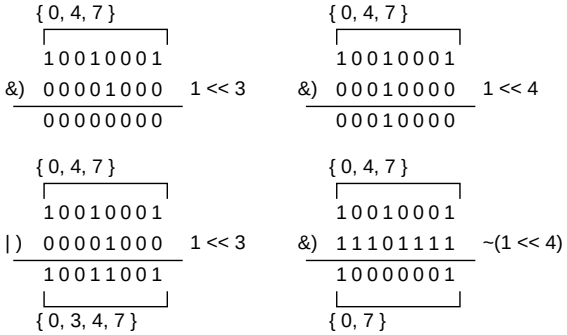


図 9.3: ビットマップによる整数集合の操作

では、要素を追加したり取り除くのはどうでしょうか。追加は簡単で、`|`演算 (ビット毎 OR) を使えばそのビット位置を「1」にしたビット列が作れます (図 9.3 下左)。取り除くのはちょっとややこしいのですが、シフトの後で`~`(ビット毎反転) 演算を使うと、そのビット位置だけが 0、残りが 1 のビット列になりますから、それと集合のビット列の`&`を取れば当該ビットを「0」にしたビット列が作れます (図 9.3 下右)。あと、図にはありませんが和集合や積集合も`|`や`&`で作れます (排他的論理和演算は C 言語では`^`です)。

それでは、この方法を使った整数集合の実装を見てみましょう。扱える値の最大は上述のように 63 ですが、とくにチェックはしていません。あと、`&=`とか`|=`とか見慣れない演算が出てきますが、これらはたとえば「`x &= 1`」であれば「`x = x & 1`」と同じ意味になります。

```
// iset_4.c --- iset impl w/ unsiged long bitmap.
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "iset.h"
#define MAXARRAY 10
struct iset { unsigned long bits; };
```

```
isetp iset_new() {
  isetp s = (isetp)malloc(sizeof(struct iset));
  s->bits = 0L; return s;
}
void iset_free(isetp s) { free(s); }
bool iset_isempty(isetp s) { return s->bits == 0L; }
bool iset_isin(isetp s, int e) { return (s->bits & 1L<<e) != 0; }
int iset_max(isetp s) {
  int i;
  printf("%lx\n", s->bits);
  for(i = 63; i > 0; --i) {
    if(iset_isin(s, i)) { return i; }
  }
  return 0;
}
void iset_addelt(isetp s, int e) { s->bits |= 1L<<e; }
void iset_subelt(isetp s, int e) { s->bits &= ~(1L << e); }
isetp iset_union(isetp s, isetp t) {
  isetp u = iset_new(); u->bits = s->bits | t->bits;
  return u;
}
```

演習 8 ビットマップによる整数集合の実装を最初に出て来た `isetdemo1.c` と組み合わせて動かし、同じに動作することを確認しなさい。OK なら、次の機能を追加してみなさい。単体テストを作成し実行すること。

a. 積集合 (intersection) の演算を作れ。
b. 差集合 (difference) の演算を作れ。
c. 排他的和集合 (exclusive or) の演算を作れ。
d. その他あるとよいと思う機能を追加してみよ。

演習 9 `unsined long` が 1 語では 63 までしか扱えないが、`unsigned long` の配列を使うことでもっと多くの値まで扱える。この場合、値 `e` は配列の `e / 64` 要素目の `e % 64` ビット目の 0/1 で表されることになる。この考え方で整数集合を実装してみなさい。最大値は固定でもよいが、必要に応じて自動的に増えるとなおよい。単体テストを作成し実行すること。

本日の課題 9A

「演習 1」〜「演習 9」で動かしたプログラム 1 つを含むレポートを本日中 (授業日の 23:59 まで) に久野までに提出してください。

1. sol または CED 環境で「`/home3/staff/ka002689/prog19upload 9a ファイル名`」で以下の内容を提出。
2. 学籍番号、氏名、ペアの学籍番号 (または「個人作業」)、提出日時。名前の行は先頭に「@@@」を付けることを勧める。
3. プログラムどれか 1 つのソースと「簡単な」説明。
4. レビュー課題。提出プログラムに対する他人 (ペア以外) からの簡単な (ただしプログラムの内容に関する) コメント。

5. 以下のアンケートの回答。

   Q1. 抽象データ型の概念と整数集合の実装方法を最低 1 つ理解しましたか。
   Q2. 乱数の生成方法と時間の計測方法が分かりましたか。
   Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

次回までの課題 9B

「演習 1」〜「演習 9」(ただし 9A で提出したものは除外、以後も同様) の (小) 課題から選択して 2 つ以上プログラムを作り、レポートを提出しなさい。できるだけ複数の演習から選ぶこと。レポートは次回授業前日 23:69 を期限とします。

1. sol または CED 環境で「`/home3/staff/ka002689/prog19upload 9b` ファイル名」で以下の内容を提出。
2. 学籍番号、氏名、ペアの学籍番号 (または「個人作業」)、提出日時。名前の行は先頭に「@@@」を付けることを勧める。
3. 1 つ目の課題の再掲 (どの課題をやったか分かればよい)、プログラムのソースと「丁寧な」説明、および考察 (課題をやってみて分かったこと、分析、疑問点など)。
4. 2 つ目の課題についても同様。
5. 以下のアンケートの回答。

   Q1. ビットマップを用いた整数集合の実現方法を理解しましたか。
   Q2. 必要に応じて配列サイズを拡張できるようになりましたか。
   Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

＃10　データの読み書きと整列

今回は次のことが目標となります。

- CSV ファイルの読み書きと取り扱いを経験する
- 基本的な (平易な) 整列アルゴリズムについて理解する

10.1　データの読み書き

10.1.1　CSV ファイルの形式

ここまで C 言語で様々なアルゴリズムと取り扱って来ましたが、そのアルゴリズムで取り扱うデータは「ちょっとしたテスト用のデータ」程度でした。しかし実際にはもちろん、様々なところから持って来た実験データや調査データをプログラムで取り扱いたいわけです。そのためには、ファイルからデータを読み込んだり、プログラムで処理した結果を書き出したりする必要があります。

データファイルの形式にも様々なものがありますが、ここでは一般に普及している (表計算ソフトで使われるという意味)、**CSV**(comma separated value) 形式を読み書きするという題材を扱います。表計算ソフトではデータを縦横のマトリクスに配置して扱いますが、CSV 形式ではそのデータを 1 行ずつ、各セルを「,」で区切って並べた形で表します。

たとえば次の例は、日本のいくつかの都市の平均気温と年間降水量のデータを CSV 形式で表しています。

```
city,temprature,precitipation
Sapporo,8.2,1129.6
Sendai,11.9,1204.5
Tokyo,15.6,1405.3
Kanazawa,14.1,2592.6
Oosaka,16.3,1318.0
Hiroshima,16.2,1511.8
Kouchi,16.4,2582.4
Fukuoka,16.2,1604.3
Naha,22.4,2036.8
```

ぎっちり詰まっていて見づらいですが、人間が見るものではないのでこれで別に良い訳です。そして、文字列と数値はとくに区別していません。

テキストの例はすべて ASCII の文字だけですが、日本語を扱う場合は euc-jp または UTF8 ならバイト単位で見ていっても「,」の文字コードは「,」としてのみ使われるのでそのままで OK です (文字列の中身を処理するときはまた別の問題ですが、今回はバイト列のままでしか扱いません。

もう 1 つ、セルの中に「,」が出て来る場合はセルの文字列全体を「"..."」で囲むことで区切りとして扱わせないようになっているのですが、ここでは簡単のためその機能は実装しません。[1]

[1]演習問題としています。教員が面倒だと思うことは演習問題にしておくというのはこの業界の伝統ですので。

10.1.2 CSV を扱うデータ構造

CSV データをプログラムに読み込んだとして、それはどのようなデータ構造になっているのがいいでしょうか。ここでは行の並びは普通に配列にするとして、1 行の中は図 10.1 左のように、構造体の先頭にセルの個数 num があり、その先に文字列ポインタの配列 cell がくっついた形としました。

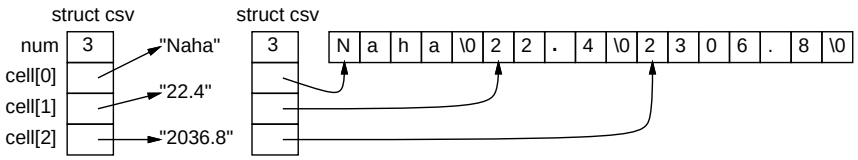


図 10.1: CSV の 1 行を表すデータ構造

実際のメモリ上では図 10.1 右のように、読み込んだ行がそのまま文字配列となっていて途中の「,」は各セルの文字列終わりを表すナル文字に変更し、各セルの先頭位置を cell[i] で指しています。

この構造を使って CSV を読み書きする API のヘッダファイルを見ましょう。これまでと違って、ヘッダファイル中に struct csv のフィールドが書かれています。これは、読み込んだ CSV を自分でも扱いたいので、そのためにはフィールド num や cell を参照する必要があるからです。言い替えれば、中身を見られるようにしているため、これは抽象データ型ではない普通のデータ構造です。

```
// csv.h --- csv file read/write API.
struct csv { int num; char *cell[1]; };
typedef struct csv *csvp;
int csv_read(char *fname, int limit, csvp arr[]);
void csv_write(char *fname, int size, csvp arr[]);
```

しかし、配列 cell の要素数が 1 になっていますが…これは、読み込む CSV に応じて (さらにもしかしたら行ごとに) セルの数は変わってくるので、いくつと書けないからです。しかし実際の領域そのものは、malloc で割り当てる時に指定すればいいので、いくつにでもできます。C 言語では配列の添字の範囲検査をしない (言語仕様上できない) ため、このような (他のフィールドと配列を直接くっつけた) 設計が可能です。[2]

関数の方ですが、csv_read はファイル名と struct csv のポインタの配列を渡し、そこに読み込んだ各行の構造体へのポインタを格納して行数を返します (ファイルが読めない、上限 limit で足りないなどのエラーがあった場合は負の数を返します)。csv_write は同じくファイル名と struct csv のポインタの配列と行数を渡し、ファイルに CSV 形式で内容を書き出します。

これまでの抽象データ型では使う側には内部構造が分からないので領域解放の関数も用意しましたが、にこのライブラリは内部を公開しているので、解放は各プログラムに任せることとしました。今回の例題ではファイルを 1 つ扱ったらもう終わるので解放はしていません。

10.1.3 CSV ライブラリの実装

では実装を見てみましょう。取り込むヘッダの中に ctype.h がありますが、これはだいぶ後の方で使いますので当面保留してください。関数 readline と read1 はこのファイル内だけの下請けなので static と指定しています。いずれも FILE*を受け取り、そこから 1 行ぶんのデータを読み取ります。

readline は単に 1 行ぶんの文字列を配列 buf に読み込み、そのデータをずっと残しておくため、必要な領域を malloc で割り当ててそこに文字列をコピーして割り当てた領域の先頭を返します。なお、1 行読み込み関数 fgets は行の最後の改行を削除しないので、文字列の最後が改行だったらそれをナル文字に変更するという処理もしています。

[2] Java など範囲検査をする言語では、レコードの領域と配列の領域は別にする必要があります。

```
// csv.c --- csv file read/write API impl.
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "csv.h"
#define MAXLINE 1000
static char *readline(FILE *f) {
  char buf[MAXLINE], *str;
  if(fgets(buf, MAXLINE, f) == NULL) { return NULL; }
  int len = strlen(buf);
  if(buf[len-1] == '\n') { buf[--len] = '\0'; }
  str = (char*)malloc(len+1); strcpy(str, buf); return str;
}
static csvp read1(FILE *f) {
  char *arr[100], *s = readline(f); if(s == NULL) { return NULL; }
  int i = 0;
  for(arr[i++] = s; *s != '\0'; ++s) {
    if(*s == ',') { *s = '\0'; arr[i++] = s+1; }
  }
  csvp r = (csvp)malloc(sizeof(struct csv) + i*sizeof(char*));
  r->num = i;
  for(i = 0; i < r->num; ++i) { r->cell[i] = arr[i]; }
  return r;
}
```

read1 は readline で 1 行を読み、各セル文字列の先頭を配列 arr に入れていきます。先頭は文字列の先頭で、あとは「','」がある毎にそこはナル文字に変更し、次の文字のアドレスを次のセルの先頭とします。行の最後まで終わったらこれでセルの数が分かりますから、malloc で構造体の領域を割り当てます。先に説明したように、セルの個数に応じて割り当てる領域を増やしています。[3]

read_csv はファイルを fopen で開き、read1 で繰り返し行を読み [4] 配列に構造体ポインタを格納し、終わったらファイルを閉じてサイズを返します。なお、エラーがあった場合は適宜負の値を返しますが、ファイルが開けなかったときはすぐ-1 を返してよいですが、配列満杯のときはファイルを fclose て後始末する必要があるため、size に-2 を入れてループを抜けるようにしています。

```
int csv_read(char *fname, int limit, csvp arr[]) {
  int size = 0; csvp line;
  FILE *f = fopen(fname, "rb"); if(f == NULL) { return -1; }
  while((line = read1(f)) != NULL) {
    if(size+1 >= limit) { size = -2; break; }
    arr[size++] = line;
  }
  fclose(f); return size;
}
```

[3] 厳密には構造体の宣言で cell[1] として 1 個ぶんを確保してあるので、増やす量は (i-1)*sizeof(char*) でよいですが、なんとなく読みやすさのために 1 引くのは省略しました。大した無駄ではないので。

[4] 「(line = read1(f))」は read1 から返された構造体のポインタを変数 line に入れますが、その入れた値を式の値として返します。もしそれが NULL だったらファイルの終わりなわけです。

```
void csv_write(char *fname, int size, csvp arr[]) {
  FILE *f = fopen(fname, "wb"); if(f == NULL) { return; }
  for(int i = 0; i < size; ++i) {
    fprintf(f, "%s", arr[i]->cell[0]);
    for(int j = 1; j < arr[i]->num; ++j) { fprintf(f, ",%s", arr[i]->cell[j]); }
    fprintf(f, "\n");
  }
  fclose(f);
}
```

csv_write は簡単で、ファイルを開いたあと各行ごとに、先頭のセルはそのまま、以後のセルは「,」に続いて出力し、行末の改行文字を出力するようになっています。実のところ、データを書き換えたりセルを増やすことを考えると、これを呼ぶより自前で出力する方が便利なことが多いでしょう。

10.1.4 CSV を読み込んで見る

では実際に先の CSV ファイルを読み込んでそのまま打ち出す、という例題を作りました (**のコメント行については後述)。

```
// csvread.c --- demonstration for csv_read.
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "csv.h"

int main(int argc, char *argv[]) {
  csvp data[1000], p;
  int size = csv_read(argv[1], 1000, data);
  if(size <= 0) { return 0; }
//qsort(data+1, size-1, sizeof(csvp), cmp1); // **
  p = data[0];
  printf("%11s %11s %11s\n", p->cell[0], p->cell[1], p->cell[2]);
  for(int i = 1; i < size; ++i) {
    p = data[i];
    printf("%11s %11.3f %11.3f\n",
           p->cell[0], atof(p->cell[1]), atof(p->cell[2])/12.0);
  }
  return 0;
}
```

CSV ファイルでは 1 行目は「各カラムが何を意味するか」を表す文字列 (ヘッダ) であるのが通例で先のもそうなっているので、1 行目は別扱いしています。揃えるためには printf の書式を使いますが、そのためには実数は数にする必要があるので、atof で文字列から変換しています。実行のようすは次の通り。

```
% gcc8 cvsread.c cvs.c
% ./a.out jcitytemp.csv
       city  temprature precitipation
    Sapporo       8.200    1129.600
     Sendai      11.900    1204.500
```

```
      Tokyo      15.600    1405.300
   Kanazawa      14.100    2592.600
     Oosaka      16.300    1318.000
  Hiroshima      16.200    1511.800
     Kouchi      16.400    2582.400
    Fukuoka      16.200    1604.300
       Naha      22.400    2036.800
```

演習 1 上の例題をそのまま動かせ。動いたら次のプログラムを作れ。実際に動かして確認すること。

a. 年間降水量のかわりに月平均降水量を打ち出す。
b. 最も降水量の多い都市のデータだけを打ち出す。
c. 最も平均気温が高い都市と低い都市のデータを打ち出す。
d. eps ライブラリと組み合わせてこのデータを好きに視覚化する。
e. その他好きなデータ処理をして打ち出す。

10.1.5 データの整列

前節のような「並んだデータ」に対し、特定の基準で**整列**する、というのは大変よくある処理です。整列アルゴリズムはあとで詳しくやりますが、その前に C の標準ライブラリにある **qsort** を使って整列をしてみましょう。`qsort` の宣言 (stdlib.h にあります) は次のものです。

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*comp)(const void*, const void*));
```

ここで `size_t` は整数と思ってください。`base` は整列する配列の先頭番地、`nmemb` は整列する要素数、`size` は要素 1 個あたりのバイト数です。これは、図 10.2 上のように、複数のフィールドから成る構造体が並んだ配列を整列する場合は要素の大きさが分かっている必要があるためです。

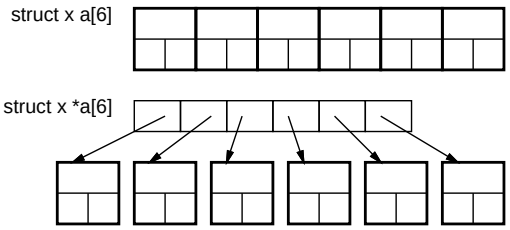


図 10.2: 構造体の配列と構造体ポインタの配列

ただ、整列は何回もデータを移動するため、構造体が直接並んだ配列だと構造体全体を何回もコピーする負荷が大きいです。そこで普通は、構造体は別の場所に置き (`malloc` で割り当てても別の構造体の配列に入っていてそのアドレスを取るのでもよい)、整列するのは「ポインタの配列」の方にすることが多いです (図 10.2 上)。前節までのデータ構造もそうなっていました。その場合、1 要素の大きさは `sizeof(csvp)`(ポインタ値は常に 8 バイト) になります。

話題を戻して、最後の引数が難しそうですが、「間接参照すると、`const void*`型の引数 2 個を取り整数を返す関数になる」値 (関数へのポインタ値) を渡します。つまり `qsort` は要素を「昇順」に並べますが、どの基準で昇順かは、パラメタで渡した**比較関数** (comparator function) を呼び出して決めます。[5]

[5]「**const** が初出ですが、これはこの関数は受け取ったポインタの指す領域を変更しませんよ、ということを表します。今日の C ではこの「変更する/しない」をきちんと書くようにライブラリのヘッダが作られていますが、ここでは記述が込み入るのでライブラリで必要とする最低限だけ扱います。

比較関数は 2 つの番地を受け取り、その番地に入っている 2 つの値どうしを比べ、「前者が小さい」なら負、「等しい」ならゼロ、「前者が大きい」なら正の整数を返すことになっています。ところで「番地を受け取る」に注意してください。これは図 10.2 上のように大きな構造を並べた配列であれば構造体をコピーするのでなく番地だけ渡した方が効率的だからそうなっていますが、図 10.2 下のようにもともとポインタの配列であっても、そのポインタが入っている配列上の番地を渡してくることになり、実際のポインタを取り出すには 1 段間接参照する必要があります。

では実際にやってみましょう。データを都市名の順に並べ替えるとして、次のような比較関数 cmp1 を作りました。これを main の上に置き、また strcmp を使っているのでヘッダファイル string.h も include します。

```
static int cmp1(const void *x, const void *y) { // comp-fn.
  csvp a = *(csvp*)x, b = *(csvp*)y;
  return strcmp(a->cell[0], b->cell[0]);
}
```

内容は見ての通りで、パラメタのポインタを csvp*にキャストしてから間接参照し、変数 a と b にポインタを取り出します。そのあと、2 つの都市名を strcmp で比較して小/等しい/大の場合にそれぞれ負/ゼロ/正の値を返します。main の中での呼び出しは先の例題にコメントで書いてありました。

なぜ data+1 から size-1 個かというと、先頭の行はヘッダ (各列の見出し文字列だけが入っている) からです。動かしてみると、確かに都市名の ABC 順になっています。なお、整列に使う欄 (鍵ないしキー) は 1 つとは限りません。たとえばこの例でも温度が同じ都市があります。そこで「まず温度順、温度が同じものは降水量順」のように 2 番目以降の整列鍵を指定するわけです。

```
% ./a.out jcitytemp.csv
       city  temprature precitipation
     Fukuoka      16.200     133.692
   Hiroshima      16.200     125.983
    Kanazawa      14.100     216.050
      Kouchi      16.400     215.200
        Naha      22.400     169.733
     Oosaka      16.300     109.833
    Sapporo       8.200      94.133
     Sendai      11.900     100.375
      Tokyo      15.600     117.108
```

演習 2 整列する例題をそのまま動かせ。動いたら次のように変更し、動かして動作を確認せよ。

a. 都市名の降順 (例題と反対の順) に並べる。
b. 降水量の多い順に並べる。
c. 平均気温の高い順に並べる。ただし平均気温が同じ場合は都市名の ABC 順に並べる (データを適当に修正して確認してよい)。
d. ヨーロッパの各都市の毎月の平均気温と年平均気温を記録した CSV を授業サイトに用意した。冒頭部分は次のようになっている。これを「国名の降順、国が同じ中では都市名の昇順」で並べて CSV として出力する。

```
Nation,City,1,2,3,4,5,6,7,8,9,10,11,12,AVG
Austria,Vienna,0.3,1.5,5.7,10.7,15.7,18.7,20.8,20.2,15.4,10.2,5.1,1.1,10.4
Belgium,Brussels,3.3,3.7,6.8,9.8,13.6,16.2,18.4,18.0,14.9,11.1,6.8,3.9,10.5
...
Finland,Helsinki,-3.9,-4.7,-1.3,3.9,10.2,14.6,17.8,16.3,11.5,6.6,1.6,-2.0,5.9
Finland,Kuopio,-9.2,-9.2,-4.1,2.0,9.1,14.5,17.5,15.0,9.7,4.1,-2.0,-6.7,3.4
Finland,Oulu,-9.6,-9.3,-4.8,1.4,7.8,13.5,16.5,14.1,8.9,3.3,-2.8,-7.1,2.7
...
```

e. 好きなデータを整列した上で何らかのデータ処理か視覚化をおこなう。

10.2 基本的な整列アルゴリズム＋α

10.2.1 選択ソート (単純選択法)

ここからは様々な整列アルゴリズムを見るため、データの方は「整数の配列を整列」という最も簡単な (分かりやすい) 形を取りましょう。整列順は昇順とします。

最初は**単純選択法** (選択ソート、selection sort) と呼ばれるアルゴリズムです。これは図 10.3 にあるように、「1〜n の範囲で最小の値を 1 番目に置く」「2〜n の範囲で最小の値を 2 番目に置く」「3〜n の範囲で最小の値を 3 番目に置く」と繰り返して行くことで整列を完成させます。この「1 番目に置く」等のとき、その場所にこれまであった値を無くすとまずいですが、幸いそこに置こうとする値がこれまであった場所は空くので、そちらに移します (要は交換するわけです)。

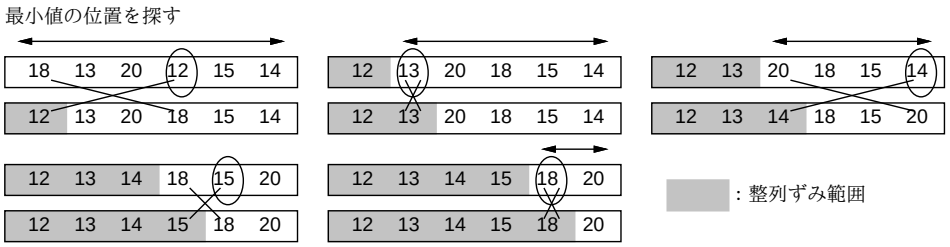


図 10.3: 単純選択法

ではコードを見てみましょう。配列 a の i 番と j 番を交換する関数 iswap がまず必要です。これはもう説明不要ですね。

次に、配列 a の i 番と j 番の範囲で「一番小さい値の位置」を調べる下請け関数 minrange を用意しました (これがある方が分かりやすい)。それには、a[i] をとりあえず min、i をとりあえず pos に入れ、それから i+1〜j の範囲について、もし a[k] が min より小さいならその値を min に入れ直し、同時に k を pos に入れ直します。

代入「=」は演算子であり、代入した値そのものを返すことから、「pos を入れ直しつつ a の添字としても使う」ように書いてありますが、このように短く書けるところは C 言語の発明です (今や大抵の言語でできますが)。調べ終わったら最後に位置 pos を返します。

```
static void iswap(int a[], int i, int j) {
  int x = a[i]; a[i] = a[j]; a[j] = x;
}
static int minrange(int a[], int i, int j) {
  int min = a[i], pos = i;
  for(int k = i+1; k <= j; ++k) {
    if(a[k] < min) { min = a[pos = k]; }
  }
  return pos;
}
void selectionsort(int n, int a[]) {
  for(int i = 0; i < n; ++i) { iswap(a, i, minrange(a, i, n-1)); }
}
```

以上の準備ができたら本体は簡単で、「0〜n-1 の範囲の i について、i 番目に i〜n-1 番の範囲の最小を置く」だけです (もともとそういうアルゴリズムです)。このように、アルゴリズムとなるべく近い形にコードが書けると分かりやすいでしょう?

10.2.2 挿入ソート (単純挿入法)

次は**単純挿入法** (挿入ソート、insertion sort) です。この方法は、1 番目は 1 個だけで並んだ列だと考え、「2 番目の値を取り上げ、1〜1 番の列の正しい位置に挿入」「3 番目の値を取り上げ、1〜2 番の列の正しい位置に挿入」「4 番目の値を取り上げ、1〜3 番の列の正しい位置に挿入」と繰り返して行きます (図 10.4)。

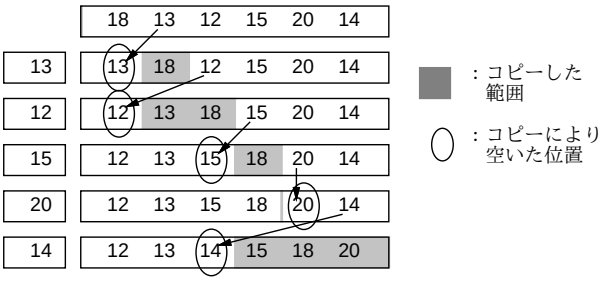


図 10.4: 単純挿入法

コードですが、「i 番目の値を 0〜i-1 番 (整列済み) の正しい位置に移す」下請け `shiftrange` を用意しました。それには、i 番目を `x` に取り出し、それから i を 1 つずつ減らしながら、i〜1 番目に i-1〜0 番の要素を入れて行きます (つまり 1 つずつ後ろにずらす)。

ただしずらすのは `x` より大きい要素だけで、そうでないものがでてきたらそこで止まります (`x` をそれより前に置いたら整列にならない)。最後まで全部ずらす場合もあります。最後に、ずらして空いた箇所に `x` を戻して終わりです。下請けができれば、本体は「1〜n-1 について、それを正しい位置に挿入」するだけです。

```
static int shiftrange(int a[], int i) {
  int x = a[i];
  for( ; i > 0 && a[i-1] > x; --i) { a[i] = a[i-1]; }
  a[i] = x;
}
void insertionsort(int n, int a[]) {
  for(int i = 1; i < n; ++i) { shiftrange(a, i); }
}
```

10.2.3 バブルソート

バブルソート (bubble sort) の原理は、昇順に並べるのですから、順に隣接する要素どうしを比較し、「大小が逆なら交換」することです。そうすると図 10.5 のように、1 巡目が終わると最大値は右端に移動します。2 巡目も同じようにすると、こんどは最大から 2 番目が最大の隣に移動します。

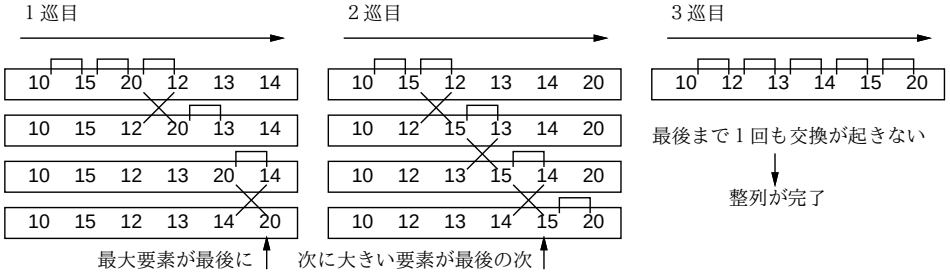


図 10.5: バブルソート

これを繰り返して行くと n-1 巡すれば必ず終わりますが、運がよければそんなにやらなくても途中で完成することもあります。それは、1 巡してみて「1 回も交換が起きなかった」ことで分かります。

このことを知るには「旗 (flag)」を使います。1 巡する前に旗を立て、比較した結果交換をするときは旗を降ろします。最後まで来て旗が降りていなかったら、1 回も交換していないので完了です。

次のコードでは、while 文を使うため、最初は旗の変数を「降りた状態」とし、while の条件は「旗が降りている間」で、while の中ではまず旗を立ててから一巡に入るようにしています。

```
void bubblesort(int n, int a[]) {
  bool done = false;
  while(!done) {
    done = true;
    for(int i = 1; i < n; ++i) {
      if(a[i-1] > a[i]) { iswap(a, i-1, i); done = false; }
    }
  }
}
```

10.2.4 コムソート

バブルソートは分かりやすいけれど手間が多く速くありません。それを少し変更しただけで速くできる例として、**コムソート** (comb sort) を見てみましょう。コム (comb) とは櫛のことで、「最初は髪がボサボサなので荒い櫛でとき、整ってきたら徐々に細かい櫛にする」というのが名前の由来だそうです。

どういうことかというと、先のバブルソートでは `a[i-1]` と `a[i]` を比較交換していましたが、これでは最初から目の細かい櫛で沢山ひっかかります。そこで距離 `d` を導入し、`a[i-d]` と `a[i]` を比較交換することにして、`d` を最初は大きく (たとえば配列のサイズ)、一定比率で小さくしていき、最後は 1 になるようにします (図 10.6)。1 になったときにはバブルソートと同じなので、旗を用いて整列が完了するまで反復します。

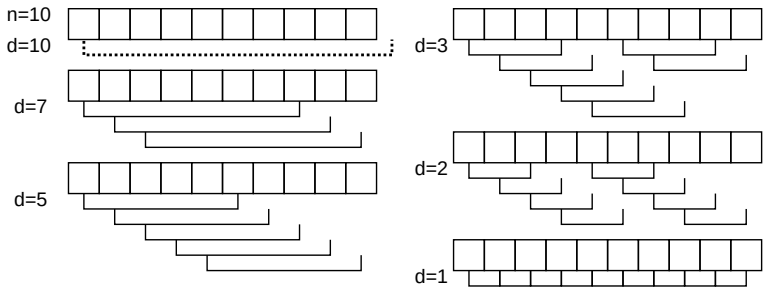


図 10.6: コムソート

ここで重要なのは「どれくらいの速さで」`d` を小さくしていくかです。あまり「一気に小さく」してしまうと、まだ髪が十分とけていないので並んでいない要素が多く、`d=1` で何回もとかすためバブルソートと変わりません。コムソートでは `d` を「毎回 $\frac{10}{13}$ 倍」にするのがよいとされています。

ということでコードを見てみましょう。バブルソートとほとんど変わりませんが、上述のように `d` 離れたところどうしを比較します。`d` の初期値は `n` とし、ループ内で 1 より大きい場合に毎回 $\frac{10}{13}$ 倍します。そして while の条件が「`d` が 1 より大きいか、または旗が降りている間」になっています。

```
void combsort(int n, int a[]) {
  int d = n; bool done = false;
  while(d > 1 || !done) {
```

```
      done = true;
      for(int i = d; i < n; ++i) {
        if(a[i-d] > a[i]) { iswap(a, i-d, i); done = false; }
      }
      if(d > 1) { d = d * 10 / 13; }
    }
  }
```

10.2.5 単体テストと時間計測

では実際に整列ができているか調べるため、単体テストを作成しましょう。これまでと違い、大量データでテストするため、間違いがなければOK、あれば大小が違っている最初の5個を出力するというふうにしました。さらに時間計測も行っています。整列を行う関数は関数ポインタで受け取り、それを時間計測しつつ呼び出しています。「`(*f)(n, a)`」は関数ポインタ `f` で指されている関数に引数 `n` と `a` を呼び出すことを意味します。使用する配列は動的に割り当て、終了時に開放します。

`main` ではテストのため生成するデータ数をコマンド引数として受け取るようにしました。整列のコードはコピーして入れてもよいですが、プロトタイプ宣言を書いておいてコンパイル時に一緒にしてもよいです。

```
  // test_sort.c --- unit test for sort algorithms.
  #include <stdbool.h>
  #include <stdio.h>
  #include <stdlib.h>
  #include <time.h>

  (ここに各種ソートのコードかプロトタイプ宣言を入れる)
  void expect_sort_iarray(int n, void (*f)(int m, int *a), char *msg) {
    int c = 0, *a = (int*)malloc(n * sizeof(int));
    srand(time(NULL));
    for(int i = 0; i < n; ++i) { a[i] = rand()%10000; }
    struct timespec tm1, tm2;
    clock_gettime(CLOCK_REALTIME, &tm1);
    (*f)(n, );
    clock_gettime(CLOCK_REALTIME, &tm2);
    for(int i = 1; i < n; ++i) {
      if(a[i-1] <= a[i]) { continue; } // correct order
      if(++c < 5) {
        printf("  wrong order at %d: %d > %d\n", i-1, a[i-1], a[i]);
      } else if(c == 5) {
        printf("  more wrong place omitted.\n");
      }
    }
    double dt = (tm2.tv_sec-tm1.tv_sec) + 1e-9*(tm2.tv_nsec-tm1.tv_nsec);
    printf("%s time=%.4f %s\n", c==0?"OK":"NG", dt, msg); free(a);
  }
  int main(int argc, char *argv[]) {
    int n = atoi(argv[1]);
    expect_sort_iarray(n, selectionsort, "selectionsort");
```

```
  //expect_sort_iarray(n, insertionsort, "inserctionsort");
  //expect_sort_iarray(n, bubblesort, "bubblesort");
  //expect_sort_iarray(n, combsort, "combsort");
    return 0;
  }
```

実行例を示します。整列はできているようです。

```
% ./a.out 1000
OK time=0.0015 selectionsort
OK time=0.0008 inserctionsort
OK time=0.0045 bubblesort
OK time=0.0002 combsort
```

これで要素数 50000 にしたときの各種アルゴリズムの時間を手元のマシンで計測してみたものが表 10.1 です。バブルソートは遅いですが、コムソートはとても速いことが分かります。

表 10.1: n = 50000 での各種整列の所要時間

アルゴリズム	時間 (秒)	時間計算量
selectionsort	3.770	$O(n^2)$
insertionsort	2.371	$O(n^2)$
bubblesort	14.824	$O(n^2)$
combsort	0.012	$O(n \log n)$

時間計算量ですが、選択ソート、挿入ソート、バブルソートはいずれも $O(n^2)$ になります。これは、外側ループの回数が n 回 (バブルソートは途中で終わるかも知れませんがいずれにせよ n に比例する回数)、内側ループの回数が前 2 者は $1 \sim n$ 回で平均 $\frac{n}{2}$ 回、バブルソートは常に n 回で、掛け算すると n^2 に比例する処理が必要になるためです。

コムソートはどうでしょう。d の値が n から初めて一定比率倍で減っていき最後は 1 になるので、1 になるまでの回数は $\log n$ に比例します。なので、「1 になったときには整列もほぼ終わっている」ようになっていれば (実際に $\frac{10}{13}$ 倍というのはそうなるように調整した値です)、この回数に 1 巡あたりの比較交換数 (徐々に多くなりますが最大 n です) を掛けて、$O(n \log n)$ になるわけです。

演習 3 コムソートの計算時間および時間計算量が、d に対して掛け算する値 (例では $\frac{10}{13}$) が変化するとどのように影響されるか、予想し、また実際に計測して検討しなさい。

演習 4 各アルゴリズムでランダムなデータを整列する時間については上で述べた通りであるが、これが「昇順に並んでいる」「降順に並んでいる」データだとそれぞれどのようになると思われるか。予想し、また実際に計測して検討しなさい。

演習 5 整列アルゴリズムについて好きなものを、先に出て来た qsort と同じ呼び方で使えるように変更し、qsrot の代わりに CSV ファイルの整列に使用して結果を確認しなさい。CSV を扱うプログラムは qsrot を渡すところ以外は変更しないこと。

本日の課題 10A

「演習 1」～「演習 2」で動かしたプログラム 1 つを含むレポートを本日中 (授業日の 23:59 まで) に提出してください。

1. sol または CED 環境で「/home3/staff/ka002689/prog19upload 10a ファイル名」で以下の内容を提出。

2. 学籍番号、氏名、ペアの学籍番号 (または「個人作業」)、提出日時。名前の行は先頭に「@@@」を付けることを勧める。
3. プログラムどれか 1 つのソースと「簡単な」説明。
4. レビュー課題。提出プログラムに対する他人 (ペア以外) からの簡単な (ただしプログラムの内容に関する) コメント。
5. 以下のアンケートの回答。

   Q1. CSV ファイルの読み込み方法が分かりましたか。
   Q2. 基本的な整列アルゴリズムを理解しましたか。
   Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

次回までの課題 10B

「演習 1」〜「演習 5」(ただし 10A で提出したものは除外、以後も同様) の (小) 課題から選択して 2 つ以上課題をやり、レポートを提出しなさい。できるだけ複数の演習から選ぶこと。レポートは次回授業前日 23:69 を期限とします。

1. sol または CED 環境で「`/home3/staff/ka002689/prog19upload 10b` ファイル名」で以下の内容を提出。
2. 学籍番号、氏名、ペアの学籍番号 (または「個人作業」)、提出日時。名前の行は先頭に「@@@」を付けることを勧める。
3. 1 つ目の課題の再掲 (どの課題をやったか分かればよい)、プログラムのソースと「丁寧な」説明、および考察 (課題をやってみて分かったこと、分析、疑問点など)。
4. 2 つ目の課題についても同様。
5. 以下のアンケートの回答。

   Q1. CSV ファイルを読み込んでデータ処理できそうですか。
   Q2. プログラムのコードを見て時間計算量が分かりそうですか。
   Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

＃11　高速な整列アルゴリズム

今回は次のことが目標となります。

- 様々な整列アルゴリズムについて理解する
- 整列における安定性の概念を理解する

11.1　計算量の検討とマージソート/クイックソート

11.1.1　より高速な整列のために必要なこと

前回扱った基本的な整列アルゴリズム (選択ソート、挿入ソート、バブルソート) ではいずれも、n 個の数それぞれに対して、n に比例する回数の操作を行うため、計算量が $O(n^2)$ となっていました。$O(n^2)$ では、数万程度くらいまでしか短い時間 (数秒程度) では扱えません。

ただ 1 つだけ、コムソートでは「間隔 d を n から始めて一定比率で狭めていき、1 になったときには整列が終わっている」ようにすることで、「n 個の数それぞれに対する操作を $\log n$ 回」で済ませることができ、$O(n \log n)$ の時間計算量を達成していました。

整列では扱う数 n は変えようがないので、それらに全体対して操作を行う回数を減らす (絶対値ではなく、n に比例から $\log n$ に比例、さらにできれば定数 C に比例というふうに) ことがポイントになります。今回はそれを実現する整列アルゴリズムを複数見て行きます。

11.1.2　マージ (併合) 操作

マージ (併合、merge) 操作とは、2 つの整列ずみの列を 1 本の整列ずみの列に合成する操作です。図 11.1 は、以前扱った「先頭に要素数を格納した整数配列」を用いた列に対するマージ操作を例示しています。

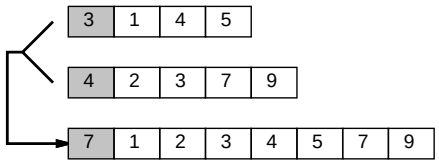


図 11.1: 列に対する併合操作

2 つの列を受け取り、併合した列を返すメソッド ivec_merge を見てみましょう。

```
int *ivec_new(int size) {
  int *a = (int*)malloc((size+1) * sizeof(int));
  a[0] = size; return a;
}
int *ivec_merge(int *b, int *c) {
  int ib = 1, ic = 1, ia = 1, *a = ivec_new(b[0]+c[0]);
  while(ia <= a[0]) {
    if(ib > b[0])          { a[ia++] = c[ic++]; }
    else if(ic > c[0])     { a[ia++] = b[ib++]; }
```

```
    else if(c[ic] < b[ib]) { a[ia++] = c[ic++]; }
    else                   { a[ia++] = b[ib++]; }
  }
  return a;
}
```

返す列の長さは 2 つの列の長さの和として分かるので、まず返す列 a の領域を確保します。それから、変数 ia を a の格納位置、ib、ic を b、c の取り出し位置 (いずれも最初は 1) としてから、列 a が一杯になるまでのループに入ります。

ループの中では、列 b の取り出し位置が最後まで来ていたら c から取り出し (取り出し位置は進める)、列 c の取り出し位置が最後まで来ていたら b から取り出し (〃)、いずれも最後でなければ 2 つの列の取り出し位置の要素の大小で小さい側から取り出し (〃)、a に格納します (格納位置は進める)。ループが終わったら a に結果ができているので返します。

11.1.3 キューを使ったマージソート

ではこれを整列に活かすにはどうしたらよいでしょう。1 つの分かりやすい方法は、最初はすべての数を「長さ 1 の列」と考え (長さ 1 ならそのままで整列ずみと言えます)、それらを順次マージしていくことです。マージ中の多くの列を蓄えるのにはキューが使えます。すなわち図 11.2 のように、キューから 2 つ deq で取り出して 1 つにマージしたものを enq で投入することを繰り返すわけです。

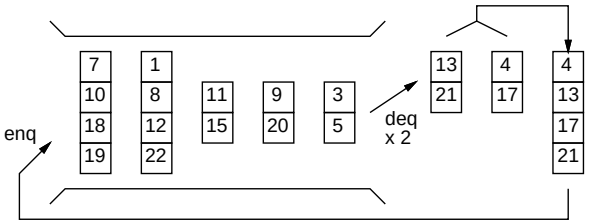


図 11.2: キューを用いたマージソート

1 回マージすることにキューの内容は差し引き 1 ずつ減って行きますから、最後に 1 つの列になり、その列がもとの列の整列ずみの内容となっています。これをコードにしたものを示します。

```
// mergesort1 --- merge sort unsing queue
#include <stdlib.h>
#include "pqueue.h"
(ivec_new, ivec_merge をここに)
void mergesort1(int *a, int n) {
  pqueuep q = pqueue_new(n+1); int *v, *w;
  for(int i = 0; i < n; ++i) {
    v = ivec_new(1); v[1] = a[i]; pqueue_enq(q, v);
  }
  while(true) {
    v = (int*)pqueue_deq(q); if(pqueue_isempty(q)) { break; }
    w = (int*)pqueue_deq(q); pqueue_enq(q, ivec_merge(v, w));
    free(v); free(w);
  }
  for(int i = 0; i < n; ++i) { a[i] = v[i+1]; }
}
```

キューの各要素はポインタ (整数へのポインタ) になるので、ポインタ用のキューを使うものとします (コードは整数用と同様、授業サイトに掲載)。まず元の配列 (これは個数つきではなく他の例列コードと合わせて個数を別に渡しています) から、n 個の長さ 1 の個数つき配列を作り、その要素としてデータを入れてキューに enq していきます。そのあと無限ループで 2 つ deq してマージしたものを enq しますが (取り出した 2 つの領域は以後不要なので `free`)、ただし 1 つ取り出した時点でキューが空なら終わりなのでループを抜け、元の領域にデータをコピーし戻します。

演習 1 キューを使ったマージソートのコードを動かし、整列できることや整列時間を確認しなさい (単体テストすること)。加えて、キューの代わりにスタック (ポインタなので `pstack`) を使うとどうなるか理由とともに予想し、こちらも単体テストして確認しなさい。今回の課題全般に、前回の `expect_sort_iarray` を (必要なら改造の上) 利用するとよいでしょう。

演習 2 「長さ 1 から始めてマージしていく」やり方は、キューを使わないでも実現可能である。そのようなマージソートのコードを作成し、整列できることや整列時間を確認しなさい (単体テストすること)。

11.1.4 分割統治による再帰マージソート

先の例では「長さ 1 から始めてマージしていく」考え方でしたが、逆に分割統治、つまり「長さ n の問題を分割して扱う」ことを再帰的に行う考え方でもできます。具体的には「$\frac{n}{2}$ ずつの列に分けて、自分を再帰呼び出しして整列させ、終わったらそれをマージする」形になります。

```
// mergesort2 --- merge sort with recuresive division
#include <stdlib.h>
(merge はここでは省略)
static void ms(int *a, int i, int j, int *b) {
  if(i >= j) { return; }
  int k = (i + j) / 2;
  ms(a, i, k, b); ms(a, k+1, j, b);
  merge(b, a+i, k-i+1, a+k+1, j-k);
  for(k = 0; k < j-i+1; ++k) { a[i+k] = b[k]; }
}
void mergesort2(int *a, int n) {
  int *b = (int*)malloc(n * sizeof(int)); ms(a, 0, n-1, b); free(b);
}
```

この場合、「配列の何番から何番を整列する」というふうにパラメタで明示する方が扱いやすいので、`mergesort2` は範囲を明示して再帰手続き `ms` を呼ぶ仲介だけの役割です。また、マージする時は作業用の配列が欲しいので、それを割り当てて `ms` に渡し、終わったら解放します。

`ms` ですが、配列と整列範囲 (`i` から `j` まで) と作業用配列 `b` を受け取ります。範囲の長さが 1 以下なら整列は済んでいるのですぐ戻ります。それ以外は、中間点 `k` を決めて、`i`〜`k` と `k+1`〜`j` をそれぞれ自分を再帰呼び出しして整列します。終わったらマージしますが、そのパラメタは結果を受け取る配列と、1 番目の配列および長さ、2 番目の配列および長さです。1 番目と 2 番目の配列はポインタ計算 `a+i`、`a+k+1` で求まることに注意。マージ後は結果を `b` から `a` の当該位置にコピーし戻します。

演習 3 省略されている `merge` を書いて補って再帰マージソートを動かし、整列できることや整列時間を確認しなさい (単体テストすること)。

11.1.5 クイックソート

再帰版マージソートは言うなれば「とりあえず列を 2 つに分けて (自分を再帰呼び出しして) 整列してしまう」ため、その整列できた 2 つの列を 1 つにするために面倒なマージという操作が必要なのでした (図 11.3 左)。そこで逆に、まず列を「ある値 p より大きい部分と小さい部分に分割し (partition)、それからそれぞれを (自分を再帰呼び出しして) 整列する」ようにすれば、再帰が終わった時はもう「p より小、p、p より大」の順になっているので整列が完了します (図 11.3 右)。これが**クイックソート** (quick sort) で、p の値のことをピボットと呼びます。

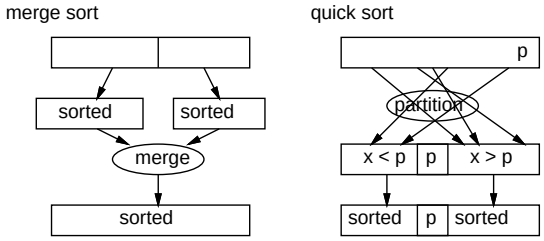


図 11.3: 再帰マージソートとクイックソート

コードを示します。quicksort は範囲を指定して下請け qs を呼びます。qs はピボット p には左端の値を使います。ランダムな列であればどこを取っても同じなのでこれでよいですが、整列済みの列だとまずいですね。

```
// quicksort --- quick sort with recuresive division
static void iswap(int *a, int i, int j) {
  int x = a[i]; a[i] = a[j]; a[j] = x;
}
void qs(int *a, int i, int j) {
  if(j <= i) { return; }
  int s = i, pivot = a[j];
  for(int k = i; k < j; ++k) {
    if(a[k] < pivot) { iswap(a, s++, k); }
  }
  iswap(a, j, s); qs(a, i, s-1); qs(a, s+1, j);
}
void quicksort(int *a, int n) { qs(a, 0, n-1); }
```

内側のループが分かりづらいですが、s は「ここより手前は p より小さい」という範囲を表す変数であり、変数 k を使って整列範囲全体を調べながら、p より小さい値があればそれを s の位置と交換することで s の場所に置き、s は 1 つ増やすようにすることで、ループの最後には全部の p より小さい値が s の手前に集まります。そこで最後に右端にあるピボットを s の位置と交換することで「p より小、p、p より大 (厳密には以上)」と並ぶわけです。

演習 4 上のクイックソートを動かし、整列できることや整列時間を確認しなさい (単体テストすること)。また、既に整列されている列を渡したときの挙動についても調べ、そのときの問題を解消する方法を実装しなさい (これも単体テストすること)。(ヒント: ピボットを整列範囲内からランダムに選べばよいです。)

11.1.6 ボゴソート

高速な整列アルゴリズムの話題の中ですが、逆に「できるだけ遅いアルゴリズム」ということで考案されたのが**ボゴソート** (bogo sort) です。その原理は簡単で「列をランダムにシャッフルし、並んでい

るかチェックする。並んでいなければまたシャッフルし…」と繰り返します。

演習 5 ボゴソートを実装し、動作と時間を確認しなさい (10 個くらいでやるのが無難かも)。また、時間計算量を見積もり、実測と比較検討しなさい (単体テストすること)。

演習 6 ボゴソートと同程度に遅い整列アルゴリズムは他にもある。考案し、実装してみなさい (単体テストすること)。

11.2 完全 2 分木の配列表現とヒープソート

11.2.1 2 分木とその表現

木 (tree) とは地面に生えている…ではなく、数学の場合「閉路を含まない**グラフ** (頂点と辺の集合)」ですが、コンピュータ科学の場合は「**根** (root)」はら始まり複数の「**節** (node)」がたどれるような (そしてやはり閉路は含まない) データ構造です。ある節にとって根に近い側の辺につながる節は「**親** (parent)」、それ以外の辺についながる節は「**子** (child)」となります。各節において、根からその節までの経路の長さ (辺の数) を深さ、子の数を**次数**と呼び、次数が 0 の節を「**葉** (leaf)」と呼びます。

データ構造として実現するときは、子の節へのポインタを親の節が持つので、その個数が問題になります。次数の最大が 2 である木は **2 分木** (binary tree)、3 以上である場合は**多分木** (N-ary tree) と呼びます。2 分木では子が最大 2 つであり、これを「左の子」「右の子」のように呼ぶことがあります。

2 分木のうち、すべての葉でない節の次数が 2 であるものを「**全 2 分木** (full binary tree)」と呼びます。ここで「**完全 2 分木** (complete binary tree)」という用語もあるのですが、これには (1) 全ての葉の深さが等しい全 2 分木 (節数は $2^n - 1$ に限られる)、(2) 前期に加えてそれに左から N 個ぶん葉を追加したもの、の 2 通りの意味があります。図 11.4 にこれらの構造を例示しました (最後の完全 2 分木で次数が 1 の節にくっついているのは「左の子」であることに注意)。

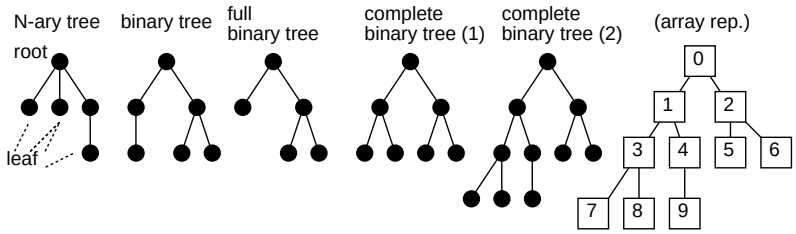


図 11.4: 木とさまざまな 2 分木

ここまで、いかにもポインタを使った動的データ構造のような流れで来ましたが、完全 2 分木 (2) については、配列を使った次のような表現方法があります。

- 根を添字 0 の位置に割り当てる。
- 任意の節 i(添字 i) について、左の子は添字 $2i+1$、右の子は添字 $2i+2$ に置く (そうすると、逆に子 i に対して親の添字は $(i-1)/2$ で表せる。除算は整数除算)。

図 11.4 の右端の図が、隣の完全 2 分木を配列表現したときの番号を表しています。[1]

11.2.2 完全 2 分木による最大ヒープと押し下げ

ヒープ (heap) とは「積み上げた束」という意味の英語ですがコンピュータ科学ではメモリの空き領域を集めて保持したものを通常言います。`malloc` はヒープからメモリを取って来る関数です。そしてもう 1 つの用法として、「**最大ヒープ** (maximum heap)」といった場合、次のような性質を持つ木構造を言います (大小を逆にした「最小ヒープ」もあります)。

[1] ここでは C 言語に合わせて根を 0 番としましたがが、根が 1 番の方が計算式は分かりやすく、左の子が $2i$、右の子が $2i+1$ になり、親が $i/2$ になります。

- 親の節の「値」が、(子があるとき) いずれの子の節の「値」よりも大きい。

上記が成り立てば当然、根には最大値があるので、その最大値を取り出し続けて並べれば整列が行えます。これをヒープソート (heap sort) と呼びます。ただし、効率がよい整列アルゴリズムであるためには、1 つ最大値を取り出したあと、そこに適当な代わりの値 (配列表現では配列に残っている最後の値を通常使います) を入れて上記の最大ヒープの条件が崩れた後、その条件を再度成り立たせる必要があり、しかもそれが $\log n$ の手間である必要があります。どうでしょうか。

実際にその方法を例示しましょう。図 11.5 左の最大ヒープにおいて、先頭の要素を取り除き、配列末尾にあった「8」をその位置に置いて埋めたとします (図 11.5 中)。この状態では最大ヒープではありません。そこで、この「8」を左または右の子と交換します。

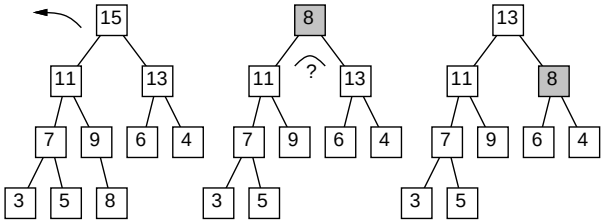


図 11.5: 最大ヒープの押し下げ操作

どちらと交換すべきでしょうか。もちろん、「大きい方と」交換すべきです。そうすれば、交換しなかった側の枝は確かに「親が大きい」が維持されていて、それ以上修正は不要です。さて、次に交換で新たに「8」を置いたところから下を検討します。2 分木は再帰的データ構造 (部分が全体と同じ構造) ですから、先と同様に進めます。まず最大ヒープになっているか調べますが、今度は左右の子とも「8」より小さいので、これで OK で終わりです (図 11.5 右)。

もしどちらかが「8」より大きいなら、再度大きい方の枝を選び交換して続けますが、最後は葉まで来ますからそこで必ず終わります。この、先頭に任意の値を置いて最大ヒープの性質を満たすように下に移して行く操作を「**押し下げ** (pushdown)」と呼んでいます。ノード数 n の完全 2 分木の高さ (葉までの深さの最大) は $\log_2 n$ ですから、押し下げ操作は最大 $\log n$ ステップで終わります。ということで、「次々に最大を取り出していく」部分の計算量は $O(n \log n)$ となります。

まだ 1 つ忘れていますね。最初に最大ヒープを作るのはどうしたらいいでしょうか。それは、今度は配列の最後から (つまり木の低い側から) 順にすべての節に対して押し下げを実施すればよいのです。葉のところはどのみち入れ換えはないですが、その上の段からは必要に応じて入れ換えが起きていきます。どの段階でも、ある節の押し下げを行うときには、その下はすべて最大ヒープになっているので、同じ押し下げの手順で OK です。ということは、n 個の値に対して最大 $\log n$ ステップの押し下げを行うので、最初にヒープを作る部分の計算量も $O(n \log n)$ です。

11.2.3 ヒープソートのコード

ここまでで全部説明はしてしまったので、あとはコードを見るだけです。まず見慣れないものとして、`P`、`L`、`R` という define がありますが、この define にはパラメタ `i` が付いています。これはちょうど関数と同じように右辺の中に埋め込まれて展開されます。関数でもよかったのですが、単なる計算式なので関数呼び出しが起きるよりは効率のよいマクロにしたかったということがあります。`iswap` はいつも通りです。

押し下げ操作ですが、配列、押し下げる節番号、そして最大ヒープの末尾の節番号を渡します。まず左の子の節番号を `k` に求めて、それが末尾を超えていない間繰り返しになります。次にループの中ですが、最初の if 文では、終わりまでに 1 以上余裕があれば右の子の節もあるので、その値の方が大きいならそちらを `k` にします (`k` は入れ換える場所なわけです)。次に、左右の子の大きいものより、親 (つまり入れ換えようかと思っていた節) が大きければ、もう入れ換える必要はないのでループを

抜けます。そうでなければ下へ行き、親と k 番を入れ換えたあと、k はその左の子にしてからループを周回します。

```
// heapsort1 --- heap sort using balanced bin-tree
#define P(i) ((i-1)/2)
#define L(i) (2*i+1)
#define R(i) (2*i+2)

static void iswap(int *a, int i, int j) {
  int x = a[i]; a[i] = a[j]; a[j] = x;
}
void pd(int *a, int i, int j) {
  for(int k = L(i); k <= j; ) {
    if(k < j && a[k] < a[k+1]) { ++k; }
    if(a[P(k)] >= a[k]) { break; }
    iswap(a, P(k), k); k = L(k);
  }
}
void heapsort1(int *a, int n) {
  for(int i = n-1; i >= 0; --i) { pd(a, i, n-1); }
  for(int i = n-1; i > 0; --i) { iswap(a, 0, i); pd(a, 0, i-1); }
}
```

ヒープソート本体は前述の通り簡単です。まず最初のループで、配列の後ろから前に無かって全ノードの押し下げをしてヒープを作ります。そのあと、最大ヒープの先頭から 1 つ取って後ろに起き、末尾を 1 減らす (実際には交換で両方一辺にやります)、そして押し下げを繰り返して行きます。

演習 7 ヒープソートの動作と実行時間を確認しなさい。またヒープソートの特徴として、このコードのように一気にヒープを作るのでなく、値を随時追加したり取り出したりして、取り出すと常に「現時点で入っているもののうち最大」が取れるようにもできることが挙げられる。ヒープソートのコードを流用してそのような機能を持つ情報隠蔽されたデータ構造 maxbuf を作ってみよ (中身はほぼ最大ヒープ)。それで n 個値を追加して n 個取り出した時もヒープソートができることになるので、その動作と時間も確認すること (単体テストすること)。

なお、「最大ヒープに 1 個値を追加する」場合は押し下げではなく、最後の位置に値を追加してから、その値を 2 分木の上に向かって (それより大きい値にぶつかるまでまたは根に着くまで) 親と交換していく押し上げ (push up) 操作が必要になります。押し上げの方が「左か右か」選択しないのでいくらか簡単です。

11.2.4 最大ヒープゲーム? option

配列を使った完全 2 分木による最大ヒープは直観的に分かりにくいので、これをゲームにしてみました。大きさを指定するとその大きさの配列に数値が埋められ、2 分木の配置で表示されます。

```
_0_
 1  2
 3  4  5  6
```

ここで「現在位置」は最初 0 番 (根) にありますが、コマンドで移動できます。現在位置と親との間で値を交換するコマンドもあります。タスクは、最大ヒープを構成した上で「移動」コマンドを使うと先頭要素が除去できるので、それを繰り返してなるべく速く全部除去することです。コマンドは次のものです。

- q — 終わる。いつの時点でも終わってよい
- l、r、p — 現在位置を左右の子/親に移す
- x — 現在位置と親の数値を交換する
- s — 配列をシャッフルする。ゲームなので
- m — 最大要素を除去し、末尾の要素をそこに置く。最大ヒープが構成できているときだけ動作する (メッセージで表示)。

コンパイル方法と動かし方は次の通り。何回か「s」を使ってよさげな配置になってから最大ヒープ化をするのがよいです。

```
% gcc8 heapgame.c -lncurses
% ./a.out 7  ← ヒープ中の数値の数
```

この部分はオプションで、詳しく説明すると大変なのでまあコードだけ掲載します。読めば読めると思います。

```
// heapgame.c --- construct heap and remove max screen game.
#include <stdio.h>
#include <stdbool.h>
#include <ncurses.h>
#include <stdlib.h>
#include <time.h>
#define P(i) ((i-1)/2)
#define L(i) (2*i+1)
#define R(i) (2*i+2)

#define MAXARR 128
static int a[MAXARR];
static int max, cur = 0;

static void pos(int n, int *x, int *y) {
  int y1 = 0, x1 = 0;
  for(int i = 1; i <= n; ++i) {
    if(i==1||i==3||i==7||i==15||i==31||i==63) { ++y1; x1 = 0; }
    else { ++x1; }
  }
  *x = x1; *y = y1;
}
static void upd() {
  char buf[10];
  int x, y;
  for(int i = 0; i < max; ++i) {
    pos(i, &x, &y); move(y*2+1, x*3+1);
    sprintf(buf, "%3d", a[i]); addstr(buf);
  }
  pos(cur, &x, &y); move(y*2+1, x*3+3);
}
void iswap(int *a, int i, int j) { int x = a[i]; a[i] = a[j]; a[j] = x; }
```

```
void shuffle(int *a, int size) {
  for(int i = 0; i < size; ++i) { iswap(a, i, rand()%(size-i)); }
}
bool checkheap(int *a, int size) {
  for(int i = 1; i < size; ++i) { if(a[P(i)] < a[i]) { return false; } }
  return true;
}
int main(int argc, char *argv[]) {
  max = atoi(argv[1]); if(max > MAXARR) { max = MAXARR; }
  for(int i = 0; i < max; ++i) { a[i] = i; }
  srand(time(NULL));
  initscr(); noecho(); cbreak(); system("stty raw"); clear();
  while(true) {
    upd(); refresh(); int ch = getch();
    switch(ch) {
case 'q': endwin(); return 0;                              // quit
case 'p': if(cur > 0) { cur = P(cur); } break;             // parent
case 'l': if(L(cur) < max) { cur = L(cur); } break;        // left-child
case 'r': if(R(cur) < max) { cur = R(cur); } break;        // right-chlid
case 'x': if(cur > 0) { iswap(a, cur, P(cur)); } break;    // excange
case 's': shuffle(a, max); break;                          // shuffle
case 'm': char *msg = "NG";                                // move top elt
          if(max <= 1) {
            msg = "GOAL!";
          } else if(checkheap(a, max)) {
            iswap(a, 0, max-1); --max; msg = "MOVED"; cur = 0;
          }
          clear(); move(15, 5); addstr(msg); clrtoeol(); break;
default: break;
    }
  }
}
```

演習 8 ゲームを何回かやって特性を体験しなさい。そのあと、次のような改訂をしてみなさい。

a. 現状では、押し下げの操作が複数キーを必要とするなど操作性がよくない。コマンドを追加して操作性を改善してみなさい。

b. 人間がやるのでなく、最大ヒープ化や押し下げをコンピュータに実行させなさい (スローモーションでようすが見られるとなおよいでしょう)。

c. 一定時間ごとに数値がヒープに追加されていき、消す速度を上回って上限を超えたらアウトみたいゲーム性を盛り込んでみなさい (一定時間ごと、は ncurses getch timeout などでぐぐって情報を調べる必要あり)。

d. 整列と関係あってもなくてもよいので、ncurses を使って何か面白いと思うものを作りなさい。

11.3 安定な整列と非安定な整列

だいぶ今更な話題なのですが、整列には「安定な整列」と「非安定な整列」があります。安定な整列とは、「キーとして同じ値の要素が複数含まれていた場合、整列結果における要素の並び順が元の並び順と同じままである」ものを言います。例えば図 11.6 を見てください。上の列を数値 (キー) の昇順に並べるとします。「3」の要素は 3 つあり、元の列では○、△、□の順に並んでいます。安定な整列では整列後もこの順番が維持されますが、非安定な整列では維持されません。

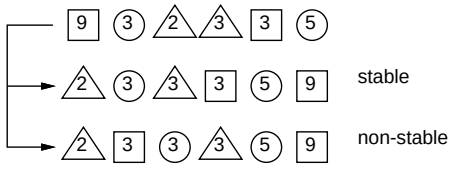


図 11.6: 安定な整列の概念

どのような整列でも、「元の項目番号」を振っておいて、それを追加のキー (もともとのキーが同じ値だったとき使う) にすれば、安定な整列にできます。ただ、それは結構面倒なので、安定性が必要なら最初から安定なアルゴリズムを利用した方が楽でしょう。

これまでのアルゴリズムで安定性について調べたいとします。それにはたとえば、これまでの例で整数を整列していたものを、すべて次のようなレコード型の値にします (unsigned short では最大は 65536 になりますが、通常の実験には十分でしょう)。

```
struct sortval { unsigned short key, seq; };
```

そして実験時に、key には乱数を入れますが、範囲を狭くして同じキーが複数現れるようにします。そして seq には一連番号を入れます。整列が終わってから、「隣接値でキーが同じで連番が逆になるもの」がないか調べればよいでしょう。

演習 9 ここまでに学んだ (またはこの後出て来るものでもよい) 整列アルゴリズムからなるべく多く選び、安定性について検討しなさい。また、実際に上記の方法で試してみて、自分の検討した結論が合っているかどうか確認しなさい。実験に用いたプログラムをきちんと説明・掲載すること。

本日の課題 11A

「演習 1」〜「演習 9」で動かしたプログラム 1 つを含むレポートを本日中 (授業日の 23:59 まで) に久野までに提出してください。

1. sol または CED 環境で「/home3/staff/ka002689/prog19upload 11a ファイル名」で以下の内容を提出。
2. 学籍番号、氏名、ペアの学籍番号 (または「個人作業」)、提出日時。名前の行は先頭に「@@@」を付けることを勧める。
3. プログラムどれか 1 つのソースと「簡単な」説明。
4. レビュー課題。提出プログラムに対する他人 (ペア以外) からの簡単な (ただしプログラムの内容に関する) コメント。
5. 以下のアンケートの回答。

   Q1. さまざまな整列手法からいくつくらい理解しましたか。
   Q2. 最大ヒープとは何か分かりましたか。
   Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

次回までの課題 11B

注意: B 課題は今回の 11B で最後で、次回からは A 課題のみとなります。期末も近くて皆様も大変でしょうから。それで、11B の期間は通常より 1 週間長く取っていますのでそのつもりで力作をどうぞ。

「演習 1」〜「演習 9」(ただし 11A で提出したものは除外、以後も同様) の (小) 課題から選択して 2 つ以上プログラムを作り、レポートを提出しなさい。できるだけ複数の演習から選ぶこと。レポートは「# 13 授業前日」23:69 を期限とします。

1. sol または CED 環境で「`/home3/staff/ka002689/prog19upload 11b` ファイル名」で以下の内容を提出。
2. 学籍番号、氏名、ペアの学籍番号 (または「個人作業」)、提出日時。名前の行は先頭に「@@@」を付けることを勧める。
3. 1 つ目の課題の再掲 (どの課題をやったか分かればよい)、プログラムのソースと「丁寧な」説明、および考察 (課題をやってみて分かったこと、分析、疑問点など)。
4. 2 つ目の課題についても同様。
5. 以下のアンケートの回答。

   Q1. 自力で書ける整列アルゴリズムは何と何でしょう。

   Q2. 整列が安定かどうか判断できるようになりましたか。

   Q3. リフレクション (今回の課題で分かったこと) ・感想・要望をどうぞ。

＃12　整数整列と探索アルゴリズム

今回から B 課題はありません。今回は次のことが目標となります。

- 整数のための整列アルゴリズムについて知る
- 探索の概念と主要な探索アルゴリズムについて知る

12.1　整数のための整列アルゴリズム

12.1.1　ビンソート (分配計数法)

前回までは、整列に最してキーに求められる要件は「比較により大/小/ 等しいが定まる」ことだけでした。しかし実際には、キーとして整数が使われることは多く、そのときは整数の性質を使うことで前回の $O(n \log n)$ を上回る性能を達成することもできます。今回はそのような手法を見てみます。

ビンソート (binsort) は、バケットソート (bucket sort) や分配計数法 (distribution counting) とも呼ばれ、「どのキー値がいくつあるか」数えることで整列をおこないます。すなわち、キーの最大値が m だとして、サイズ $m+1$ の配列は 0〜m の添字でアクセスできるので、初期値を 0 としてから整列データ中に「各キー値が何回現れるか」を数えることができますね。そうしたらあとは、添字の小さい方から順に「各キーを現れた個数ずつ並べていけば」整列が完了します (図 12.1)。

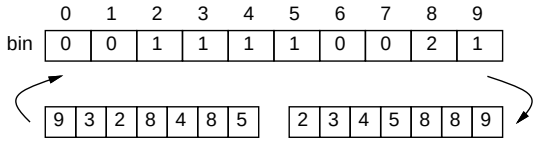


図 12.1: ビンソートの原理

さっそく、ビンソートのコードを見てみましょう。キーの最大値 `max` を受け取り、そのサイズの配列を割り当てて計数に使うようになっています (これまでのテストプログラムでは 4 桁の整数でやっていたので、最大値は 9999 ですね)。値を戻すときに「`bin[i]` 回繰り返す」のところを「`while(--bin[i] >= 0) ...`」としていますが、これも (その変数の値を壊してしまってのであれば) C 言語で便利に使える書き方です。

```
// binsort.c --- binsort with specified maximum
#include <stdlib.h>
void binsort(int n, int *a, int max) {
  int i, k = 0, *bin = (int*)malloc((max+1) * sizeof(int));
  for(i = 0; i <= max; ++i) { bin[i] = 0; }
  for(i = 0; i < n; ++i) { ++bin[a[i]]; }
  for(i = 0; i <= max; ++i) {
    while(--bin[i] >= 0) { a[k++] = i; }
  }
  free(bin);
}
```

このコードはどう見ても n に比例する仕事しかせず、$O(n)$ ということになります。ただし、キーの最大値 E が大きくなければですが。E が大きいと、そちらの影響がより大きくなるので、正確には $O(n+E)$ ということになるでしょうか。そしてあまり E が大きいと、そんな巨大な配列は取れませんということになるので、そもそも使えなくなります。

演習 1 ビンソートのプログラムについて、動作および時間を確認しなさい。そのうえで、次のことをやってみなさい。すべて単体テストすること。今回は全般に単体テストのコードは、前々回の `expect_sort_iarray` などを参考に、工夫して作る必要がある。

a. 「整数のみ」の整列なら上のコードでよいが、実際にはレコードの列を整列し、そのキーが整数である、ということが普通である。そのようなプログラムを作れ。(ヒント: 個々の `bin[i]` から始まる単連結リストにレコードを追加していくなどする。)
注意: 実装するとき、「安定な」整列にすること。少し気をつけるだけで安定にできるので、どうせなら非安定は避けた方がよい。

b. キーの最大値 E が大きい場合、その多くの値は使われない。そこで、E 要素の配列を作る代わりに D 個ずつまとめて $\frac{E}{D}$ 要素の配列を作り、個々の要素に「実際はどのキーがいくつ」という情報を記録させることで対応できる。そのようなプログラムを作れ。またその弱点を検討しなさい。

c. 上記の a と b の両方を行うプログラムを作れ。

12.1.2 基数ソート

基数ソート (radix sort) とは、易しく言えば「複数の整列の繰り返し」です。たとえば十進表現で 3 桁の数であれば、まず百の位が 0〜9 の順になるように並べ、その中でそれぞれ十の位が 0〜9 の順になるように並べ、さらにその中で一の位が 0〜9 の順になるように並べれば、整列が完了します。数値を何進法で (基数いくつで) 考えかに依存するため、基数ソートと呼ぶわけです。

なお、先の説明では「上の桁から」並べていましたが、「下の桁から」並べることもできます。その場合 (十進 3 桁なら)、まず一の位が 0〜9 の順になるように並べ、次に「キーが同じなら元の順番を崩さずに」十の位が 0〜9 の順になるように並べ、次に「キーが同じなら元の順番を崩さずに」百の位が 0〜9 の順になるように並べます。この「…」はつまり安定な整列をするということですね。

では「2 進表現で」「上の桁から」扱うコードを例に示します。2 進であれば各桁は「0 か 1」なので、クイックソートの時と同じようにして「ある桁が 0 の値を配列の前半に集める」ことができます。外から呼ばれる `radixsort2` は整列範囲と「どのビット」を表すマスク値 (整数の 32 ビットのうち 1 ビットだけが 1 であとが 0 の値、負の数は扱わないとしたので符号ビットの次の 30 ビット目が 1 の値から始めます) を指定して再帰手続き `rs` を呼ぶだけです。

```
// radixsort.c --- radixsort (upper->lower) from specified mask
static void swap(int *a, int i, int j) {
  int x = a[i]; a[i] = a[j]; a[j] = x;
}
static void rs(int *a, int i, int j, int mask) {
  if(i >= j || mask == 0) { return; }
  int k, s;
  for(s = k = i; k <= j; ++k) {
    if((a[k]&mask) == 0) { swap(a, s++, k); }
  }
  rs(a, i, s-1, mask/2); rs(a, s, j, mask/2);
}
void radixsort(int n, int *a) { rs(a, 0, n-1, 0x40000000); }
```

rs ですが、範囲の長さが 1 以下か、マスクが 0 (最下位まで到達) なら何もせず戻ります。次は変数 s を左端の添字とし、配列 a の範囲内の k について、a[k] とマスクのビット毎 AND 演算をおこない、0 であれば swap を使って a[k] を s の位置に置き、s を 1 増やします。そうすれば、最後には s の手前が「その桁が 0 の値」、以後が「その桁が 1 の値」となります。その後、「その桁が 0」「その桁が 1」それぞれの範囲について、さらに 1 つ下の桁での並べ替えを自分自身を再帰呼び出しして行わせます。1 つ下の桁ということは、マスクを 2 で割ればよいわけです。

演習 2 基数ソートの例題を動かし、動作と時間を確認しなさい。最大が 9999 ならマスクをもっと小さな値からにできるので、それも検討すること。その後、次のことをしてみなさい。すべて単体テストすること。

a. 2 進表現で「下の桁から」並べるプログラムを作ってみる (例題の方法で前半と後半に分けた場合は安定にはならないので注意)。
b. 単語リスト /usr/share/dict/words から長さ 10 文字の文字列を抜き出し [1]、それを基数ソートで (つまりこの場合「数字」は a〜z となる) 整列してみなさい。上の桁からやっても下の桁からやってもよい。
c. 同様だが大文字と小文字の両方が含まれる場合でやりなさい。大小順は「$a < A < b < B < \cdots$」となること。[2]
d. 同様だが長さ 1 文字以上任意でやりなさい [3]。上の桁から/下の桁からのいずれでもよい。両方試せるとなおよい (後ろが共通の語を調べるニーズもあるので下の桁からも有用)。

12.2 探索と探索アルゴリズム

12.2.1 表と探索の定式化

コンピュータ科学的には、**表** (table) という用語にはおおむね 2 通りの意味があります。1 つ目は日常生活でいう表に近いもので、次のように定式化されます。

- 表は**レコード**の集まりで、各レコードは複数の**フィールド**から成る。1 つの表の中ではすべてのレコードは同じフィールド群を持つ。

そしてもう 1 つは、上記をより抽象化したもので、次のように定式化されます。

- 表は**鍵** (key) とそれ対応する**値** (value) を格納する抽象データ型であり、鍵と値を指定して値を格納する操作と、鍵を指定して格納した値を取り出す (または格納されていなければそのことを知らせる) 操作を持つ。

データベースやデータ処理では 1 番目の意味での表が使われます。ただし、2 番目の定義であっても、その「値」が複数のフィールドを持つレコード値であってよいので、2 番目の定義のもとにデータ処理を扱うことも何ら問題なくできます。そして 2 番目にある「鍵を指定して値を格納/取り出す」操作のことを**表の探索** (table lookup) と呼び、多くのアルゴリズムが研究されています。以下ではこれらについて取り上げて行きます。

ところで、1 番目の定義の場合、探索はどうなのでしょうか。もちろん、データが多量にある場合、その効率的な処理は重要です。そこで、フィールドのうちで探索に使うものについては、2 番目の意味でいう探索アルゴリズムを実装するデータ構造を追加します。これを**索引** (index) と呼びます。複数のフィールドに対して索引をつけることもあります。ということで、探索アルゴリズムはいずれの場合でも重要なわけです。

---

[1] 「egrep ^[a-z]{10}$ ファイル >出力ファイル」でできます。
[2] 「egrep ^[a-zA-Z]{10}$ ファイル >出力ファイル」
[3] 「egrep ^[a-zA-Z]+$ ファイル >出力ファイル」

12.2.2 線形探索

ここでもスタック等のときと同じように構造体を使って情報隠蔽で実装コードを記述することにして、まず呼び出し API を定めます。ここでは簡単のため、鍵も値も整数値であるものとします。外から呼び出す操作は最初に構造を割り当てる itbl_new、鍵を指定して値を格納する itbl_put、鍵を指定して検索し値を返す itbl_get の 3 つです (検索して見付からなかったときは -1 を返すことにします)。

```
// itbl.h --- int table api.
struct itbl;
typedef struct itbl *itblp;
itblp itbl_new();                       // allocate new tbl
void itbl_put(itblp t, int k, int v); // store value
int itbl_get(itblp t, int k);         // obtain value
```

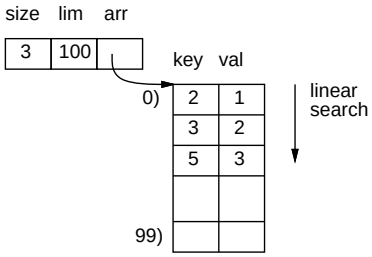


図 12.2: 線形探索の表

では一番シンプルな実装として、表の 1 つの項目を構造体で表し、そのフィールドとして鍵と値が格納されている、という形のものを作ります (図 12.2)。新しい値を追加するときは配列の末尾に入れます (配列が満杯になったら倍の大きさの配列を作って値をコピーします)。値の検索は上から順に鍵の一致する項目を探します。これを**線形探索** (linear search) と呼びます。探したり配列を拡張する作業は下請けの関数を呼ぶようにすることで、本体の関数が複雑にならないようにしています。

```
// itbl.c --- itbl impl with arry of records.
#include <stdlib.h>
#include "itbl.h"
typedef struct ent { int key, val; } * entp;
struct itbl { int size, lim; entp arr; };
itblp itbl_new() {
  itblp p = (itblp)malloc(sizeof(struct itbl));
  p->arr = (entp)malloc(100 * sizeof(struct ent));
  p->size = 0; p->lim = 100; return p;
}
static void enlarge(itblp p) {
  entp a = (entp)malloc(p->lim * 2 * sizeof(struct ent));
  for(int i = 0; i < p->size; ++i) { a[i] = p->arr[i]; }
  free(p->arr); p->arr = a; p->lim *= 2;
}
static entp lookup(itblp p, int k) {
  for(int i = 0; i < p->size; ++i) {
    if(p->arr[i].key == k) { return p->arr + i; }
  }
```

```
  return NULL;
}
void itbl_put(itblp p, int k, int v) {
  entp e = lookup(p, k);
  if(e != NULL) { e->val = v; return; }
  if(p->size + 1 >= p->lim) { enlarge(p); }
  p->arr[p->size].key = k; p->arr[p->size++].val = v;
}
int itbl_get(itblp p, int k) {
  entp e = lookup(p, k); return e == NULL ? -1 : e->val;
}
```

ではこれを使ってみる例として「素数を鍵、それが何番目の素数かを値」とするような表を作ってみます。isprime はおなじみ素数判定で、regist が指定された最大値までの範囲で「素数、何番目」を表に入れていきます (何個入れたかを値として返す)。main ではまず値を登録し、次にコマンド引数で渡されたそれぞれの整数について、表の検索結果を返します。

```
// itbldemo.c --- register primes with ranks.
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <math.h>
#include "itbl.h"

bool isprime(int n) {
  int lim = (int)sqrt(n);
  for(int i = 2; i <= lim; ++i) { if(n % i == 0) { return false; } }
  return true;
}
int regist(itblp t, int lim) {
  int r = 0;
  for(int i = 2; i <= lim; ++i) { if(isprime(i)) { itbl_put(t, i, ++r); } }
  return r;
}
int main(int argc, char *argv[]) {
  itblp t = itbl_new();
  printf("max rank = %d\n", regist(t, 10000));
  for(int i = 1; i < argc; ++i) {
    int k = atoi(argv[i]); printf("%d: %d\n", k, itbl_get(t, k));
  }
  return 0;
}
```

実行例を示します。3 は (もちろん)2 番目の素数、97 は 25 番目の素数と分かります。

```
% gcc8 itbldemo.c itbl.c -lm  ← sqrt のため -lm 指定必要
% ./a.out 3 97 100
max rank = 1229              ← 10000 までに素数は 1229 個
3: 2
```

```
97: 25
100: -1
```

では整列に引き続き、時間計測もおこなう単体テストを作ってみます。単体テストなのでデータはチェックの簡単な「鍵の値+1」を登録することにしました。データの件数はコマンド引数で指定します。このプログラムは itbl.h にプロトタイプ宣言が含まれている抽象データ型をテストするので、さまざまな実装と一緒にしてそのままテストできます。最初に指定された個数の乱数を配列に入れ、まず配列のそれぞれの値を鍵として値 (鍵+1) を登録し、次に検索してそれぞれの時間を計測します。最後に再度それぞれの鍵で検索して結果が合っているか確認します。

```
// test_itbl.c --- unit test for itable.
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "itbl.h"
void expect_itbl(int n) {
  struct timespec tm1, tm2, tm3;
  itblp t = itbl_new();
  int *a = (int*)malloc(n * sizeof(int));
  for(int i = 0; i < n; ++i) { a[i] = rand(); }
  clock_gettime(CLOCK_REALTIME, &tm1);
  for(int i = 0; i < n; ++i) { itbl_put(t, a[i], a[i]+1); }
  clock_gettime(CLOCK_REALTIME, &tm2);
  for(int i = 0; i < n; ++i) { itbl_get(t, a[i]); }
  clock_gettime(CLOCK_REALTIME, &tm3);
  int c = 0;
  for(int i = 0; i < n; ++i) {
    int v = itbl_get(t, a[i]);
    if(v == a[i]+1) { continue; }
    if(++c < 5) {
      printf(" NG: #\%d get[%d] == %d, expected %d\n", i, a[i], v, a[i]+1);
    } else if(c == 5) {
      printf("more wrong value omitted.\n");
    }
  }
  double dt1 = (tm2.tv_sec-tm1.tv_sec) + 1e-9*(tm2.tv_nsec-tm1.tv_nsec);
  double dt2 = (tm3.tv_sec-tm2.tv_sec) + 1e-9*(tm3.tv_nsec-tm2.tv_nsec);
  printf("%s size=%d tget=%.4f tput=%.4f %s\n", c==0?"OK":"NG", n, dt1, dt2);
  free(a);
}
int main(int argc, char *argv[]) {
  srand(time(NULL));
  for(int i = 1; i < argc; ++i) { expect_itbl(atoi(argv[i])); }
  return 0;
}
```

性能はどうでしょうか。線形探索では、表の項目数が n のとき、見付かるとすれば平均して半分の項目と鍵を比較する必要があり、見付からない場合は全部の項目と鍵を比較する必要があります。時

間計算量としては $O(n)$ となりますね。ここでは n までの範囲の素数について鍵を登録した後、、その範囲の全部の整数について検索して時間を計測してみます (ということは計測回数の n 回を掛けて全体では $O(n^2)$ になるはずです。上のテストを動かしてみます。

```
% gcc8 test_itbl.c itbl2.c
% ./a.out 1000
OK size=1000 tget=0.0012 tput=0.0012 OK
% ./a.out 10000
OK size=10000 tget=0.1207 tput=0.1204 OK
% ./a.out 100000
OK size=100000 tget=12.0635 tput=12.0559 OK
```

確かに、登録数が 10 倍になると所要時間は登録も検索も 100 倍になっています。

12.2.3 2 分探索

上の表では鍵 (素数) が小さい方から並んでいるため、端から順に探さなくても、中央にある値と比べることで、求める鍵が表の前半分にあるか後ろ半分にあるかが分かります。これを繰り返すことで探す範囲を半分ずつにしていき、最後は範囲の長さが 1 になって見付かる (か、または表に入っていないことが分かる) はずです。このアルゴリズムを **2 分探索** (binary search) と呼びます (図 12.3)。

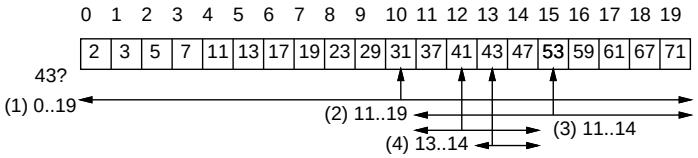


図 12.3: 2 分探索

2 分探索では、範囲 n を半分ずつにしていって長さ 1 になるまでに探索が終わるので、1 回の探索に掛かる時間計算量は $O(\log n)$ になります。ただし、項目が鍵の昇順に並んでいるという条件が必要になるので、その手間も考慮すべきです。表が完成したら変更しないというのであれば、これまでに学んで来た $O(n \log n)$ のアルゴリズムで整列すればよいのですが、1 個値を追加するごとに適切な位置に挿入すると考えるなら、それは挿入ソートと同じであり、値の追加にも $O(n)$ が必要となります。この様子を整理した表を示しておきます。一般には挿入回数より検索回数が多いので、挿入にコストを掛けても引き合うことは多いです。

	線形探索の表	2 分探索の表
挿入操作	$O(1)$	$O(n)$
検索操作	$O(n)$	$O(\log n)$

では実装を見てみましょう。ヘッダファイルは変更せず、実装だけ差し替えています。`itbl_new` や `enlarge` も変更しません。`lookup` を 2 分探索を用いた `lookup2` に変更しますが、こちらは範囲を指定するパラメタが必要です。また、`itbl_put` で値を追加するときは、末尾に追加したあと `shiftup` を呼んで適切な位置までその項目を移動しています。たまたま鍵の昇順に追加していけば、`shiftup` のループは実行されず、$O(1)$ で追加が終了します。

```
// itbl2.c --- itbl impl with sorted arry of records.
(途中略)
static entp lookup2(itblp p, int k, int i, int j) {
  if(i > j) { return NULL; }
  int m = (i + j) / 2;
```

```
    if(p->arr[m].key == k)     { return p->arr + m; }
    else if(p->arr[m].key > k) { return lookup2(p, k, i, m-1); }
    else                       { return lookup2(p, k, m+1, j); }
  }
  void shiftup(itblp p) {
    entp a = p->arr;
    for(int i = p->size-1; i > 0 && a[i-1].key > a[i].key; --i) {
      struct ent x = a[i-1]; a[i-1] = a[i]; a[i] = x;
    }
  }
  void itbl_put(itblp p, int k, int v) {
    entp e = lookup2(p, k, 0, p->size-1);
    if(e != NULL) { e->val = v; return; }
    if(p->size + 1 >= p->lim) { enlarge(p); }
    p->arr[p->size].key = k; p->arr[p->size++].val = v; shiftup(p);
  }
  int itbl_get(itblp p, int k) {
    entp e = lookup2(p, k, 0, p->size-1);
    return e == NULL ? -1 : e->val;
  }
```

先と同じ単体テストをやってみましょう。次のように、登録時間はやはり $O(n^2)$ ですが、検索は線形探索よりもずっと高速です。

```
% gcc8 test_itbl.c itbl2.c
% ./a.out 1000
OK size=1000 tget=0.0014 tput=0.0002 OK
% ./a.out 10000
OK size=10000 tget=0.1193 tput=0.0018 OK
% ./a.out 100000
OK size=100000 tget=11.6172 tput=0.0274 OK
```

演習 3 線形探索と 2 分探索の例題をひととおり動かし、動作を確認しなさい。動いたら、次のことをやってみなさい。すべて単体テストすること。

a. 例題では 2 分探索が再帰関数で実現されているが、再帰を使わない版に書き換えて動作を確認しなさい。

b. より多くのフィールドを持つ構造体の配列に対して探索が行えるような API を設計し実装しなさい。

c. その他、自分独自の (特定の使い方をした時に有利な) 改良をおこない、その効果を確認しなさい。

12.2.4 ハッシュ表

前節まで見て来たように、2 分探索でも探索には $O(\log n)$ を要していましたが、もっと速い $O(1)$ の方法があります。どうするかというと、たとえば鍵が整数で値が 0 ～ 9999 であれば、図 12.4 左のように、大きさ 10000 の配列を作って「鍵 i を i 番に入れる」ようにすればよいのです (i 番の場所に i に対応する値を入れるのなら、鍵を格納するフィールドは実際はなくてもよい)。実はビンソートもまさにこれと同様のことをやっています。

ただし問題は、この方法は鍵の範囲が広いと使えない、ということです (ビンソートも同じ問題がありました)。今日のコンピュータではサイズ百万の配列は作れますが、もう 1 桁多いとだいぶ無理があります。

ここで、鍵の範囲が広い場合でも、その範囲の全部の鍵が使われることはない (むしろ範囲が広くても、扱う実際に扱うデータの数はそれよりずっと少ないのが普通)、ということに着目します。そこで、鍵の値 k を受け取る関数 $h(k)$ を定義し、その結果があまり大きくない (たとえば $0 \sim 9999$) 範囲にします。そして、その範囲の配列を作り、鍵と値を入れるわけです。

put でも get でも、同じ k を指定すれば $h(k)$ も同じになりますから、その値の場所をアクセスすればよいわけです (図 12.4 右)。これが**ハッシュ表** (hash table) の原理で、関数 $h(k)$ のことを**ハッシュ関数** (hash function) と呼びます。ハッシュ表も上の方法と同様、ハッシュ関数の計算、配列アクセスとも一定時間ですから、$O(1)$ で登録や検索が行えます。

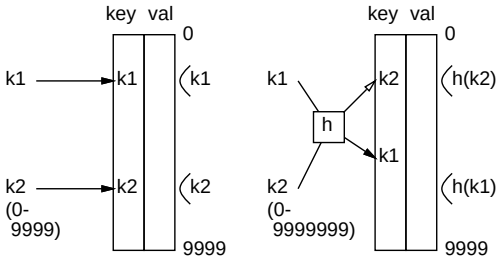


図 12.4: 配列の直接使用とハッシュ表

ただし今度は、広い範囲の鍵を一定範囲に「折り畳む」ため、異なる鍵 k_1、k_2 について $h(k_1) = h(k_2)$ となってしまう場合があります。これを**衝突** (collision) と呼びます。衝突を減らすためには、ハッシュ関数はできるだけ「ランダムに」値域内に値をばらまくことが望まれますが、それでも衝突は避けられません。

衝突に対応する方法は次の 2 つの方向があります。

- 連結リストなどを使い、1 つの場所に複数の値が入れられるようにする。
- 衝突が起きた場合、片方を別の場所に入れる。

ここでは余分なデータ構造がいらないという利点を持つ後者について取り上げます。「別の場所」をどのように定めるかがポイントです。たとえば「次の場所に入れる」だと、衝突が増えると「鍵が入っている場所のかたまり」ができてしまい、そこに別の鍵が衝突しやすくなります。

では、定数 d を決めて d だけ先にしてはどうでしょうか。そうしても、$h(k)$ が同じになる (衝突する) 鍵が複数あると、それが全部この「d 飛びの場所」に並ぶので、衝突が増えるとその連鎖が長くなります。

そこで、もう 1 つ別のハッシュ関数 $h2(k)$ を用意し、衝突が起きたら $d = h2(k)$ として d だけ先の場所に入れるのはどうでしょうか。別の関数なので、最初のハッシュ関数で衝突が起きても、$h2(k)$ は異なる値になるため、「次の場所」はそれぞれ異なるのですぐに入れる場所が見つかります。これを**ランダムリハッシュ**(random rehash) と呼びます。

では実装を見てみましょう。本体が構造体の配列なのはこれまでと変わりませんが、最初にすべてのエントリの鍵を -1(空いているという印) で初期化します。

```
// hashtbl.c --- itbl impl w/ random rehashing.
#include <stdlib.h>
#include "itbl.h"
typedef struct ent { int key, val; } *entp;
struct itbl { int size; entp arr; };
```

```
#define INITSIZE 999983
#define REHASH 97
itblp itbl_new() {
  itblp p = (itblp)malloc(sizeof(struct itbl));
  p->arr = (entp)malloc(INITSIZE * sizeof(struct ent));
  p->size = INITSIZE;
  for(int i = 0; i < p->size; ++i) { p->arr[i].key = -1; }
  return p;
}
```

ではハッシュ関数から見ていきます。h1 は単に表のサイズで剰余を取るだけです。後で示す理由により、表のサイズは素数でなければいけません。h2 は適当な素数で剰余を取ってから 3 を足しています。この 3 を足す理由ですが、運が悪くて h2 の結果が 0 になると「次の場所」に行けなくなるため、正の数になるようにしています。

```
static int h1(itblp p, int n) { return n % p->size; }
static int h2(int n) { return n % REHASH + 3; }
void itbl_put(itblp p, int k, int v) {
  int i = h1(p, k), d = h2(k), c = 0;
  while(p->arr[i].key != k && p->arr[i].key != -1) {
    if(++c > p->size) { return; }
    i = (i + d) % p->size;
  }
  p->arr[i].key = k; p->arr[i].val = v;
}
int itbl_get(itblp p, int k) {
  int i = h1(p, k), d = h2(k), c = 0;
  while(p->arr[i].key != k) {
    if(++c > p->size || p->arr[i].key == -1) { return -1; }
    i = (i + d) % p->size;
  }
  return p->arr[i].val;
}
```

次は、put から見てみましょう。まず h1 で場所を計算し、そこが「そのキーの場所であるか、空いている」ならすぐ入れればよいですが、そうでなければ次の場所へ行くので「キーが一致せず -1 でもない間繰り返す」ループになっています。ループの中でカウンタを増やしてチェックしていますが、これは表が満杯のときはいくら探しても一致も空きもないので、表のサイズ回やってだめならあきらめるためです。しかし、d 飛びに見ていったら空きがあっても元の場所に戻って以後同じ場所を繰り返し見るだけにならないでしょうか? それは、表のサイズを素数にしておけば d との最大公約数が 1 なので全部の場所を見終わるまでは元の位置に戻ることがな、というふうにして防ぐわけです。get の方も基本的に同様ですが、こちらは -1 があったら「無い」と分かるという点が違っています。

では、計測をしてみましょう。確かに $O(1)$ な感じです (しかも速い)。ただし、最後の 1000000 は少し遅くなっています。それは下の演習で。

```
% gcc8 test_itbl.c hashtbl.c
% ./a.out 1000
OK size=1000 tget=0.0000 tput=0.0000 OK
% ./a.out 10000
```

```
OK size=10000 tget=0.0003 tput=0.0003 OK
% ./a.out 100000
OK size=100000 tget=0.0037 tput=0.0030 OK
% ./a.out 1000000
OK size=1000000 tget=0.2262 tput=0.2108 OK
```

演習 4 ハッシュ表のコードを動かし、動作を確認しなさい。動いたら、次のことをやってみなさい。

a. 例題のコードでは「削除」の機能が無い。鍵を削除できるようにしてみなさい。(ただ鍵を -1 に戻すだけではまずい。その理由も検討し、対策を考え実装すること。)

b. 表のサイズが素数でないと表にあきがあるのに登録ができなくなることがある。その現象を確認しなさい。

c. ランダムリハッシュ法では、表が満杯に近付くほど登録も検索も遅くなる。その現象を確認しなさい。

d. 上記の問題を解決するには、表の充填率が一定値 (8 割とか) を超えたら表のサイズを大きくして作り直す方法がある。この方法を実装し、動作を確認しなさい。

本日の課題 12A

「演習 1」〜「演習 4」で動かしたプログラム 1 つを含むレポートを本日中 (授業日の 23:59 まで) に提出してください。

1. sol または CED 環境で「`/home3/staff/ka002689/prog19upload 12a ファイル名`」で以下の内容を提出。
2. 学籍番号、氏名、ペアの学籍番号 (または「個人作業」)、提出日時。名前の行は先頭に「@@@」を付けることを勧める。
3. プログラムどれか 1 つのソースと「簡単な」説明。
4. レビュー課題。提出プログラムに対する他人 (ペア以外) からの簡単な (ただしプログラムの内容に関する) コメント。
5. 以下のアンケートの回答。

   Q1. 整数の性質を利用した整列方法について分かりましたか。

   Q2. 線形探索、2 分探索、ハッシュ表について理解しましたか。

   Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

＃13　2分木と多分木

今回は次のことが目標となります。

- 2 分木、2 分探索木とバランスのためのアルゴリズムについて知る
- 多分木と B 木について知る

13.1　2 分探索木とバランス

13.1.1　2 分探索木

前回、配列を用いた表のデータ構造を取り扱いましたが、配列は基本的に値を「詰めて並べて」しまうため、キーの昇順/降順に並べておこうとすると挿入/削除時にその時点のサイズを n として $O(n)$ の時間計算量が掛かります。今回は動的データ構造を用いて上記の問題を克服するという話題を扱います。なお、2 分木については最大ヒープのところで完全 2 分木の配列表現を学びましたが、今回は配列表現ではなく普通の動的データ構造として木構造を扱います。

動的データ構造ではポインタを用いてノード (node, 節) を結びつけますが、今回は木なので「子」の節が複数あります。まず 2 分木、つまり子が最大で 2 つある木構造を扱います。図 13.1 左のように、個々の節には (整数を鍵・値とする表の例題なので) `key`、`val` というフィールドに加え、子へのポインタを格納する `left`、`right` があります。子の節もまた木構造になっているので、これらは左側と右側の**部分木** (subtree)、とも呼びます。ポインタ値が `NULL` であれば、その部分木は空ということになります。

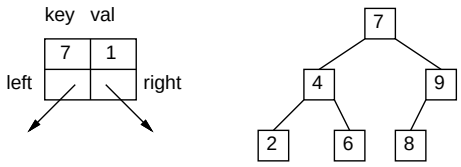


図 13.1: 2 分探索木の構造

ここで、2 分木の構造全体が **2 分探索木** (binary search tree) であるとは、次の条件が成り立っていることを言います。

> 任意の節 n について、n に格納された鍵 k に対し、n の左の部分木には k より小さい鍵のみが格納され、n の右の部分木には k より大きい鍵のみが格納されている。

図 13.1 右には、2 分探索木の例が示してあります。たとえば「7」の左の部分木には「2、4、6」が入っていて、これらは確かに「7」より小さいです。さらに、「4」の右の部分木には「6」だけがあり、これは確かに「4」より大きいです。

13.1.2　2 分探索木を用いた表の実装

2 分探索木をどのように用いて表の機能を実現するのかは、前節の説明でほぼ自明だと思います。探したい鍵の値を持って木の根からはじめ、今見ている節の鍵と一致すれば「見付かった」ことになります。そうでない場合は、今見ている節の鍵と探したい鍵の大小関係に応じて左または右の部分木へ行けばよいわけです。見付かる前に `NULL` に遭遇した場合は「無い」と分かります。

API は同様なので itbl.h を再掲しますが、末尾に項目を削除する itbl_del と現在の中身を打ち出す itbl_pr という関数を追加しました (削除は当面コメントアウト)。

```
// itbl.h --- int table api.
struct itbl;
typedef struct itbl *itblp;
itblp itbl_new();                       // allocate new tbl
void itbl_put(itblp t, int k, int v); // store value
int itbl_get(itblp t, int k);         // obtain value
//void itbl_delete(itblp t, int k);     // delete key/value
void itbl_pr(itblp t);                // print contents
```

実装を見ます。節の構造体 struct ent は先に示した通りです。して表を表す構造体 struct itbl は根の節へのポインタだけを持っています (無駄のようですが、空の表を表すにはこうする必要)。なので、空の表を作るのは簡単です。そのあと、今回は itbl_get から読みます。根をパラメタとして下請けの再帰関数 get を呼びます。get では、渡されたポインタが NULL なら「無い」ので-1 を返します。キーが一致するなら、対応する値を返します。それ以外は…探しているキーと現在の節のキーの大小に応じて、左または右の子のポインタを持って自分を再帰呼び出しします。

```
// bstree.c --- itbl impl with binary search tree.
#include <stdlib.h>
#include <stdio.h>
#include "itbl.h"
typedef struct ent *entp;
struct ent { int key, val; entp left, right; };
struct itbl { entp root; };
itblp itbl_new() {
  itblp p = (itblp)malloc(sizeof(struct itbl));
  p->root = NULL; return p;
}
static int get(entp p, int k) {
  if(p == NULL) { return -1; }
  if(k == p->key) { return p->val; }
  return get((k < p->key) ? p->left : p->right, k);
}
int itbl_get(itblp p, int k) { return get(p->root, k); }
```

では itbl_put の方を見ましょう。これも下請けの get を呼び、再帰的にキーに対応する探して行きますが、今度は渡して行くパラメタの方が entp*つまりポインタのポインタになっています。なぜかというと、新しい節を追加する場合には「ポインタの根元を書き込む」必要があるからです。そして、その場所に入っているのが NULL であれば、新しい節を割り当ててそこへのポインタをその場所に書き込み、あとのデータを初期化します。それ以外の場合は探すので先の put と同様ですが、見つかったときはそこに値を設定します。

```
static void put(entp *p, int k, int v) {
  if(*p == NULL) {
    entp q = *p = (entp)malloc(sizeof(struct ent));
    q->left = q->right = NULL; q->key = k; q->val = v;
  } else if(k == (*p)->key) {
```

```
    (*p)->val = v;
  } else {
    put((k < (*p)->key) ? &((*p)->left) : &((*p)->right), k, v);
  }
}
void itbl_put(itblp p, int k, int v) { put(&(p->root), k, v); }
```

最後に内容表示を見ましょう。itbl_pr はかっこで囲んで根に大して下請けの再帰手続き pr を呼びます。pr では自分の内容を表示しますが、ただし左や右の子が NULL でないならそれらも (自分の表示の前/後に) かっこで囲んで出力します (出力そのものは pr を再帰呼び出しして行います)。

```
static void pr(entp p) {
  if(p->left != NULL) { printf("("); pr(p->left); printf(") "); }
  printf("%d:%d", p->key, p->val);
  if(p->right != NULL) { printf(" ("); pr(p->right); printf(")"); }
}
void itbl_pr(itblp p) {
  if(p->root != NULL) { printf("("); pr(p->root); printf(")\n"); }
}
```

では、これらを使って木構造のようすを見るためのプログラムを示します。コマンド引数として値を指定すると、それらを鍵として次々に表に登録します (値は何でもいいのですが「コマンド引数の何番目か」の値にしました。マイナスの値なら削除ということにしますが、まだ削除は実装していないのでコメントアウトしています。

```
// treedemo.c --- demonstration of bstree operation
#include <stdlib.h>
#include <stdio.h>
#include "itbl.h"

int main(int argc, char *argv[]) {
  int n = atoi(argv[1]);
  itblp p = itbl_new();
  for(int i = 1; i < argc; ++i) {
    int v = atoi(argv[i]);
    if(v >= 0) { itbl_put(p, v, i); }
//  else       { itbl_delete(p, -v); }
    itbl_pr(p);
  }
  return 0;
}
```

では、実行のようすを見てみましょう。次々に節が追加されていくようすを図 13.2 に示します。なお、所要時間を計測してみたい場合は前回の計測プログラムを使えばよいです。

```
% gcc8 bstreedemo.c itbl3.c
% ./a.out 5 2 1 7 6 9
(5:1)
((2:2) 5:1)
(((1:3) 2:2) 5:1)
```

```
(((1:3) 2:2) 5:1 (7:4))
(((1:3) 2:2) 5:1 ((6:5) 7:4))
(((1:3) 2:2) 5:1 ((6:5) 7:4 (9:6)))
```

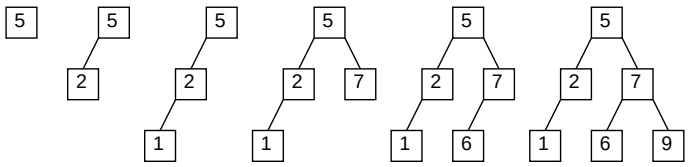


図 13.2: 2 分探索木ができていく様子

演習 1 2 分探索木による表の実装を動かし動作を確認しなさい。OK なら以下をやってみなさい。

a. 書き込むところで「ポインタのポインタ」を使うと、コードはコンパクトですが分かりにくい点もあります。ポインタのポインタを使わないように書き換えてみなさい。
ヒント: 左や右に降りる際、再帰関数の返値を「`p->left = put(p->left, k, v)`」のように元のフィールドに入れ直すと書き換えが実現でき、ポインタのポインタを使わずに済みます。この場合、put は「NULL が渡されたら節を作ってデータを登録し、その節を返す」「NULL でなければ右ないし左の子に対して put を呼び出して返値を同じ場所に書き込み、自分の返値は元の節をそのまま返す」となります。

b. 探索 (get) のとき、再帰を使わない方が分かりやすいという人もいるかと思います。再帰を除去してループに書き直してみなさい。

c. 書き込み (put) のときも同様にループに書き直してみなさい。

d. 現在の表示方法は 1 行におさまるうちは見やすいですが、データが多くなると見づらいかもしれません。データが多くなっても見やすい表示方式を考えて実装してみなさい。

13.1.3 2 分探索木からの削除

値の検索と追加は前述のように比較的簡単でしたが、削除には面倒なところがあります。削除したい値をまず探しますが、その値が木のどこに位置していたかで「面倒さ」が違います。

- 削除したい値が葉の位置にあったなら、その節を削除する (親の節からその節に至るポインタを `NULL` にする) だけで OK です。
- 削除したい値が途中の節にあったとしても、その左か右の子へのポインタが `NULL` であれば、線形リストと同じことで、親の節から直接子の節を指すように変更すればよい (図 13.3 左)。
- 一番面倒なのは、両方に子がある場合。削除した鍵の位置にどれかの鍵を持って来る必要があるが、2 分探索木の条件を崩さないようにするため、持って来るのは「左の部分木で最大の鍵」または「右の部分木で最小の鍵」となる (どちらかは任意に決めてよい)。

最後の場合についてもう少し説明しましょう。「左の部分木で最大の鍵」を用いるとして、最大の鍵ということは左部分木を行けるだけ右へたどった所にある鍵です。そこが葉ならその節を削除して持って来るだけですが、その節に左の子があるなら、左の子へのポインタを持って来る節へのポインタが入っていた箇所に格納することで木に残します。ですが、葉の場合は左ポインタが `NULL` ですから、左ポインタをコピーするという同じ動作で両方の場合が扱えます (図 13.3 中・右)。

演習 2 2 分探索木による表の実装に、鍵と値の対を削除する機能を追加しなさい。先の `main` 等のコメントアウトを外し、実際に木の挿入や削除のようすを表示して動作を確認すること。(「2」のように正の整数はその整数を登録するが、「-2」のように負にすると、対応する正の整数「2」を削除するようにできている。)

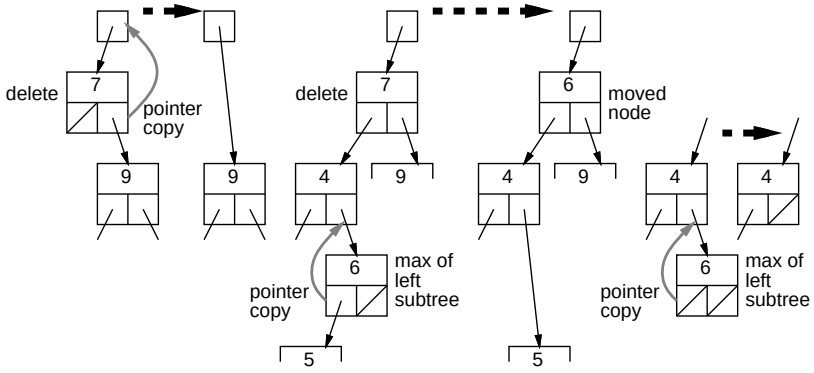


図 13.3: 2 分探索木からの鍵の削除

13.1.4 木のバランスと AVL 木

2 分探索木の (検索、追加、削除における) 時間計算量は木の高さで押えられ、木がバランスしていれば (偏りがなければ) 節数を n としたときに $O(\log n)$ となります。これは、鍵の登録時にランダムに鍵が追加されれば確かにそうなりますが、場合によっては「小さい順に」追加するプログラムもありそうですね。もしそうなると、片方向に伸びた (アンバランスな) 木ができてしまい、高さが n に比例するため、時間計算量も $O(n)$ になってしまいます。

この問題は古くから知られており、これを防ぐために「2 分探索木をバランスさせる」さまざまなアルゴリズムが考案されています。ここでは古くからある方法である **AVL 木** (AVL tree) について説明します。[1]AVL 木は、2 分探索木の条件を満たした上で、さらに次の条件を満たすような木です。

> すべての節において、左部分木の高さと右部分木の高さの差はたかだか 1 である。

確かにこの条件が満たされていれば、木はバランスしていると言えますが、それを具体的にどうやって実現するのでしょうか。それには、鍵の挿入や削除において常にこの条件を満たすように (なおかつ 2 分探索木の条件も満たすように) 木を変形するのです。

具体的にはまず、木の節数が 0 とか 1 であれば、部分木がないので (高さ 0)、確かに条件は満たされています。次に、鍵を 1 つ挿入・削除すると、挿入により高さが 1 高くなる「ことがあり」ますし、削除により高さが 1 低くなる「ことがあり」ます。1 つの挿入または削除なのですから、高さが 2 以上変化することはない、というのはよいでしょうか。

だとすれば、これまでは条件を満たしていて「挿入・削除により AVL 木の条件を満たさなくなった」時は、「高さの差が 2 になった」時です。その「2 になった」なるべく下の節 (それより下では AVL 木の条件は満たされている) に着目すると、状況は図 13.4(とその左右対称の状況) で網羅されます。

いずれのケースも書き換え矢線の左が「差が 2」の状況を表します。節の横に数字が書いてありますが、これは「右の部分木の高さから左の部分木の高さを引いた値」です (仮に「バランス値」と呼びます)。何も書いてない場合はバランス値が 0(両側の高さが同じ) です。そして左右対称のうち、図では「左の方が高い場合」を網羅しています (高さの差が 2 なのでバランス値はすべて-2 です)。

まず (A) を見ましょう。Y の左に X から始まる部分木があり、その X のバランス値は-1、0、1 のいずれかですが、そのうち-1 が (A) の場合です。先に言ってしまうと、0 は (B) であり、1 は (節の記号が着け変えてありますが)(C)、(D)、(E) のいずれかです。1 の時がなぜ 3 つかというと、左側の節のさらに右の子の節の-1、1、0 で分けてあるからです。これで-2 の場合は網羅され、2 の場合はこの左右対称のもので網羅されます。

話を戻して (A) の場合ですが、矢線で示すように、X が上でその右に Y がつながる形に変更すると (これを回転と呼びます)、X も Y もバランス値が 0 になり、木の高さは 1 減ります (よく見て確認のこと)。2 分探索木の条件については、部分木 t1 は「X より小さい値」、t2 は「X と Y の間の値」、

[1]ほかに代表的なものとして、AA 木、赤黒木、スプレー木などがあります。

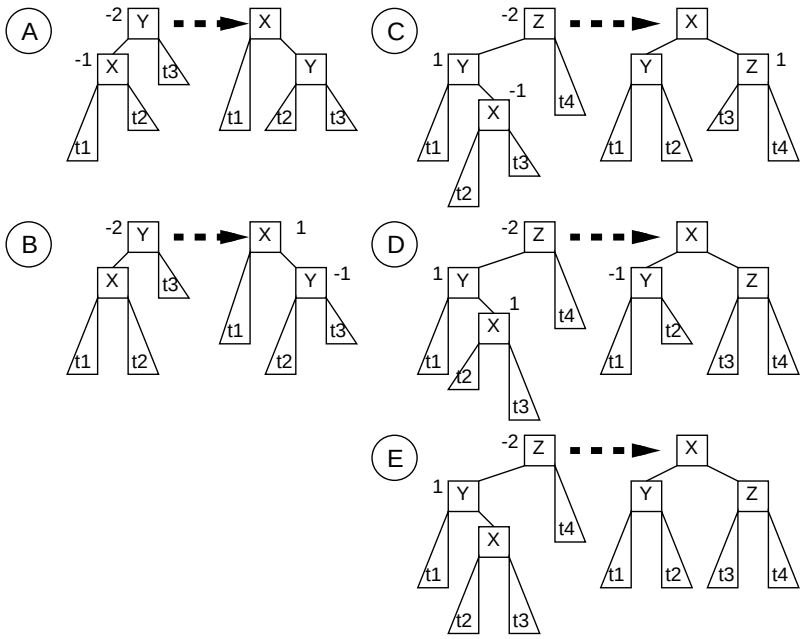


図 13.4: AVL 木のための木の回転

t3 は「Y より大きい値」から成るので、回転した後でも維持されます。(B) の場合も同じ回転をなので 2 分探索木の条件についても同じですが、回転後のバランス値は (A) と異なります。

(C)、(D)、(E) を 3 つに分けた理由は、左の節の右部分木が 1 高いので、(A) や (B) と同じ回転ではバランス値が 1〜-1 にならないためです。そこで、右部分木の根の節をさらに細かく見て t1〜t4 の 4 つの部分木に分け、X〜Z の節を図のようにつなぎ替えます (2 重回転と呼びます)。これによってバランス値が AVL 木の条件を満たすよとともに、木の高さは 1 減ります。

実際に 2 分探索木の実装に組み込む場合は、まず各節のデータとしてバランス値も保持するようにし、挿入・削除の際に正しいバランス値が維持されるうようにします。そのうえで、バランス値が-2 や 2 になったときは、回転・2 重回転を行ってバランスを回復します。

なお、挿入時には「1 高くなることで」バランスが崩れるので、回転により木の高さが 1 減って「高くなる状況は無くなる」のでそこで OK ですが、削除時には「1 低くなることで」バランスが崩れ、回転によってその部分木のバランスは回復しても、「部分木の高さが 1 低くなる」状況は同じままですから、さらに上の (親の) 部分で回転が必要になることもあります。これに対処するのは、再帰手続きであれば「子の節の挿入削除が終わった後で自分の節のバランスをチェックして必要なら回転する」という形で自然に対応できます。

演習 3 次の順番で AVL 木を実装しなさい (すごく大変だと思うのでよほどやりたい人に限る)。

a. 各節にバランス値を保持するフィールドを追加し、まず挿入に限定してすべての節でバランス値を維持するようにしなさい。`itbl_pr` で表示するときにバランス値も表示するように変更した上で正しく維持されているか確認すること。

b. 上記に加えて、バランス値が 2 と-2 になったときに回転・2 重回転をおこなってバランスが回復されるようにしなさい。実際にさまざまな順番で挿入を行い、回転が正しく行えていることをチェックすること。

c. 引き続いて、削除時のバランス値の維持機能を追加しなさい。

d. 引き続いて、削除時にも回転を行い、AVL 木の実装を完成させなさい。

13.2 多分木と B 木

13.2.1 平衡木と B 木

前節までで見てきたように、木構造では木の高さにより処理時間が決まってくるので、木のバランスを維持することが重要になります。バランスを維持する機能を組み込んだ木構造を**平衡木** (baranced tree) と呼びます。

2 分木をもとにした平衡木では、AVL 木のようにバランスが崩れたことを検出して回転によりバランスを回復したり、(本稿では述べていませんが) 乱数を用いてランダムに木を変形することで確率的にアンバランスが解消されるようにするなどの方法を取ります。

一方、多分木とは次数 (子の数) の最大が 2 より大きい木構造を言います。多文木では、その性質をうまく用いると、最初からすべての葉が同じ深さにあり、常にその性質が維持されるという形で平衡木を実現できます。その代表的なものである **B 木** (B-tree) について見てみましょう。

B 木も 2 分探索木同様、鍵を効率よく探索する機能を提供します。B 木では、ある整数 d が決められ、節に格納される鍵の個数は d〜$2d$ の範囲に制約されます (根は例外で最小が 1)。そして、節には鍵数より 1 多い子ポインタが格納され、それをたどって行くことで目的の鍵を効率よく探せます。

B 木の次数は一般に子ポインタの数 ($d+1 \sim 2d+1$) を用いて「2-3 B 木 (2-3 木)」、「3-5 B 木 (3-5 木)」のように表します (根は例外的に鍵が 1 個でもよいことに注意)。

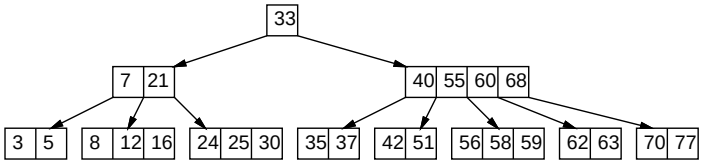


図 13.5: B 木の例 (3-5 木)

図 13.5 は 3-5 木の例です。そしてこの図のように、B 木を描くときには鍵と子ポインタを交互に描くのが分かりやすいです。子ポインタが指す部分木は、両側の鍵ではさまれた範囲の値を格納します。

たとえば図 13.5 左では、「7」「21」の間にあるポインタで指されている部分木はその間の範囲の鍵だけを持ちますし、「21」の右の部分木については、根の「33」から左にたどってきてここへ来ているので、21〜33 の範囲の鍵だけを持ちます。

この性質を用いれば、2 分探索木と同様、根から始めて特定の鍵のある方へ間違いなく降りてゆけ、木の高さだけおりてゆけば鍵が見付かる (または無いことが分かる) わけです。

なお、実用的には B 木は 2 時記憶装置上のデータ構造として多く使われます。つまり、1 つのディスクブロック (4096 バイト等) に 1 つの節を格納し、次数を大きくすることで少ない段数 (ブロック読み込み回数) で目的のデータに到達できるのです。ここでは概念の説明なので普通にメモリ上のデータ構造として扱っています。

13.2.2 B 木における鍵の追加

検索について分かったところで、鍵の追加がどのように働くかを見ます。図 13.6 も先と同様、3-5 木を表しているのとします。ここで下の節に「27」を追加しようとした場合、鍵の数が 5 になると $2d$ を超えてしまいます。そのような場合は、節全体を 2 つに分割し、中央の鍵 (この場合では「23」) を親に移します。このとき、親の節では「3」「23」の間の子ポインタが左の節、「23」「41」の間の子ポインタが右の節を指すようにすることで、B 木の性質が維持されます。

しかし最初の「27」はどこから来たのでしょうか。また親の節に「23」を追加しようとしたとき、親の節が満杯ならどうするのでしょうか。これらの答えは簡単で、まず B 木では値の追加は葉の節でのみ行います。そして、葉の節で鍵の個数が $2d$ を超えたため分割が起き、その結果図の節に「27」が挿入されようとしているわけです。

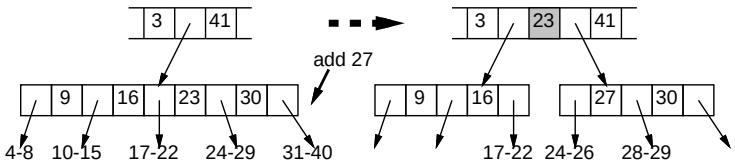


図 13.6: B 木への値の挿入

また、親の節も $2d$ を超えるならやはり分割が起き、それが上に伝播していって $2d$ で収まるところで止まります。根まで来ても $2d$ を超えるときは、根の節を 2 つに分割し、1 つの鍵と 2 つの子ポインタを持つ新しい根を作ってそこから指させます。このときは木の段数が 1 段増えます。削除については後の節で扱います。

13.2.3 例題: 2-3 木の実装

それでは例として、2-3 木の実装を見てみましょう。AVL 木と同様、削除は練習問題にするのでここでは省略しています。まず冒頭部分ですが、表は根だけを持つ構造体で、節が重要になります。

各節には葉かどうかを表すフィールド `leaf`、現在格納してる鍵の数 `nkey` と、鍵、対応する値、そして子ポインタの配列 `key`、`val`、`child` を持たせます。鍵 `key[`i`]` に対して、その左の子ポインタは `child[`i`]`、右の子ポインタは `child[`$i+1$`]` ということになります。配列のサイズは子ポインタだけ $2d+1$ であとは $2d$ でいいのですが、コードを少し簡単にするためすべて同じにしました。表の初期化は根を `NULL` に初期化するだけです。

```
// btree23.c --- itbl impl with 2-3 B-tree.
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include "itbl.h"
#define D 1
typedef struct ent *entp;
struct ent {
  bool leaf;
  int nkey, key[D*2+1], val[D*2+1];
  entp child[D*2+1];
};
struct itbl { entp root; };
itblp itbl_new() {
  itblp p = (itblp)malloc(sizeof(struct itbl));
  p->root = NULL; return p;
}
```

検索ですが、まず根が `NULL` なら木が空っぽなので何も含まれていません。それ以外の場合は `itbl_get` は根をパラメタとして下請けの `get` を呼びます。

`get` は節に含まれている鍵との一致を見て行き、一致すれば対応する値を返します。一致せず、なおかつ探している鍵 `k` より大きい鍵が現れたら、その左の子ポインタをたどって探します。どの鍵も `k` より小さく右端に来たら、最後の子ポインタを使用します。いずれにせよ、子ポインタの指す先から探すのは再帰呼び出しによって行います。葉まで来ていたら見つからなかったので -1 を返します。

```
static int get(entp p, int k) {
  int i;
```

```
    for(i = 0; k >= p->key[i] && i < p->nkey; ++i) {
      if(k == p->key[i]) { return p->val[i]; }
    }
    return (p->leaf) ? -1 : get(p->child[i], k);
  }
  int itbl_get(itblp p, int k) {
    return (p->root == NULL) ? -1 : get(p->root, k);
  }
```

さていよいよ、ややこしい挿入の処理です。下請け用として、鍵と対応する値を指定して鍵を 1 個持つ節を作る関数 newent2(左と右の子ポインタを指定)、newent(葉の節なので子ポインタは NULL)、および鍵と値を別の節の i 番目から取って来る cpent2、cpent を用意しました。また、shift はある節の i 番に新しい鍵を挿入したい場合に使用し、i 番より後ろにある鍵、値、子ポインタをすべて 1 つずつ後ろにずらして場所をあけます。

```
  static entp newent2(int k, int v, entp l, entp r) {
    entp p = (entp)malloc(sizeof(struct ent));
    p->leaf = false; p->nkey = 1; p->key[0] = k; p->val[0] = v;
    p->child[0] = l; p->child[1] = r; return p;
  }
  static entp newent(int k, int v) {
    entp q = newent2(k, v, NULL, NULL); q->leaf = true; return q;
  }
  static entp cpent2(entp p, int i, entp l, entp r) {
    entp q = newent2(p->key[i], p->val[i], l, r); return q;
  }
  static entp cpent(entp p, int i) {
    entp q = cpent2(p, i, NULL, NULL); q->leaf = true; return q;
  }
  static void shift(entp p, int i) {
    for(int j = ++p->nkey; j > i; --j) {
      p->key[j] = p->key[j-1]; p->val[j] = p->val[j-1];
      p->child[j] = p->child[j-1];
    }
  }
```

次のレベルの下請けとして、節が葉で位置 i に鍵を挿入すると決まった場合の処理を行う pleaf を示します。節を分割する場合があるので、分割した場合は上に移す 1 つの鍵と分割した 2 つの節を持つ節を作成して返すようにします (以下の pnode、put も同じ)。分割が不要なら NULL を返します。

節に入る鍵の上限までに余裕があるなら、i より後を shift でずらして空いた位置に鍵と値を挿入すれば完了で、NULL を返します。それ以外は分割が必要で、分割位置 i が 0、1、2 のそれぞれに応じて上に移す鍵、左右の節を適切に設定した節を作り返します。

```
  static entp pleaf(entp p, int i, int k, int v) {
    if(p->nkey < D*2) {
      shift(p, i); p->key[i] = k; p->val[i] = v; return NULL;
    } else if(i == 0) {
      return newent2(p->key[0], p->val[0], newent(k, v), cpent(p, 1));
    } else if(i == 1) {
```

```
    return newent2(k, v, cpent(p, 0), cpent(p, 1));
  } else {
    return newent2(p->key[1], p->val[1], cpent(p, 0), newent(k, v));
  }
}
```

中間ノードの場合は、葉の方から分割の情報が (上述のような形で) 節 q として渡されて来て、それを取り扱う下請け pnode を用意しました。まず q が NULL なら分割をしなかったのでこちらも作業はなく、NULL を返します。また格納する鍵の数に余裕があれば、shift でずらして空いた位置 i に格納すれば終わりで、これも NULL を返します。それ以外は葉の時と同じように 3 つに場合分けですが、ただしこちらは左右の子ポインタも適切に返す必要があるのでやや面倒です。

```
static entp pnode(entp p, entp q, int i) {
  if(q == NULL) { return NULL; }
  entp r = NULL;
  if(p->nkey < D*2) {
    shift(p, i); p->key[i] = q->key[0]; p->val[i] = q->val[0];
    p->child[i] = q->child[0]; p->child[i+1] = q->child[1];
  } else if(i == 0) {
    r = newent2(p->key[0], p->val[0],
                newent2(q->key[0], q->val[0], q->child[0], q->child[1]),
                cpent2(p, 1, p->child[1], p->child[2]));
  } else if(i == 1) {
    r = newent2(q->key[0], q->val[0],
                cpent2(p, 0, p->child[0], q->child[0]),
                cpent2(p, 1, q->child[1], p->child[2]));
  } else {
    r = newent2(p->key[1], p->val[1],
                cpent2(p, 0, p->child[0], p->child[1]),
                newent2(q->key[0], q->val[0], q->child[0], q->child[1]));
  }
  free(q); return r;
}
```

put は部分木に鍵と値の組を書き込み、分割が起きた場合は分割の中央の鍵と左右の子ポインタのみを持つ節を返します。その内容ですが、まず鍵の並びを調べて一致する鍵があれば、対応する値を書き込むだけで完了です。そうでない場合、変数 i は k より大きい最初の鍵の位置になっていることに注意。そして、この節が葉かどうかに応じて pleaf、pnode のいずれかを呼び出せば仕事は完了です。

```
static entp put(entp p, int k, int v) {
  int i;
  for(i = 0; k >= p->key[i] && i < p->nkey; ++i) {
    if(k == p->key[i]) { p->val[i] = v; return NULL; }
  }
  return p->leaf ? pleaf(p, i, k, v) : pnode(p, put(p->child[i], k, v), i);
}
void itbl_put(itblp p, int k, int v) {
```

```
  int k1, v1;
  if(p->root == NULL) { p->root = newent(k, v); return; }
  entp r = put(p->root, k, v);
  if(r != NULL) { free(p->root); p->root = r; }
}
```

以上の下請けが用意されているとして、itbl_put は根が NULL なら書き込む鍵と値の組を持った根ノード (葉でもある) を作成して指させます。それ以外の場合は根をパラメタとして下請けの put を呼び、返値がノードであればそれを新たな根とします (木の高さが 1 段高くなる場合)。NULL であれば根の節での分割は起きなかったのでとくにすることはありません。

最後に、B 木全体をプリントする関数も用意しました。2 分探索木と似ていますが、部分木と「鍵:値」とを交互に打ち出すように一般化されています。

```
static void pr(entp p) {
  int i;
  if(p == NULL) { printf("NULL"); return; }
  if(p->leaf) {
    if(p->nkey > 0) { printf("%d:%d", p->key[0], p->val[0]); }
    for(i = 1; i < p->nkey; ++i) {
      printf(" %d:%d", p->key[i], p->val[i]);
    }
  } else {
    for(i = 0; i < p->nkey; ++i) {
      printf("("); pr(p->child[i]);
      printf(") %d:%d ", p->key[i], p->val[i]);
    }
    printf("("); pr(p->child[i]); printf(")");
  }
}
void itbl_pr(itblp p) { printf("("); pr(p->root); printf(")\n"); }
```

では、先のテストプログラムと組み合わせて動かします。確かに、番号順に挿入しているのにすべての葉の深さは同じになるように木が作られて行きます。

```
% gcc8 treedemo.c btree32.c
% ./a.out 1 2 3 4 5 6 7 8 9 10
(1:1)
(1:1 2:2)
((1:1) 2:2 (3:3))
((1:1) 2:2 (3:3 4:4))
((1:1) 2:2 (3:3) 4:4 (5:5))
((1:1) 2:2 (3:3) 4:4 (5:5 6:6))
(((1:1) 2:2 (3:3)) 4:4 ((5:5) 6:6 (7:7)))
(((1:1) 2:2 (3:3)) 4:4 ((5:5) 6:6 (7:7 8:8)))
(((1:1) 2:2 (3:3)) 4:4 ((5:5) 6:6 (7:7) 8:8 (9:9)))
(((1:1) 2:2 (3:3)) 4:4 ((5:5) 6:6 (7:7) 8:8 (9:9 10:10)))
```

演習 4 2-3 木の例題を動かし、B 木の挿入アルゴリズムを確認しなさい。納得したら、3-5 木 (ないしもっと次数の大きい木) を実装してみなさい。(ヒント: 例題のコードは、分割処理けだけは

0/1/2 の場合に分けて扱っていますが、そこ以外は 2-3 木であることをとくに使っていないの、変更しないで大丈夫なはずです。)

13.2.4 B 木における鍵の削除

ここまでは挿入だけを扱って来ましたが、B 木における鍵の削除はどうでしょうか。削除ではどの位置の値でも削除できますが、中間の節から鍵を削除する場合は、2 分探索木と同様に「その鍵の左の子ポインタが指している部分木で最大の鍵」または「その鍵の右の子ポインタが指している部分木で最小の鍵」を持って来てそこに移す必要があります。これらは葉にあるので、いずれにせよ葉から要素を削除するという動作になるわけです。

削除の結果、鍵の数が d より少なくなる場合は隣接する節から融通します。融通する場合は、隣接する節の間にある鍵を少ない方の節に移し、代わりにその位置に多い側の節から持って来た鍵を入れます (図 13.7)。葉では子ポインタはないですが、中間の節の場合は隣接する子ポインタも対応して移すことになります。

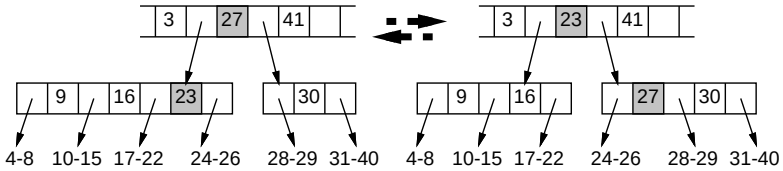


図 13.7: 隣接ノード間での鍵の融通

隣接する節の鍵の数も d で融通できないときは、2 つの節を 1 つに統合し、親から間の鍵を取って来ることで $2d$ の鍵を持つ節にします。このとき、親の節の鍵が d を下回る場合はさらに同じ処理を繰り返し、結果として根から鍵が 1 つも無くなる時は木の高さが 1 段減ります。

演習 5 2-3 木に削除操作を追加して動かしなさい。

演習 6 より次数の大きい B 木の実装について、削除操作を追加して動かしなさい。

本日の課題 13A

「演習 1」〜「演習 6」で動かしたプログラム 1 つを含むレポートを本日中 (授業日の 23:59 まで) に提出してください。

1. sol または CED 環境で「`/home3/staff/ka002689/prog19upload 13a ファイル名`」で以下の内容を提出。
2. 学籍番号、氏名、ペアの学籍番号 (または「個人作業」)、提出日時。名前の行は先頭に「@@@」を付けることを勧める。
3. プログラムどれか 1 つのソースと「簡単な」説明。
4. レビュー課題。提出プログラムに対する他人 (ペア以外) からの簡単な (ただしプログラムの内容に関する) コメント。
5. 以下のアンケートの回答。

   Q1. 2 分探索木がプログラムできるようになりましたか。

   Q2. AVL や B 木のアルゴリズムを理解しましたか。

   Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

＃14　動的計画法

今回は次のことが目標となります。

- 再帰関数のメモ化と動的計画法の関係を理解する
- 2 次元の動的計画法を使えるようになる

14.1　メモ化と動的計画法

14.1.1　再帰関数とメモ化

今回は以前やった再帰関数の中から、フィボナッチ数のコードを取り上げます。

```
int fib(n) {
  if(n <= 1) { return 1; }
  return fib(n-1) + fib(n-2);
}
```

このコードは再帰 1 段ごとに自分自身を 2 回呼ぶので、計算量が $O(2^n)$ になってしまい、大変遅いです。しかしそもそも、遅い理由というのは、同じ引数に対する関数値を何回も重複して計算するためですよね (図 14.1)。

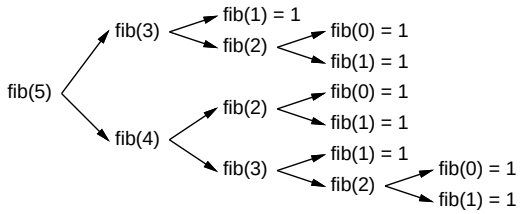


図 14.1: fib 再帰関数での計算の重複

この関数では与えられた引数に対する結果は何回計算しても同じわけですから、「最初に計算したときにそこ結果を覚えておき」「2 回目からは覚えている結果をそのまま返す」ようにすれば、同じ引数に対する計算を何回も行わなくて済みます。このような処理を (結果をメモしておくことから) **メモ化** (memoization) と呼びます。実際にやって見ましょう。

```
// fibmemo.c --- calcurate fib with memoization.
#include <stdio.h>
#include <stdlib.h>
#define ARRSIZE 100
int fib(int n) {
  if(n <= 1) { return 1; }
  return fib(n-1) + fib(n-2);
}
int fib1(int n) {
  static int memo[ARRSIZE];
```

```
  if(memo[n] != 0) { return memo[n]; }
  int r = 1;
  if(n > 1) { r = fib1(n-1) + fib1(n-2); }
  memo[n] = r; return r;
}
int main(int argc, char *argv[]) {
  int n = atoi(argv[1]);
  printf("fib(%d) = %d\n", n, fib1(n));
  return 0;
}
```

これまでの fib の方は比較用に入れてあります (時間を比較するときに利用してください)。メモ化を行った方は fib1 です。その先頭でメモ用の配列を宣言していますが、この配列は値を保管するので「ずっと存在していてくれないと」困ります (プログラムの実行中全体が存在期間)。そのような場合はローカル変数でも冒頭に static をつけます。もちろん配列を関数の外に置いてもそうなりますが、この場合関数の中だけで使うのでこの方がお行儀がよいです。なお、C 言語ではグローバル変数および static 変数は内容が 0 で初期化されることになっています。

次に、渡された引数 n の位置の memo を調べ、0 でなければ既に計算した結果が入っているので、直ちにそれを返します。これがメモ化の「後半 (値を使う方)」になります。続いて、変数 r に関数の値を計算しますが、これは直接 return で返してしまうとメモ配列に書き込むタイミングが無いためです。最後に、引数 n の位置に r を書き込むのがメモ化の「前半 (値を入れる方)」で、そのあとその r を返して終わります。実行例は当り前なので省略します。

演習 1 例題を動かし、正しく計算されていることを確認しなさい。メモ化する前の版も動かし、結果と時間を比較すること。納得したら、次の関数のメモ化版を作り、元の版と比較しなさい (あくまでもここに示した遅い再帰関数をもとにメモ化すること)。なお、最後のもののように引数が 2 つある場合は、メモ配列も 2 次元配列にする必要があることに注意。

a. 3 の n 乗

```
int pow3(n) {
  if(n <= 0) { return 1; }
  return pow3(n-1) + pow3(n-1) + pow3(n-1);
}
```

b. n の階乗

```
int fact(int n) {
  if(n <= 1) { return 1; }
  int r = 0;
  for(int i = 0; i < n; ++i) { r += fact(n-1); }
  return r;
}
```

c. ${}_nC_r$

```
int comb(int n, int r) {
  if(r == 0 || r == n) { return 1; }
  return comb(n, r-1) + comb(n-1, r-1);
}
```

14.1.2 メモ化から動的計画法へ

前節では再帰関数が呼ばれた時にそのパラメタに対する値を配列 memo に保管していました。しかし fib の場合でいうと、ある値 n に対する計算を行うためには結局 $0 \sim n-1$ すべてについて値を計算する必要があるので、最初からこれらを「小さい順に」計算してしまう方が簡単になります。これが**動的計画法** (dynamic programming, DP) の考え方です。言い替えると、動的計画法とは次のような手法です。

> ある問題に対する解が、その問題より小さいサイズの部分問題に基づいて求められる (1) 場合に、小さいサイズの問題から順に解を記録しながら解く (2) ことで元の問題に対する解を求める。

ここで (1) は分割統治手法であること、(2) はメモ化 (通常は配列への記録) を用いることを表していると言えます。なお、英語名称の dynamic programming というのは考案者がそうつけただけで、特別に dynamic でも特別に programimng でもないので注意してください。

ではさっそく、先の fib の計算を動的計画法に書き替えた版を示します。メモ配列と関数 fib2 の型が unsigned(32 ビット符号なし 2 進表現) になっていますが、これは最大の 99 まで扱うのには通常の int だとあふれてしまい負の数になるからです。また、符号なし整数の出力には printf の書式文字列で「%u」を指定します。

コードについてはほとんど説明はいらないですが、メモ配列は関数の外に出し (2 つの関数で利用するため)、まず initfib で配列に値を計算してしまい、fib2 では指定した要素を取り出して返すだけとなっています。

```
// fibdp.c --- calcurate fib with dynamic progrmming.
#include <stdio.h>
#include <stdlib.h>
#define ARRSIZE 100
static unsigned memo[ARRSIZE];
void initfib() {
  memo[0] = memo[1] = 1;
  for(int i = 2; i < ARRSIZE; ++i) { memo[i] = memo[i-1]+memo[i-2]; }
}
unsigned fib2(int n) { return memo[n]; }
int main(int argc, char *argv[]) {
  int n = atoi(argv[1]); initfib();
  printf("fib(%d) = %u\n", n, fib2(n));
  return 0;
}
```

演習 2 上のフィボナッチ数の動的計画法版を動かし、確認しなさい。納得したら、演習 1 に出て来た以下の再帰関数による計算に対し、動的計画法を用いて計算するプログラムを作ってみなさい。通常版と結果を比較して間違った計算になっていないことを確認すること。

a. 3 の n 乗
b. n の階乗
c. ${}_nC_r$ (ヒント: メモ配列は 2 次元配列にする必要あり)

14.2 2 次元の動的計画法

14.2.1 例題: 経路数問題

前節では分割統治とメモ化の組み合わせとして動的計画法を定めていましたが、もっとくだけた言い方として「全部答えが埋まるまで配列を埋めて行く」というふうにとらえた方が分かりやすいことがあります。とくに、2 次元配列を用いる場合は (2 次元は紙の上に描きやすいので) そうです。

典型例として、経路数問題を取り上げます。問題は簡単で、図 14.2 のようなます目で「S からスタートし、東 (右) か南 (下) のます目へのみ動けるものとしたとき、G に到達する経路は何通りあるか」というものです。右のバージョンでは、網掛けになっているます目は通れないものとします。

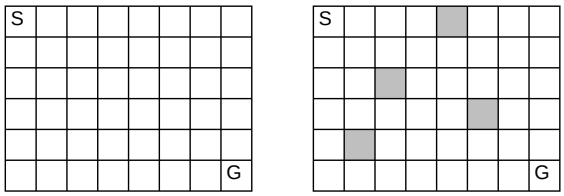


図 14.2: ゴールまでの経路数は?

この問題はどのように考えたらいいでしょうか。まず、全部通れる左の問題からです。上端の 1 列、左端の 1 列については「S からずっと横/縦に移動してくる」以外の方法がないことはすぐ分かります。なので、これらのます目では経路数は「1」です。

それ以外のます目についてはどうでしょう? あるます目に来るには「左隣のます目から 1 個右に来る」場合と「上隣のます目から 1 個下に来る」場合の 2 通りがあり、そしてこれらは (当然) 違う経路です。ということは、「左隣のます目の経路数」「上隣のます目の経路数」が分かっていれば、その 2 つを足したものが「このます目の経路数」なわけです。

前記の「1」を記入したあと、配列を上/左から順に埋めていけば、どの場所でも左/上は既に記入済みですから、上の計算は問題なくできます。プログラムを示しましょう。

```
// numpath1.c --- number of paths using DP.
#include <stdio.h>
#define W 8
#define H 6
static int map[H][W];
int main(void) {
  for(int i = 0; i < W; ++i) { map[0][i] = 1; }
  for(int j = 0; j < H; ++j) { map[j][0] = 1; }
  for(int i = 1; i < W; ++i) {
    for(int j = 1; j < H; ++j) {
      map[j][i] = map[j][i-1] + map[j-1][i];
    }
  }
  printf("%d\n", map[H-1][W-1]);
  return 0;
}
```

では動かしてみましょう。

```
% gcc8 numpath1.c
% ./a.out
792
```

792 通りだそうです。実際、図 14.3 左のように手で埋めると (あまりやりたくないですが)、確かに 792 通りです。

1	1	1	1	1	1	1	1
1	2	3	4	5	6	7	8
1	3	6	10	15	21	28	36
1	4	10	20	35	56	84	120
1	5	15	35	70	126	210	330
1	6	21	56	126	252	462	792

1	1	1	1		0	0	0
1	2	3	4	4	4	4	4
1	3		4	8	12	16	20
1	4	4	8	16		16	36
1		4	12	28	28	44	80
1	1	5	17	45	73	117	197

図 14.3: 経路数のます目を埋めた結果

実は左側の障害物のない問題は「左上から右下まで移動する 12 ステップのうち、5 ステップは下への移動、残りは右への移動」であることから、すべての場合は組合せの数 ${}_{12}C_5$ となり、確かに 792 です。ですが、図の右のように「通れない所がある」場合はそう簡単には行きませんから、そのような場合はプログラムで動的計画法を使うのがよいでしょう。あと、「斜めに右下に行ける」ことにした場合もそうですね。

演習 3 上の経路数のプログラムを動かし、結果を確認しなさい。ます目が 8x6 以外の場合も試してみること。そのあと、次のことをしなさい。

a. 図 14.2 右の網掛けの部分が通れないとした場合の S から G までの経路数を求めるプログラムを書きなさい。(ヒント: 通れない部分は「-1」などの目印を入れておき、その目印だったら加算しない。)
b. 斜めにも移動でき、障害物がない場合について経路数を求めるプログラムを書きなさい。
c. 斜めにも移動でき、障害物がある場合について経路数を求めるプログラムを書きなさい。

14.2.2 方向の決まらない場合/トレースバック

さて、先の問題 (障害物あり) をさらにおしすすめて、図 14.4 のように迷路にしたとします。今度は、迷路を最短何ステップで抜けられるかという問題にします。今度は、進む方向が右と下だけとは限りませんが、どうしたらいいでしょうか。次の疑似コードを見てください。

- S のます目に 1 を記入し、残りの (障害でない) ます目に 999 を記入
- 値が変化しなくなるまで繰り返し、
- 　すべてのます目について、
- 　　上下左右のます目 (障害物除く) の数値+1 の最小値が現在のます目の値より小さいなら、
- 　　ます目にその最小値を書き込む
- 　　ここまでが枝分かれの範囲
- 　ここまでが繰り返しの範囲
- ここまでが繰り返しの範囲

「変化しなくなるまで繰り返し」というループがありますが、その実現にはバブルソートの時と同じように旗を使い、ループの先頭で旗を立て、最小値を書き込んだら降ろすようにすれば、次のループ先頭で旗が立ったままのときは変化しなくなったことが分かる、というふうにします。

ます目の計算を行う方向が一定でなかったとしても、「最小のステップ数」を求めるのですから、上の疑似コードのように「これ以上変化しなくなるまで繰り返し」最小値を記入していけばよいのです。そして、正の整数値は無限に小さくなり続けることはできませんから、このアルゴリズムは必ず停止し、そのとき G のます目に記入されている値が最小のステップ数です (図 14.5)。なお、最初に G のます目に記入されたときに停止してはいけないことに注意。ほかの経路でもっと短いものがまだ残っている可能性がありますから。

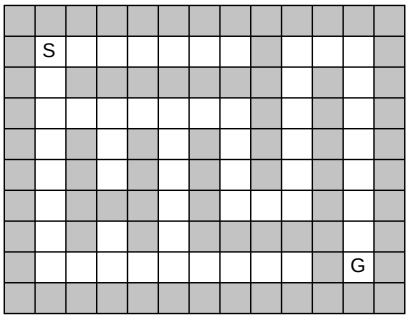


図 14.4: ます目を使った迷路

	1	2	3	4	5	6	7		19	20	21	
	2								18		22	
	3	4	5	6	7	8	9		17		23	
	4		6		8		10		16		24	
	5		7		9		11		15		25	
	6				10		12	13	14		26	
	7		11		11						27	
	8	9	10	11	12	13	14	15	16		28	

図 14.5: ます目にステップ数を記入した迷路

ところで、迷路にはもう 1 つ別の問題が残っています。それは「その最短ステップ数の経路は具体的にどこを通る経路か」も知りたい、ということです (知りたいですよね?)。

それには、次のようにします。まず、ます目を表すのとまったく同じサイズの 2 次元配列を用意します。そして、ます目に最小値を記入するとき、「上下左右のどちらから来た値が最小か」をこちらの配列に記録しておきます。そうすれば、G から逆に「記録した最小の方向を次々にたどる」ことで S までの経路が分かります。これを、目的地から逆向きにたどることから、**トレースバック** (trace back) と呼びます。

このように、動的計画法で最小値や最大値を求めたあと、具体的にどの選択を取った結果その最小/最大を達成したかを知りたい場合、各地点での選択を記録するような (トレースバックのための) データ構造が別途必要になるわけです。

演習 4 迷路の問題について次のことをおこないなさい。迷路そのものは好きな大きさ/形にしてよい。

a. 迷路の最小ステップ数を出力する動的計画法プログラムを作りなさい。
b. 上記に加えてトレースバックをおこない最短経路を表示しなさい。
c. さらに上記に加えてトレースバックを分かりやすく表示しなさい。画面に ASCII 文字を使って図を表示してもいいですし、EPS ライブラリを使ってもいいですし、いっそアニメーションを生成してもよいです。

なお、トレースバックでは「目的地から逆向きの」列が求まりますが、それを順方向に直したければ、配列なりスタックなりを使って適宜行ってください。

14.2.3 最長共通部分列

動的計画法は「2 つの列がどれくらい似ているか」を調べるのにも使えます。その具体例として、2 つの文字列の対応づけの問題を考えましょう。たとえば、「isasaka」「sassa」「wakasa」の 3 つの文字列のうち、互いに一番似ているのはどの 2 つだと思いますか。

「似ている」の定義によりますが、ここでは「2 つの文字列の同じ文字を対応させた組ができるだけ多くできる」ものが似ているものと考えます (図 14.6)。ただし、対応づけの線はクロスしてはいけないものとします。こうすると、「isasaka」「sassa」の組が一番多く (4 箇所で) 対応づけができると分かります。

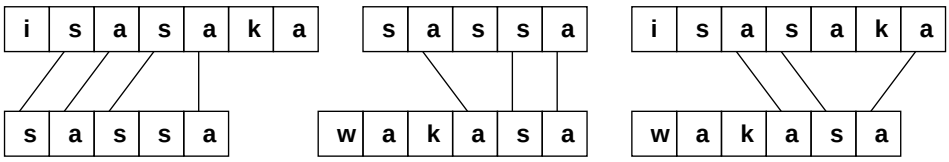


図 14.6: 最長共通部分列

なお、この問題「**最長共通部分列** (longest common subsequence, LCS) と呼ばれています。なぜなら、両方の文字列から一部を (順序を変えずに) 抜き出して来たもの (部分列) で、互いに一致するもの (共通部分列) のうち、最長のものを求めているからです。

では、これをどのようにして求めるのがいいでしょう。実は、文字列の対応づけは先にやった「ます目の経路」で表すことができます。具体的には、次のように考えます。

図 14.7 左のように、比較する 2 つの文字列それぞれに「カーソルを」対応させます。カーソルは、2 つの文字列で次の文字位置を対応づける場合は、同時に進めます。そうでない場合は、a だけ、または b だけを進めます。a の文字列の各位置と b の文字列の各位置を縦横に並べた図 14.7 中のような表を考えます。ここで、a のカーソルだけを進めることは、右のます目への移動を表し、b のカーソルだけを進めることは、下のます目への移動を表し、同時に進めることは斜め右下への移動を表します。そうすると、あらゆるカーソルの移動はます目の中の移動で表されます。

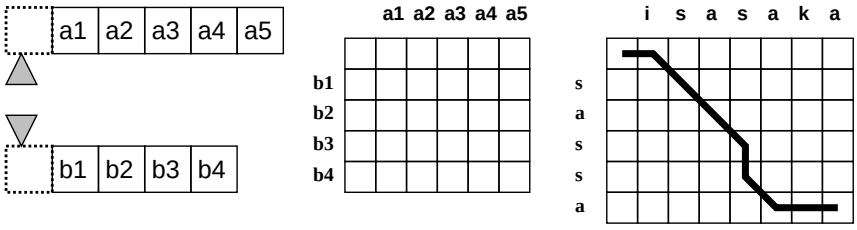


図 14.7: 文字列の対応づけの考え方

ここで LCS の問題をこの表現にあてはめると、対応づけ (斜めの移動) は対応する位置の文字どうしが等しい場合に限るという条件で、斜めの移動の最大数を求める、という問題と同等です。

そこでまず、上端/左端の列は (斜めの移動はないので) すべて 0 を入れます。その上で、それぞれのます目を図 14.8 左のように、「斜め移動できるなら斜め左上の数プラス 1、そうでないなら上または左の数値のうち大きい方」を入れる形で埋めていきます。そうすると、右下隅に入った値が対応づけられる個数の最大数となるわけです。

これを C のプログラムにしたものを示します。上に示したように配列の上端と左端を 0 に初期化し、あとは左上から順にそれぞれのます目の値を計算すれば済みます。文字列の対応が無い場合は上と左のます目のうち大きい方、文字列の対応がある場合はさらに斜め左上のます目の値+1 も加えた最大値を入れるということです。

```
// lcs.c --- longest common sequence length for two strings.
#include <stdio.h>
#include <string.h>
#define MAXSTR 50
```

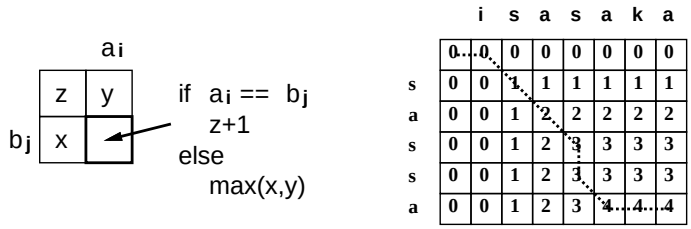


図 14.8: 動的計画法による LCS の計算

```
static int a[MAXSTR][MAXSTR];
int lcs(char *s1, char *s2) {
  int l1 = strlen(s1), l2 = strlen(s2);
  for(int i = 0; i <= l1; ++i) { a[0][i] = 0; }
  for(int j = 0; j <= l2; ++j) { a[j][0] = 0; }
  for(int j = 1; j <= l2; ++j) {
    for(int i = 1; i <= l1; ++i) {
      int m = a[j][i-1];
      if(m < a[j-1][i]) { m = a[j-1][i]; }
      if(s1[i-1] == s2[j-1] && m < a[j-1][i-1]+1) { m = a[j-1][i-1]+1; }
      a[j][i] = m;
    }
  }
//for(int j = 0; j <= l2; ++j) {
//  for(int i = 0; i <= l1; ++i) { printf("%3d", a[j][i]); }
//  printf("\n");
//}
  return a[l2][l1];
}
int main(int argc, char *argv[]) {
  char *s1 = argv[1], *s2 = argv[2];
  printf("lcs(%s,%s) = %d\n", s1, s2, lcs(s1, s2));
  return 0;
}
```

2 次元配列の表示はコメントアウトしてありますが、必要ならはずして見てください。実行例を示します。

```
% gcc8 lcs.c
% ./a.out isasaka sassa
lcs(isasaka,sassa) = 4
% ./a.out sassa wakasa
lcs(sassa,wakasa) = 3
% ./a.out isasaka wakasa
lcs(isasaka,wakasa) = 3
```

なぜこれで LCS が計算できるかを考えるには、LCS を次のように再帰関数として定義してみるの

が分かりやすいかも知れません。

$$lcs(a_1 \cdots a_m,\ b_1 \cdots b_n) = \begin{cases} 0 & (m = 0 \ \ or \ \ n = 0) \\ lcs(a_1 \cdots a_{m-1}, b_1 \cdots b_{n-1}) + 1 & (a_m = b_n) \\ max(lcs(a_1 \cdots a_m, b_1 \cdots b_{n-1}), \\ \quad lcs(a_1 \cdots a_{m-1}, b_1 \cdots b_n)) & (otherwise) \end{cases}$$

つまり、2 つの文字列の最後の文字が等しければ、その両方を削除した文字列どうしの LCS より 1 多い値が全体の LCS であり、等しくなければ、a から最後の 1 文字を除いた場合の LCS と b から最後の 1 文字を除いた場合の LCS のうち大きい方が全体の LCS ということになります。

演習 5 上の LCS のコードでは具体的な最長部分文字列を求めていない。トレースバックをおこない、具体的な最長部分文字列も求めるようにしなさい。

演習 6 LCS の問題の類似品として**編集距離** (edit distance) というものがある。これは、文字列 s_1 に対して何回 (1)1 文字挿入、(2)1 文字削除、(3) 隣接する 2 文字の交換を行ったら文字列 s_2 に変更できるかの回数の最小値である。LCS のプログラムを参考に編集距離のプログラムを作れ。

本日の課題 14A

「演習 1」〜「演習 6」で動かしたプログラム 1 つを含むレポートを本日中 (授業日の 23:59 まで) に提出してください。

1. sol または CED 環境で「`/home3/staff/ka002689/prog19upload 14a` ファイル名」で以下の内容を提出。
2. 学籍番号、氏名、ペアの学籍番号 (または「個人作業」)、提出日時。名前の行は先頭に「@@@」を付けることを勧める。
3. プログラムどれか 1 つのソースと「簡単な」説明。
4. レビュー課題。提出プログラムに対する他人 (ペア以外) からの簡単な (ただしプログラムの内容に関する) コメント。
5. 以下のアンケートの回答。

   Q1. この授業開始時の自分と現在の自分を比べてどのような変化があったと思いますか。

   Q2. 「プログラミングができる」ようになるためにはどのように学ぶ (教わる) のがよいと考えますか。

   Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

索 引

プログラミング通論 **2020**