プログラミング通論'19 ＃4 – スタック・キューと探索

久野 靖 (電気通信大学)

2019.4.20

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。

1　キューとその実装

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 1)。

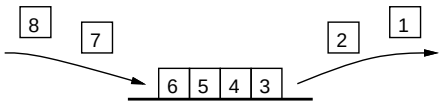

図 1: キューの概念

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 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。

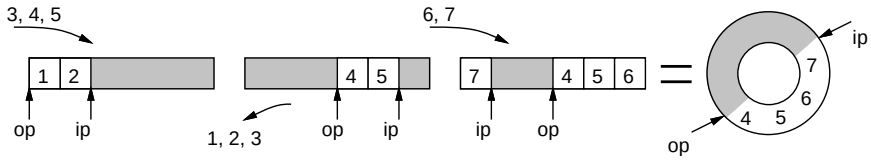

図 2: 配列を用いたキューの実装

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。

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 3)。

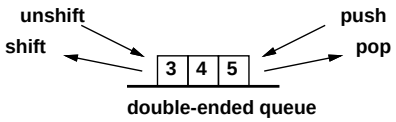

図 3: DEQ の概念

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。

2 スタック・キューと探索

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 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; };
```

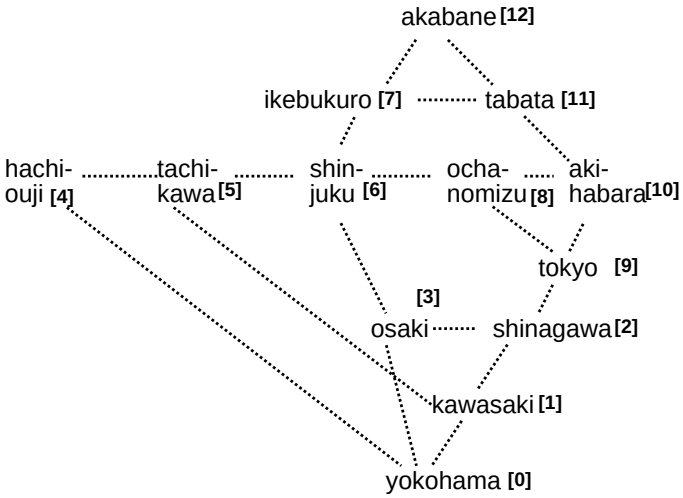

図 4: とある鉄道路線図

```
struct node map[13] = {
  { "yokohama", 3, { 1, 3, 4 }, -1 },     // 0
  { "kawasaki", 3, { 0, 2, 5 }, -1 },     // 1
  { "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。

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。

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 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。

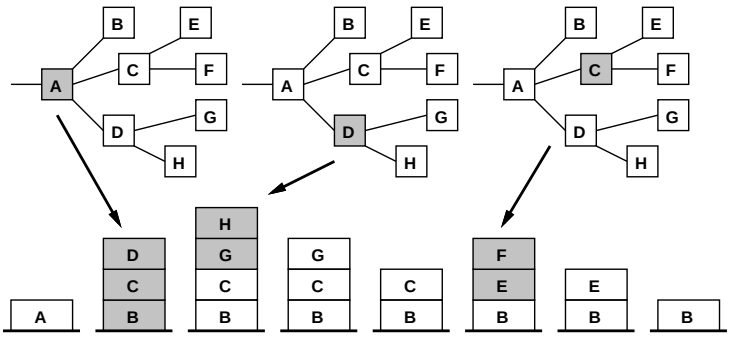

図 5: スタックと深さ優先のたどり

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 6)。これを**幅優先** (breadeth first) のたどり、と呼びます。

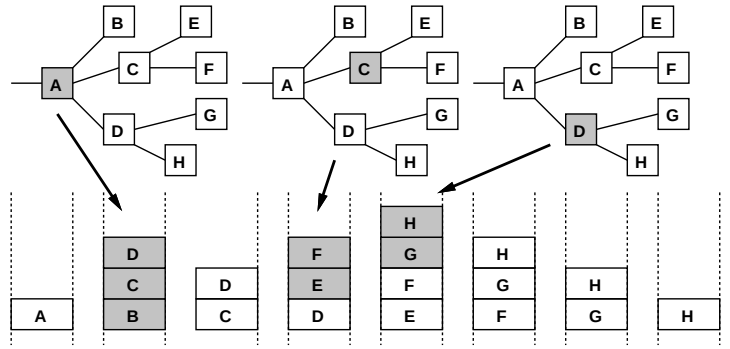

図 6: キューと幅優先のたどり

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

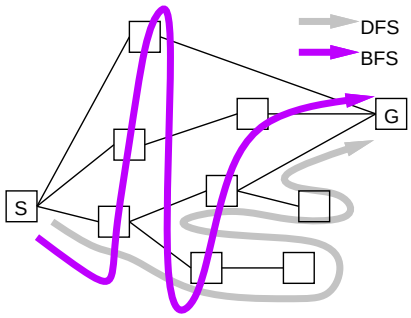

図 7: 深さ優先と幅優先の対比

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 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)
#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
```

[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。

```
0: yokohama, 1
2: shinagawa, 1
5: tachikawa, 1
3: osaki, 2
4: hachiouji, 2
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. sol または CED 環境で「`/home3/staff/ka002689/prog19upload 4a 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. 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. sol または CED 環境で「`/home3/staff/ka002689/prog19upload 4b` 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. 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。