List 4-4 IntQueue.h

```
// int型キューIntQueue（ヘッダ部）

#ifndef ___IntQueue
#define ___IntQueue

/*--- キューを実現する構造体 ---*/
typedef struct {
    int max;    // キューの容量
    int num;    // 現在のデータ数
    int front;  // 先頭要素カーソル
    int rear;   // 末尾要素カーソル
    int *que;   // キュー本体（の先頭要素へのポインタ）
} IntQueue;

/*--- キューの初期化 ---*/
int Initialize(IntQueue *q, int max);

/*--- キューにデータをエンキュー ---*/
int Enque(IntQueue *q, int x);

/*--- キューからデータをデキュー ---*/
int Deque(IntQueue *q, int *x);

/*--- キューからデータをピーク ---*/
int Peek(const IntQueue *q, int *x);

/*--- 全データを削除 ---*/
void Clear(IntQueue *q);

/*--- キューの容量 ---*/
int Capacity(const IntQueue *q);

/*--- キュー上のデータ数 ---*/
int Size(const IntQueue *q);

/*--- キューは空か ---*/
int IsEmpty(const IntQueue *q);

/*--- キューは満杯か ---*/
int IsFull(const IntQueue *q);

/*--- キューからの探索 ---*/
int Search(const IntQueue *q, int x);

/*--- 全データの表示 ---*/
void Print(const IntQueue *q);

/*--- キューの後始末 ---*/
void Terminate(IntQueue *q);

#endif
```

### キュー構造体：IntQueue

キューを管理するための構造体です。5個のメンバで構成されます。

#### ▪ キュー本体用の配列：que

押し込まれたデータを格納するキュー本体用の配列（の先頭要素へのポインタ）です。

#### ▪ キューの容量：max

キューの容量（キューに押し込める最大のデータ数）を表す **int** 型のメンバです。

配列 ***que*** の要素数と一致します。

#### ▪ 先頭／末尾カーソル：front／rear

キューに押し込まれているデータのうち、最初に押し込まれた先頭要素の添字を表すメンバが ***front*** です。

また、キューに押し込まれているデータのうち、最後に押し込まれた末尾要素の一つ後ろの添字（次にエンキューが行われる際に、データが格納される要素の添字）を表すメンバが ***rear*** です。

#### ▪ データ数：num

キューに蓄えられているデータ数を表すメンバです。キューが空のときに **0** となって、満杯のときに ***max*** と同じ値となります。

この変数が必要なのは、変数 ***front*** と ***rear*** の値が等しい場合、キューが空なのか満杯なのかが区別できなくなるからです（**Fig.4-10**：右ページ）。

▶ 図**a**が空の状態です。***front*** と ***rear*** は同じ値です。図**b**は満杯の状態です。この図でも、***front*** と ***rear*** は同じ値です（***que*[2]** が先頭要素で、***que*[1]** が末尾要素です）。図には示していませんが、両方とも **0** 以外の値であって、キューが空である、ということもありえます。

List 4-5 [A] chap04/IntQueue.c

```
// int型キューIntQueue（ソース部）

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

/*--- キューの初期化 ---*/
int Initialize(IntQueue *q, int max)
{
    q->num = q->front = q->rear = 0;
    if ((q->que = calloc(max, sizeof(int))) == NULL) {
        q->max = 0;                                    // 配列の生成に失敗
        return -1;
    }
    q->max = max;
    return 0;
}
```

## 初期化：Initialize

関数 *Initialize* は、キュー本体用配列の生成などの準備処理を行う関数です。

▶ 第1引数 q は、処理の対象となる**キュー構造体オブジェクトへのポインタ**です（これ以降のほとんどの関数も同様です）。

生成時のキューは空（データが 1 個もない状態）ですから、*num*、*front*、*rear* の値すべてを 0 にします。さらに、仮引数 *max* に受け取った《キューの容量》をメンバ *max* にコピーするとともに、要素数 *max* の配列 *que* の本体を確保します（図 a の状態となります）。

▶ 配列確保の失敗時にメンバ *max* に 0 を代入する理由は、スタックの場合と同じです。

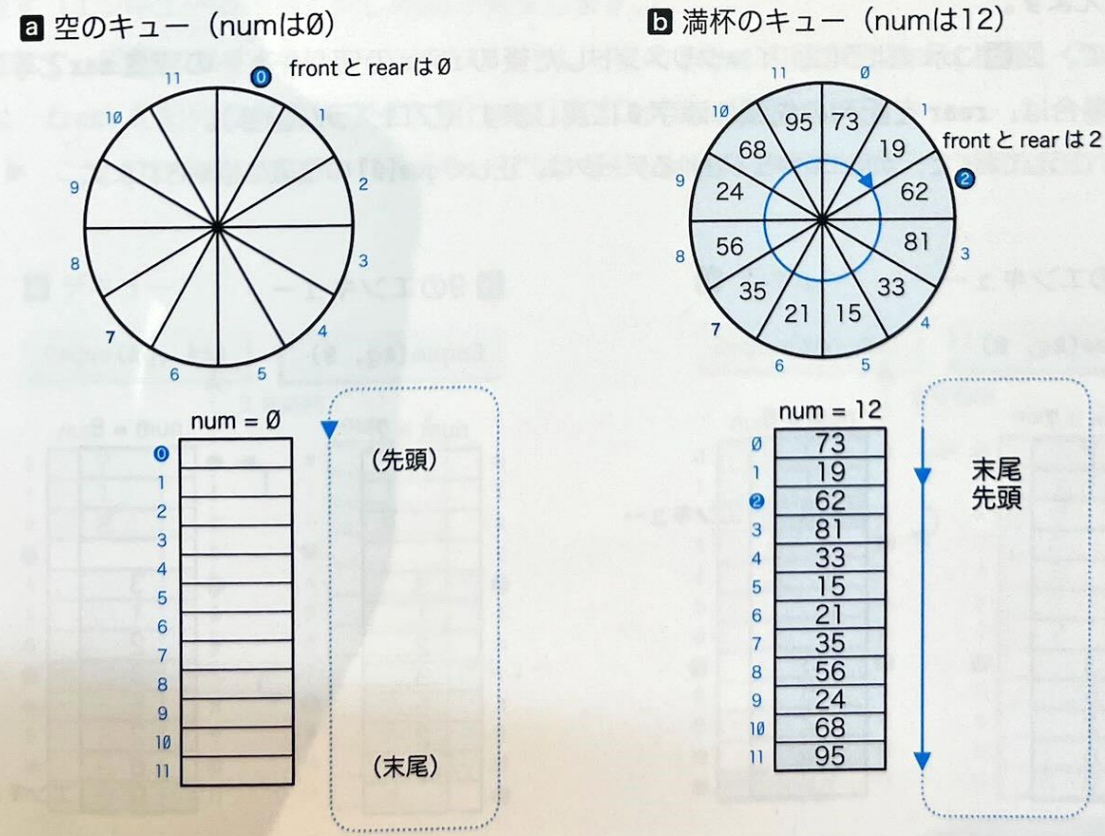


Fig.4-10 空のキューと満杯のキュー

## エンキュー：Enque

関数 Enque は、キューにデータをエンキューする関数です。

▶ エンキューに成功すると 0 を返却し、キューが満杯でエンキューできなければ -1 を返却します。

**List 4-5 [B]** chap04/IntQueue.c

```
/*--- キューにデータをエンキュー ---*/
int Enque(IntQueue *q, int x)
{
    if (q->num >= q->max)
        return -1;                          // キューは満杯
    else {
        q->num++;
        q->que[q->rear++] = x;              ← 1
        if (q->rear == q->max)              ← 2
            q->rear = 0;
        return 0;
    }
}
```

エンキューを行う Fig.4-11 を見ながら理解しましょう。

図 a は、{3, 5, 2, 6, 9, 7, 1} が押し込まれているキューに 8 をエンキューする様子です。que[rear] すなわち que[2] にエンキューするデータを格納して、rear と num の値をインクリメントする（プログラム 1 部）と、エンキューは完了します。

＊

さて、エンキュー前の rear が配列の物理的な末尾（本図の例では 11）であるときに rear をインクリメントすると、その値が max（本図の例では 12）と等しくなって、**配列の添字の上限を超えます。**

そこで、図 b に示すように、インクリメントした後の rear の値がキューの容量 max と等しくなった場合は、rear を配列の先頭の添字 0 に戻します（プログラム 2 部）。

▶ こうしておくと、次にエンキューされるデータは、正しく que[0] の位置に格納されます。

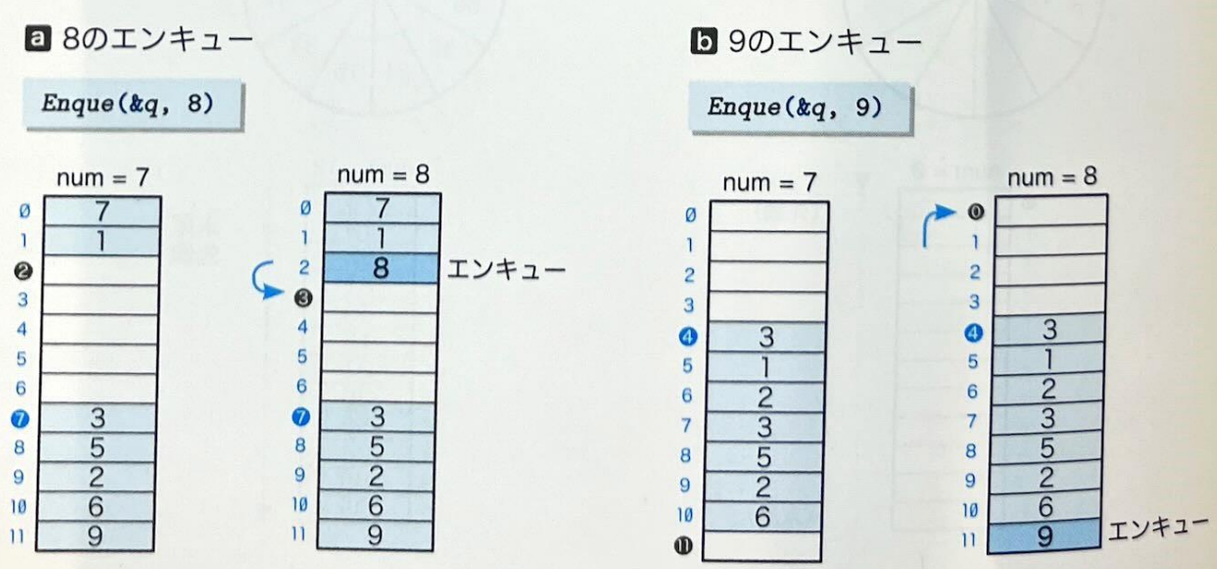


**Fig.4-11　キューへのエンキュー**

## デキュー：Deque

関数 *Deque* は、キューからデータをデキューする関数です。

▶ デキューに成功すると 0 を返却し、キューが空でデキューできなければ -1 を返却します。

**List 4-5 [C]** chap04/IntQueue.c

```
/*--- キューからデータをデキュー ---*/
int Deque(IntQueue *q, int *x)
{
    if (q->num <= 0)                          // キューは空
        return -1;
    else {
        q->num--;                                          1
        *x = q->que[q->front++];
        if (q->front == q->max)                            2
            q->front = 0;
        return 0;
    }
}
```

デキューを行う **Fig.4-12** を見ながら理解しましょう。

図**a**は、{3, 5, 2, 6, 9, 7, 1, 8} が押し込まれているキューからデキューする様子です。*que[front]* すなわち *que[7]* に格納されている値 3 を取り出して、*front* の値をインクリメントして *num* の値をデクリメントする（プログラム**1**部）と、デキューは完了します。

＊

さて、デキュー前の *front* が配列の物理的な末尾（本図の例では 11）であるときに *front* をインクリメントすると、その値が *max*（本図の例では 12）となって、配列の**添字の上限を超えます**（エンキューの場合と同じ問題が発生します。）

そこで、図**b**に示すように、インクリメントした後の *front* の値が容量 *max* と等しくなった場合は、*front* を配列の先頭添字 0 に戻します（プログラム**2**部）。

▶ こうしておくと、次に行われるデキューでは、正しく *que[0]* の位置からデータが取り出されます。

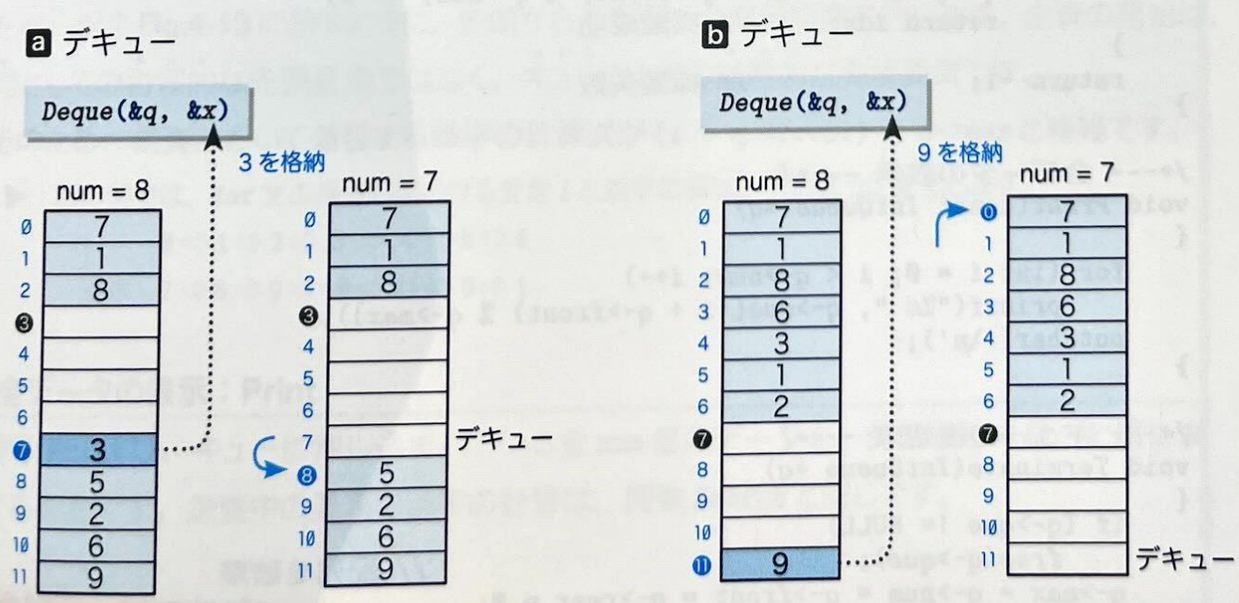


**Fig.4-12　キューからのデキュー**

**List 4-5 [D]** chap04/IntQueue.c

```
/*--- キューからデータをピーク ---*/
int Peek(IntQueue *q, int *x)
{
    if (q->num <= 0)
        return -1;                          // キューは空
    *x = q->que[q->front];
    return 0;
}

/*--- 全データを削除 ---*/
void Clear(IntQueue *q)
{
    q->num = q->front = q->rear = 0;
}

/*--- キューの容量 ---*/
int Capacity(const IntQueue *q)
{
    return q->max;
}

/*--- キューに蓄えられているデータ数 ---*/
int Size(const IntQueue *q)
{
    return q->num;
}

/*--- キューは空か ---*/
int IsEmpty(const IntQueue *q)
{
    return q->num <= 0;
}

/*--- キューは満杯か ---*/
int IsFull(const IntQueue *q)
{
    return q->num >= q->max;
}

/*--- キューからの探索 ---*/
int Search(const IntQueue *q, int x)
{
    for (int i = 0; i < q->num; i++) {
        int idx;
        if (q->que[idx = (i + q->front) % q->max] == x)
            return idx;                     // 探索成功
    }
    return -1;                              // 探索失敗
}

/*--- 全データの表示 ---*/
void Print(const IntQueue *q)
{
    for (int i = 0; i < q->num; i++)
        printf("%d ", q->que[(i + q->front) % q->max]);
    putchar('\n');
}

/*--- キューの後始末 ---*/
void Terminate(IntQueue *q)
{
    if (q->que != NULL)
        free(q->que);                       // 配列を破棄
    q->max = q->num = q->front = q->rear = 0;
}
```

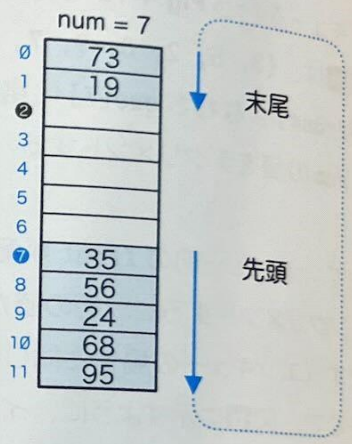


**Fig.4-13　キュー内の線形探索**

### ピーク：Peek

関数 *Peek* は、先頭のデータ、すなわち、次のデキューで取り出されるデータを“覗き見”する関数です。*que[front]* の値を調べるだけであって、データの取出しは行いませんので、*front* や *rear* や *num* の値の更新は行いません。

▶ ピークに成功すると 0 を返却し、キューが空でピークできなければ -1 を返却します。

### 全データの削除：Clear

関数 *Clear* は、現在キューに押し込まれている全データを削除する関数です。

▶ エンキューやデキューは *num*、*front*、*rear* の値に基づいて行われますので、それらの値を 0 にするだけです（キュー本体用の配列要素の値を変更する必要はありません）。

### 容量／データ数を調べる：Capacity／Size

関数 *Capacity* は、キューの容量、すなわちキューに押し込める最大のデータ数を返す関数です。メンバ *max* の値をそのまま返します。

関数 *Size* は、現在キューに押し込まれているデータ数を返す関数です。メンバ *num* の値をそのまま返します。

### 空であるか／満杯であるかを判定する：IsEmpty／IsFull

関数 *IsEmpty* は、キューが空（データが 1 個も押し込まれていない状態）であるかどうかを判定する関数です。空であれば 1 を、そうでなければ 0 を返します。

関数 *IsFull* は、キューが満杯（これ以上データが押し込めない状態）であるかどうかを判定する関数です。満杯であれば 1 を、そうでなければ 0 を返します。

### 探索：Search

関数 *Search* は、キューの配列内の、*x* と等しいデータが含まれている位置を調べる関数です。探索成功時は見つけた要素の添字を返し、失敗時は -1 を返します。

左ページの **Fig.4-13** に示すように、先頭から末尾側へと線形探索を行います。走査の開始は、配列としての物理的な先頭要素ではなく、キューとしての論理的な先頭要素です。

そのため、走査において着目する添字の計算式が *(i + q->front) % q->max* と複雑です。

▶ 図の例では、*for* 文の繰返しにおける変数 *i* と添字の値は、次のように変化します。

*i*　0 ⇨ 1 ⇨ 2 ⇨ 3 ⇨ 4 ⇨ 5 ⇨ 6

添字　7 ⇨ 8 ⇨ 9 ⇨ 10 ⇨ 11 ⇨ 0 ⇨ 1

### 全データの表示：Print

関数 *Print* は、キューに押し込まれている全 *num* 個のデータを、先頭から末尾へと順に表示する関数です。走査中の要素の添字の計算は、関数 *Search* と同じです。

### 後始末：Terminate

関数 *Terminate* は、キュー本体用の配列を破棄する後始末用の関数です。

### 利用例

キューを利用するプログラムを作りましょう。**List 4-6** にプログラム例を示します。

▶ 本プログラムのコンパイルには、"IntQueue.h" と "IntQueue.c" が必要です。

**List 4-6** chap04/IntQueueTest.c

```
// int型キューIntQueueの利用例

#include <stdio.h>
#include "IntQueue.h"

int main(void)
{
    IntQueue que;

    if (Initialize(&que, 64) == -1) {
        puts("キューの生成に失敗しました。");
        return 1;
    }

    while (1) {
        int m, x;

        printf("現在のデータ数：%d / %d\n", Size(&que), Capacity(&que));
        printf("(1)エンキュー　(2)デキュー　(3)ピーク　(4)表示　(0)終了：");
        scanf("%d", &m);

        if (m == 0) break;

        switch (m) {
         case 1: /*--- エンキュー ---*/
                 printf("データ：");   scanf("%d", &x);
                 if (Enque(&que, x) == -1)
                    puts("\aエラー：エンキューに失敗しました。");
                 break;

         case 2: /*--- デキュー ---*/
                 if (Deque(&que, &x) == -1)
                    puts("\aエラー：デキューに失敗しました。");
                 else
                    printf("デキューしたデータは%dです。\n", x);
                 break;

         case 3: /*--- ピーク ---*/
                 if (Peek(&que, &x) == -1)
                    puts("\aエラー：ピークに失敗しました。");
                 else
                    printf("ピークしたデータは%dです。\n", x);
                 break;

         case 4: /*--- 表示 ---*/
                 Print(&que);
                 break;
        }
    }

    Terminate(&que);

    return 0;
}
```

容量 64 のキューを生成して、エンキュー、デキュー、ピーク、キュー内データの表示を対話的に行います。

**実行例**

```
現在のデータ数：0 / 64
(1)エンキュー　(2)デキュー　(3)ピーク　(4)表示　(0)終了：1↵
データ：1↵                                                    ← 1をエンキュー
現在のデータ数：1 / 64
(1)エンキュー　(2)デキュー　(3)ピーク　(4)表示　(0)終了：1↵
データ：2↵                                                    ← 2をエンキュー
現在のデータ数：2 / 64
(1)エンキュー　(2)デキュー　(3)ピーク　(4)表示　(0)終了：4↵
1 2                                                           ← キューの中身を表示
現在のデータ数：2 / 64
(1)エンキュー　(2)デキュー　(3)ピーク　(4)表示　(0)終了：2↵
デキューしたデータは1です。                                    ← 1をデキュー
現在のデータ数：1 / 64
(1)エンキュー　(2)デキュー　(3)ピーク　(4)表示　(0)終了：4↵
2                                                             ← キューの中身を表示
現在のデータ数：1 / 64
(1)エンキュー　(2)デキュー　(3)ピーク　(4)表示　(0)終了：3↵
ピークしたデータは2です。                                      ← 2をピーク
現在のデータ数：1 / 64
(1)エンキュー　(2)デキュー　(3)ピーク　(4)表示　(0)終了：4↵
2                                                             ← キューの中身を表示
現在のデータ数：1 / 64
(1)エンキュー　(2)デキュー　(3)ピーク　(4)表示　(0)終了：0↵
```

### 演習 4-4

**int** 型キューのプログラムに、任意のデータを探索する関数を追加せよ。

```
int Search2(const IntQueue& q, int x);
```

関数 ***Search*** のように見つけた位置の配列の添字を返すのではなく、キュー内での論理的なデータの並びにおいて、先頭の何個後ろに存在するのかを返すこと。なお、探索に失敗した場合に返す値は **-1** とする。

たとえば、**Fig.4-13**（p.164）の例であれば、**35** を探索すると **0** を、**56** を探索すると **1** を、**73** を探索すると **5** を返す。また、キューに存在しない **99** を探索すると **-1** を返す。

### 演習 4-5

**List 4-6** で利用しているのは、**"IntQueue.c"** で提供される関数のうちの一部である。前問で作成した関数 ***Search2*** を含め、すべての関数を利用するプログラムを作成せよ。

### 演習 4-6

一般に**デック**と呼ばれる**両方向待ち行列**（deque ／ double ended queue）は、下図に示すように、先頭と末尾の両方に対して、データの押込み・取出しが行えるデータ構造である。両方向待ち行列を実現するプログラムを作成せよ。なお、デックに格納するデータは **int** 型の値とする。