キューと幅優先探索

キュー(Queue)

•キューは次の条件を満たすデータ構造

1. 新しい情報を追加できる
2. 古い情報の一つを取り出す。そのときには、最も古いものから順に取り出す

追加と取り出し方法が制限された「配列」とみることもできる

「先に入った情報から先に出て行く」という性質から、FIFO(First In First Out)とも呼ばれている

例：宝くじ売り場に並んでいる人の列

キューのデータ構造と固有な操作

- キューの実現方法 ：**配列と変数**を用いる

  配列： データの記憶用

  変数head, tail： 配列の先頭の位置と末尾の位置を記憶。

  変数max, 変数number: キューで記憶できるデータ数（配列の大きさ）、現在記憶しているデータ数（初期値は0）

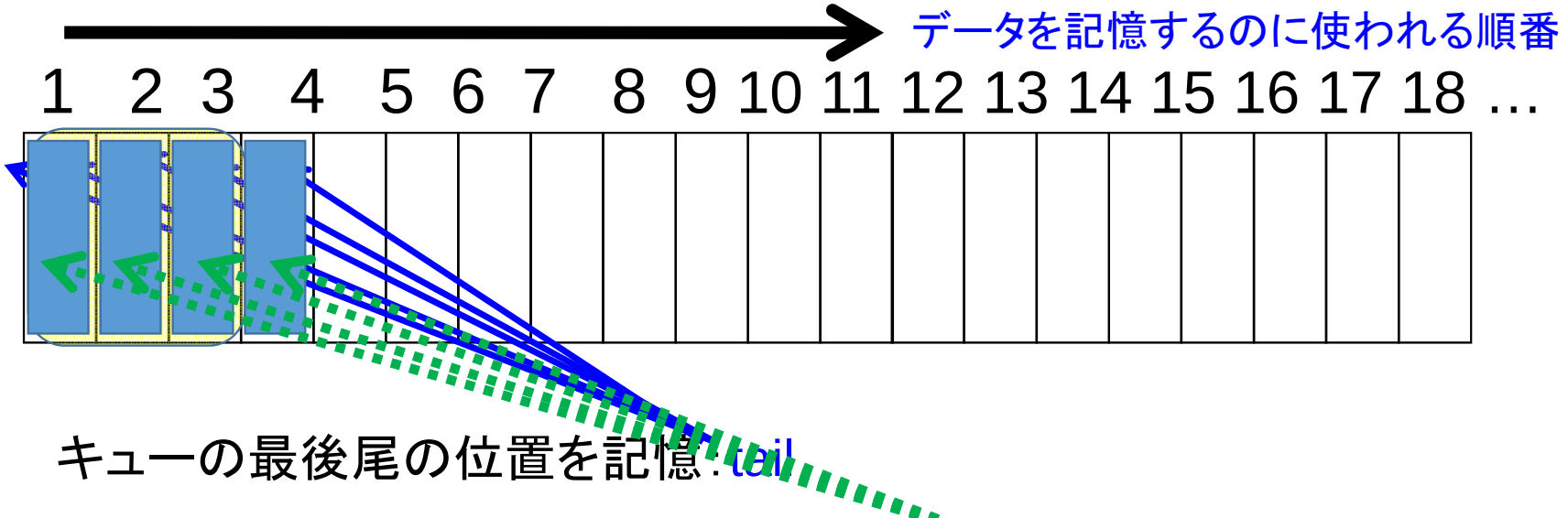
  配列のheadからtailまでが、キューに記憶されているデータ(空のときを除く)

- 固有な操作

  エンキュー(enqueue)：キューにおいてデータを記憶

  デキュー (dequeue)：キューにおいてデータを取出す

配列を用いた「単純な」キュー



キューの最後尾の位置を記憶：tail

キューの先頭の位置を記憶：head

エンキュー（キューに記憶）により、tailの値が増える

デキュー(キューから取出し）により、headの値が増える

の部分は永久に使えなくなる！

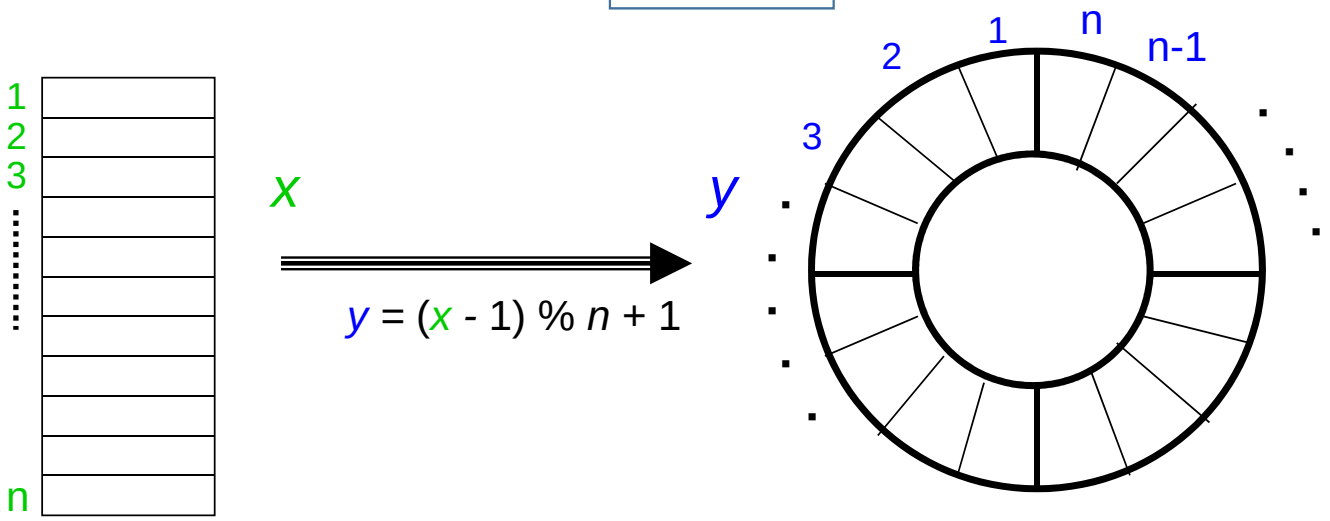
直線を環状とみなす

データを記憶するための「場所」

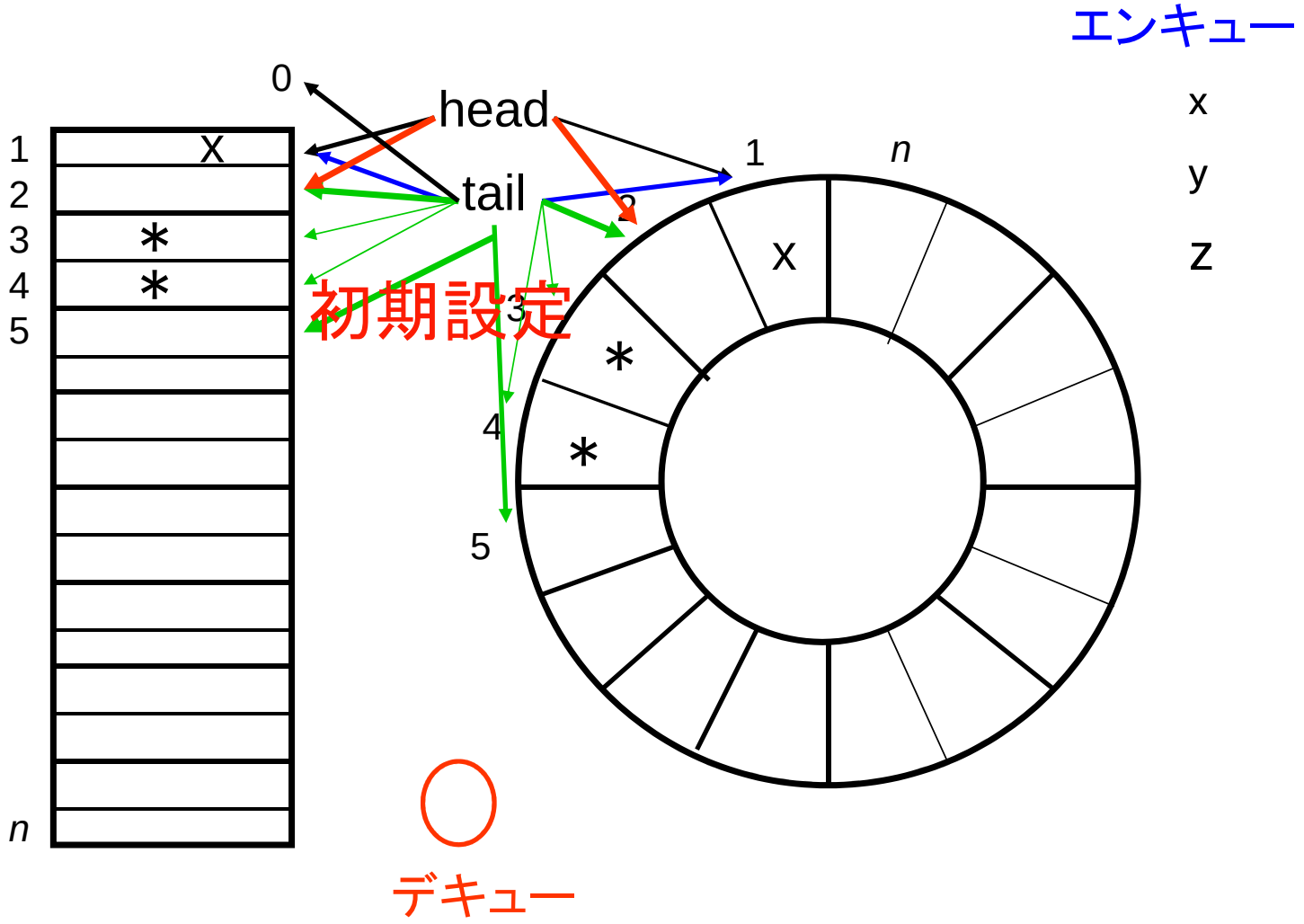
- 配列は、一直線上に要素を並べたもの
- キューでは、先頭も末尾も動く

  ⇒　一直線よりも環状に要素を並べた方が記憶場所が有効に使える

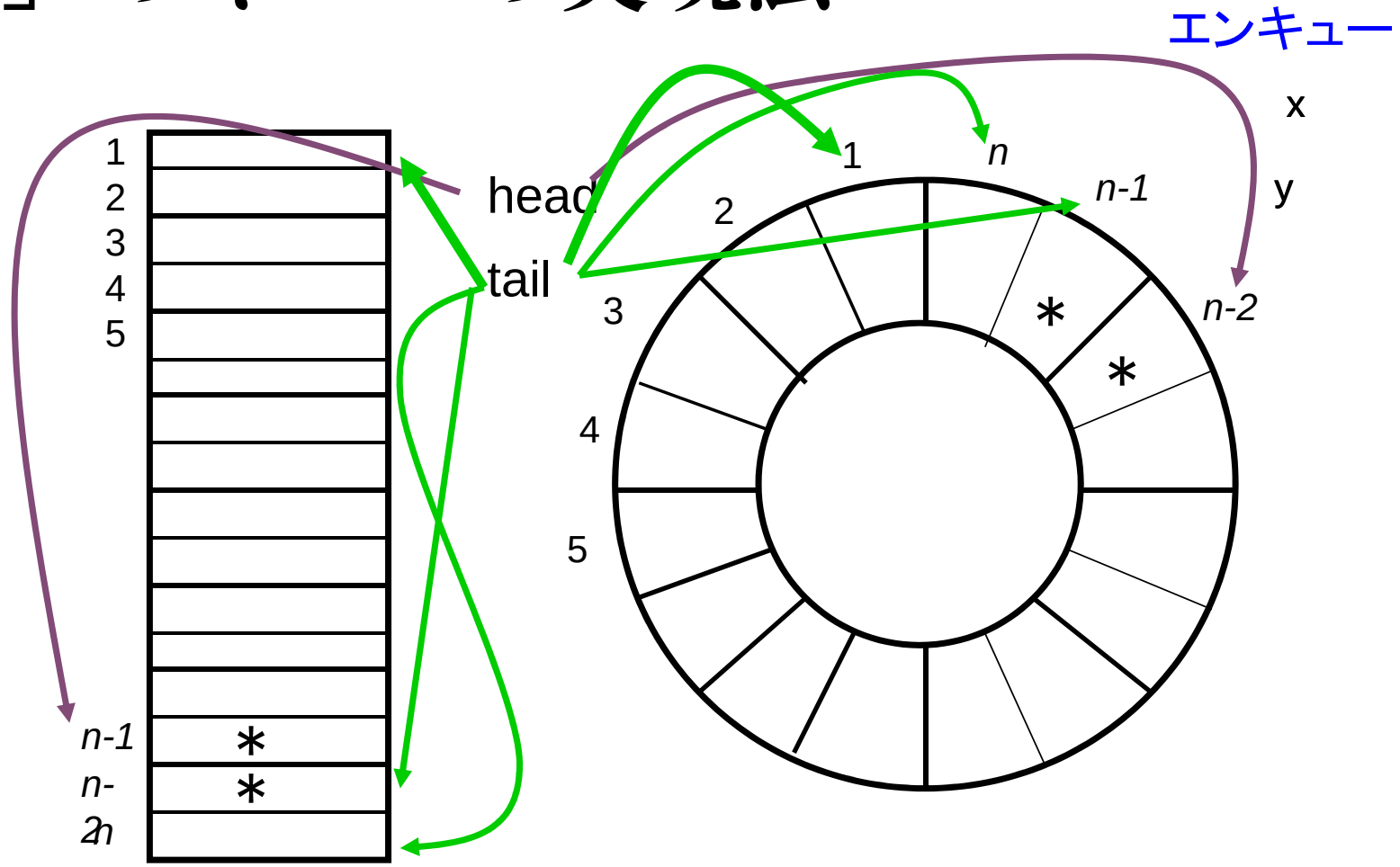
見かけ上



「環状」のキューの実現法



「環状」のキューの実現法



Cによる実現

```
// キューに記憶する要素の最大数
#define NUM 100
// キューに記憶する要素の型---ここでは整数と仮定
#define TYPE  int
typedef  struct  queue {
    TYPE  *data[NUM];     // データへのポインタを記憶
    int head, tail, max, number;
} QUEUE;
void enqueue(QUEUE *q, TYPE *item) ;   // エンキュー
TYPE *dequeue(QUEUE *q) ;   // デキュー
```

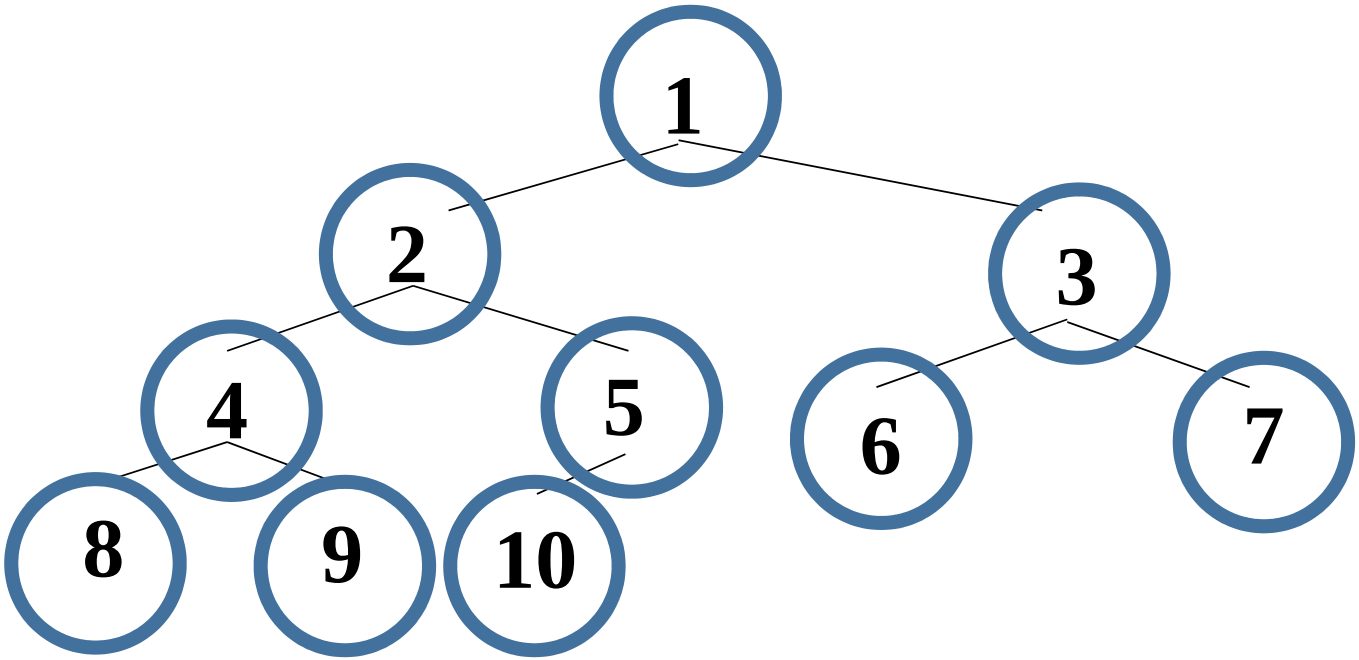
エンキュー

```
void enqueue(QUEUE *q, TYPE  *item) {
    if (q->number < NUM) {       // 記憶容量を超えなければ記憶する
        // 要素は tail に入れる--- tailの値を+1しておく
        q->tail = (q->tail + 1) % NUM;  // tailの値の更新 ： 環状記憶
        q->number++;                    // 記憶している要素数が1増える
        q->data[q->tail] = item;    // tailの場所にitem(ポインタ)をいれる
     } else {
        fprintf(stderr, “Queue overflows.¥n”);  // エラーメッセージ表示
     }
}
```

デキュー

```
TYPE  *dequeue(QUEUE *q) {   // データの取り出し：headにあるものを取り出す
      TYPE  *save;
     if (q->number > 0) {   // 記憶しているものがあれば
          q->number--;          // 記憶している要素数を－1する
          save = q->data[q->head];   // headにあるデータを取り出す
          q->head = (q->head + 1) % NUM;      // headの値の更新 ： 環状記憶
          return save;          // データを返す
       } else {       // エラーの場合
          fprintf(stderr, "Queue is empty¥n");
          return NULL;
       }
}
```

幅優先による木のなぞり



取り出されたノード：

1　2　3　4　5　6　7　8　9　10

二進木のCによる実現

構造体NODEの定義

```
typedef struct node {
    char label[LEN];
    struct node *left;
    struct node *right;
} NODE;
```

NODE

ラベル(文字列として定義)

左(left)の子(NODE型オブジェクトへのポインタ)

右(right)の子(NODE型オブジェクトへのポインタ)

幅優先の実現方法

キューの表現:

```
typedef struct queue {
    NODE *data[NUM];
    int head, tail, number;
} QUEUE;
```

TYPEとしていたところを
NODEとすればよい

エンキュー:

```
void enqueue(QUEUE *q, NODE *item) ;
```

デキュー:

```
NODE *dequeue(QUEUE *q) ;
```

幅優先の実現方法（続)

```
void breadthFirst(NODE *n) {      //根のノードから始める
    NODE *x;
    QUEUE aq;
    aq.head = 1; aq.tail = 0; aq.number = 0;     // キューの初期化
    enqueue(&aq, n); // まずキューに根ノードをいれる
    while (aq.number > 0) {   // キューにデータがある限り繰り返す
      x = dequeue(&aq);                // キューの先頭からデータを取り出す
      printf("%6s", x->label);    // ラベルを表示する
      if (x->left != NULL) enqueue(&aq, x->left);     // 左の子をキューに入れる
      if (x->right != NULL) enqueue(&aq, x->right);   // 右の子をキューに入れる
    }
    printf("¥n");
}
```