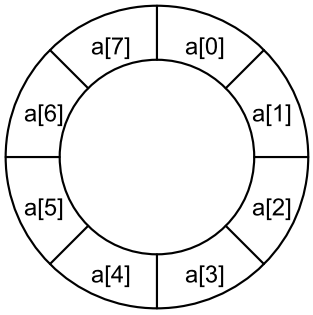
アルゴリズムとデータ構造　第９回演習　　年　　組　　番 氏名

1. **キュー（待ち行列）**とは，先入れ先出し方式（FIFO，先着順処理）のデータ構造である。これを配列を使って素朴な方法で実現すると，先頭のデータを取り出すアルゴリズムは以下のようになる。この方法の計算量を O 記法で表せ。取り出し処理を高速にするためにはどうすればいいか考えてみよ。

```
data = a[0];  n--;
for (i = 0; i < n; i++)
    a[i] = a[i + 1];
```

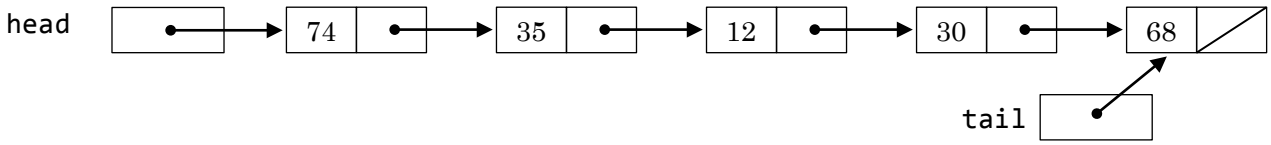
2. **リングバッファ**は右図のように配列の末尾と先頭がつながっているものとして扱うデータ構造である。下記はリングバッファによってキューを実現したプログラムである。キューの末尾にデータを追加する操作を**エンキュー**，キューの先頭からデータを取り出す操作を**デキュー**という。適切に空欄を埋めてプログラムを完成させ，動作を確認せよ。




```
#include <stdio.h>
#include <stdlib.h>

#define N 8    /* キューの最大長 */
int queue[N];  /* キュー */
int head = 0;  /* 先頭要素の位置 */
int tail = 0;  /* 末尾要素の次の位置 */

void enqueue(int data)
{
    /* 空きがあるかチェック */
    if ((tail + 1 == head) ||
        (tail + 1 == N && head == 0)) {
        printf("Queue is full.¥n");
        exit(1);
    }
    /* 末尾にデータを付け加える */
    queue[tail] = data;


    if (tail == N)

}

int dequeue(void)
{
    int data;

    /* データがあるかチェック */
    if (     ==     ) {

        printf("Queue is empty.¥n");
        exit(1);
    }
    /* 先頭からデータを取り出す */
    data = queue[head];


    if (head == N)

    return data;
}
```

```
void print_queue(void)
{
    int i;

    printf("[ ");
    i = head;
    while (i != tail) {
        printf("%d ", queue[i]);
        if (++i == N) i = 0;
    }
    printf("]¥n");
}

/* キューの動作確認プログラム */
int main(void)
{
    int data;
    int i;

    for (i = 0; i < 10; i++) {
        printf("Data %d: ", i);
        scanf("%d", &data);

        if (data > 0) {
            /* 入力が正数ならその数を格納 */
            enqueue(data);
            printf("enqueued: %d¥n", data);
        } else {
            /* 入力が負 or 0 なら取り出し */
            data = dequeue();
            printf("dequeued: %d¥n", data);
        }
        print_queue();
    }
    return 0;
}
```

3. 下記のプログラムは，2.と同様のキューを単方向リストに末尾を指すポインタ tail を加えて実現したものである。適切に空欄を埋めてプログラムを完成させ，動作を確認せよ。




```
#include <stdio.h>
#include <stdlib.h>

/* リスト要素（ノード）の構造体の定義 */
struct node {
    int data;           /* 格納データ */
    struct node *next; /* 次へのリンク */
};

/* 先頭と末尾のノードを指すポインタ */
struct node *head = NULL;
struct node *tail = NULL;

struct node *make_node(int data)
{
    /* 前回の make_node の内容をコピー */
}

/* リストの末尾にノードを加える */
void enqueue(int data)
{
    struct node *p = make_node(data);

    if (head == NULL) {
      head = tail = p;
    } else {
      /* tail の後ろに p をつなげる */



    }
}
```

```
/* リストの先頭からノードを取り出す */
int dequeue(void)
{
    int data;
    struct node* p = head;

    if (head == NULL) {
        printf("Queue is empty.¥n");
        exit(1);
    }
    /* 以下は stack の pop とほぼ同じ */
    data =

    if (head == tail) {
        head = tail = NULL;
    } else {


    }
    free(p);
    return data;
}

void print_queue(void)
{
    /* 前回の print_stack の内容をコピー */
}

int main(void)
{
    /* 2.の main と全く同じ内容 */
}
```

4. リストの途中で要素を挿入・削除することが多い場合は，逆方向にもリンクをたどれる**双方向リスト**が便利である。適切に空欄を埋めて双方向リストに関するノードの削除・挿入操作のための関数を完成させよ。

```
struct node {
    int data;
    struct node *prev, *next;  /* 前と次 */
};

void remove_node(struct node *p)
{
    if (p->prev != NULL)
        p->prev->next =

    if (p->next != NULL)
        p->next->prev =

    free(p);
}
```

```
/* 指定ノードの直後に新ノードを挿入
   ただし，問題の簡単のため，先頭または
   末尾への挿入は扱わないものとする */
void insert_node(
    struct node *prev, struct node *p)
{
    p->prev =

    p->next =

    p->prev->next =

    p->next->prev =

}
```