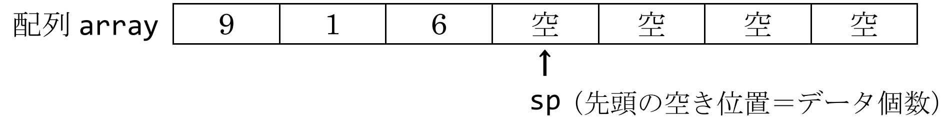
アルゴリズムとデータ構造 2023　第８回　演習課題　「スタックとキュー」

1. **スタック**とは後入れ先出し（LIFO，先入れ後出し FILO）のデータ構造であり，下記のプログラムはこれを配列で実現したものである。スタックにデータを格納する操作を**プッシュ**，スタックからデータを取り出す操作を**ポップ**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。




```
/* Stack.java */
public class Stack {

   private int[] array; // スタック本体
   private int sp;  // スタックポインタ

   public Stack(int size) {
      array = new int[size];
      sp = 0;
   }

   public void push(int data)
         throws Exception {
      // 空きがない場合をチェック
      if (                         ) {

         throw new Exception("full!");
      }
      // スタックにデータを「積む」

      array[sp] =


   }

   public int pop() throws Exception {
      // データがない場合をチェック
      if (          ) {

         throw new Exception("empty!");
      }
      // トップにあるデータを取り出す


                = array[sp];

      return data;
   }

   public void printAll() {
      System.out.print("[ ");
      for (int i = 0; i < sp; i++) {

      }
      System.out.println("]");
   }
}
```

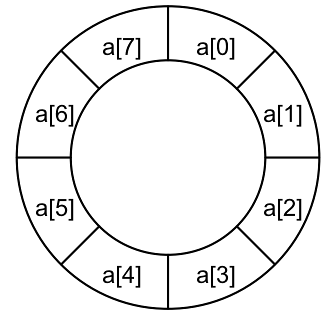
```
/* Main.java */
import java.util.Scanner;

public class Main {

   public static void main(String[] args)
         throws Exception {

      // 容量 5 個のスタックの新規生成
      Stack stack = new Stack(5);
      Scanner sc = new Scanner(System.in);

      for (int i = 0; i < 10; i++) {
         System.out.print("number? ");
         int data = sc.nextInt();

         if (data > 0) {
            // 入力が正数ならその数を格納
            stack.push(data);
         } else {
            // 入力が負数か 0 なら取り出し
            data = stack.pop();
            System.out.println(data);
         }
         stack.printAll();
      }
   }
}
```

2. 1.のプログラムのプッシュとポップの計算量は，スタック内のデータ数 n に対して$O(1)$（一定）である。スタックポインタ（sp）がない場合と比較して，それが高速化に果たしている役割を考察せよ。

3. **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。この方法では，スタックと比べて処理が遅くなってしまう原因を考察せよ。

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

4. **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。また，計算量の改善結果を考察せよ。




```
/* Queue.java */
public class Queue {

   private int[] array; // リングバッファ
   private int head; // キューの先頭（添字）
   private int n;  // 格納しているデータ数

   public Queue(int size) {
      array = new int[size];
      head = 0;
      n = 0;
   }

   public void enqueue(int data)
         throws Exception {
      // 空きがない場合をチェック
      if (n                         ) {

         throw new Exception("full!");
      }
      // キューの末尾の位置（添字）を求める
      int tail = (head + n) %

      array[tail] = data;

   }

   public int dequeue() throws Exception {
      // データがない場合をチェック
      if (n       ) {

         throw new Exception("empty!");
      }
      int data = array[head];

      // 先頭位置を 1 つ進める
      head = (head + 1) %


      return data;
   }

   public void printAll() {
      System.out.print("[ ");
      for (int i = 0; i < n; i++) {
         int k = (head + i) % array.length;
         System.out.print(array[k] + " ");
      }
      System.out.println("]");
   }
}
```

```
/* Main.java */
import java.util.Scanner;

public class Main {

   public static void main(String[] args)
         throws Exception {

      // 容量 8 個のキューの新規生成
      Queue queue = new Queue(8);
      Scanner sc = new Scanner(System.in);

      for (int i = 0; i < 10; i++) {
         System.out.print("number? ");
         int data = sc.nextInt();

         if (data > 0) {
            // 入力が正数ならその数を格納
            queue.enqueue(data);
         } else {
            // 入力が負 or 0 なら取り出し
            data = queue.dequeue();
            System.out.println(data);
         }
         queue.printAll();
      }
   }
}
```

5.〜6.　1.と 4.のプログラムをもとにして文字列を要素とするスタックとキューを作成し，動作を確認せよ。

7. 【発展】スタックとキューの機能をあわせ持ち，先頭からも末尾からもデータの出し入れができるデータ構造を**両端キュー**（double ended queue; deque; **デック**）という。以下は，4.のプログラムをもとにした両端キューの実装例である。適切に空欄を埋めてクラスを完成させ，main を補って動作を確認せよ。

```
/* Deque.java */
public class Deque {

   private int[] array; // リングバッファ
   private int head; // キューの先頭位置
   private int n;  // 格納しているデータ数

   public Deque(int size) {
      array = new int[size];
      head = 0;
      n = 0;
   }

   // 末尾への追加：enqueue と同じ
   public void pushBack(int data)
         throws Exception {
      // 空きがない場合をチェック
      if (n                        ) {

         throw new Exception("full!");
      }
      // キューの末尾の位置を求める
      int tail = (head + n) %

      array[tail] = data;

   }

   // 先頭からの取り出し：dequeue と同じ
   public int popFront() throws Exception {
      // データがない場合をチェック
      if (n       ) {

         throw new Exception("empty!");
      }
      int data = array[head];

      // 先頭位置を 1 つ進める
      head = (head + 1) %


      return data;
   }

   // 末尾からの取り出し：pop とほぼ同じ
   public int popBack() throws Exception {
      // データがない場合をチェック
      if (n       ) {

         throw new Exception("empty!");
      }
      // キューの末尾の新しい位置を求める

      int last = (head + n) %

      return array[last];
   }

   // 先頭への追加
   public void pushFront(int data)
         throws Exception {
      // 空きがない場合をチェック
      if (                        ) {

         throw new Exception("full!");
      }
      // 先頭位置を 1 つ戻す（広げる）
      if (--head < 0)
         head =

      array[head] = data;
      n++;
   }

   public void printAll() {
      System.out.print("[ ");
      for (int i = 0; i < n; i++) {
         int k = (head + i) % array.length;
         System.out.print(array[k] + " ");
      }
      System.out.println("]");
   }

}
```