Урок 3

Стек и очередь

Обзор структуры данных. Стек, очередь и приоритетная очередь.

Введение

В отличие от массивов, стеки и очереди являются абстрактными структурами данных. Они определяются своими интерфейсами, которые содержат определенные методы.

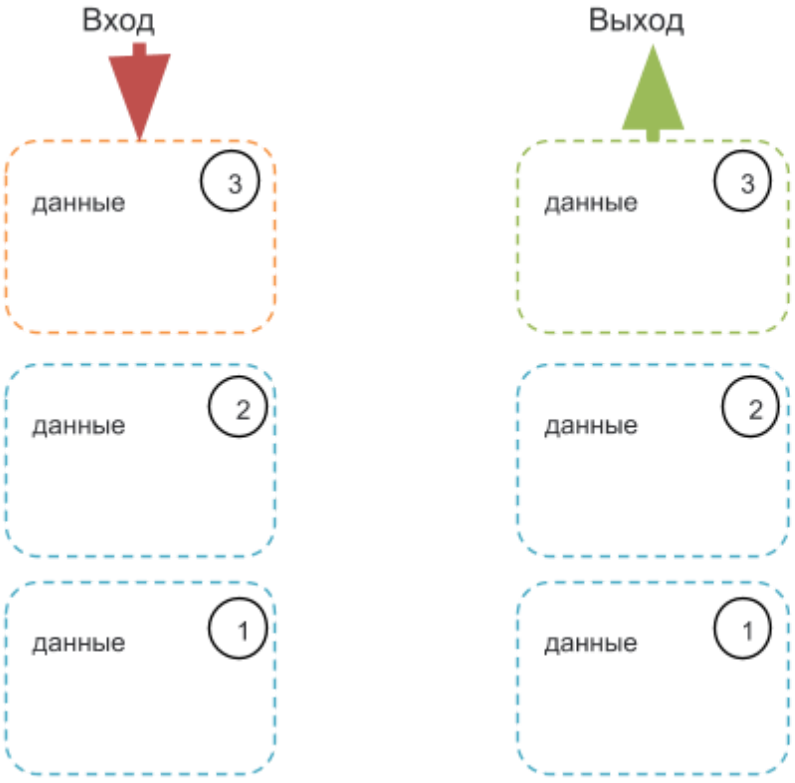
Разберемся, зачем нужны стеки, очереди и приоритетные очереди.

Стеки

Стек — это абстрактная структура данных, которая содержит список элементов. Но есть небольшое ограничение: стек работает по принципу «последний пришел — первый вышел» (LIFO, Last In — First Out).

Одно из объяснений работы стека, которое встречается во многих источниках — это стопка тарелок. Представьте, что мы кладем на стол красную тарелку, на нее синюю, а на синюю — зеленую. Чтобы достать красную тарелку, сначала из стопки вытаскиваем зеленую, потом синюю.

В этом уроке посмотрим, как можно использовать стек для проверки сбалансированности скобок в программном коде.



В отличие от массива, в стеке нельзя получить доступ к произвольному элементу — только к последнему.

Для реализации стека понадобится разработать методы для добавления, удаления, вывода в консоль элемента, проверки на пустоту и на переполненность. Добавлять и удалять элементы будем с левой стороны стека.

Необходимо заполнить следующие поля и создать конструктор, занимающийся этим:

```
private int maxSize;
private int[] stack;
private int top;

public Stack(int size){
      this.maxSize = size;
      this.stack = new int[this.maxSize];
      this.top = -1;
}
```

В поле **maxSize** хранится значение максимального размера стека. Поле **top** — это вершина стека. Это позиция элемента, который «последним вошел — первым вышел». Значение поля **top** «**-1**» говорит о том, что стек пустой. В качестве структуры данных, в которой будет храниться стек, используется массив. Реализуем метод проверки стека на пустоту и на переполненность:

```
public boolean isEmpty(){
      return (this.top == -1);
}

public boolean isFull(){
      return (this.top == this.maxSize-1);
}
```

Добавляем элемент

Каждый новый элемент добавляют в конец стека. Для этого напишем простой метод **push**:

```
public void push(int i){
        this.stack[++this.top] = i;

}
```

Префиксная форма инкремента сначала прибавляет к полю **top** единицу, а уже потом с новым индексом присваивает элементу стека значение. Метод **push** ничего не возвращает.

Удаляем элемент

Чтобы убрать элемент из стека, реализуем метод **pop**. Он удаляет элемент из стека, который находится в позиции **top**.

```
public int pop(){
      return this.stack[this.top--];

}
```

Получаем элемент

Для получения элемента стека, который находится в позиции **top**, реализуем метод **peek**.

```
public int peek(){
      return this.stack[this.top];

}
```

Полный листинг созданного класса **Stack**:

```
class Stack{
    private int maxSize;
    private int[] stack;
    private int top;

    public Stack(int size){
        this.maxSize = size;
        this.stack = new int[this.maxSize];
        this.top = -1;
    }

    public void push(int i){
        this.stack[++this.top] = i;
    }

    public int pop(){
        return this.stack[this.top--];
    }

    public int peek(){
        return this.stack[this.top];
    }

    public boolean isEmpty(){
        return (this.top == -1);
    }

    public boolean isFull(){
        return (this.top == this.maxSize-1);
    }

}
```

Стек не является сложной структурой данных, все его методы связаны только с одним элементом, который находится на его вершине (**top**).

В классе специально были созданы методы **isEmpty** и **isFull**, чтобы перенести ответственность за проверку стека на пользователя, а не вносить их в методы **push** и **pop**.

Эффективность стека

Методы стека работают всегда с одним последним элементом, поэтому эффективность операций вставки, удаления и просмотра элемента — **O(1)**.

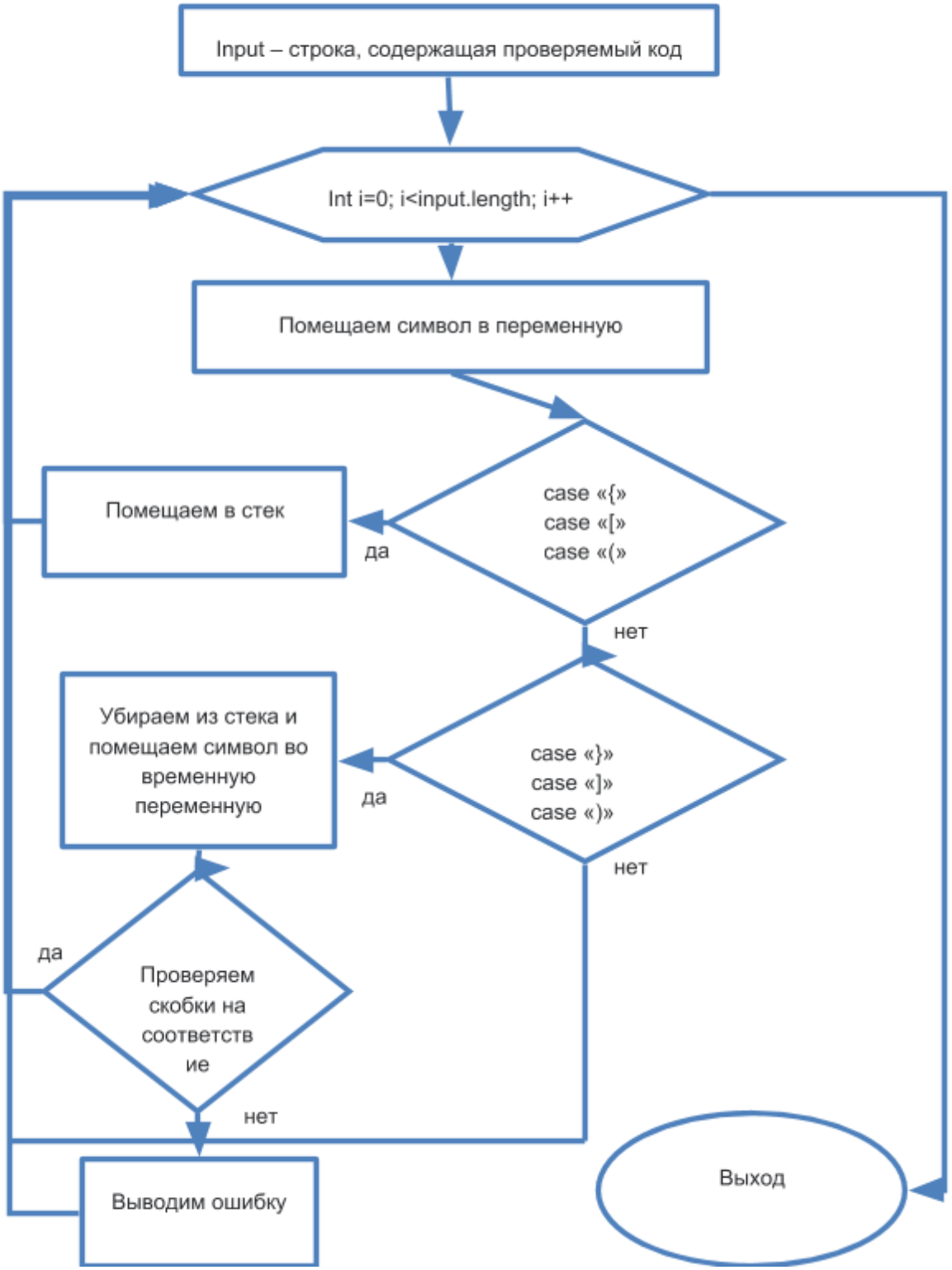
Пример использования

Представим, что мы пишем **IDE** для языка Java, и в этой среде разработки необходимо проверять код программы на правильность написания скобок. Например, есть метод, который возвращает строку:

```
public String getString(){
      return stringArr[0];

}
```

Задача нашей программы — проверить соответствие открывающих и закрывающих круглых, фигурных и квадратных скобок в представленном методе. Каждая открывающаяся скобка должна иметь пару. Если это соответствие нарушено, программа должна выводить на экран ошибку.

Алгоритм работы программы будет следующим:



Листинг программы, которая проверяет скобки на соответствие:

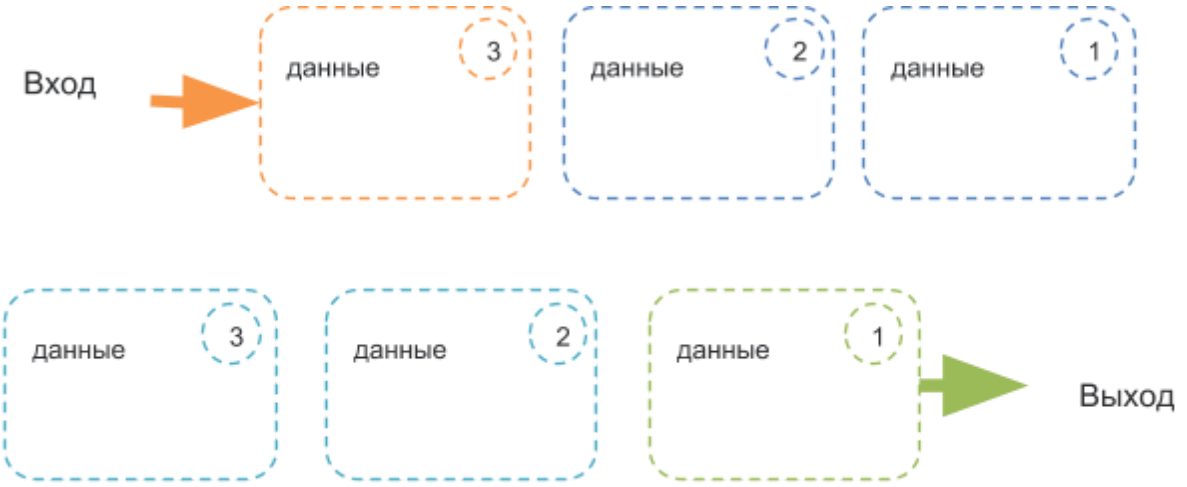
```
class Stack{
    private int maxSize;
    private char[] stackArr;
    private int top;

    public Stack(int size){
        this.maxSize = size;
        this.stackArr = new char[size];
        this.top = -1;
    }

    public void push(char i){
        stackArr[++top] = i;
    }

    public char pop(){
        return stackArr[top--];
    }

    public boolean isEmpty(){
        return (top == -1);
    }
}

class Bracket{
    private String input;

    public Bracket(String in){
        input = in;
    }

    public void check(){
        int size = input.length();
        Stack st = new Stack(size);

        for (int i=0; i<input.length(); i++){
            char ch = input.charAt(i);
            switch(ch){
                case '[':
                case '{':
                case '(':
                    st.push(ch);
                    break;
                case ']':
                case '}':
                case ')':
                    if (!st.isEmpty()){
                        char chr = st.pop();
                           if ((ch == '}' && chr != '{') || (ch == ']' && chr != 
'[') || (ch == ')' && chr != '(')){
                            System.out.println("Error: "+ch+" at "+i);
                        }
                    }else {
                        System.out.println("Error: "+ch+" at "+i);
                    }
                    break;
                default:
```

```
                    break;
                }
            }
            if (!st.isEmpty()){
                System.out.println("Error: missing right delimiter");
            }
        }
}

public class JavaStack {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) throws IOException{
        String input;
        while (true) {
            input = getString();
            if (input.equals("")) break;

            Bracket br = new Bracket(input);
            br.check();
        }

    }

    public static String getString() throws IOException{
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(isr);

        return br.readLine();
    }


}
```

Здесь реализован класс **Stack**, который содержит методы по добавлению и удалению элемента стека. Класс **Bracket** содержит метод проверки строки на соответствие скобок. В пользовательском классе создан метод **getString()**, который принимает строку, введенную с клавиатуры.

Очереди

Очереди напоминают стек, но в них первым извлекается элемент, который был добавлен первым.



Для описания очереди используют аббревиатуру FIFO (First In — First Out) — «первый вошел — первый вышел». Очереди могут использоваться при моделировании, например, банковских систем. Очереди активно применяются в Java Message Service — это спецификация, которая позволяет организовывать системы обмена сообщениями.

Реализуем основные методы вставки в конец очереди, удаления из начала, просмотра первого элемента очереди, возврата ее размера, а также проверок на пустоту и переполнение.

Опишем основные поля и конструктор очереди:

```
private int maxSize;
       private int[] queue;
       private int front;
       private int rear;
       private int items;

 public Queue(int s){
    maxSize = s;
    queue = new int[maxSize];
    front = 0;
    rear = -1;
    items = 0;

 }
```

В качестве очереди будем использовать массив. Переменная **front** — маркер начала очереди. **Rear** — маркер конца. **Items** — количество элементов в очереди.

Вспомогательные методы

```
public boolean isEmpty(){
      return (items==0);
 }

 public boolean isFull(){
      return (items==maxSize);
 }

 public int size(){
      return items;

 }
```

Метод **isEmpty** — проверка на пустоту очереди.

Метод **isFull** — проверка на переполнение.

Метод **size** — возвращает размер очереди.

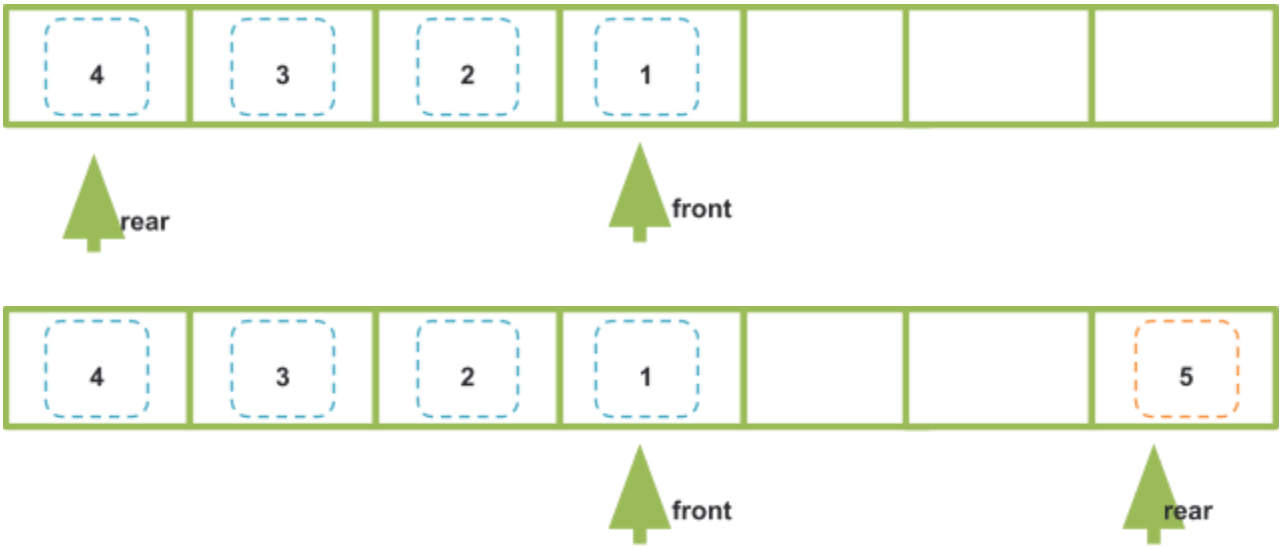
Циклический перенос

Прежде чем переходить к добавлению и удалению элемента очереди, необходимо рассмотреть ситуацию, связанную с маркерами **front** и **rear**. Представим, что наша очередь — это массив элементов.

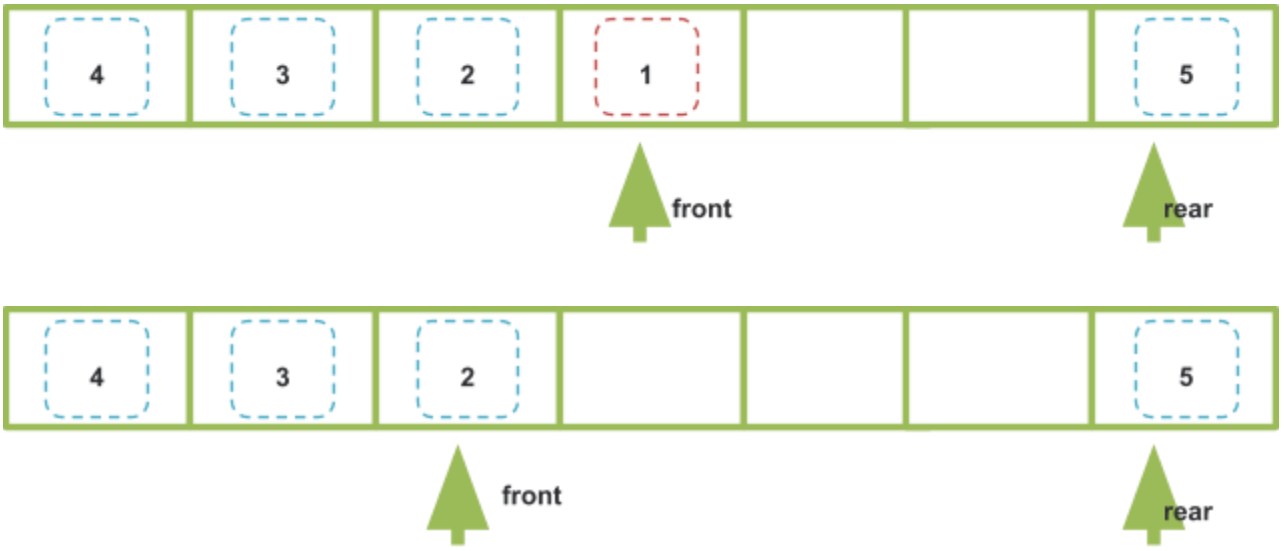
Чтобы добавить элемент в начало очереди, необходимо сдвинуть все элементы на единицу вправо.

Такая операция требует дополнительно переносить каждый элемент. А если очередь будет большой? Тогда и количество операций возрастет. Чтобы ускорить процесс вставки и удаления элементов очереди, используется циклический перенос.

Начало и конец очереди помечается маркерами **rear** и **front**. При вставке нового элемента остальные остаются на своих местах, а двигается маркер:



При удалении элемента двигается маркер **front**:



Рассмотрим методы **insert**, **remove** и **peek**.

Добавляем элемент

```
public void insert(int i){
        if(rear == maxSize-1)
        rear = -1;
        queue[++rear] = i;
        items++;

 }
```

Удаляем элемент

```
public long remove(){
    int temp = queue[front++];
    if(front == maxSize)
    front = 0;
    items--;
    return temp;

 }
```

Получаем элемент

```
public int peek(){
      return queue[front];

 }
```

Пример программы

Программа реализует очередь. Класс **Queue** содержит методы вставки, удаления и получения элемента очереди, а также проверку на ее пустоту, переполнение и размер. Создается очередь из десяти ячеек. Сначала заполняются пять из них, потом две удаляются и добавляются еще четыре.

```
class Queue{
    private int maxSize;
    private int[] queue;
    private int front;
    private int rear;
    private int items;

    public Queue(int s){
       maxSize = s;
       queue = new int[maxSize];
       front = 0;
       rear = -1;
       items = 0;
    }
    public void insert(int i){
       if(rear == maxSize-1)
       rear = -1;
       queue[++rear] = i;
       items++;
    }

    public int remove(){
        int temp = queue[front++];
        if(front == maxSize)
        front = 0;
        items--;
        return temp;
    }

    public int peek(){
```

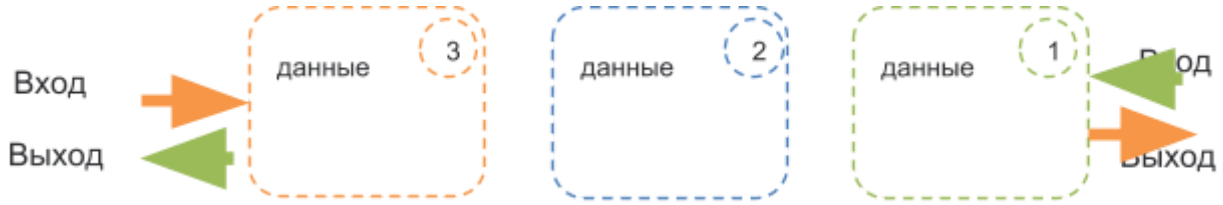
```
        return queue[front];
    }
    public boolean isEmpty(){
        return (items==0);
    }

    public boolean isFull(){
        return (items==maxSize);
    }

    public int size(){
        return items;
    }
}

public class Main{
    public static void main(String[] args){
        Queue q = new Queue(5);
        q.insert(10);
        q.insert(20);
        q.insert(30);
        q.insert(40);
        q.insert(50);
        q.remove();
        q.remove();

        q.insert(50);
        q.insert(60);
        q.insert(70);
        q.insert(80);
        while( !q.isEmpty() ){
            int n = q.remove();
        System.out.println(n);

        }
    }

}
```

В нашей реализации методы **insert** и **delete** могут привести к исключительной ситуации, поэтому их необходимо обернуть в конструкцию **if** и проверить на полноту или пустоту.

Эффективность очередей: вставка — **O(1),** удаление — **O(1)**.

Дек

Дек (**deque**) представляет собой двустороннюю очередь. И вставка, и удаление элементов могут производиться с обоих концов. Соответствующие методы могут называться **insertLeft()/insertRight()** и **removeLeft()/removeRight()**. Если ограничиться только методами **insertLeft()** и **removeLeft()** (или их эквивалентами для правого конца), дек работает как стек. Если же ограничиться методами **insertLeft()** и **removeRight()** (или противоположной парой), он работает как очередь. По гибкости деки превосходят и стеки, и очереди. Тем не менее, используются они реже стеков или очередей, поэтому подробно рассматривать мы их не будем.

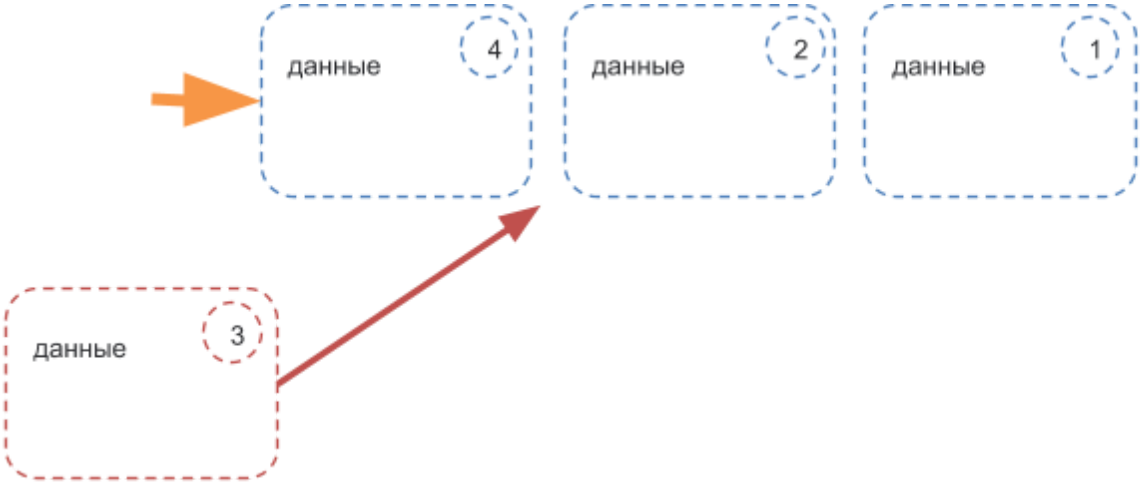


Приоритетные очереди

Приоритетная очередь похожа на обычную. У нее есть начало и конец, элементы извлекаются из начала очереди, а вот попадают в нее немного по-другому. Данные, которые находятся в приоритетной очереди, упорядочены по ключу. В начале очереди находится элемент, у которого ключ имеет минимальное значение — иначе говоря, наивысший приоритет. Когда происходит вставка нового элемента, он занимает позицию согласно своему ключу, чтобы не нарушить сортировку.

Можно провести аналогию с сортировкой почты. Каждый раз при получении писем их можно отсортировать по срочности. Срочные письма складываются наверх, а остальные — вниз стопки.

В компьютерных системах — например, Windows — каждому процессу, который выполняет программу, также можно задать соответствующий приоритет.



Реализуем приоритетную очередь. В качестве контейнера, который будет хранить элементы очереди, используем массив.

```
class PriorityQueue{
    private int maxSize;
    private int[] queueArray;
    private int items;

    public PriorityQueue(int i){
        maxSize = i;
        queueArray = new int[maxSize];
        items = 0;
    }
    public void insert(int item){
        int i;
        if(items==0)
            queueArray[items++] = item;
        else{
            for(i=items-1; i>=0; i--){
                if( item > queueArray[i] )
                    queueArray[i+1] = queueArray[i];
                else
                    break;
            }
            queueArray[i+1] = item; // Вставка элемента
            items++;
        }
    }
    public int remove(){
        return queueArray[--items];
    }
    public long peek(){
         return queueArray[items-1];
     }

    public boolean isEmpty(){
        return (items==0);
    }
    public boolean isFull(){
        return (items == maxSize);
    }
}
class PriorityQApp{
      public static void main(String[] args) throws IOException{
            PriorityQueue q = new PriorityQueue(5);
            q.insert(30);
            q.insert(50);
            q.insert(10);
            q.insert(40);
            q.insert(20);
            while( !q.isEmpty() )
            {
               int item = q.remove();
               System.out.print(item + " ");
            }
            System.out.println("");
    }

}
```

Эффективность приоритетных очередей:

- Вставка — **O(N)**;
- Удаление — **O(1)**.

Домашнее задание

1. Реализовать рассмотренные структуры данных в консольных программах.
2. Создать программу, которая переворачивает вводимые строки (читает справа налево).
3. Создать класс для реализации дека.

Дополнительные материалы

1. Стек и очередь.
2. Стек и очередь в JDK.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Лафорте Р. Структуры данных и алгоритмы в Java. Классика Computers Science. 2-е изд. — СПб.: Питер, 2013. — 121-178 сс.