

Практическая работа № 23. Абстрактные типы данных. Очередь

Цель: цель данной практической работы — научиться разрабатывать программы с абстрактными типами данных на языке Джава

Теоретические сведения. Очередь

1. Очередь можно предельно определить как упорядоченный список, который позволяет выполнять операции вставки на одном конце, называемом REAR, и операции удаления, которые выполняются на другом конце, называемом FRONT

2. Очередь называется работает по дисциплине обслуживания «первый пришел — первый обслужен» (FCFS, first come first served)

3. Например, люди, стоящие в кассу магазина образуют очередь оплаты покупок.

Пример очереди можно увидеть на рис. 23.1. Операция dequeue означает удаление элемента из начала очереди, а операция enqueue добавление элемента в конец очереди.

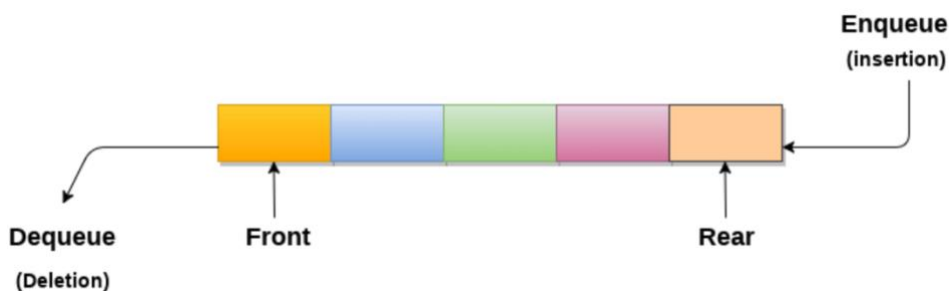


Рисунок 23.1. Общая схема работы очереди

Использование очередей в разработке программ.

Очереди широко используются

- в качестве списков ожидания для одного общего ресурса, такого как принтер, диск, ЦП;
- при асинхронной передаче данных (когда данные не передаются с одинаковой скоростью между двумя процессами), например. трубы, файловый ввод-вывод, сокеты;
- в качестве буферов в большинстве приложений, таких как медиаплеер MP3, проигрыватель компакт-дисков и т. д.;
- для ведения списка воспроизведения в медиаплеерах, чтобы добавлять и удалять песни из списка воспроизведения;

- в операционных системах для обработки прерываний и при реализации работы алгоритмов планирования и диспетчизации.

Таблица 23.1 Сравнение временной сложности для различных операций над очередью

	Временная сложность							
	Среднее				Худшее			
	Доступ	Поиск	Вставка	Удаление	Доступ	Поиск	Вставка	Удаление
Очередь	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$

Классификация очередей

Очередь — это структура данных, похожая на очередь в реальном мире.

Очередь — это структура данных, в которой элемент, который приходит в очередь первым, удаляется первым, то есть работа очереди строго соответствует политике FIFO (First-In-First-Out).

Очередь также можно определить как список или коллекцию, в которой вставка выполняется с одного конца, известного как конец очереди, тогда как удаление выполняется с другого конца, известного как начало очереди.

Реальным примером очереди является очередь за билетами возле кинозала, где человек, входящий в очередь первым, получает билет первым, а последний человек, входящий в очередь, получает билет последним. Аналогичный подход используется в очереди как в структуре данных.

На рис.23.2 представлена структура очереди из четырех элементов.



Рисунок 23.2. Очередь из четырех элементов

Виды очередей:

- Простая очередь или линейная очередь

- Циклическая очередь
- Очередь с приоритетами
- Двусторонняя очередь или Дек (англ. Deque)

Простая или линейная очередь

В линейной очереди или Queue вставка элемента происходит с одного конца, а удаление — с другого. Конец, на котором происходит вставка, называется задняя часть очереди, а конец, на котором происходит удаление, называется передняя часть очереди. Работа линейной очереди организуется по правилу FIFO – первый пришел-первый ушел.



Рисунок 23.3. Очередь из четырех элементов

Основным недостатком использования линейной очереди является то, что вставка выполняется только с заднего конца. Если первые три элемента будут удалены из очереди, мы не сможем вставить больше элементов, даже если в линейной очереди есть свободное место. В этом случае линейная очередь показывает состояние переполнения, поскольку задняя часть указывает на последний элемент очереди.

Циклическая очередь

В циклической очереди все узлы представлены как элементы цепочки. После последнего элемента очереди сразу идет первый или начальный элемент. Эта очередь похожа на линейную очередь, за исключением того, что последний элемент очереди соединяется с первым элементом. Эта очередь получила название кольцевого буфера, так как все ее концы соединены друг с другом. Схематично эта очередь представлена на рис. 23.4.

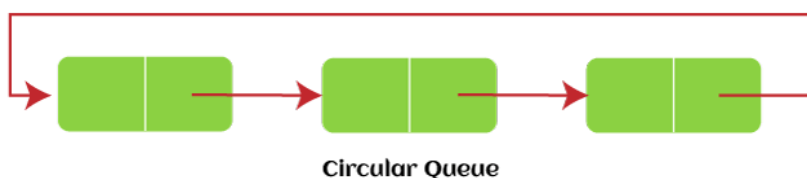


Рисунок 23.4. Кольцевая очередь

Недостаток линейной очереди преодолевается при использовании круговой очереди. Если в циклической очереди есть пустое место, новый

элемент можно добавить в пустое место, просто увеличив значение Rear.

Основным преимуществом использования циклической очереди является лучшее использование памяти.

Очередь с приоритетами

Это особый тип очереди, в которой элементы располагаются в зависимости от их приоритета. Это особый тип структуры данных очереди, в которой каждый элемент такой очереди имеет связанный с ним приоритет. Допустим, какие-то элементы встречаются с одинаковым приоритетом, тогда они будут располагаться по принципу FIFO. Представление очереди с приоритетами показано на рис.23.5.

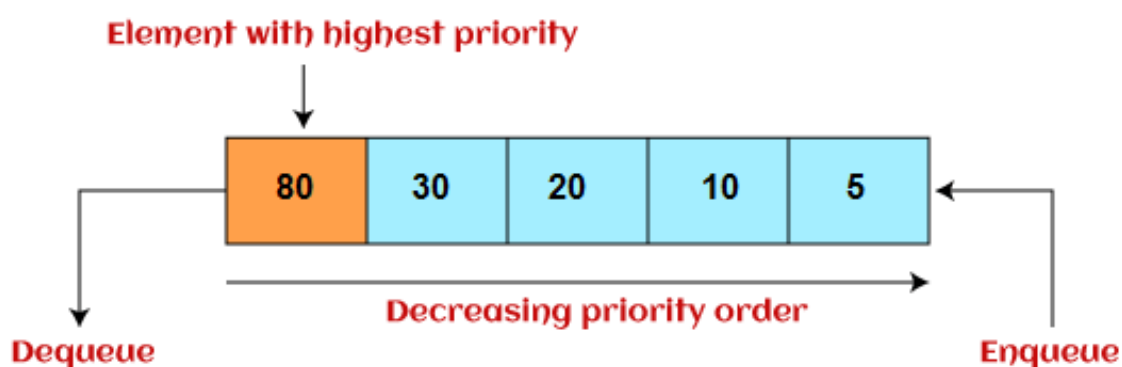


Рисунок 23.5. Очередь с приоритетами

Вставка в такую очередь происходит на основе поступления элемента в соответствии с его приоритетом, а удаление в данной очереди происходит на основе приоритета. Очередь с приоритетом в основном используется для реализации алгоритмов планирования ЦП в операционных системах.

Типы очередей с приоритетами

Существует два типа очереди, которые обсуждаются следующим образом:

- 1) Очередь с возрастающим приоритетом. В очереди с возрастающим приоритетом элементы могут быть вставлены в произвольном порядке, но только самые маленькие могут быть удалены первыми. Предположим массив с элементами 7, 5 и 3 в одном порядке, поэтому вставка может быть выполнена с той

же последовательностью, но порядок удаления элементов 3, 5, 7.

- 2) Очередь с убывающим приоритетом. В очереди с убывающим приоритетом элементы могут быть вставлены в произвольном порядке, но первым может быть удален только самый большой элемент, то есть с самым высоким приоритетом. Предположим, массив с элементами 7, 3 и 5 вставлены в одном порядке, поэтому вставка может быть выполнена с той же последовательностью, но порядок удаления элементов 7, 5, 3.

Deque (или двойная очередь)

В Deque или Double Ended Queue вставка и удаление могут выполняться с обоих концов очереди либо спереди, либо сзади. Это означает, что мы можем вставлять и удалять элементы как с переднего, так и с заднего конца очереди.

Deque можно использовать как средство проверки палиндрома, что означает, что если мы читаем строку с обоих концов, то строка будет одинаковой.

Deque можно использовать и как стек, и как очередь, поскольку он позволяет выполнять операции вставки и удаления на обоих концах. Deque можно рассматривать как стек, потому что стек следует принципу LIFO (Last In First Out), в котором вставка и удаление могут выполняться только с одного конца. А в структуре данных Дек можно выполнять и вставку, и удаление с одного конца, и нужно отметить что работа дека не следует правилу FIFO. Схематично представление дека представлено на рис. 23.6.



Рисунок 23.6. Очередь Дек

Существует два типа дека:

- Очередь ограниченным вводом
- Очередь с ограниченным выводом

Очередь с ограниченным вводом. Как следует из названия, в очереди

с ограниченным вводом операция вставки может выполняться только на одном конце, а удаление может выполняться на обоих концах.

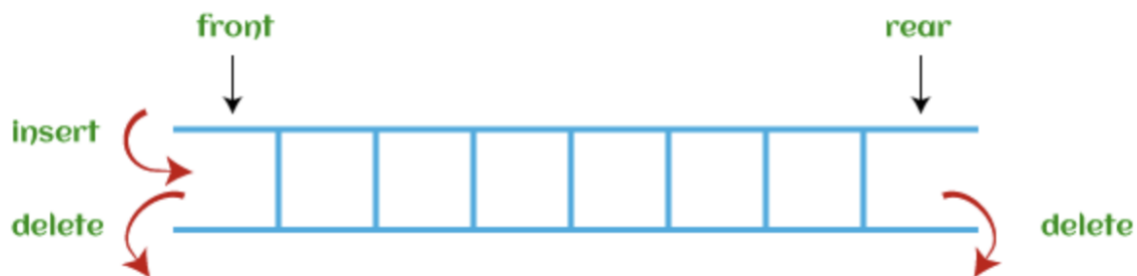


Рисунок 23.7. Очередь с ограниченным вводом

Очередь с ограничениями на вывод. Ограниченная очередь вывода — как следует из названия, в очереди вывода с ограниченным доступом операция удаления может выполняться только на одном конце, а вставка — на обоих концах.



Рисунок 23.8. Очередь с ограниченным выводом

Операции, выполняемые над очередью

Основные операции, которые можно выполнять в очереди, перечислены ниже:

- **Enqueue():** эта операция используется для вставки элемента в конец очереди. Возвращает пустоту.
- **Dequeue():** Операция выполняет удаление из внешнего интерфейса очереди. Он также возвращает элемент, который был удален из внешнего интерфейса. Он возвращает целочисленное значение.
- **Peek()** Просмотр очереди: это третья операция, которая возвращает элемент, на который указывает передний указатель в очереди, но не удаляет его.
- **isFull() (Queue overflow):** Проверка переполнения очереди (заполнено): показывает состояние переполнения, когда

очередь полностью заполнена.

- **isEmpty() (Queue underflow):** проверка очереди на пустоту
Показывает состояние потери значимости, когда очередь пуста, т. е. в очереди нет элементов.

Способы реализации очереди

Существует два способа реализации очереди:

- Реализация с использованием массива: последовательное размещение в очереди может быть реализовано с использованием массива.
- Реализация с использованием связанного списка: размещение связанного списка в очереди может быть реализовано с использованием связанного списка.

Листинг 23.1 Реализация очереди на языке Джава

```
public class Queue {
    int SIZE = 5;
    int items[] = new int[SIZE];
    int front, rear;

    Queue() {
        front = -1;
        rear = -1;
    }
    // check if the queue is full
    boolean isFull() {
        if (front == 0 && rear == SIZE - 1) {
            return true;
        }
        return false;
    }
    // check if the queue is empty
    boolean isEmpty() {
        if (front == -1)
            return true;
        else
            return false;
    }
}
```

```

// insert elements to the queue
void enQueue(int element) {
    // if queue is full
    if (isFull()) {
        System.out.println("Queue is full");
    }
    else {
        if (front == -1) {
            // mark front denote first element of
queue
            front = 0;
        }
        rear++;
        // insert element at the rear
        items[rear] = element;
        System.out.println("Insert " + element);
    }
}

// delete element from the queue
int deQueue() {
    int element;
    // if queue is empty
    if (isEmpty()) {
        System.out.println("Queue is empty");
        return (-1);
    }
    else {
        // remove element from the front of queue
        element = items[front];
        // if the queue has only one element
        if (front >= rear) {
            front = -1;
            rear = -1;
        }
        else {
            // mark next element as the front

```



```

        front++;
    }
    System.out.println( element + "
Deleted");
    return (element);
}
}
// display element of the queue
void display() {
    int i;
    if (isEmpty()) {
        System.out.println("Empty Queue");
    }
    else {
        // display the front of the queue
        System.out.println("\nFront index-> " +
front);
        // display element of the queue
        System.out.println("Items -> ");
        for (i = front; i <= rear; i++)
            System.out.print(items[i] + " ");
        // display the rear of the queue
        System.out.println("\nRear index-> " +
rear);
    }
}

public static void main(String[] args) {
    // create an object of Queue class
    Queue q = new Queue();
    // try to delete element from the queue
    // currently queue is empty
    // so deletion is not possible
    q.deQueue();
    // insert elements to the queue
    for(int i = 1; i < 6; i ++) {
        q.enqueue(i);
    }
}
}

```

```

        }
        // 6th element can't be added to queue
because queue is full
        q.enqueue(6);
        q.display();
        // dequeue removes element entered first i.e.
1
        q.dequeue();
        // Now we have just 4 elements
        q.display();
    }
}

```

Результат выполнения программы на листинге 23.1

```

Queue is empty
Insert 1
Insert 2
Insert 3
Insert 4
Insert 5
Queue is full

Front index-> 0
Items ->
1  2  3  4  5
Rear index-> 4
1 Deleted

```

```

Front index-> 1
Items ->
2  3  4  5
Rear index-> 4

```

Вам не обязательно самим писать АТД Очередь. Ведь язык Java предоставляет встроенный интерфейс Queue, который можно использовать для реализации очереди. Такая реализация представлена на листинге ниже.

Листинг 23.2 Реализация очереди с помощью интерфейса Queue

```

import java.util.Queue;
import java.util.LinkedList;

class Main {

    public static void main(String[] args) {
        // Creating Queue using the LinkedList class
        Queue<Integer> numbers = new LinkedList<>();

        // enqueue
        // insert element at the rear of the queue
        numbers.offer(1);
        numbers.offer(2);
        numbers.offer(3);
        System.out.println("Queue: " + numbers);

        // dequeue
        // delete element from the front of the queue
        int removedNumber = numbers.poll();
        System.out.println("Removed Element: " +
removedNumber);

        System.out.println("Queue after deletion: " +
numbers);
    }
}

```

Результат выполнения программы на листинге 23.2

Queue: [1, 2, 3]

Removed Element: 1

Queue after deletion: [2, 3]

Понятие инварианта в объектно-ориентированном программировании относится к некоторому набору условий или утверждений, которые должны выполняться на протяжении всей жизни объекта класса. Эти утверждения должны выполняться с момента вызова конструктора для создания объекта, в конце вызова каждого метода, изменяющего состояние объекта (сеттера) до конца жизни объекта. Эти

условия подтверждают, что поведение объекта неизменно в течение всей его жизни и что объект поддерживает свое четко определенное состояние, как и предполагалось при его проектировании. Однако инвариант не обязательно должен оставаться истинным во время выполнения метода, изменяющего состояние (сеттера), но должен оставаться истинным в конце его выполнения.

Инвариант класса - это просто свойство, которое выполняется для всех экземпляров класса, всегда, независимо от того, что делает другой код.

Задания на практическую работу №23

Замечания

Для выполнения данной практической работы вам необходимы следующие знания:

1 Классы в Джава

- ✓ Инвариант класса
- ✓ Задачи инкапсуляции

2 Интерфейсы в Джава

- ✓ Интерфейс как синтаксический контракт
- ✓ Интерфейс как семантический контракт

3 Абстрактные базовые классы и наследование

- ✓ Устранение дублирования
- ✓ Вынос изменяемой логики в наследников

Задание 1. Реализовать очередь на массиве

- Найдите инвариант структуры данных «очередь». Определите функции, которые необходимы для реализации очереди. Найдите их пред- и постусловия.
- Реализуйте классы, представляющие циклическую очередь с применением массива.
 - ✓ Класс `ArrayQueueModule` должен реализовывать один экземпляр очереди с использованием переменных класса.
 - ✓ Класс `ArrayQueueADT` должен реализовывать очередь в виде абстрактного типа данных (с явной передачей ссылки на экземпляр очереди).

- ✓ Класс `ArrayQueue` должен реализовывать очередь в виде класса (с неявной передачей ссылки на экземпляр очереди).
- ✓ Должны быть реализованы следующие функции(процедуры)/методы:
 - `enqueue` – добавить элемент в очередь;
 - `element` – первый элемент в очереди;
 - `dequeue` – удалить и вернуть первый элемент в очереди;
 - `size` – текущий размер очереди;
 - `isEmpty` – является ли очередь пустой;
 - `clear` – удалить все элементы из очереди.
- Инвариант, пред- и постусловия записываются в исходном коде в виде комментариев.
- Обратите внимание на инкапсуляцию данных и кода во всех трех реализациях.
- Напишите тесты реализованным классам.

Задание 2. Очередь на связанном списке

4 Определите интерфейс очереди `Queue` и опишите его контракт.

5 Реализуйте класс `LinkedListQueue` — очередь на связанном списке.


6 Выделите общие части классов `LinkedListQueue` и `ArrayQueue` в базовый класс `AbstractQueue`.

Дополнительные задания

Задание 3. Вычисление выражений

1. Разработайте классы `Const`, `Variable`, `Add`, `Subtract`, `Multiply`, `Divide` для вычисления выражений с одной переменной.
2. Классы должны позволять составлять выражения вида

```
new Subtract(new Multiply(new Const(2), new Variable("x")), new Const(3)).evaluate(5)
```

 **Замечание.** При вычислении такого выражения вместо каждой переменной подставляется значение, переданное в качестве параметра методу `evaluate` (на данном этапе имена переменных игнорируются). Таким образом, результатом вычисления приведенного примера должно стать число 7.

3. Для тестирования программы должен быть создан класс `Main`, который вычисляет значение выражения $x^2 - 2x + 1$, для x , заданного в командной строке.

4. При выполнении задания следует обратить внимание на:
- Выделение общего интерфейса создаваемых классов.
 - Выделение абстрактного базового класса для бинарных операций.

Задание 4

1. Доработайте предыдущее задание, так чтобы выражение строилось по записи вида $x * (y - 2) * z + 1$
2. Для этого реализуйте класс `ExpressionParser` с методом `TripleExpression parse(String)`.
3. В записи выражения могут встречаться: умножение `*`, деление `/`, сложение `+`, вычитание `-`, унарный минус `-`, целочисленные константы (в десятичной системе счисления, которые помещаются в 32-битный знаковый целочисленный тип), круглые скобки, переменные (`x`, `y` и `z`) и произвольное число пробельных символов в любом месте (но не внутри констант).
4. Приоритет операторов, начиная с наивысшего
 - унарный минус;
 - умножение и деление;
 - сложение и вычитание.
5. Для выражения $1000000 * x * x * x * x * x / (x - 1)$ вывод программы должен иметь следующий вид:

x	f
0	0
1	division by zero
2	32000000
3	121500000
4	341333333
5	overflow
6	overflow
7	overflow
8	overflow
9	overflow
10	overflow

Ограничения

1. Результат `division by zero (overflow)` означает, что в процессе вычисления произошло деление на ноль (переполнение).
2. Разбор выражений рекомендуется производить методом рекурсивного спуска. Алгоритм должен работать за линейное время.

3. При выполнении задания следует обратить внимание на дизайн и обработку исключений.
4. Человеко-читаемые сообщения об ошибках должны выводиться на консоль.
5. Программа не должна «вылетать» с исключениями (как стандартными, так и добавленными)