МИНИСТЕРСТВО ТРАНСПОРТА РФ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «РОССИЙСКИЙ УНИВЕРСИТЕТ ТРАНСПОРТА (МИИТ)»

Кафедра УТБиИС

**Отчет**

По курсовой работе по теме:

«Структуры данных»

Дисциплина:

«Применение математических методов к решению прикладных задач»

Выполнил:

Студент группы УВП-311,

Новопашенный М. В.

Проверил:

Заманов Е.А.

Москва 2020 г.

Содержание:

1. Описание курсовой работы
2. Листинг кода
3. Вывод
4. **Описание курсовой работы**

В курсовой работе я реализовал структуру данных Двусвязный список (DoublyLinkedList). В качестве рода деятельности я выбрал завод запчастей для автомобилей. В курсовой работе реализован конвейер, на котором после изготовления размещаются детали. В изготовительный станок вводится название детали и ее размер. Каждая третья деталь оказывается неисправной во время изготовления и на конвейер в склад отправляются только исправные детали.

1. **Листинг кода программы**

**public class** Main {

**public static void** main(String[] args) {

Factory factory = **new** Factory();

DoublyLinkedList<Detail> details = **new** DoublyLinkedList<Detail>();

factory.addDetailsToManufactureConveyor(details);

factory.manufacture(details);

factory.addToOutputConveyor(details);

}

}

**public class** Detail **implements** Comparable<Detail> {

**private final** String **title**;

**private final int size**;

**private boolean isBroken** = **false**;

**public** Detail(String title, **int** size) {

**this**.**title** = title;

**this**.**size** = size;

}

**public boolean** isBroken() {

**return isBroken**;

}

**public void** setBroken(**boolean** broken) {

**isBroken** = broken;

}

@Override

**public** String toString() {

**return "Title - "** + **title** +

**", Size - "** + **size** +

**", Broken - "** + **isBroken**;

}

@Override

**public int** compareTo(Detail o) {

**if** (!**this**.**isBroken** && o.**isBroken**) **return** 1;

**if** (!**this**.**isBroken** && **this**.**size** > o.**size**) **return** 1;

**if** (**this**.**isBroken** && !o.**isBroken**) **return** -1;

**if** (**this**.**isBroken** && **this**.**size** < o.**size**) **return** -1;

**return** 0;

}

}

**public class** Node<T> {

**private** Node<Detail> **previous**;

**private** Node<Detail> **next**;

**private** Detail **element**;

**public** Node(Detail element) {

**this**.**element** = element;

}

**public** Node<Detail> getPrevious() {

**return previous**;

}

**public void** setPrevious(Node<Detail> previous) {

**this**.**previous** = previous;

}

**public** Node<Detail> getNext() {

**return next**;

}

**public void** setNext(Node<Detail> next) {

**this**.**next** = next;

}

**public** Detail getElement() {

**return element**;

}

**public void** setElement(Detail element) {

**this**.**element** = element;

}

}

**public class** MyIterator **implements** Iterator<Detail> {

**private** DoublyLinkedList<Detail> **list**;

**int i**=-1;

**public** MyIterator(DoublyLinkedList<Detail> list) {

**this**.**list** = list;

}

@Override

**public boolean** hasNext() {

**return list**.getNode(**i**).getNext() != **null**;

}

@Override

**public** Detail next() {

Detail element = **list**.getNode(**i**).getNext().getElement();

**i**++;

**return** element;

}

@Override

**public void** remove() {

}

}

**public class** DoublyLinkedList<T> {

**private** Node<Detail> **head**;

**private** Node<Detail> **last**;

**private int size**;

**public** DoublyLinkedList() {

}

**public** MyIterator getIterator(DoublyLinkedList<Detail> details) {

**return new** MyIterator(details);

}

**public void** addElement(Detail element) {

Node<Detail> node = **new** Node<Detail>(element);

**if** (**size** == 0) {

**head** = node;

} **else** {

**last**.setNext(node);

node.setPrevious(**last**);

}

**last** = node;

**size**++;

}

**public** Node<Detail> getNode(**int** index) {

Node<Detail> current = **null**;

**if** (index >= 0 || index < **size**) {

current = **head**;

**int** k = 0;

**while** (k < index) {

current = current.getNext();

k++;

}

}

**return** current;

}

**private** Detail getElement(**int** index) {

**return** getNode(index).getElement();

}

**public void** insertBefore(Detail element, **int** index) {

Node<Detail> node = **null**;

**if** (index == **size**) {

addElement(element);

**return**;

}

node = **new** Node<Detail>(element);

**if** (index == 0) {

node.setNext(**head**);

**head**.setPrevious(node);

**head** = node;

**size**++;

**return**;

}

Node<Detail> current = getNode(index);

current.getPrevious().setNext(node);

node.setPrevious(current.getPrevious());

node.setNext(current);

current.setPrevious(node);

**size**++;

}

**public void** removeAtIndex(**int** index) {

Node<Detail> node = getNode(index);

**if** (**head** == node) {

**if** (**head**.getNext() != **null**) {

**head**.getNext().setPrevious(**null**);

**head** = **head**.getNext();

} **else** {

**head** = **null**;

}

} **else if** (**last** == node) {

**if** (**last**.getPrevious() != **null**) {

**last**.getPrevious().setNext(**null**);

**last** = **last**.getPrevious();

} **else** {

**last** = **null**;

}

} **else** {

node.getPrevious().setNext(node.getNext());

node.getNext().setPrevious(node.getPrevious());

}

**size**--;

}

**private void** setElement(Detail val, **int** index) {

getNode(index).setElement(val);

}

@Override

**public** String toString() {

Node<Detail> current = **head**;

StringBuilder s = **new** StringBuilder();

**while** (current != **null**) {

s.append(current.getElement().toString()).append(**", "**);

current = current.getNext();

}

**return** String.*format*(**"TwoWayList: { count = %d , elements ={ %s }}"**, **size**, s.toString());

}

**public** Node<Detail> getHead() {

**return head**;

}

**public** Node<Detail> getLast() {

**return last**;

}

**public int** getSize() {

**return size**;

}

}

**public class** MyIterator **implements** Iterator<Detail> {

**private** DoublyLinkedList<Detail> **list**;

**int i**=-1;

**public** MyIterator(DoublyLinkedList<Detail> list) {

**this**.**list** = list;

}

@Override

**public boolean** hasNext() {

**return list**.getNode(**i**).getNext() != **null**;

}

@Override

**public** Detail next() {

Detail element = **list**.getNode(**i**).getNext().getElement();

**i**++;

**return** element;

}

@Override

**public void** remove() {

}

}

**public class** Factory {

**public void** addDetailsToManufactureConveyor(DoublyLinkedList<Detail> details) {

details.addElement(**new** Detail(**"motor"**, 10));

details.addElement(**new** Detail(**"wheel"**, 2));

details.addElement(**new** Detail(**"gear"**, 5));

details.addElement(**new** Detail(**"switcher"**, 1));

details.addElement(**new** Detail(**"stick"**, 3));

details.addElement(**new** Detail(**"bumper"**, 6));

details.addElement(**new** Detail(**"fender"**, 3));

details.addElement(**new** Detail(**"tube"**, 2));

details.addElement(**new** Detail(**"butterfly door"**, 4));

}

**public void** manufacture(DoublyLinkedList<Detail> details) {

**for** (**int** i = 0; i < details.getSize(); i++) {

details.getNode(i).getElement().setBroken(i % 3 == 0);

}

}

**public void** addToOutputConveyor(DoublyLinkedList<Detail> details) {

**for** (MyIterator it = details.getIterator(details); it.hasNext(); ) {

Detail detail = it.next();

**if** (detail.isBroken()) {

it.remove();

} **else** {

System.***out***.println(detail);

}

}

}

}

Скачать код можно по ссылке: <https://github.com/comougi/conveyor-comparator>

1. **Вывод:**

В результате выполнения курсового проекта, я работал с двусвязным списком. У этой структуры данных есть преимущества и недостатки. Я считаю, что основным преимуществом двусвязного списка является быстрое внесение изменений в середину списка. Динамический массив теоретически мог вызывать смещение каждого элемента, а связанный список сохраняет каждый другой объект на своём месте. Тем не менее двусвязный список имеет ряд недостатков. Один из них это то, что элементы в памяти соединены в цепочку. Если нам нужен определенный элемент, то мы идем по цепочке от первого элемента к искомому. Это нагружает память.