Ознакомление с динамическими структурами данных, в Java существует только один встроенный на уровне языка контейнер и тот статичен-массив. Для примера в Python встроено на уровне языка ???? контейнеров…Следовательно возникает необходимость создавать собственные контейнеры данных. Для корректной работы с контейнерами разных типов хранения данных, необходимо абстрагироваться от их особенностей, для этого и применяется шаблон Итератор. Шаблон Iterator позволяет перебирать элементы контейнеров, вне зависимости от способа их хранения, его задача - инкапсуляция имплементации способа перебора элементов, скрытие архитектуры хранения контейнера данных от конечного пользователя и предоставление высокоуровневого интерфейса для перебора элементов.

Начнем с создания статичного контейнера с ограниченным размером (Static Basket), строить который будем на основании статичного инкапсулированного массива, дефолтный размер которого будет задаваться константой DEFAULT\_SIZE, и использоваться соответственно в дефолтном конструкторе. Для отслеживания реального количества объектов создаем внутреннюю переменную size, в дефолтном конструкторе, задаем ей значение 0. Создаем также дополнительный конструктор, который в качестве параметра принимает общее зарезервированное пространство capacity, для того чтобы извне задать необходимое место в контейнере. Тж создаем конструктор, с возможностью передачи готового массива продуктов.

public class StaticBasket {  
 public static final int *DEFAULT\_SIZE* = 10;  
 private Product[] array;  
 private int size;  
  
 public StaticBasket() {  
 array = new Product[*DEFAULT\_SIZE*];  
 size = 0;  
 }  
  
 public StaticBasket(int capacity) {  
 array = new Product[capacity];  
 size = 0;  
 }  
  
 public StaticBasket(Product[] array) {  
 this.array = array;  
 size = array.length;  
 }

Для обeспечения доступа к элементам для ShopAssistance, реализуем возможность добавлять элементы🡪 реализуем методы set&get и методы возвращающие зарезервированное пространство capacity, и фактический размер контейнера size(в данном случае, в качестве имени методов используется им. сущ., т.к. он не производит никаких вычислений, это удобно при вычислении алгоритмической сложности. Т.к. сразу понятно, что операция этого метода константна и на АС не влияет).

public Product getProduct(int index) {  
 return array[index];  
}  
  
public void setProduct(int index, Product product) {  
 array[index] = product;  
}  
  
public int capacity() {return array.length;}  
  
public int size() {return size;}

Реализуем метод add :

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Prod | null | null | null | null | null | null | null | null |

size = 0; количество реальных объектов

capacity общая вместимость контейнера

add() size = 1 capacity = 9; – зарезервированное пространство

public void add(Product product) {  
 if (product == null) {  
 return;  
 }  
 if (size < array.length) {  
 array[size++] = product;  
 }  
  
}

Реализуем метод remove по индексу, дополняя функционал клааса методом для проверки наличия содержимого корзины, для возможности использования простого метода add, необходима грамотная реализация remove, обеспечивающая корректность size, по-этому удаление объектов реализуем путем смещения элементов с заменой удаленного:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Prod | Prod | Prod | Prod | Prod | null | null | null | null |

size = 5 ; capacity = 9;

remove();

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Prod | Prod | null | Prod | Prod | null | null | null | null |

size = 4; capacity = 9;

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Prod | Prod | Prod | Prod | null | null | null | null | null |

public boolean isEmpty() {  
 return size == 0;  
}  
  
//!!!  
public void remove(int index) {  
 if (isEmpty() || index == 0 && index >= size) {  
 return;  
 }  
 if (index == size - 1) {  
 array[index] = null;  
 } else {  
 for (int i = index; i < size(); i++) {  
 array[i] = array[i + 1];  
 }  
 }  
  
 size--;  
}

переходим к toString():

@Override  
public String toString() {  
 StringBuilder builder = new StringBuilder();  
  
 if (isEmpty()) {  
 builder.append("Basket is empty.");  
 } else {  
 builder.append("Basket:");  
 for (int i = 0; i < size; i++) {  
 builder.append("\n").append(array[i]);  
 }  
 }  
  
 return builder.toString();  
}

Используем стандартный for, чтобы получать описание только существующих объектов (до size)

Теперь переходим к реализации корзины, с помощью односвязного списка (динамическая структура данных):

Рассмотрим недостатки динамичного контейнера, реализованного при помощи массива:

- при необходимости расширения массива (включая вставку элемента по индексу)или удаления объектов, необходимо создавать новый объект массива, тк массивы по природе своей статичны

🡪сильно проигрывают на производительности

add🡪O(N)

remove🡪O(N)

Преимущества:

-индексный доступ за константное время

get🡪O(1)

Перейдем к рассмотрению особенностей списков:

Сама переменная хранит ссылку только на первый элемент, остальные элементы хранятся в любых доступнах ячейках памяти, каждый элемент содержит 2 раздела: в 1 хранится сам объект, а во 2м тех данные, ссылку на следующий элемент, отсюда и название , тк каждый элемент содержит ссылку на следующий, те имеет одну связь с ним, последний элемент содержит ссылку на null, в кольцевом – на 1й объект

add🡪O(1)создается новый элемент и перенастраиваются ссылки

remove🡪O(N)просто меняется ссылка в преыдущем элементе

get🡪O(N)тк для получения адреса по индексу надо обратится ко всем предыдущим элементам

4 add

List 1 @20 @90

First-var @10 2 @70 3 @78

0 @10 4 @123

remove

Кольцевой @10

Двухсвязный список-элементы содержат еще одно поле в котором хранится адрес предыдущего элемента:

add🡪O(1)

remove🡪O(1)

get🡪O(N)

List 1 @20

First-var @10 2 @70 3 @78

0 @10 4 @123

@123 @10

Кольцевой

Создаем инкапсулированный класс элемента (узла), в которм реализуем 2 поля – объект и связь со следующим элементом (если объекты класса используются только для инфраструктуры одного класса, то созданный класс (используются любые модификаторы доступа) обычно описывается внутри соответствующего(изучить внутренние, вложенные и анонимные классы)-разделение ответственности)

public class ListBasket {  
 private class Node{  
 public Product product;  
 public Node next;  
  
 public Node(Product product) {  
 this.product = product;  
 }  
 }  
}

Реализуем список, аналогично статичной корзине, реализуем только один дефолтный конструктор , в котором size = 0, а first = null, реализуем базовые методды возвращающие количество элементов и состояние корзины:

private Node first;  
private int size;  
  
public ListBasket() {  
 first = null;  
 size = 0;  
}  
  
public int size() {  
 return size;  
}  
  
public boolean isEmpty() {  
 return size == 0;  
}

Реализуем add @12

ListBasket:

1)add(m);

Node first.product = m (@12);

Node.next = null;

2)add(o); @12 @27

ListBasket:

Node first.product = m (@12);

Node.next = null;

public void add(Product product) {  
 if (isEmpty()) {  
 first = new Node(product);  
 } else {  
 Node temp = first;  
 while (temp.next != null) {  
 temp = temp.next;  
 }  
 temp.next = new Node(product);  
 }  
}

Реализуем toString:

@Override  
public String toString() {  
 StringBuilder builder = new StringBuilder();  
  
 if (isEmpty()) {  
 builder.append("Basket is empty.");  
 } else {  
 builder.append("Basket:");  
 Node temp = first;  
 while (temp.next != null) {  
 builder.append("\n").append(temp.product);  
 temp = temp.next;  
 }  
 builder.append(temp.product);  
 }  
  
 return builder.toString();  
}

Используем стандартный for, чтобы получать описание только существующих объектов (до size)