

Algorithms and data structures

lecture #6. Dynamic Arrays (Array List) and amortized analysis
Mentor: <....>

lecture #6. Dynamic Arrays (ArrayList) and amortized analysis

- Amortized analysis

- Общие понятия
- Где используется
- Амортизированная стоимость
- Методы AA
- AA операции add в динамический массив

- Dynamic Arrays


- Как это работает
- Особенности динамического массива
 - Добавить
 - Удалить
 - Изменить размер
- ArrayList как структура
- ArrayList и Array
- Базовые операции
- Доступные методы ArrayList Class



Amortized analysis

Амортизированный анализ - используется для алгоритмов, в которых отдельные операции выполняются очень медленно, но большинство других операций выполняются быстрее.

В амортизированном анализе мы анализируем последовательность операций и гарантируем среднее время наихудшего случая, которое ниже, чем время наихудшего случая конкретной дорогостоящей операции.



Амортизированный анализ - используется для алгоритмов, в которых отдельные операции выполняются очень медленно, но большинство других операций выполняются быстрее.

DynamicArray

Add: от $O(1)$ до $O(n)$.

1. Добавление элемента в конец – $O(1)$;
2. Добавление элемента по индексу – $O(n/2)$;
3. Добавление элемента в начало – $O(n)$;

Set: $O(1)$.

1. Изменение элемента – $O(1)$;

Remove: от $O(1)$ до $O(n)$.

1. Удаление последнего – $O(1)$;
2. Удаление элемента по индексу – $O(n/2)$;
3. Удаление первого – $O(n)$;

В амортизированном анализе мы анализируем последовательность операций и гарантируем среднее время наихудшего случая, которое ниже, чем время наихудшего случая конкретной дорогостоящей операции.

Основной идеей амортизационного анализа является то, что любая трудоёмкая операция меняет состояние программы таким образом, что до следующей трудоёмкой операции обязательно пройдёт достаточно много мелких, тем самым «амортизируя» вклад трудоёмкой операции.

Вставляем 1 элемент	1							
Вставляем 2 элемент	1	2						
Вставляем 3 элемент	1	2	3					
Вставляем 4 элемент	1	2	3	4				
Вставляем 5 элемент	1	2	3	4	5			
Вставляем 6 элемент	1	2	3	4	5	6		
Вставляем 7 элемент	1	2	3	4	5	6	7	

- 1) Выделите память для таблицы большего размера, обычно в два раза больше старой таблицы.
- 2) Скопируйте содержимое старой таблицы в новую таблицу.
- 3) Освободите старую таблицу.



Используется с такими структурами данных:



Хэш таблицы (HashTable, HashMap)

Непересекающиеся наборы (ArrayList, LinkedList)

Расширенные деревья (Splay tree, AVL)

Компромисс пространства и времени:

если мы делаем размер хэш-таблицы большим, время поиска становится меньше, но требуемое пространство становится большим




Методы анализа:

Групповой/Агрегированный (aggregate analysis)

Метод бухгалтерского учета (accounting method)

Метод потенциалов (potential method)

Все методы позволяют получить одну и ту же оценку,
но разными способами



Простой
вопрос?

Как создать бесконечный цикл на пустом месте?


```
public static void main(String[] args) {  
    // для решения добавить одну строку здесь  
    for (int i = start; i <= start + 1; i++) {  
        // тут должен быть бесконечный цикл, менять или добавлять здесь ничего нельзя!  
    }  
}
```




– Dynamic arrays

Динамический массив – это массив, который автоматически увеличивается, когда мы пытаемся сделать вставку, и для нового элемента больше не осталось места.

Динамический массив используется для обработки наборов однородных данных, размер которых неизвестен на момент написания программы.



Ключевые особенности динамического массива

1. Добавить элемент

- Добавьте элемент в конец, если размера массива недостаточно, увеличьте размер массива и добавьте элемент в конец исходного массива. Выполнение всего этого копирования занимает $O(n)$ времени, где n — количество элементов в нашем массиве.
- Это дорогая стоимость для приложения.
- В массиве фиксированной длины добавление занимает всегда время $O(1)$.
- Но добавление занимает время $O(n)$ только тогда, когда мы вставляем в полный массив. И это довольно редко, особенно если мы удваиваем размер массива каждый раз, когда нам не хватает места.
- Таким образом, в большинстве случаев добавление по-прежнему занимает время $O(1)$, и только иногда время = $O(n)$.

Простой вопрос?

Статический массив:

1. Создаем массив – `new int[10];`

1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----



Что произойдет при вставке числа 11 в ячейку с индексом 10

Динамический массив:

1. Создаем массив – `new DynamicArray(10);`

1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----



Что произойдет при вставке числа 11 в ячейку с индексом 10

Вопрос?

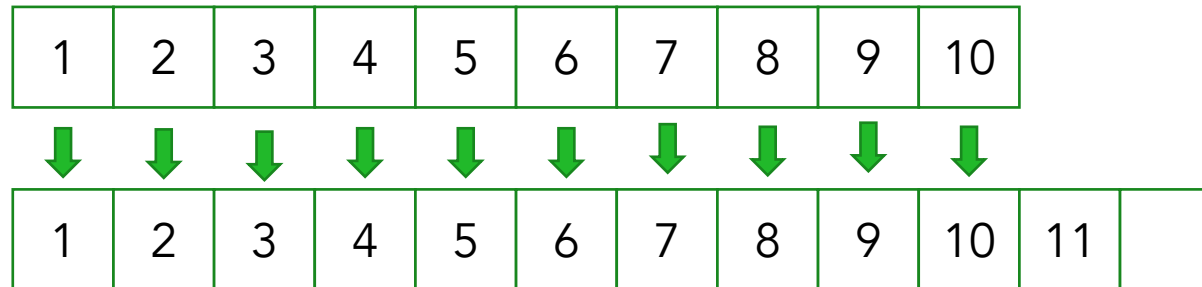
Статический массив:

1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----



`java.lang.ArrayIndexOutOfBoundsException: Index 11 out of bounds for length 10`

Динамический массив:



В памяти создается еще один массив, увеличенной емкости в n -раз
Старый копируется в новый



Ключевые особенности динамического массива

2. Удалить элемент

- По умолчанию удаляет элемент с конца, просто сохраняет ноль в последнем индексе
- Удалить элемент по определенному индексу, все правые элементы сдвигаются в левую часть от заданного индекса.

3. Изменить размер

- При удалении элементов размер может не уменьшаться
- Или массив имеет нулевые данные в правой части массива, занимающего нераспределенную память

Амортизированный анализ операции add

```
function add(x)
if Size = 0 then Size = 1, Array = new Array(1), and
count = 0
do Array[count] = x, count = count + 1
else if
count = Size then Size = Size * 2, NewArray= new
Array(Size) for i = 0 to count - 1
do NewArray[i] = Array[i]
end for clean(Array), Array = NewArray
end if Array[count] = x, count = count + 1
end function
```

- 1 call – 1 операция
- 2 call – 2 операции
- 3 call – 3 операции
- 4 call – 1 операция
- 5 call – 5 операций
-

Амортизированная стоимость – это стоимость операции

1									
1	2								
1	2	3							
1	2	3	4						
1	2	3	4	5					
1	2	3	4	5	6				
1	2	3	4	5	6	7			

Номер вставки:	1	2	3	4	5	6	7	8	9	10
Размер таблицы:	1	2	4	4	8	8	8	8	16	16
Стоимость:	1	2	3	1	5	1	1	1	9	1

Применим Амортизированный анализ операции add в динамический массив (Агрегированный метод – все взять и поделить)

Два метода усреднения для динамического массива

1. Аддитивная схема

Динамический массив увеличивает размер константно или случайно.

При такой схеме усредненное время для `add()` одного элемента = $O(n)$

2. Мультипликативная схема

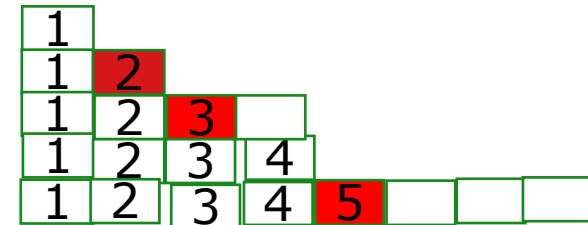
K - растет геометрически

Итак возьмем за основу роста удвоение размера ($K = 2$)

сколько элементов нам нужно будет скопировать в процессе добавления $N+1$: $1+2+4+\dots+N = 2N-1 \approx O(N)$

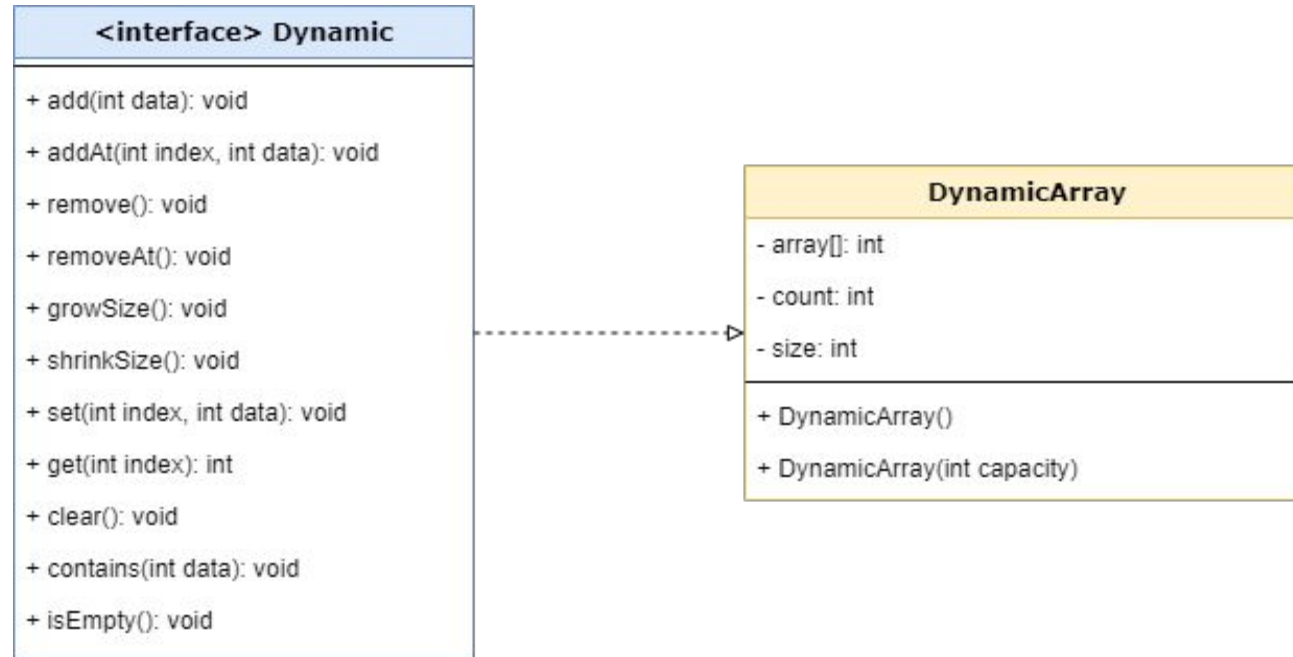
Таким образом, копирование элементов при изменении размера занимает всего $O(N)$ времени. Разделив это на количество вставок $N+1$, оценка **сверху** амортизированной сложности каждого вызова `add()` будет равна:

$$\frac{O(N)}{N+1} \approx O(1)$$



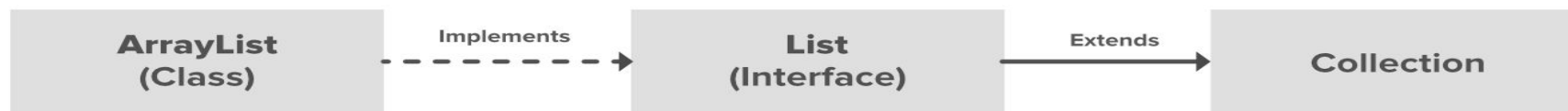
$K = 1$
$K = 2$
$K = 4$
$K = 8$
$K = 16$

Задача: реализовать Динамический массив



ArrayList

ArrayList является частью структуры <коллекции> и находится в пакете java.util. Это и есть классический динамический массив.



Важные особенности:

- ArrayList реализует интерфейс List.
- ArrayList инициализируется размером. Однако размер увеличивается автоматически, если коллекция увеличивается или уменьшается, если объекты удаляются из коллекции.
- ArrayList позволяет нам произвольно обращаться к списку.
- ArrayList **нельзя использовать для примитивных типов**, таких как int, char и т. д. Для таких случаев нам нужен класс-оболочка.
- ArrayList **не синхронизирован**. Его эквивалентный синхронизированный класс в Java — Vector. (используем Concurrent collection API из пакета java.util.concurrent)



Конструкторы в ArrayList

1. `ArrayList()`: этот конструктор используется для создания пустого списка массивов. Если мы хотим создать пустой `ArrayList` с именем `arr`, то его можно создать как:

```
ArrayList list = new ArrayList();
```

2. `ArrayList(Collection c)`: этот конструктор используется для построения списка массивов, инициализированных элементами из коллекции `c`. Предположим, мы хотим создать массив `ArrayList`, который содержит элементы, присутствующие в коллекции `c`, тогда его можно создать как:

```
ArrayList list = new ArrayList(c);
```

3. `ArrayList(int capacity)`: этот конструктор используется для создания списка массивов с указанием начальной емкости. Предположим, мы хотим создать `ArrayList` с начальным размером `N`, тогда его можно создать как:

```
ArrayList list = new ArrayList(N);
```

Простой вопрос?

В array при создании мы обозначаем тип данных хранимых в этом массиве

```
int[] = new int[1];
```

```
String[] = new String[2];
```

```
Object[] = new Object[3];
```

Сможем добавить те же данные в ArrayList?

```
ArrayList list = new ArrayList();
```

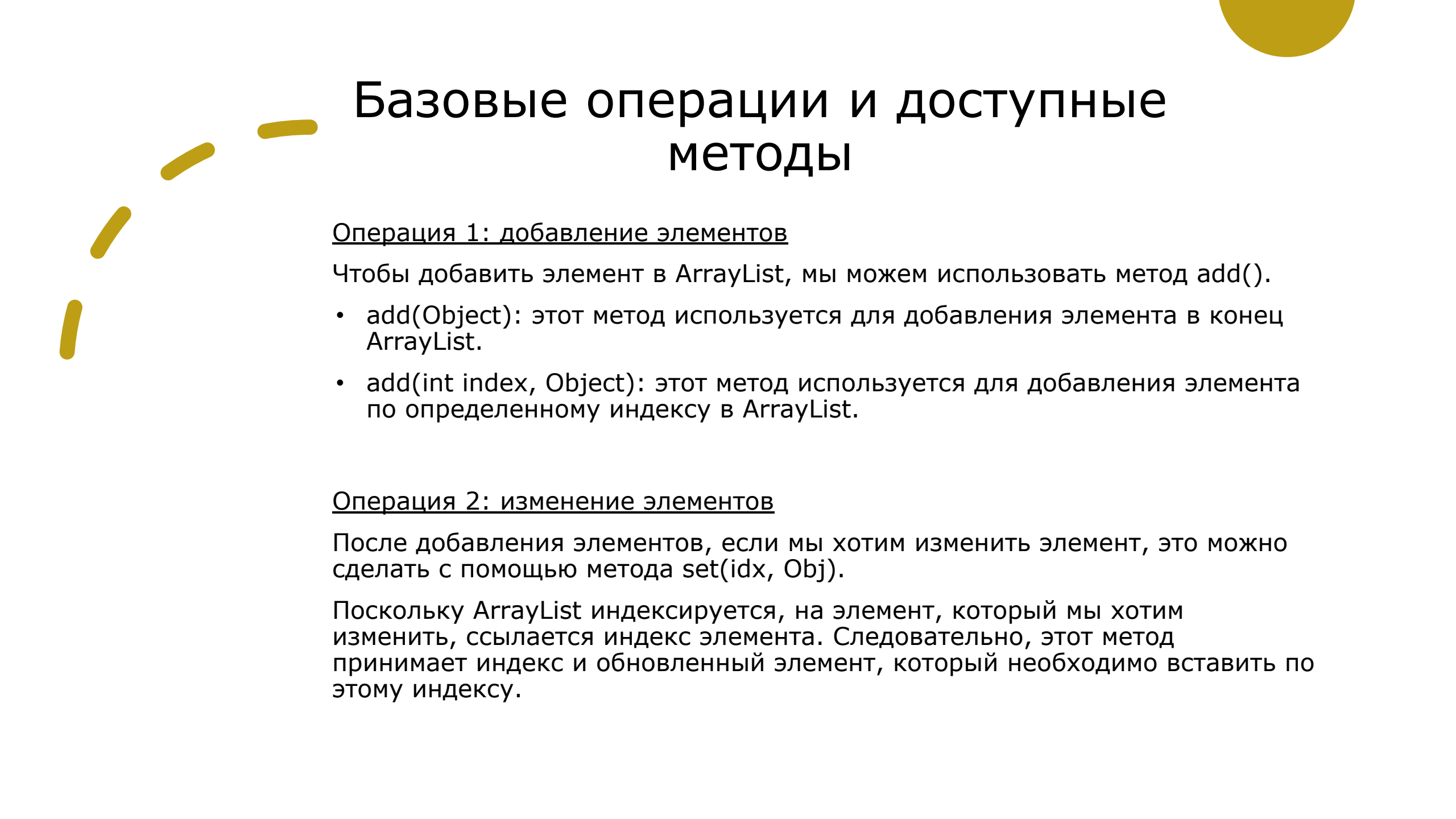
```
list.add(1);
```

```
list.add("str");
```

```
list.add(new Object());
```

```
boolean isInt = list.get(0) instanceof Integer ? true : false;
```

isInt будет правдой или ложью?



Базовые операции и доступные методы

Операция 1: добавление элементов

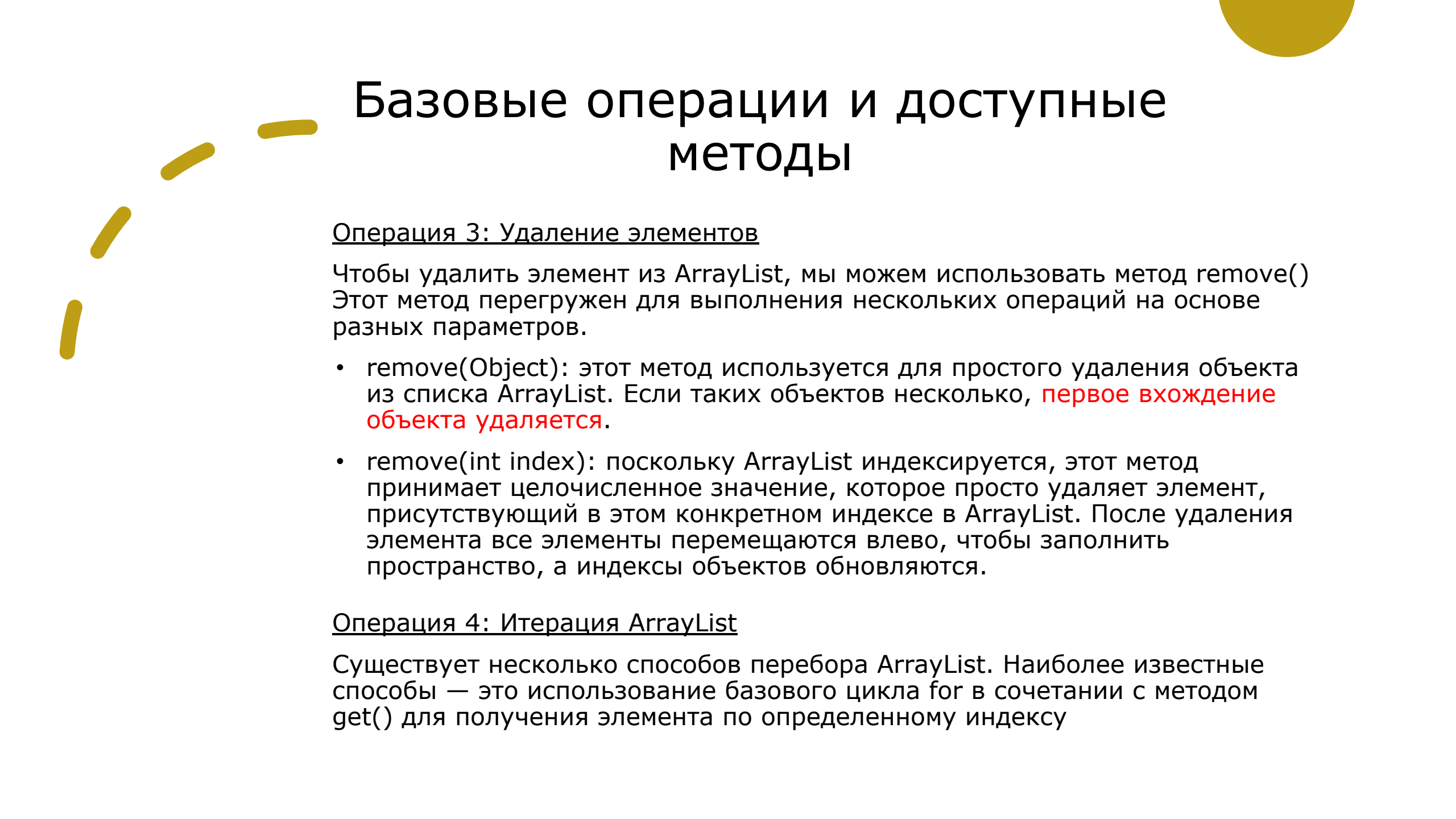
Чтобы добавить элемент в ArrayList, мы можем использовать метод add().

- add(Object): этот метод используется для добавления элемента в конец ArrayList.
- add(int index, Object): этот метод используется для добавления элемента по определенному индексу в ArrayList.

Операция 2: изменение элементов

После добавления элементов, если мы хотим изменить элемент, это можно сделать с помощью метода set(idx, Obj).

Поскольку ArrayList индексируется, на элемент, который мы хотим изменить, ссылается индекс элемента. Следовательно, этот метод принимает индекс и обновленный элемент, который необходимо вставить по этому индексу.



Базовые операции и доступные методы

Операция 3: Удаление элементов

Чтобы удалить элемент из ArrayList, мы можем использовать метод `remove()`. Этот метод перегружен для выполнения нескольких операций на основе разных параметров.

- `remove(Object)`: этот метод используется для простого удаления объекта из списка ArrayList. Если таких объектов несколько, **первое вхождение объекта удаляется**.
- `remove(int index)`: поскольку ArrayList индексируется, этот метод принимает целочисленное значение, которое просто удаляет элемент, присутствующий в этом конкретном индексе в ArrayList. После удаления элемента все элементы перемещаются влево, чтобы заполнить пространство, а индексы объектов обновляются.

Операция 4: Итерация ArrayList

Существует несколько способов перебора ArrayList. Наиболее известные способы — это использование базового цикла `for` в сочетании с методом `get()` для получения элемента по определенному индексу.



И еще несколько методов:

- `addAll(Collection C)` - Этот метод используется для добавления всех элементов из определенной коллекции в конец упомянутого списка в таком порядке, чтобы значения возвращались итератором указанной коллекции.
- `get(int index)` - Возвращает элемент в указанной позиции в этом списке.
- `clear()` - Этот метод используется для удаления всех элементов из любого списка.
- `contains(Object o)` - Возвращает `true`, если этот список содержит указанный элемент.
- `isEmpty()` - Возвращает `true`, если этот список не содержит элементов.
- `size()` - Возвращает количество элементов в этом списке.
- `indexOf(Object O)` - Возвращается либо индекс первого вхождения определенного элемента, либо `-1`, если элемента нет в списке.
- `trimToSize()` - Этот метод используется для сокращения емкости экземпляра `ArrayList` до текущего размера списка.



Мы поговорили о:

- Амортизированном анализе, его назначении и как применить его на практике
- Динамическом массиве, его базовых операциях и способности расширяться
- ArrayList как структура данных в Java, часто используемые методы

Полезные ссылки:

<https://proglib.io/p/awesome-algorithms>

https://en.wikipedia.org/wiki/Potential_method

[https://en.wikipedia.org/wiki/Accounting_method_\(computer_science\)](https://en.wikipedia.org/wiki/Accounting_method_(computer_science))