**Что такое структуры данных, для чего они и какие есть в Java?**

Под структурами данных подразумевается хранение данных и их организация таким образом, чтобы решать поставленную задачу наиболее эффективным способом. В Java есть следующие структуры данных:

* Массив
* Список (Динамический массив)
* Стек
* Очередь
* Связный список
* HashTable и HashMap
* Дерево

**Массив**

Массив - это нумерованный набор переменных одного типа.

Объявляется следующем образом:

int[] arr = new int[10];

* Все массивы в Java одномерные. В случае с многомерными массивами каждый элемент содержит только ссылку на вложенный массив
* Можно создать нулевого размера, может быть полезно если нужно вернуть пустой массив из какого-либо метода
* Оператор new используется для создания ссылочного типа данных. Ссылка хранится на стеке, а объект в куче. Если на объект нет ссылок, то он будет удалён автоматически. Удаление объекта может быть осуществлено с задержкой

**Список (Динамический массив)**

Идея списка или же динамического массива заключается в автоматическом расширении емкости.

Объявляется следующем образом:

ArrayList<Integer> arr = new ArrayList<Integer>();

* Примитивный тип данных передать не можем, поэтому передаем класс обертку. О классах обертках, можно прочитать [здесь](https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html#:~:text=Autoboxing%20is%20the%20automatic%20conversion,way%2C%20this%20is%20called%20unboxing.). При желании можно написать универсальную реализацию ArrayList, сделать его массивом Object и тогда можно будет хранить еще и примитивы благодаря автоупаковке
* Если не указать в конструктор начальную емкость, то будет создан пустой список с емкостью в 10 элементов
* В случае, когда зарезервированной емкости не хватает, при достижении максимального количества элементов будет создан новый массив с емкостью: новая\_емкость = (старая\_емкость \* 3) / 2 + 1. Существующие элементы списка будут скопированы в новый массив
* Чтобы не тратить память напрасно, при удалении элементов следует вызывать метод trimToSize()

Плюсы и минусы

+ доступ к элементам по индексу за O(1), к элементам по значению за O(n);  
+ можно хранить любые значения и даже null.

- вставка или удаление элемента в середину списка вызывает перезапись всех элементов, работает за O(n);  
- поиск за O(n);  
- не синхронизирован.

**Стек**

Очередь работает по принципу LIFO. В Java наследуется от Vector<E>, реализует следующие интерфейсы: Iterable<E>, Collection<E>, List<E>, RandomAccess, Serializable, Cloneable.

Объявляется следующем образом:

Stack<Integer> arr = new Stack<Integer>();

Основные методы

* push() - добавляет в конец очереди;
* peek() - возвращает последний элемент и *не удаляет* его;
* pop() - *удаляет* последний элемент и возвращает его;
* empty() - вернет true - если очередь пуста и false - в противном случае;
* search() - возвращает *номер позиции* с конца очереди.

**Очередь**

Интерфейс Queue<E> описывает одностороннюю очередь, а Deque<E> - двухстороннюю. Прежде чем перейти к объявлению в Java, стоит отметить иерархию наследования. Иерархия следующая:

* Iterable<T> => Collection<E> => Queue<E> => Deque<E>

Интерфейсы Queue<E> и Deque<E> реализуют следующие классы:

* ArrayDeque<E> - двухсторонняя очередь
* LinkedList<E> - связный список
* PriorityQueue<E> - очередь с приоритетами

Объявляется следующем образом:

Queue<Integer> arr = new ArrayDeque<Integer>();

Deque<Integer> arr1 = new ArrayDeque<Integer>();

PriorityQueue<Integer> arr2 = new PriorityQueue<Integer>();

// Очередь на LinkedList'е

Queue<Integer> arr = new LinkedList<Integer>();

Deque<Integer> arr = new LinkedList<Integer>();

Разница между реализацией на листе и деку

* ArrayDeque реализует дек на массиве, поэтому он эффективнее по памяти и работает быстрее, чем LinkedList

Пару слов о PriorityQueue.  
Этот класс реализует следующие интерфейсы: Iterable<E>, Collection<E>, Queue<E>, Serializable. У этого класса есть свои особенности:

* Из очереди первым возвращается элемент с наибольшим приоритетом
* Значение null добавить нельзя

**Связный список**

LinkedList<E> реализует связный список, элементы которого хранят ссылки на предыдущий и следующий элементы.

Класс реализует следующие интерфейсы:  
Iterable<E>, Collection<E>, List<E>, Queue<E>, Deque<E>, Serializable, Cloneable.

Объявляется следующем образом:

LinkedList<Integer> arr = new LinkedList<Integer>();

Сравнение ArrayList и LinkedList по сложности выполнения операций



LinkedList занимает гораздо больше памяти, чем ArrayList. Использовать нужно в определенных случаях, чаще всего когда речь идет о двусвязном списке. Также стоит отметить, что элементы у ArrayList в памяти хранятся линейно, поэтому доступ по индексу происходит за O(1)

**HashTable и HashMap**

HashTable считается устаревшей, поэтому приведена лишь разница между мапой и таблицей. HashMap используется для хранения пары «ключ-значение». В качестве примера использования хэш-мапы можно привести пациента больницы, у которого есть Ф.И.О.и номер медицинского полиса.

* Если конструктору не передать никаких значений, то будет создан пустой словарь с емкостью в 16 элементов и коэффициентом заполнения 0.75
* Если коэффициент заполнения достигает максимума, то число bucket'ов увеличивается в два раза

Класс HashMap<K, V> реализует следующие интерфейсы:  
Map<K, V>, Serializable, Cloneable.

Объявляется следующем образом:

HashMap<String, Integer> map = new HashMap<String, Integer>();

Разница между HashTable и HashMap

* Хэш-Таблица не может хранить null, в отличии от Хэш-Мапы
* В Хэш-Таблице все методы синхронизированы, что сказывается на скорости работы
* Хэш-Таблица не рекомендуется к использования, так как считается устаревшей, Хэш-Мапа предпочтительнееP.S. Если требуется выбрать структуру, которая справится с параллельными вычислениями, то есть ConcurrentHashMap

**Дерево**

Стоит заметить, что готовой реализации бинарного дерева в Java нет, но есть TreeMap<K, V> и TreeSet<E>, которые описывают словари, в котором ключи хранятся в отсортированном порядке. TreeSet инкапсулирует в себе TreeMap, который в свою очередь использует сбалансированное бинарное красно-черное дерево для хранения элементов.

Класс TreeSet<E> реализует следующие интерфейсы:  
Itearble<E>, Collection<E>, Set<E>, SortedSet<E>, NavigatbleSet<E>, Serializable, Cloneable.

Класс TreeMap<K, V> реализует следующие интерфейсы:  
Map<K,V>, SortedMap<K, V>, NavigatbleMap<K, V>, Serializable, Cloneable.

Преимущества иерархической организации данных

* простота восприятия человеком;
* высокое быстродействие при транзакционной обработке.

Недостатки

* медленный доступ к данным нижних уровней;
* склонность к избыточности;
* на больших объёмах данных требуется индексация элементов.

Объявляется следующем образом:

TreeSet<Integer> set = new TreeSet<Integer>();

TreeMap<String, Integer> map = new TreeMap<String, Integer>();

В чем разница между HashSet и TreeSet

* Под капотом у TreeSet лежит красно-черное дерево и упорядочивание элементов происходит именно по принципу красно-черных деревьев. HashSet не поддерживает упорядочивание
* Сложность TreeSet - O(log(n)), HashSet - O(1) (речь идет о методах add(), contains(), remove())