GeekBrains

Лекция 3. Структуры данных. Связный список.

Оглавление

Цель лекции:

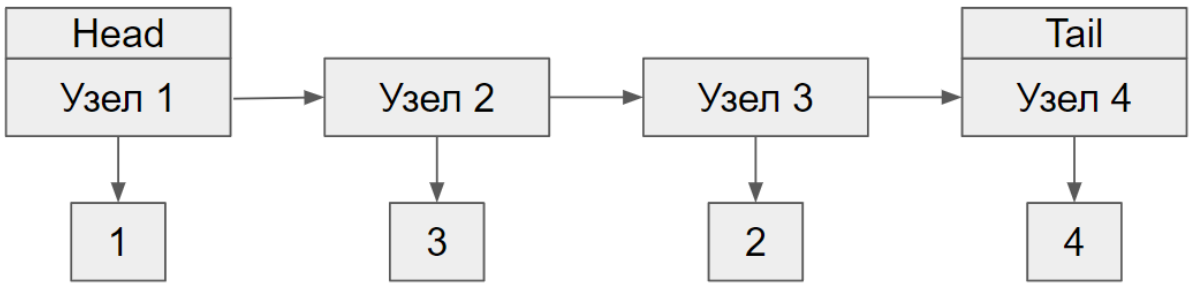
- Познакомиться со структурой данных «связный список»
- Оценить сложность для операций поиска, добавления и удаления элементов. Сравнить с аналогичными операциями массива
- Разобрать различные алгоритмы разворота и сортировки связного списка
- Познакомиться с частными случаями связного списка – стеком и очередью

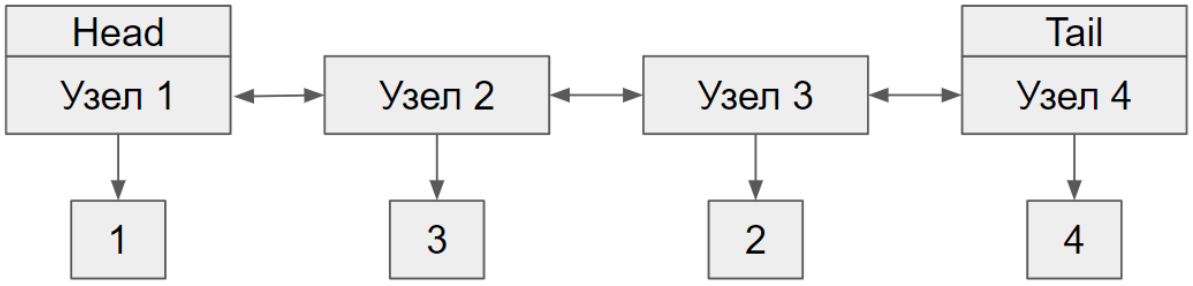
План лекции:

- Связный список, внутренняя структура.
- Поиск элементов в связном списке.
- Операции добавления и удаления элементов из связного списка
- Алгоритм разворота связного списка.
- Сортировка связного списка.
- Очередь и стек.

Связный список

Связный список – базовая структура данных, состоящая из узлов, где каждый узел содержит одну или две ссылки, который ссылаются на следующий или на следующий и предыдущий узел соответственно.





Структурно, связные списки бывают **однонаправленными** – когда каждый узел содержит информацию только о следующем элементе цепочки, и **двунаправленными** – когда каждый узел ссылается на следующий и предыдущий узлы. Первый узел связного списка принято называть **head**, т.к. именно с него начинается обход. Последний элемент принято называть **tail**, благодаря которому можно обойти связный список не только от начала, но и в обратной последовательности (для двунаправленного) или просто добавить новый узел в конец цепочки.

Базовая реализация связного списка будет выглядеть следующим образом:

```java
public class LinkedList {
    private Node head;
    private Node tail;

    class Node {
        private int value;
        private Node nextNode;
        private Node previousNode;
    }
}
```

С точки зрения использования – связные списки ближе всего к массивам и обычно именно с массивами сравнивают данную структуру, выбирая наиболее подходящую для решения задачи. Давайте проведем сравнение базовых операций, общих для обоих из них.

Поиск элементов в связном списке

Первая и самая базовая операция – это операция поиска значений. Для массива существует 2 основных метода – метод перебора (для любых массивов) и бинарный поиск для отсортированных массивов. Метод перебора прекрасно подходит и для связного списка, только перебирать мы будем не индексы, а непосредственно узлы.

```java
    public Node findNode(int value) {
        Node node = head;
        while (node.nextNode != null) {
            node = node.nextNode;
            if (node.value == value) {
                return node;
            }
        }
        return null;
    }
```

Как несложно догадаться, подобный перебор имеет точно такую же сложность, как и у массива – **O(n)**.

А вот если посмотреть в сторону бинарного поиска, который давал значительно большую производительность – **O(log n)**, то все становится не так хорошо. Большая часть алгоритмов, использующихся с массивом, завязана на операции с индексом элементов и бинарный поиск не исключение. Вспомним, что операция обращения по индексу для элементов массива имеет сложность **O(1)**, т.к. система всегда точно знает, где именно в памяти какой элемент располагается. Это базовое свойство массива, отраженное в способе хранения данных в памяти компьютера. Но если мы посмотрим на наш связный список, то увидим, что у экземпляров **Node** нет никакого упоминания индекса, т.е. абсолютного местоположения в структуре. Есть только относительные маркеры – ссылки на предыдущий и следующий элементы. Зная это, мы также можем найти элемент нужного индекса. Базовый алгоритм поиска будет выглядеть следующим образом:

```java
    public Node findNode(int index) {
        Node node = head;
        for (int i = 0; i < index; i++) {
            node = node.nextNode;
            if (node == null){
                return null;
            }

        }
        return node;
    }
```

Мы берем первую ноду в списке и начинаем по очереди перебирать значения, пока не найдем требуемый узел, либо список не закончится. Как несложно догадаться, подобный поиск будет иметь сложность **O(n)**. А это значит, что все плюсы бинарного поиска, связанные с пропуском элементов, не сработают для связного списка – перемещение индекса влево или вправо после сравнения снова потребует совершить перебор элементов до нужного индекса, тем самым количество операций будет от **n/2** (искомый элемент является серединой списка и первым кандидатом на проверку при алгоритме бинарного поиска) до **n** (элемент первый или последний в списке), что дает сложность поиска все те же **O(n)**, что и в простом переборе.

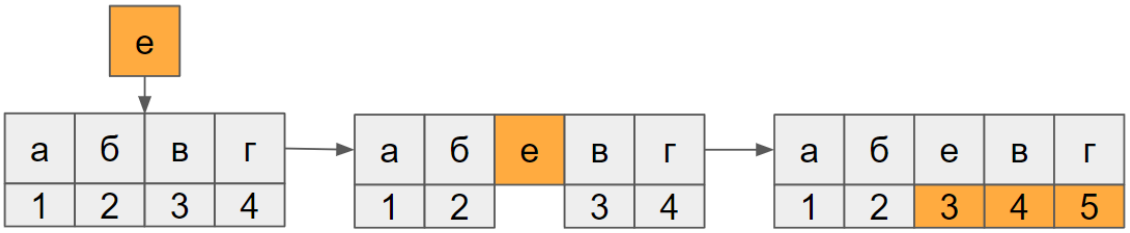
Операции добавления и удаления элементов

Далее следует рассмотреть операции вставки и удаления элементов. Для массива сложность и временные потери во многом будут зависеть от конкретной реализации конкретного языка программирования, но давайте рассмотрим базовый принцип. Так как все элементы массива имеют свой порядковый номер, то добавление элементов в конец массива не требует каких-то дополнительный операций – просто размерность увеличится на единицу, а за новым порядковым номером будет закреплен новый элемент. Операция удаления последнего элемента делается аналогичным образом – она не задевает другие элементы массива и не требует дополнительных операций. В данной задаче операция со связным списком будет выглядеть так же аналогично – у нас есть прямая ссылка на последний элемент и нам необходимо присоединить к нему новый.

```java
    public void addLast(int value){
        Node node = new Node();
        node.value = value;
        tail.nextNode = node;
        node.previousNode = tail;
        tail = node;
    }
```

Как и в случае с массивом – сложность операции **O(1)**.

А вот операции добавления в начало или середину массива будут уже иметь свои накладные расходы, а именно – при добавлении нового элемента необходимо обновить индексы всех элементов, находящихся правее вставляемого значения, т.к. вставляемый элемент занимает индекс уже существующего объекта, а значит для всех правых элементов индекс должен увеличиться на единицу.



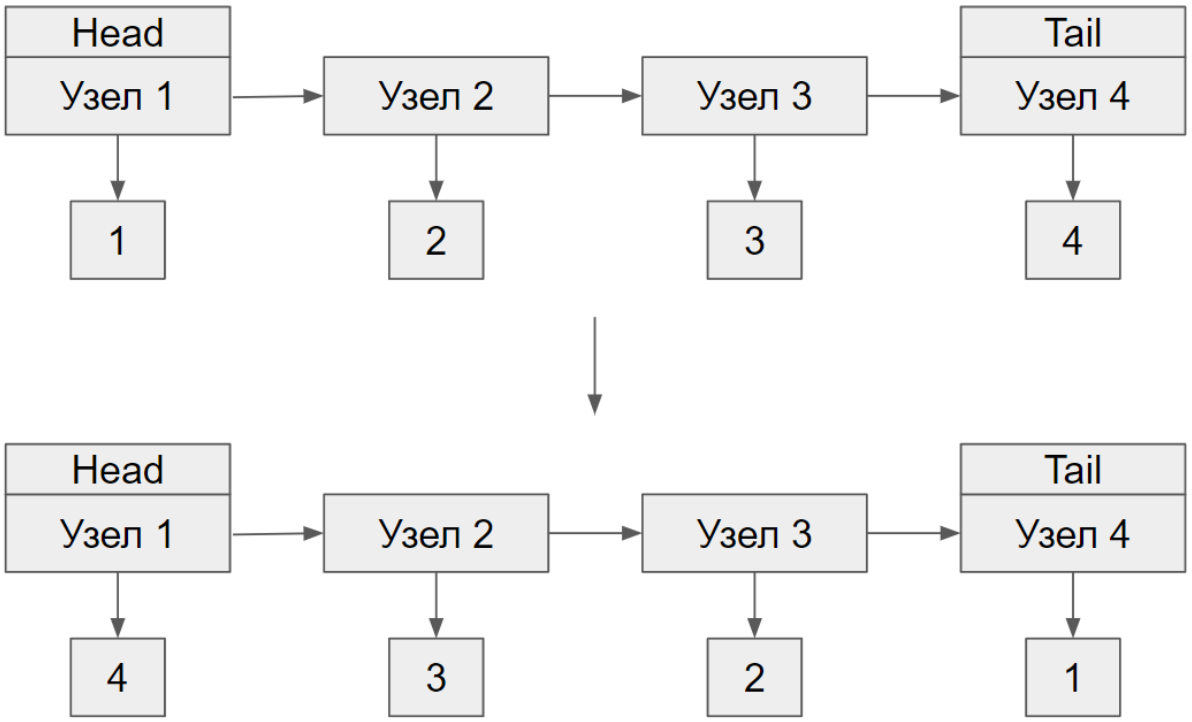
Соответственно, чем ближе к началу вставляется новый элемент, тем более количество элементов нужно будет сдвинуть вправо, а значит сложность становится **O(n)**. Для операции удаления ситуация аналогичной – при удалении элемента, все правые индексы уменьшаются на единицу.
В связном списке же сдвигать индексы после вставки не требуется, т.к. они физически не отслеживаются и не определяются. Любая операция вставки будет похожа на операцию вставки в конец и иметь константную сложность **O(1)**

```
1    public void add(int value, Node previousNode) {
2        Node node = new Node();
3        node.value = value;
4        Node nextNode = previousNode.nextNode;
5        previousNode.nextNode = node;
6        node.previousNode = previousNode;
7        node.nextNode = nextNode;
8        nextNode.previousNode = node;
9    }
```

Остается проблема поиска нужной ноды, которая, как мы говорили ранее, **O(n)**, что суммарно дает сложность **O(1) + O(n) => O(n)** для всех случаев, кроме вставки в начало – мы всегда имеем ссылку на первый элемент и искать его не нужно. Для вставки в начало сложность останется **O(1)**.
Из этого можно сделать вывод, что у связного списка преимущество во вставке в первую половину своей размерности, а у массива наоборот во вторую. Вставка в конец одинаково работает для обеих структур данных. Так же различными оптимизациями алгоритмов мы можем сократить издержки для тех или иных операций для обеих структур. Например, если мы храним размер двунаправленного списка и можем точно определить, что искомый индекс находится во второй половине размерности, мы можем начать обход не с начала, а с конца списка, тем самым нивелировав разницу в сложности для вставки в первую или вторую половину списка.
Также стоит учитывать поведение памяти при работе с массивом, но оно будет отличаться у разных реализаций. Например, в Java массив имеет строго фиксированную длину и, если мы хотим добавить в него элемент, чей индекс не влезает в размерность, нам нужно полностью пересоздать массив, выделив под него новое адресное, а список не имеет таких ограничений и легко дополняется новыми элементами в любой момент времени.

Разворот связного списка

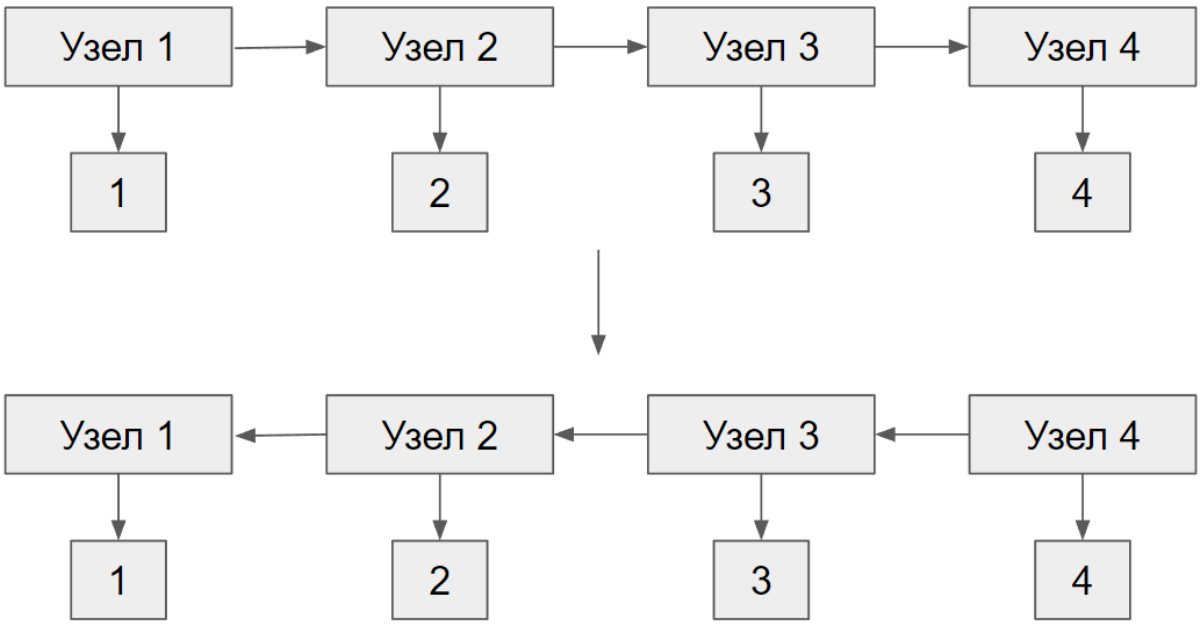
Для связного списка также существуют специфические задачи – например, **разворот**. **Разворотом** называется операция, когда последний элемент становится первым, предпоследний вторым и т.д.



Когда речь идет про алгоритмы разворота, то оценивать сложность с точки зрения количества операций смысла не имеет – все равно тем или иным образом нам придется обратиться к каждому из элементов списка, а значит получить сложность **O(n)**. А вот количество памяти, которое придется затратить будет иметь значение.

Например, самый просто способ получить развернутый список (если мы говорим про двунаправленный список), это создать новый экземпляр и заполнить его, прочитав исходный список с конца. Но создание полностью нового экземпляра списка со всеми значениями потребует **O(n)** памяти, т.к. для каждого узла придется создать копию в новом объекте. Для больших объектов это может стать проблемой. Давайте разберем вариант, при котором расход памяти будет меньше, а именно **O(1)**, что более оптимально.

На самом деле нам не имеет смысл создавать новый список, если нам нужно развернуть существующий. Каждый узел списка всегда хранит в себе ссылки на соседние элементы, а значит их нужно просто поменять местами, что последний элемент стал первым. Проще всего это продемонстрировать на однонаправленном списке.



А значит алгоритм будет представлять из себя обычный перебор, где мы меняем ссылки местами.

```java
public void revert() {
    Node node = head;

    //меняем метсами указатели на head и tail
    Node temp = head;
    head = tail;
    tail = temp;

    //перебираем список, переворачивая указатели
    while (node.nextNode != null) {
        temp = node.nextNode;
        node.nextNode = node.previousNode;
        node.previousNode = temp;
        node = node.previousNode;
    }

}
```

Таким образом нам требуется только 1 дополнительная переменная в памяти для хранения значения во время замены элементов местами, что дает константную сложность **O(1)**.

Сортировка связного списка

Сортировка связного списка имеет те же особенности, что и поиск по связному списку – большая часть алгоритмов основана на работе с индексами и для корректной реализации необходимо учитывать поведение поиска элементов и заменять его на работу с узлами. Например, сортировка пузырьком ничем от реализации для массива не отличается, т.к. всегда проверяются только соседние элементы, и они же меняются местами. Сортировка вставками ведет себя аналогично – процесс обмена элементами происходит по мере продвижения по списку. Сортировка выбором так же легко модифицируются под работу со списками – хранение нужного индекса заменяется на хранение ссылки на нужную ноду.

Единственные существенные отличия – корректная обработка ссылок при обмене и отслеживание ссылок на head и tail.

```java
//Сортировка выбором
public void sort() {
    Node node = head;
    while (node.nextNode != null) {
        Node minPositionNode = node;
        Node innerNode = node.nextNode;
        while (innerNode != null) {
            if (innerNode.value < minPositionNode.value) {
                minPositionNode = innerNode;
            }
            innerNode = innerNode.nextNode;
        }

        if (minPositionNode != node) {
            swap(node, minPositionNode);

            //обновляем ссылки на head и tail если необходимо
            if (minPositionNode.previousNode == null){
                head = minPositionNode;
            }
            if (node.nextNode == null){
                tail = node;
            }

            //сдвигаем
            node = minPositionNode.nextNode;
        } else {
            node = node.nextNode;
        }
    }
}

private void swap(Node node1, Node node2) {
    //меняем объекты местами, меняя ссылки на соседние элементы
    Node temp = node1.previousNode;
    node1.previousNode = node2.previousNode;
    node2.previousNode = temp;
    temp = node1.nextNode;
    node1.nextNode = node2.nextNode;
    node2.nextNode = temp;

    //корректируем ссылки соседних элементов на корректные после обмена
    if (node2.previousNode != null) {
        node2.previousNode.nextNode = node2;
    }
    if (node2.nextNode != null) {
        node2.nextNode.previousNode = node2;
    }
    if (node1.previousNode != null) {
        node1.previousNode.nextNode = node1;
    }
    if (node1.nextNode != null) {
        node1.nextNode.previousNode = node1;
    }
}
```

Несмотря на выросший объем кода, логика сортировки не изменилась, равно как и сложность – **O(n²)**. Аналогично процедура пройдет и для алгоритма быстрой сортировки.

Стек

Стеком называют структуру данных, реализующую функцию работы с элементами по принципу LIFO – Last in - First out – последний пришел, первый ушел. Проще всего сравнить стек со стопкой книг – книга, положенная последней, окажется сверху и именно ее возьмут в первую очередь. Т.е. базово стек должен поддерживать 2 метода – добавление и получение элемента. С программной точки зрения подобный механизм очень просто организовать как частный случай очереди – добавление и удаление первого элемента очереди всегда имеет сложность O(1).

```
1     public void push(int value) {
2         Node node = new Node();
3         node.value = value;
4         node.nextNode = head;
5         head = node;
6     }
7
8     public int pull() {
9         if (head == null) {
10            throw new IllegalArgumentException("Stack is empty");
11        }
12        int result = head.value;
13        head = head.nextNode;
14        return result;
15    }
```

Стек используется, например, для отмены действий в приложении – история хранится в виде стека и при отмене последнее добавленное действие достается и отменяется.

Очередь

Очередь очень похожа на стеку, но работает по принципу FIFO – First in – First out – первый зашел, первый ушел. Название этой структуры данных говорящее и примером служит, например, живая очередь в магазине – те, кто занял ее раньше буду раньше обработаны. Соответственно очередь так же должна реализовывать 2 метода, но в отличии от стека, доставать элементы не с начала, а с конца. Что так же легко реализуется в связном списке – доступ к последнему элементу и его удаление имеет константную сложность **O(1)**.

```
1     public void push(int value) {
2         Node node = new Node();
3         node.value = value;
4         node.nextNode = head;
5         head.previousNode = node;
6         head = node;
7     }
8
9     public int pull() {
10        if (tail == null) {
11            throw new IllegalArgumentException("Queue is empty");
12        }
13        int result = tail.value;
14        tail = tail.previousNode;
15        tail.nextNode = null;
16        return result;
17    }
```

Очередь чаще всего используется для хранения списка задач и их постепенного исполнения.

Итого

На этом уроке мы познакомились с такой структурой данных как связный список. Рассмотрели его сильные и слабые стороны в сравнении с массивом. Так же мы познакомились со стеком и очередью, структурами данных которые легко реализуются на без связного списка и позволяют решать узкоспециализированные задачи.