



Урок 4

Связанные списки

Учимся создавать и использовать списки.

[Введение](#)

[Связанный список](#)

[Связи в списках](#)

[Метод insert](#)

[Метод delete](#)

[Пример программы LinkedList](#)

[Поиск и удаление заданных элементов](#)

[Двусторонние списки](#)

[Эффективность связанных списков](#)

[Реализация стека на базе связанного списка](#)

[Реализация очереди на базе связанного списка](#)

[Итераторы](#)

[Домашнее задание](#)

[Дополнительные материалы](#)

[Используемая литература](#)

Введение

Список – это структура данных, которая состоит из связанных узлов. Эта структура данных очень похожа на массив. Массив, как мы знаем, имеет ряд недостатков, например, его размер задается при инициализации и не может быть изменен. Список – это динамическая структура, размер которой может изменяться в любой момент. В этом уроке мы рассмотрим, простой связанный список и двусторонний связанный список, а также реализуем стек и очередь на основе связанного списка.

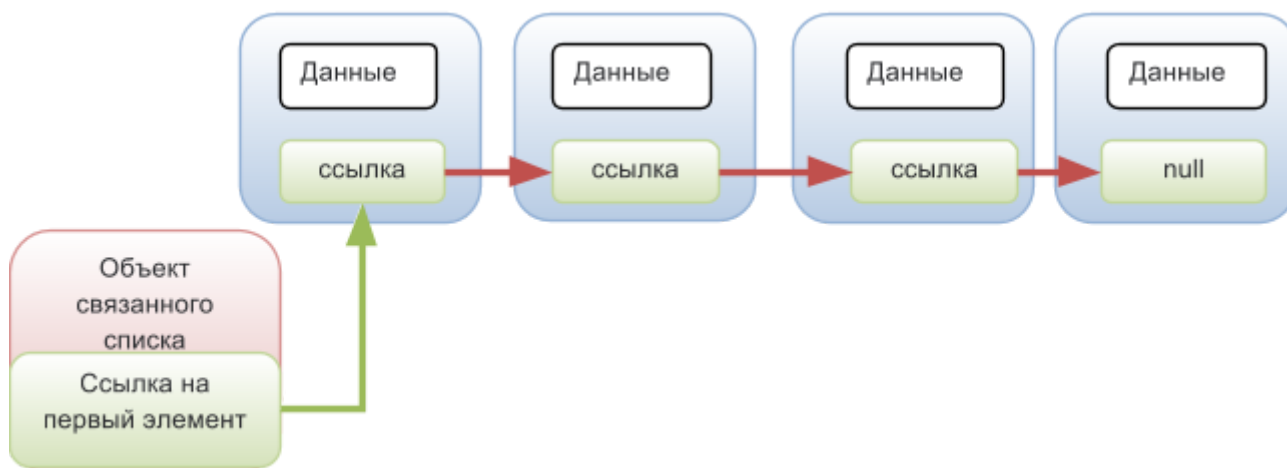
Связанный список

Каждый элемент связанного списка это объект, который хранит данные, а также ссылку на следующий элемент.



В отличие от массивов, порядок элементов связанного списка может не совпадать с порядком расположения элементов данных в памяти компьютера.

Помимо элемента списка создается объект связанного списка, который содержит ссылку на первый элемент.



Рассмотрим класс, который реализует элемент связанного списка. Назовем его Link.

```
class Link {  
    public string name;  
    public int age;  
    public Link next;  
}
```

В этом классе созданы два поля, которые содержат данные и объект типа Link, который содержит ссылку на следующий элемент списка. Отметим, что ссылка является объектом того же класса в которой она описана. Количество и типы полей, в которых хранятся данные, могут быть изменены, их количество может быть разным, а типы данных зависят от описываемого объекта.

Связи в списках

В массиве к каждому элементу можно обратиться через индекс элемента. В связанном списке нет индексов, поэтому конкретный элемент можно найти, отследив его по цепочке элементов от начала списка. Обратиться напрямую к элементу данных связанного списка невозможно. Для поиска используются отношения между элементами списка.

Простые связанные списки поддерживают следующие операции: вставка в начало списка, удаление элемента из начала списка, перебор списка для вывода содержимого.

В начале урока мы рассмотрели поля, которые используются в классе Link. Давайте дополнительно создадим конструктор и метод для вывода на экран элемента списка.

```
class Link{
    private String name;
    private int age;

    private Link next;

    public Link(String name, int age){
        this.name = name;
        this.age = age;
    }

    public void display(){
        System.out.println("Name: "+this.name+", age: "+this.age);
    }
}
```

Заметьте, что мы не заполнили поле Link. По умолчанию, при создании объекта, этому полю будет присвоено значение null. После того, как будет создан следующий элемент списка, в поле Link предыдущего элемента, поместится адрес на него. Еще один нюанс, публичные поля, это сделано для облегчения написания кода, т.к. для полей private придется создавать методы получения и установки, чтобы получить к ним доступ.

Рассмотрим класс, который создает и удаляет элементы нашего списка. Назовем его LinkedList. Для начала создадим в этом классе поле first, которое будет содержать ссылку на первый элемент и конструктор.

```

class LinkedList{
    private Link first;

    public LinkedList(){
        first = null;
    }

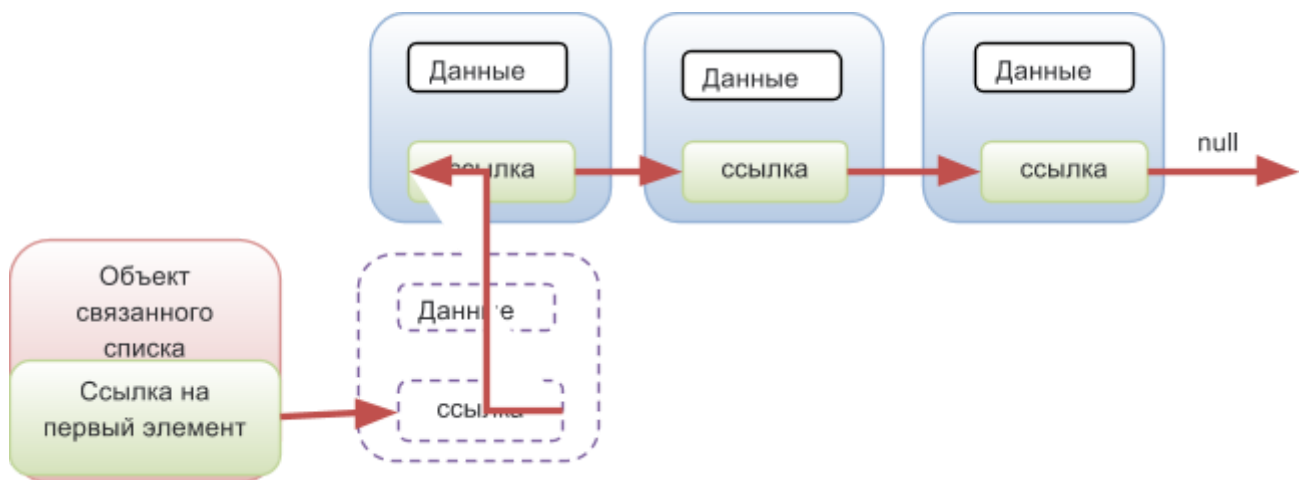
    public boolean isEmpty(){
        return (first == null);
    }
}

```

В конструкторе явно задаем значение поля first. Также создаем метод, который проверяет список на пустоту. Теперь необходимо создать методы для добавления и удаления элементов связанного списка.

Метод insert

Метод insert отвечает за вставку нового элемента в начало связанного списка. В этом методе передаем полю next адрес предыдущего элемента, а в поле first записываем адрес создаваемого элемента.



```

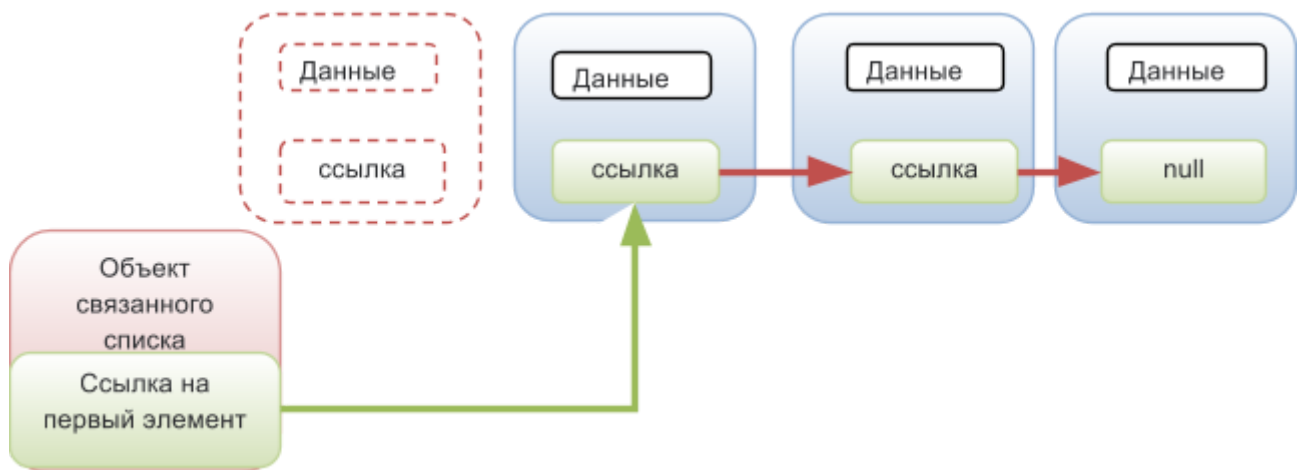
public void insert(String name, int age){
    Link newLink = new Link(name, age);
    newLink.next = first;
    first = newLink;
}

```

Рассмотрим построчно код метода insert. В первой строке создается объект типа Link. Во второй строке полю next присваиваем ссылку на предыдущий элемент списка. В третьей строке изменяем ссылку в поле first, тем самым говоря нашему списку, что добавился новый элемент и вот на него новая ссылка.

Метод delete

Метод delete удаляет элемент из начала связанного списка. Он работает противоположно методу insert. В поле first устанавливается значение следующего за удаляемым элементом, т.е. first.next.



```
public Link delete(){
    Link temp = first;
    first = first.next;
    return temp;
}
```

Посмотрите на код написанный выше. Все, что нужно для удаления элемента – это вторая строка. Остальные строки нужны для возвращения удаляемого объекта. Также данный метод не предусматривает проверки списка на пустоту. Поэтому он должен использоваться совместно с методом isEmpty.

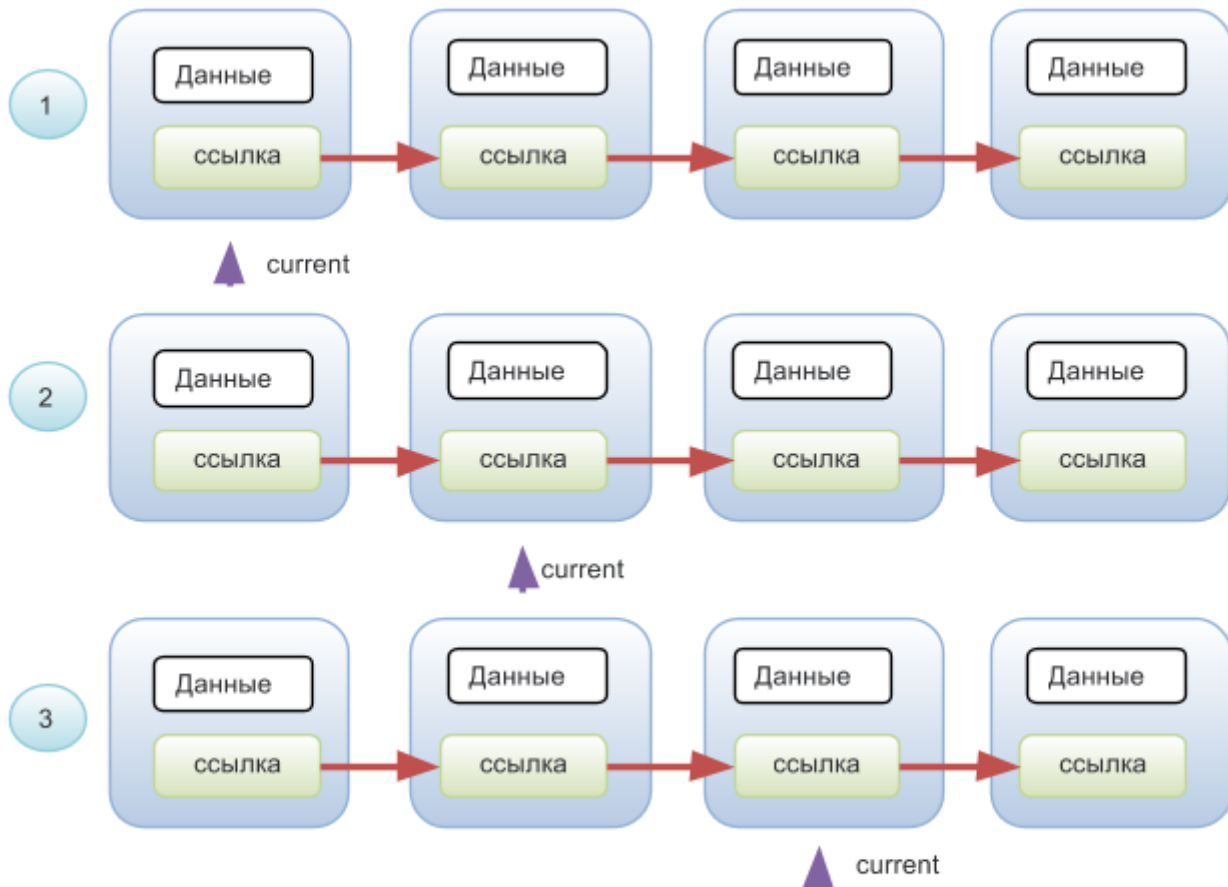
Конечно, встает вопрос, а как же быть с памятью, ведь мы не удалили элемент, а просто стерли ссылку из связанного списка, а объект Link остался. Если бы мы писали код на C++, то необходимо было бы позаботиться об удалении самого элемента, но в Java за нас это сделает сборщик мусора.

Метод display

Реализуем метод, который выводит элементы списка на экран консоли. Назовем этот метод display. Начинать вывод будем со ссылки на объект first и будем перемещаться по списку через поле next, пока next не будет равен null.

```
public void display() {
    Link current = first;
    while(current != null)
    {
        current.display();
        current = current.next;
    }
}
```

На каждой итерации цикла с помощью метода `current.next` мы будем получать следующий элемент.



Пример программы LinkedList

Ниже в листинге приведен полный код программы, реализующий простой связанный список с использованием обобщений.

```
class People {  
    private String name;  
    private int age;  
  
    public People(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public int getAge() {  
        return age;  
    }  
  
    @Override  
    public int hashCode() {  
        int hash = 5;  
    }  
}
```

```

        hash = 53 * hash + Objects.hashCode(this.name);
        hash = 53 * hash + this.age;
        return hash;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        final People other = (People) obj;
        if (this.age != other.age) {
            return false;
        }
        if (!Objects.equals(this.name, other.name)) {
            return false;
        }
        return true;
    }

    @Override
    public String toString() {
        return "Name: "+this.name+", age: "+this.age;
    }
}

class Link<T> {
    private T link;
    private Link<T> next;

    public Link(T link){
        this.link = link;
    }

    public Link<T> getNext() {
        return next;
    }

    public void setNext(Link<T> next) {
        this.next = next;
    }

    public T getValue(){
        return link;
    }
}

class LinkedList<T> {

```

```

private Link<T> first;

public LinkedList(){
    first = null;
}

public boolean isEmpty(){
    return (first == null);
}

public void insert(T link){
    Link<T> l = new Link<>(link);
    l.setNext(first);
    this.first = l;
}

public Link<T> delete(){
    Link<T> temp = first;
    first = first.getNext();
    return temp;
}

public void display(){
    Link<T> current = first;
    while (current != null) {
        System.out.println(current.getValue());
        current = current.getNext();
    }
}

public T find(T searchNode){
    Link<T> findNode = new Link<>(searchNode);
    Link<T> current = first;
    while (current != null) {
        if (current.getValue().equals(findNode.getValue())){
            return findNode.getValue();
        }
        current = current.getNext();
    }
    return null;
}
}

public class GenericListApp {

    public static void main(String[] args) {
        LinkedList<String> list = new LinkedList<>();

        list.insert("Artem");
        list.insert("Roman");

        System.out.println(list.find("Artem"));
    }
}

```



```

        LinkedList<People> peopleList = new LinkedList<>();
        peopleList.insert(new People("Artem", 22));
        peopleList.insert(new People("Roman", 18));

        System.out.println(peopleList.find(new People("Artem", 22)).toString());
    }
}

```

В методе main создается новый список, в который вставляется три элемента с помощью метода insert. Далее список выводится в консоль. Для этого был создан метод display. Потом запускается цикл while, который удаляет элементы, пока список не будет пуст.

Результат работы программы будет следующим:

```

Name: Vova, age: 5
Name: Misha, age: 10
Name: Artem, age: 30
Удаление элементов списка
Удален: Name: Vova, age: 5

Удален: Name: Misha, age: 10

Удален: Name: Artem, age: 30

```

Поиск и удаление заданных элементов

Давайте добавим в нашу программу методы, которые удаляют заданный элемент. Для удаление заданного элемента необходимо произвести две операции. Первая – это поиск, а вторая удаление ссылки из списка.

Рассмотрим сначала принцип работы поиска. Он напоминает метод вывода списка элементов display. Так как у нас в элементе поля для имени и возраста, искать будем по имени.

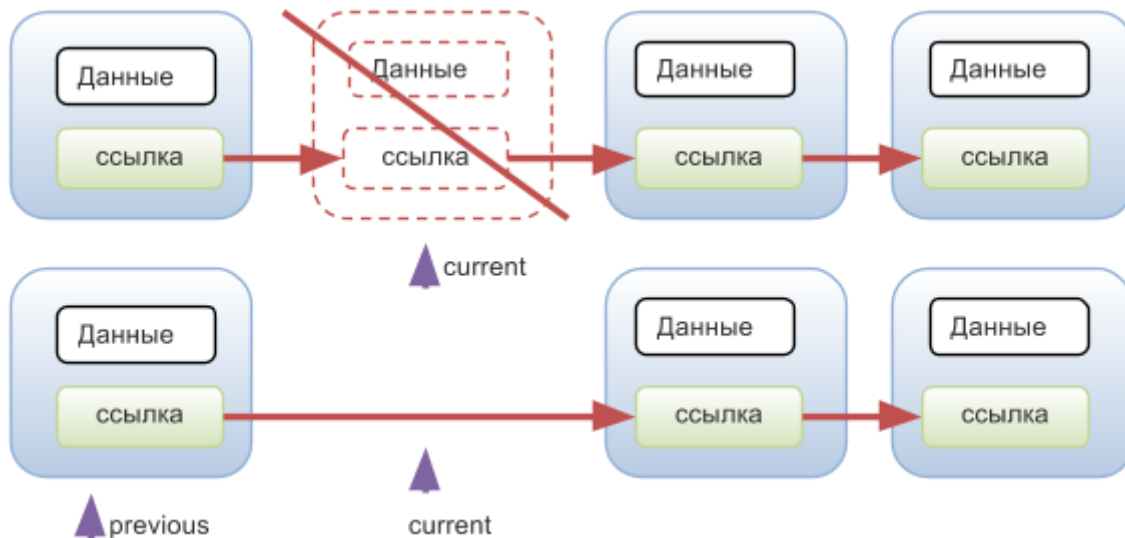
```

public Link find(String name){
    Link current = first;
    while(current.name != name){
        if(current.next == null)
            return null;
        else
            current = current.next;
    }
    return current;
}

```

Начинать поиск будем с первого элемента, поэтому переменной `current` присваиваем значение `first`. Далее в цикле `while`, пока искомое значение не совпадет со значением элемента списка, будем перебирать все его элементы. Если в поле `current.next` будет `null` и искомое значение не будет найдено, то метод вернет `null`, что будет означать, что элемент не был найден. Если искомое значение совпадет с полем `current.next`, тогда цикл прервется и будет возвращена ссылка на искомый элемент.

Что касается метода `delete`, то он использует такой же перебор, как и в методе `find`. Только теперь необходимо сохранить ссылку на предыдущий элемент, для того чтобы можно было связать список.



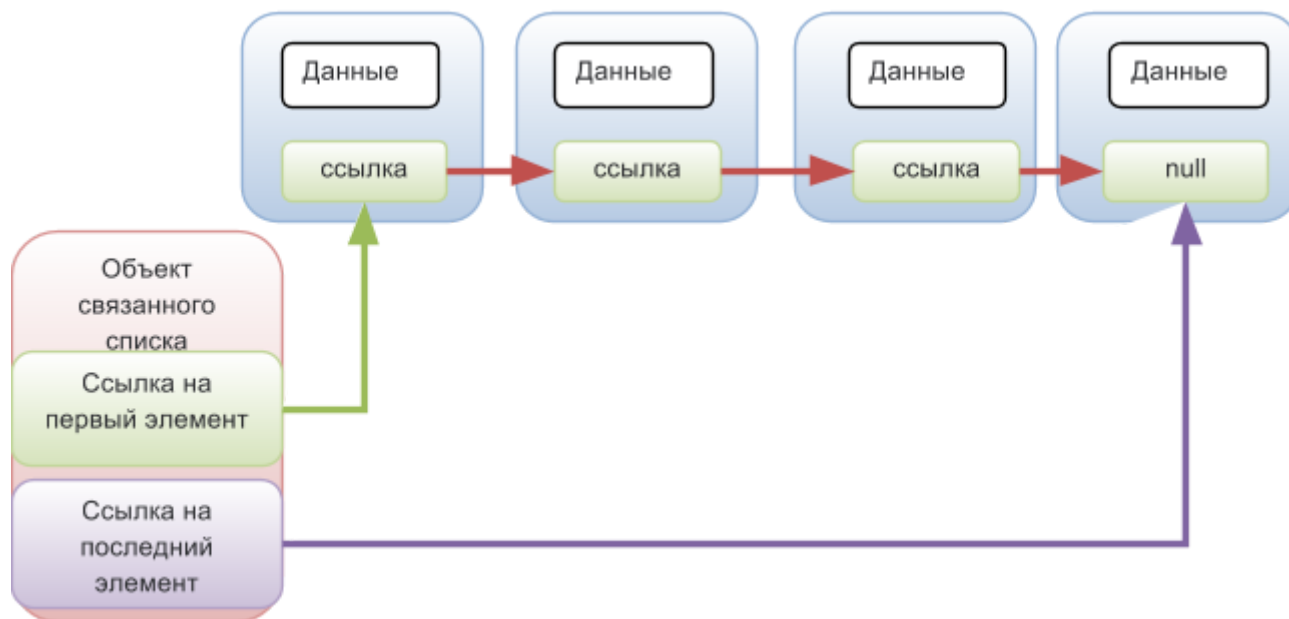
Реализация метода `delete` представлена в листинге ниже. Переменной `current` и `previous` задается значение первого элемента списка. Далее происходит поиск по имени. Пока имена не совпадут или не будет достигнут конец списка, переменной `current` будет присваиваться следующее значение в списке, а переменной `previous` текущее.

Если совпадение будет найдено и это окажется первый элемент, то необходимо заменить его на следующий, иначе присвоить ссылку следующему элементу после удаляемого.

```
public Link delete(String name){
    Link current = first;
    Link previous = first;
    while(current.name != name){
        if(current.next == null)
            return null;
        else{
            previous = current;
            current = current.next;
        }
    }
    if(current == first)
        first = first.next;
    else
        previous.next = current.next;
    return current;
}
```

Двусторонние списки

Двухсторонний список позволяет вставлять элементы не только в начало, но и в конец списка. Это и есть отличие от простого связанного списка. Для этого для класса списка создается дополнительное поле `last`.



Конечно, и в простом связанном списке можно было реализовать метод `insertLast`, но алгоритм был бы неэффективным, т.к. пришлось бы перебирать все элементы списка, перед тем как вставить новый элемент.

Дополним наш код методом `insertLast` и полем для хранения ссылки на последний элемент. Немного изменим методы `insert` и `delete`.

В методе `insert` добавим проверку на пустой список. Если список пустой и это первый элемент, то делаем его и последним.

```
public void insert(String name, int age){
    Link newLink = new Link(name, age);
    if (this.isEmpty())
        last = newLink;
    newLink.next = first;
    first = newLink;
}
```

В методе `delete` проверяем список на пустоту, и если следующий элемент после удаляемого равен `null`, то значение поля `last` делаем `null`.

```

public Link delete(){
    Link temp = first;
    if (first.next == null)
        last = null;
    first = first.next;
    return temp;
}

```

Новый метод insertLast выглядит следующим образом.

```

public void insertLast(String name, int age){
    Link newLink = new Link(name, age);
    if (this.isEmpty()){
        first = newLink;
    } else {
        last.next = newLink;
    }
    last = newLink;
}

```

Полный листинг программы представлен ниже.

```

class Link{
    public String name;
    public int age;

    public Link next;

    public Link(String name, int age){
        this.name = name;
        this.age = age;
    }

    public void display(){
        System.out.println("Name: "+this.name+", age: "+this.age);
    }
}

class LinkedList{
    public Link first;
    public Link last;

    public LinkedList(){
        first = null;
        last = null;
    }

    public boolean isEmpty(){
        return (first == null);
    }

    public void insert(String name, int age){
        Link newLink = new Link(name, age);
    }
}

```

```

        if (this.isEmpty())
            last = newLink;
            newLink.next = first;
            first = newLink;
    }

    public void insertLast(String name, int age){
        Link newLink = new Link(name, age);
        if (this.isEmpty()){
            first = newLink;
        } else {
            last.next = newLink;
        }
        last = newLink;
    }

    public Link delete(){

        Link temp = first;
        if (first.next == null)
            last = null;
            first = first.next;
            return temp;

    }

    public void display(){
        Link current = first;
        while(current != null){
            current.display();
            current = current.next;
        }
    }

    public Link find(String name){
        Link current = first;
        while(current.name != name){
            if(current.next == null)
                return null;
            else
                current = current.next;
        }
        return current;
    }

    public Link delete(String name){
        Link current = first;
        Link previous = first;
        while(current.name != name){
            if(current.next == null)
                return null;
            else{
                previous = current;
                current = current.next;
            }
        }
        if(current == first)
            first = first.next;
        else
            previous.next = current.next;
        return current;
    }
}

```

```

}

public class ListApp {

    public static void main(String[] args) {
        LinkedList list = new LinkedList();
        list.insert("Artem", 30);
        list.insert("Misha", 10);
        list.insert("Vova", 5);
        list.insertLast("Petya", 25);

        list.display();
        System.out.println("Удаление элементов списка");

        list.delete("Vova");
        list.display();
    }
}

```

Эффективность связанных списков

Вставка в начало связанного списка происходит очень быстро и равна $O(1)$. Удаление элемента из начала списка выполняется также за время $O(1)$. А вот поиск и удаление конкретного элемента выполняется за время $O(N)$, т.к. существует вероятность, что элемент будет последним и придется перебрать все элементы списка. Но все равно выполнение этих действий у списка требуют меньше времени, чем у массива, т.к. не требуется сдвигать элементы после удаления.

Реализация стека на базе связанного списка

Мы уже реализовывали стек и знаем, что эта структура данных, в которой элементы, входящие последними, выходят первыми. Эту структуру данных мы реализовали с использованием массива. Попробуем реализовать ее с использованием связанного списка.

Методы push и pop, с помощью которых осуществлялась вставка и удаление элементов из стека, фактически, выполняли операции массива, такие как `arr[++top] = data;` и `data = arr[top--];`.

Попробуем добавить дополнительный уровень абстракции и, не изменяя методы связанного списка, создать новый класс для реализации стека.

```

class StackList{
    private LinkedList list;
    public StackList(){
        list = new LinkedList();
    }

    public void push(String name, int age){
        list.insert(name, age);
    }

    public String pop(){
        return list.delete().name;
    }

    public boolean isEmpty(){
        return list.isEmpty();
    }

    public void display(){
        list.display();
    }
}

```

Для реализации стека созданы методы push и pop. В методе push выполняется метод insert класса LinkedList, а в методе pop выполняется метод delete. Также созданы обертки для методов isEmpty и display.

Полный листинг реализации стека на базе связанного списка представлен в листинге ниже.

```

class Link{
    public String name;
    public int age;

    public Link next;

    public Link(String name, int age){
        this.name = name;
        this.age = age;
    }

    public void display(){
        System.out.println("Name: "+this.name+", age: "+this.age);
    }
}

class LinkedList{
    public Link first;

    public LinkedList(){
        first = null;
    }

    public boolean isEmpty(){
        return (first == null);
    }
}

```

```

    }

    public void insert(String name, int age){
        Link newLink = new Link(name, age);
        newLink.next = first;
        first = newLink;
    }

    public Link delete(){

        Link temp = first;
        first = first.next;
        return temp;

    }

    public void display(){
        Link current = first;
        while(current != null){
            current.display();
            current = current.next;
        }
    }
}

class StackList{
    private LinkedList list;
    public StackList(){
        list = new LinkedList();
    }

    public void push(String name, int age){
        list.insert(name, age);
    }

    public String pop(){
        return list.delete().name;
    }

    public boolean isEmpty(){
        return list.isEmpty();
    }

    public void display(){
        list.display();
    }
}

public class LinkStackApp {

    public static void main(String[] args) {
        StackList sl = new StackList();
        sl.push("Artem", 30);
        sl.push("Viktor", 20);
        sl.push("Sergey", 10);
        sl.display();
        while (!sl.isEmpty()) {

```



```
        System.out.println("Элемент "+ sl.pop()+" удален из стека");
    }
}
```

В методе main создается объект типа StackList. Выполняется вставка трех элементов в стек, после чего происходит их удаление в цикле. Результат выполнения программы будет следующим.

Name: Sergey, age: 10

Name: Viktor, age: 20

Name: Artem, age: 30

Элемент Sergey удален из стека

Элемент Viktor удален из стека

Элемент Artem удален из стека

Реализация очереди на базе связанного списка

Реализуем очередь на базе связанного списка. Для ее реализации будем использовать метод `insertLast`, который вставляет элемент в конец очереди и метод `delete`, который удаляет элемент из начала очереди.

```
class Queue{
    private LinkedList queue;

    public Queue(){
        queue = new LinkedList();
    }

    public boolean isEmpty(){
        return queue.isEmpty();
    }

    public void insert(String name, int age){
        queue.insert(name, age);
    }

    public String delete(){
        return queue.delete();
    }

    public void display(){
        queue.display();
    }
}
```

Полный код программы, реализующий очередь на базе связанного списка, представлен в листинге ниже.

```
class Link{
    public String name;
    public int age;

    public Link next;

    public Link(String name, int age){
        this.name = name;
        this.age = age;
    }

    public void display(){
        System.out.println("Name: "+this.name+", age: "+this.age);
    }
}

class LinkedList{
    public Link first;
```

```

public Link last;

public LinkedList(){
    first = null;
    last = null;
}

public boolean isEmpty(){
    return (first == null);
}

public void insert(String name, int age){
    Link newLink = new Link(name, age);
    if (this.isEmpty())
        first = newLink;
    else
        last.next = newLink;
    last = newLink;
}

public String delete(){
    Link temp = first;
    if (first.next == null)
        last = null;
    first = first.next;
    return temp.name;
}

}

public void display(){
    Link current = first;
    while(current != null){
        current.display();
        current = current.next;
    }
}
}

class Queue{
    private LinkedList queue;

    public Queue(){
        queue = new LinkedList();
    }

    public boolean isEmpty(){
        return queue.isEmpty();
    }

    public void insert(String name, int age){
        queue.insert(name, age);
    }

    public String delete(){
        return queue.delete();
    }

    public void display(){
        queue.display();
    }
}

```

```

}

public class LinkQueueApp {

    public static void main(String[] args) {
        Queue q = new Queue();
        q.insert("Artem", 30);
        q.insert("Viktor", 20);
        q.insert("Sergey", 10);
        q.display();
        while (!q.isEmpty()) {
            System.out.println("Элемент "+ q.delete()+" удален из стека");
        }
    }
}

```

Подводя итог, хотелось бы уточнить, когда стоит использовать для реализации стека и очереди связанный список, а когда массив. Наверно, это зависит, от того, насколько точно можно предсказать размер стека или очереди. Если размер не определен, то лучше использовать связанный список.

Итераторы

В завершение данного урока хотелось немного сказать про итераторы. Итераторы часто используются, когда мы перебираем элементы списка. В уроке были рассмотрены методы поиска и удаления элемента, но мы никак не влияем на них. Например, мы хотим сравнить все элементы списка с одним из них и удалить, которые не удовлетворяют определенному критерию (у которых поле возраст меньше выбранного элемента). Для этого и были созданы итераторы, которые указывают на конкретный элемент списка.

Без методов класс итератора будет выглядеть следующим образом.

```

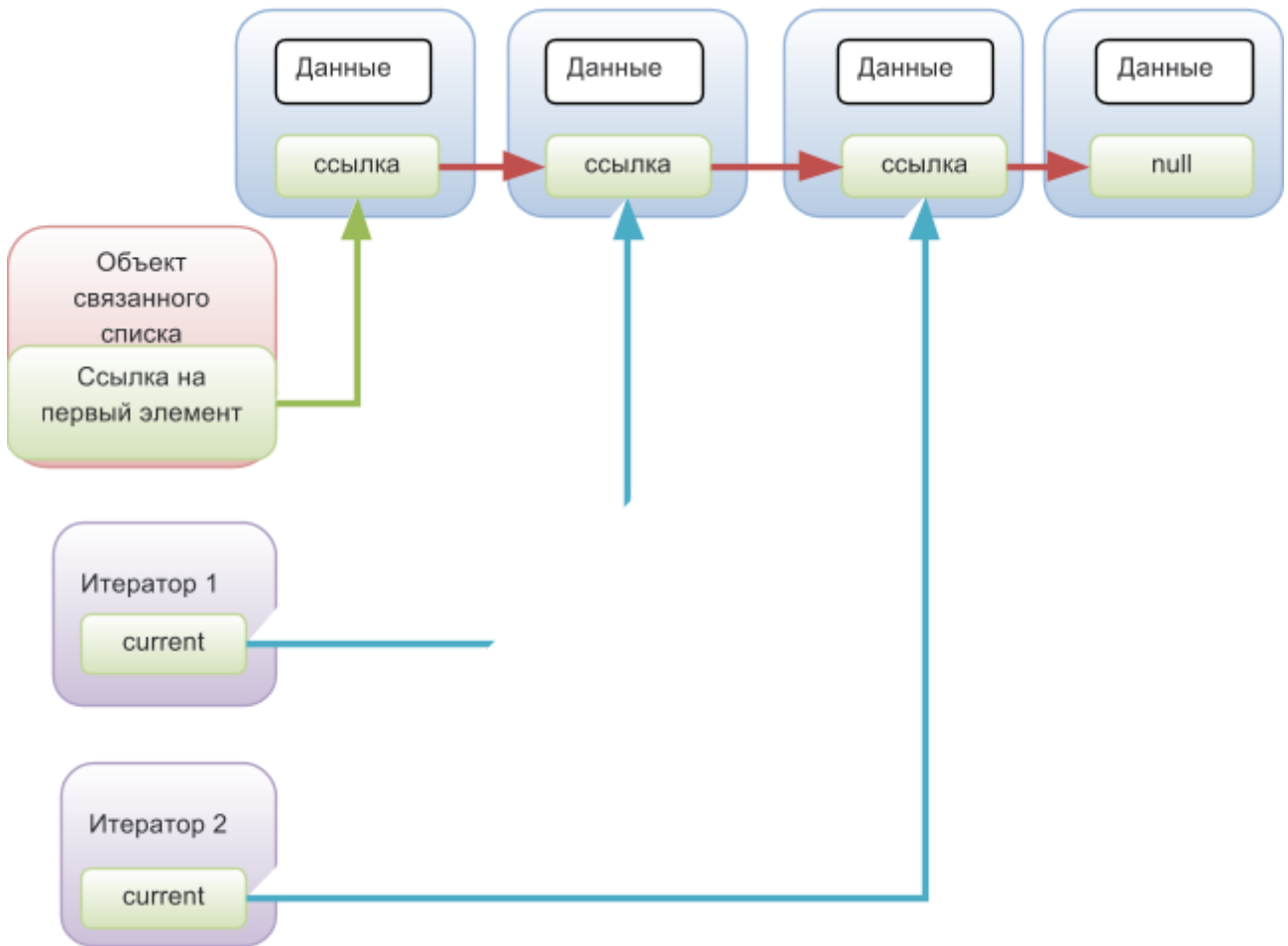
class LinkIterator{
    private Link current;
}

```

Как правило, итераторы содержат следующие методы:

- ✓ reset() – перемещение в начало списка;
- ✓ nextLink() – перемещение итератора к следующему элементу;
- ✓ getCurrent() – получение элемента, на который указывает итератор;
- ✓ atEnd() – возвращает true, если итератор находится в конце списка;
- ✓ insertAfter() – вставка элемента после итератора;

- ✓ insertBefore() – вставка элемента до итератора;
- ✓ deleteCurrent() – удаление элемента в текущей позиции итератора.



Реализуем класс LinkIterator и добавим код в класс с реализацией списка LinkedList. Первое, что мы сделаем – изменим модификатор доступа у поля first на private и создадим для него методы получения и установки.

```
public Link getFirst() {
    return first;
}

public void setFirst(Link first) {
    this.first = first;
}
```

Добавим метод getIterator, который будет создавать новый итератор. В качестве аргумента для итератора передаем текущий список.

```
public LinkIterator getIterator(LinkedList list){
    return new LinkIterator(this);
}
```

Реализованный класс LinkIterator и программный код связанного списка представлен ниже.

```
class Link{
    public String name;
    public int age;

    public Link next;

    public Link(String name, int age){
        this.name = name;
        this.age = age;
    }

    public void display(){
        System.out.println("Name: "+this.name+", age: "+this.age);
    }
}

class LinkedList{
    private Link first;

    public LinkedList(){
        first = null;
    }

    public Link getFirst() {
        return first;
    }

    public void setFirst(Link first) {
        this.first = first;
    }

    public LinkIterator getIterator(){
        return new LinkIterator(this);
    }

    public boolean isEmpty(){
        return (first == null);
    }

    public void display(){
        Link current = first;
        while(current != null){
            current.display();
            current = current.next;
        }
    }
}

class LinkIterator{
    private Link current;
    private Link previous;
    private LinkedList list;

    public LinkIterator(LinkedList list){
        this.list = list;
        this.reset();
    }
}
```

```

public void reset(){
    current = list.getFirst();
    previous = null;
}

public boolean atEnd(){
    return (current.next == null);
}

public void nextLink(){
    previous = current;
    current = current.next;
}

public Link getCurrent(){
    return current;
}

public void insertAfter(String name, int age){
    Link newLink = new Link(name, age);
    if (list.isEmpty()){
        list.setFirst(newLink);
        current = newLink;
    } else {
        newLink.next = current.next;
        current.next = newLink;
        nextLink();
    }
}

public void insertBefore(String name, int age){
    Link newLink = new Link(name, age);
    if(previous == null){
        newLink.next = list.getFirst();
        list.setFirst(newLink);
        reset();
    }
    else{
        newLink.next = previous.next;
        previous.next = newLink;
        current = newLink;
    }
}

public String deleteCurrent(){
    String name = current.name;
    if (previous == null){
        list.setFirst(current.next);
        reset();
    } else {
        previous.next = current.next;
        if (atEnd()){
            reset();
        } else {
            current = current.next;
        }
    }
}

return name;

```

```
}  
}  
public class LinkIteratorApp {  
  
    public static void main(String[] args) {  
        LinkedList list = new LinkedList();  
  
        LinkIterator itr = new LinkIterator(list);  
  
        itr.insertAfter("Artem", 20);  
        itr.insertBefore("Sergey", 10);  
  
        list.display();  
    }  
}
```

Домашнее задание

1. Реализовать все классы, рассмотренные в данном уроке.
2. В методе main LinkIteratorApp проверить все методы итератора.

Дополнительные материалы

1. [Связанный список](#)
2. [Еще немного о списках](#)

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Лафорте Р. Структуры данных и алгоритмы в Java. Классика Computers Science. 2-е изд.—СПб.: Питер, 2013. — 46-119 сс.