

Сортированные списки

ЛЕКЦИЯ 4-2

Сортированные списки



В связанных списках, встречавшихся нам до настоящего момента, данные хранились в произвольном порядке. Однако в некоторых приложениях бывает удобно хранить данные списка в упорядоченном виде. Список, обладающий этим свойством, называется **сортированным списком.**

Сортированные списки

В общем случае, сортированный список может использоваться почти во всех ситуациях, в которых может использоваться сортированный массив. Преимущества сортированного списка перед сортированным массивом — быстрая вставка (не требующая перемещения элементов) и возможность динамического расширения списка (тогда как массив ограничивается фиксированным размером). С другой стороны, сортированный список реализуется несколько сложнее, чем сортированный массив.

Вставка элемента

```
public void insert(long key){ // Вставка в порядке сортировки
    Link newLink = new Link(key); // Создание нового элемента
    Link previous = null; // От начала списка
    Link current = first;
    // До конца списка
    while(current != null && key > current.dData) { // или если key > current,
        previous = current;
        current = current.next; // Перейти к следующему элементу
    }
    if(previous==null) // В начале списка
        first = newLink; // first --> newLink
    else // Не в начале
        previous.next = newLink; // старое значение prev --> newLink
    newLink.next = current; // newLink --> старое значение current
}
```

Эффективность сортированных списков

Вставка и удаление произвольных элементов в сортированных связанных списках требуют $O(N)$ сравнений (в среднем $N/2$), потому что позицию для выполнения операции приходится искать перебором списка. С другой стороны, поиск или удаление наименьшего значения выполняется за время $O(1)$, потому что оно всегда находится в начале списка. Если приложение часто обращается к наименьшему элементу, а скорость вставки не критична, то сортированный связанный список будет достаточно эффективным. Например, приоритетная очередь может быть реализована на базе сортированного связанного списка.



Сортировка методом вставки

Сортированный список может использоваться в качестве достаточно эффективного механизма сортировки. Допустим, у вас имеется массив с несортированными данными. Если последовательно читать элементы из массива и вставлять их в сортированный список, они будут автоматически отсортированы. Остается извлечь их из списка и поместить обратно в массив. Подобный вид сортировки заметно превосходит по эффективности обычные виды сортировки методом вставки в массиве, потому что она требует меньшего количества операций копирования. Она также выполняется за время $O(N^2)$, потому что при вставке в сортированный список каждый элемент приходится сравнивать в среднем с половиной элементов, уже находящихся в списке; для N вставляемых элементов количество сравнений составит $\frac{N^2}{4}$. Однако каждый элемент в этом случае копируется только два раза: из массива в список и из списка в массив. $N \times 2$ операций копирования выгодно отличается от сортировки методом вставки в массив, требующей в среднем N^2 таких операций.



Создать класс Person, сделать два поля у класса name и lastName.

Сделать данный класс узлом связанного списка.

Создать массив из объектов класса Person.

Реализовать сортировку массива при помощи сортированного связанного списка.

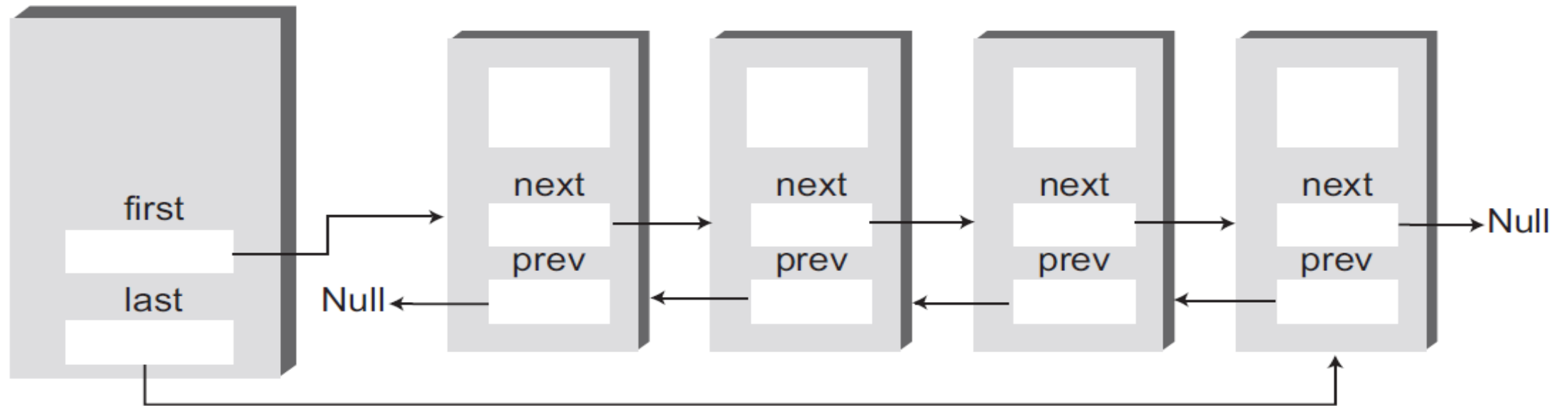


Двусвязные списки

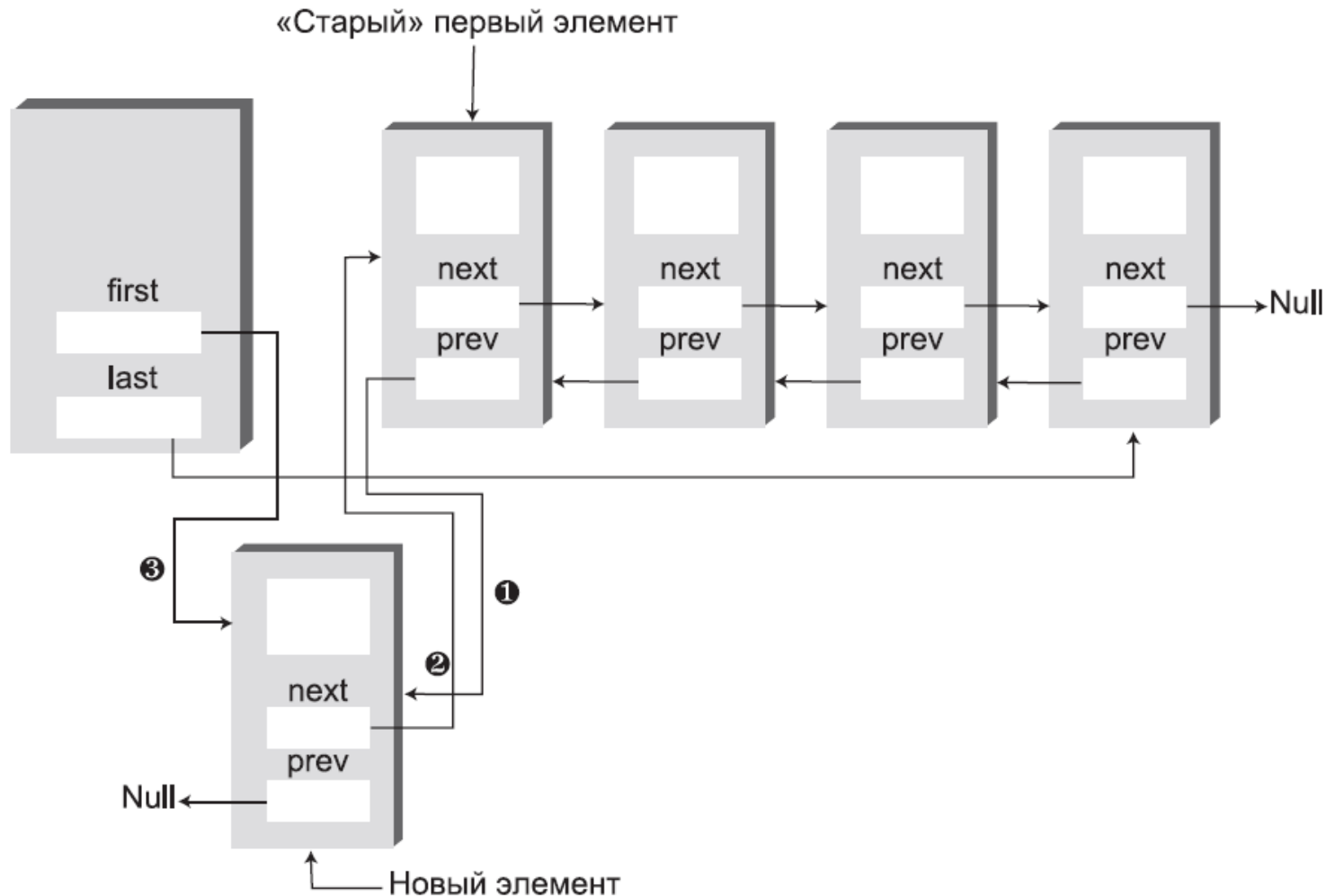


Двусвязный список - это структура данных, которая состоит из узлов, которые хранят полезные данные, указатели на предыдущий узел и следующий узел.

Двусвязные списки



ВСТАВКА В НАЧАЛО



Для того чтобы добавить новый элемент в начало необходимо записать ссылку на новый элемент в поле `previous` «старого» первого элемента, а ссылку на «старый» первый элемент — в поле `next` нового элемента.

```
public void insertFirst(int value) {
    DoubleLink newLink = new DoubleLink(value);

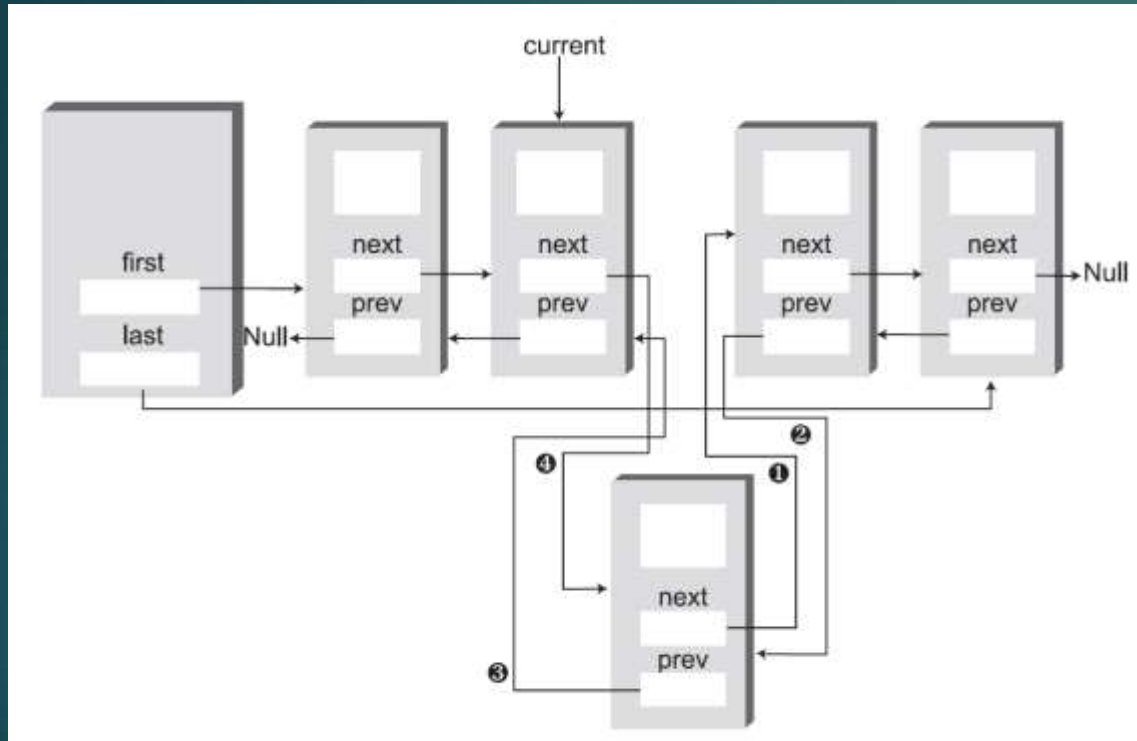
    if (isEmpty())
        last = newLink;
    else
        first.prev = newLink;

    newLink.next = first;
    first = newLink;
}

public void insertLast(int value) {
    DoubleLink newLink = new DoubleLink(value);

    if (isEmpty()) {
        first = newLink;
    } else {
        last.next = newLink;
        newLink.prev = last;
    }
    last = newLink;
}
```

Вставка в произвольной позиции



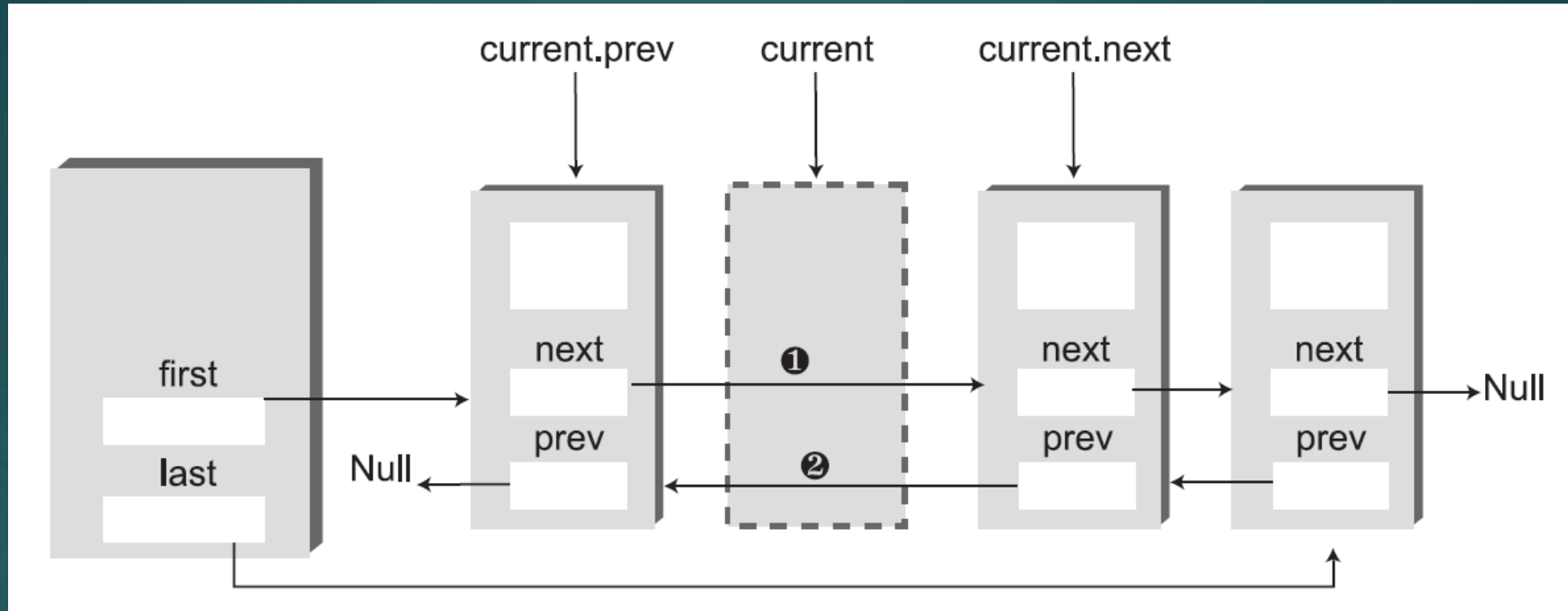
При вставке нового элемента после элемента с заданным ключом, операция несколько усложняется, потому что в этой ситуации необходимо изменить четыре ссылки. Прежде всего следует найти элемент с заданным ключом;

Затем, если позиция вставки находится не в конце списка, необходимо создать две связи между новым и следующим элементом, и еще две — между *current* и новым элементом.

Если новый элемент должен вставляться в конце списка, то его поле *next* должно содержать *null*, а поле *last* — ссылку на новый элемент.


```
public boolean insertAfter(int key, int value) {
    if (isEmpty())
        return false;
    DoubleLink current = first; // От начала списка
    while (current.iData != key) // Пока не будет найдено совпадение
    {
        current = current.next; // Переход к следующему элементу
        if (current == null)
            return false; // Ключ не найден
    }
    DoubleLink newLink = new DoubleLink(value); // Создание нового элемента
    if (current == last) // Для последнего элемента списка
    {
        newLink.next = null; // newLink --> null
        last = newLink; // newLink <-- last
    } else // Не последний элемент
    {
        newLink.next = current.next; // newLink --> старое значение next
        // newLink <-- старое значение next
        current.next.prev = newLink;
    }
    newLink.prev = current; // старое значение current <-- newLink
    current.next = newLink; // старое значение current --> newLink
    return true;
}
```

Удаление



Если удаляемый элемент не является ни первым, ни последним в списке, то в поле `next` элемента `current.previous` (элемент, предшествующий удаляемому) заносится ссылка на `current.next` (элемент, следующий после удаляемого), а в поле `previous` элемента `current.next` заносится ссылка на `current.previous`. В результате элемент `current` исключается из списка.

Если удаляемый элемент находится в первой или последней позиции списка, это особый случай, потому что ссылка на следующий или предыдущий элемент должна быть сохранена в поле `first` или `last`.



```
public DoubleLink deleteFirst() {
    DoubleLink deleteItem = first;

    if (first.next == null) {
        last = null;
    } else {
        first.next.prev = null;
    }

    first = first.next;
    return deleteItem;
}
```

```
public DoubleLink deleteLast() {
    DoubleLink deleteItem = last;

    if (first.next == null) {
        first = null;
    } else {
        last.prev.next = null;
    }

    last = last.prev;
    return deleteItem;
}
```

```
public DoubleLink deleteKey(long key) // Удаление элемента с заданным ключом
{
    if (isEmpty())
        return null;

    DoubleLink current = first; // От начала списка
    while(current.iData != key) // Пока не будет найдено совпадение
    {
        current = current.next; // Переход к следующему элементу
        if(current == null)
            return null; // Ключ не найден
    }
    if(current==first) // Ключ найден; это первый элемент?
        first = current.next; // first --> старое значение next
    else // Не первый элемент
        // старое значение previous --> старое значение next
        current.prev.next = current.next;
    if(current==last) // Последний элемент?
        last = current.prev; // старое значение previous <-- last
    else // Не последний элемент
        // Старое значение previous <-- старое значение next
        current.next.prev = current.prev;
    return current; // Возвращение удаленного элемента
}
```


Недостатки ДС

К недостаткам двусвязных списков следует отнести то, что при каждой вставке или удалении ссылки вам приходится изменять четыре ссылки вместо двух: две связи с предыдущим элементом и две связи со следующим элементом. И конечно, каждый элемент списка занимает чуть больше места из-за дополнительной ссылки.



Итераторы

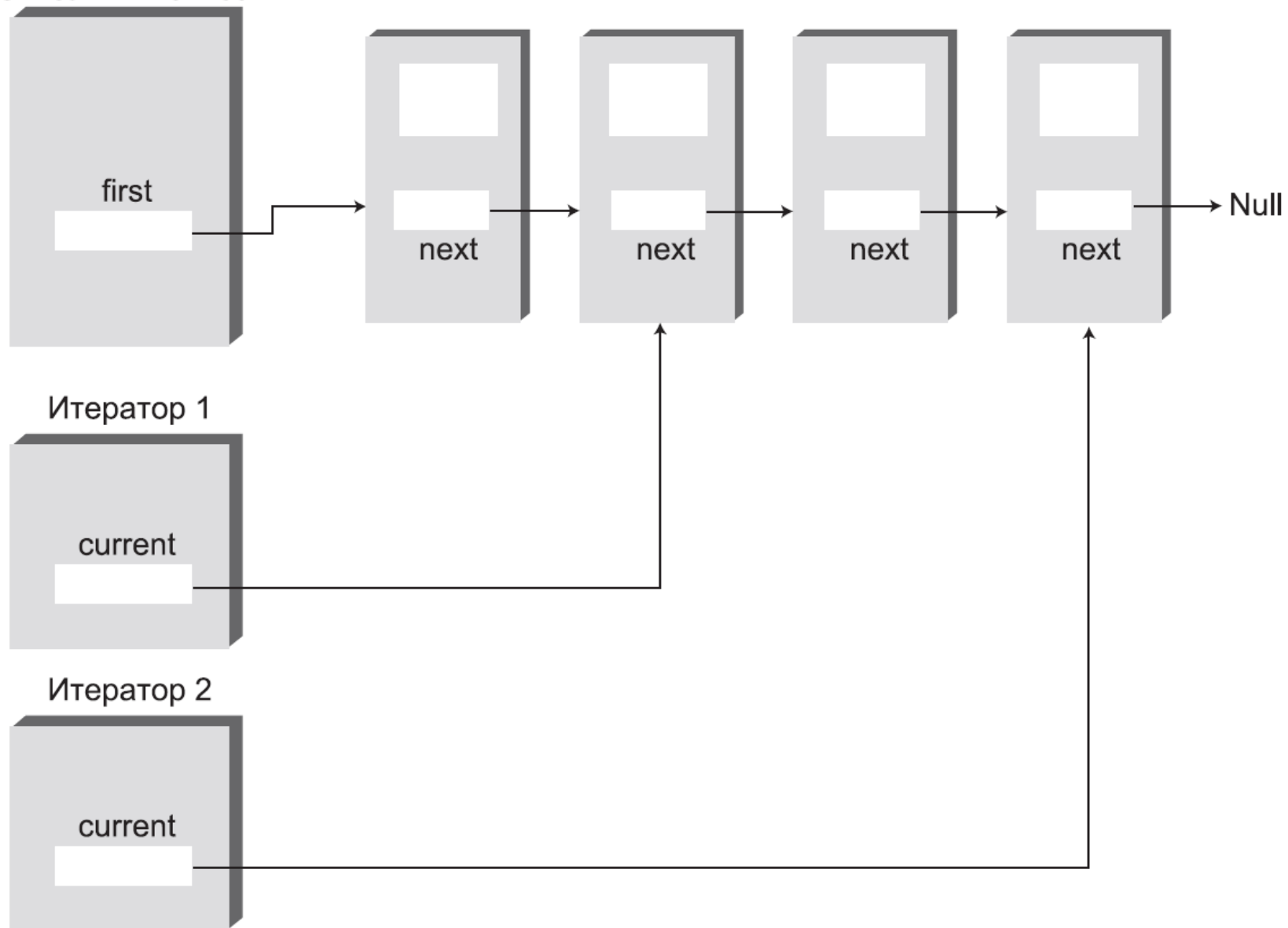


Объекты, содержащие ссылки на элементы структур данных и используемые для перебора элементов этих структур, обычно называются итераторами.

Итераторы


Чтобы использовать итератор, пользователь сначала создает список, а затем объект-итератор, ассоциированный с этим списком. На самом деле проще поручить создание итератора списку, потому что он может передать итератору полезную информацию — скажем, ссылку на `first`. По этой причине в класс списка включается метод `getIterator()`, который возвращает пользователю объект-итератор для данного списка.

Связанный список



Методы итераторов

- ▶ `reset()` — перемещение итератора в начало списка.
- ▶ `nextLink()` — перемещение итератора к следующему элементу.
- ▶ `getCurrent()` — получение элемента, на который указывает итератор.
- ▶ `atEnd()` — `true`, если итератор находится в конце списка.
- ▶ `insertAfter()` — вставка нового элемента после итератора.
- ▶ `insertBefore()` — вставка нового элемента перед итератором.
- ▶ `deleteCurrent()` — удаление элемента в текущей позиции итератора.




```
public class ListIterator {  
  
    private DoubleLink current; // Текущий элемент списка  
    private DoubleLink previous; // Предыдущий элемент списка  
    private LinkListIter ourList; // Связанный список  
  
    public DoubleLink getCurrent() // Получение текущего элемента  
    {  
        return current;  
    }  
}
```

```
public ListIterator(LinkListIter list) // Конструктор
{
    ourList = list;
    reset();
}


public void reset() // Возврат к 'first'
{
    current = ourList.getFirst();
    previous = null;
}

public boolean atEnd() // true, если текущим является
{
    return (current.next == null);
} // последний элемент

public void nextLink() // Переход к следующему элементу
{
    previous = current;
    current = current.next;
}
```

```
public void insertAfter(int dd) // Вставка после
{ // текущего элемента
    DoubleLink newLink = new DoubleLink(dd);
    if (ourList.isEmpty()) // Пустой список
    {
        ourList.setFirst(newLink);
        current = newLink;
    } else // Список не пуст
    {
        newLink.next = current.next;
        current.next = newLink;
        nextLink(); // Переход к новому элементу
    }
}
```



```
public void insertBefore(int dd) // Вставка перед
{ // текущим элементом
    DoubleLink newLink = new DoubleLink(dd);
    if (previous == null) // В начале списка
    { // (или пустой список)
        newLink.next = ourList.getFirst();
        ourList.setFirst(newLink);
        reset();
    } else // Не в начале списка
    {
        newLink.next = previous.next;
        previous.next = newLink;
        current = newLink;
    }
}
```

```
public long deleteCurrent() // Удаление текущего элемента
{
    long value = current.iData;
    if (previous == null) // Если в начале списка
    {
        ourList.setFirst(current.next);
        reset();
    } else // Не в начале списка
    {
        previous.next = current.next;
        if (atEnd())
            reset();
        else
            current = current.next;
    }
    return value;
}
```