

Дисциплина „Алгоритмы и структуры данных”

Лекційні заняття, їх тематика та обсяг

№ пор.	Назва теми	Обсяг навчальних занять (год.)	
		Лекції	СРС
<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>
1.1	Вступ до структур даних. Лінійні та нелінійні структури даних	2	4
1.2	Поняття алгоритму, його властивості. Рекурсивні алгоритми	2	3
1.3	Поняття сортування, класифікація алгоритмів сортування. Елементарні алгоритми	2	6
1.4	Сортування. Складні алгоритми сортування	2	10
1.5	Пошук. Алгоритми пошуку даних	2	2
1.6	Балансування BST-дерев	2	6
1.7	Методи аналізу алгоритмів	2	3
Усього за навчальною дисципліною		14	34

Лабораторні заняття, їх тематика та обсяг

№ пор.	Назва теми	Обсяг навчальних занять (год.)	
		Лабор. Заняття	СРС
<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>
„Структури даних та алгоритми їх обробки”			
1.1	Дослідження лінійних структур даних	2	2
1.2	Дослідження нелінійної структури даних "Хеш-таблиця"	2	2
1.3	Дослідження нелінійної структури даних «Дерево»	4	3
1.4	Дослідження алгоритмів сортування	4	4
1.5	Дослідження алгоритмів пошуку	2	4
1.6	Дослідження методів аналізу алгоритмів	4	3
Усього за навчальною дисципліною		18	18

Оцінювання окремих видів навчальної роботи студента

3 семестр		
„Структури даних та алгоритми їх обробки”		Мах кількість балів
Вид навчальної роботи	Мах кількість балів	
Виконання та захист лабораторної роботи	10 x 6 = 60	40
<i>Для допуску до виконання екзамену студент має набрати не менше 38 балів</i>		
Семестровий екзамен		
Усього за 3 семестр		100

**Відповідність рейтингових оцінок за окремі види навчальної роботи
в балах оцінкам за національною шкалою**

Рейтингова оцінка в балах		Оцінка за національною шкалою
Виконання та захист лабораторної роботи	Виконання екзаменаційної роботи	
9 – 10	36 – 40	Відмінно
8	30 – 35	Добре
6 – 7	24 – 29	Задовільно
менше 6	менше 24	Незадовільно

Відповідність підсумкової семестрової рейтингової оцінки в балах оцінці за національною шкалою та шкалою ECTS

Оцінка в балах	Оцінка за національною шкалою	Оцінка за шкалою ECTS	
		Оцінка	Пояснення
90-100	Відмінно	A	Відмінно (відмінне виконання лише з незначною кількістю помилок)
82 – 89	Добре	B	Дуже добре (вище середнього рівня з кількома помилками)
75 – 81		C	Добре (в загальному вірне виконання з певною кількістю суттєвих помилок)
67 – 74	Задовільно	D	Задовільно (непогано, але зі значною кількістю недоліків)
60 – 66		E	Достатньо (виконання задовольняє мінімальним критеріям)
35 – 59	Незадовільно	FX	Незадовільно (з можливістю повторного складання)
1 – 34		F	Незадовільно (з обов'язковим повторним курсом)

Литература

1. РОБЕРТ СЕДЖВИК, КЕВИН УЭЙН. Алгоритмы на Java, 4 изд-е – М.: ООО «И.Д. Вильямс», 2013. – 848с.
2. РОБЕРТ СЕДЖВИК. Фундаментальные алгоритмы на С++. Алгоритмы на графах. – СПб.: ООО "ДиаСофтЮП", 2002. – 496с.
3. Т. КОРМАН, Ч.ЛЕЙЗЕРСОН, Р.РИВЕСТ, К.ШТАЙН. Алгоритмы: построение и анализ, 2-е изд. – М.: Издательский дом «Вильямс», 2007. – 1296 с.
4. ДЖ. ФРИДЛ. Регулярные выражения. 2- изд. – СПб.: Питер, 2003. – 464с.
5. Г ШИЛДТ. Java. Полное руководство, 8 изд. – М.: ООО “И.Д. Вильямс”, 2012. – 1104с.

СТРУКТУРЫ ДАННЫХ

Структуры данных

Тип данных – это множество значений и набор операций с ними.

Набор данных – это совокупность объектов, описанных простыми или сложными типами данных.

Структура данных – это сгруппированный определенным образом набор данных, для которого определены правила обработки его элементов.

(способ хранения элементов, способы доступа к элементу)

Классификация структур данных

I) По правилу доступа:

1. Линейные

1.1. Массив (индексируемый тип данных)

1.2. Список

2. Нелинейные

2.1. Дерево

2.2. Хэш-таблица

II) По способу хранения

1. Векторное представление

2. Связанное представление

Операции над структурами данных

1. Добавить (вставить) элемент
 2. Удалить (изъять) элемент
 3. Копировать элемент
 4. Заменить элемент
 5. Переместить элемент
 6. Отобразить (вывести) набор данных
- 
- Над одним элементом

Структуры данных

Список – это набор данных, в котором элементы обрабатываются последовательно, начиная с некоторого основного (главного) элемента.

Списки бывают:

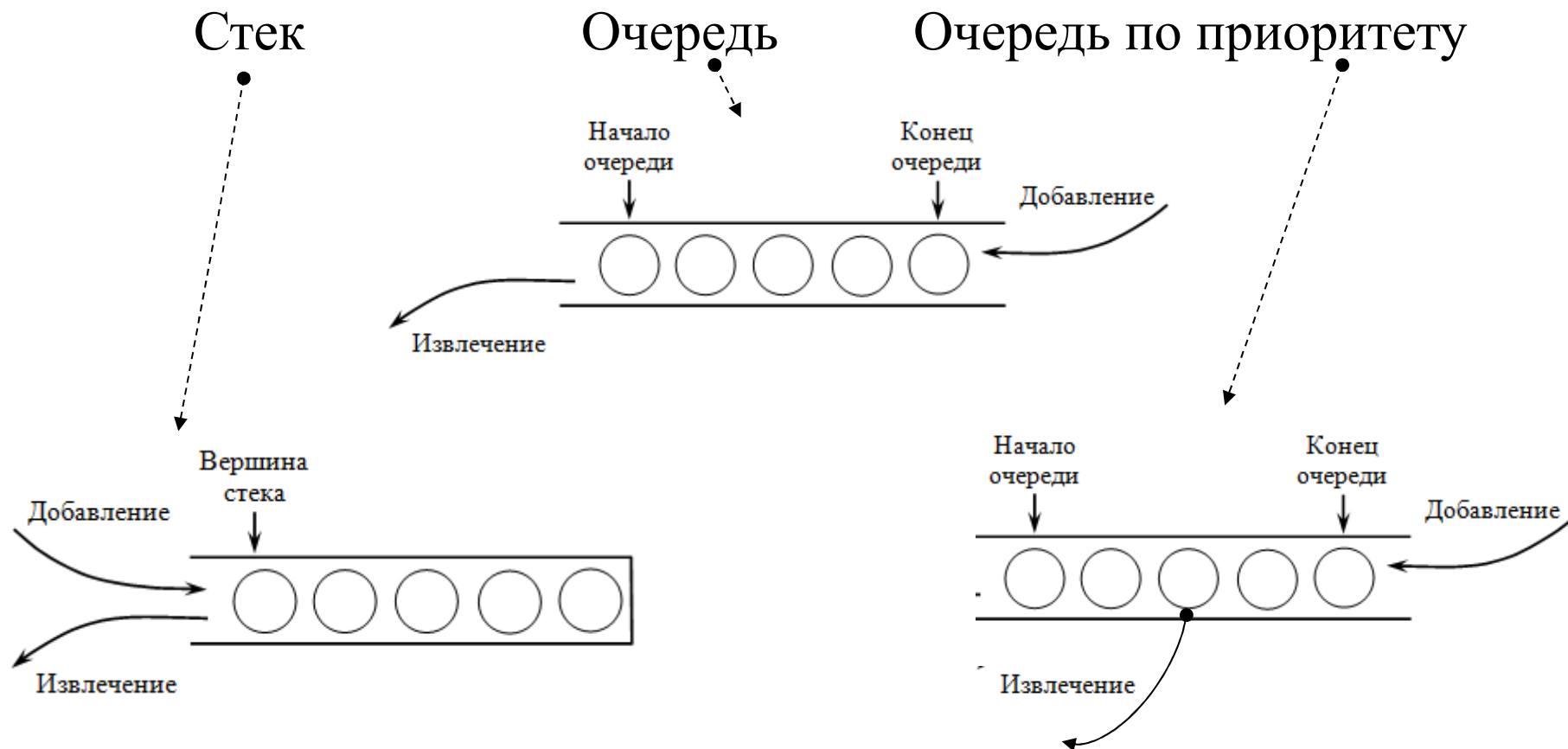
- однонаправленными;
- двунаправленными.

Способы реализации списков:

- Векторное представление;
- Связанное представление.

Структуры данных

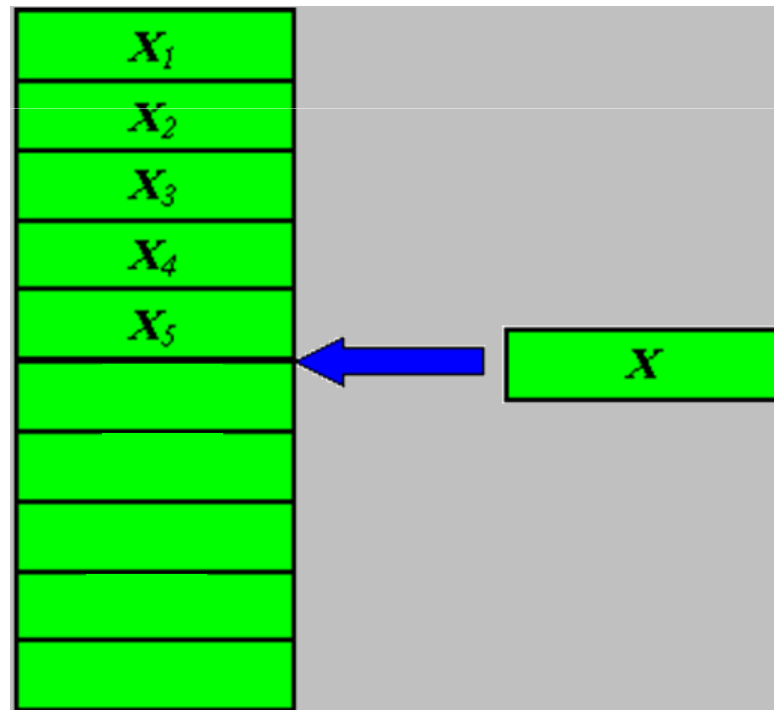
Частные случаи списков



Векторное представление списка

Особенности:

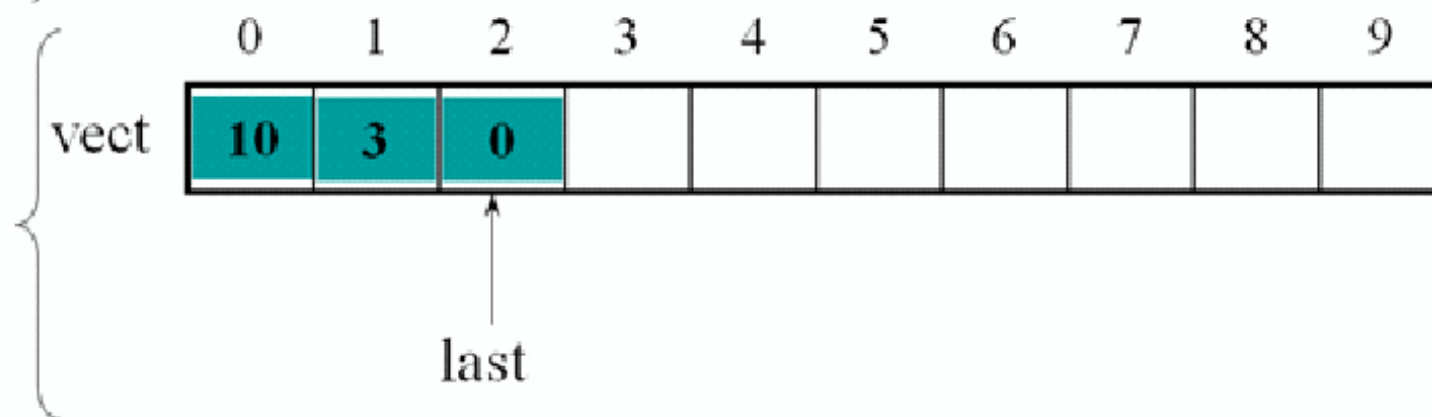
- ограничение размера (конечная область);
- наличие указателя на помещаемый элемент.



Структуры данных

Реализация списка как очереди

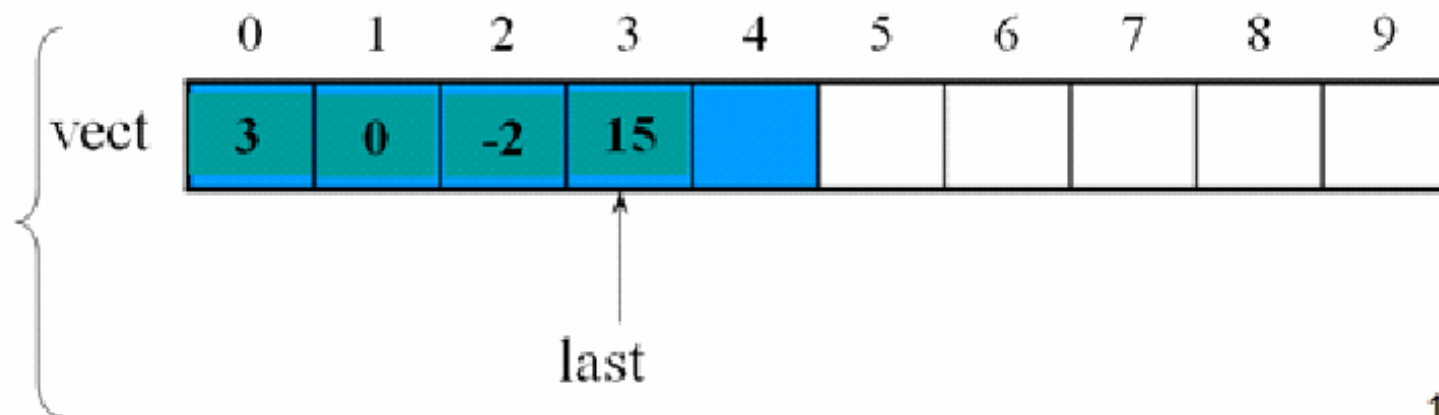
```
class Queue {  
    private int vect[];  
    private byte last;  
    Queue(int N) {  
        vect = new int [N];  
        last = -1;  
    }  
    boolean add(int elem) {  
        if (last == vect.length-1)  
            return false;  
        vect[++last] = elem;  
        return true;  
    }  
}
```



Реализация списка как очереди

```
Integer delete() {  
    if (last < 0)  
        return null;  
    int del = vect[0];  
    for(int i=0; i<last; i++)  
        vect[i] = vect[i+1];  
    vect[last--] = 0;  
    return del;  
}
```

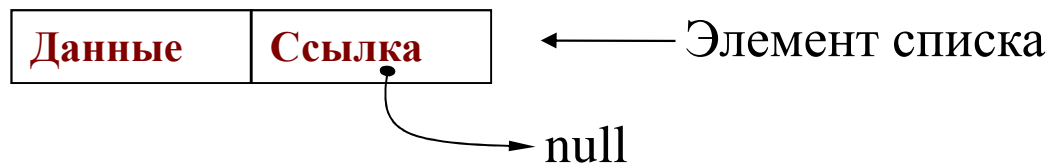
del = 10



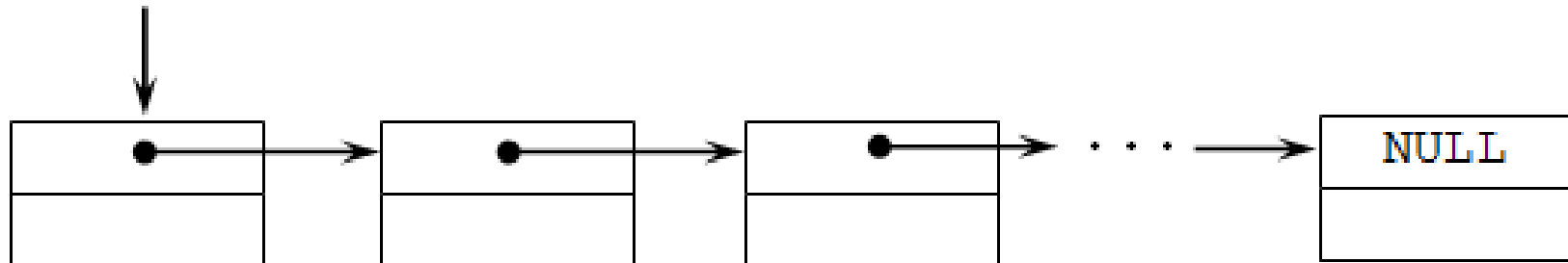
Структуры данных

В связанном представлении список – это набор данных, в котором каждый элемент состоит из двух частей: данных и ссылок на другие элементы списка.

Например, однонаправленный список



Указатель на первый
элемент списка



Структуры данных

```
class Element {  
    int data;  
    Element prev;  
    Element(int x, Element s) { data=x; prev = s;}  
}
```

```
class Stack {
```

```
    Element last;
```

```
    public void prin() {
```

```
        Element current = last;
```

```
        if (last == null)
```

```
            System.out.println("Stack empty!");
```

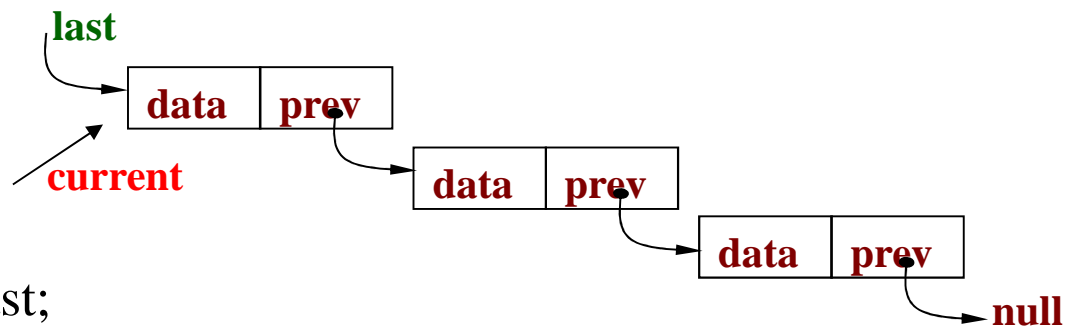
```
        else
```

```
            while (current != null) {
```

```
                System.out.print(current.data + "\t");
```

```
                current = current.prev;
```

```
            } }
```



Структуры данных

```
public boolean add(int data) {  
    last = new Element(data, last);  
    return true;  
}
```



```
public Integer delete() {  
    int del;  
    if (last != null) {  
        del = last.data;  
        last = last.prev;  
        return del;  
    }  
    return null;  
}
```

del = data

Структуры данных

Enter the element? Yes - press key 'y', No - press key 'n' y

Enter value -> 1

Enter the element? Yes - press key 'y', No - press key 'n' y

Enter value -> 2

Enter the element? Yes - press key 'y', No - press key 'n' y

Enter value -> 3

Enter the element? Yes - press key 'y', No - press key 'n' y

Enter value -> 4

Enter the element? Yes - press key 'y', No - press key 'n' y

Enter value -> 5

Enter the element? Yes - press key 'y', No - press key 'n' y

Enter value -> 6

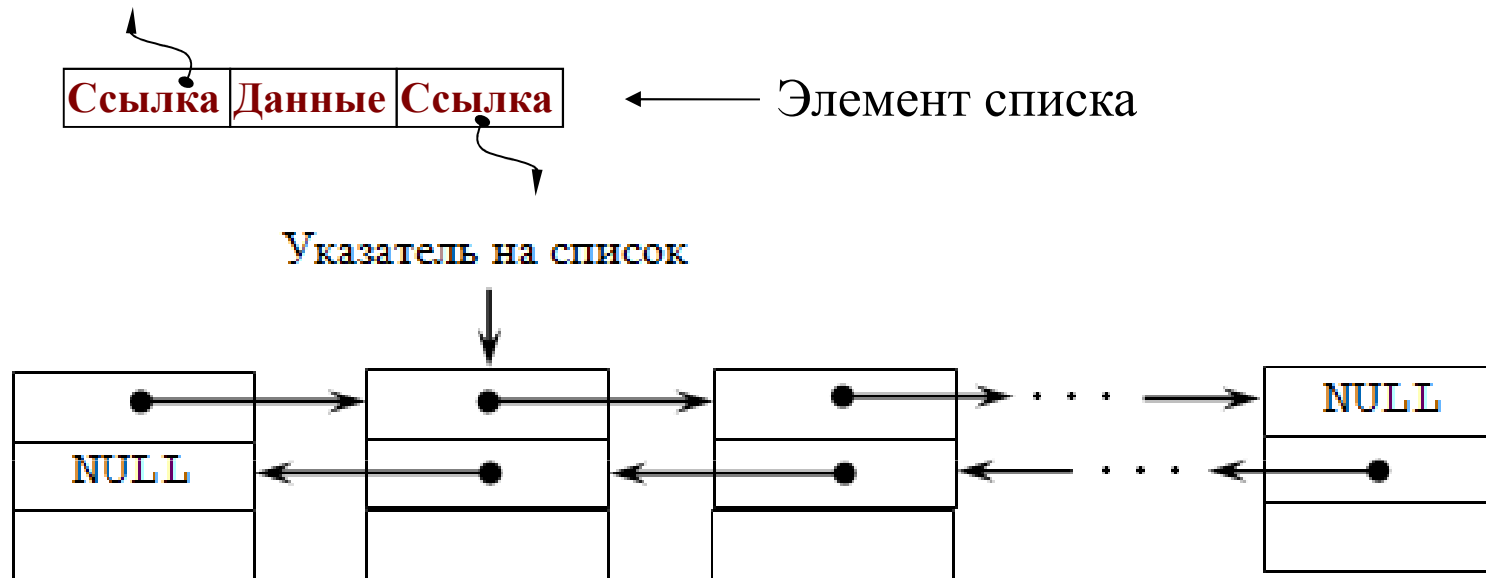
Enter the element? Yes - press key 'y', No - press key 'n' n

All element add!

Stack -> 6 5 4 3 2 1

Структуры данных

Пример двунаправленного списка в связанном представлении:

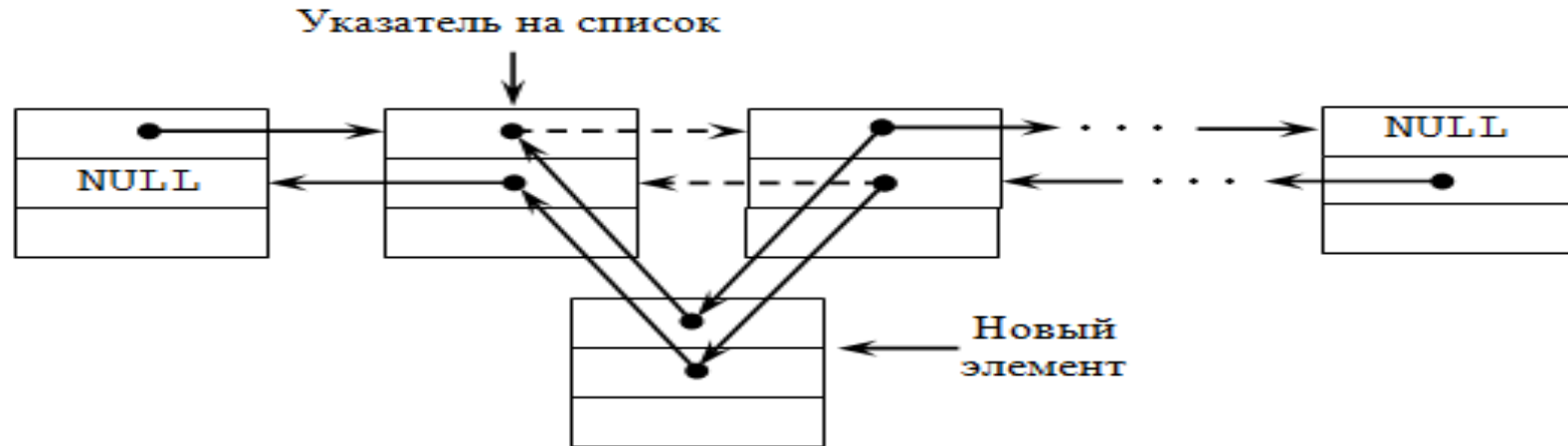


Добавление элемента в список может происходить в любую позицию.

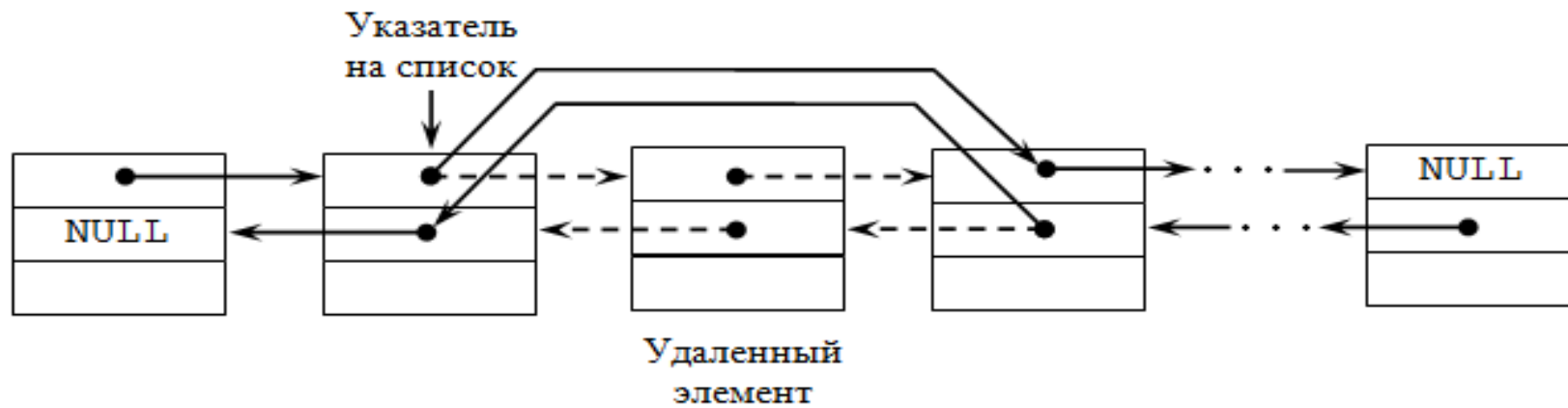
Удаление элемента – из любой позиции.

Структуры данных

Операция добавления:



Операция удаления:



Структуры данных

Реализация двунаправленного списка в связанном представлении:

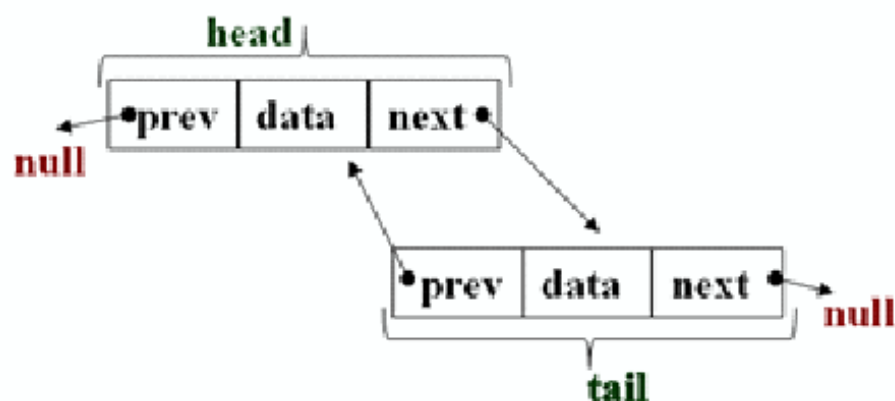
```
class Element {  
    int data;  
    Element prev, next;  
    Element(int data) { this.data = data; }  
}
```

```
class Spisok {  
    Element head,  
           tail;
```

Структуры данных

// добавление элемента в конец списка

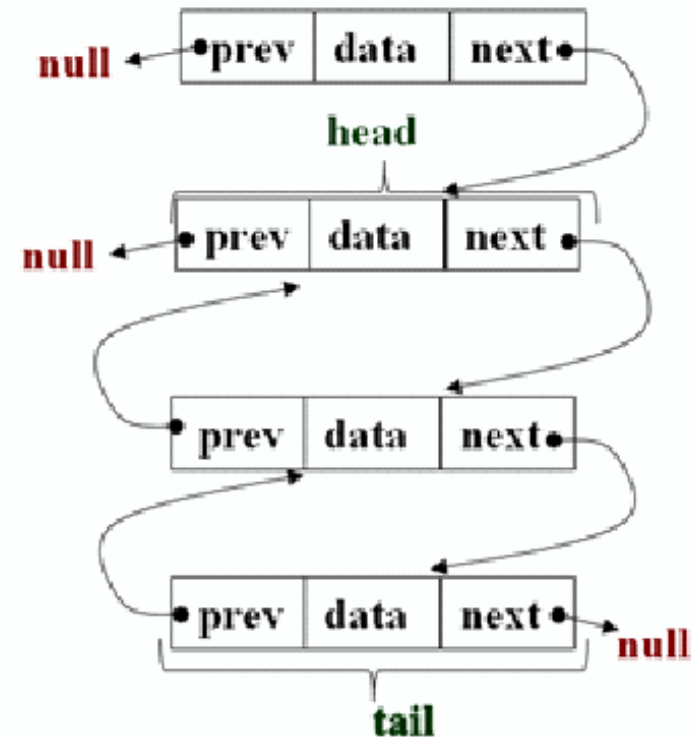
```
public boolean addLast(int data) {  
    Element newElement = new Element(data);  
    if (head == null) {  
        head = newElement;  
        tail = head;  
    }  
    else {  
        newElement.prev = tail;  
        tail.next = newElement;  
        tail = newElement;  
    }  
    return true;  
}
```



Структуры данных

// удаление элемента по ключу

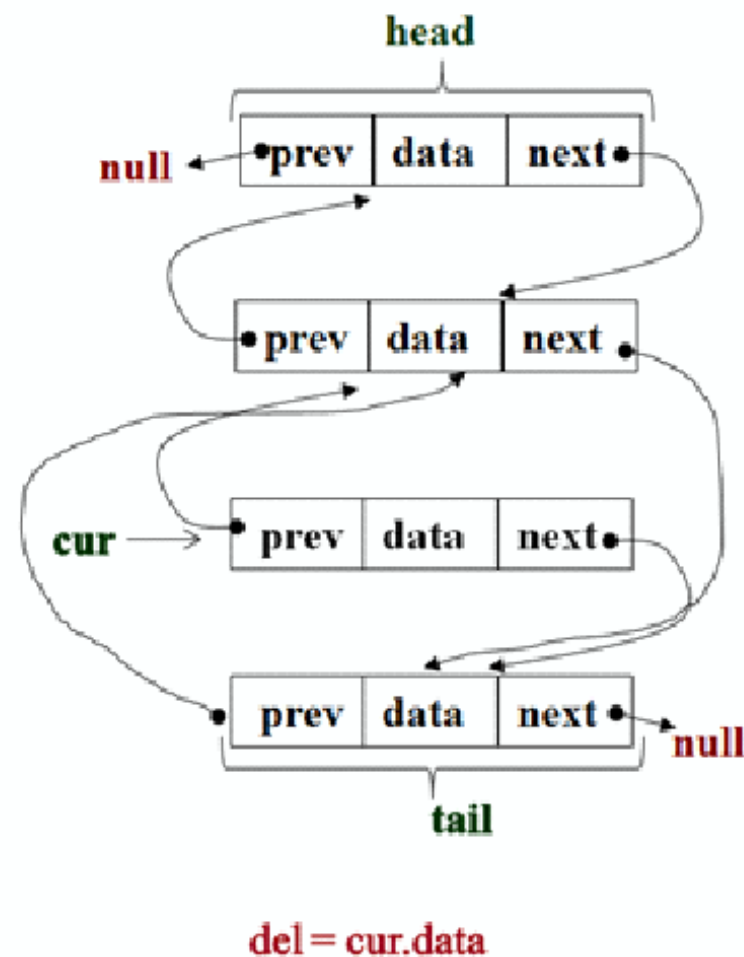
```
Integer delete (int value) {  
    Integer del = null;  
    if (head.data == value) {  
        del = head.data;  
        head = head.next;  
        if (head != null)  
            head.prev = null;  
        else  
            tail = null;  
    }  
    else {  
        if (tail.data == value) {  
            del = tail.data;  
            tail = tail.prev;  
            tail.next = null;  
        }  
    }  
}
```



del = head.data

Структуры данных

```
else {      Element cur = head.next;
            while(cur != tail){
                if (cur.data != value)
                    cur = cur.next;
                else {
                    del = cur.data;
                    cur.next.prev = cur.prev;
                    cur.prev.next = cur.next;
                    break;
                }
            }
            return del;
        }
    }
```



МНОЖЕСТВО

Структуры данных

Множество – это набор уникальных объектов (*например, словари*), для представления которых используется хэш-таблица, основанная на массивах.

Хэш-таблица – это массив ассоциативного типа, в котором элементы сохраняются (распределяются) по методу хэширования.

Хэширование – это процесс кодирования (преобразования) объектов индексированием по ключу.

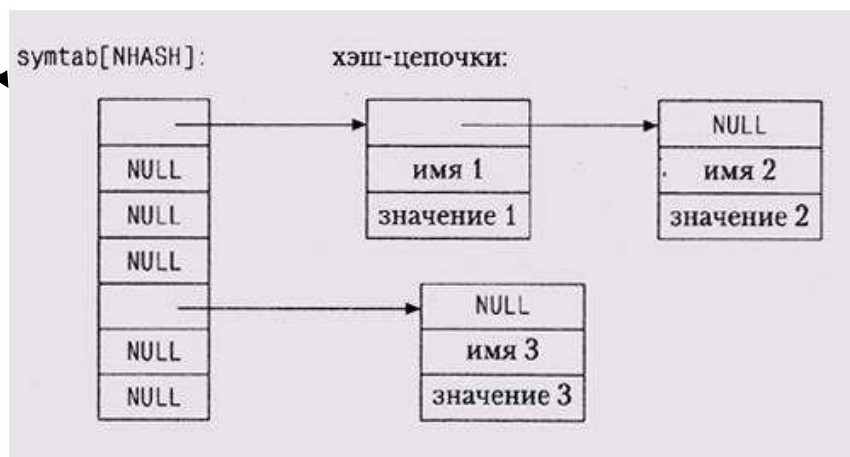
Особенность:

Объекты должны идентифицироваться ключом (значение любого типа).

Структуры данных

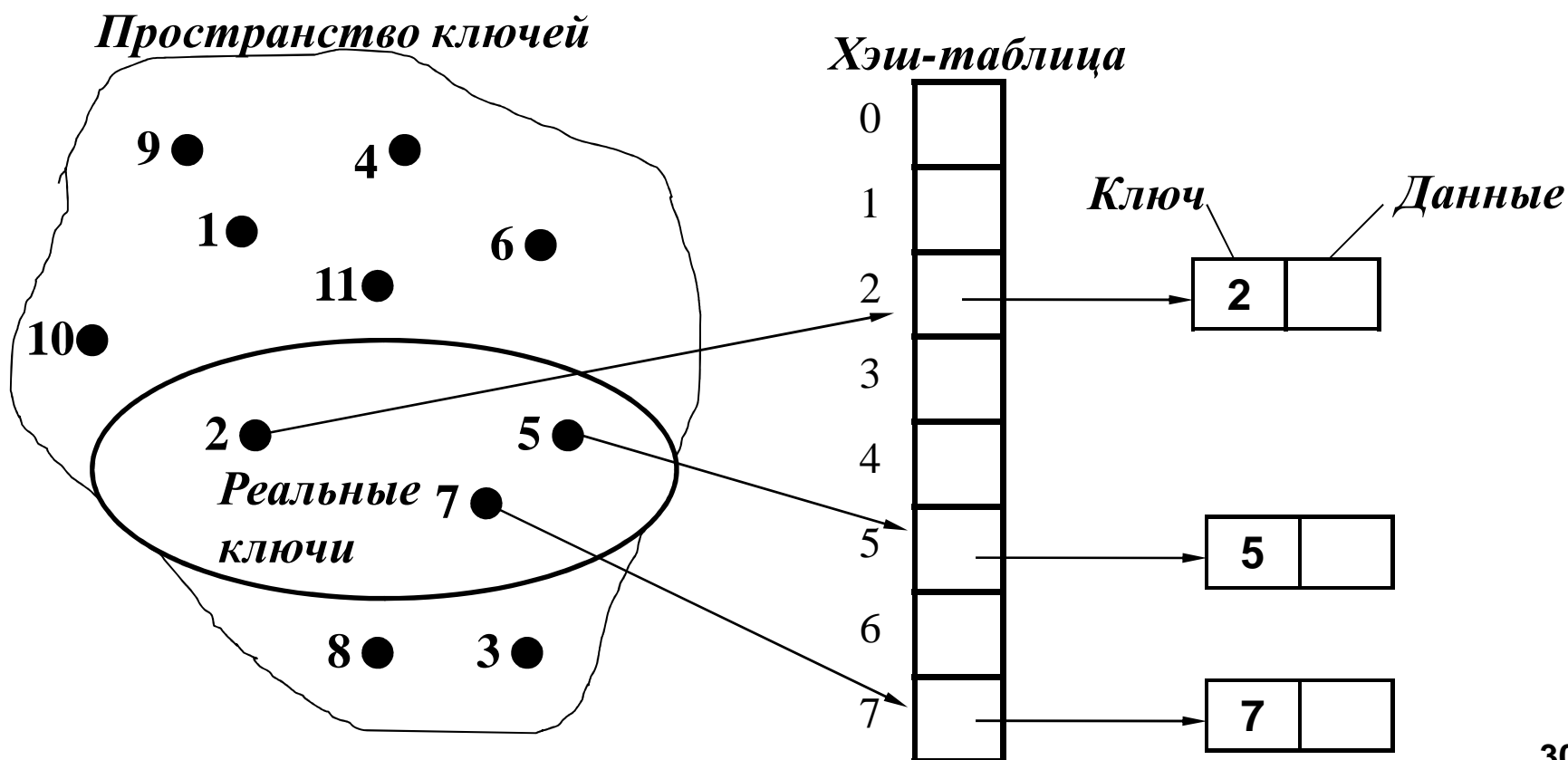
В зависимости от способа использования ключей объектов различают:

1. Таблицы с прямой адресацией
- 2. Хэш-таблицы с раздельным связыванием
3. Хэш-таблицы с открытой адресацией



Таблицы с прямой адресацией

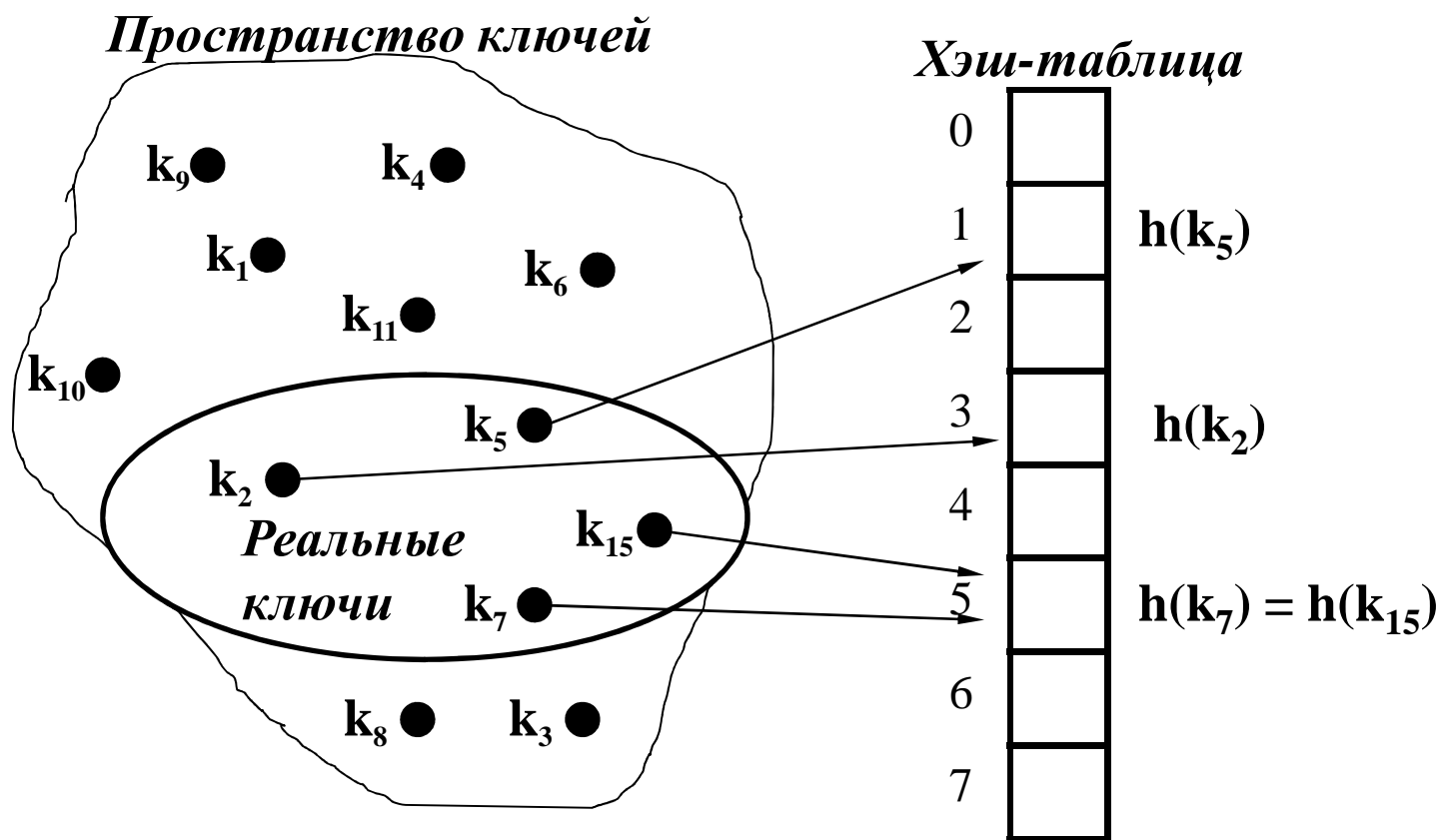
Объекты (элементы) располагаются непосредственно в ячейках хэш-таблицы (массива) и ключ объекта соответствует индексу массива.



Структуры данных

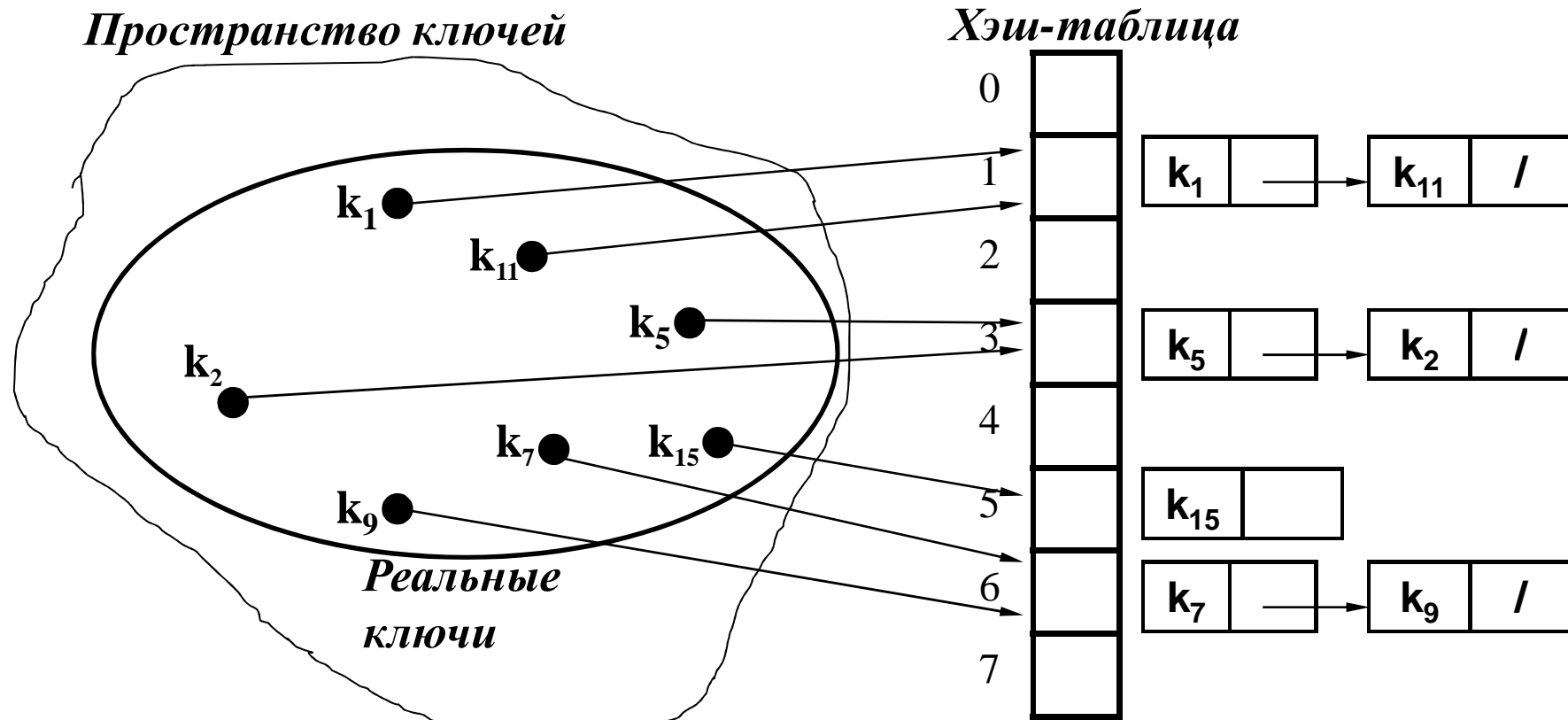
Динамические хэш-таблицы

Для вычисления индекса элемента в массиве используется хэш-функция.



Коллизия – это ситуация, когда два ключа могут быть хэшированы в элемент массива с одинаковым индексом. 31

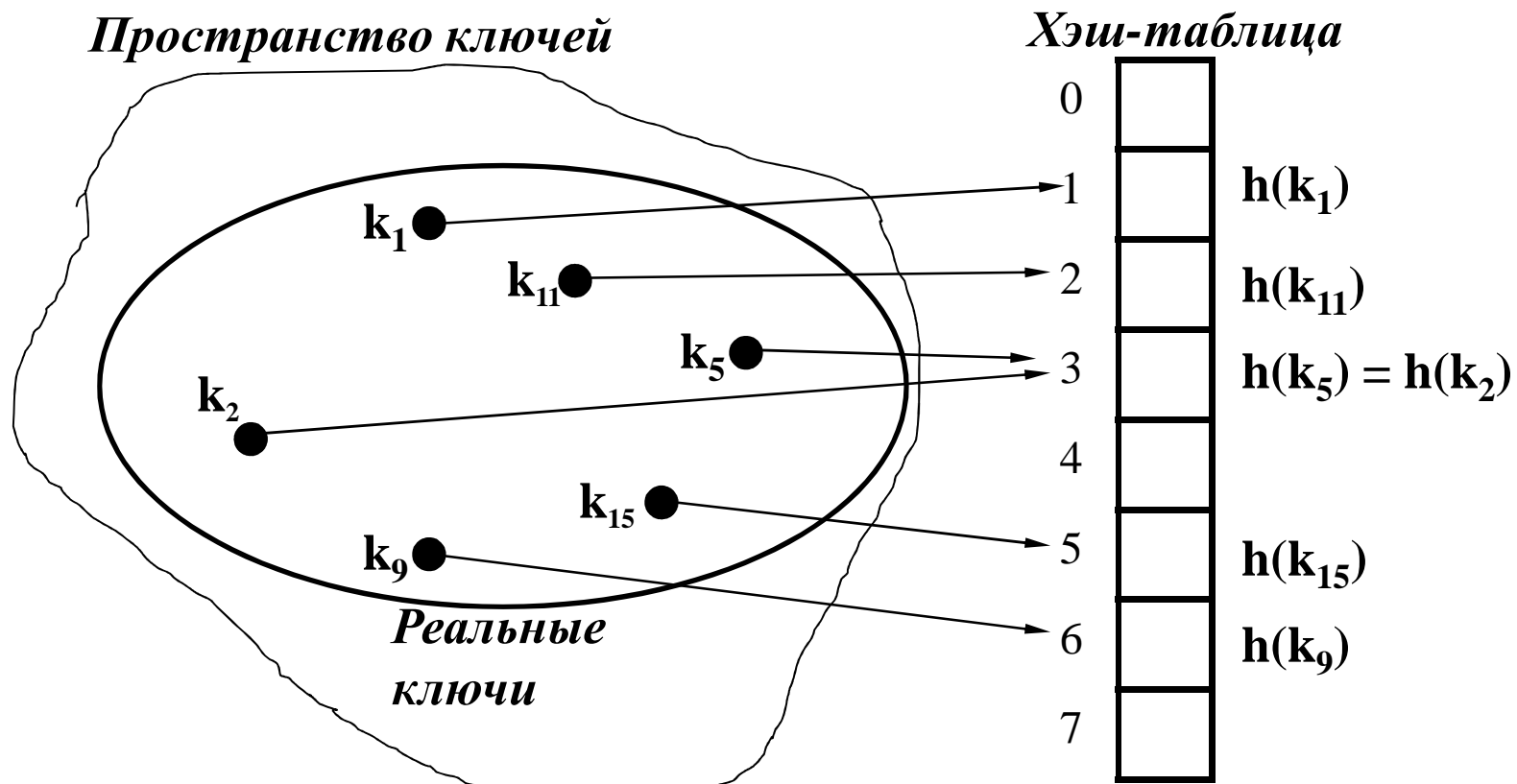
Структуры данных



Хэш-таблицы с открытой адресацией

1. Все объекты хранятся непосредственно в ячейках хэш-таблицы.
2. Для вычисления индекса размещения объекта используется хэш-функция.
3. Для разрешения коллизии определяется последовательность элементов, среди которых отыскивается свободный элемент.

Структуры данных



Структуры данных

Определение последовательности элементов для разрешения коллизии в хэш-таблицах с открытой адресацией называется *зондированием*.

Для вычисления индексов элементов последовательности используется выражение:

$$P(k) = c_1 k^2 + c_2 k + c_3.$$

где c_1, c_2, c_3 – некоторые константы со значениями $[0, 0.5, 1]$;
 $k = 1, 2, 3, \dots$

Структуры данных

1. Линейное зондирование ($c_1 = c_3 = 0, c_2 = 1$) $\rightarrow P(k) = k$

$$\text{index} = (i + k) \% \text{size},$$

где i – индекс, вычисленный хэш-функцией;

$k=1, 2, 3, \dots, \text{size};$

size – размер хэш-таблицы

$\text{size} = 8$

$i = 3$

1) $\text{index} = (i + 1) \% \text{size} = 4$

2) $\text{index} = (i + 2) \% \text{size} = 5$

Хэш-таблица

0	■
1	□
2	■
3	■
4	■
5	□
6	■
7	□

Структуры данных

2. Квадратичное зондирование ($c_2 = c_3 = 0, c_1 = 1$) $\rightarrow P(k) = k^2$

$$\text{index} = (i + k^2) \% \text{size},$$

где i – индекс, вычисленный хэш-функцией;

$k=1, 2, 3, \dots, \text{size};$

size – размер хэш-таблицы

$\text{size} = 8$

$i = 1$

1) $\text{index} = (i + 1^2) \% \text{size} = 2$

2) $\text{index} = (i + 2^2) \% \text{size} = 5$

Хэш-таблица

0	
1	
2	
3	
4	
5	
6	
7	

3. Двойное зондирование

$$\mathbf{index = (i + k * p) \% size,}$$

где i – индекс, вычисленный хэш-функцией;

$k=1, 2, 3, \dots, size$;

$size$ – размер хэш-таблицы;

$p = h_2(key) \rightarrow$ индекс, вычисленный вторичной хэш-функцией.

Например,

$$h_2(key) = i + (key \% size)$$

или

$$h_2(key) = (key \% size) + 1$$

Структуры данных

Хэш-таблица

key = 77

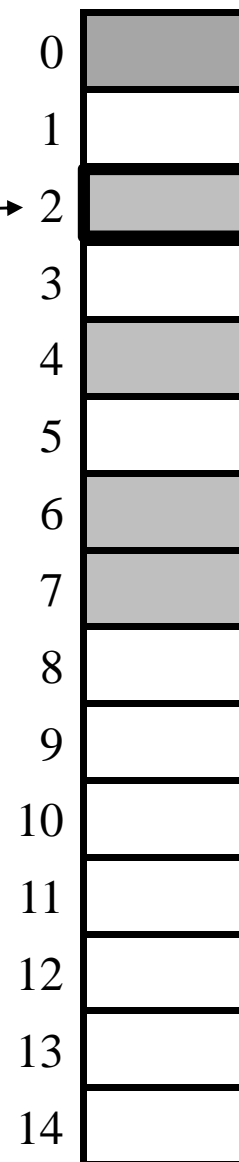
size = 15

i = 2

k = i + (key % size) = 4

1) index = (i+1*k)%size = 6

2) index = (i+2*k)%size = 10

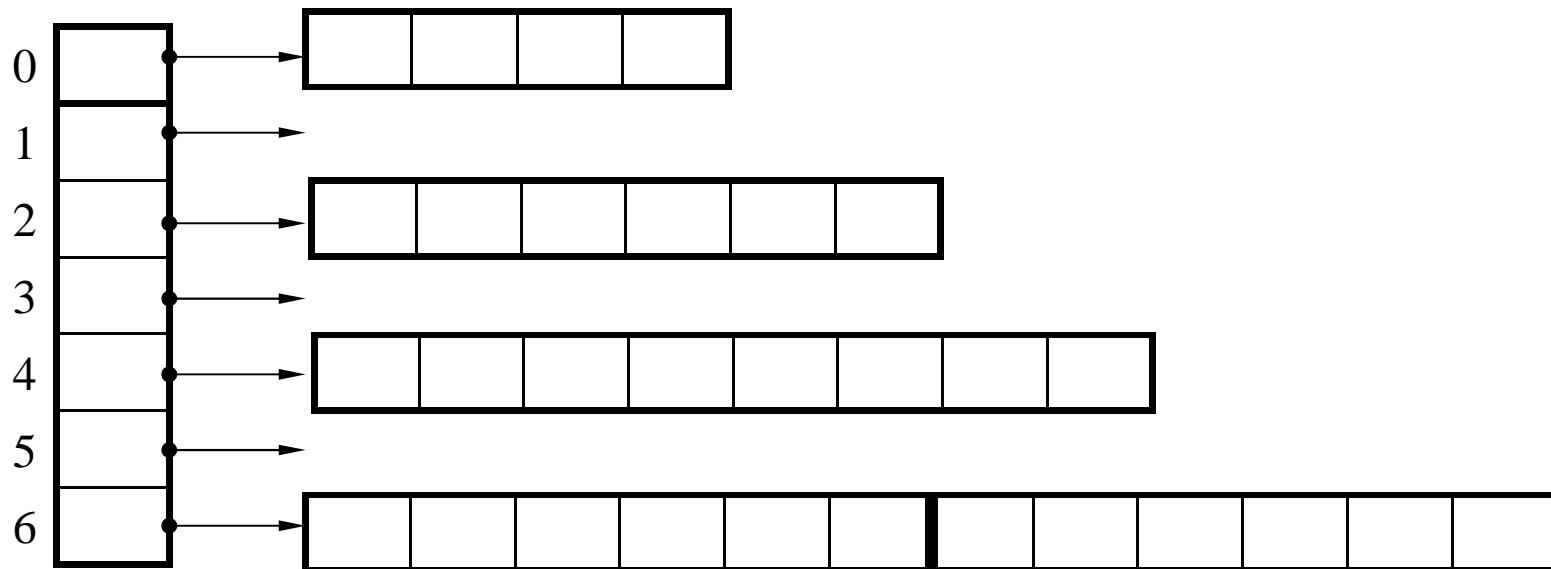


0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	

4. Идеальное зондирование

Использование двумерного массива и двух хэш-функций:
первичной и вторичной:

- первичная для вычисления индекса строки массива;
- вторичная для вычисления индекса столбца массива.



Структуры данных

Применяется для статического множества ключей, поэтому размер вторичных хэш-таблиц (S_i) определяется количеством объектов, хешированных в i -ю ячейку первичной хэш-таблицы ($m_i = n^2$).

Вторичные хэш-функции подбираются таким образом, чтобы исключить возникновение коллизий.

Методы хэширования

1. Метод деления

- а) значение ключа – целое положительное число (или ключ легко можно представить таким числом);
- б) делитель – простое число далекое от степени двойки.

Индекс вычисляется путем применения операции получения остатка от деления ключа на размерность хэш-таблицы.

Например, $m=13$, $key = 24$.

$$h(key) = key \% m = 11.$$

Структуры данных

2. Метод умножения

- а) значение ключа – должно быть представлено вещественным значением в диапазоне от 0 до 1;
- б) множитель – значение равное степени двойки.

Индекс вычисляется по следующему правилу:

1. ключ умножается на константу $A = \underline{0,6180339887}$
2. из результата выделяется дробная часть
3. полученное значение умножается на размерность хэш-таблицы и округляется до меньшего целого.

Например, $m = 128$, $key = 1234565$.

$$h(key) = \lfloor key * A \rfloor * m = 16$$

$$\lfloor 1234565 * 0,6180339887 \rfloor = 763003, \underline{\underline{1312594155}}$$

$$0,1312594155 * 128 = \underline{\underline{16}},801205184.$$

3. Метод универсального хэширования

Индекс – вычисляется с использованием хэш-функции, которая случайно выбирается из некоторого сформированного класса хэш-функций.

Пример реализации хэш-таблиц

Структуры данных

//Хэш-таблица с раздельным связыванием

```
class Coord {  
    int x;  
    int y;  
  
    Coord(int a, int b) { x=a; y=b; }  
  
    public int hashCode() { return ((x<<1) + y); }  
  
    public boolean equals(Object o) {  
        return (x == ((Coord )o).x)&&(y == ((Coord )o).y); }  
  
    public String toString() {  
        return (String.valueOf(x) + " " + String.valueOf(y)); }  
}
```

Структуры данных

```
class Table {  
    // внутренний класс для описания элемента хэш-таблицы  
    class HashObj {  
        Coord data;  
        HashObj next;  
        HashObj(Coord data) {  
            this.data = data;  
        }  
    }  
  
    HashObj mas[];  
  
    Table(int r) {    mas = new HashObj [r];    }  
    private int Code(Coord obj) {  
        return (obj.hashCode() % mas.length);  
    }  
}
```

Структуры данных

```
public boolean ins(Coord new_ob) {  
    int h = Code(new_ob);  
    if (mas[h] == null)  
        mas[h] = new HashObj(new_ob);  
    else  
        return Col(mas[h], new HashObj(new_ob));  
    return true;  
}  
  
private boolean Col(HashObj cur, HashObj nElem) {  
    while ((cur != null) && (!cur.data.equals(nElem.data)))  
        if (cur.next == null) {  
            cur.next = nElem;  
            return true;  
        } else cur = cur.next;  
    return false;  
}
```


Структуры данных

Enter size HashTable -> 10

Add the object? Yes - press key 'y', No - press any key y

Enter values element -> a and b 5 6

Add the object? Yes - press key 'y', No - press any key y

Enter values element -> a and b 8 9

Add the object? Yes - press key 'y', No - press any key y

Enter values element -> a and b 0 1

Add the object? Yes - press key 'y', No - press any key y

Enter values element -> a and b 2 3

Add the object? Yes - press key 'y', No - press any key y

Enter values element -> a and b 7 7

Add the object? Yes - press key 'y', No - press any key y

Enter values element -> a and b 0 1

Add the object? Yes - press key 'y', No - press any key y

Enter values element -> a and b 1 5

Add the object? Yes - press key 'y', No - press any key n

Структуры данных

HashTable

0)

1) 0 1; 7 7;

2)

3)

4)

5) 8 9;

6) 5 6;

7) 2 3; 1 5;

8)

9)

Структуры данных

Пример хэш-таблицы с открытой адресацией (при возникновении коллизии выполняется квадратичное зондирование)

```
class DataI {  
    int data;  
    short key;  
    DataI(int a) {    key = (short)(Math.random()*10);  
                    data = a;  
    }  
    public boolean equals(Object obj) {  
        return (data == ((DataI)obj).data) && (key == ((DataI)obj).key); }  
}  
  
class HashTable {  
    private DataI hash[];  
    HashTable(int s) {    hash = new DataI [s];    }  
    private int Index(short k){    return k%hash.length;    }
```

Структуры данных

```
boolean insert(DataI a) {  
    int i = Index(a.key);  
    int index ;  
    for(int k = 0; k<hash.length; k++) {  
        index = (i + (int)Math.pow(k, 2) ) % hash.length;  
        if (hash[index] == null) {    hash[index] = a;        return true;        }  
        else if (hash[index].equals(a))  
            return false;  
    }  
}  
  
void prin() {  
    for(int i=0; i<hash.length; i++)  
        if (hash[i] == null)        System.out.println(i);  
        else    System.out.println(i + ": " + hash[i].data + "(" + hash[i].key + ")");  
}  
}
```

Структуры данных

Hash table:

0: 67(0)

1: 89(0)

2: 34(2)

3: 56(3)

4: 12(4)

5

6

7: 23(7)

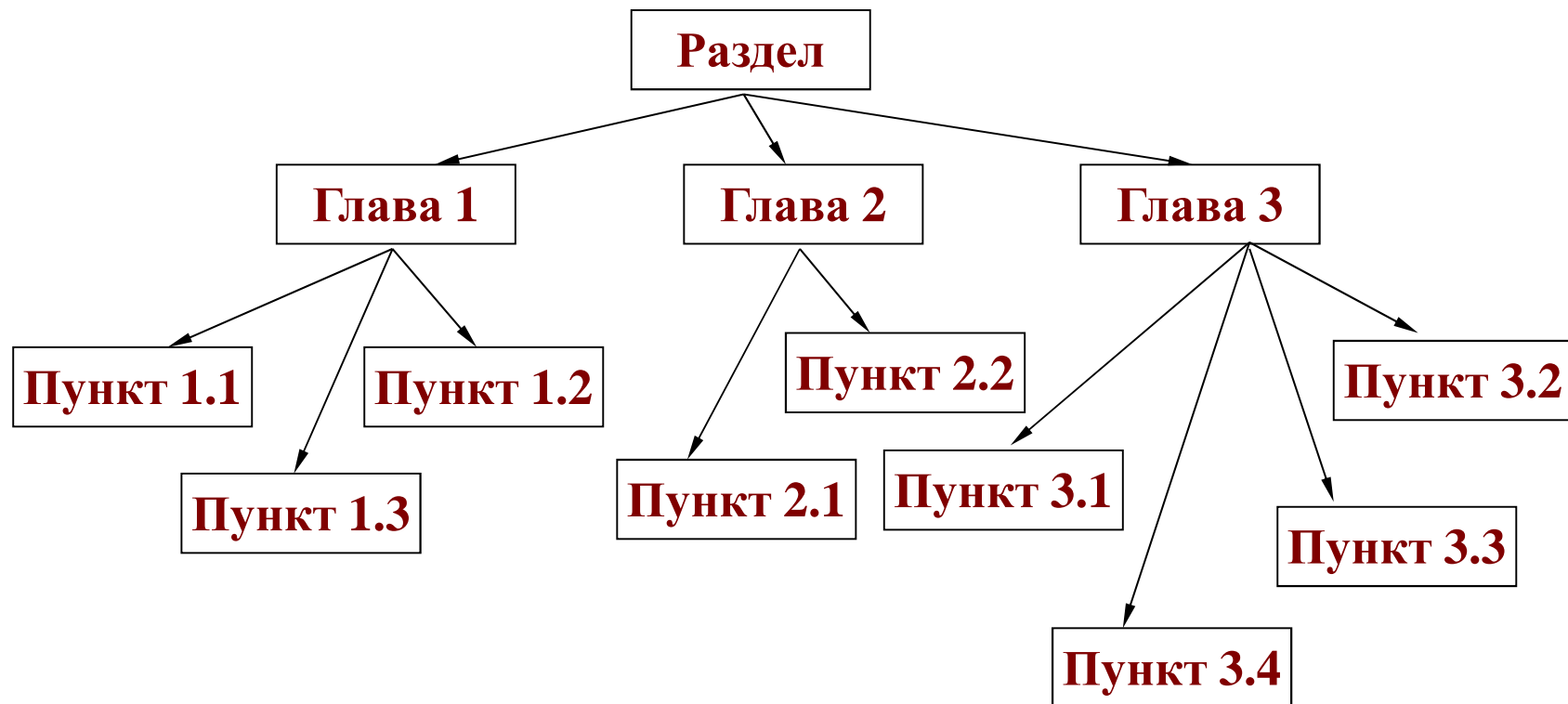
8: 90(3)

9: 78(3)

Нелинейная структура данных «дерево»

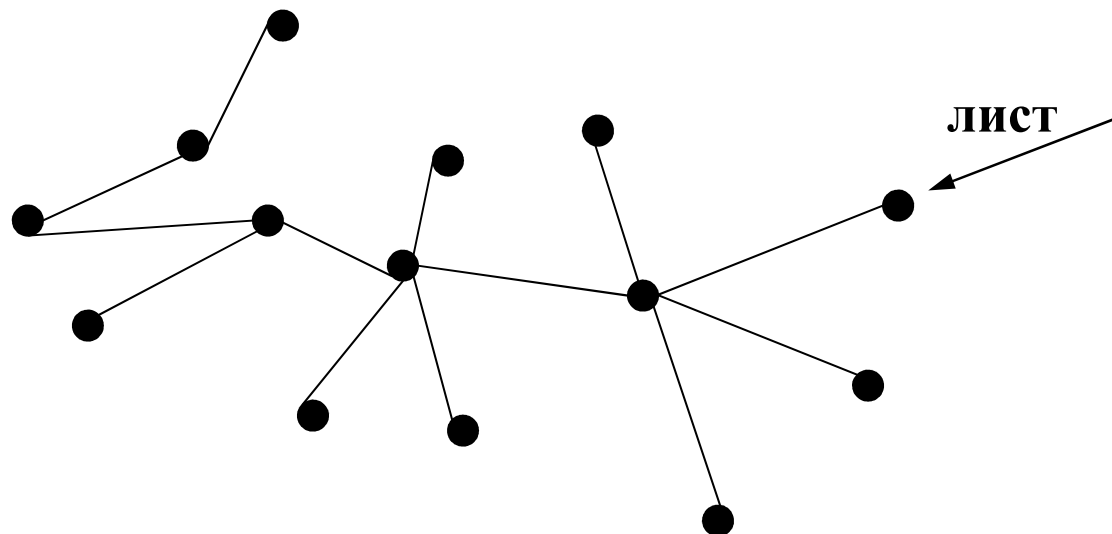
Структуры данных

Дерево – нелинейная структура данных, в которой каждый элемент (узел) имеет ссылки на множество других элементов (узлов), связанных с ним определенными правилами (требованиями).



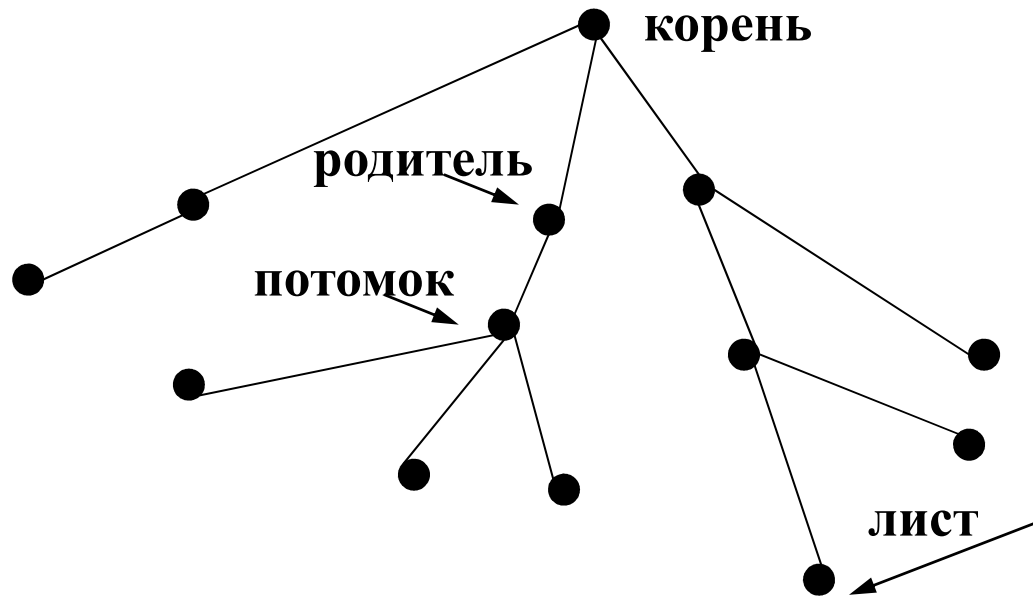
Типы деревьев

Свободное дерево - неупорядоченное дерево со множественными связями между узлами (листами).



Структуры данных

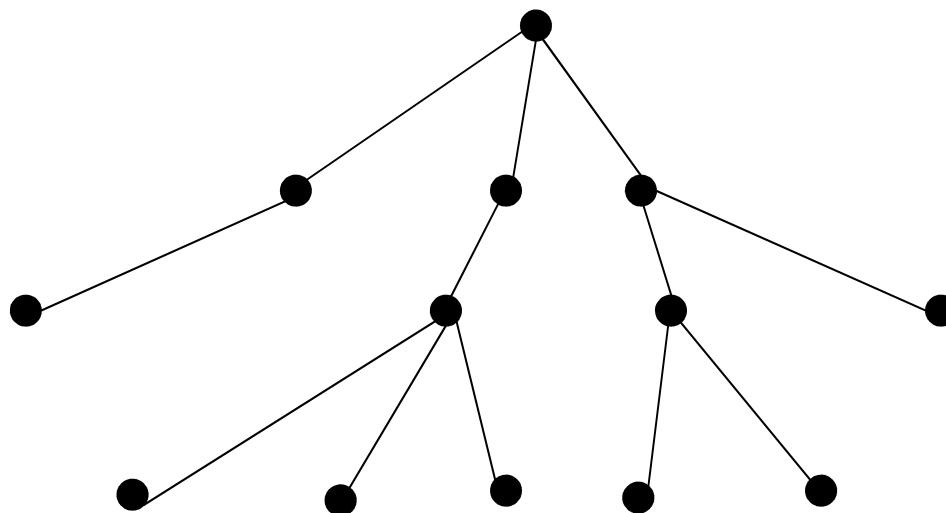
Дерево с корнем – неупорядоченное дерево, в котором один из узлов называется корнем и не имеет других узлов над собой, т.е. у каждого узла есть только один узел над ним (родитель), кроме корневого узла.



Структуры данных

Упорядоченное дерево - это дерево с корнем, в котором определен порядок следования потомков (дочерних узлов), однако каждый узел имеет произвольное количество ссылок на другие узлы.

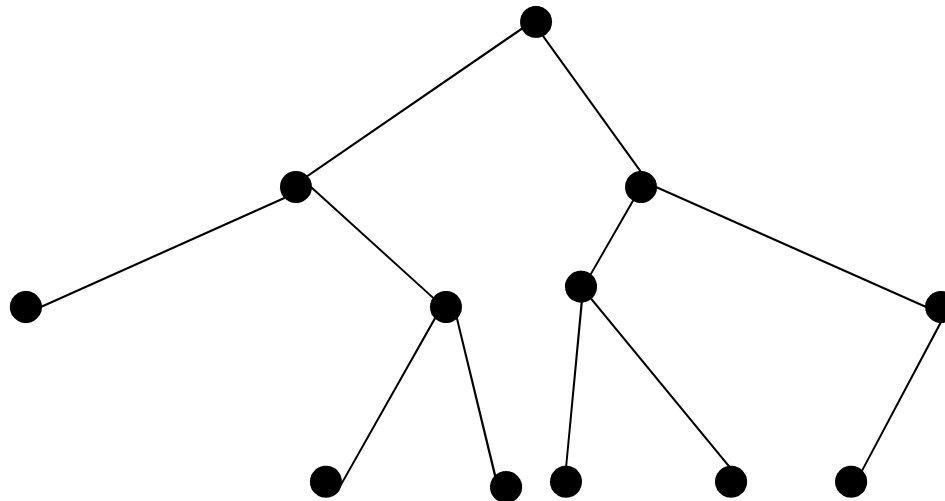
Например, генеалогическое древо.



Структуры данных

М-арное дерево – это упорядоченное дерево с корнем, у которого каждый узел имеет конечное одинаковое количество ссылок на другие узлы.

Бинарное дерево – это упорядоченное дерево с корнем, у которого каждый узел имеет две ссылки (левую и правую) на другие узлы.

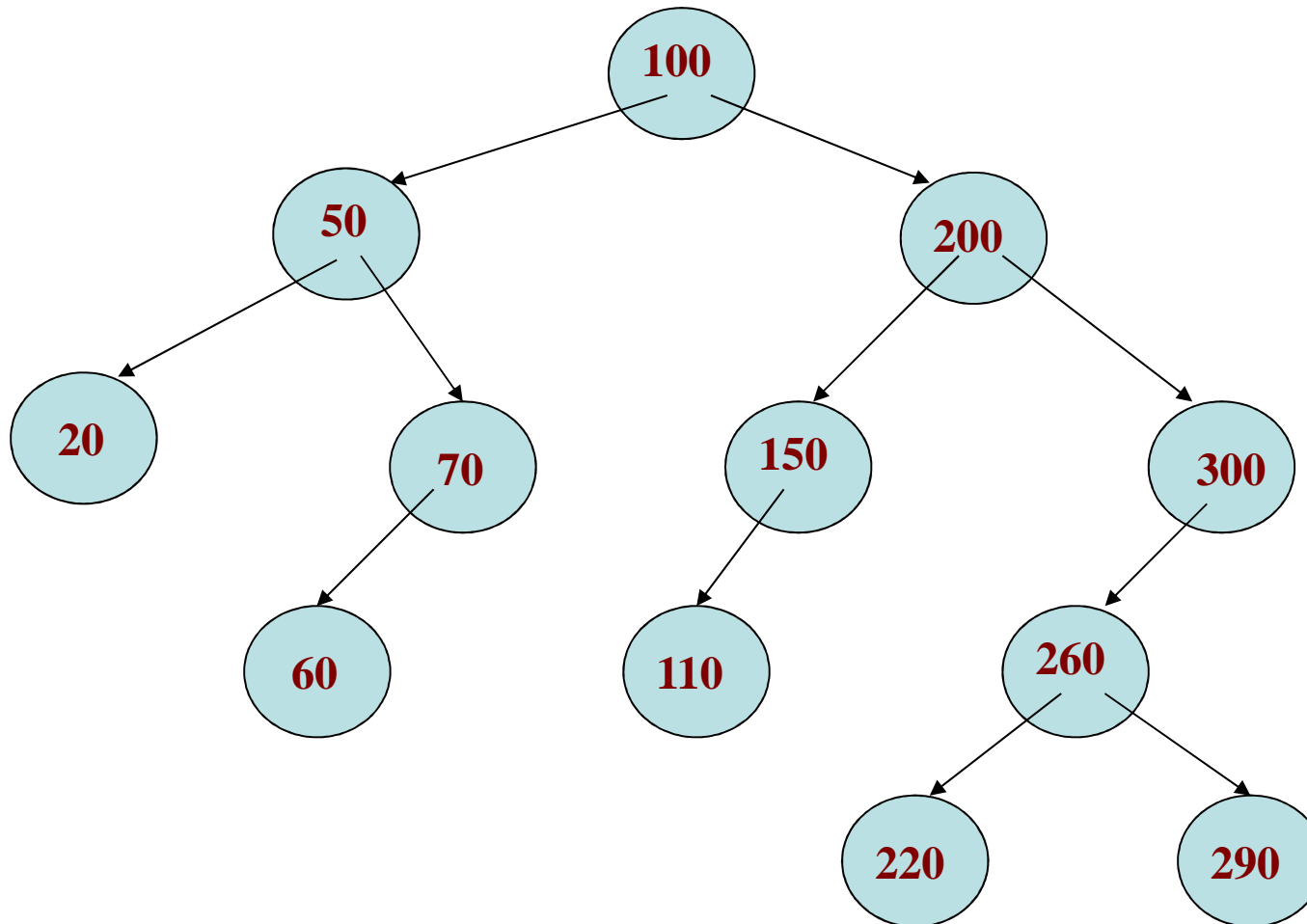


Правило формирования бинарного дерева сравнения

- новый элемент (узел) всегда становится листом дерева;
- поиск позиции начинается с вершины дерева, постепенно смещаясь вниз;
- если значение нового узла меньше текущего, то переходим в левую ветку текущего узла;
- если левый узел у текущего отсутствует, то новый узел становится его левым узлом;
- если значения нового узла больше текущего, то переходим в правую ветку текущего узла;
- если правый узел у текущего отсутствует, то новый узел становится его правым узлом.

Структуры данных

100, 50, 20, 70, 60, 200, 300, 150, 110, 260, 290, 220



Структуры данных

Пример реализации дерева

// описание узла дерева

```
class Node {  
    int data;  
    Node left;  
    Node right;  
    Node(int x) { data=x; }  
}
```

// описание дерева

```
class Tree {  
    Node root;
```

Структуры данных

```
boolean add(Node elem) {           // добавить элемент
    if (root == null)
        root = elem;
    else
        return ins(root, elem);
    return true;
}
private boolean ins(Node cur, Node el) { // вставить элемент
    while(true) {
        if (cur.data > el.data) {
            if (cur.left == null) { cur.left = el; break; }
            else cur = cur.left;
        }
        else if (cur.data < el.data) {
            if (cur.right == null) { cur.right = el; break; }
            else cur = cur.right;
        } else return false;
    }
    return true; }
}
```

Структуры данных

```
public static void main(String[] args) {  
    Scanner sc = new Scanner(System.in);  
    Tree tr = new Tree();  
    while(true) {  
        System.out.print("\nAdd the node? Yes - press key 'y', No - press key 'n' ");  
        if (sc.next().equals("y")) {  
            System.out.print("Enter value -> ");  
            tr.add(new Node(sc.nextInt()));  
        }else {  
            System.out.print("\nTree the create!!!!\n");  
            break;  
        }  
    }  
    tr.prin();  
}
```


Структуры данных

Add the node? Yes - press key 'y', No - press key 'n' y

Enter value -> 100

Add the node? Yes - press key 'y', No - press key 'n' y

Enter value -> 70

Add the node? Yes - press key 'y', No - press key 'n' y

Enter value -> 90

Add the node? Yes - press key 'y', No - press key 'n' y

Enter value -> 30

Add the node? Yes - press key 'y', No - press key 'n' y

Enter value -> 200

Add the node? Yes - press key 'y', No - press key 'n' y

Enter value -> 150

Add the node? Yes - press key 'y', No - press key 'n' n

Tree the create!!!!

Tree:

30 70 90 100 150 200