

Алгоритмы и контейнеры данных

Электронная презентация

Захаров Алексей Сергеевич

Кафедра Компьютерной Фотоники
Факультет Фотоники и
Оптоинформатики
СПбГУ ИТМО

Введение

- В рамках курса будут изучаться
 - Алгоритмы сортировки и поиска
 - Контейнеры данных
- Необходимо освоить
 - Реализацию алгоритмов и контейнеров
 - Рациональный выбор и использование стандартных алгоритмов и контейнеров

Введение

- Курс разрабатывался, исходя из использования языка программирования C++
- Допускается использование других объектно-ориентированных языков для выполнения заданий

Введение

- Стандартная схема сдачи курса
 - два задания на разработку алгоритмов (8+15)
 - одно задание на разработку контейнера данных (15)
 - одно задание на разработку программного обеспечения с использованием стандартных алгоритмов и контейнеров данных (20-32)
 - два теста (5+5)
 - Личностные качества (5+5)
 - Экзамен (20)

Введение

- Альтернативная схема сдачи курса
 - Есть специальное задание для одного-двоих разработчиков. Желательно знание языка C#.

Тема 1.1. Вычислительная
сложность алгоритмов.
Алгоритмы сортировки и поиска

Лекция 1. Понятие вычислительной сложности алгоритма

- Время выполнения программой той или иной вычислительно сложной задачи является ключевой характеристикой программы. Следует выбирать алгоритм так, чтобы минимизировать время работы программы.
- Точно оценить время работы программы при разработке невозможно (неизвестны исходные данные, характеристики компьютера и многое другое)

Время работы программы

- Время работы программы зависит от
 - Алгоритма
 - Числа обрабатываемых элементов
 - Конкретного набора элементов
 - Характеристик компьютера
 - Особенности реализации алгоритма на языке программирования

Время работы программы

- Рассмотрим несколько программ, выполняемых на одной машине в одинаковых условиях с входными наборами различной длины
- В таблице иллюстрируется зависимость времени работы программы от размера входных данных

Изменение времени работы

Формула \ N	2	100	10000
$10N^2$	40	100000	1000000000
$10N^2+30N$	100	103000	1000300000
$20N^2$	80	200000	2000000000
N^3+5N^2	28	1050000	1000500000000
$3N^3$	24	3000000	3000000000000

Время работы программы

- Можно заметить, что при больших N существенно различие между первыми тремя программами и последними двумя программами.
- Иными словами, существенно различие между программами, работающими за время «порядка N^2 » [или $O(N^2)$] и «порядка N^3 » [или $O(N^3)$].

Утверждение

- Пусть компьютер соответствует принципу адресности фон Неймана (имеет оперативную память, время обращения к каждой ячейке которой по ее целочисленному адресу одинаково)
- Пусть компьютер поддерживает принцип программного управления и принцип последовательного исполнения команд (допустима конвейеризация или параллельное исполнение на фиксированном числе процессоров)

Утверждение

- Пусть компьютер имеет примерно соответствующий общепринятому набор команд (т.е. в нем нет готовых команд сортировки, например).

Утверждение

- Тогда для большинства задач порядок роста времени работы программы в зависимости от числа элементов определяется алгоритмом.
- Коэффициенты в формуле зависимости времени работы программы определяются деталями реализации, характеристиками компьютера и т.д.

Выводы

- При разработке программы невозможно точно определить время ее работы в будущем.
- Для практических нужд, как правило, достаточно знание порядка роста времени работы программы в зависимости от числа элементов.

Выводы

- Исследование вычислительной сложности алгоритма возможно без знания деталей его реализации на конкретном языке программирования на конкретном компьютере.
 - Для большинства алгоритмов при выполнении базовых предположений о компьютере порядок роста времени работы в зависимости от числа элементов не зависит от реализации

Асимптотическое поведение функции

Говорят, что

$$f(n) \in O(g(n))$$

если

$$\exists c_1 \geq 0, c_2 \geq 0, n_0 \geq 0$$

$$\forall n \geq n_0$$

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$$

Асимптотическое поведение функции

Говорят, что

$$f(n) \in o(g(n))$$

если

$$\forall c_1 > 0$$

$$\exists n_0 \geq 0$$

$$\forall n \geq n_0$$

$$0 \leq f(n) < c_1 g(n)$$

Асимптотическое поведение функции

- Верно, что

$$f(n) \in O(g(n)) \Leftrightarrow g(n) \in O(f(n))$$

$$f(n) \in O(g(n)), h(n) \in O(g(n)) \Rightarrow f(n) \in O(h(n))$$

$$kg(n) + o(g(n)) \in O(g(n))$$

Асимптотическое поведение функции. Примеры

$$3n^2 + 10n = O(n^2)$$

$$500n = o(n^2)$$

$$10n \lg(n) + 50n = O(n \lg(n))$$

$$1000n = o(n \lg(n))$$

$$2^n + 10n^3 = O(2^n)$$

Асимптотическое поведение функции

- Для исследования алгоритма работы достаточно выяснить асимптотическое поведение функции, задающей зависимость времени работы от количества элементов
- Как правило, эта характеристика определяется алгоритмом, а не реализацией программы

Асимптотическое поведение функции.

- Поскольку

$$kg(n) + o(g(n)) \in O(g(n)),$$

мы можем пренебрегать постоянными коэффициентами и меньшими по порядку добавками $[o(g(n))]$ при оценивании времени работы функции

Пример

```
max = 0;  
for ( i = 0 ; i < n ; i++ )  
    if ( max < A[i] )  
        max = A[i];
```

Пример. Команды процессора

SET R1,0	C ₁		
LOAD R2, <адрес n>		C ₂	
LOAD R3, <адрес A>		C ₂	
SET R4, 0;	C ₁		
start: CMP R4,R2		C ₃	
JZ finish	C ₄		
LOAD R5, [R3]		C ₂	
CMP R1, R5		C ₃	
JZ next	C ₄		
SET R1, R5		C ₁	
next:ADD R4,R4,1			C ₅
ADD R3, R3, 4 [sizeof(unsigned int)]			C ₅
JMP start	C ₆		
finish: SAVE R4, <адрес max>			C ₇

Пример:

Время работы программы (k – количество раз, когда условие выполнено, $0 \leq k \leq n$)

$$T = 2c_1 + 2c_2 + n(2c_3 + 2c_4 + c_2 + 2c_5 + c_6) + kc_1 + c_7$$

$$2c_1 + 2c_2 + c_7 + n(2c_3 + 2c_4 + c_2 + 2c_5 + c_6) \leq T$$

$$T \leq 2c_1 + 2c_2 + c_7 + n(2c_3 + 2c_4 + c_2 + c_1 + 2c_5 + c_6)$$

$$T = O(n)$$

Пример

```
max = 0;  
for ( i = 0 ; i < n ; i++ )  
    if ( max < A[i] )  
        max = A[i];
```

При взгляде на код интуитивно понятно, что сложность алгоритма $T=O(n)$

Мы это доказали строго

Вычислительная сложность алгоритма

- Часто время работы алгоритма зависит не только от размера входных данных, но и от их значений.
- В этом случае можно говорить о времени работы:
 - Для наилучших входных данных
 - Для средних входных данных (матожидание времени работы)
 - Для наихудших входных данных

Вычислительная сложность алгоритма

- Часто асимптотическая сложность алгоритма для средних и наихудших входных данных совпадает
- Когда я говорю о вычислительной сложности алгоритма, не уточняя детали – я имею в виду, что для этого алгоритма асимптотическая сложность совпадает в среднем и наихудшем случае

Вычислительная сложность алгоритма

- Существуют алгоритмы (например, QuickSort), вычислительная сложность которых отличается в среднем $O(n \lg(n))$ и наихудшем $O(n^2)$ случаях
- Используя такие алгоритмы, подумайте, не оказывается ли наихудший случай самым распространенным в вашей задаче

Вычислительная сложность алгоритма

- Вычислительная сложность алгоритма в наилучшем случае обсуждается реже
- Подумайте, не можете ли Вы организовать наилучший случай в своей задаче.

Выводы

- Порядок роста времени выполнения программы, как правило, определяется алгоритмом
- Ключевая характеристика алгоритма – порядок роста (асимптотическая сложность)
- Асимптотическую сложность алгоритма часто можно оценить интуитивно

Лекция 2. Понятие сортировки и поиска. Обзор основных алгоритмов.

- Линейный поиск в массиве
- Бинарный поиск в массиве
- Сортировка прямым выбором
- Другие квадратичные сортировки
- Сортировка Merge Sort
- Другие $n \lg(n)$ сортировки

Методы поиска

- Линейный поиск
- Бинарный поиск
- Другие методы

Линейный поиск в массиве

- Пусть есть массив A длины n
- Необходимо найти элемент, равный a .
- Мы можем просто перебрать все элементы массива, сравнивая их с a

Линейный поиск в массиве

```
int result = -1;
int i = 0;
while ( i < n && result < 0 )
{
    if ( A[ i ] == a )
        result = i;
    i++;
}
```

Линейный поиск в массиве

- Легко показать, что время работы алгоритма в наихудшем и среднем случае – $O(n)$.
- Действительно, наихудший случай – когда элемент не найден, трудоемкость равна c_1n+c_2
- Если элемент найден, трудоемкость в среднем $c_1(n/2)+c_3$

Бинарный поиск в массиве

- В общем случае реализовать поиск с трудоемкостью, меньшей $O(n)$, невозможно
- Если мы не делаем предположений о хранении данных в массиве – то любой элемент может оказаться нужным, и проверять необходимо все
- Предположим, массив был отсортирован. Тогда ситуация меняется

Поиск в отсортированном массиве

18

1	3	4	8	9	10	11	14	16	17	18	19	23	25	27
---	---	---	---	---	----	----	----	----	----	----	----	----	----	----

18

1	3	4	8	9	10	11	14	16	17	18	19	23	25	27
---	---	---	---	---	----	----	----	----	----	----	----	----	----	----

18

1	3	4	8	9	10	11	14	16	17	18	19	23	25	27
---	---	---	---	---	----	----	----	----	----	----	----	----	----	----

18

1	3	4	8	9	10	11	14	16	17	18	19	23	25	27
---	---	---	---	---	----	----	----	----	----	----	----	----	----	----

Бинарный поиск

- Количество сравнений – $\log_2 N$
- Неудобство хранения данных в отсортированном массиве – дорогая вставка элемента (потребуется переместить в среднем $N/2$ элементов)
- Решение этой проблемы будет рассмотрено в лекции 3, посвященной контейнерам

Поиск

- Если мы хотим еще более быстрого поиска – мы должны наложить еще более жесткие ограничения на механизм хранения данных.
- Подробнее вопрос будет рассмотрен в лекции 4, посвященной хэшированию.

Поиск минимального элемента

- Задача решается за время, равное $O(n)$

```
min = 0;  
for ( i = 0 ; i < n ; i++ )  
    if ( A[i] < min )  
        min = A[i];
```

Методы сортировки

- Сортировка за $O(n^2)$
- Сортировка за $O(n \lg(n))$

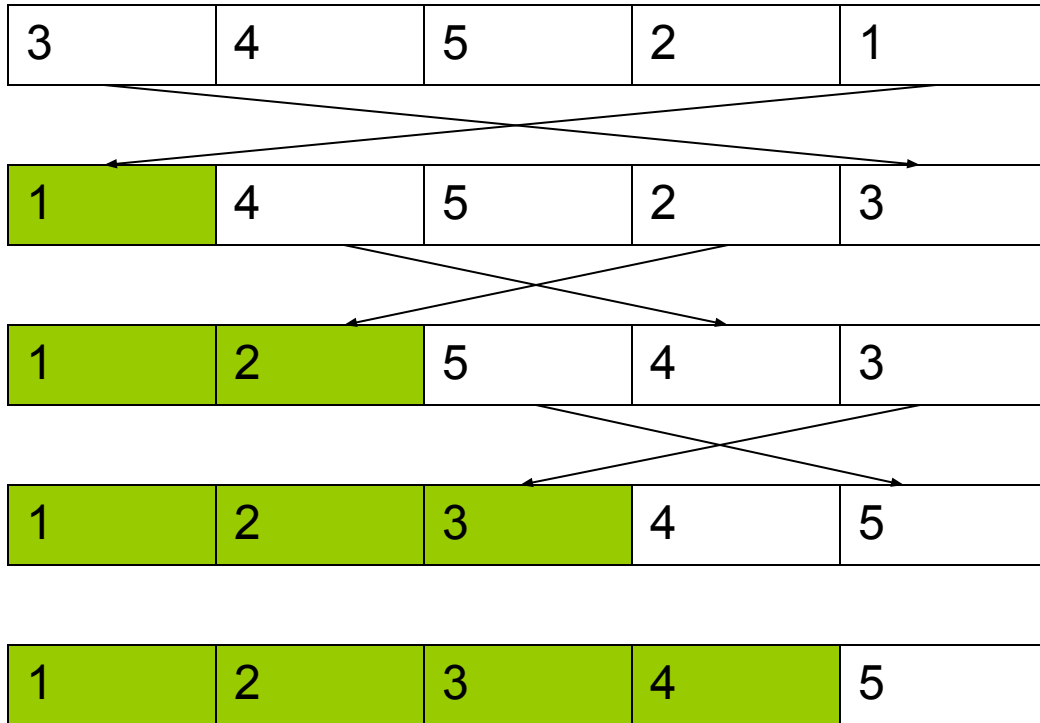
Сортировка прямым выбором

- На первом шаге выбирается минимальный элемент и ставится первым
- После этого мы решаем ту же задачу для $N-1$ элемента – начиная со второго
- Так пока число сортируемых элементов не станет 1

Пример

- Демонстрационная программа
SortStraightSel

Пример работы



Сортировка прямым выбором

- Мы просматриваем на первом шаге N элементов, на втором – $N-1$, и так далее.
- Всего – $N + N-1 + \dots + 1 = (N^2 + N)/2$
- Время работы алгоритма - $O(N^2)$

Сортировка пузырьком

- На каждом шаге перебираются все пары соседних элементов, и если меньший элемент стоит позже – элементы меняются местами
- Таким образом, малые значения «всплывают» в начало массива, а большие «опускаются» в конец
- Нужно выполнить $N-1$ шаг, чтобы массив стал отсортированным

Пример

3	4	5	2	1
---	---	---	---	---

3	4	5	2	1
---	---	---	---	---

3	4	5	2	1
---	---	---	---	---

3	4	2	5	1
---	---	---	---	---

3	4	2	1	5
---	---	---	---	---

Пример

3	4	2	1	5
---	---	---	---	---

3	4	2	1	5
---	---	---	---	---

3	2	4	1	5
---	---	---	---	---

3	2	1	4	5
---	---	---	---	---

3	2	1	4	5
---	---	---	---	---

Можно уже не
сравнивать

Пример

3	2	1	4	5
---	---	---	---	---

2	3	1	4	5
---	---	---	---	---

2	1	3	4	5
---	---	---	---	---

2	1	3	4	5
---	---	---	---	---

2	1	3	4	5
---	---	---	---	---

Можно не
сравнивать

Пример

2	1	3	4	5
---	---	---	---	---

1	2	3	4	5
---	---	---	---	---

1	2	3	4	5
---	---	---	---	---

1	2	3	4	5
---	---	---	---	---

1	2	3	4	5
---	---	---	---	---

Можно уже не
сравнивать

Сортировка пузырьком

- Необходимо $N-1$ шагов.
- На каждом шаге – $N-1$ сравнение (и, при необходимости, перестановка).
- Итого – $(N-1)^2$, т.е. $O(N^2)$ шагов
- Если не делать лишних сравнений – $(N^2 - N)/2$

Быстрые алгоритмы сортировки

Алгоритм сортировки MergeSort

- Представим себе, что левая и правая половина массива отсортированы.
- Тогда отсортировать весь массив можно за N шагов. Как?

Merge Sort

--	--	--	--	--	--	--	--

1	3	6	8	2	4	5	7
---	---	---	---	---	---	---	---

1							
---	--	--	--	--	--	--	--

1	3	6	8	2	4	5	7
---	---	---	---	---	---	---	---

1	2						
---	---	--	--	--	--	--	--

1	3	6	8	2	4	5	7
---	---	---	---	---	---	---	---

- На каждом шаге сравниваются два элемента - один из первой половины, один из второй.
- Меньший из них записывается в результирующий массив

Merge Sort

1	2						
---	---	--	--	--	--	--	--

1	3	6	8	2	4	5	7
---	----------	---	---	---	----------	---	---

1	2	3					
---	---	---	--	--	--	--	--

1	3	6	8	2	4	5	7
---	---	----------	---	---	----------	---	---

1	2	3	4				
---	---	---	---	--	--	--	--

1	3	6	8	2	4	5	7
---	---	----------	---	---	---	----------	---

1	2	3	4	5			
---	---	---	---	---	--	--	--

1	3	6	8	2	4	5	7
---	---	----------	---	---	---	---	----------

Merge Sort

1	2	3	4	5			
---	---	---	---	---	--	--	--

1	3	6	8	2	4	5	7
---	---	----------	---	---	---	---	----------

1	2	3	4	5	6		
---	---	---	---	---	---	--	--

1	3	6	8	2	4	5	7
---	---	---	----------	---	---	---	----------

1	2	3	4	5	6	7	
---	---	---	---	---	---	---	--

1	3	6	8	2	4	5	7
---	---	---	----------	---	---	---	---

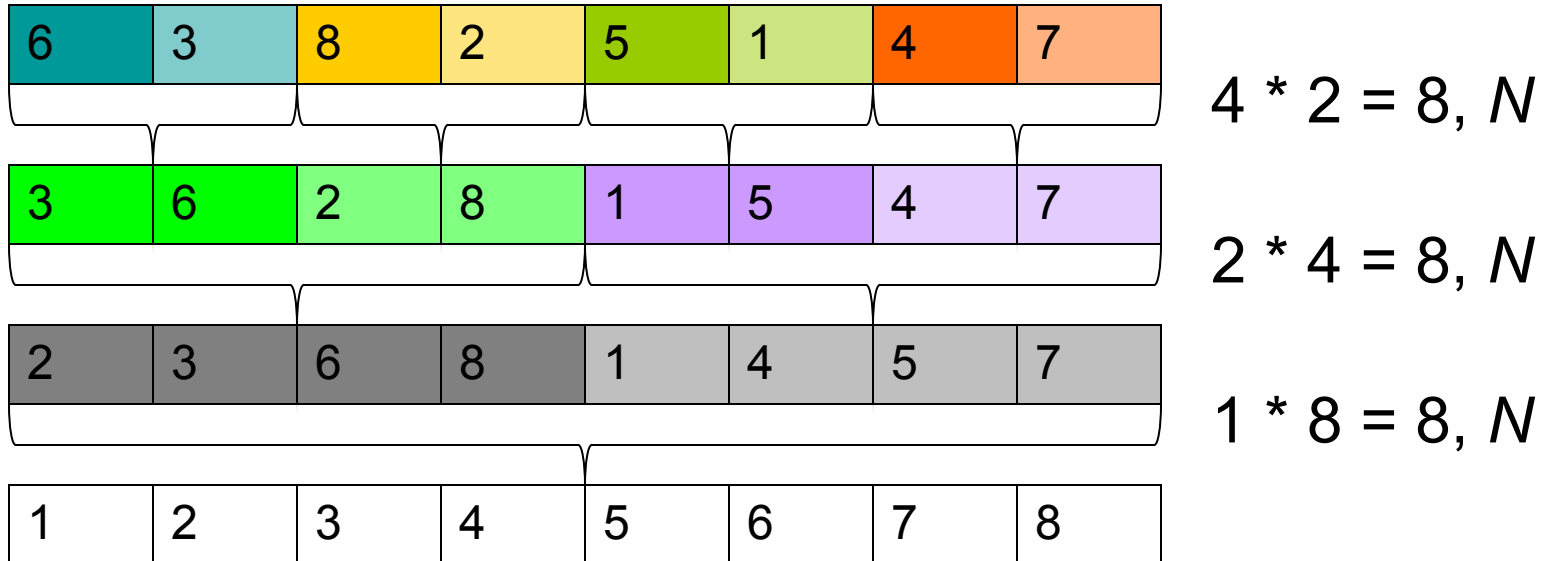
1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---

1	3	6	8	2	4	5	7
---	---	---	---	---	---	---	---

Merge Sort

- Как же сделать половинки массива отсортированными?
 - В массиве из двух элементов половинки отсортированы всегда
 - Отсортировав все фрагменты массива из двух элементов каждый, можно сортировать фрагменты из четырех – и так до конца
 - Если длина массива – не 2^n , ничего страшного – просто один из двух массивов будет короче

Merge Sort. Неотсортированный массив



Ступенек – $\log_2 N$, общая трудоемкость – $N \log_2 N$

MergeSort

- Алгоритм MergeSort позволяет нам решить задачу сортировки массива за время, пропорциональное $M \log_2 N$
- Мы знаем, что $\log_2 N = \log_a N * \log_2 a = K \log_a N$
- Следовательно, если время работы алгоритма – $O(\log_2 N)$, то оно равно и $O(\log_a N)$
- Поэтому часто говорят просто $O(M \log N)$, не уточняя основание логарифма

Пирамидальная сортировка

- Основана на помещении значений в пирамиду и извлечении их из пирамиды

QuickSort

3	7	9	2	1	6	5
---	---	---	---	---	---	---

3

1	2
---	---

5	6	7	9
---	---	---	---

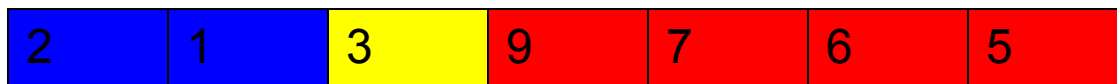
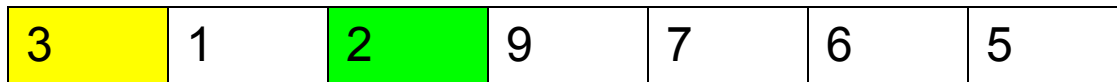
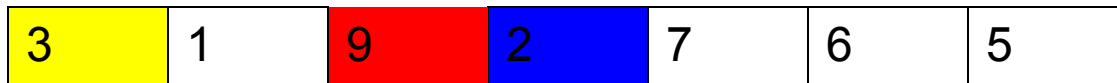
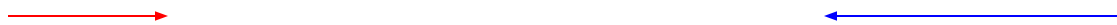
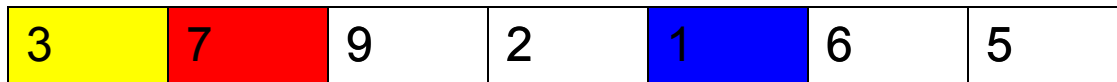
Мы взяли число и разделили массив на две части – значения меньше данного и больше данного. После этого мы можем продолжить сортировки половинок массива

В среднем и лучшем случае сортировка занимает время $O(N \lg N)$ – лучший случай это деление массива пополам на каждом шаге

В худшем случае – $O(N^2)$

QuickSort

- Как выполнить QuickSort без использования дополнительной памяти?



CombSort

- В сортировке пузырьком мы сравниваем соседние элементы и меняем их местами
- Эффективнее на первых шагах сравнивать более удаленные друг от друга элементы
- Постепенно снижаем расстояние между сравниваемыми элементами
- На последнем шаге повторим пузырек, но проходов потребуется немного

CombSort

- Начальный шаг – длина массива, деленная на 1.3
- Уменьшение шага – в 1.3 раза

CombSort

3	7	9	2	1	6	5
---	---	---	---	---	---	---

Шаг 5 (1 проход)

3	5	9	2	1	6	7
---	---	---	---	---	---	---

Шаг 3 (1 проход)

2	1	6	3	5	9	7
---	---	---	---	---	---	---

Шаг 2 (2 прохода)

2	1	5	3	6	9	7
---	---	---	---	---	---	---

Шаг 1 (2 прохода)

1	2	3	5	6	7	9
---	---	---	---	---	---	---

IntroSort

- Сочетание пирамидальной и быстрой сортировки
- Быстрая сортировка лучше в среднем случае, пирамидальная – в наихудшем
- При достижении предельной глубины быстрой сортировки переходим на пирамидальную

Методы сортировки за $O(N)$

- Сортировка подсчетом
- Цифровая сортировка
- Карманная сортировка

Сортировка подсчетом

- Предположим, в массиве лежат значения, равные 0, 1 и 2
- Как выполнить его сортировку за время $O(N)$?

0	2	2	0	1	1	0	2
---	---	---	---	---	---	---	---

Сортировка подсчетом

0	2	2	0	1	1	0	2
---	---	---	---	---	---	---	---

Этап 1 – подсчитываем число 0, единиц и двоек

0	3
1	2
2	3

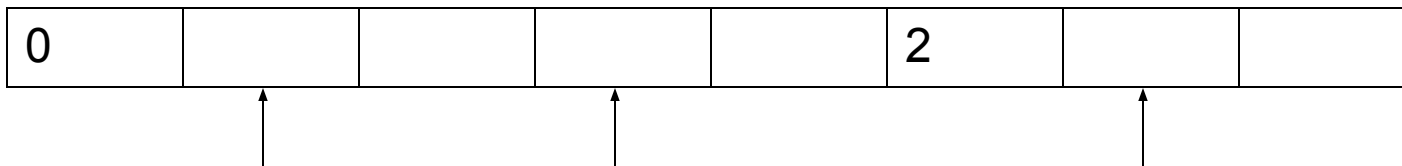
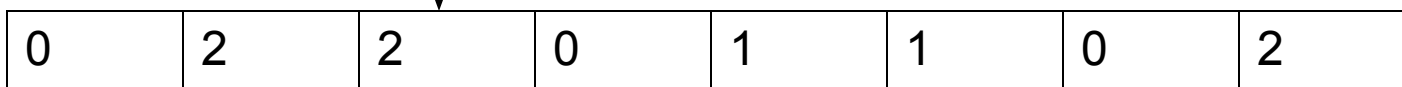
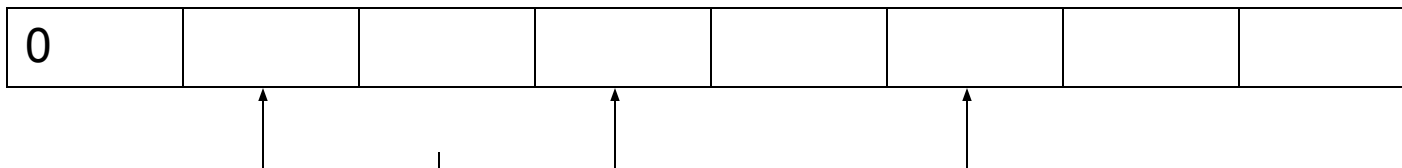
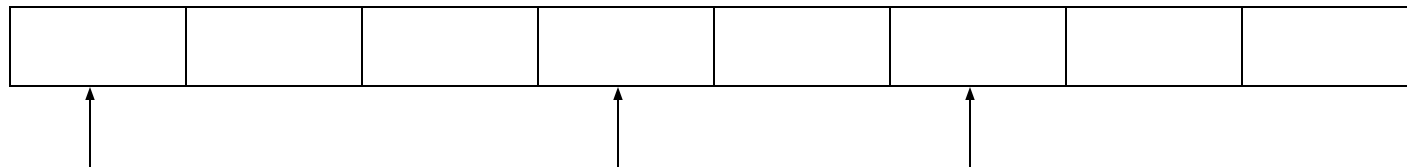
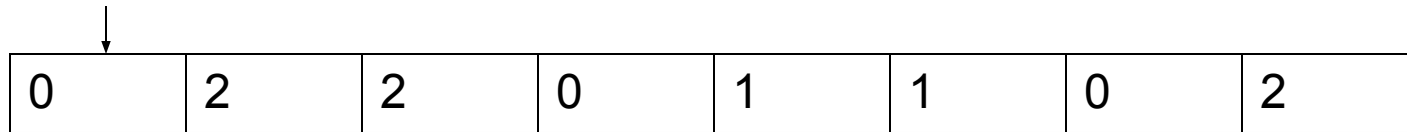
Сортировка подсчетом

0	2	2	0	1	1	0	2
---	---	---	---	---	---	---	---

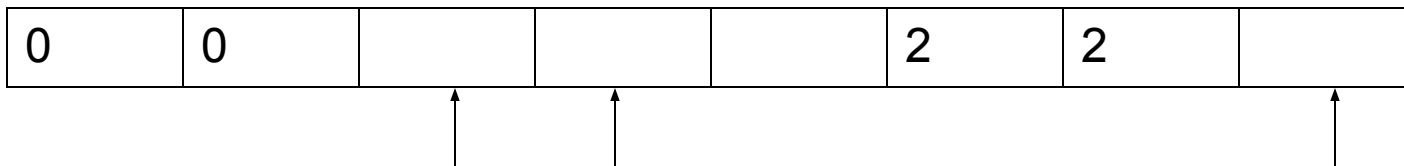
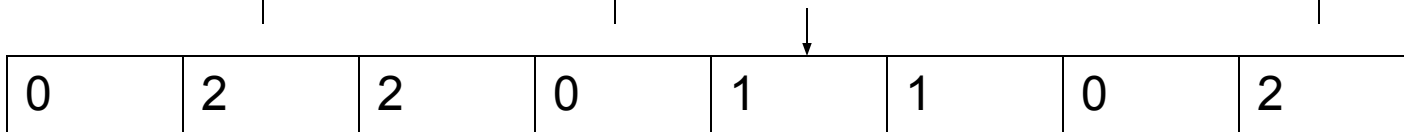
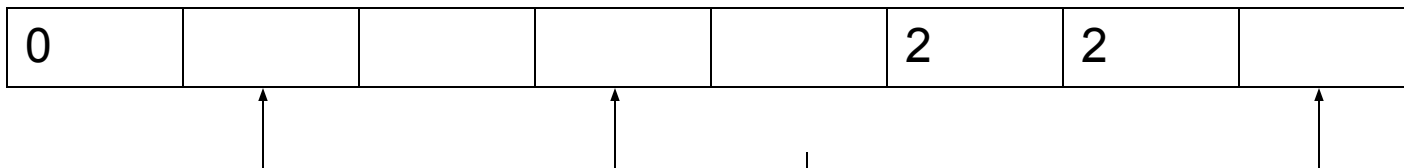
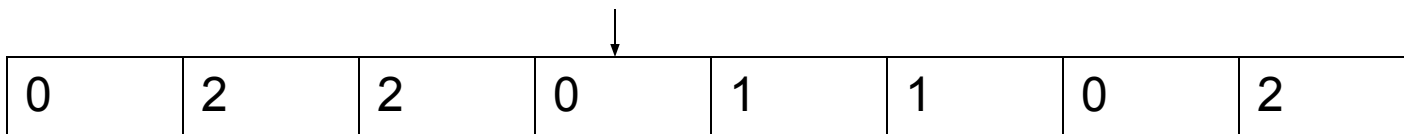
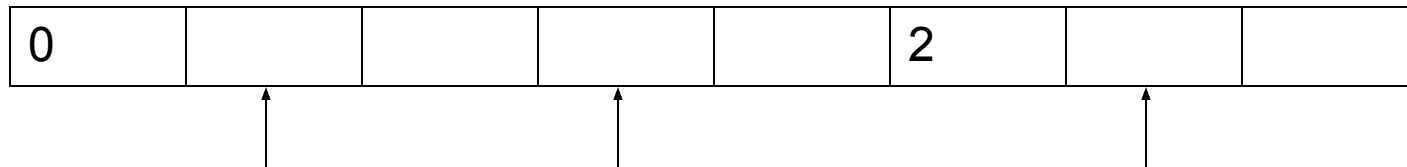
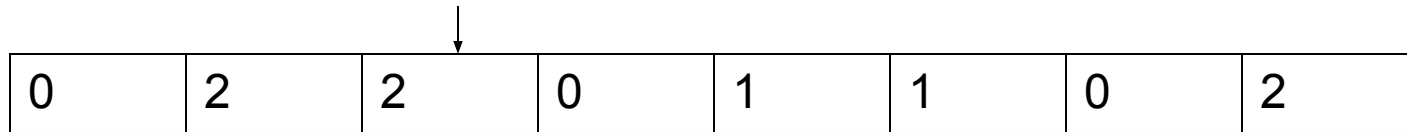
Этап 2 – Определяем позиции, на которых должны лежать 0, 1 и 2

Значение	Число	Min	Max
0	3	0	2
1	2	3	4
2	3	5	7

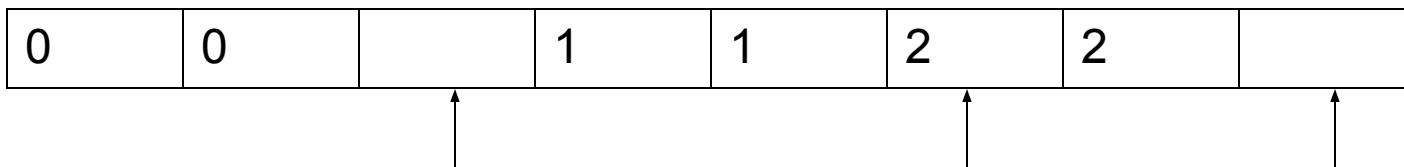
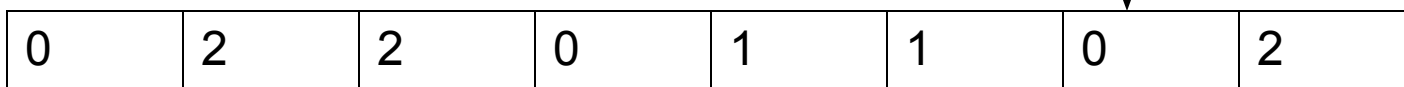
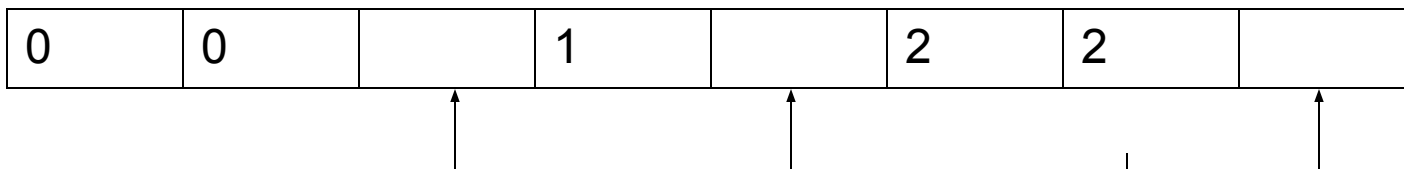
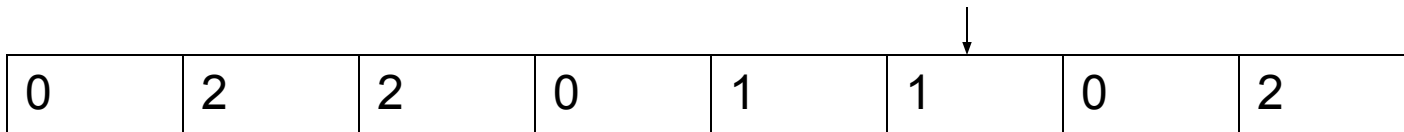
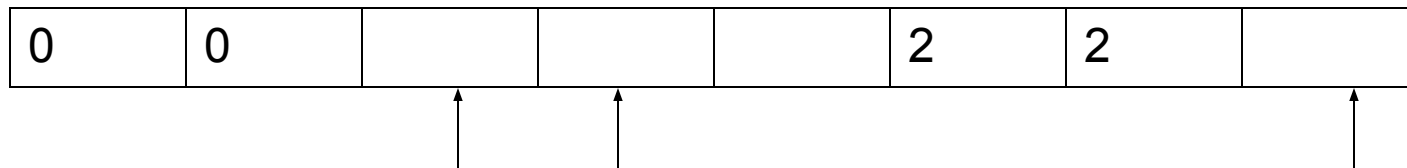
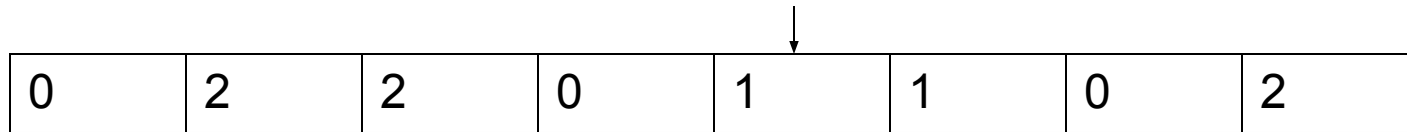
Сортировка подсчетом



Сортировка подсчетом



Сортировка подсчетом



Сортировка подсчетом

- Работает за время $O(N+K)$, где N – число значений в массиве, K – число возможных значений
- Требуется дополнительной памяти в объеме $O(N+K)$

Сортировка подсчетом

Фамилия	Имя	Курс
Алексеев	Иван	3
Борисов	Кирилл	2
Васильев	Андрей	3
Иванова	Ольга	1
Петрова	Дарья	3
Сидоров	Артем	2
Широков	Владимир	1
Яковлев	Алексей	2

Фамилия	Имя	Курс
Иванова	Ольга	1
Широков	Владимир	1
Борисов	Кирилл	2
Сидоров	Артем	2
Яковлев	Алексей	2
Алексеев	Иван	3
Васильев	Андрей	3
Петрова	Дарья	3

Сортировка подсчетом

- Порядок студентов был алфавитным
- Мы отсортировали список по номеру курса. Порядок студентов внутри курса остался алфавитным

Цифровая сортировка

- Для массивов с большим диапазоном значений сортировка подсчетом не годится
- Учитывая сохранение порядка элементов с равными значениями в сортировке подсчетом, можно ее использовать и в этом случае

Цифровая сортировка

532	2	170	7	913	9	170
718	8	191	9	718	7	191
191	1	532	3	532	5	265
265	5	743	4	743	7	489
743	3	913	1	265	2	532
489	9	265	6	170	1	718
170	0	718	1	489	4	743
913	3	489	8	191	1	913

- Последовательно сортируем по цифрам, начиная с последней.
- Трудоемкость $O(R*(N+K))$, где R – число цифр, K – число значений цифры, N – число значений в массиве. Дополнительная память - $O(N+K)$

Карманная сортировка

- Пусть есть массив N вещественных значений от 0 до 1.
- Создадим N списков. В список K будем помещать значения из диапазона $[K/N , (K+1)/N)$
- Любым методом отсортируем списки (они будут очень короткими)
- Объединим списки в результирующий массив

Другие алгоритмы сортировки

- Быстрая сортировка (Quick Sort)
- Сортировка Шелла
- Сортировка Шейкером
- Сортировка подсчетом
- Цифровая сортировка (по младшему разряду, потом по старшему и т.д.)
- Пирамидальная сортировка (Heap Sort)

Другие алгоритмы сортировки

- Сортировка расческой (Comb Sort)
- Плавная сортировка (Smooth Sort)
- Блочная сортировка
- Patience sorting
- Introsort

Лабораторная работа №1.

Реализация алгоритмов сортировки и
поиска.

Реализация алгоритмов сортировки и поиска

- Предлагаются индивидуальные варианты заданий, связанные с реализацией алгоритмов
- Предпочтительна реализация алгоритма, сопровождаемая подготовкой доклада об алгоритме
- Доклады целесообразны для алгоритмов повышенной сложности

Варианты заданий

- Реализовать бинарный поиск в массиве
- Реализовать сортировку Шелла
- Реализовать сортировку шейкером
- Реализовать сортировку подсчетом (данные типа char)
- Реализовать сортировку расческой (CombSort)

Варианты заданий

- Реализовать метод IntroSort
- Реализовать цифровую сортировку значений типа `int` по их двоичной записи
- Реализовать цифровую сортировку значений типа `int` по их восьмеричной записи
- Реализовать цифровую сортировку значений типа `int` по их десятичной записи
- Реализовать цифровую сортировку значений типа `int` по их шестнадцатеричной записи

Варианты заданий повышенной сложности

- Реализовать пирамидальную сортировку
- Реализовать плавную сортировку (Smooth Sort)
- Реализовать быструю сортировку (QuickSort)
- Реализовать рандомизированную быструю сортировку

Варианты заданий повышенной сложности

- Реализовать карманную (bucket) сортировку
- Реализовать алфавитную сортировку M строк суммарной длиной N символов за время $O(N)$

Варианты заданий повышенной сложности

- Реализовать поиск i -ой порядковой статистики [i -ого по величине числа] методом RandomizedSelect (за $O(N)$ в среднем).
- Реализовать поиск i -ой порядковой статистики [i -ого по величине числа] за время $O(N)$ в наихудшем случае
- Реализовать поиск наибольшей возрастающей подпоследовательности (Patience Sorting)

Понятие порядковой статистики

2	1	7	4	9	3	0
---	---	---	---	---	---	---

- 1-ая порядковая статистика – 0
- 2-ая – 1
- 3-я – 2
- 4-ая – 3
- 5-ая – 4
- 6-ая – 7
- 7-ая - 9

Тема 1.2. Контейнеры данных. Идея хэширования

Лекция 3. Понятие контейнера данных. Основные типы контейнеров

Понятие контейнера данных

- Контейнер – программный объект, отвечающий за хранение набора однотипных данных (элементов контейнера) и организацию доступа к ним

Контейнеры в языках программирования

- Контейнер может быть
 - Стандартным объектом языка программирования (массивы фиксированной длины в C)
 - Объектом класса, разработанного пользователем
 - Объектом класса стандартной библиотеки

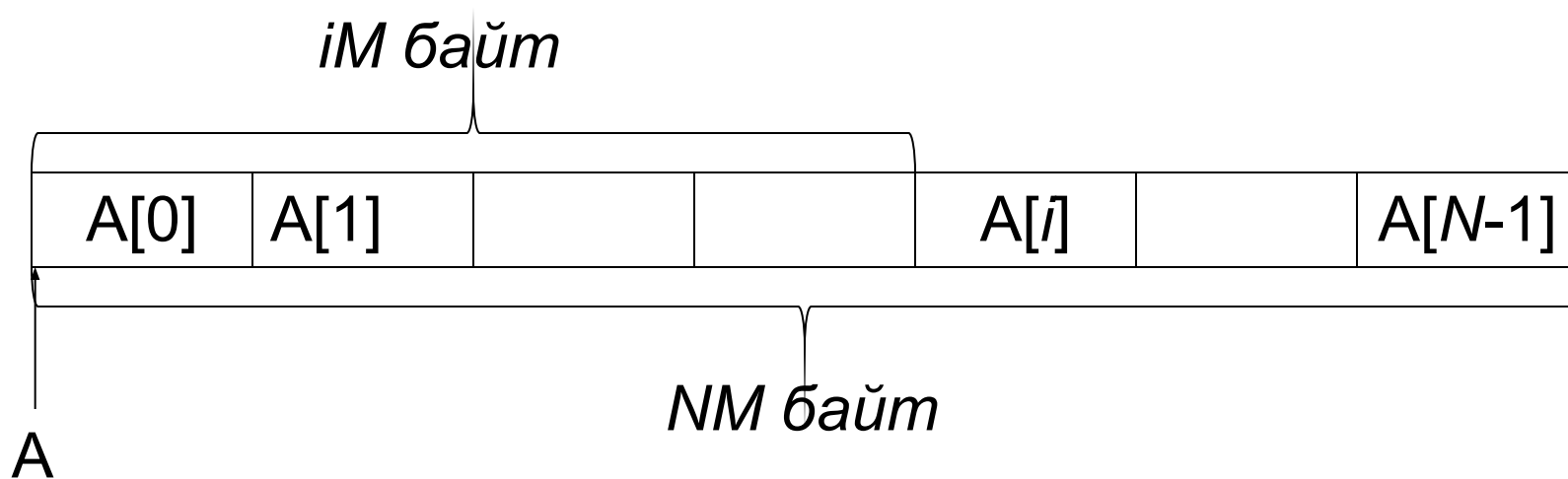
Виды контейнеров

- Массивы
- Списки
- Деревья
- Словари
- Стеки и очереди
- Пирамиды. Очереди с приоритетами

Массивы

- Массивом называется контейнер, в котором элементы лежат в памяти компьютера подряд
- Размер массива из N элементов, каждый из которых занимает M байт – NM .
- Если адрес начала массива в памяти – A , то адрес i -ого элемента – $A+iM$

Массивы



Массивы. Ключевые свойства

- Быстрый поиск элемента по индексу (за $O(1)$)
- На C/C++
 $\&(A[n]) = \&(A) + n$
- Медленная вставка элемента в середину (важно для отсортированного массива) – за $O(N)$
- Проблемы при росте массива сверх заранее запланированного размера

Массив. Рост сверх планового размера



?

Игнорируем



Переезжаем



Массивы

- Запрещая «переезд» массива, мы ограничиваем рост его размера
- Разрешая «переезд», мы лишаем себя права запоминать адреса объектов массива

Пример

```
std::vector< int > array;
```

```
...
```

```
int* ptr = &(array[0]); //Запомнили адрес
```

```
array.push_back( 7 ); //Добавили элемент
```

```
        //Возможен «переезд»
```

```
std::cout << *ptr; //Может упасть.
```

```
    //Может и не упасть.
```

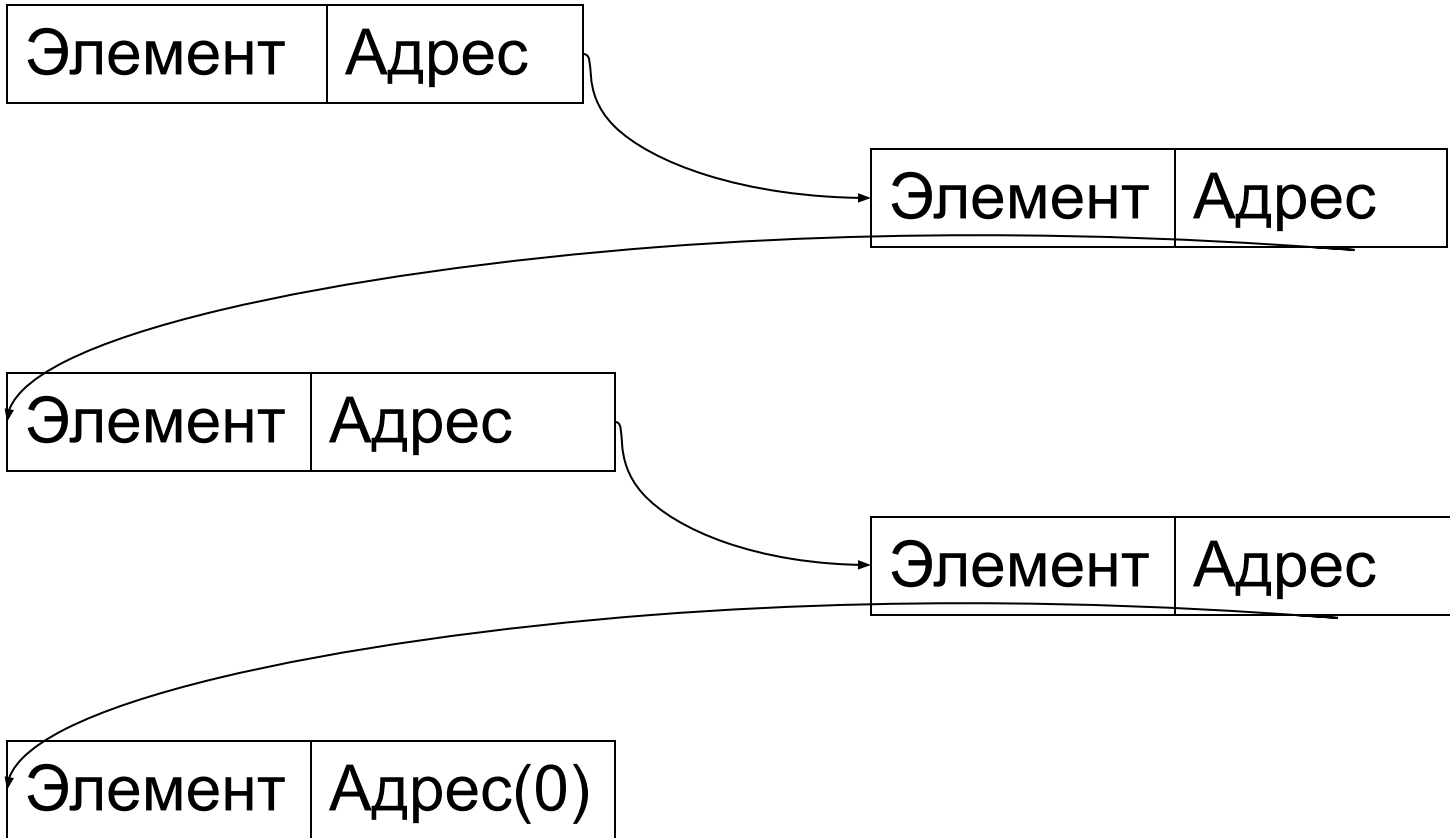
Списки

- Существенным ограничением массива является хранение элементов подряд
- Оно приводит к сложности расширения массива и вставки элемента в середину
- Попробуем от него отказаться

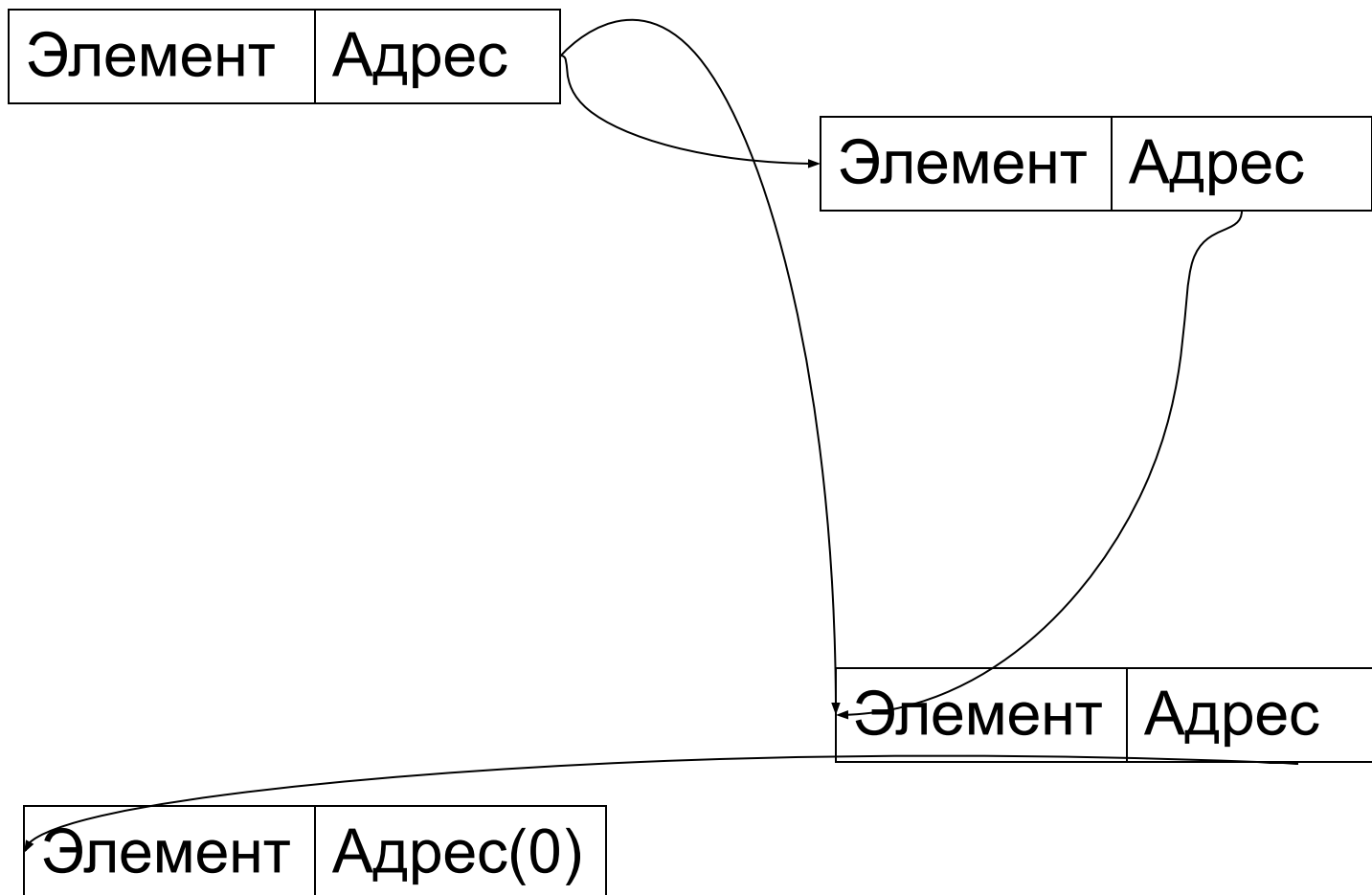
Списки

- Пусть каждый элемент помнит, где лежит следующий (хранит его адрес)
- Тогда достаточно запомнить адрес нулевого элемента, и мы легко найдем любой
- Пример списка приведен на слайде

Списки



Список: вставка элемента



Список: вставка элемента

- Время вставки элемента в середину списка – $O(1)$, т.е. не зависит от размера списка
- Время поиска i -ого элемента по индексу – $O(i)$

Списки

- Недостаток списка: в нем, даже отсортированном, нельзя реализовать бинарный поиск (слишком дорого искать середину списка)

Списки

- Бывают:
 - Однонаправленными (каждый элемент знает следующий)
 - Двухнаправленными (каждый элемент знает следующий и предыдущий)

Деревья

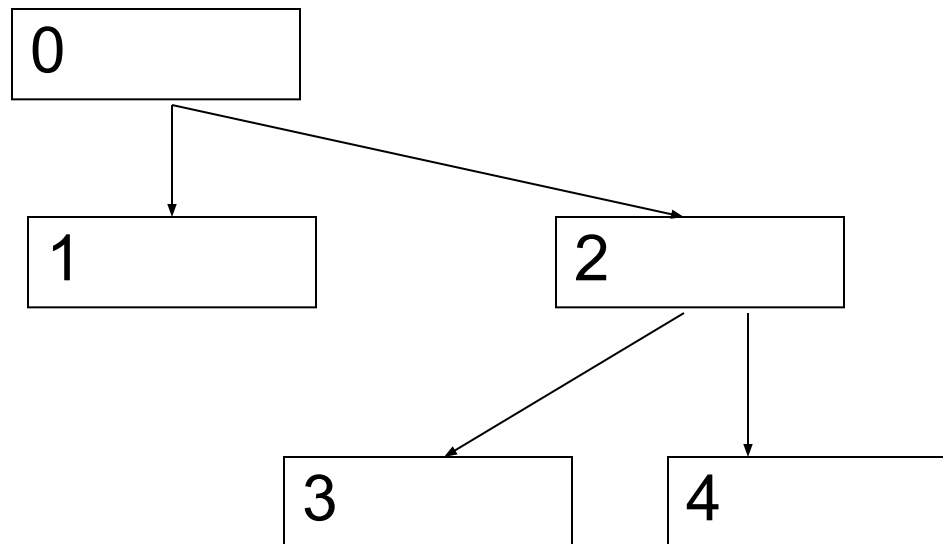
- Отсортированный массив хорош, поскольку позволяет бинарный поиск за время $O(\log N)$
- Добавление нового элемента при этом занимает время $O(N)$
- Мы попробуем с этим справиться
- Начнем с краткого экскурса в теорию графов

Граф

- Рассмотрим множество **A** из N элементов и множество **B**, состоящее из пар элементов множества **A** и не содержащее повторяющихся пар
- A: {0, 1, 2, 3, 4}
- B: {{0, 1}, {0, 2}, {2, 3}, {2, 4}}

Граф

- Это множество называется графом и может быть представлено в виде



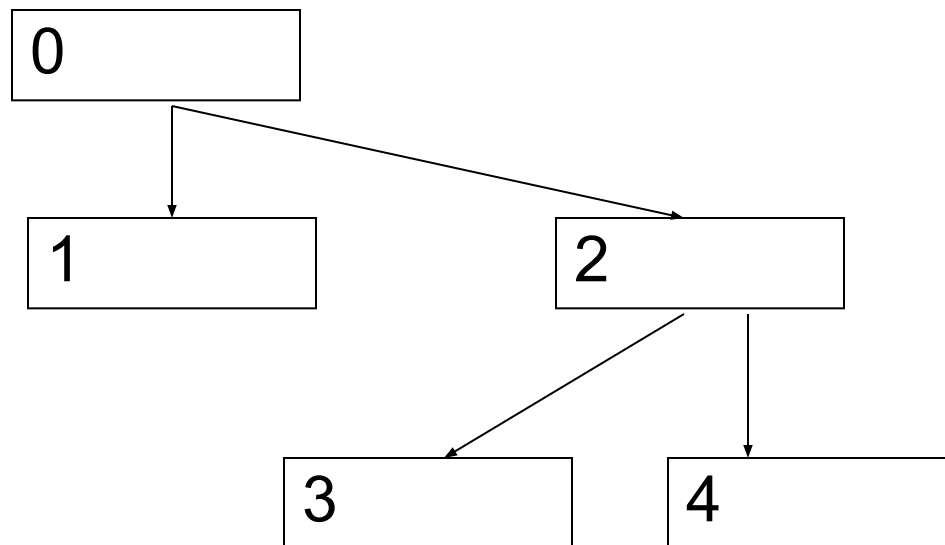
Граф

- Элементы A – узлы графа
- Элементы B – ребра графа. Ребро задается своим начальным и конечным узлом

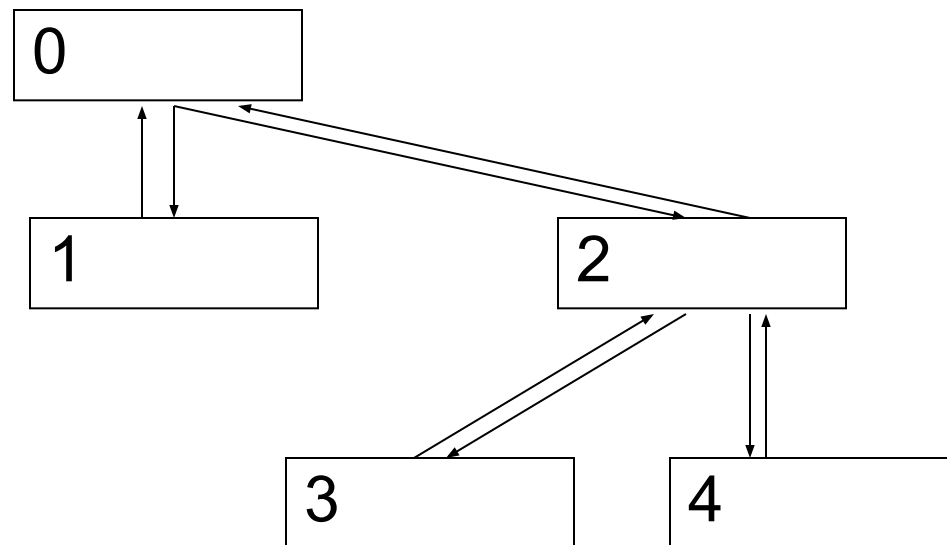
Граф

- Граф называется неориентированным, если для любого ребра $\{a,b\}$, входящего в граф, ребро $\{b,a\}$ тоже входит в граф

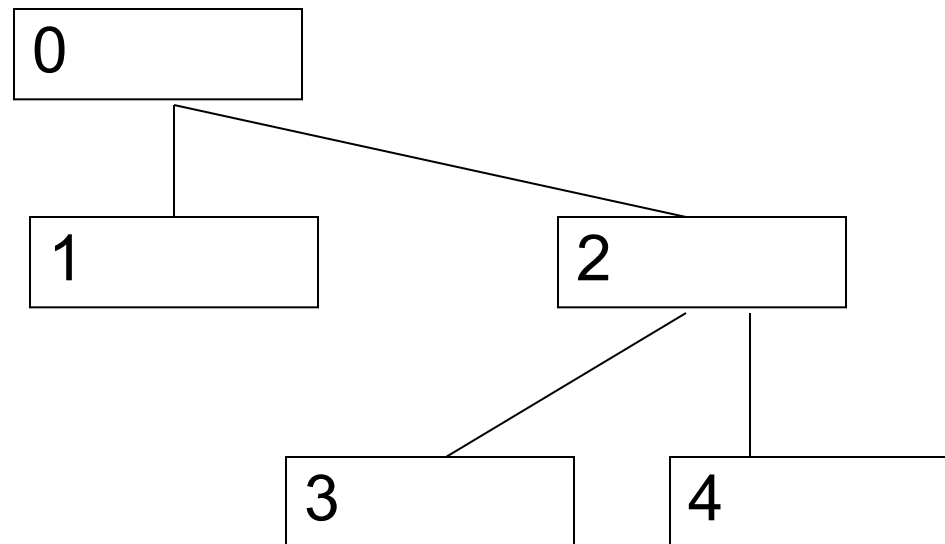
Неориентированный граф?



Неориентированный граф?



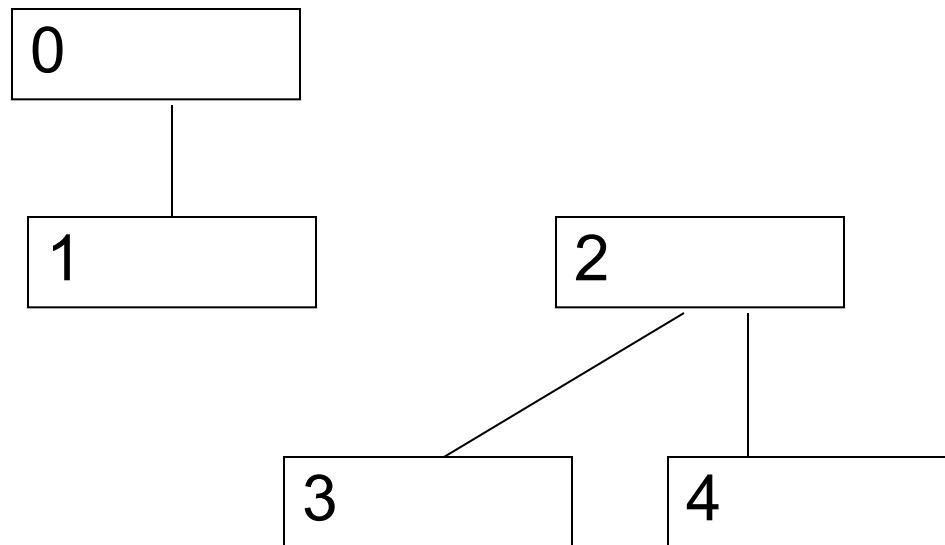
Упрощенное изображение неориентированного графа



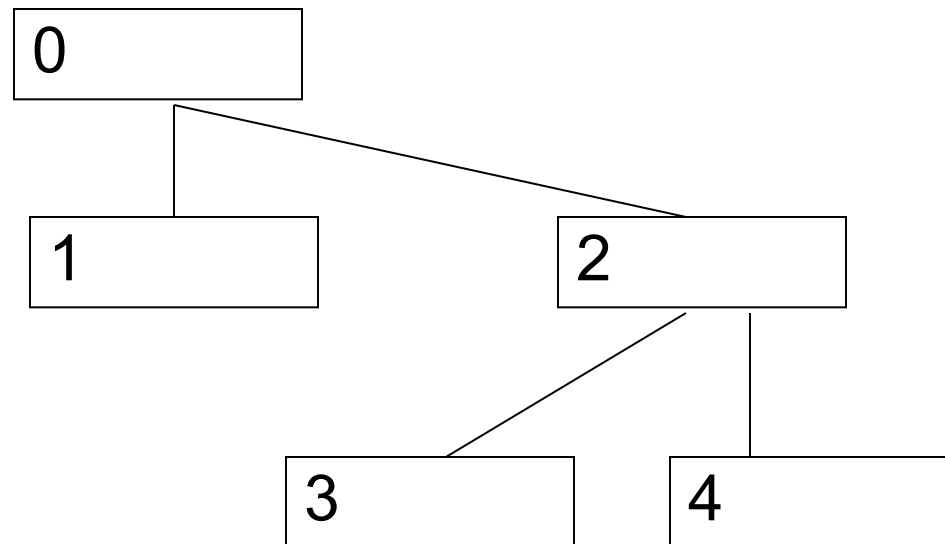
Неориентированные графы

- Неориентированный граф является связным, если из любого узла a можно попасть в любой узел b
- Т.е. для любых a и b существует набор ребер графа $\{a, x_0\}, \{x_0, x_1\}, \dots, \{x_{n-1}, x_n\}, \{x_n, b\}$

Связный граф?



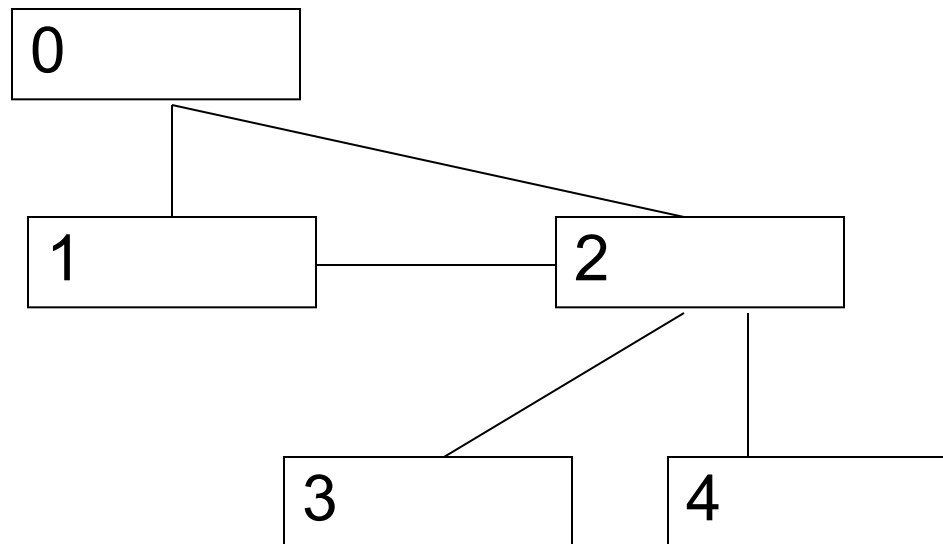
Связный граф?



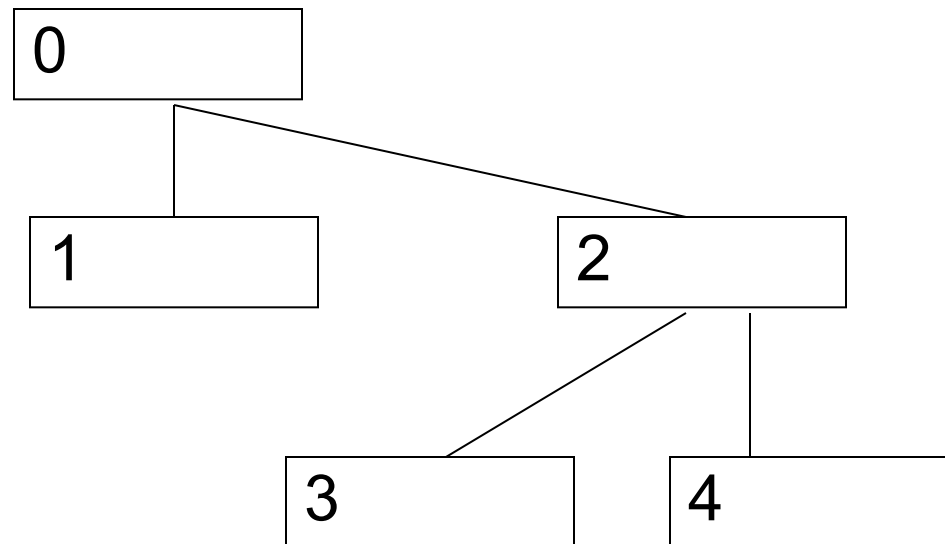
Неориентированные графы

- Неориентированный граф является ациклическим, если в нем не существует маршрутов без повторения ребер, которые начинаются и заканчиваются в одной точке

Ациклический граф?



Ациклический граф?



Деревья

- Деревом называется связный ациклический неориентированный граф
- Если ациклический неориентированный граф – не связный, то это лес (совокупность нескольких деревьев – компонент связности)

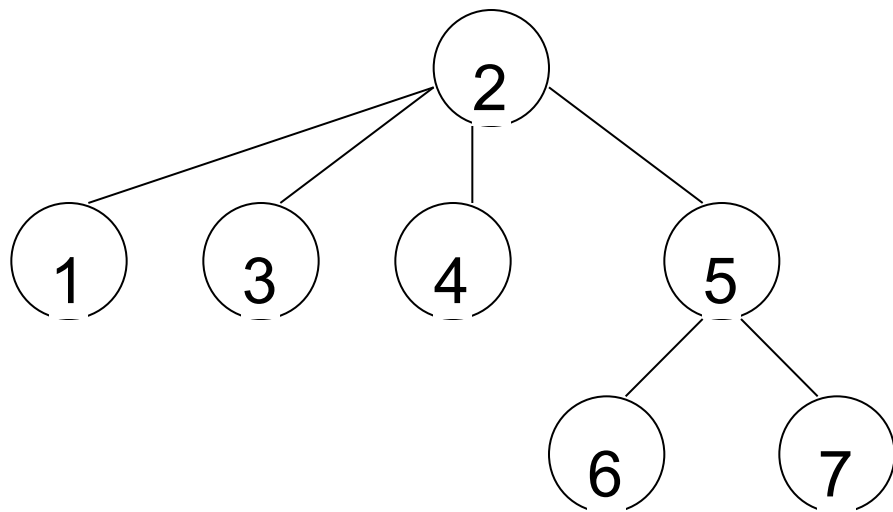
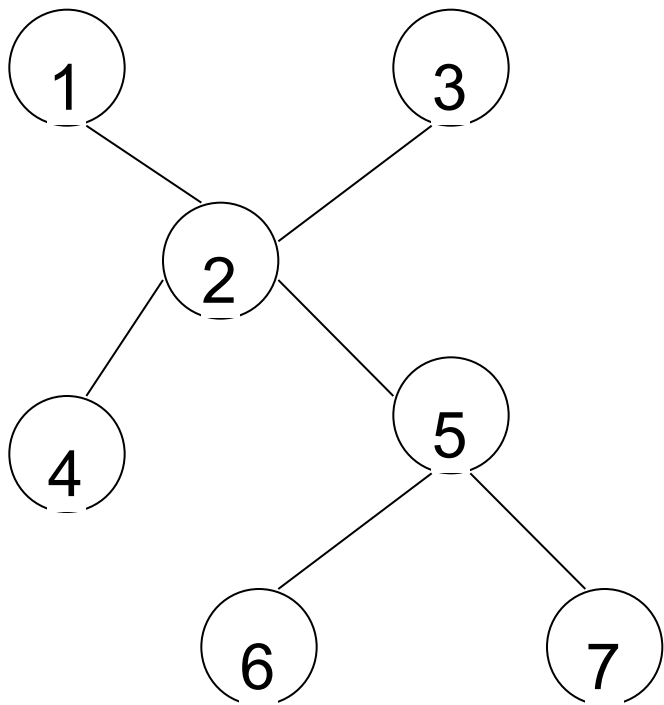
Утверждение

- В любом дереве можно ввести отношение предок-потомок со следующими свойствами
 - Предок соединен с потомком ребром дерева
 - Если элементы соединены ребром – один из них предок другого
 - У каждого элемента 0 или 1 предок
 - У элемента может быть любое число потомков
 - Отношение предок-потомок не имеет циклов (т.е. нельзя быть потомком своего потомка, потомком потомка своего потомка и т.д.)
 - Элемент, не имеющий предков, только один – корень дерева.

Доказательство

- Возьмем произвольный узел и объявим его корнем.
- Все соединенные с ним узлы – его потомки и узлы 1-ого уровня
- Все узлы, соединенные с узлами первого уровня, кроме корня – их потомки и узлы 2-ого уровня
- ...
- Поскольку граф ациклический, отношение предок-потомок не будет иметь циклов

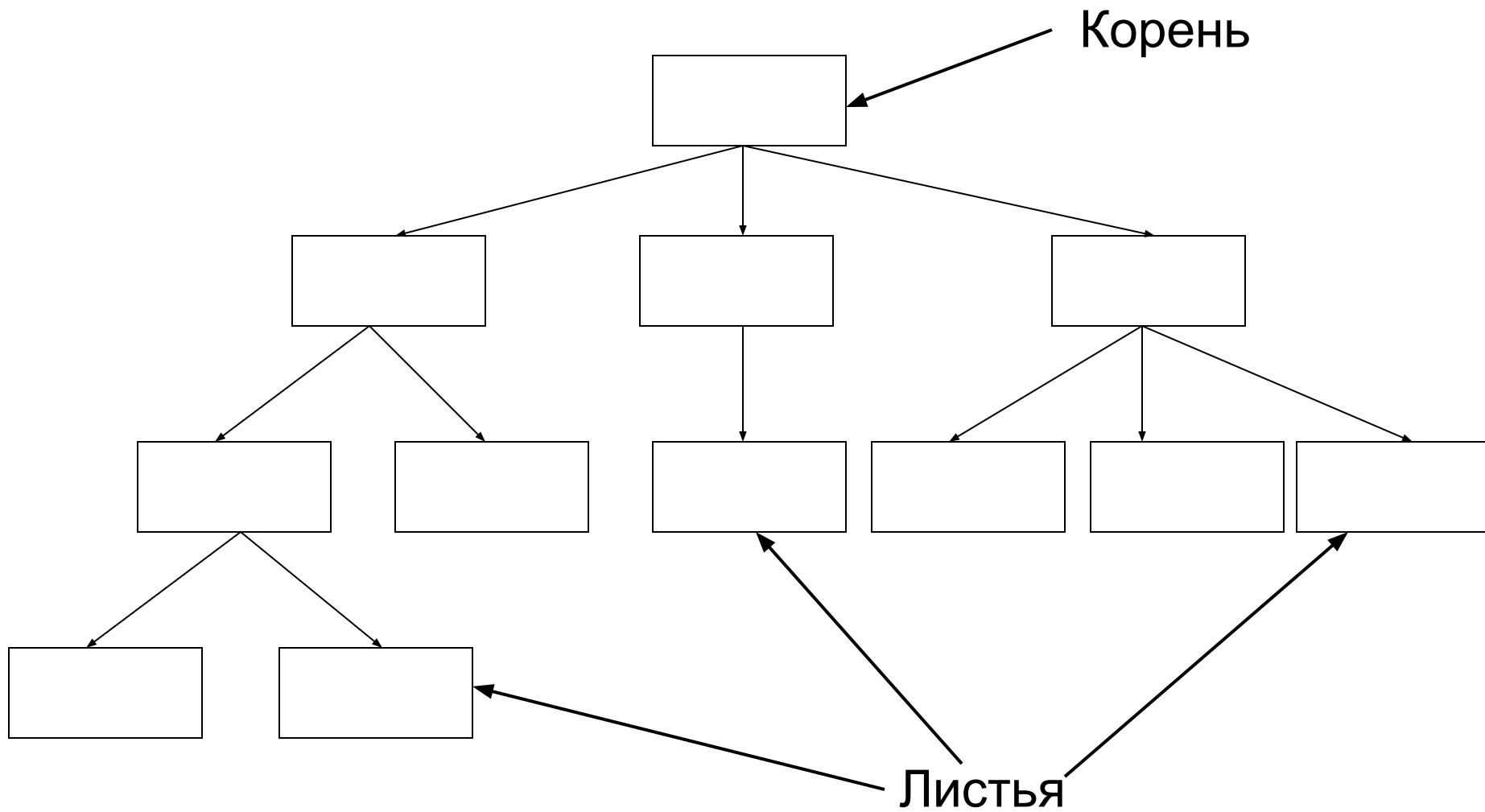
Иллюстрация



Дерево

- Итак, деревом называется контейнер, в котором
 - Элементы связаны отношением предок-потомок
 - У каждого элемента 0 или 1 предок. Как правило, элемент знает его адрес.
 - У каждого элемента могут быть потомки, и он знает их адреса
 - Отношение предок-потомок не имеет циклов (т.е. нельзя быть потомком своего потомка, потомком потомка своего потомка и т.д.)
 - Элемент, не имеющий предков, только один – корень дерева. Он один (иначе это лес, а не дерево)
- Концевые (не имеющие потомков) элементы - листья

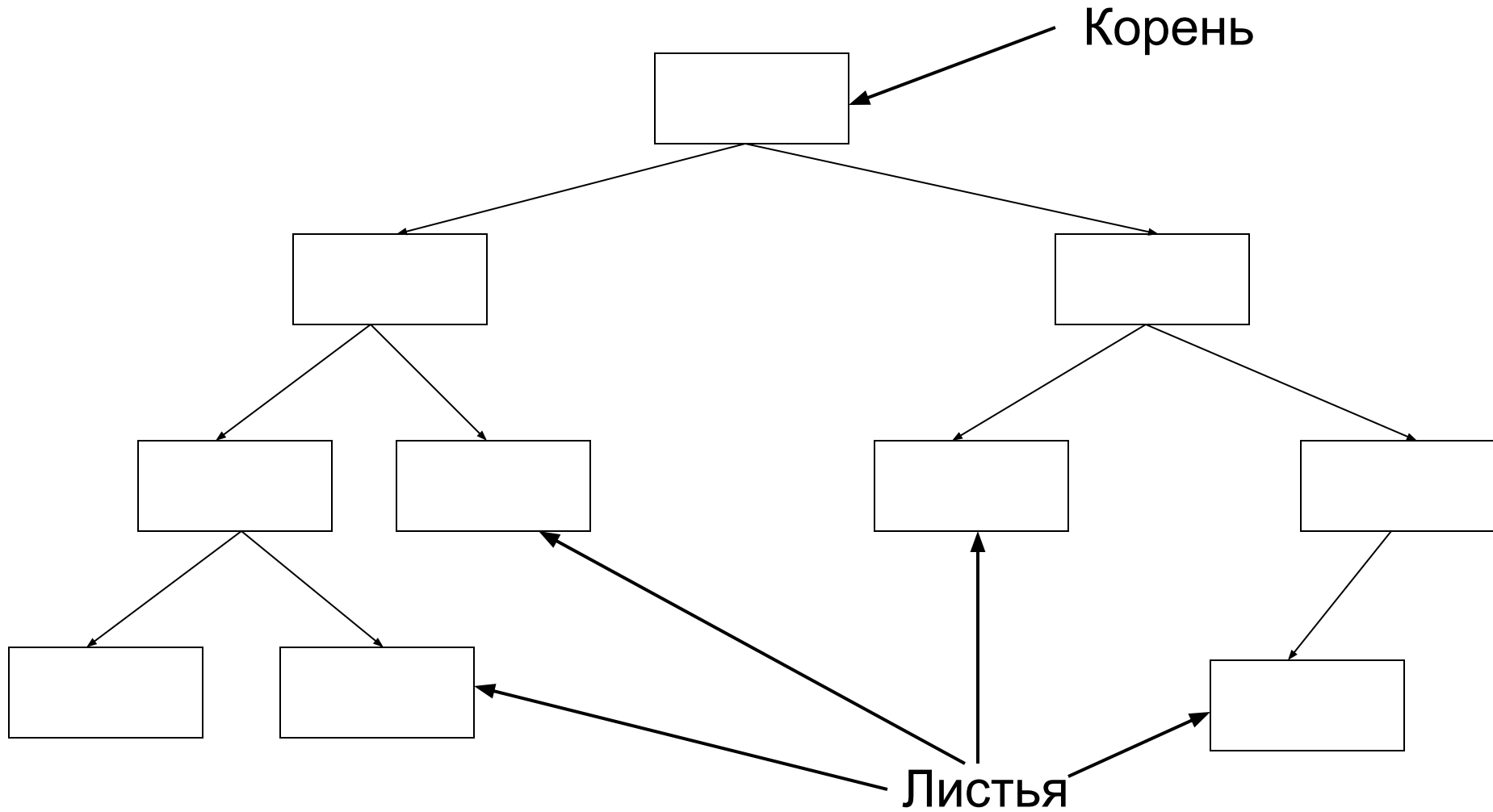
Дерево



Бинарное дерево

- Бинарным называется дерево, в котором у каждого элемента не более 2 потомков
- Один из них называется левым, другой правым

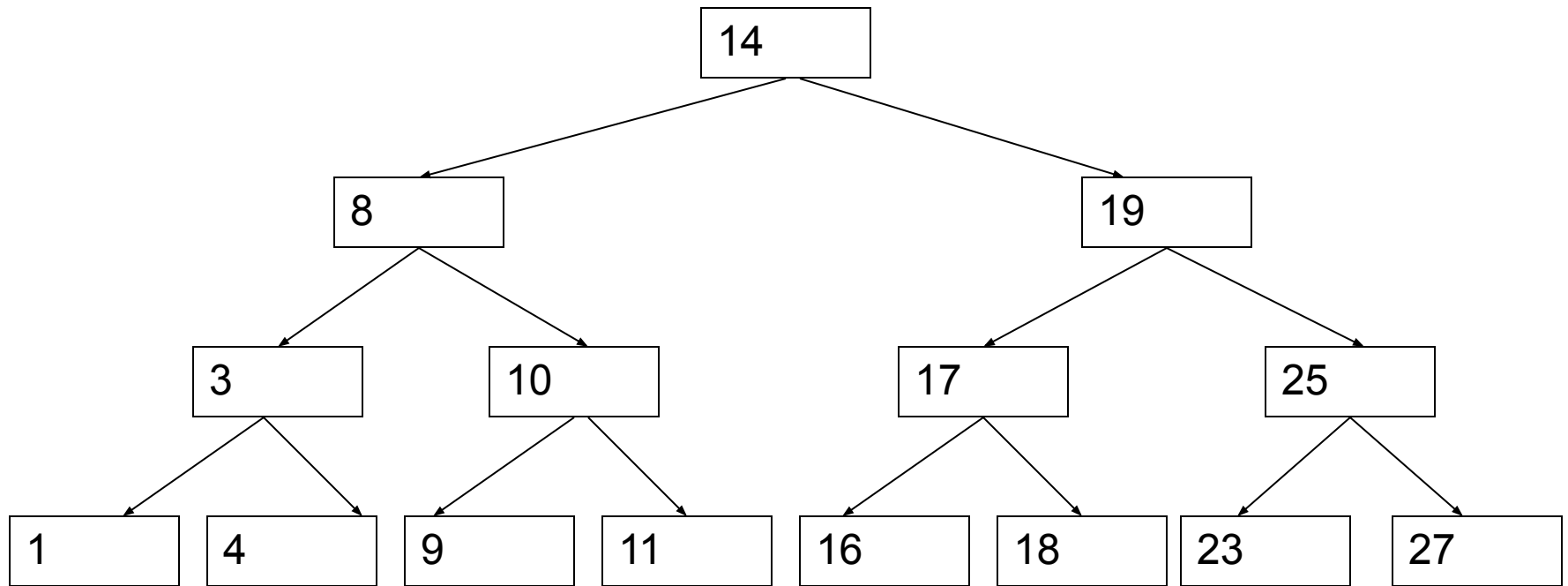
Бинарное дерево



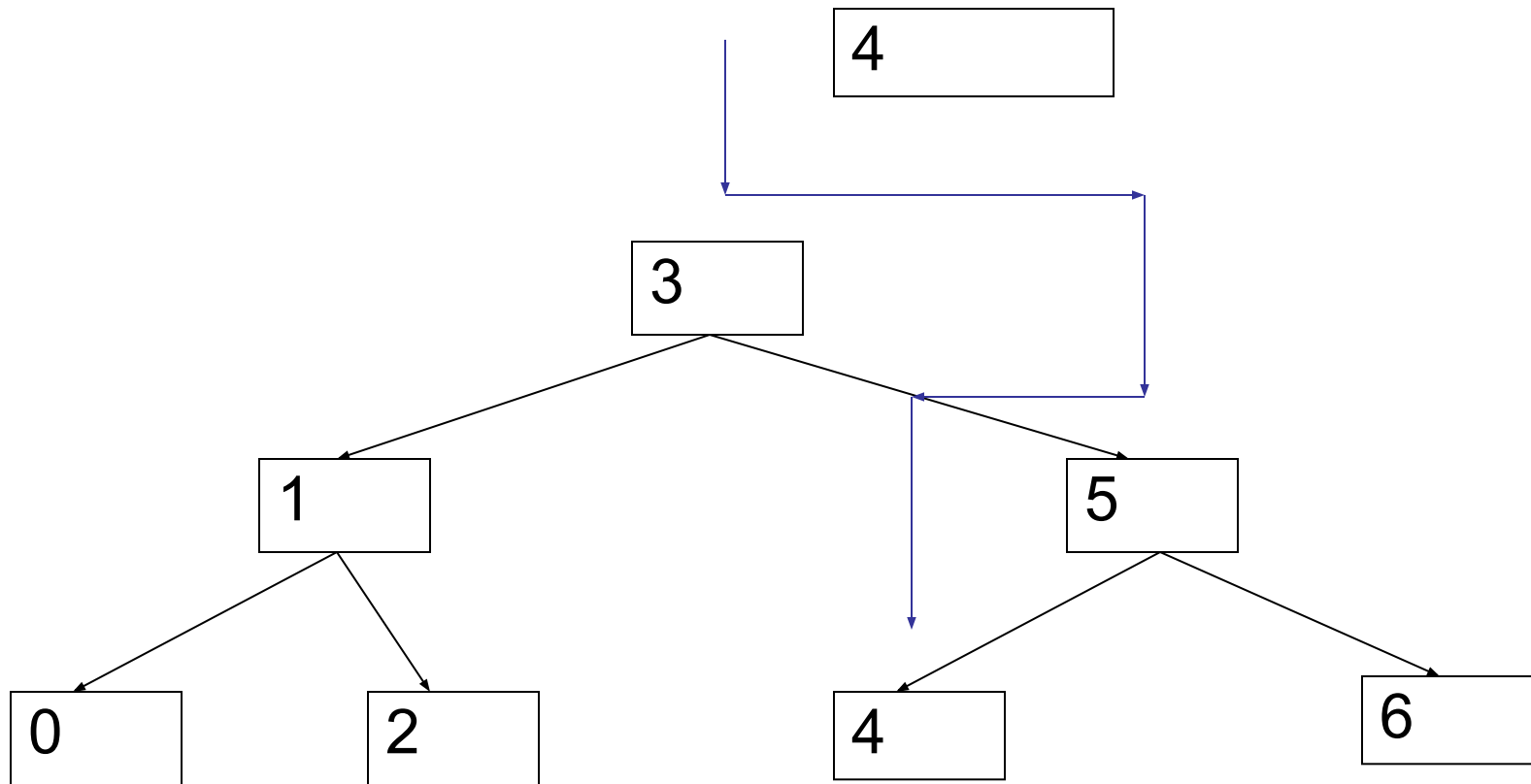
Бинарное дерево поиска

- Бинарное дерево называется деревом поиска, если
 - Левый потомок любого элемента и все элементы поддеревя, растущего из левого потомка, меньше данного элемента
 - Правый потомок любого элемента и все элементы поддеревя, растущего из правого потомка, больше данного элемента

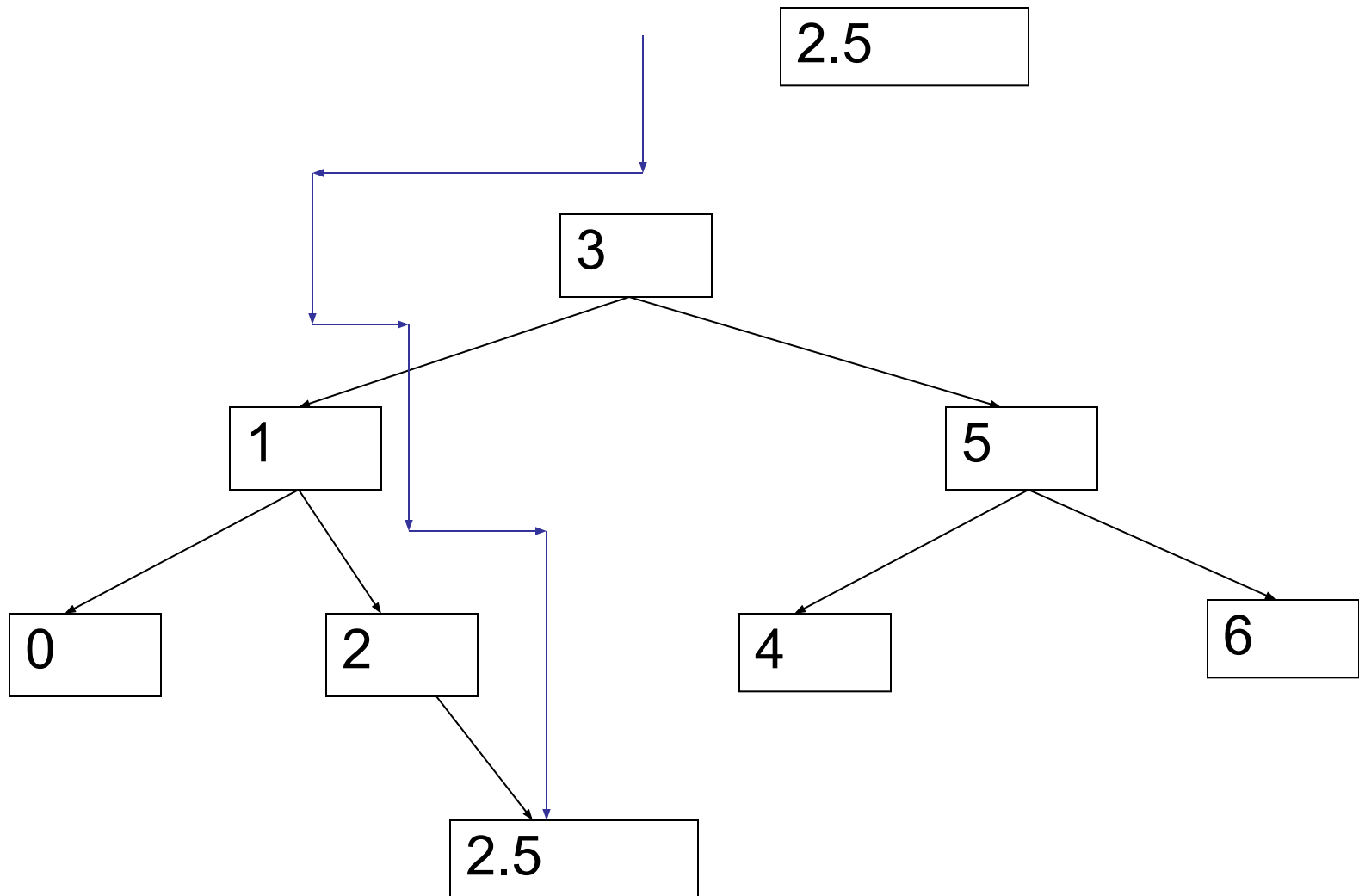
Бинарное дерево поиска



Бинарное дерево. Поиск



Бинарное дерево. Добавление элемента



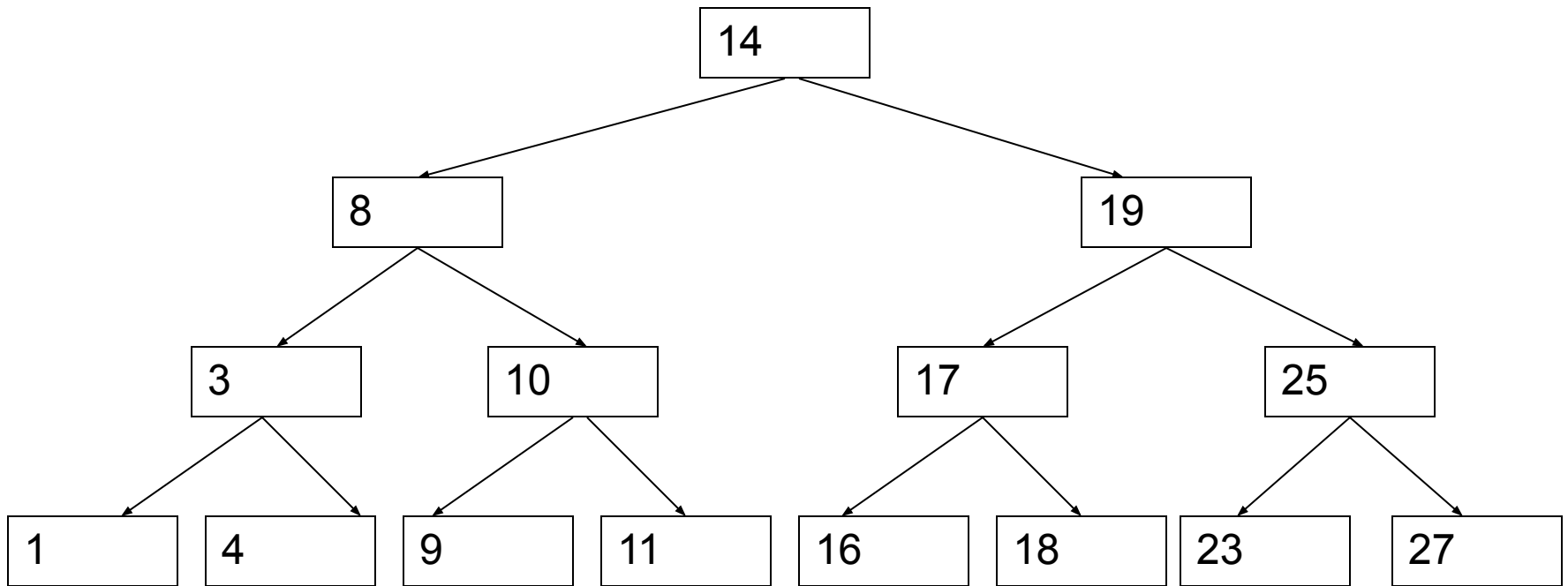
Бинарное дерево поиска

- Как и отсортированный массив, поддерживает поиск за $\log(N)$
- В отличие от отсортированного массива, поддерживает добавление элемента за $\log(N)$

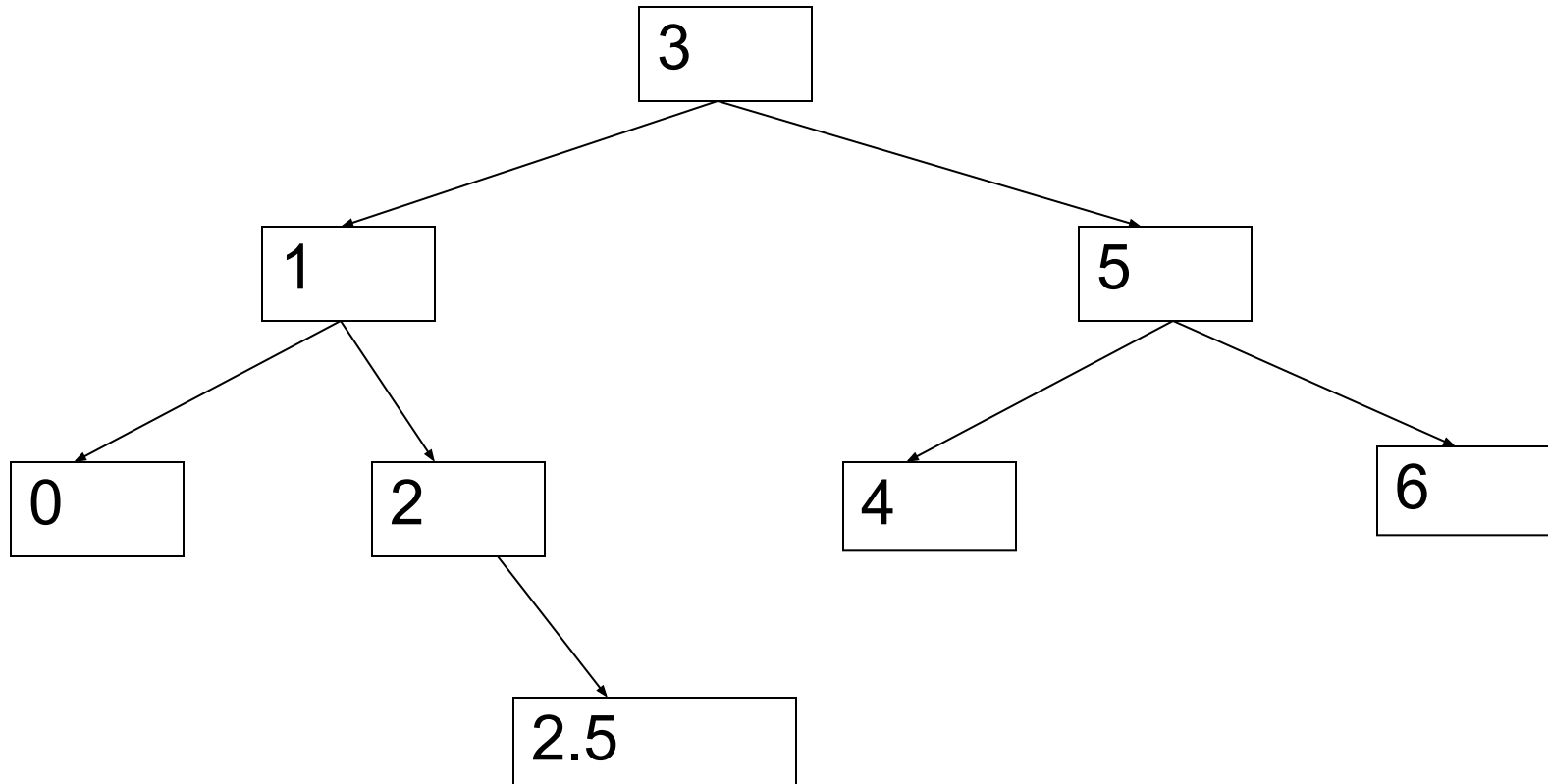
Сбалансированное дерево

- Дерево является сбалансированным, если разница между его максимальной и минимальной глубиной (количеством элементов от корня до листа) не больше 1.

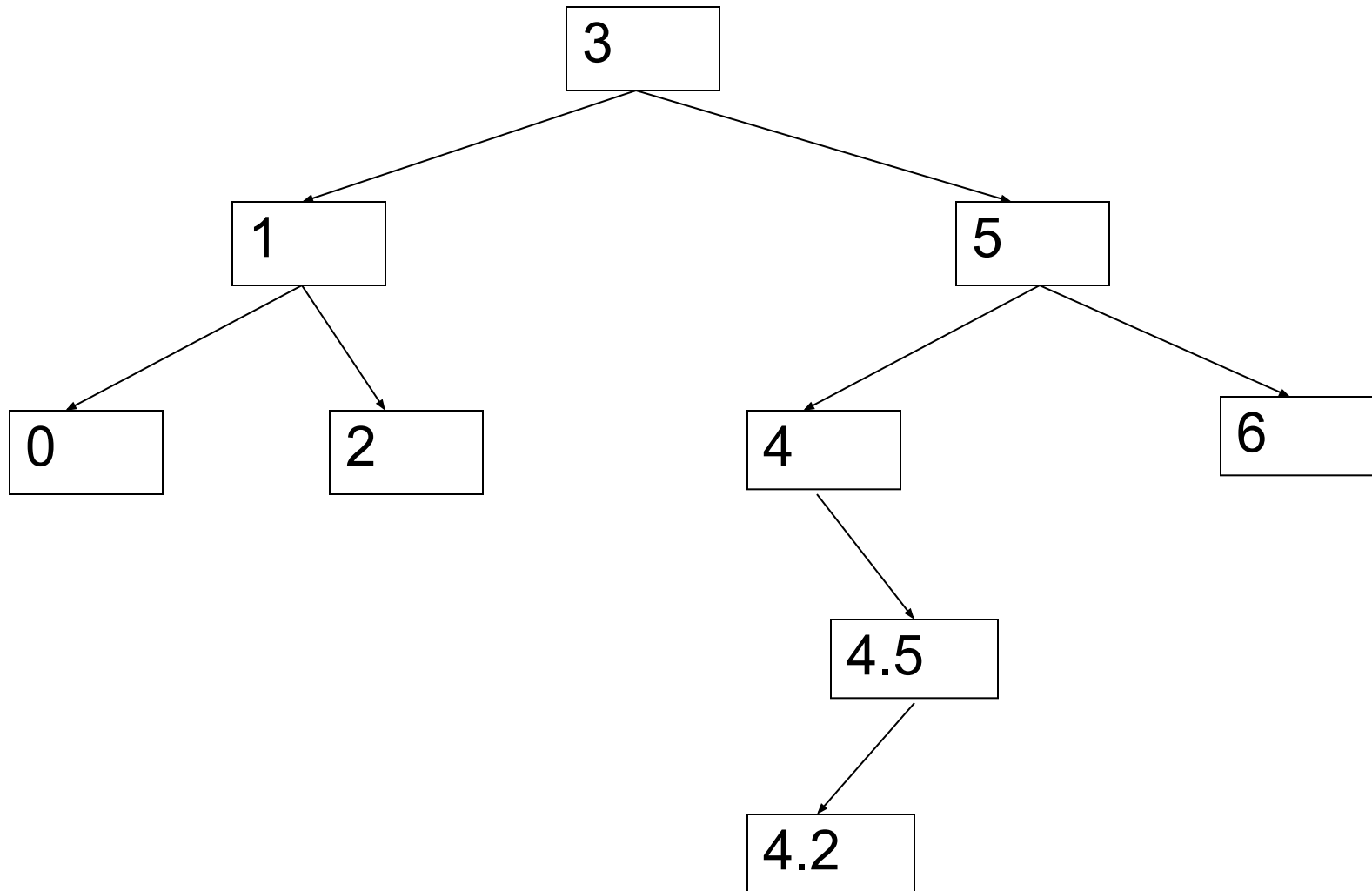
Сбалансированное дерево



Сбалансированное дерево



Несбалансированное дерево



Сбалансированное дерево

- Дерево должно быть сбалансированным, чтобы поддерживать поиск и добавление элемента за $\log(N)$
- Существуют различные алгоритмы реализации бинарных деревьев поиска
- Они отличаются способом обеспечения сбалансированности дерева

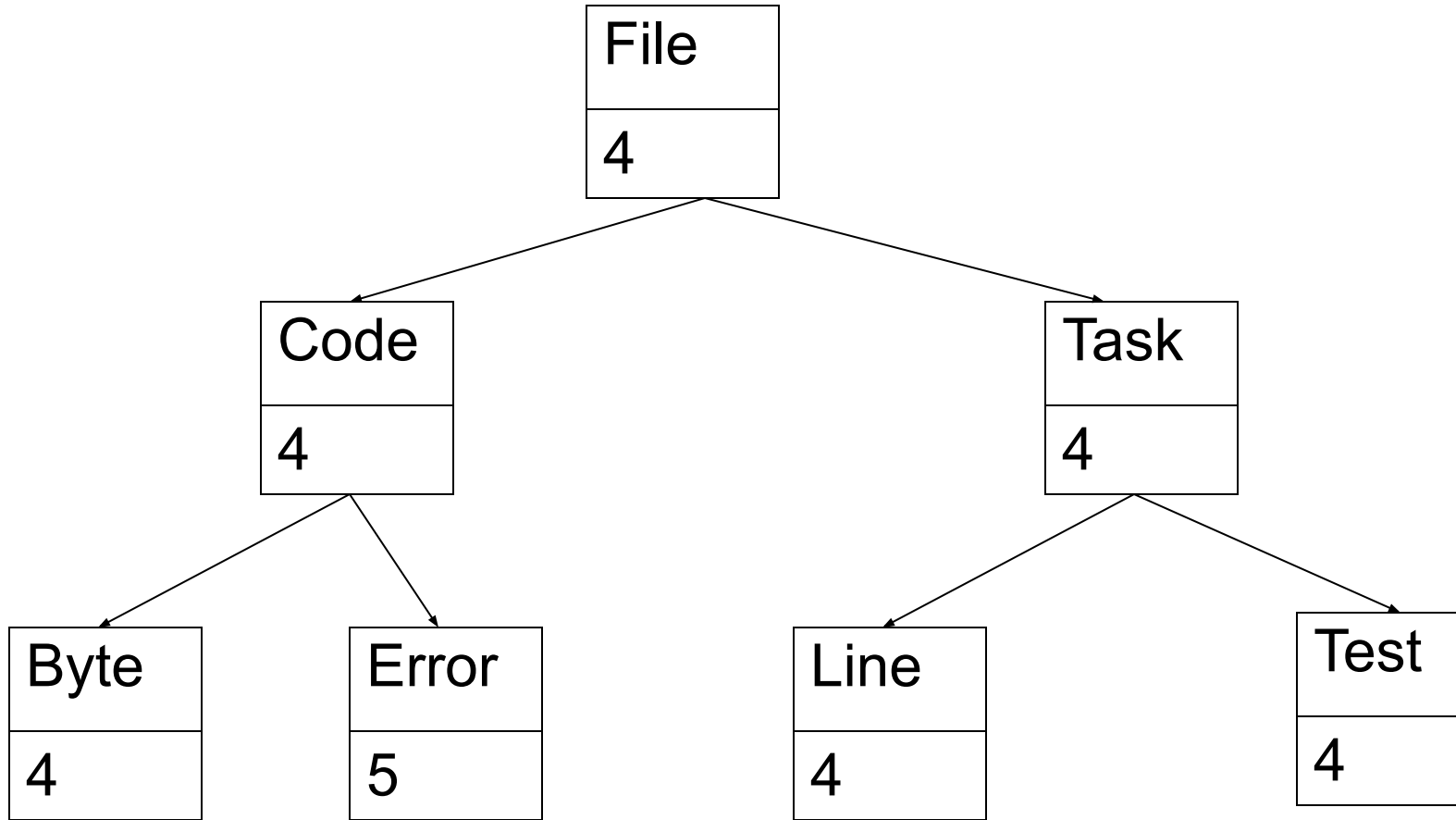
Сбалансированное дерево

- Варианты:
 - Красно-черные деревья
 - AVL-деревья

Словари

- Словарь – структура данных, в которой ключам сопоставляются значения (как в толковом словаре словам сопоставляются определения)
- Словарь должен поддерживать быстрый поиск по ключу и быстрое добавление значения
- Словарь строят на основе бинарного дерева поиска

Словарь



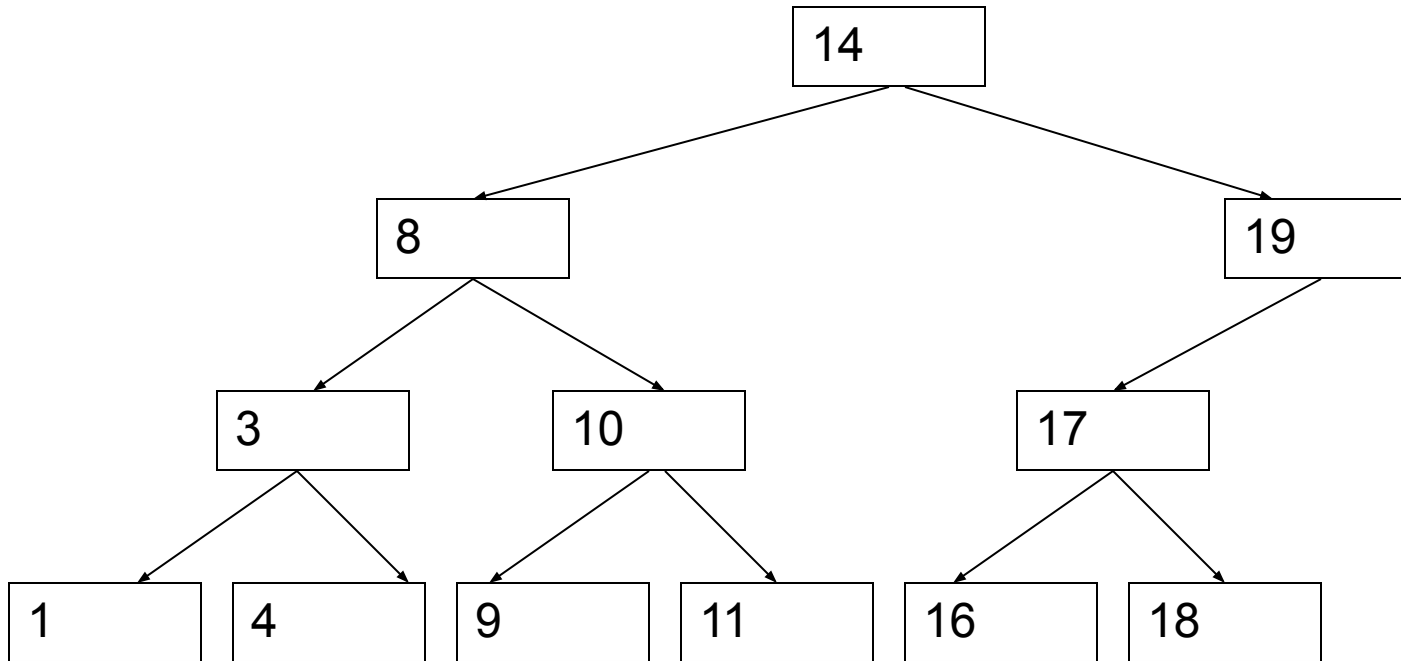
Словарь

- Ключи (в данном случае строковые) отсортированы по алфавиту
- Значения (в данном случае целочисленные) не влияют на сортировку

Пирамиды

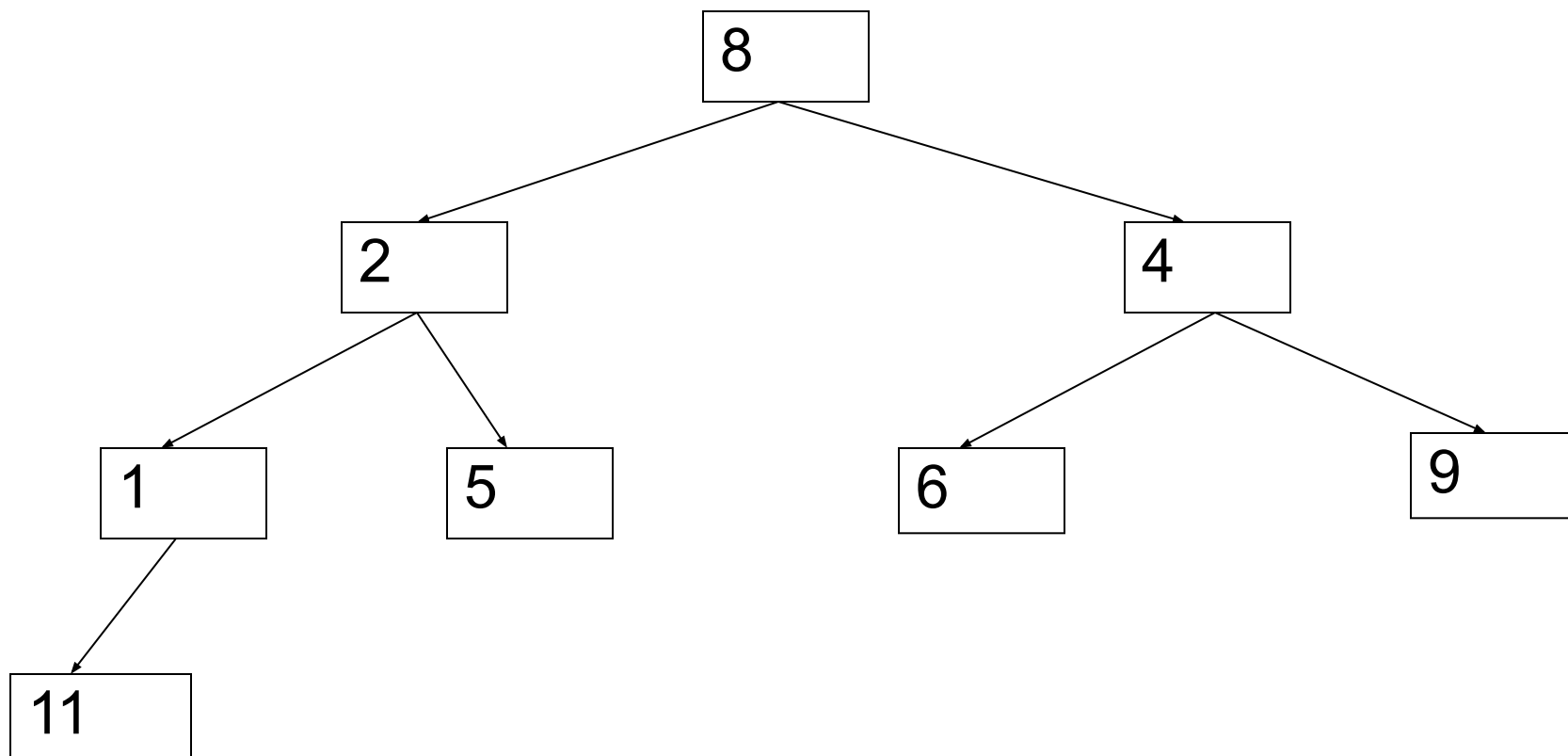
- Пирамида – это бинарное дерево со следующими свойствами
 - Все уровни дерева, возможно кроме последнего, полностью заполнены (сбалансированность дерева)
 - На последнем уровне заполнены несколько элементов, начиная с самого левого

Пирамида?



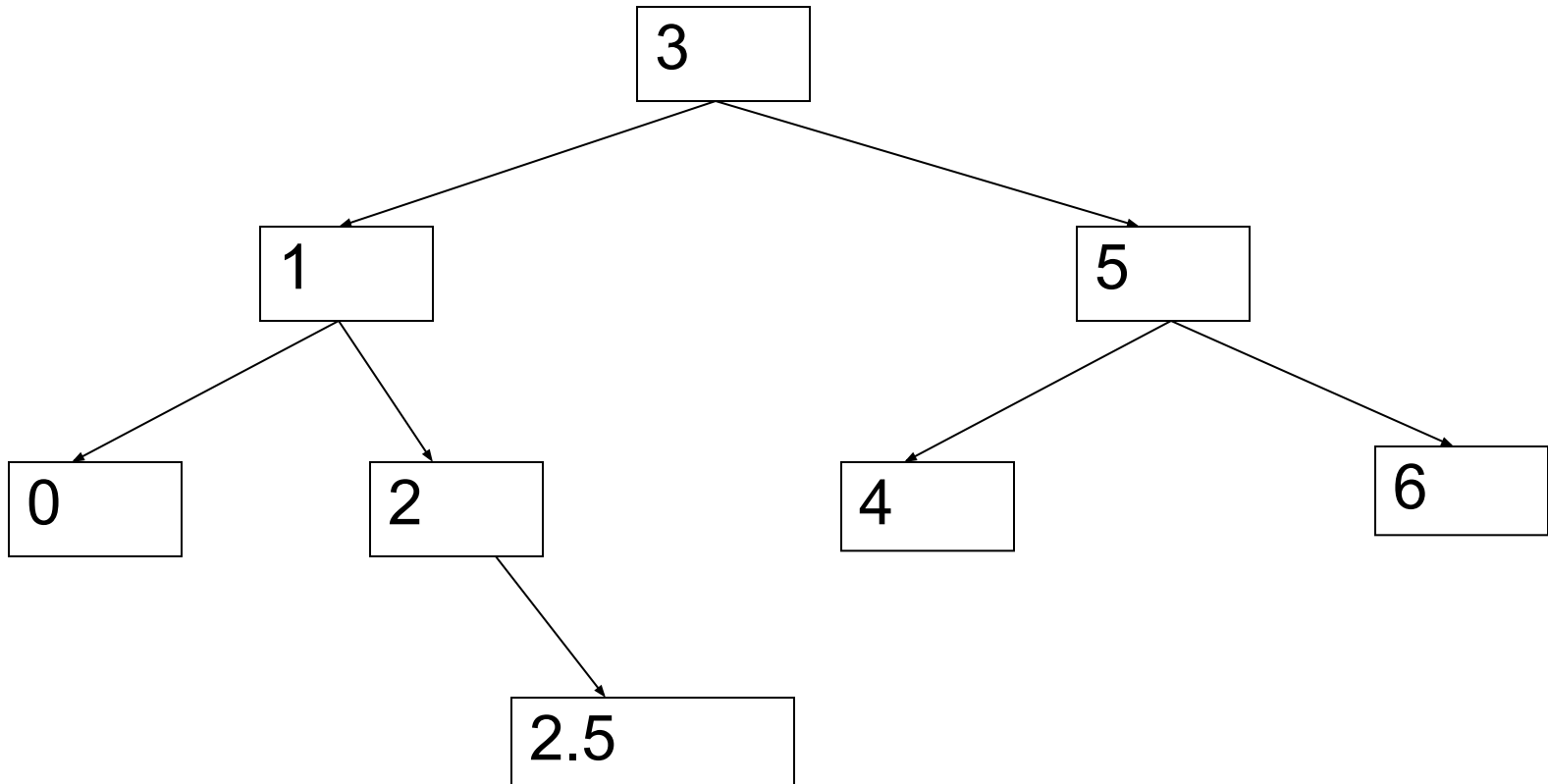
Нет – не заполнен 3-ий уровень

Пирамида?



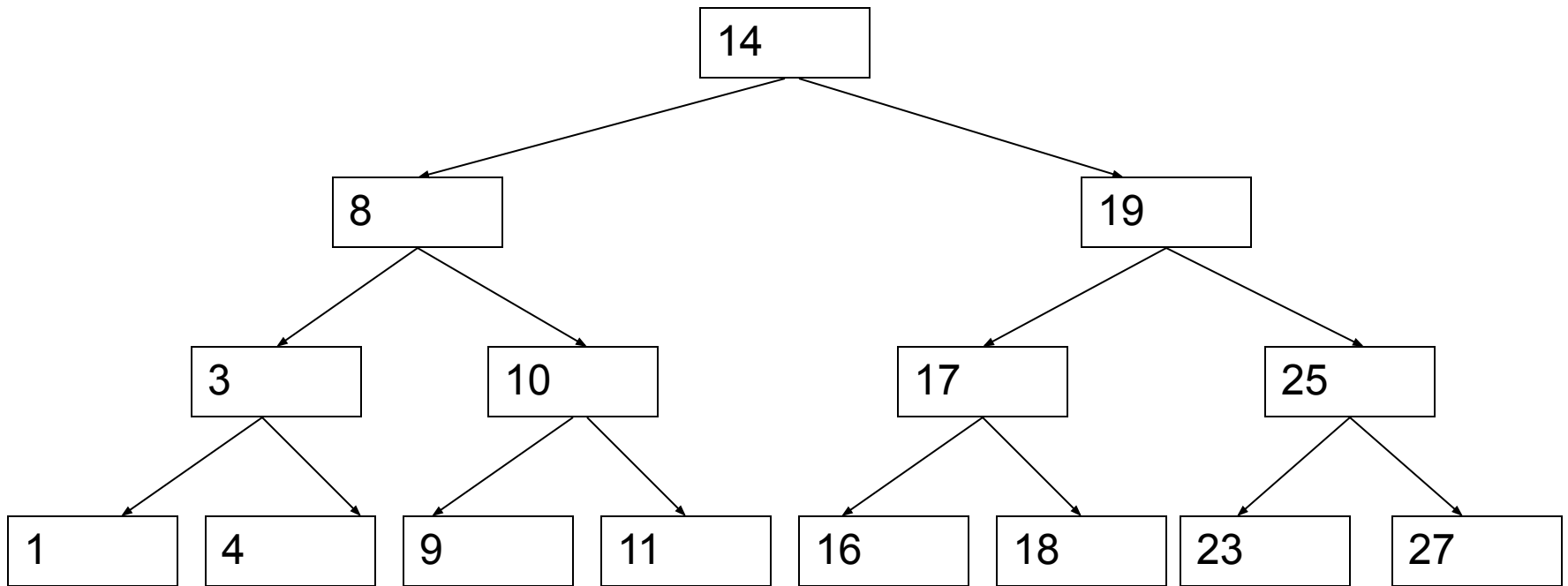
Да

Пирамида?



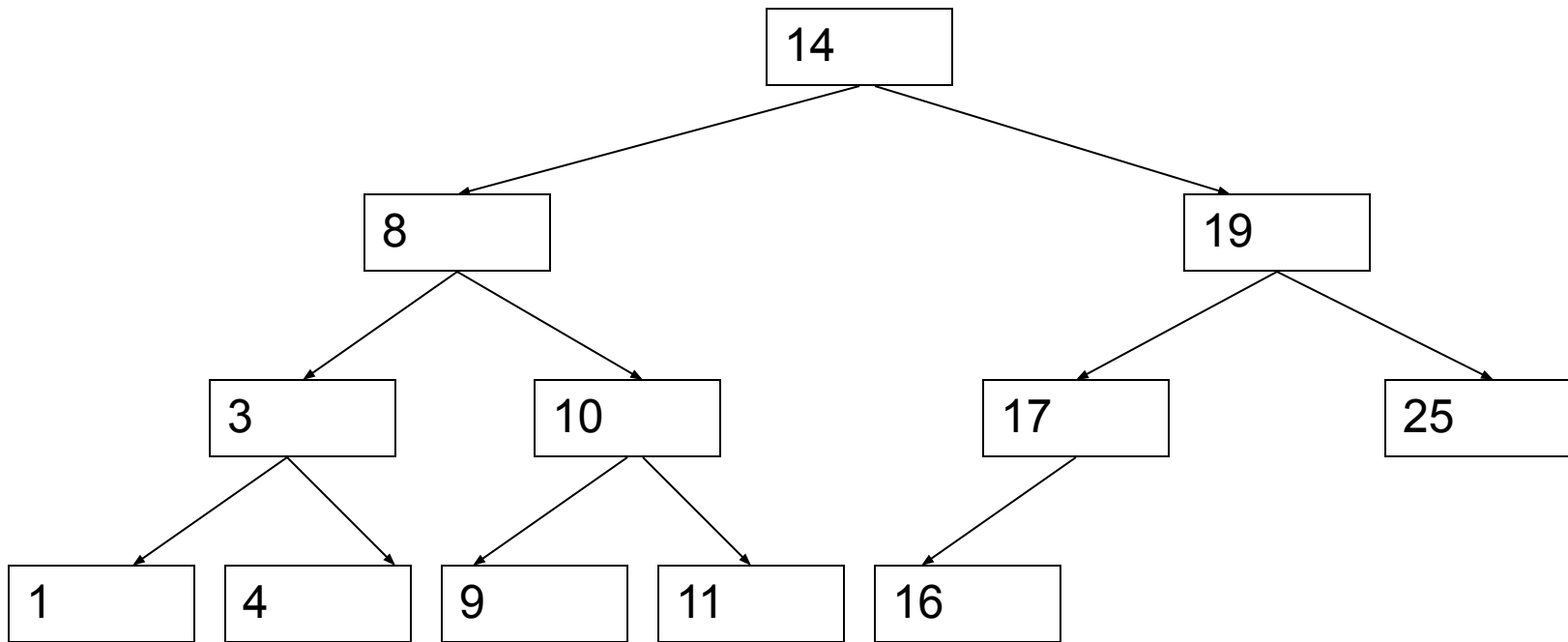
Нет – на 4-ом уровне заполнен не самый левый элемент

Пирамида?



Да

Пирамида?

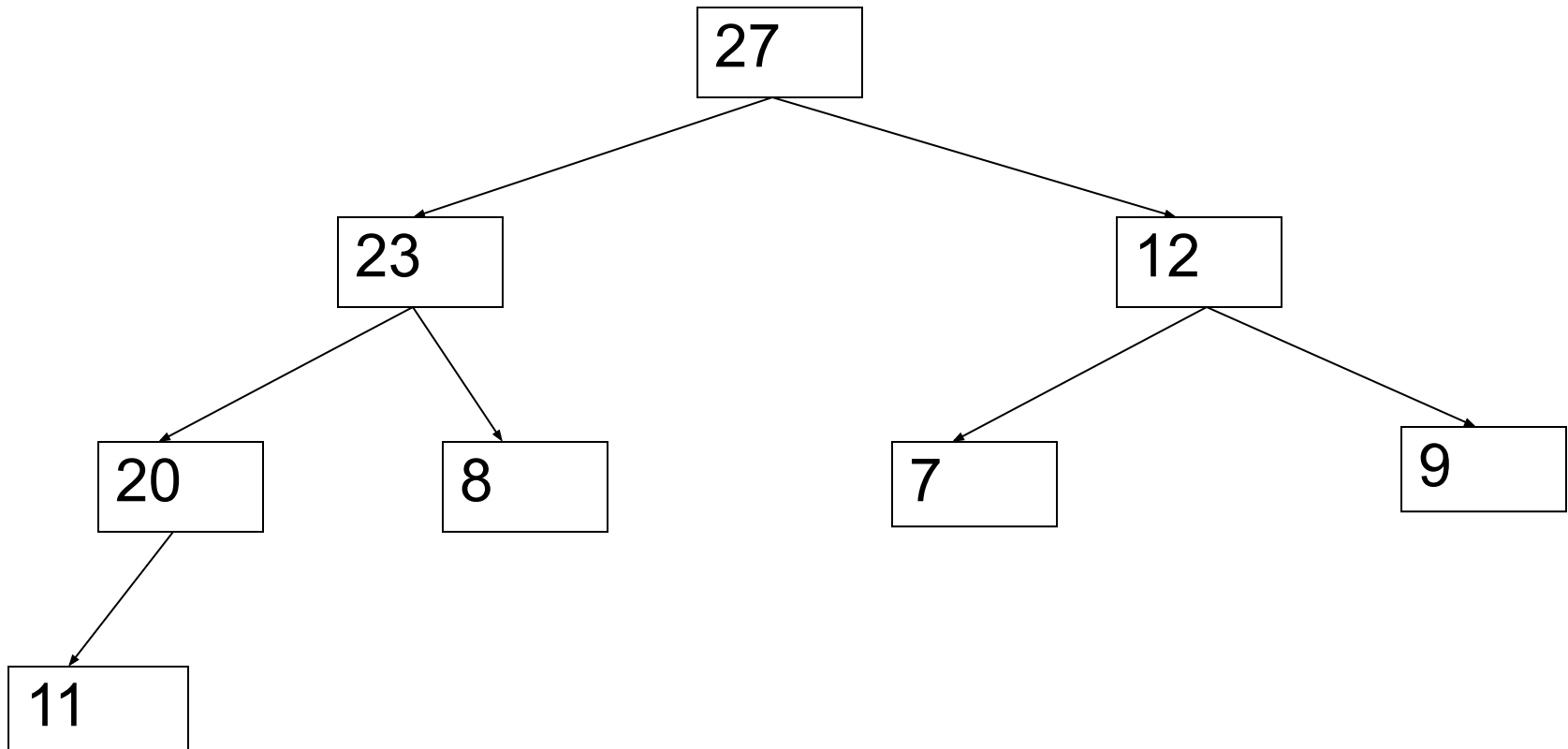


Да

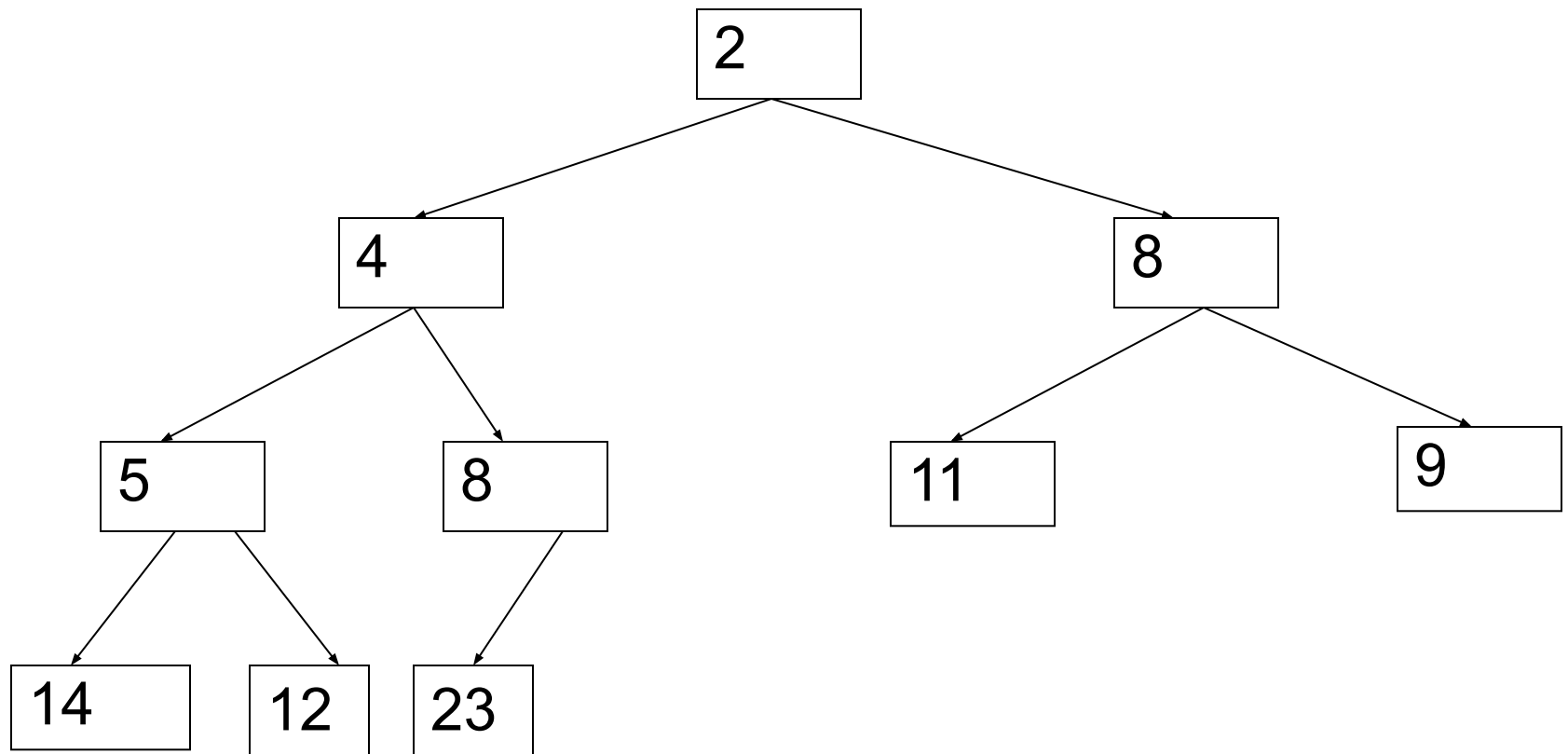
Пирамида

- Пирамида называется невозрастающей, если любой родительский элемент больше (либо равен) обоим дочерним элементам
- Пирамида называется неубывающей, если любой родительский элемент меньше (либо равен) обоим дочерним элементам

Невозрастающая пирамида



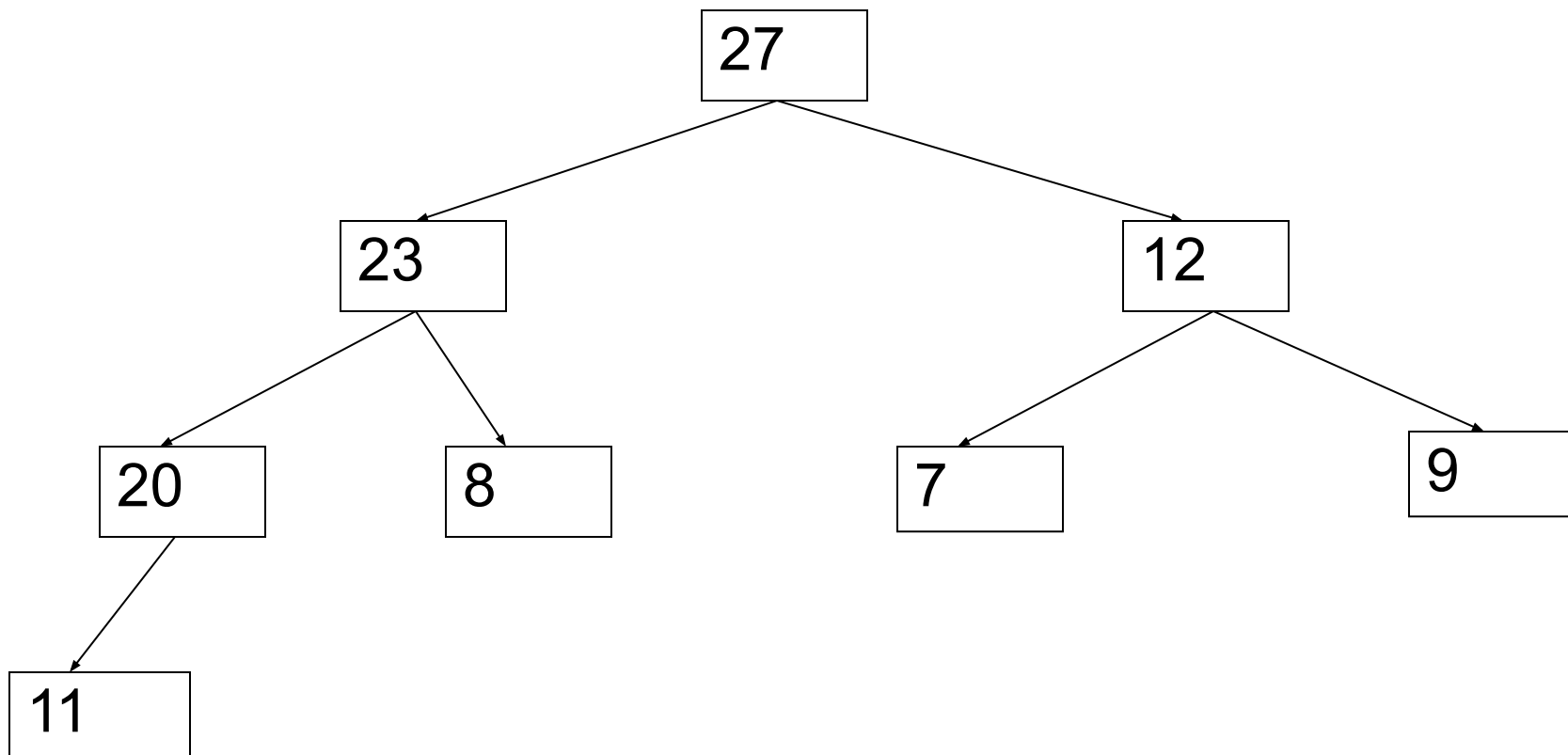
Неубывающая пирамида



Операции над невозрастающей пирамидой

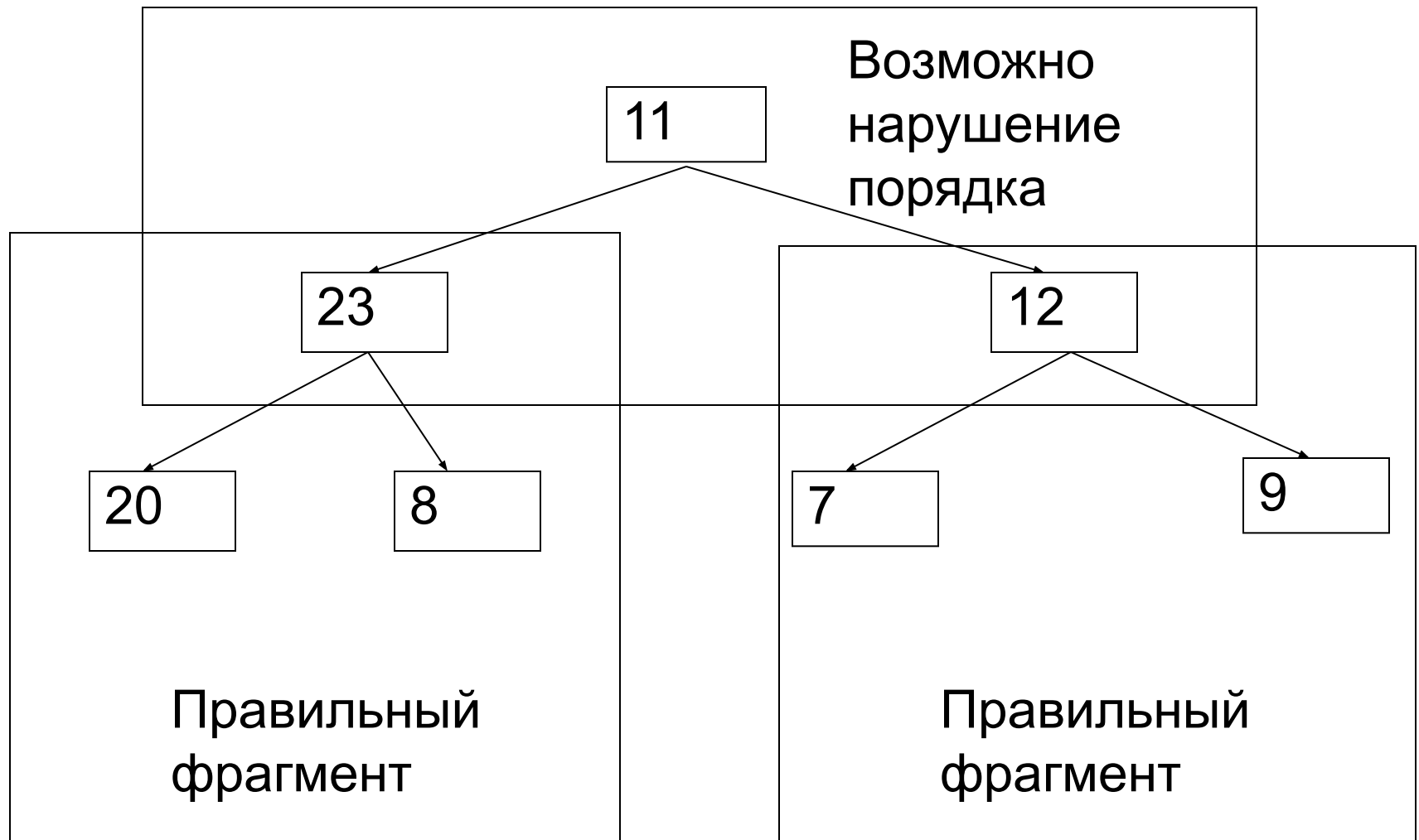
- Из невозрастающей пирамиды можно извлечь максимальный элемент за время $O(\log N)$ так, чтобы она осталась невозрастающей
- В невозрастающую пирамиду можно добавить элемент за время $O(\log N)$ так, чтобы она осталась невозрастающей

Извлечение элемента из пирамиды



Извлечение элемента из пирамиды

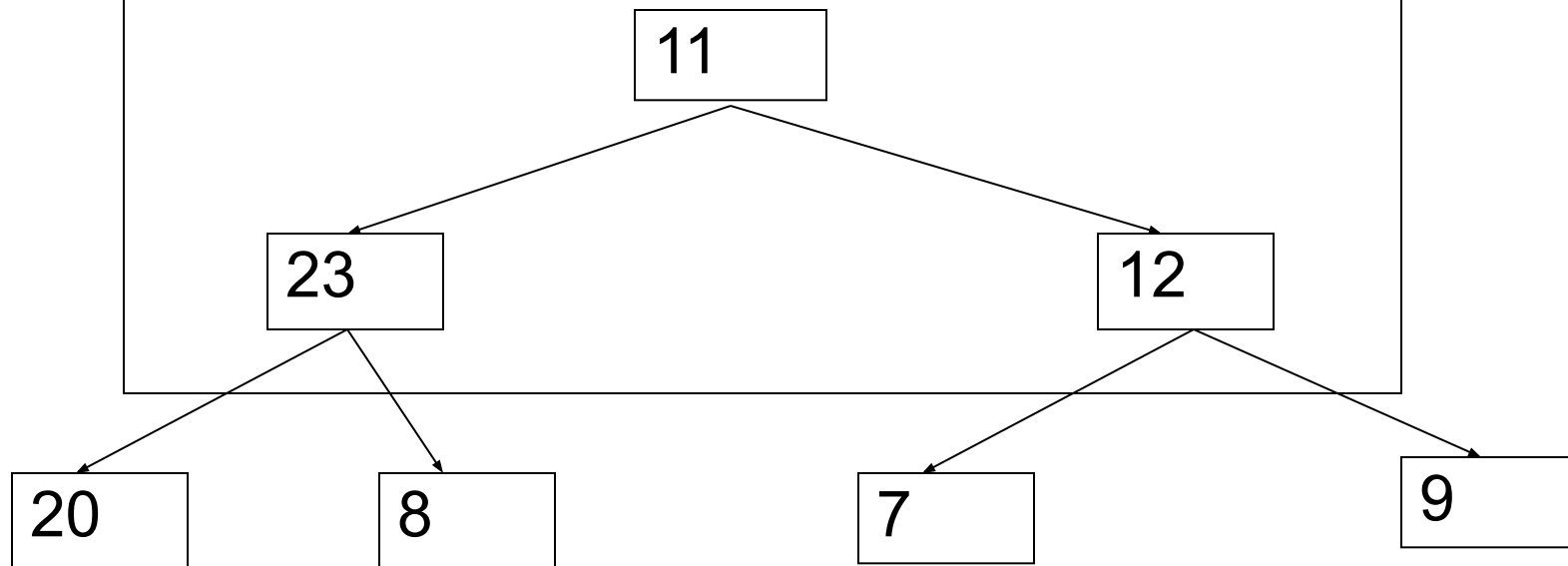
27



Извлечение элемента из пирамиды

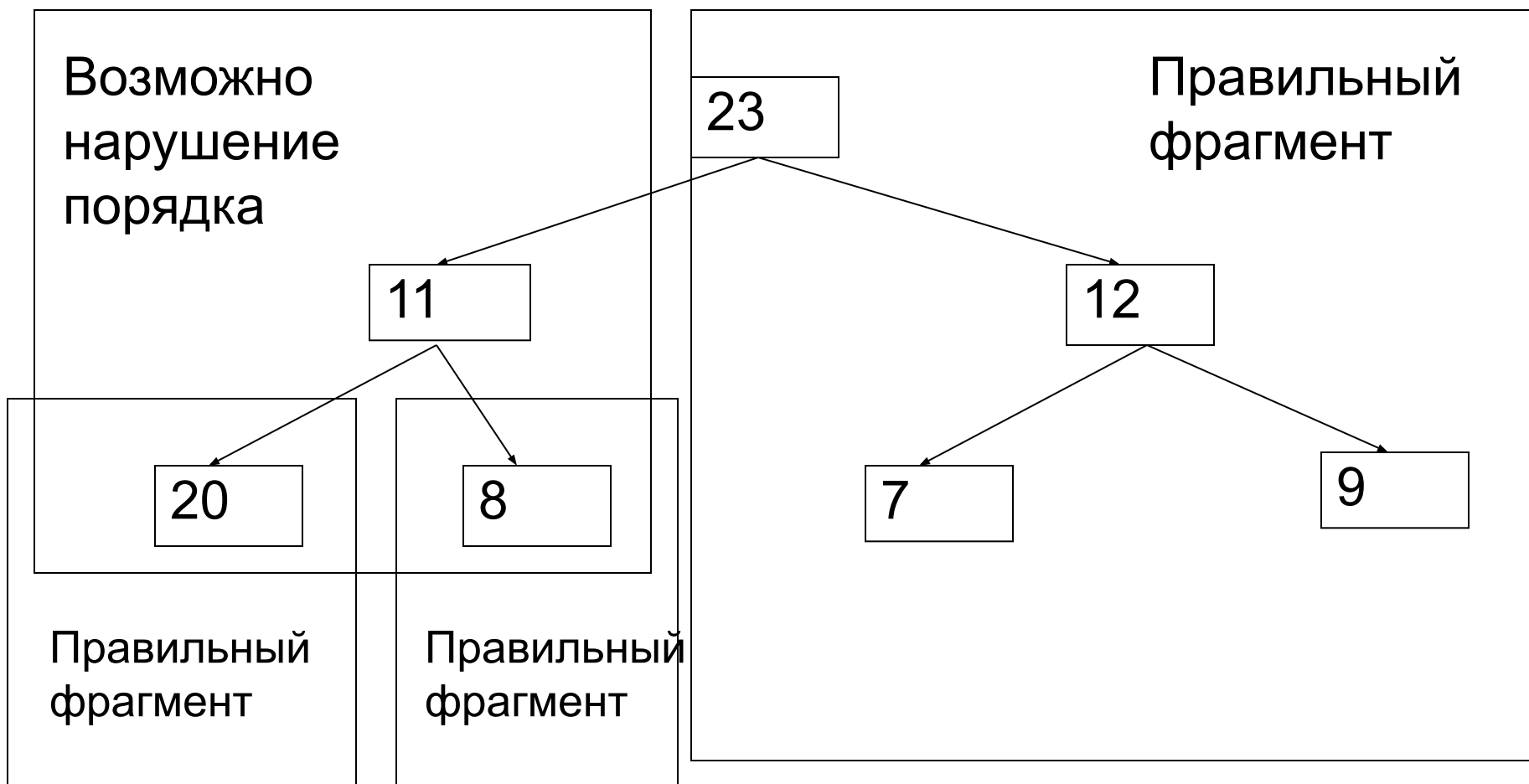
27

Выберем максимум и поменяем местами с верхним элементом



Извлечение элемента из пирамиды

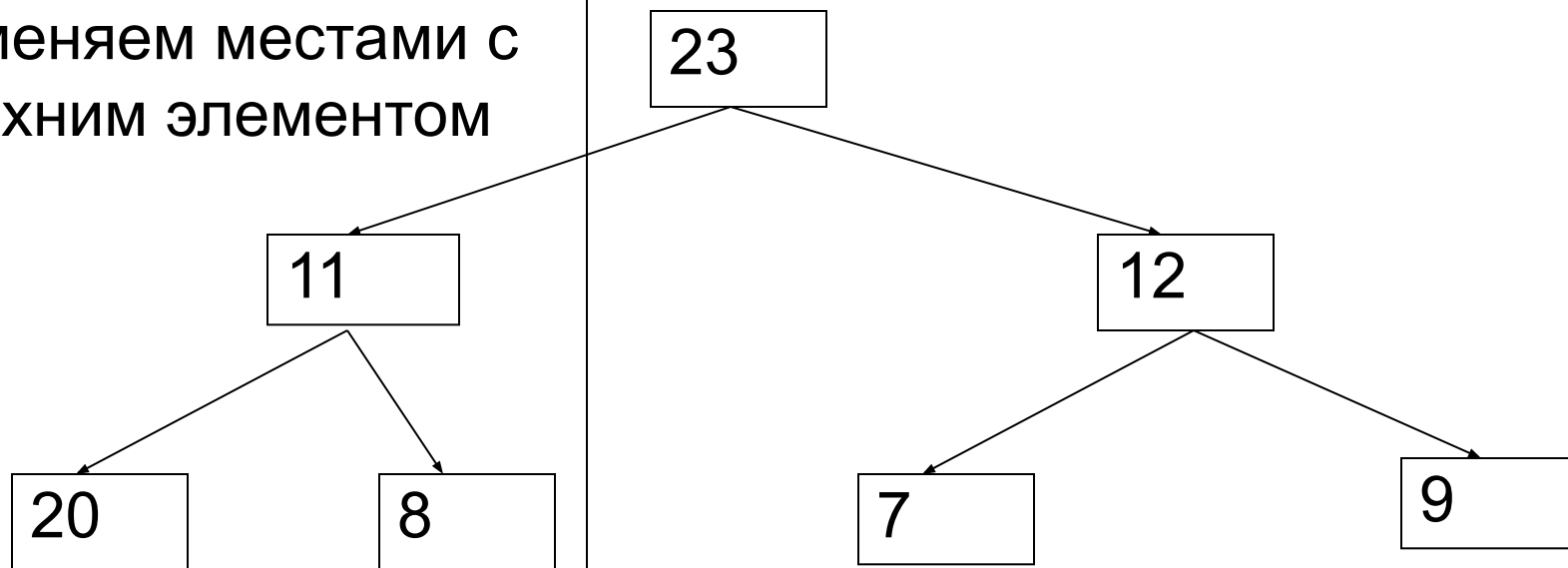
27



Извлечение элемента из пирамиды

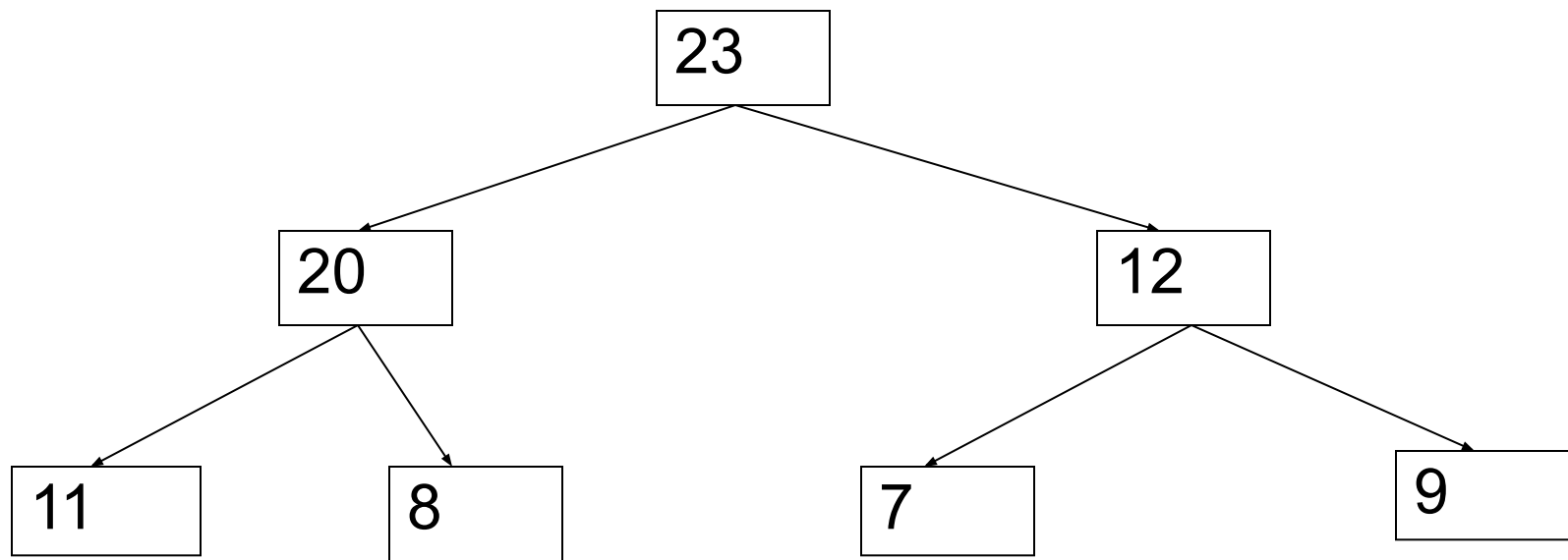
27

Выберем максимум и поменяем местами с верхним элементом



Извлечение элемента из пирамиды

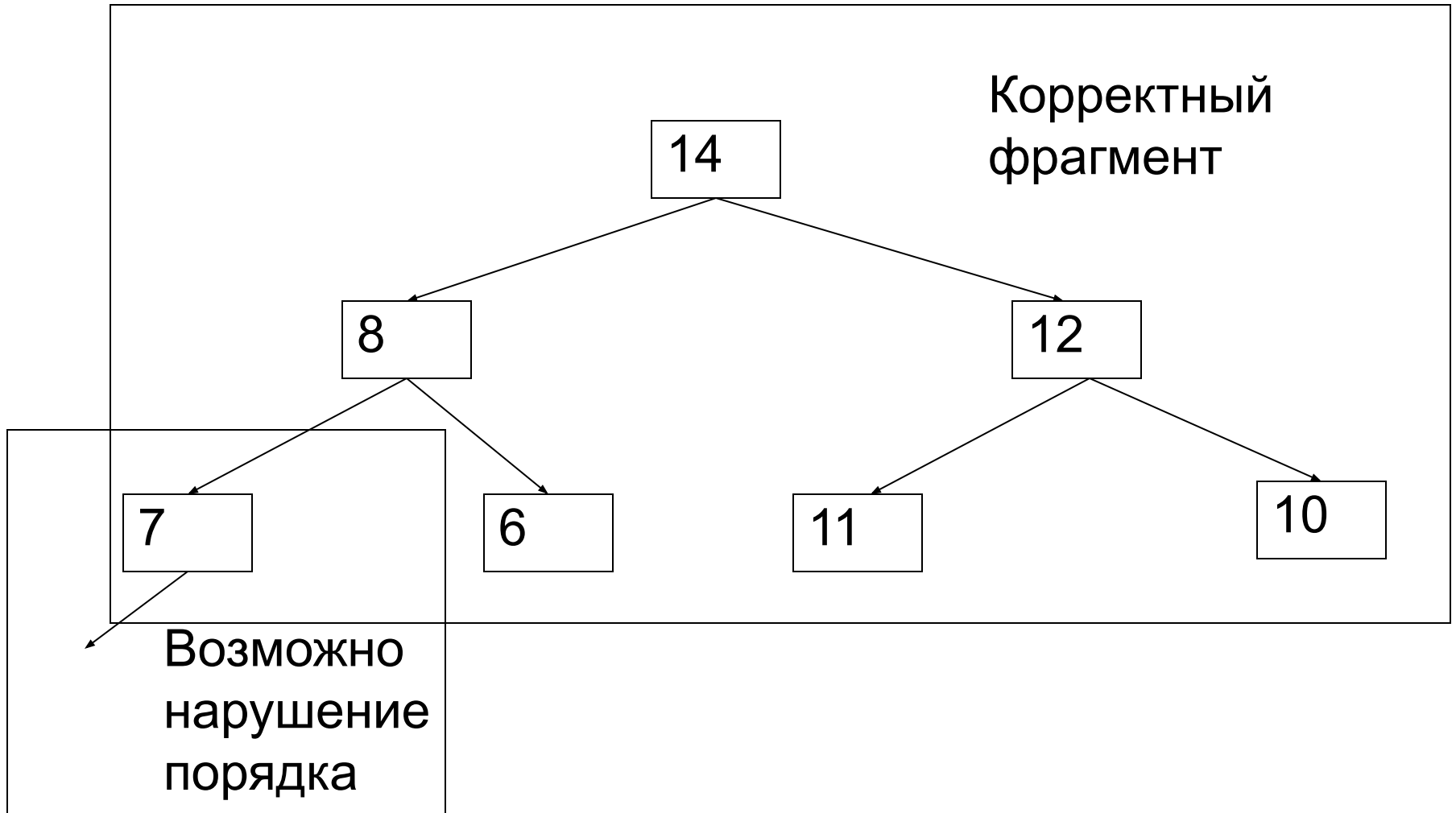
27



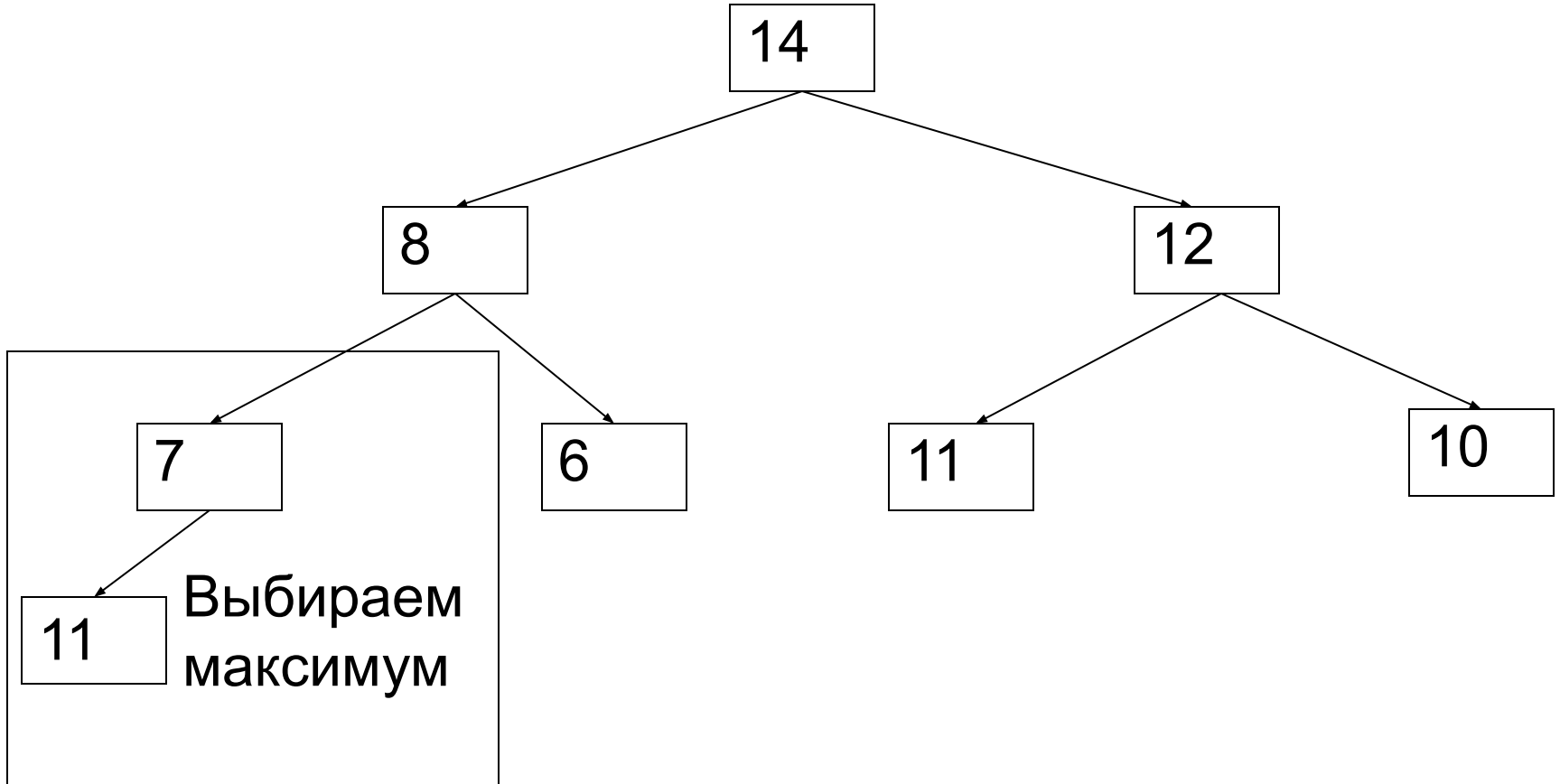
Завершено!

Добавление элемента в пирамиду

11



Добавление элемента в пирамиду



Добавление элемента в пирамиду

Возможно нарушение

Корректный фрагмент

14

8

12

11

6

11

10

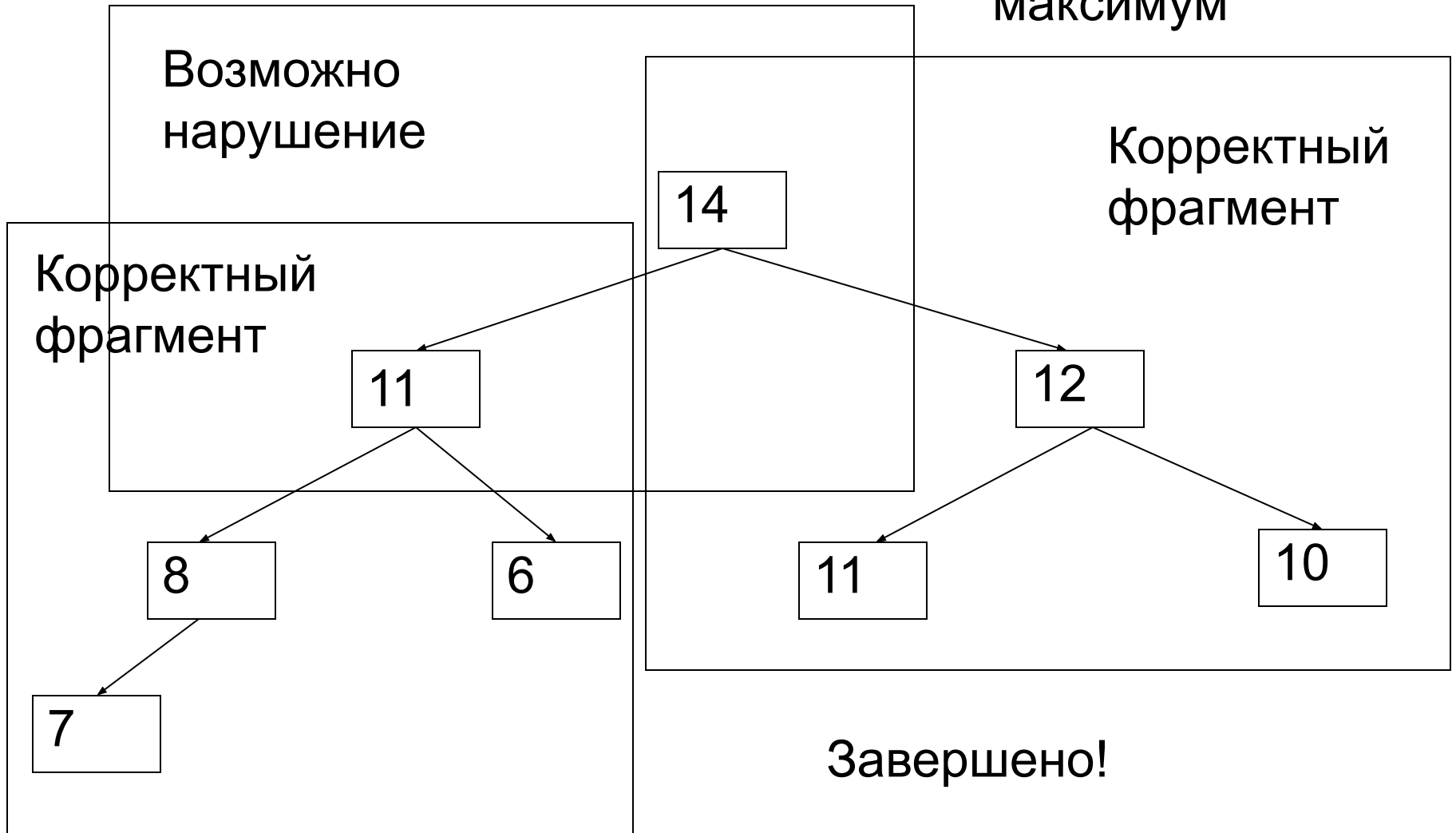
7

Выбираем максимум

Корректный фрагмент

Добавление элемента в пирамиду

Выбираем максимум



Применение пирамиды

- Пирамида используется в пирамидальной сортировке – построив пирамиду и извлекая из нее элементы, мы реализуем сортировку за $O(N \log N)$
- Пирамида может рассматриваться как очередь с приоритетами. В ней можно выполнить за $O(\log N)$ операции
 - Выборки максимального элемента
 - Добавления нового элемента в очередь
 - Повышения приоритета элемента

Хранение пирамиды

- Мы можем хранить пирамиду как обычное бинарное дерево (каждый узел представляется как структура, состоящая из значения элемента, указателей на дочерние узлы и родительский узел)
- Этот механизм требует использовать дополнительную память для хранения указателей

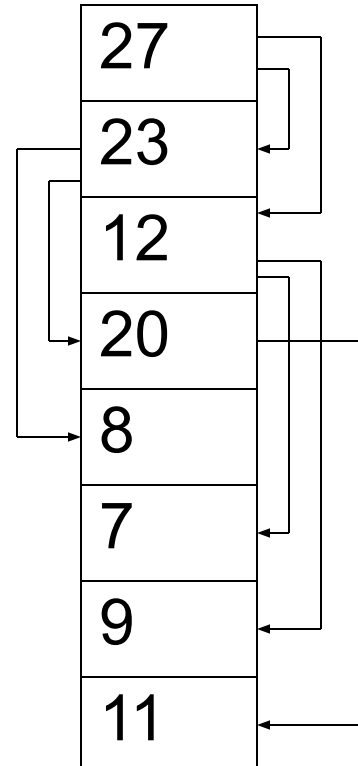
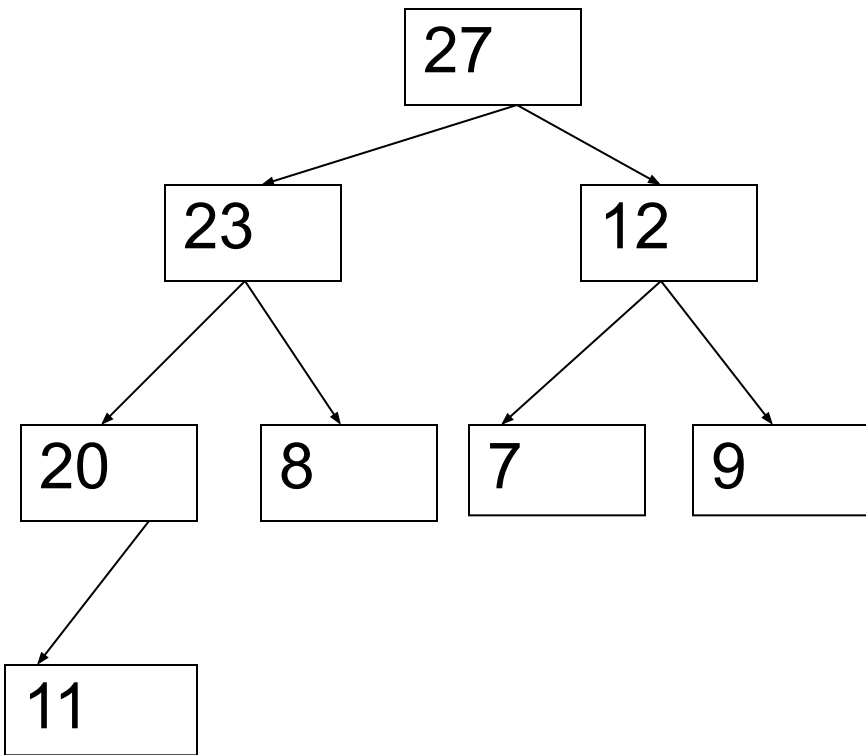
Хранение пирамиды

- Пирамиду можно хранить без выделения дополнительной памяти
- Для этого пирамида представляется как массив

Хранение пирамиды

- Уровень K пирамиды занимает в массиве позиции от $2^K - 1$ до $2^{K+1} - 2$
- Например, уровень 0 (корень) находится в позиции 0
- Уровень 1 (2 элемента) – в позициях от 1 до 2
- Уровень 3 (8 элементов) – в позициях от 7 до 14

Хранение пирамиды



Хранение пирамиды

- Потомками элемента $A[K]$ являются
 - $A[2 * K + 1]$ – левый потомок
 - $A[2 * K + 2]$ – правый потомок
- Например, у элемента 4 (2-ой слева элемент на 3-ем уровне) потомками будут
 - Элемент 9 – 3-ий слева элемент 4-ого уровня, левый потомок
 - Элемент 10 – 4-ый слева элемент 4-ого уровня, правый потомок

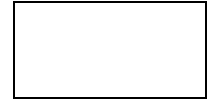
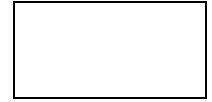
Задание

- Как выглядит код, проверяющий массив на то, что он является невозрастающей пирамидой?

Стек

- Стекком называется контейнер, поддерживающий принцип Last In – First Out
- Мы можем в любой момент добавить новый элемент, посмотреть последний добавленный элемент, удалить последний добавленный элемент

Стек



Стек

- Стек может быть построен на базе практически другого контейнера, например массива
- Стек ограничивает количество операций контейнера

Очередь

- Очередь – это контейнер, поддерживающий принцип First In – First Out
- Существуют операции добавления элемента в очередь и удаления элемента, который был добавлен раньше всех

Очередь

Очередь

- Очередь также легко реализуется на базе другого контейнера (например, массива)

Лекция 4. Хэш-таблицы. Понятие о хэш-функции. Идея хэширования.

Хэш-таблицы. Постановка задачи.

- Бинарные деревья поиска позволили реализовать поиск элемента в контейнере за $O(\log N)$
- Это правило удалось реализовать, введя ограничения на структуру контейнера (не любой элемент не в любую ячейку можно положить)
- Может, если ограничения сделать больше, удастся повисить результат?

Хэш-таблицы – прямая адресация

- Пусть в контейнере планируется хранить целые числа от 0 до $2^{32}-1$
- Для упрощения скажем, что числа могут быть только разные
- Если бы мы могли завести массив длиной 2^{32} - проблема была бы решена
- Хранить каждый элемент только в ячейке, номер которой совпадает с его значением

Хэш-таблицы – прямая адресация

Исходное состояние – значение всех элементов не совпадает с номером, набор пустой

1	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---

Добавление элемента

5

1	0	0	0	0	5	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---

Добавление элемента

7

1	0	0	0	0	5	0	7	0	0	0
---	---	---	---	---	---	---	---	---	---	---

Хэш-таблицы – прямая адресация

1	0	0	0	0	5	0	7	0	0	0
---	---	---	---	---	---	---	---	---	---	---

Поиск элемента

2

Не совпали – значит,
такого нет

Поиск элемента

7

Совпали – значит,
такой есть

О достоинствах и недостатках схемы

- Поиск любого элемента выполняется за фиксированное время ($O(1)$)
- Добавление нового элемента выполняется за фиксированное время ($O(1)$)
- Количество требуемой памяти пропорционально количеству возможных значений ключа

Идея хэш-функции

- Обеспечить поиск и добавление элемента за время, равное $O(1)$, возможно, если позиция полностью определяется значением (например, в рассмотренном методе прямой адресации – совпадает со значением). Тогда время вычисления позиции по значению фиксировано и не зависит от количества элементов
- Простое правило: «номер совпадает со значением» возможно только для целых чисел и приводит к перерасходу памяти

Идея хэш-функции

- Итак, необходимо, чтобы элемент со значением x сохранялся в позиции $h(x)$.
- $h(x)$ – хэш-функция (от to hash – перемешивать)
- Тогда поиск и добавление элемента выполняются за время $O(1)$

Пример хэш-таблицы

X	X	X	X	X	X	X	X	X	X	X
---	---	---	---	---	---	---	---	---	---	---

Добавление элемента

52

$$52 \% 11 = 8$$

X	X	X	X	X	X	X	X	52	X	X
---	---	---	---	---	---	---	---	----	---	---

Добавление элемента

37

$$37 \% 11 = 4$$

X	X	X	X	37	X	X	X	52	X	X
---	---	---	---	----	---	---	---	----	---	---

Пример хэш-таблицы

X	X	X	X	37	X	X	X	52	X	X
---	---	---	---	----	---	---	---	----	---	---

Поиск элемента

16

$$16 \% 11 = 5$$

Не найден

X	X	X	X	37	X	X	X	52	X	X
---	---	---	---	----	---	---	---	----	---	---

Поиск элемента

19

$$19 \% 11 = 8$$

Не найден

Пример хэш-таблицы

X	X	X	X	37	X	X	X	52	X	X
---	---	---	---	----	---	---	---	----	---	---

Поиск элемента

37

$$37 \% 11 = 4$$

Найден

Коллизии

- Мы не хотим выделять память на каждое возможное значение элемента (реально встретившихся значений обычно много меньше, чем возможных)
- Значит, возможных значений $h(x)$ меньше, чем возможных значений x
- И существуют такие x_1, x_2 , что $h(x_1)=h(x_2)$

Коллизии

- Значит, возможна ситуация, когда мы пытаемся добавить элемент, а место занято.
- Эта ситуация называется **коллизией**
- Вернемся к примеру

Пример коллизии

X	X	X	X	37	X	X	X	52	X	X
---	---	---	---	----	---	---	---	----	---	---

Добавление элемента

96

$$96 \% 11 = 8$$

Коллизия

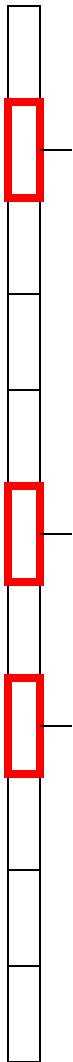
Необходимо разрешение КОЛЛИЗИЙ

- Правила разрешения коллизий должны определять, что делать при коллизии (куда поместить полученный элемент)
- Важно обеспечить, чтобы:
 - Правила разрешения коллизий позволяли бы разместить в контейнере любой набор значений
 - Правила поиска позволяли найти любой элемент, размещенный по правилам разрешения коллизий

Разрешение коллизий: хранение списков

- Будем хранить в каждом элементе массива не значение, а список значений
- Новое значение добавляем в конец списка
- Поиск выполняется по списку

Разрешение коллизий: хранение списков, $h(x) = x \% 11$, добавление



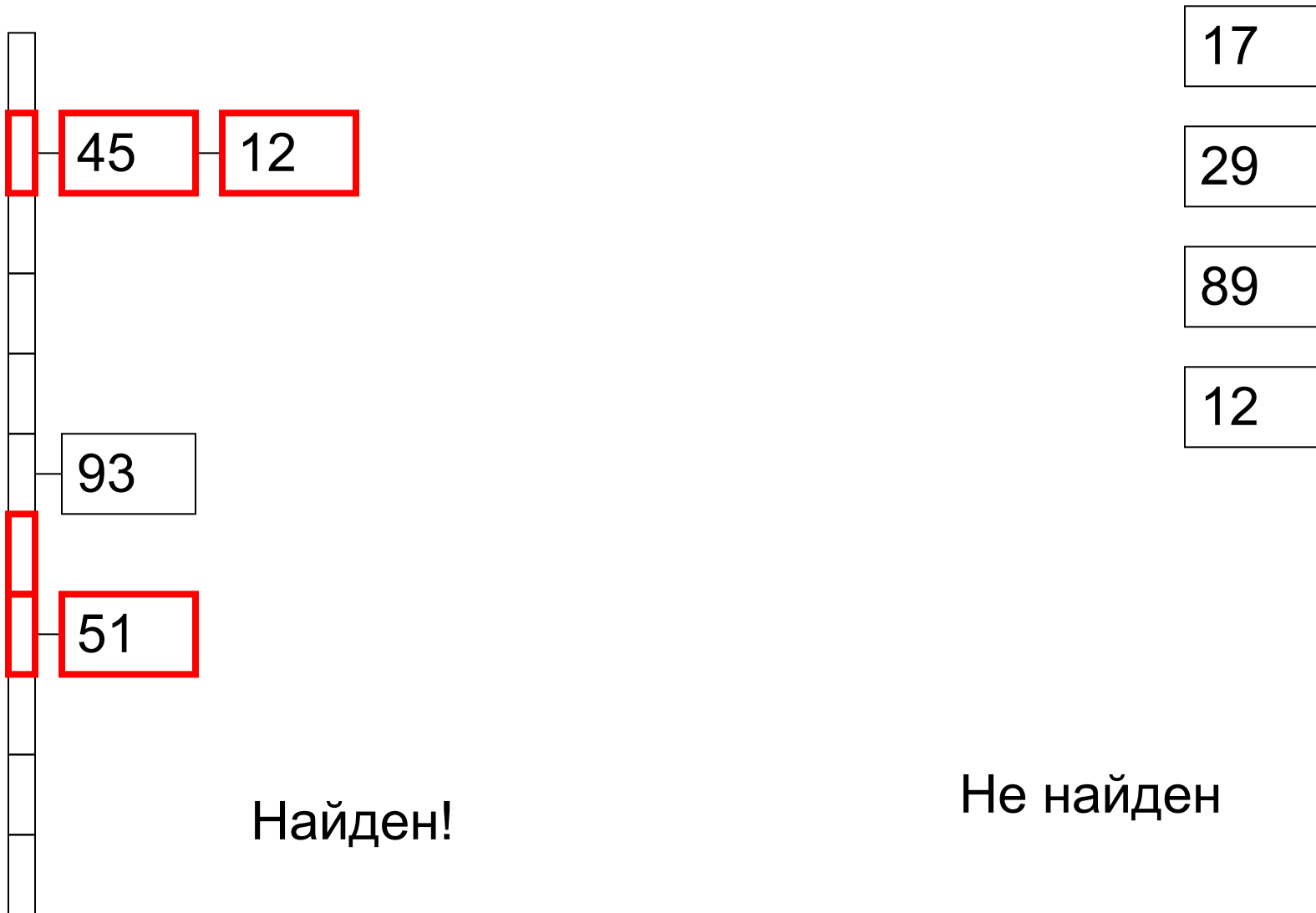
45

93

51

12

Разрешение коллизий: хранение списков, $h(x) = x \% 11$, поиск



Разрешение коллизий хранением СПИСКОВ

- В наихудшем случае время поиска $O(N)$
– если возникнет один список
- Время добавления элемента в
наихудшем случае – $O(N)$ или $O(1)$
[если хранить адрес последнего
элемента списка]

Разрешение коллизий хранением СПИСКОВ

- Предположим, что
 - Вероятности попадания элемента в любую ячейку равны
 - Количество ячеек M равно количеству элементов N (или хотя бы пропорционально)
- Тогда средняя длина списка – 1, среднее время поиска и добавления элемента – $O(1)$

Разрешение коллизий методом сдвига

- Достаточно легко удалить элемент – просто удаляем его из списка. Время удаления - $O(1)$

Разрешение коллизий методом сдвига

- Часто хочется упростить структуру и не хранить массив списков
- В этом случае можно применить разрешение коллизий методом сдвига (хэширование с открытой адресацией, метод линейного исследования)

Разрешение коллизий методом сдвига

- Если мы не можем положить элемент в нужную ячейку – пытаемся положить в следующую, и так пока не найдется свободная
- При поиске перебираем элементы, пока не встретим пустую ячейку
- Встретив конец массива – переходим на первый элемент

Почему линейное исследование?

- При попытке № i поместить значение k мы пробуем ячейку $h(k, i)$
- $h(k, i) = (h'(k) + i) \% m$
- Функция - линейная

Разрешение коллизий методом сдвига , $h(x) = x \% 11$, добавление



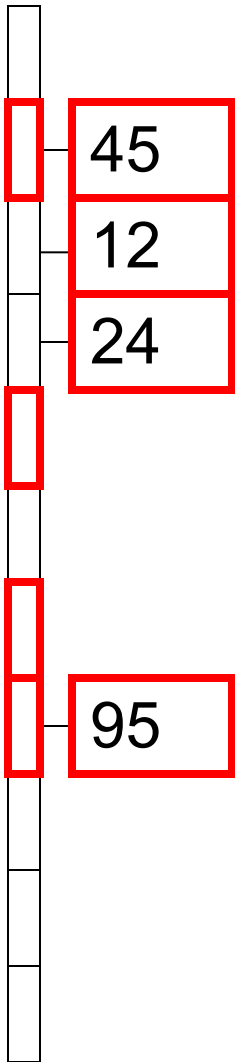
Разрешение коллизий методом сдвига , $h(x) = x \% 11$, поиск

17

95

12

89



Найден!

Не найден

Разрешение коллизий методом сдвига

- Метод работает, только если длина массива не меньше числа элементов
- Когда элементов в массиве становится достаточно много, эффективность хэширования мала (приходится перебирать множество элементов)
- Этот эффект называется **кластеризацией** (возникает кластер из занятых элементов)

Разрешение коллизий: квадратичное исследование

- При попытке № i поместить значение k мы пробуем ячейку $h(k, i)$
- $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \% m$
- В отличие от линейного исследования, кластеризация слабее

Квадратичное исследование,

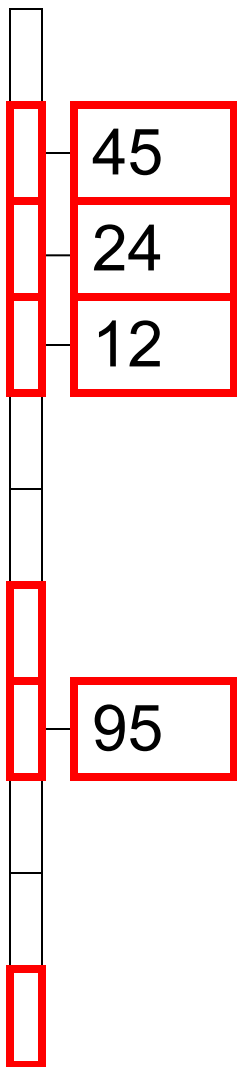
$$h(x, i) = (x \% 11 + i + i^2) \% 11)$$

17

95

12

89



Найден!

Не найден

Квадратичное исследование, $h(x, i) = (x \% 11 + i + i^2) \% 11$

45

- $45 \% 11 = 1$
- $(45 + 1 + 1) \% 11 = 3$
- $(45 + 2 + 4) \% 11 = 7$
- $(45 + 3 + 9) \% 11 = 2$
- $(45 + 4 + 16) \% 11 = 10$
- $(45 + 5 + 25) \% 11 = 9$
- $(45 + 6 + 36) \% 11 = 10$, повторная попытка



Квадратичное исследование, $h(x, i) = (x \% 8 + i / 2 + i^2 / 2) \% 8$

45



- $45 \% 8 = 5$
- $(45 + 1 / 2 + 1 / 2) \% 8 = 6$
- $(45 + 2 / 2 + 4 / 2) \% 8 = 0$
- $(45 + 3 / 2 + 9 / 2) \% 8 = 3$
- $(45 + 4 / 2 + 16 / 2) \% 8 = 7$
- $(45 + 5 / 2 + 25 / 2) \% 8 = 4$
- $(45 + 6 / 2 + 36 / 2) \% 8 = 2$
- $(45 + 7 / 2 + 49 / 2) \% 8 = 1$

Выводы:

- Квадратичное исследование менее подвержено опасности кластеризации, чем линейное.
- При квадратичном исследовании важен выбор функции так, чтобы перебрать все ячейки.
- Докажите, что при выборе функции вида $(h(x) + i/2 + i^2/2) \% 2^m$, мы попробуем все ячейки (от 0 до $2^m - 1$).

Двойное хэширование

- Методы линейного и квадратичного исследования неприемлемы при большом числе коллизий
- Если мы добавляем N элементов с одинаковым значением хэш-функции, то для последнего элемента придется сделать N попыток его размещения
- Эту проблему может решить метод **двойного хэширования**

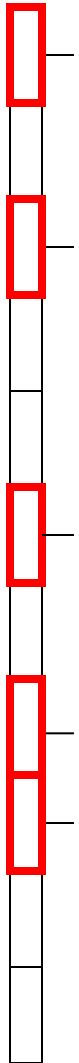
Двойное хэширование

- Идея двойного хэширования в том, чтобы использовать вторую хэш-функцию для определения смещения
- $h(k, i) = (h_1(k) + ih_2(k)) \bmod m$
- Важно, чтобы для любого k $h_2(k)$ было взаимно простым с m

Варианты:

- m – степень двойки
- $h_2(k)$ – нечетная для любого k , $h_2(k) = 2h_3(k) + 1$
- m – простое число
- $h_2(k)$ строго меньше m , например
- $h_1(k) = k \% m$
- $h_2(k) = 1 + (k \% m - 1)$

Двойное хэширование, $h_1(x) = x$
 $\% 11$, $h_2(x) = 1 + x \% 10$



95

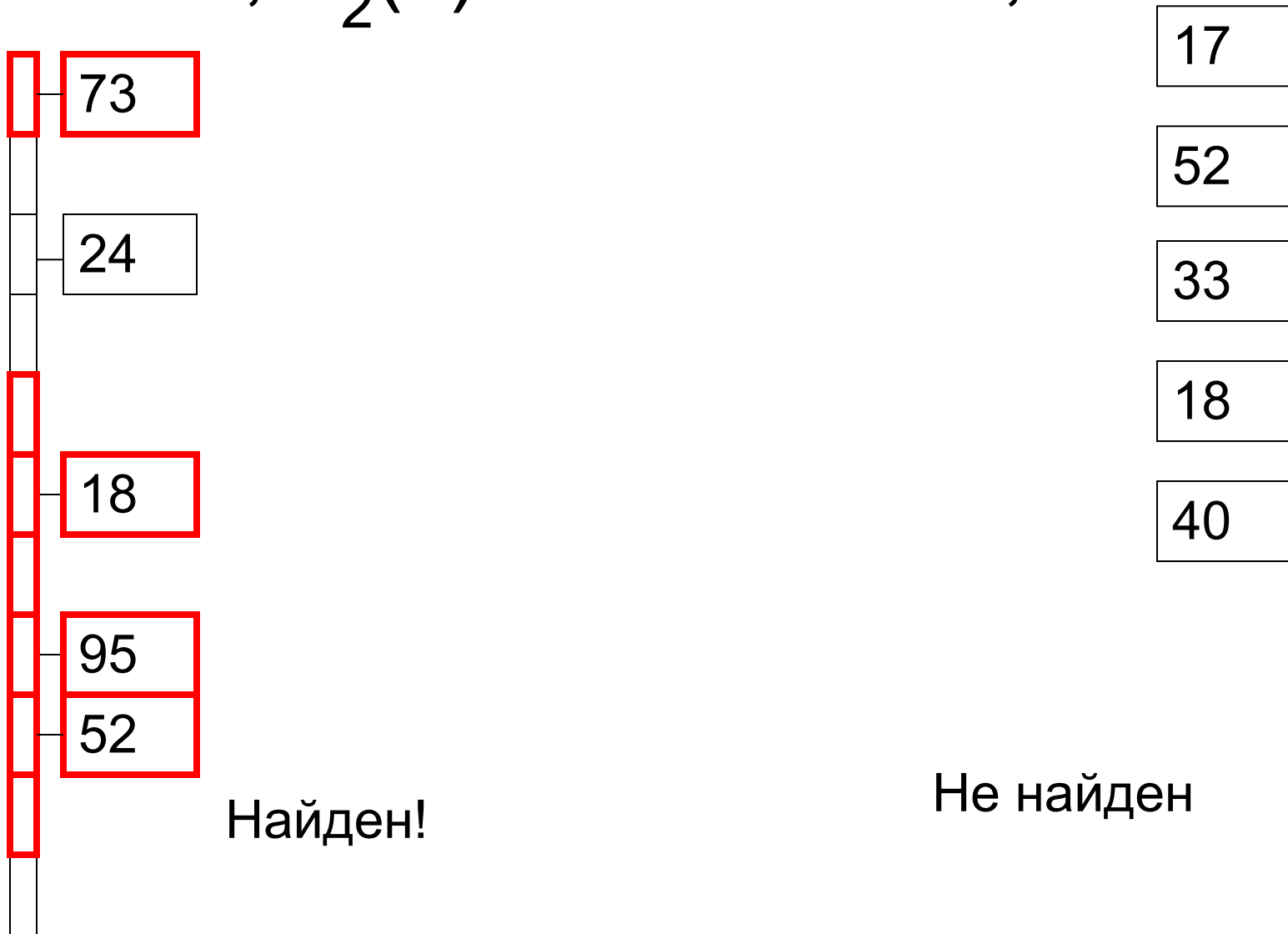
18

24

52

73

Двойное хэширование, $h_1(x) = x \% 11$, $h_2(x) = 1 + x \% 10$, поиск



Двойное хэширование: выводы

- Двойное хэширование – лучший из методов с открытой адресацией (т.е. с хранением значений непосредственно в массиве)

Удаление элементов из хэш-таблицы с открытой адресацией $h_1(x) = x \% 11$, $h_2(x) = 1 + x \% 10$



Удаление элементов

- Просто удалить элемент нельзя – нарушится поиск тех, которые были добавлены после него
- Можно заменить значение на пометку Deleted

Удаление элементов из хэш-таблицы с открытой адресацией $h_1(x) = x \% 11$, $h_2(x) = 1 + x \% 10$



Удаление элементов

- Специальное значение Deleted позволяет удалить элемент
- Но позиция в таблице после этого остается занятой и замедляет поиск
- Этот подход годится, если потребность удалить элемент возникает в результате крайне экзотической ситуации
- Если действительно нужно удалять – используйте разрешение коллизий методом списков

Выбор хэш-функции

- Мы будем считать, что элементы массива – целые числа
- Если они не целые числа – их всегда можно сделать целыми (возможно, очень большими)
- Приведем примеры

Пример: строки ANSI

- «Аlexey»
- В памяти -

65('A')	108('l')	101('e')	120('x')	101('e')	121('y')	0
---------	----------	----------	----------	----------	----------	---

- В числовой форме – 71933814662521
 $121+101*256+120*256^2+101*256^3$
 $+108*256^4+65*256^5$

Варианты хэш-функции

- Метод деления
- Метод умножения
- Универсальное хэширование

Метод деления

- $h(k) = k \% m$
- m – число позиций в хэш-таблице
- Преимущество – простота
- Недостаток – ограничения на величину m (нежелательна степень двойки – тогда на позицию влияют только младшие биты числа)
- Оптимально – простое число, далекое от степени двойки

Метод умножения

- $h(k) = [m(kA - [kA])]$
- $[x]$ – целая часть x
- Кнут предложил $A = (\sqrt{5} - 1) / 2$
- Можно избежать вещественных вычислений.

Метод умножения

- Можно избежать вещественных вычислений. $m=2^w$, $A=s/2^w$, $0 < s < 2^w$
- $h(k) = [m(kA - [kA])] = [(ks - 2^w[ks / 2^w])] = ks \% 2^w$
- И происходит только одно умножение и 1 деление на степень 2 (очень быстрое)

Универсальное хэширование

- Ясно, что для любой хэш-функции можно подобрать значения, при которых она работает плохо (коллизии на каждом шаге).
- Злоумышленник может посылать нам такие значения и спровоцировать неработоспособность нашей программы.

Универсальное хэширование

- Идея универсального хэширования – случайный выбор хэш-функции так, чтобы для любой сгенерированной злоумышленником последовательности вероятность проблем была мала

Универсальное хэширование

- Множество N хэш-функций $h_n(k)$ универсально, если для любых ключей k, l существует не больше N/m таких i , что $h_i(k) = h_i(l)$
- Т.е. для любой пары ключей вероятность коллизии не больше, чем вероятность совпадения двух случайных значений

Универсальное хэширование

- Пример функции
- Пусть p – простое число, ключи – от 0 до $p - 1$
- m – размер таблицы, $h(k)$ – от 0 до $m - 1$
- Рассмотрим семейство функций вида
- $h_{a,b}(k) = ((ak + b) \bmod p) \bmod m$
 $a = \{ 1, \dots, p - 1 \}, b = \{ 0, \dots, p - 1 \}$
- Оно является универсальным

Другие применения хэш-функций

- Криптография.
 - Криптография с закрытым ключом – зная ключ, можно построить хэш-функции для шифрования и расшифровки
 - Криптография с открытым ключом. Кто угодно может зашифровать сообщение открытым ключом, а для расшифровки нужно знать секретный закрытый
 - Электронная цифровая подпись. Кто угодно может открытым ключом расшифровать сообщение, а зашифровать – нужно знать закрытый. Если расшифровалось – значит, автор знает закрытый ключ

Лабораторная работа №2. Реализация контейнеров данных.

Реализация контейнеров данных

- Предлагаются индивидуальные варианты заданий, связанные с реализацией контейнеров
- Предпочтительна реализация контейнера, сопровождаемая подготовкой доклада об контейнере
- Доклады целесообразны для контейнеров повышенной сложности

Варианты заданий

- Реализовать класс списка с операциями добавления элемента, удаления элемента, доступа к первому элементу, доступа к следующему за данным. ([1], раздел 10.2)
- Реализовать класс бинарного дерева с операциями поиска, добавления и удаления элемента. ([1], раздел 12)
- Реализовать класс ассоциативного массива. ([1], раздел 12)

Варианты заданий

- Реализовать класс массива элементов, значение которых может быть 0 или 1, с выделением 1 бита на каждый элемент (т.е. если мы храним 32 элемента – внутри должна лежать одна переменная типа int).
- Реализовать класс стека с операциями добавления элемента, удаления элемента, доступа к первому элементу. ([1], раздел 10.1)
- Реализовать класс очереди с операциями добавления элемента, удаления элемента, доступа к первому элементу. ([1], раздел 10.1)

Варианты заданий повышенной сложности

- Реализовать класс AVL-дерева с операциями добавления элемента, удаления элемента, доступа к первому элементу ([1], раздел 13, задача 13-3)
- Красно-черное дерево с операциями добавления элемента, удаления элемента, доступа к первому элементу ([1], раздел 13)
- Реализовать класс очереди с приоритетами на базе пирамиды с операциями добавления элемента, извлечения очередного элемента ([1], раздел 6.5).

Тема 2.1. Библиотека STL как
пример стандартной библиотеки
языка программирования.
Использование контейнеров и
алгоритмов STL.

Лекция 5. Шаблоны и пространства имен в C++

Шаблоны

- Рассмотрим функцию сортировки массива целых чисел и функцию сортировки телефонной книги (программа Sort).
- Они очень похожи. Но объединить их в одну функцию мы не можем – разные типы параметров.

Шаблоны

- Для решения этой проблемы придуманы шаблоны.
- Шаблон – это «заготовка» функции, которая может быть конкретизирована несколькими способами. Например, заготовка функции сортировки в `SortTemplates`

SortTemplates

- Мы определили заготовку функции сортировки для произвольного типа.
- Когда компилятор видит попытку вызова Sort для массива целого типа, он генерирует функцию, в которой вместо T подставлено int, включает ее в программу и вызывает ее.
- Потом компилятор видит Sort для TelephoneRecord, генерирует из заготовки еще одну функцию, и включает ее в программу.

Шаблоны

- Параметром шаблона может быть не только тип данных, но и число (режим работы функции)
- Пример работы – функция `Print` в `SortTemplates`.

Синтаксис определения функции-шаблона

```
template < параметры шаблона >  
    имя функции( параметры функции)  
{  
    тело функции  
}
```

Для параметра шаблона указывается его тип (int, typename, class – что должно быть параметром) и имя.

Имя параметра шаблона может использоваться в списке параметров функции и в теле функции.

Вопрос

- Медленнее ли работа шаблона, чем работа нормальной функции?

Ответ

- Нет, не медленнее – это механизм уровня компиляции. Еще при сборке шаблон заменяется на несколько обычных функций, и вызов функции-шаблона заменяется на вызов одной из них.

Шаблоны классов

- Точно так же, как функция, шаблоном может быть и класс.
- Шаблоны классов часто используются для классов векторов и других подобных объектов, работающих с произвольным типом данных (например, `int`, `float`, `double`, `TComplex_` - для вектора).

Синтаксис определения класса-шаблона

```
template <параметры шаблона> class имя  
{  
    //Определение класса. В нем могут  
    //использоваться параметры шаблона  
    ...  
};
```

```
template <параметры шаблона>  
имя класса<параметры шаблона>::  
    имя метода (параметры метода )  
{  
    ...  
}
```

Пример шаблона класса

- Класс комплексного числа,
работающего с типами `double`, `float` -
`ComplexTemplate`

Задание

- Написать класс вектора, который сможет работать как с вещественными, так и с комплексными числами. Также написать класс комплексного числа.

Частичная спецификация шаблона

- Предположим, некоторый класс работает одинаково для всех типов данных
- При этом для одного типа данных он работает иначе (применения обсуждаются в лекции [алгоритмы](#) При этом для одного типа данных он работает иначе (применения обсуждаются в лекции алгоритмы [STL](#))
- Хочется использовать шаблон – но как

Частичная спецификация шаблона

```
template < class T >  
class TemplateClass  
{  
};  
template <>  
class TemplateClass < int >  
{  
};
```

Пространства имен

- В большой программе велик риск, что имена классов и функций будут повторяться.
- Для борьбы с этим придуманы пространства имен (namespace).

Пространства имен. Пример

```
namespace N1
{
    class A { ...};
}
namespace N2
{
    class A { ...};
}
N1::A a1;
N2::A a2;
```

Пространства имен

- Как видно на предыдущем слайде, заключив классы в пространство имен, мы можем не бояться совпадения имен двух классов и при обращении четко указать, с каким именно классом мы работаем.
- Если разработчик класса спрятал его в пространство имен, а нам писать везде имя пространства имен не хочется, можно написать один раз

```
using namespace N1;
```

Тогда после этой строчки можно к классам и функциям из N1 обращаться просто по имени, без N1::

Лекция 6. Контейнеры STL – общие принципы

Основные контейнеры

- `vector` – массив
- `list` – список
- `valarray` – вектор (массив с арифметическими операциями)
- `set` – упорядоченное множество.
- `map` – ассоциативный массив

Требования к реализации контейнеров

- Независимость реализации контейнера от типа используемых данных (могут предъявляться минимальные требования к типу – наличие копирования и проверки на равенство)
- Возможность одновременной работы с контейнером из нескольких потоков

Требования к реализации контейнеров

- Возможность единообразной реализации операций (например, перебора) для нескольких контейнеров
- Константность логически константных методов контейнера
- Независимость от используемых механизмов оперативной памяти
- Возможность хранения данных одного типа с сортировкой по разным критериям (для пирамид и деревьев поиска)

Решения

- Для обеспечения независимости от типа элемента используем шаблоны C++
- Для обеспечения независимости контейнера от конкретного способа выделения памяти передаем контейнеру объект-**аллокатор**, отвечающий за выделение и освобождение памяти (контейнер не использует new-delete, malloc-free). Существует аллокатор по умолчанию, работающий через new-delete.

Решения

- Для возможности сортировки данных одного типа по разным критериям контейнер не использует оператор сравнения у объекта (т.е. нигде в реализации контейнера нет кода `if (a<b)`). Вместо этого для сравнения используется специальный объект-компаратор.

Решения

- Для обеспечения константности логически константных операций, устойчивости к многопоточности и возможности единообразной работы с несколькими контейнерами вводим понятие **итератора**.

Итераторы

- Итератором называется программный объект со следующими свойствами
 - Объект связан с определенным объектом-контейнером и указывает на конкретный элемент этого контейнера.
 - У объекта можно вызвать оператор++ и он станет указывать на следующий элемент того же контейнера.
 - Если ++ вызывается у итератора, указывающего на последний элемент, он переходит в состояние «ни на что не указывающего итератора» и мы можем проверить, находится ли итератор в этом состоянии

Итераторы

- Каждому типу контейнера соответствует свой тип итератора. Для контейнеров STL этот тип можно получить как `ContainerType::iterator` (например, `std::vector<int>::iterator`).

Итераторы. Контрольный массив

- Есть массив в стиле C

```
int a[100];
```

- Существует ли итератор у этого контейнера?

Итераторы. Контрольный вопрос.

- Да! Это переменная типа `int*`, указывающая на любой его элемент.
 - Указывает на элемент контейнера
 - Переходит к следующему элементу вызовом `++`.
 - Если элементы закончились – переходит в невалидное состояние. Можно проверить состояние
- ```
if (ptr < a + 100)
```

# Простейшее применение итераторов

- Практически все контейнеры STL имеют
  - Метод `begin()` – возвращает итератор, указывающий на первый элемент
  - Метод `end()` – возвращает итератор, указывающий на элемент, следующий за последним.
- Пусть есть контейнер STL типа `A` с элементами типа `T`. Необходимо распечатать все элементы контейнера

# Простейшее применение итераторов

```
void Print (T element)
```

```
void PrintAll(A container)
```

```
{
 for (A::iterator iter = container.begin() ;
 iter != container.end() ;
 iter++)
 {
 Print (*iter);
 }
}
```

# Простейшее применение итераторов

- Код работоспособен для любого контейнера STL и любого типа элемента (если для него существует функция Print)

# Классификация итераторов

- Итератор всегда имеет оператор ++
- Кроме того, он может иметь (а может – не иметь) еще ряд операций
  - Доступ к объекту на чтение ( `A=*iter` )
  - Доступ к объекту на запись ( `*A=iter` )
  - Доступ к полям объекта ( `iter->field` )
  - Методы итерации ( `iter--`, `iter+=N`, `iter -=N` )
  - Сравнение на равенство ( `iter1 == iter2`, `iter1 != iter2` )
  - Сравнение на неравенство ( `iter1 < iter2` )

# Классификация итераторов

- Мы хотим иметь возможность применять итераторы для чтения данных из потока ввода (например, из файла). Мы можем создать итератор файла целых чисел

```
std::ifstream file_in("in.txt");
```

```
std::istream_iterator< int > iter_in (file_in);
```

- У такого оператора есть только две операции – итерация (++) и доступ к элементу на чтение
- Это **итератор чтения**



# Классификация итераторов

- Мы хотим использовать итераторы для записи данных в файл.

```
std::ofstream file_out("out.txt");
```

```
std::ostream_iterator< int > iter_out (file_out);
```

- У такого итератора две операции – доступ на запись и переход к следующему элементу.
- **Это итератор записи**

# Классификация итераторов

- Любой итератор контейнера имеет
  - Операцию доступа к объекту на чтение
  - Операцию доступа к объекту на запись
  - Операцию доступа к полям объекта
  - Операцию сравнения на равенство
  - Операцию ++
- Если набор операций ограничивается этим, итератор называется **однонаправленным итератором**
- Например, однонаправленным является итератор однонаправленного списка

# Классификация итераторов

- Если к набору операций однонаправленного итератора добавить операцию – (переход к предыдущему элементу), мы получим **двунаправленный итератор**
- Двунаправленный итератор реализуется для бинарных деревьев поиска, словарей, двунаправленных СПИСКОВ

# Классификация итераторов

- Если к набору операций двунаправленного итератора добавить возможность сдвига на  $N$  позиций вперед или назад по контейнеру и возможность сравнения на неравенство, мы получим **итератор с произвольным доступом**
- Итератор с произвольным доступом реализуется для массива, двусторонней очереди

# Вопрос

- Ясно, что технически возможно реализовать сдвиг по списку или бинарному дереву поиска на  $N$  позиций вперед или назад
- Почему для них не реализуется итератор с произвольным доступом?

# Ответ

- Сдвиг на  $N$  позиций работал бы за время  $O(N)$  для списка и бинарного дерева
- Пользователь привык к тому, что для массива сдвиг работает за время  $O(1)$
- Не следует вводить его в заблуждение
- Смещение на  $N$  реализуется как метод итераторов только для контейнеров, для которых оно работает за время  $O(1)$ .

# Классификация итераторов

|                     | Итератор записи | Итератор чтения | Однонаправленный итератор | Двунаправленный итератор              | Итератор произвольным доступом |
|---------------------|-----------------|-----------------|---------------------------|---------------------------------------|--------------------------------|
| Доступ к полям      |                 | ->              | ->                        | ->                                    | ->, []                         |
| Чтение              |                 | ==*p            | ==*p                      | ==*p                                  | ==*p                           |
| Запись              | *p=             |                 | *p=                       | *p=                                   | *p=                            |
| Итерация            | ++              | ++              | ++                        | ++, --                                | ++, --, +, -, +=, -=           |
| Сравнение           |                 | ==, !=          | ==, !=                    | ==, !=                                | ==, !=, <, >, <=, >=           |
| Примеры контейнеров | Поток вывода    | Поток ввода     | Однонаправленный список   | Двунаправленный список, дерево поиска | Массив                         |

# Компараторы

- Вспомним алгоритм сортировки пузырьком

```
void sort (T* A , int N)
{
 for (i = 0 ; i < N - 1 ; i++)
 for (j = 0 ; j < N - i ; j++)
 if (A[j] < A[j+1])
 {
 swap (A[j] , A[j + 1]);
 }
}
```



# Компараторы

- Мы можем применить этот алгоритм для любого типа, имеющего оператор сравнения
- Предположим, у нас есть два массива элементов одного типа  $T$  –  $A$  и  $B$ .
- Мы хотим отсортировать их по разным критериям (список студентов по алфавиту и по успеваемости)

# Компараторы

- Использовать приведенный выше код мы не сможем
- Что делать?

# Компараторы

- Мы должны передать критерий сортировки как параметр функции или параметр шаблона
- Значит, критерий сортировки может быть либо типом, либо объектом
- Можно разрешить критерию сортировки быть и типом, и объектом

# Компараторы

```
template < class TComparator >
void sort (T* A , int N , TComparator comparator)
{
 for (i = 0 ; i < N - 1 ; i++)
 for (j = 0 ; j < N - i ; j++)
 if (comparator (A[j] , A[j+1]))
 {
 swap (A[j] , A[j + 1]);
 }
}
```

# Компараторы

```
class UsualComparator
{
 bool operator()(T a , T b)
 {
 return a < b;
 }
};
```

```
T a[50];
sort (a , 50 , UsualComparator());
```

# Компараторы

- Код на предыдущем слайде приводит к обычной сортировке с использованием оператора сравнения.
- В функцию `sort` в качестве третьего параметра придет созданный конструктором по умолчанию объект `UsualComparator`
- При необходимости сравнить два элемента массива они будут передаваться методу `operator()` этого объекта и сравниваться обычным образом

# Компараторы

- Мы можем реализовать другие типы компараторов и создать другие объекты компараторы
- Передавая их в качестве параметров функции, мы настраиваем используемый функцией метод сравнения.

# Компараторы

- Компаратор можно передать и контейнеру, нуждающемуся в упорядочении своих элементов (неубывающей пирамиде, дереву поиска, словарю).
- Все контейнеры STL могут использовать компараторы.
- Компаратор по умолчанию – `std::less`, использует обычное сравнение (реализован примерно как приведенный выше `Usual Comparator`)



# Аллокаторы

- Компараторы позволяют настроить метод сравнения объекта
- Аналогично аллокаторы позволяют настроить метод выделения и освобождения памяти для хранения объектов.

# Лекция 7. Контейнеры STL - реализация

# Массивы в STL - `std::vector`

- Реализует массив
- Тип элемента задается как параметр шаблона.
- Тип элемента должен иметь конструктор по умолчанию и конструктор копирования
- Есть доступ по индексу с естественным синтаксисом за время  $O(1)$

```
vector a;
```

```
...
```

```
a[i]=3;
```

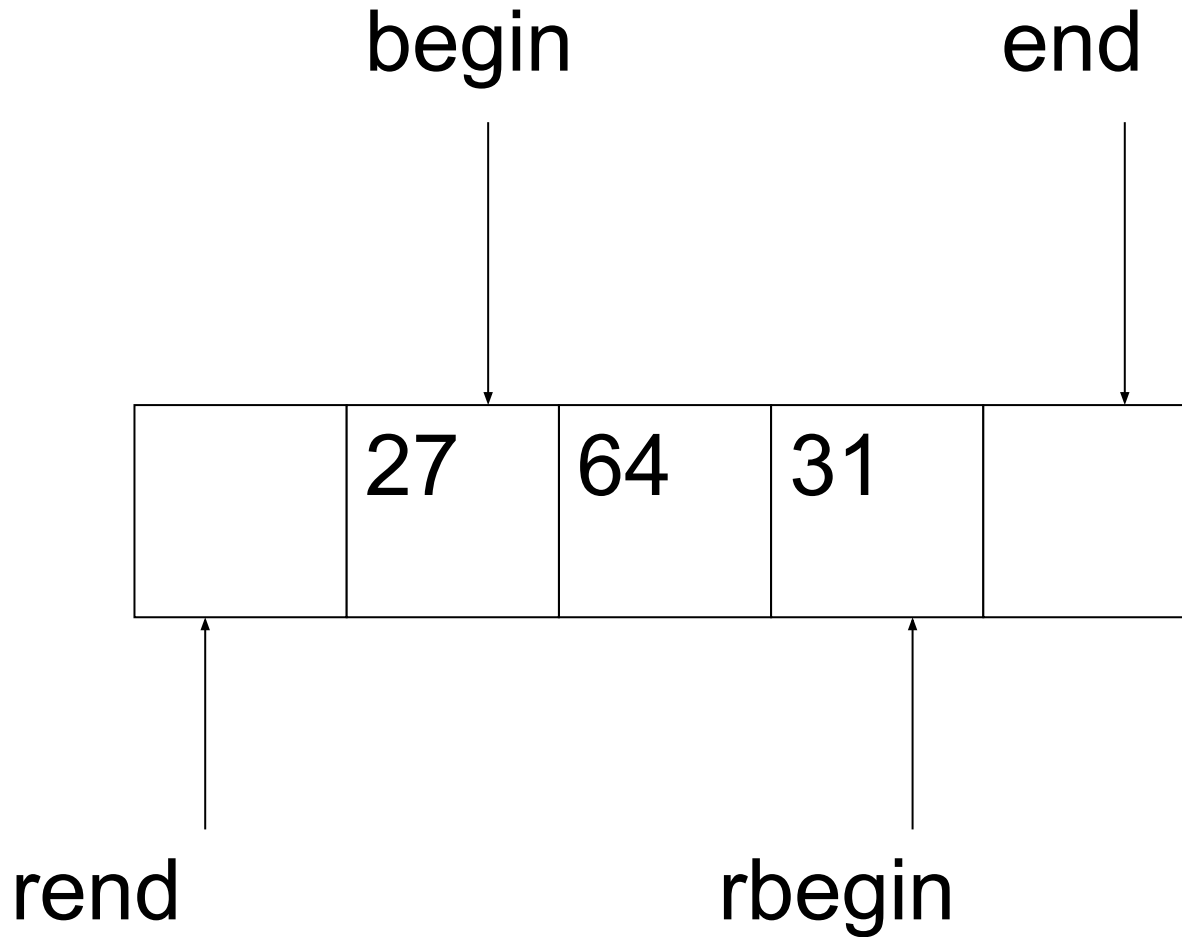
# Массивы в STL - `std::vector`

- Метод `at` – доступ по индексу с проверкой корректности, также за время  $O(1)$
- Методы `front()`, `back()` предоставляют доступ к первому и последнему элементу контейнера за время  $O(1)$ .
- Методы `push_back`, `pop_back` позволяют добавлять и удалять последний элемент в среднем за время  $O(1)$ . Работа `push_back()` в наихудшем случае медленнее из-за необходимости перераспределения памяти.

# Массивы в STL - `std::vector`

- `std::vector` определяет тип итератора `std::vector<T>::iterator`. Этот итератор является итератором с произвольным доступом и имеет полный набор операций, характерных для итератора с произвольным доступом.
- Вектор определяет константный итератор, итератор с обратным порядком и константный итератор с обратным порядком.
- Вектор имеет функции `begin()`, `end()`, `rbegin()`, `rend()` для доступа к началу и концу последовательности при прямой и обратной итерации.

# Массивы в STL - `std::vector`



# Массивы в STL - `std::vector`

- Для размещения элементов в памяти `std::vector` использует [аллокатор](#). Тип аллокатора задается вторым параметром шаблона. Ссылка на конкретный экземпляр аллокатора, который следует использовать, может быть передана в конструктор вектора. По умолчанию используется стандартный класс STL `std::allocator`.
- Операции вставки элемента после заданного элемента (`insert`) и удаления элемента (`erase`) работают за линейное время.

# Списки в STL – `std::list`

- `std::list` реализует стратегию работы со списками независимо от типа хранимых элементов. Тип элемента задается как параметр шаблона.
- Тип элемента должен иметь конструктор по умолчанию и конструктор копирования



# Списки в STL – `std::list`

- Методы `front()`, `back()` предоставляют доступ к первому и последнему элементу контейнера за время  $O(1)$ .
- Методы `push_back`, `pop_back` позволяют добавлять и удалять последний элемент за время  $O(1)$ . Аналогично работают операции `push_front`, `pop_front`

# Списки в STL – `std::list`

- `std::list` определяет тип итератора `std::list<T>::iterator`. Этот итератор является двунаправленным итератором и предоставляет соответствующий набор операций.
- Список определяет константный итератор, итератор с обратным порядком и константный итератор с обратным порядком.
- Список имеет функции `begin()`, `end()`, `rbegin()`, `rend()` для доступа к началу и концу последовательности при прямой и обратной итерации.

# Списки в STL – `std::list`

- Используются аллокаторы Используются аллокаторы так же, как в массиве.
- Операции вставки элемента в середину (после заданного элемента) и удаления элемента работают за время  $O(1)$ .
- Список определяет дополнительные операции, такие как `merge` (сортировка двух объединяемых списков), `splice` (перемещение элемента одного списка в другой без физического копирования, простой перестановкой указателей).

# Бинарное дерево поиска в STL – std::set

- std::set реализует работу с бинарным деревом поиска независимо от типа хранимых элементов. Тип элемента задается как параметр шаблона.
- Тип элемента должен иметь конструктор по умолчанию и конструктор копирования.
- Необходим компаратор. Компаратор по умолчанию std::less использует оператор сравнения.

# Бинарное дерево поиска в STL – `std::set`

- Бинарный поиск реализуется методом `find`, работает за время  $O(\log N)$
- Доступны и работают за время  $O(\log N)$  операции
  - `lower_bound` (поиск минимального элемента, больше либо равного данному)
  - `upper_bound` (поиск минимального элемента, большего данного)
  - `equal_range` (одновременный поиск `lower_bound` и `upper_bound`)

# Бинарное дерево поиска в STL – `std::set`

- Добавление элемента реализуется методом `insert`. Результатом добавления является итератор, указывающий на добавленный элемент, и флаг, говорящий об успехе добавления.
- Для возврата двух значений используется `std::pair`
- Удаление элемента реализуется методом `erase`

# Бинарное дерево поиска в STL – `std::set`

- `std::set` определяет тип итератора `std::set<T>::iterator`. Этот итератор является двунаправленным итератором и перебирает элементы в порядке возрастания.
- `std::set` определяет константный итератор, итератор с обратным порядком и константный итератор с обратным порядком.
- `std::set` имеет функции `begin()`, `end()`, `rbegin()`, `rend()`

# Бинарное дерево поиска в STL – `std::set`

- Используются аллокаторы Используются аллокаторы так же, как в массиве
- Хранить несколько одинаковых значений нельзя (`insert` вернет `false`). Если необходимо – используйте `multi_set`



# std::multi\_set

- Набор операций аналогичен std::set
- find возвращает первый элемент, равный данному
- insert возвращает только итератор. Успех добавления элемента гарантируется.

# std::multi\_set

- Перебор элементов, равных данному:

```
for (TContainer::iterator iter =
 Container.lower_bound(x);
 iter != Container.upper_bound(x);
 iter ++)
{
 ...
}
```

# Словарь в STL – `std::map`

- `std::map` реализует работу со [словарем](#), имеющим произвольный тип ключа и произвольный тип значения. Тип ключа и тип значения – два первых параметра шаблона.
- Типы ключа и значения должны иметь конструктор по умолчанию и конструктор копирования.
- Необходима реализация [компаратора](#) – объекта, обеспечивающего сравнение ключей

# Словарь в STL – `std::map`

- Методы `find`, `lower_bound`, `upper_bound`, `equal_range`, `insert`, `erase` – аналогичны [`std::set`](#)
- Доступ на чтение и запись к значению, соответствующему ключу, можно получить вот так:

```
... = Map[key]
```

```
Map[key] = ...
```

- Словарь называют ассоциативным массивом
- Если элемента с таким ключом нет – он конструируется со значением по умолчанию

# Словарь в STL – `std::map`

- `std::map` определяет тип итератора `std::map<T>::iterator`. Этот итератор является двунаправленным итератором и перебирает элементы в порядке возрастания. Итератор указывает на пару (`std::pair`) ключ-значение
- `std::map` определяет константный итератор, итератор с обратным порядком и константный итератор с обратным порядком.
- `std::map` имеет функции `begin()`, `end()`, `rbegin()`, `rend()`

# Словарь в STL – `std::map`

- Используются аллокаторы Используются аллокаторы так же, как в массиве
- Хранить несколько значений для одного ключа нельзя (`insert` вернет `false`). Если необходимо – используйте `multi_map`

# std::multi\_map

- Аналогичен [std::map](#)
- Не реализуется обращение по индексу `map[ key ]`.
- Как и в [std::multiset](#), метод `find` выдает первый (в порядке итерации) из элементов с данным ключом; `insert` возвращает не пару (итератор, флаг успеха), а только итератор

# Двусторонняя очередь – `std::deque`

- `std::deque` реализует поведение двусторонней очереди
- `std::deque` позволяет задать тип элемента как параметр шаблона.
- Тип элемента должен иметь конструктор по умолчанию и конструктор копирования, необходимые для работы с ним.



# Двусторонняя очередь – `std::deque`

- Быстрый доступ по индексу – как в `std::vector`

`Deq[ i ]`

`Deq.at( i )`

- Напомните, в чем разница?

# Двусторонняя очередь – `std::deque`

- Методы `front()`, `back()` предоставляют доступ к первому и последнему элементу контейнера за время  $O(1)$ .
- Методы `push_back`, `pop_back` позволяют добавлять и удалять последний элемент в среднем за время  $O(1)$ .
  - Работа `push_back()` в наихудшем случае медленнее из-за необходимости перевыделения памяти.
- Аналогичные операции с началом очереди – `push_front`, `pop_front()`

# Двусторонняя очередь – `std::deque`

- `std::deque` определяет тип итератора `std::deque<T>::iterator`. Этот итератор является итератором с произвольным доступом.
- Двусторонняя очередь определяет константный итератор, итератор с обратным порядком и константный итератор с обратным порядком.
- Двусторонняя очередь имеет функции `begin()`, `end()`, `rbegin()`, `rend()`

# Двусторонняя очередь – `std::deque`

- Для размещения элементов в памяти `std::deque` использует аллокатор Для размещения элементов в памяти `std::deque` использует аллокатор так же, как массив.
- Операции вставки элемента в середину (после заданного элемента) и удаления элемента работают за линейное время.

# Очередь – `std::queue`

- Реализует очередь
- Тип элемента задается как параметр шаблона.
- Необходимо существование конструктора по умолчанию и конструктора копирования для элемента.

# Очередь – `std::queue`

- Набор операций включает методы
  - `push` (добавить элемент в конец очереди)
  - `pop` (извлечь элемент из начала)
  - `front` (доступ к начальному элементу)
  - `back` (доступ к конечному элементу)
  - `size` (доступ к количеству элементов)
  - `empty`( проверка на пустоту)
- Все операции должны выполняться за время  $O(1)$ .

# Очередь – `std::queue`

- Очередь может эффективно работать при различных стратегиях размещения данных в памяти, поэтому не навязывает одну стратегию
- Для хранения своих данных `std::queue` создает контейнер какого-либо другого типа (либо использует готовый контейнер, заданный ей как параметр конструктора).

# Очередь – `std::queue`

- Тип внутреннего контейнера задается как второй параметр шаблона `std::queue`.
- Этот внутренний контейнер должен иметь операции `size()`, `back()`, `front()`, `push_back()` и `pop_front()`.
- Несложно убедиться, что из рассмотренных выше контейнеров нас устраивают [std](#) Несложно убедиться, что из рассмотренных выше контейнеров нас устраивают `std::` Несложно убедиться, что из рассмотренных выше



# Стек – `std::stack`

- Реализует [стек](#)
- Тип элемента задается как параметр шаблона.
- Необходимо существование конструктора по умолчанию и конструктора копирования для элемента.

# Стек – `std::stack`

- Набор операций включает
  - `push` (добавить элемент)
  - `pop` (извлечь последний добавленный элемент)
  - `back` (доступ к последнему добавленному элементу)
  - `size` (доступ к количеству элементов)
  - `empty`( проверка на пустоту).
- Все операции должны выполняться за время  $O(1)$ .

# Стек – `std::stack`

- Стек может быть реализован на базе различных контейнеров.
- Базовый контейнер может быть задан как параметр шаблона. От него требуется наличие методов `size()`, `push_back()`, `pop_back()`, `back()`.
- Базовым контейнером может быть `std::vector`Базовым контейнером может быть `std::vector`Базовым контейнером может быть `std::vector`Базовым контейнером может быть `std::vector`

# Очередь с приоритетами – `std::priority_queue`

- Очередь с приоритетами – это очередь, в которой элементам сопоставлен приоритет и первым в очереди считается элемент с максимальным приоритетом

# Очередь с приоритетами – `std::priority_queue`

- Тип элемента задается как первый параметр шаблона.
- Необходимо существование конструктора по умолчанию и конструктора копирования для элемента.
- Для сравнения двух элементов и проверки, какой из них больше (т.е. имеет больший приоритет) используется [компаратор](#), задаваемый как третий параметр шаблона. По умолчанию используется компаратор `std::less`

# Очередь с приоритетами – `std::priority_queue`

- Набор операций включает
  - `push` (добавить элемент)
  - `pop` (извлечь элемент с максимальным приоритетом)
  - `top` (доступ к элементу с максимальным приоритетом)
  - `size` (доступ к количеству элементов)
  - `empty`( проверка на пустоту).
- `push` и `pop` выполняются за время  $O(\log N)$ , остальные операции за время  $O(1)$ .

# Очередь с приоритетами – `std::priority_queue`

- Как реализуется очередь с приоритетами?

# Очередь с приоритетами – `std::priority_queue`

- Очередь с приоритетами строится на базе невозрастающей пирамиды
- Используется хранение пирамиды в виде массива



# Очередь с приоритетами – `std::priority_queue`

- Для хранения «пирамиды как массива» может использоваться любой контейнер, имеющий итератор с произвольным доступом, т.е. [std::vector](#)  
Для хранения «пирамиды как массива» может использоваться любой контейнер, имеющий итератор с произвольным доступом, т.е. `std::vector` или [std::deque](#)
- Тип используемого контейнера задается как параметр шаблона

# Хэш-таблица – `std::hash_map`

- Класс `std::hash_map` реализует [хэш-таблицу](#)
- Как и [std::map](#), `std::hash_map` хранит пары ключ-значение и требует уникальности ключа.
  - Если уникальность не требуется или требуется хранение только ключей существуют классы `std::hash_multimap`, `std::hash_set`, `std::hash_multiset`.
- Типы ключа и значения задаются как параметры шаблона. Должны иметь конструкторы по умолчанию и конструкторы копирования

# Хэш-таблица – `std::hash_map`

- За вычисление хэш-функции и проверки на равенство отвечает специальный объект – хэш-компаратор. Он способен как вычислять значение хэш-функции, так и проверять два значения на равенство.

# Хэш-таблица – `std::hash_map`

- Необходимый размер хэш-таблицы вычисляется и динамически меняется.
  - Задаваемая пользователем хэш-функция должна лишь вычислять требуемый индекс в диапазоне (в данный момент) от 0 до  $2^{32}-1$ .
  - Индекс особым преобразованием (зависящим от текущего размера массива) превращается в реальный индекс.
  - Естественно, при изменении размера хэш-таблицы и преобразования гарантируется сохранение доступности ранее добавленных элементов.
- Для выделения памяти используется [аллокатор](#), задаваемый как четвертый параметр шаблона.

# Не совсем контейнеры

- Существуют объекты библиотеки STL, которые не являются контейнерами но реализуют определенные возможности контейнеров
- Это строки, вектора (`valarray`), битовые массивы, потоки ввода-вывода

# Строка – `std::basic_string`

- Строка является массивом символов
- Для представления символов могут использоваться различные типы данных (`char`, `wchar_t`, `unsigned short`, ...)
- Не любой массив можно рассматривать как строку
- Строка реализуется в STL классом `std::basic_string`

# Строка как массив

- `std::basic_string` определяет тип итераторов с произвольным доступом – `std::basic_string<T>::iterator`.
- `std::basic_string` имеет методы `begin`, `end`, `rbegin`, `rend`.
- Для строки возможно обращение к символу по индексу (`operator []` и метод `at()` ).
- Существует метод `push_back()`.
- Есть возможность задания аллокаторов, используемых строкой для выделения памяти.

# Отличия строки

- `std::basic_string` требует от используемого типа символов расширенного набора операций
- См. `char_traits`
- `std::basic_string` определяет дополнительные операции, характерные для строк (выдача `null-terminated` строки `c_str`, выдача подстроки `substr`, ...)



# Вектор – `std::val_array`

- Есть доступ по индексу `[]`
- Есть метод `size`
- Реализует математические операции над векторами

# БИТОВЫЙ МАССИВ – `std::bit_set`

- Возможен доступ к биту с помощью оператора `[]`
- Дополнительно реализуются побитовые операции

# Потоки ввода-вывода и итераторы

- Основным инструментом ввода-вывода в STL являются **потоки ввода-вывода**
- Поток ввода – это объект, из которого можно прочитать значения различных типов
- Поток ввода может быть файл, строка, датчик, ввод с экрана консольного приложения
- Большинство потоков ввода в STL наследуются от `std::basic_istream`

# Потоки ввода-вывода и итераторы

- Поток вывода – это устройство, в которое можно вывести значение того или иного типа
- Это может быть экран, строка, файл,...

# Потоки ввода-вывода и итераторы

- Если мы читаем из потока или записываем в поток однотипные значения, целесообразно использовать для чтения и записи в поток итераторы.
- Для ввода данных из потока используется [итератор чтения](#)
- Для вывода данных в поток используется [итератор записи](#)

# Задание

- Напишите программу, читающую набор целых чисел из файла и записывающую их в другой файл
- Используйте итераторы чтения и записи
- Не забудьте решить проблему разделителей

# Лабораторная работа №3. Использование стандартных контейнеров данных

# Задание

- Разработать программу на языке C++, реализующую функциональность в соответствии с вариантом задания.
- Настоятельно рекомендуется использование стандартных контейнеров из библиотеки STL.



# Варианты задания

- Реализовать программу, хранящую совокупность многоугольников на плоскости и позволяющую организовать быстрый поиск многоугольников, попадающих в заданный прямоугольник
  - Необходимо обеспечить добавление многоугольника и поиск многоугольников, попадающих в прямоугольник.
  - Предложение: Храните один массив многоугольников и 4 массива или бинарных дерева номеров многоугольников, упорядоченных по самой левой, самой правой, самой верхней и самой нижней точке многоугольника.
  - Это позволит быстро отфильтровать многоугольники, лежащие заведомо выше, ниже, левее или правее данного прямоугольника, и только для оставшихся реализовывать медленные алгоритмы содержательной проверки пересечения прямоугольника.

# Варианты задания

- Реализовать программу, хранящую совокупность отрезков на плоскости и поддерживающую добавление отрезка и быстрый поиск отрезков, попадающих в прямоугольник
  - Предложение: Храните один массив отрезков и 4 массива или бинарных дерева номеров отрезков многоугольников, упорядоченных по самой левой, самой правой, самой верхней и самой нижней точке отрезка.
  - Это позволит быстро отфильтровать отрезки, лежащие заведомо выше, ниже, левее или правее данного прямоугольника, и только для оставшихся реализовывать медленные алгоритмы содержательной проверки пересечения прямоугольника.

# Варианты задания

- Реализовать программу, хранящую множество шариков, летающих в комнате, поддерживающих добавление и удаление шарика и выдающей информацию о 5 ближайших столкновениях шарика со стенкой. Движение шарика равномерное и прямолинейное, удар упругий, возможностью столкновения шариков друг с другом пренебречь. При добавлении шарика указываются его положение, скорость и время начала полета.
- В электронной картотеке библиотеки для каждой книги хранится номер зала, стеллажа и полки. При этом необходим быстрый поиск книги по фамилии автора (считаем, что автор один) и по слову из названия (падежами и т.д. пренебрегаем, считаем, что слово должно быть в названии точно таким же, как его вводит пользователь). Разработать программу электронной картотеки с операциями добавления книги и поиска.

# Варианты задания

- Реализовать систему регистрации сделок на бирже. Для каждой сделки указывается, какой товар продан, в какой день, какое количество и по какой цене. Необходимо по запросу выводить среднюю цену на данный товар в данный день.
- Реализовать систему, хранящую информацию о доходах налогоплательщиков (для каждого налогоплательщика указывается его заработок в каждом году). Система должна быть в состоянии дать отчет о доходах данного налогоплательщика в данные годы и отчет о среднем уровне дохода в каждом году.

# Варианты задания

- Реализовать программу электронного магазина, поддерживающую три операции
  - Добавление информации о появлении в продаже очередной партии товара (указывается цена, количество и наименование).
  - Покупку партии товара.
  - Формирование отчета об имеющихся на складе товарах.
- Реализовать программу, хранящую информацию о вкладчиках банка. Для каждого вкладчика указывается фамилия и номер паспорта, и для каждого из его вкладов – сумма, валюта и срок возврата. Поддерживать операции добавления и снятия вклада, отчета о всех вкладах и об отдельном вкладчике.

# Варианты задания

- Реализовать программу, которая получает результаты измерений одной и той же меняющейся величины 10 датчиками. Если больше 3 значений подряд, приходящих с одного датчика не соответствуют значениям с остальных – объявить датчик испортившимся и более не учитывать. Операции
  - Добавить результат очередных измерений (10 чисел)
  - Вывести среднее значение величины по итогам последнего измерения.
  - Вывести информацию об исправных датчиках.

# Варианты задания

- При голосовании приходят результаты в виде «На участке № такой-то такая-то партия получила столько-то голосов.» Система должна в любой момент выдать информацию о доступных результатах по данному участку и о суммарном количестве проголосовавших за партию.
- Несколько датчиков установлены в разных местах планеты и присылают свои результаты измерения температуры (указывая номер датчика, температуру и время). Необходимо по запросу пользователя выводить отчет о любом датчике (все его измерения), или данные со всех датчиков, говорящие о температуре в заданном интервале времени.

# Варианты задания

- Корабли присылают в каждый момент времени данные о своей скорости и направлении и свои координаты. Необходимо предупредить пользователя, если данные не согласованы (т.е. если изменение координат не соответствует скорости и направлению движения корабля). Землю считать плоской.
- В базу данных вводятся результаты футбольных матчей. По запросу пользователя выдать турнирную таблицу чемпионата (количество побед, ничьих, поражений, очков и разницу мячей у каждой команды)



# Варианты задания

- Завод по сборке автомобилей покупает комплекты комплектующих и производит автомобили из них. Необходимо хранить информацию о количестве комплектов на складе комплектующих и количестве готовых к отгрузке автомобилей. Основные действия – это покупка  $N$  комплектов комплектующих, производство  $N$  автомобилей, продажа  $N$  автомобилей, выдача отчета о количестве комплектующих и автомобилей на складах.
- В базе данных животных в зоопарке хранится информация о виде животного, кличке и количестве потребляемой в день еды (сколько килограммов какого продукта необходимо в неделю). Необходимо формировать отчеты о потребностях данного животного, о потребностях всех животных данного вида и сообщать о суммарной потребности в данном продукте в неделю.

# Варианты задания

- Подразделения фирмы, нуждающиеся в покупке компьютеров, вносят заказы в базу данных. Отдел закупок вносит информацию о ценах на соответствующее оборудование. Необходимо иметь возможность вывести всю информацию о потребностях каждого подразделения и о данном виде оборудования.
- Предприятие хранит базу данных о сотрудниках. Фамилия, №паспорта, должность, зарплата. Основные операции – прием на работу, увольнение, перевод на другую должность, изменение зарплаты, отчет о всех сотрудниках, выдача информации о конкретном сотруднике.

# Варианты задания

- Операционная система хранит базу данных процессов. Процесс имеет постоянный приоритет (константа, задается пользователем) и дополнительный приоритет (у каждого следующего процесса на 1 меньше, чем у предыдущего – чтобы те, кто дольше ждал, имели преимущество). Набор поддерживаемых операций:
  - Добавить процесс с данным именем и постоянным приоритетом
  - Выбрать из очереди процесс с наибольшим приоритетом (суммой постоянного и дополнительного). Он отработает и завершится.
  - Выбрать из очереди процесс с наибольшим приоритетом (суммой постоянного и дополнительного). Он отработает, после этого нужно снова поставить его в очередь (уже с новым дополнительным приоритетом).
  - Все операции должны работать за логарифмическое время.
- Указание: `priority_queue`.

# Лекция 8. Стандартные алгоритмы STL.

- Простейший стандартный алгоритм `for_each`
- Возможности применения алгоритмов на примере `for_each`
- Другие алгоритмы STL.

# std::for\_each

- Алгоритм `std::for_each` заключается в вызове заданной функции для каждого элемента контейнера
- `for_each` не делает предположений о типе контейнера – достаточно, чтобы у него был итератор чтения
- `for_each` не модифицирует перебираемые элементы

# std::for\_each - пример

```
for_each(v1.begin() , v1.end() , Print)
```

ЭКВИВАЛЕНТНО

```
for (v1::iterator iter = v1.begin() ;
```

```
 iter != v1.end() ;
```

```
 iter++)
```

```
{
```

```
 Print(*iter);
```

```
}
```

# std::for\_each

- В приведенном примере мы вызывали функцию Print, единственным параметром которой был элемент контейнера, для которого она вызывалась
- Это простейший случай
- Чаще встречаются другие ситуации

# Пример – вызов функции с несколькими параметрами

```
for (v1::iterator iter = v1.begin() ;
 iter != v1.end() ;
 iter++)
{
 Print(*iter , file);
}
```



# Пример – вызов метода класса с несколькими параметрами

```
for (v1::iterator iter = v1.begin() ;
 iter != v1.end() ;
 iter++)
{
 Processor.Process(*iter , param2);
}
```

# std::for\_each

- Ясно, что мы должны уметь применять `for_each` для таких ситуаций – иначе этот механизм бесполезен

# Шаблоны. Взаимозаменяемость классов и функций

- `for_each` – это шаблон функции.
- Шаблоны C++ являются механизмом времени компиляции.
- Это означает, что еще до компиляции происходит замена `for_each` на соответствующий код (примерно такая, как показано [выше](#))

# Шаблоны. Взаимозаменяемость классов и функций

- Но это означает, что с точки зрения `for_each` не важно, что такое `Print`
- Это может быть функция с одним параметром
- Это может быть класс, имеющий метод `operator()` с одним параметром

# Класс-функция

```
class Printer
{
public:
 Printer(std::ostream& stream)
 :Stream(stream) {}

 void operator()(int a)
 {
 Print(Stream , a);
 }
private:
 std::ostream& Stream;
};
```

# Класс-функция

- С точки зрения шаблона `for_each`, объект класса `Printer` – полный аналог функции, имеющей один параметр.
- И мы можем дать указание `for_each` вызвать этот объект (т.е. его метод `operator()`) для всех элементов контейнера

# Класс-функция

```
Printer printer(stream1);
std::for_each(v1.begin() , v1.end() , printer);
ЭКВИВАЛЕНТНО
for (v1::iterator iter = v1.begin() ;
 iter != v1.end() ;
 iter++)
{
 printer(*iter); //или printer.operator()(*iter)
}
```

# Класс-функция

- И это уже эквивалентно

```
for (v1::iterator iter = v1.begin() ;
 iter != v1.end() ;
 iter++)
{
 Print(*iter , stream1);
}
```



# Вызов метода класса

```
for (v1::iterator iter = v1.begin() ;
 iter != v1.end() ;
 iter++)
{
 processor.Process(*iter);
}
```

# Вызов метода класса

```
class ProcessorAdapter
{
public:
 ProcessorAdapter (Processor& processor)
 :Proc(processor) {}
 void operator()(int cur)
 {
 Proc.Process(cur);
 }
private:
 Processor& Proc;
};
```

# Вызов метода класса

```
ProcessorAdapter adapter(processor);
std::for_each(int_vector.begin() , int_vector.end() ,
 adapter);
```

ЭКВИВАЛЕНТНО

```
for (v1::iterator iter = v1.begin() ;
 iter != v1.end() ;
 iter++)
{
 adapter.operator()(*iter);
}
```

# Возвращаемое значение `for_each`

- Функция `for_each` возвращает тот объект, метод `operator()` которого она вызвала для всех элементов контейнера
- Это означает, что если вызов метода приводил к изменению состояния объекта, то измененное состояние нам доступно

# Задание

- Реализуйте поиск максимума массива вещественных чисел через `for_each`

# Решение

```
class MaxSearch
{
public:
 MaxSearch(double first)
 :CurMax(first) {}
 void operator|(double cur)
 {
 if (cur > CurMax)
 CurMax= cur;
 }
 double GetMax()
 {
 return CurMax;
 }
private:
 double CurMax;
};
```

# Решение

```
std::vector < double > double _vector;
```

```
...
```

```
MaxSearch search(*double _vector.begin());
```

```
search = std::for_each(double_vector.begin() ,
 double_vector.end() ,
 search);
```

```
double max = search.GetMax();
```

# Вызовы функций с параметрами – ГОТОВЫЕ МЕХАНИЗМЫ

Если мы хотим вызвать для всех методов контейнера функцию

```
void Print (std::istream& stream , int a)
{
 stream << a << " ";
},
```

мы можем просто написать:

```
std::for_each(v1.begin(), v1.end(),
 std::bind1st(Print , stream1));
```



# Вызовы функций с параметрами – ГОТОВЫЕ МЕХАНИЗМЫ

- Другие готовые механизмы для вызова функций и методов классов из `for_each` есть в библиотеке Boost (`boost::bind`)

# Методы поиска

- Все методы принимают два итератора (указывающие на начало последовательности и на следующий за последним элемент)
- Возвращают итератор, указывающий на найденный элемент (или на следующий за последним, если элемент не найден)

# Методы поиска

- `find` – поиск равного данному
- `find_if` – поиск соответствующего условию
- `find_first_of` – поиск в первой последовательности первого символа, присутствующего во второй (задается компаратор)
- `adjacent_find` – поиск двух равных последовательных символов (задается компаратор)

# Задание

- Как найти первый символ, больший квадрата предыдущего?
- Предложите два метода

# Поиск нарушения порядка в массиве в стиле C

```
bool Test(double a , double b)
{
 return a > b;
}
```

...

```
double array[4]={3,5,35,27};
```

...

```
double* ptr = std::adjacent_find(array , array+4 ,
 Test);
```

# Методы подсчета

- `count` – подсчет элементов, равных данному
- `count_if` – подсчет количества элементов, соответствующих условию
- Входные параметры – два итератора и условие для `count_if`
- Возвращаемое значение?

# Методы подсчета

- Фиксированный тип – не годится.

Вариант:

```
template < class TIterator , class TValue >
TIterator::difference_type count(
 TIterator begin ,
 TIterator end ,
 TValue value)
```

# Методы подсчета

- В данном случае в классе TIterator должно быть что-то вроде

```
class TIterator
```

```
{
```

```
public:
```

```
 typedef int difference_type;
```

```
};
```

Ваша оценка решения?



# Методы подсчета

- В этом случае мы не сможем использовать указатель как итератор массива в стиле C и нам не удастся написать

```
int A[5];
```

```
...
```

```
int n = std::count(A , A+5 , 3);
```

# Методы подсчета - решение

Определим шаблонный класс  
iterator\_traits вида

```
template < class TIterator >
class iterator_traits
{
 typedef TIterator::difference_type
 difference_type;
};
```

# Методы подсчета - решение

```
template < class TIterator , class TValue >
iterator_traits<TIterator>::difference_type
count(
 TIterator begin ,
 TIterator end ,
 TValue value)
```

# Методы подсчета - решение

- Внешне кажется, что ничего не изменилось

`iterator_traits<TIterator>::difference_type`

это эквивалент

`TIterator::difference_type`

- Но теперь пользователь может воспользоваться частичной спецификацией шаблонов

# Методы подсчета - решение

- Определим частичную спецификацию шаблона `iterator_traits` вида

```
template< >
class iterator_traits<int*>
{
 typedef int difference_type;
};
```

# Методы подсчета - решение

```
int A[5];
```

```
...
```

```
int n = std::count(A , A+5 , 3);
```

A и A+5 имеет тип int\*

Поэтому тип возвращаемого значения –  
Iterator\_traits<int\*>::difference\_type

# Минимумы и максимумы

- `max_element` и `min_element` ищут максимальный или минимальный элемент последовательности
- Принимают итераторы, указывающие на начало и конец, и функцию сравнения (или объект-[компаратор](#))

# Сравнение последовательностей

- `equal` – проверка на равенство
- `mismatch` – поиск первого различия
- `lexicographical_compare`
- Задается объект-компаратор
- Типы элементов могут различаться.



# Сравнение последовательностей

- В одном массиве строки, в другом числа
- Нужно проверить, что длина строки номер  $i$  в первом массиве равна числу номер  $i$  во втором

# Подпоследовательности

- `search` - поиск первого вхождения подпоследовательности в последовательность. Задаются 4 итератора и компаратор
- `find_end` - поиск последнего вхождения подпоследовательности в последовательность. Задаются 4 итератора и компаратор
- `search_n` – поиск в последовательности идущих подряд  $n$  чисел, равных данному. Задаются два итератора, значение и компаратор

# Задание

- Предложите два способа поиска трех нечетных чисел подряд – с помощью `search` и `search_n`

# Копирование

- сору копирует одну последовательность в другую
- Задаются 3 итератора – начало и конец первой последовательности и начало второй
- Первые – итераторы чтения, второй – итератор записи
- Пользователь отвечает за то, чтобы во второй последовательности было достаточно места

# Копирование

```
vector2.resize(vector1.size());
std::copy(vector1.begin() , vector1.end() ,
 vector2.begin());
```

ИЛИ

```
std::copy(vector1.begin() , vector1.end() ,
 std::back_inserter(vector2));
```

# Вопрос

Корректен ли код, копирующий 5 первых элементов последовательности в конец?

```
const int N = ...;
```

```
double a[N];
```

```
...
```

```
std::copy(a , a+5 , a+N-5);
```

# Копирование

Не корректен, если  $N < 10$ . Мы затрем элементы до того, как их копировать.

Если есть двунаправленный итератор, можно использовать

```
const int N = ...;
```

```
double a[N];
```

```
...
```

```
std::copy_backward(a , a+5 , a+N-5);
```

# Преобразование

- Преобразование последовательности

```
double TransformT(double c)
```

```
{
```

```
 return 1.8 * c + 32;
```

```
}
```

```
std::vector<double> temperatures;
```

```
...
```

```
std::transform(temperatures.begin() ,
 temperatures.end() ,
 temperatures.begin() ,
 TransformT)
```



# Преобразование двух последовательностей

Результат преобразования записывается в третью.

```
double Fib(double a , double b)
```

```
{
```

```
 return a + b;
```

```
}
```

```
...
```

```
 std::vector < int > vec_fib;
```

```
 vec_fib.push_back(0);
```

```
 vec_fib.push_back(1);
```

```
 vec_fib.resize(42);
```

```
 transform(vec_fib.begin() , vec_fib.begin() + 40 ,
 vec_fib.begin() + 1 , vec_fib.begin() + 2 , Fib);
```

# Удаление

- `std::remove` удаляет из последовательности, заданной двумя итераторами, элементы, равные данному
- `std::remove` не может изменить количество элементов в последовательности, т.к. эту операцию нельзя однозначно выполнить для всех контейнеров

# Удаление

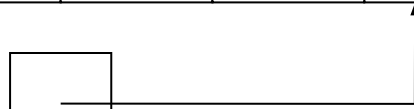
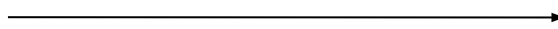
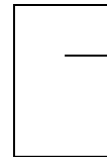
**Исходная  
последовательность**

|    |   |    |   |   |    |
|----|---|----|---|---|----|
| 27 | 3 | 31 | 2 | 3 | 28 |
|----|---|----|---|---|----|

**Измененная  
последовательность**

|    |    |   |    |   |    |
|----|----|---|----|---|----|
| 27 | 31 | 2 | 28 | 3 | 28 |
|----|----|---|----|---|----|

**Возвращенное  
значение**



# Удаление

- Результатом `remove` является итератор, указывающий на элемент, следующий за последним оставшимся
- После `remove` следует специфичным для контейнера способом освободить память из-под всех элементов, начиная с возвращенного значения.

```
std::vector<int> vec;
```

```
...
```

```
std::vector<int>::iterator iter = std::remove (vec.begin() ,
 vec.end() , 3);
```

```
vec.erase(iter , vec.end());
```

# Удаление

- `remove_if` – удаление элементов, соответствующих условию
- Задача: удалить первые 10 отрицательных чисел
- `unique` – встретив несколько идущих подряд равных элементов, заменяет их на один. Может получать компаратор.

# Удаление

- `remove_cору` – копирует элементы во вторую последовательность, удаляя равные данному
- `remove_cору_if` - копирует элементы во вторую последовательность, удаляя соответствующие условию
- `unique_cору` - копирует элементы во вторую последовательность, заменяя последовательности равных на один элемент

# Замена

- Аналогично удалению, но заменяет на заданное значение
- `std::replace`
- `std::replace_if`
- `std::replace_copy`
- `std::replace_copy_if`

# Заполнение

- `std::fill` – принимает начальный и конечный итераторы, значение
- `std::fill_n` – принимает итератор вывода, значение и количество элементов, которое необходимо вывести



# Заполнение. Примеры

```
std::vector< int> int_vector;
int_vector.resize(100);
std::fill(int_vector.begin() , int_vector.end() , 0);
```

```
std::vector< int> int_vector;
std::fill_n(back_inserter(int_vector.begin()), 100 , 0);
```

```
std::ostream_iterator <int> outiter(std::cout);
std::fill_n(outiter , 100 , 0);
```

# Заполнение

- Можно задать не значение, а функцию (которая будет вызвана для каждого элемента контейнера и ее возвращаемое значение записано в элемент) или объект-генератор (имеющий оператор `()` ).
- `std::generate`
- `std::generate_n`

# Заполнение

```
class FibonacciGenerator
{
public:
 FibonacciGenerator()
 :First(0),Second(1) {}
 int operator()()
 {
 int val = First; First = Second; Second = Second + val;
 return val;
 }
private:
 int First;
 int Second;
};

std::ostream_iterator <int> outiter(std::cout);
std::generate_n(outiter , 40 , FibonacciGenerator());
```

# Перестановки

- `std::swap` – меняет местами два значения, принимая ссылки
- `std::iter_swap` – меняет местами значения, на которые указывают заданные итераторы
- `std::swap_ranges` – меняет местами две последовательности

# Перестановки

- Какой итератор требуется для выполнения `swap_ranges`?

# Перестановки

- `std::reverse`, `std::reverse_copy` – переставляет в обратном порядке
- `std::rotate`, `std::rotate_copy` – циклический сдвиг
- `std::random_shuffle` – случайные перестановки

# Лексикографические перестановки

- abc
- acb
- bac
- bca
- cab
- cba

# Лексикографические перестановки

- `prev_permutation` – предыдущая перестановка
- `next_permutation` – следующая перестановка
- Принимает два двунаправленных итератора и объект-компаратор



# Сортировки

- `std::sort` – сортировка (обычно быстрая сортировка)
- `std::stable_sort` – сортировка с сохранением порядка равных элементов
- `std::partial_sort` – сортирует первые N элементов
- `std::partial_sort_copy` – копирует заданное число минимальных элементов во вторую последовательность

# Сортировки

```
vector2.resize(10);
std::partial_sort_copy(
 vector1.begin() , vector1.end() ,
 vector2.begin() , vector2.end());
```

# Сортировки

- `std::nth_element` – поиск порядковой статистики (гарантирует, что на позиции  $N$  будет тот элемент, который был бы там в отсортированном массиве, меньшие левее, большие правее)

# Сортировки

```
class Student
{
public:
 double AverageGrade() const;
};
class StudentComparator
{
public:
 bool operator(const Student& a , const Student& b)
 {
 return a.AverageGrade() > b.AverageGrade();
 }
};
std::vector <Student> vec_studs;
...
vec_studs.nth_element(vec_studs.begin() ,
 vec_studs.begin() + 10 ,
 vec_studs.end());
```

# Бинарный поиск

- `std::binary_search` – бинарный поиск в отсортированной последовательности (true, если найден)
- `std::lower_bound` - первый элемент, больший либо равный данному.
- `std::upper_bound` - первый элемент, больший данного.
- `std::equal_range` - оба этих элемента.
- Достаточно однонаправленного итератора, осмысленно только для итератора с произвольным доступом

# Слияние

- `std::merge` – объединяет две отсортированные последовательности в одну
- `std::inplace_merge` – объединение двух отсортированных половин последовательности на месте

# Слияние

```
for (int k = 1 ; k < n; k *= 2)
{
 for (int i = 0 ; i + k < n ; i += 2 * k)
 {
 int last = std::min(i + 2 * k , n);
 std::inplace_merge(array + i , array + i + k ,
 array + last);
 }
}
```

# Разделение

- Делим последовательность на группы, соответствующие условию и не соответствующие ему - `partition`
- Если нужно сохранить порядок внутри групп – `stable_partition`
- Результат – итератор, указывающий на начало второй группы.



# Пирамиды

- `std::make_heap` – расставляет элементы в последовательности так, как они лежали бы в невозрастающей пирамиде в [виде массива](#)
- `push_heap` – включает элемент в пирамиду
- `pop_heap` – извлекает из пирамиды максимальный элемент и ставит последним
- `sort_heap` – преобразует пирамиду в отсортированный массив

# make\_heap

**Исходная  
последовательно  
сть**

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 5 | 6 | 7 | 8 | 9 | 4 | 3 | 2 |
|---|---|---|---|---|---|---|---|

**Измененная  
последовательнос  
ть**

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 9 | 8 | 7 | 5 | 6 | 4 | 3 | 2 |
|---|---|---|---|---|---|---|---|

# Вопрос

- Как реализовать пирамидальную сортировку вектора?

# Пирамидальная сортировка

```
std::make_heap (vec.begin() , vec.end());
std::sort_heap(vec.begin() , vec.end())
```

# Множественные операции

- Реализуются над отсортированными последовательностями
- `std::includes` – проверка включения
- `std::set_union` - объединение
- `std::set_intersection` - пересечение
- `std::set_difference` – множественная разность
- `std::set_symmetric_difference` – присутствующие в одном и только одном множестве элементы

# Лабораторная работа №4. Использование стандартных алгоритмов STL.

# Задание

- Разработать программу на языке C++, реализующую функциональность в соответствии с вариантом задания.
- Настоятельно рекомендуется использование стандартных алгоритмов из библиотеки STL.

# Варианты задания

- Реализовать программу хранения массива геометрических фигур в двумерном пространстве. Фигура – это окружность или  $N$ -угольник. Программа должна поддерживать поворот и растяжение/сжатие всех фигур относительно заданного пользователем центра. Необходима устойчивость программы к выбору контейнера данных.



# Варианты задания

- Реализовать программу, хранящую в отсортированном массиве список пользователей операционной системы с информацией об имени и пароле. Пользователь вводит имя и пароль, программа сообщает, правильный ли пароль.
  - Указание: используйте функцию `binary_search`
  - Пожелание: Чтобы не хранить пароль в открытом виде, придумайте хэш-функцию, и храните имя и хэш-значение пароля. При проверке применяйте хэш-функцию к паролю и сравнивайте хэш-значения.

# Варианты задания

- Разработайте программу, хранящую базу данных телефонной компании (фамилия, номер, остаток денег на счету) и по запросу пользователя выдающую количество пользователей с отрицательным остатком и их список.
  - Указание: можно использовать `count_if`, `remove_copy_if`, `for_each...`, `equal_range`

# Варианты задания

- Реализуйте программу, заполняющую массив фиксированной длины прочитанными из файла значениями или случайными значениями (по выбору пользователя).
  - Указание: `generate`
  - Пожелание: используя стандартную библиотеку `boost` и функцию `boost::bind`, реализуйте чтение из файла в `generate`, не открывая файл каждый раз и не заводя глобальных переменных.

# Варианты задания

- Реализуйте программу, считывающие из двух файлов два набора строчек и проверяющую их на совпадение.
  - Указание: `generate`, `equal`
  - Пожелание: используя стандартную библиотеку `boost` и функцию `boost::bind`, реализуйте чтение из файла в `generate`, не открывая файл каждый раз и не заводя глобальных переменных.

# Варианты задания

- База данных телефонной компании реализована в форме отсортированного массива. Периодически приходит дополнение к базе – также отсортированный массив, который необходимо включить в главный.
  - Указание: используйте `merge` или `inplace_merge`.
- В словаре – пары слово + объяснение. Напечатать список статей об отраслях науки, в которых слово заканчивается на «логия».
  - Указание: Например, `remove_copy_if` или `for_each`.

# Варианты задания

- Прочитайте из файла последовательность чисел и выведите все возможные их перестановки в лексикографическом порядке (первая – по возрастанию, последняя – по убыванию).
  - Указание: `sort`, `next_permutation`
- В текстовом файле – список сотрудников фирмы. Распечатайте списки сотрудников, принятых на работу до и после 01.01.2005.
  - Указание: `partition`

# Литература

1. Кормен Т.Х., Лейзерсон Ч.И., Ривест Р. Л., Штайн К. Алгоритмы: построение и анализ. 2-ое издание. : Пер.с англ. – М.: ИД «Вильямс», 2007.
2. Б. Страуструп. Язык программирования С++. Специальное издание. Пер. с англ. –М.: ООО «Бином-Пресс», 2005 г. - 1104с.