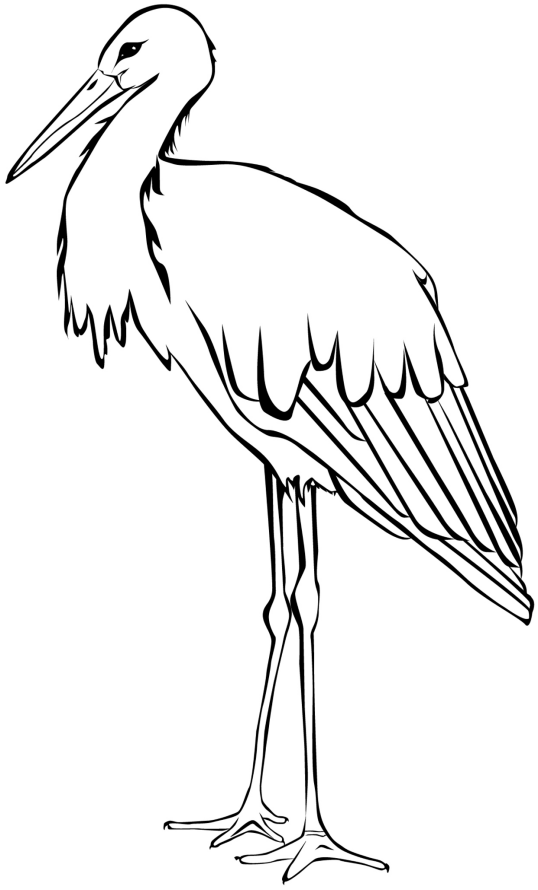


Алгоритмы и структуры данных

Конспекты лекций основного потока

Лектор: С. А. Объедков

Орлов Никита, Евсеев Борис, Рубачев Иван

НИУ ВШЭ, 2017

Содержание

Лекция 1. Асимптотика, простые алгоритмы, сортировка вставками

Пусть перед нами стоит задача: найти в некотором массиве медиану. Техническое задание выглядит так: на вход программе подается массив A, на выходе хотим получить одну из медиан, неважно какую.

Напомним определение медианы m:

$$m \in A = \begin{cases} |\{a \in A \mid a < m\}| \leqslant \frac{|A|}{2} \\ |\{a \in A \mid a > m\}| \leqslant \frac{|A|}{2} \end{cases}$$

Словами: медиана это такое число, что оно не больше половины элементов, но и не меньше половины элементов.

Легко видеть, что разных медиан в массиве может быть не больше двух, в зависимости от четности числа элементов.

Есть несколько способов решить эту задачу. Приведем несколько из них:

Алгоритм 1 Алгоритм поиска медианы

Ввод: Массив A
Вывод: Медиана m массива A

```
1: function MEDIAN(A)
2:     n := len(A)
3:     for i := 0 to (n − 1) do
4:         l := 0
5:         g := 0
6:         for j := 0 to (n − 1) do
7:             if A[j] < A[i] then
8:                 l := l + 1
9:             else if A[j] > A[i] then
10:                g := g + 1
11:        if l ⩽ n/2 and g ⩽ n/2 then
12:            return A[i]
```

Посмотрим еще на один способ:

Алгоритм 2 Примитивный алгоритм поиска медианы

Ввод: Массив A
Вывод: Медиана m массива A

```
1: function MEDIAN(A)
2:     n := len(A)
3:     B := sorted(A)
4:     return B[⌊n/2⌋]
```

На первый взгляд это сложный подход, так как мы должны отсортировать массив и пока не знаем, как это сделать.

Асимптотика

Итак, у нас есть как минимум два способа найти медиану. Возникает абсолютно естественное желание как-нибудь выяснить, какой лучше. Оказывается, в программировании можно провести сразу несколько таких оценок по разным критериям. Два главных ресурса, которые потребляют алгоритмы, это процессорное время и память вычислительного устройства.

Определение 1.1. Время (измеренное в некой абстрактной единице), необходимое алгоритму для завершения своей работы, называется *временем работы алгоритма* и обозначается как $T(n)$, где n - длина входных данных.

Время работы можно считать в разных единицах, например в *секундах*, если реализация алгоритма и исполнитель фиксированы, или в *элементарных операциях*, если речь идет про машину Тьюринга.

Различают несколько оценок времени работы:

1. *Худший случай* - максимально возможное $T(n)$ на входе длины n. Чаще всего используется на практике, так как дает верхнюю оценку времени работы алгоритма.
2. *Средний случай* - математическое ожидание $T(n)$ на входе длины n. Используется на практике реже, чем худший случай, в силу частой неопределенности вероятностного пространства для вычисления матожидания.
3. *Лучший случай* - минимально возможное $T(n)$ на входе длины n. На практике не используется, так как к любому сколько угодно неэффективному алгоритму можно приписать проверку на оптимальность входных данных и выдать ответ быстрее, чем средний или худший случай. Например, в задаче про поиск медианы можно проверять, отсортирован ли массив, и, если он не отсортирован, честно запускать поиск.

Для всего зоопарка алгоритмов существует инструмент их анализа - *асимптотический анализ*. Это методология, в которой время работы и занимаемая память алгоритма ставятся в соответствие классу функций.

Для начала дадим несколько определений.

Определение 1.2. O-большим от $g(n)$ называют такое множество функций, которое удовлетворяет условию

$$\underline{\underline{O}}(g(n)) = \{f(n) \mid \exists c_2 > 0, n_0 > 0 \ \forall n \geqslant n_0 : 0 \leqslant f(n) \leqslant c_2 g(n)\}$$

Иными словами, запись $f(n) \in O(g(n))$ означает, что $f(n)$ растет не быстрее, чем $g(n)$.

Определение 1.3. o-малым от $g(n)$ называют такое множество функций, которое удовлетворяет условию

$$\overline{\overline{o}}(g(n)) = \{f(n) | \forall c_2 > 0 \exists n_0 > 0 : \forall n \geqslant n_0 : 0 \leqslant f(n) < c_2 g(n)\}$$

Определение 1.4. Ω-большим от $g(n)$ называют такое множество функций, которое удовлетворяет условию

$$f(n) \in \Omega(g(n)) \leftrightarrow g(n) = \underline{\underline{O}}(f(n))$$

Определение 1.5. ω-малым от $g(n)$ называют такое множество функций, которое удовлетворяет условию

$$f(n) \in \omega(g(n)) \leftrightarrow g(n) = \overline{\overline{o}}(f(n))$$

Определение 1.6. $\Theta(g(n))$ называется такое множество функций, которое удовлетворяет условию

$$\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2, n_0 > 0 \ \forall n \geqslant n_0 : \ 0 \leqslant c_1 \cdot g(n) \leqslant f(n) \leqslant c_2 \cdot g(n)\}$$

Иными словами, $\Theta(g(n))$ растет примерно также, как и $f(n)$.

В нашем курсе мы часто будем писать что-то похожее на

$$T(n) = \Theta(f(n))$$

Такая запись с точки зрения математики некорректна, но мы будем понимать знак равенства как

$$T(n) \in \Theta(f(n))$$

Например:

$$4n^2 + 12n + 12 = \Theta(n^2)$$

$$c_1 = 1, \ c_2 = 16, \ n_0 = 2$$

$$\forall n \geqslant n_0 : \ 0 \leqslant n^2 \leqslant 4n^2 + 12n + 12 \leqslant 16n^2$$

В общем случае верно следующее:

Лемма 1.7. *Если многочлен $p(n)$ представим в виде*

$$p(n) = \sum_{i=0}^{d} a_i n^i, \ d = \deg(p), \ a_d > 0,$$

то

$$p(n) = \Theta(n^d)$$

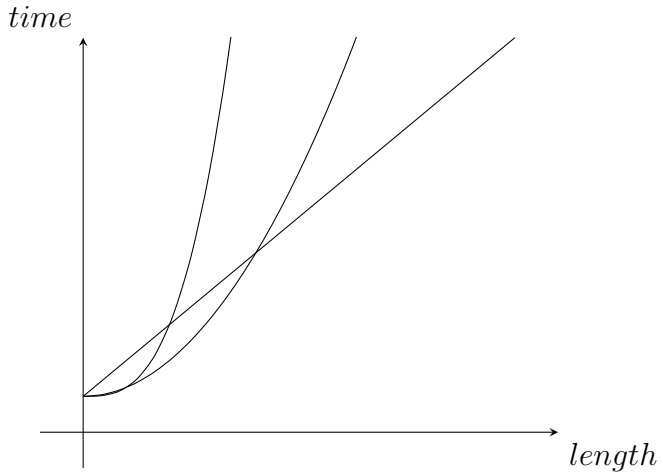
Замечание 1.8. Обычно функцию, описывающую время работы или память, занимаемую алгоритмом, называют *оценкой времени работы или памяти* алгоритма.

Замечание 1.9. По соглашению мы рассматриваем *асимптотически неотрицательные* функции, то есть такие, что

$$\exists n_0 \ \forall n > n_0 : \ f(n) \geqslant 0$$

Теперь поймем, что скрывается за классами функций Θ.

Пусть есть классы $\Theta(n)$, $\Theta(n^2)$, $\Theta(n^3)$. Для некоторых алгоритмов существует *оценка*, принадлежащая одному из этих классов. Помня про константу, можно сказать, что на достаточно большом объеме данных алгоритм с меньшей оценкой будет работать в среднем быстрее. Это ключевая мысль асимптотического анализа. Представить ее можно, построив графики неких линейной, квадратичной и кубической функций.



Для теоретического анализа сложности алгоритма берутся достаточно большие числа, но нужно понимать, что на практике может оказаться так, что входные данные могут быть меньше, чем n_0

Теперь получив матаппарат, оценим время работы алгоритмов поиска медианы.

Замечание 1.10. Будем считать, что элементарные арифметические операции, операции присваивания, копирования и тому подобные выполняются за $\Theta(1)$, иначе говоря, время их выполнения константо.

Первый алгоритм:

1. Лучший случай: медиана на первом месте. Тогда алгоритм выполнит одну итерацию внешнего цикла, n итераций внутреннего цикла, каждая из которых занимает константное время, и завершит работу. Сложность: $\Theta(1 \cdot n) = \Theta(n)$. Такая сложность считается достаточно хорошей.

2. Худший случай: медиана на последнем месте. Тогда алгоритм выполнит n итераций внешнего цикла, на каждой итерации произойдет n итераций внутреннего цикла. Сложность: $\Theta(n^2 - n) = \Theta(n^2)$.

Доказательство корректности заключается в том, что алгоритм *реализует* определение медианы. В таком случае он корректен, пока нет ошибок на уровне написания кода.

Второй алгоритм:

Второй алгоритм сложнее для оценки, так как мы не знаем, как сортируем массив. Операция взятия элемента выполняется за $\Theta(1)$. Остается сортировка, которую можно выполнить разными способами за разное время.

Сортировка вставками

Давайте возьмем простой алгоритм сортировки и оценим его сложность.

Алгоритм 3 Сортировка вставками

Ввод: Массив A с заданным на нем порядком $<$.

Вывод: Отсортированный по возрастанию массив A.

```
1: function INSERTION_SORT(A)
2:     n := len(A)
3:     for i := 1 to (n − 1) do
4:         k := A[i]
5:         for j := i − 1 to 0 do
6:             if k < A[j] then
7:                 A[j + 1] := A[j]
8:             else
9:                 break
10:        A[j + 1] := k
11:    return A
```

Словами: смотрим каждый i элемент и ищем его место среди первых $i-1$ элементов.

Для начала докажем корректность алгоритма. Для этого будем использовать *инвариант* - свойство математического объекта, которое не меняется после преобразования объекта.

Теорема 1.11. *Пусть есть неупорядоченный пронумерованный набор A элементов с заданным на них порядком меньше, и мы исполняем над ним алгоритм. Инвариант: элементы $A[0], \ldots, A[i-1]$ являются перестановкой исходных элементов в правильном порядке.*

Доказательство. Докажем по индукции. База $i = 1$ верна. Пусть инвариант верен для $i-1$ шага. Тогда смотрим $k = A[i]$ элемент.

Возможны 3 случая:

1. k - самый большой среди первых i элементов. Тогда алгоритм пропустит эту итерацию и перейдет к следующему.

2. k - самый маленький среди первых i элементов. Тогда алгоритм передвинет его в начало, пройдя весь цикл.

3. В противном случае, мы начинаем перебирать все элементы среди первых i до тех пор, пока операция сравнения на "меньше" не вернет ложь. Это означает, что мы в отсортированном массиве нашли элемент под номером j, который меньше либо равен k:

$$A[j] \leqslant k \ \leqslant \ A[j+1]$$

   Тогда мы сдвигаем элементы с $j+1$ до i на одну позицию вправо, и на $j+1$ место ставим k.

$$A[j] \leqslant k = A[j+1] < A[j+2]$$

   Все элементы с 0 по j позицию меньше либо равны k, а элементы с $j+2$ по i позицию они больше k.

[:|||:]

Из доказательства корректности инварианта прямо следует доказательство корректности алгоритма: когда алгоритм закончит свою работу, $i = n$, а значит инвариант верен для n элементов, а значит массив отсортирован.

Теперь можно оценить время работы сортировки вставками:

1. Лучший случай: массив уже отсортирован. Но тогда внешний цикл совершит $n-1$ итерацию, на каждой из которых произойдет одно сравнение. Сложность получилась $\Theta(n)$.

2. Худший случай: массив отсортирован в обратном порядке. Тогда на каждой итерации число шагов внутреннего цикла будет уменьшаться на 1. Значит

$$T(n) = \sum_{i=1}^{n-1} i = \Theta(n^2)$$

Лекция 2. Merge sort, Binary search, рекуррентные соотношения

Лекция будет дополнена примерами решения рекуррентных соотношений без основной теоремы, а так же доказательством теоремы. Текущая версия вычитана лектором.

Поговорим еще немного про сортировки. Сортировка вставками имеет квадратичную сложность, что не оптимально. Есть более быстрые алгоритмы, один из них называется *сортировкой слиянием.*

Сортировка слиянием

Суть сортировки достаточно проста: если у нас есть две отсортированные последовательности, мы можем их объединить в одну отсортированную, а последовательность длины один уже отсортирована, а значит мы можем разбить наш массив на блоки одинаковой длины, и рекурсивно отсортировать.

Опишем функцию слияния двух массивов разной длины.

Алгоритм 4 Функция слияния отсортированных массивов

Ввод: Отсортированные массивы A, B.
Вывод: Отсортированный массив C

```
1:  function MERGE(A, B)
2:      i := 0
3:      j := 0
4:      vector C
5:      while i < len(A) and j < len(B) do
6:          if A[i] ⩽ B[j] then
7:              C.push_back(A[i])
8:              i := i + 1
9:          else
10:             C.push_back(B[i])
11:             j := j + 1
12:     C := C + A[i : len(A)] + B[j : len(B)]
13:     return C
```

Доказательство корректности. На входе отсортированные массивы. На каждой итерации цикла выбирается наименьший из еще не выбранных элементов. **[:|||:]**

Время работы. Алгоритм смотрит на каждый элемент каждого массива, тогда его сложность получается $\Theta(len(A) + len(B))$. **[:|||:]**

Теперь сам алгоритм сортировки слиянием:

Алгоритм 5 Merge Sort

Ввод: Массив A

Вывод: Отсортированный массив C

```
1: function MERGE_SORT(A)
2:     if len(A) < 2 then
3:         return A
4:     else
5:         n := len(A)
6:         A_1 := merge_sort(A[0 : n/2])
7:         A_2 := merge_sort(A[n/2 : n])
8:         return merge(A_1, A_2)
```

Замечание 2.1. Сортировку слиянием можно написать разными способами. Например, изначально разбить массив на куски длиной 10, каждую из них отсортировать вставками, а потом уже последовательно слить в один отсортированный массив.

Доказательство корректности. Пока корректна функция *merge*, весь алгоритм корректен, так как вся задача разбивается на меньшие подзадачи, а рекурсия остановится на массивах размера 1. **[:|||:]**

Время работы. Время работы данного алгоритма $T(n) = \Theta(n \cdot \log(n))$: У нас $\log(n)$ - глубина рекурсии, а на каждом шаге мы пройдемся в сумме по всему массиву.

[:|||:]

Время работы сортировки слиянием можно записать и по-другому, с помощью *рекуррентной формулы*. В нашем случае она будет иметь вид

$$T(n) = \begin{cases} c, & n = 1; \\ 2T(\frac{n}{2}) + O(n), & n > 1; \end{cases}$$

где $2T(\frac{n}{2})$ - сложность рекурсивных вызовов, $O(n)$ - сложность слияния. Для таких соотношений хочется получить правило их раскрытия в явную формулу. О таком преобразовании говорит *основная теорема о рекуррентных соотношениях*.

Основная теорема

Теорема 2.2. *Пусть у нас есть рекуррентное соотношение, записанное в виде*

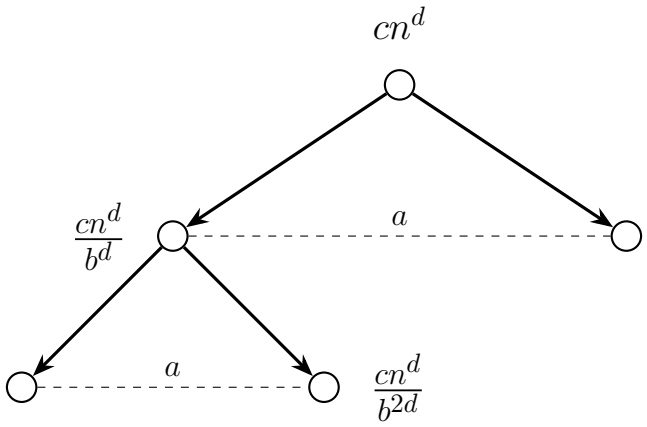
$$T(n) \leqslant \begin{cases} c, & n = 2 \\ aT(\frac{n}{b}) + cn^d, & n > 2 \end{cases}$$

Тогда его можно представить в следующем виде:

$$T(n) = \begin{cases} O(n^d \log(n)), & a = b^d \\ O(n^d), & a < b^d \\ O(n^{\log_b(a)}), & a > b^d \end{cases}$$

TODO: *Доказательство*

Доказательство. Рассмотрим алгоритм, который нашу задачу делит на каждом шаге на b подзадач.



На каждом уровне мы делим задачу на a подзадач, размер каждой отличается от другого уровня в b раз.

[:|||:]

Binary Search

Теперь давайте разберем другой рекуррентный алгоритм - *бинарный поиск*. Это алгоритм ищет в отсортированном элементе некоторый элемент и возвращает его индекс.

Алгоритм 6 Binary Search

Ввод: Отсортированный массив A, элемент x
Вывод: Индекс элемента e если он есть в массиве, -1 в противном случае.

```
1:  function BINARY_SEARCH(A, x)
2:      b := 0
3:      e := len(A)
4:      while b < e do
5:          m = round((b+e)/2)
6:          if A[m] == x then
7:              return m
8:          else if A[m] < x then
9:              b := m + 1
10:         else
11:             e := m
12:     return -1
```

Словами: смотрим в середину, сравниваем, сдвигаем границы поиска в сторону, где элемент может быть, повторяем пока не нашли или пока $b! = e$.

Время работы. Формула времени нашего алгоритма:

$$T(n) = \begin{cases} c, & n = 2 \\ T(\frac{n}{2}) + c, & n > 2 \end{cases}$$

По основной теореме

$$a = 1,\ b = 2,\ d = 0$$
$$T(n) = O(\log(n))$$

[:|||:]

Лекция 3. Quick sort, оптимальность сортировки слиянием

CAUTION! RECURSION IS HERE!

В стратегии *разделяй и властвуй*, которая используется во многих алгоритмах обработки данных, есть 3 четко выделяемые фазы.

1. *Деление*. Задача размера n делится на несколько меньших подзадач, подобных исходной.
2. *Решение*. Каждая из подзадач рекурсивно решается тем же алгоритмом. При этом важно, чтобы рекурсия остановилась на элементарном размере подзадачи. Под элементарным подразумевается, что такую подзадачу можно решить за константное время.
3. *Слияние*. Полученные результаты сливаются в одно общее решение.

На примере Merge Sort'а это выглядит следующим образом:

Деление	$A_1 = A[: \frac{n}{2}]$ $A_2 = A[\frac{n}{2} :]$
Решение	$A'_1 = mergesort(A_1)$ $A'_2 = mergesort(A_2)$
Слияние	**return** $merge(A_1, A_2)$

С ним все просто и понятно. Поделили - решили - объединили. Но существуют и другие алгоритмы сортировки. Один из самых знаменитых из них - *Quick Sort*.

QuickSort

Идея алгоритма в чем-то схожа с сортировкой слиянием: мы делим массив на две половины специальным образом и применяем алгоритм к каждой из частей рекурсивно. Главное отличие - алгоритм не имеет отдельной процедуры слияния и тратит всего $O(1)$ памяти, вместо совершенно нерационального потребления Merge Sort'ом.

Делить задачу мы будем следующим образом: возьмем *опорный* элемент $p \in A$, и приведем массив в следующее состояние:

$$\begin{array}{ccccccccc} & & A_1 & & & & A_2 & & \\ [& A[i_0] & \dots & A[i_k] & p & A[i_{k+1}] & \dots & A[i_{n-1}] &] \\ & & \leqslant p & & & & \geqslant p & & \end{array}$$

Словами: делим массив на части, в каждой из которых все элементы либо больше или равны опорного, либо меньше него.

Затем рекурсивно запускаем алгоритм от A_1 и A_2, пока не дойдем до задачи размера 1. Утверждается, что когда рекурсия остановится, задача будет решена, и массив A будет отсортирован.

Эффективность алгоритма полностью зависит от процедуры выбора опорного элемента. В идеальном случае надо брать медиану на каждом шаге, тогда мы будем делить массив ровно на две части, и время работы $T(n)$ будет выражаться так:

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + O(n) = O(n \log n)$$

В конце стоит $O(n)$, потому что нам нужно во время перемещения элементов посмотреть хотя бы на каждый из них.

Но если мы хотим сделать так, чтобы такая асимптотика достигалась постоянно, нужно искать медиану, а это как минимум $O(n)$, если повезет, но асимптотика тогда увеличится. Нерационально.

Самая главная часть в алгоритме называется *Partition*, именно она ответственна за разделение массива. Приведем псевдокод оптимальной по памяти версии, которая разделяет in-place.

Алгоритм 7 Partition

Ввод: Массив A, границы разделяемой части l, r.

```
1: function PARTITION(A, l, r)
2:     i := l
3:     for j := l + 1 to r do
4:         if A[j] ⩽ A[l] then
5:             swap(A[i], A[j])
6:             i += 1
7:         else
8:             continue
9:     swap(A[l], A[i])
```

Эта версия алгоритма за опорный элемент берет левую границу интервала. Получается такой переход:

$$[A[0] \ \ldots \ A[l] \qquad\qquad \ldots \qquad \ldots \qquad\qquad A[r] \ \ldots \ A[n-1]]$$

$$[A[0] \ \ldots \ A[i_0] \ \ldots \ A[i_{\frac{k}{2}}] \ A[l] \ A[i_{\frac{k}{2}+1}] \ \ldots \ A[i_k] \ A[r] \ \ldots \ A[n-1]]$$

Время работы. Посчитаем для худшего случая - когда наш массив отсортирован, значит операция Partition будет вызываться от массивов, каждый раз размером на 1 меньше. Получается

$$T(n) = T(n-1) + O(n) = \sum_{i=1}^{n}(n-i) = O(n^2)$$

В лучшем случае время будет как у Merge Sort'a - $O(n \log n)$ **[:|||:]**

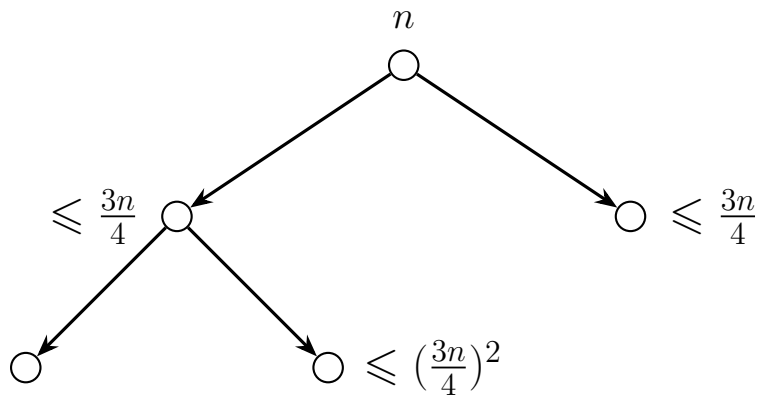
Этот алгоритм есть пример тех алгоритмов, для которых лучше рассматривать время в среднем случае.

Определение 3.1. Назовем элемент *центральным*, если он больше либо равен четверти всех элементов и меньше либо равен хотя бы четверти всех элементов

Такой элемент оказывается очень удобным, так как его можно брать за опорный, причем он гарантированно разобьет массив на непустые части. Искать такой элемент можно *детерминированно*, или точно вычитывать по какому-то алгоритму, но сложность такого выбора будет $O(n)$. Мы же будем использовать *вероятностный* подход, или брать случайный элемент в массиве.

С этого момента Quick Sort становится *рандомизированным* алгоритмом, так как на одном и том же входе шаги алгоритма неизвестны заранее. При этом становится легко считать его сложность

Время работы. Построим дерево исполнения алгоритма для случая, когда он разбивает задачи на подзадачи меньшие, чем $\frac{3}{4}$.



Определение 3.2. *j-подзадача* - это подзадача размера s такая, что

$$\exists j: \ \left(\frac{3}{4}\right)^{j-1} \leqslant s \leqslant \left(\frac{3}{4}\right)^{j}$$

На j подзадачу тратится время $O((\frac{3}{4})^j n)$, на все j подзадач - $O((\frac{4}{3})^{j+1})$. В итоге получаем, что среднее время на все j-подзадачи

$$T(n) = O\left(\left(\frac{3}{4}\right)^{j} n\right) \cdot O\left(\left(\frac{4}{3}\right)^{j}\right)$$

Прологарифмируем и получим, что

$$T(n) = O(n \log n)$$

[:|||:]

На практике обычно Quick Sort работает быстрее, чем Merge Sort, несмотря на одинаковую асимптотику в среднем случае. Связано это с тем, что QS сортирует на месте, в исходном массиве, MS тратит время на работу с памятью, так как ему нужно еще $O(n)$ памяти.

Замечание 3.3. При реализации алгоритма Partition можно вместе с границами передавать опорный элемент, переписав тело цикла. Получим процедуру, которая будет делить в интервале массив на две части, которые меньше либо равны и больше либо равны опорного.

Оптимальность сортировки слиянием

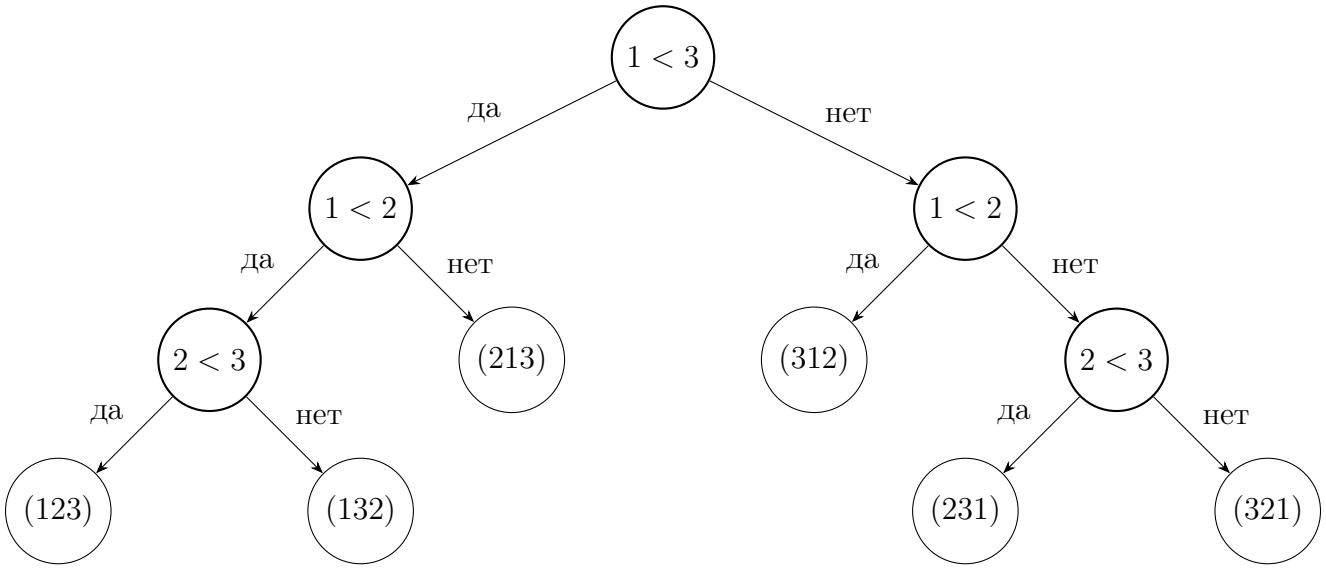
Утверждается, что время работы сортировки слиянием асимптотически самое маленькое.

Теорема 3.4. *Нельзя отсортировать массив быстрее, чем за $O(n \log n)$, если разрешено использовать только сравнение элементов и их перемещение.*

Доказательство. Рассмотрим массив A. Пусть ответом на задачу будет такая перестановка σ, что после ее применения массив окажется отсортирован.

Рассмотрим *дерево решений* некоторого абстрактного алгоритма сортировки. Пусть вершиной будет операция сравнения, ребром - ее исход, а листом - искомая перестановка. Такое дерево можно построить для каждого алгоритма.

Покажем пример такого дерева для сортировки массива длины 3:



Временем работы всего алгоритма будет высота этого дерева h. Число листьев не меньше, чем $n!$.

$$n! \leqslant 2^h$$

$$\log_2 n! \leqslant h$$

$$h \geqslant \log_2 n! \geqslant \log_2 \left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2} \log_2(n) - 1 = \Omega(n \log n).$$

[:|||:]

Лекция 4. Алгоритмы стратегии «разделяй и властвуй»

Это черновик лекции, окончательный релиз будет как только вычитаю у лектора.

Стратегия «разделяй и властвуй» используется в еще некоторых полезных алгоритмах. Относительно простым является алгоритм быстрого возведения в степень.

Возведение в степень

Для начала приведем тривиальный алгоритм.

Алгоритм 8 Возведение в степень

Ввод: Числа $x, y \in Z$.
Вывод: Число x^y.

```
1: function POWER(x, y)
2:     z := 1
3:     for i := 1 to y do
4:         z := z * x
5:     return z
```

Время работы. Так как ответ на задачу имеет экспоненциальный размер, то и сложность будет по крайней мере экспоненциальная. Если смотреть точнее, то получится $T(n) = O(n^2 10^n)$, где n - число цифр в записи числа.ответ **[:|||:]**

Алгоритм невероятно медленен, так как он выполнит честно каждое умножение, но есть способ вычислить степень быстрее. Для этого мы воспользуемся свойством степеней, а именно:

$$x^n = \begin{cases} x^{2^{\frac{n}{2}}}, & n = 2k \\ x \cdot x^{n-1}, & n = 2k+1 \end{cases}$$

Отсюда мы получаем рекуррентное соотношение для построения алгоритма, причем сделаем маленькую оптимизацию – будем вычислять ответ по какому-то модулю p. На практике арифметика работы с целыми числами в компьютере сама по себе модульная: при переполнении типа данных мы начнем с нуля. Псевдокод будет выглядеть так:

Алгоритм 9 Бинарное возведение в степень

Ввод: Числа $x, y, p \in Z$.

Вывод: Число $x^y \mod p$.

```
1: function POWER2(x, y, p)
2:     if y == 0 then
3:         return 1
4:     else
5:         t := power2(x, y/2, p)
6:         if y mod 2 == 0 then
7:             return t^2 mod p
8:         else
9:             return t^2 · x mod p
```

Время работы. При использовании этой оптимизации мы используем $O(n)$ возведений в квадрат чисел длины n. **[:|||:]**

Вычисление медианы

Вернемся к самому нашему первому алгоритму. Вычисление медианы является частным случаем другой задачи - выбора k-ой порядковой статистики, поэтому логичнее решить более общую задачу.

Алгоритм 10 Рандомизированный алгоритм выбора к-ой порядковой статистики

Ввод: Массив A, порядок k.

Вывод: .

```
1: function SELECT(A, k)
2:     choose p
3:     i = Partition(A, p)
4:     if i == k then
5:         return b[i]
6:     else if i < k then
7:         return Select(A[i:], k - i)
8:     else
9:         return Select(A[:i], k)
```

Словами: выбираем случайный элемент p, запускаем Partition. Если попали в цель, то все хорошо, иначе же отправляем алгоритм в ту часть, где элемент заведомо будет.

Все хорошо, но такой алгоритм дает в худшем случае асимптотику $O(n^2)$ из-за случайности выбора p. Посчитаем сколько времени потребуется алгоритму при таком случайном выборе.

Определение 4.1. Назовем текущий рекурсивный шаг алгоритма j-фазой, если для некоторого j обрабатываемая часть массива n' имеет размер

$$\left(\frac{3}{4}\right)^{j+1} \cdot n < n' \leqslant \left(\frac{3}{4}\right)^{j} \cdot n$$

Для подсчета сложности будем использовать факт, что в данном алгоритме время, проведенное в одной j фазе в среднем равно двум. Это понятно на уровне интуиции, ибо если нам не повезло с первого раза попасть в нужную половину, со второго раза мы попадем с большей вероятностью.

Отсюда получаем время работы:

$$\mathbb{E}(T(n)) \leqslant \mathbb{E}\left(\sum_{j=0}^{\log_{\frac{4}{3}} n} 2c\left(\frac{3}{4}\right)^j n\right) = 8cn$$

Получили линейное время в среднем случае. Для улучшения асимптотики алгоритм можно переписать, сделав выбор опорного элемента детерминированным.

Алгоритм 11 Детерменированный алгоритм выбора к-ой порядковой статистики

Ввод: Массив A, число $1 \leqslant k \leqslant A.size$.

Вывод: k порядковая статистика.

```
1:  function DSELECT(A, k)
2:      a_1 = A[0:5], a_2 = A[5:10], ...
3:      sort(a_1, ..., a_n)
4:      m_1 = a_1[2], ...,
5:      p = dselect((m_1, ...,), n/10)
6:      i = Partition(A, p)
7:      if i == k then
8:          return b[i]
9:      else if i < k then
10:         return Select(A[i:], k - i)
11:     else
12:         return Select(A[:i], k)
```

Суть этого алгоритма в том, что опорный элемент выбирается более-менее близко к медианному элементу, а значит и сложность будет более или менее близкой к линейной. На практике же на этот выбор тратится уйма времени, так что быстрее работает рандомизированный алгоритм.

Лекция 5. Введение в структуры данных

Когда мы рассматривали алгоритмы, мы говорили об обработке данных: как быстро найти результат, при этом мы не сильно говорили об эффективности по *памяти*. В этой части курса мы будем говорить о том, как данные *хранить* и как предоставлять к ним быстрый доступ.

Сейчас мы рассмотрим примеры нескольких странных структур данных, которые нам помогут понять, как это работает

«Быстрый» массив

Пусть мы хотим описать структуру со следующими свойствами:

Название	Описание	Желаемое время
init(n)	Создает массив нулей на n элементов	O(1)
read(i)	Возвращает i-ый элемент массива	O(1)
write(i, v)	Записывает в i-ый элемент массива значение v	O(1)

Попробуем описать такую структуру:

Алгоритм 12 Структура fast_array

```
struct fast_array:
  array A
  function INIT(n):
      A := malloc(n)
      for i := 0 to n-1 do
          A[i] = 0
  function READ(i):
      return A[i]
  function WRITE(i, v):
      A[i] = v
```

При таком раскладе наша инициализация работает за линейное время, поэтому придется отказаться от цикла в init.

Другим подходом к инициализации может быть *ленивая* инициализация: хранить информацию о том, была ли записана ячейка и пользоваться этим при чтении и записи. Заведем массив B длины n, в котором на i позиции будем хранить порядковый номер, под которым была инициализированна ячейка i. Заведем еще один массив C длины n, в котором на i позиции будем хранить номер ячейки, которая была проинициализирована i по счету. Также заведем счетчик инициализированных ячеек. Наша структура примет следующий вид:

Алгоритм 13 Структура fast_array

```
struct fast_array:
  array A
  array B
  array C
  integer k
  function INIT(n):
      A := malloc(n)
      B := malloc(n)
      C := malloc(n + 1)
      k := 0
  function READ(i):
      if is_inited(i) then
          return A[i]
      else
          return 0
  function WRITE(i, v):
      A[i] = v
      if not is_inited(i) then
          k := k + 1
          B[i] = k
          C[k] = i
  function IS_INITED(i)
      return 0 < B[i] ⩽ k and C[B[i]] == i
```

Главную роль тут играет функция is_inited. Ее задача - узнать, была ли проинициализирована ячейка. Ее корректность основана на том факте, что массив C заполняется по порядку, а значит мы можем гарантировать, что если было проинициализировано k ячеек, то в массиве C ячейки с 1 по k тоже проинициализированы.

N вставок

Рассмотрим теперь структуру данных, поддерживающую логарифмический поиск.

Название	Описание	Желаемое время
insert(e)	Вставляет в структуру элемент e	$O(\log n)$
search(e)	Возвращает индекс i, если элемент есть в структуре, -1 иначе	$O(\log^2 n)$

Основная идея состоит в том, что поиск в отсортированном массиве занимает логарифмическое время. Значит нам необходимо постоянно поддерживать структуру в отсортированном состоянии. Тут мы жертвуем временем вставки, но при этом поиск будет быстрым. Также в нашей структуре данных будем поддерживать следующий инвариант: $\forall i |A_i| \in \{0, 2^i\}$.

Алгоритм 14 Структура fast_search

```
struct fast_search:
  arrays A_0 ... A_{k-1}
  function SEARCH(e)
      for i := 0 to k - 1 do
          j := BinarySearch(a_i, e)
          return j
  function INSERT(e)
      B := [e]
      for i := 0 to k - 1 do
          if |A_i| == 0 then
              A_i = B
              break
          else
              B:= merge(A_i, B)
              A_i = []
```

Время работы. Формула времени для поиска в худшем случае, когда $\forall i |A_i| = 2^i$:

$$T(n) = O\left(\sum_{i=0}^{k-1} \log_2 2^i\right) = O\left(\sum_{i=0}^{k-1} i\right) = O(\log^2 n)$$

[:|||:]

Для вставки в худшем случае, когда $\forall i |A_i| = 2^i$:

$$T(n) = O\left(\sum_{i=0}^{k-1} 2^i\right) = O\left(\sum_{i=0}^{\lceil \log(n+1) \rceil} 2^i\right) = O(n)$$

Амортизационный анализ

Определение 5.1. *Амортизационный анализ* - метод подсчета времени работы нескольких запусков алгоритма При таком подходе время выполнения n запусков берется как сумма времени выполнения запусков.

Основная идея состоит в том, что если n операций затратили $O(n)$ времени, то амортизированная стоимость одного запуска - константа.

Одной из реализаций этого метода является *банковский метод*.

Банковский метод

Суть метода в состоит в том, что за выполнение каждой элементарной операции мы «кладем» в банк некоторое количество условных единиц (назовем ее *монеткой*), которые оплачивают нашу операцию.

Введем обозначения:

Обозначение 5.2. D_i - структура данных D после применения i-ой операции.

Обозначение 5.3. c_i - настоящая стоимость (время) выполнения i-ой операции

Обозначение 5.4. $\hat{c}$ - уплаченная стоимость i-ой операции.

Также введем *банк*, который является некоторой абстракцией, для работы с нашими монетками. Мы умеем класть монетки на счет в банк и забирать из банка наши монетки.

Условимся, что на D_0 наш счет в банке пуст. Наш анализ будет базироваться на следующем инварианте:

$$\forall n : \sum_{i=0}^{n} \hat{c}_i - \sum_{i=0}^{n} c_i \geqslant 0$$

Иными словами, наш баланс в банке неотрицателен.

Наш инвариант гарантирует, что при выполнении какой-нибудь тяжелой операции нам хватит монеток для ее выполнения. То есть мы должны платить такое количество монет во время легких операций, чтобы тяжелая операция смогла выполниться за излишек монет.

Схема будет понятнее на следующем примере.

Двоичный счетчик

Будем реализовывать простой двоичный счетчик при помощи массива нулей и единиц.

Алгоритм 15 Структура binary_counter

```
struct binary_counter:
  array A
  n := len(A)
  function INCREMENT
    for i := 0 to n − 1 do
      if c[i] == 0 then
        c[i] = 1
        break
      else
        c[i] = 0
```

Посчитаем время на выполнение n операций используя банковский метод. Будем платить 2 монетки за операцию установки разряда из нуля в единицу – одна монета на установку и одна в запас, и не будем платить ничего за сброс разряда в ноль. Инвариант амортизационного анализа гарантирует, что при каждой единице есть монетка, чтобы сбросить ее в ноль. Получаем, что

$$\forall i : \sum_{i=1}^{n} \hat{c}_i \leqslant 2n \Rightarrow T(n) = O(n)$$

$$\log(n^2) = o(\log(n^3)) \Leftrightarrow \lim_{n\to\infty} \frac{\log(n^2)}{\log(n^3)} = 0$$

Лекция 6. Продолжение амортизационного анализа. Динамический массив. Стек.

«Функция «фи» будет оценивать все то плохое, что у нас есть»

С. А. Объедков

CAUTION! MATHS IS HERE!

Динамический массив

Рассмотрим нам уже знакомую из курса программирования структуру данных: *динамический массив*. Суть его в том, что его размер ограничен только доступной нам памятью. Эта структура поддерживает следующие операции:

Название	Описание	Сложность
insert(el)	Вставляет в конец массива элемент	$O(n)$ в худшем случае
get(i)	Получает элемент по индексу	O(1)
remove_back()	Удаляет элемент из конца массива	O(1)

Алгоритм вставки тривиален: если нам не хватило уже выделенной памяти, чтобы вставить элемент, мы просто выделяем новую память, размер которой будет в два раза больше, чем есть сейчас, копируем уже записанные элементы и приписываем в конец новый элемент. Именно поэтому сложность добавления $O(n)$ в худшем случае.

Оценим время, требуемое на n операций вставок в худшем случае. Для примера распишем таблицу нескольких первых вызовов insert'а.

Шаг	Стоимость	Вместимость до	Вместимость после
1	1	1	1
2	2	1	2
3	3	2	4
4	1	4	4
5	5	4	8

Пусть $\forall i : \hat{c}_i = 3$. Тогда

$$\sum_{i=1}^{n} c_i \leqslant \sum_{i=1}^{n} 3 = O(n)$$

Значит время работы n операций, а значит амортизированное время выполнения одной операции - константа.

Потенциальный анализ

Теория

Определение 6.1. *Потенциал* – величина Φ, зависимая от некоторого состояния D_i структуры данных, на которую накладываются следующие условия:

$$\begin{cases} \Phi(D_0) = 0 \\ \Phi(D_i) \geqslant 0 \quad \forall i \end{cases}$$

Неформально можно сказать, что при приближении структуры данных к «плохому» состоянию потенциал растет. Так, например, в динамическом массиве потенциал перед реаллокацией максимален.

Основная формула, связывающая амортизированную и фактическую стоимость операции:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

Отсюда при анализе n операций получаем, что

$$\sum_{i=0}^{n} c_i = \sum_{i=0}^{n} \Big(\hat{c}_i + \Phi(D_{i-1}) - \Phi(D_i)\Big) = \sum_{i=0}^{n} \big(\hat{c}_i\big) + \Phi(D_0) - \Phi(D_n) \leqslant \sum_{i=0}^{n} \hat{c}_i$$

Значит наш метод дает ту же амортизированную оценку, но другим способом.

Потенциальный анализ вектора

Проведем анализ для вектора. Пусть $\Phi(D_i) = 2 \cdot D.size_i + D.cap_i$, где $D.size_i$ – размер массива на i шаге, $D.cap_i$ – вместимость массива том же шаге.

Тогда возможны два случая:

1. Вставка элемента происходит без реаллокации. Тогда

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + (2size_i - cap_i) - (2size_{i-1} - cap_{i-1}) = 3$$

2. Вставка с реаллокацией. Тогда

$$\hat{c}_i = size_i + (2size_i - cap_i) - (2size_{i-1} - cap_{i-1}) = 3$$

Отсюда делаем вывод, что в среднем вставка в вектор занимает константное время.

Анализ операции delete.

Доопределим в нашей структуре данных операцию удаления последнего элемента. При этом возникает вопрос: в какой момент стоит уменьшать вместимость структуры? Ответ прост: тогда, когда занята четверть памяти, уменьшать вместимость в два раза. Тогда мы гарантируем, что при последующем добавлении не будет происходить частых реаллокаций.

Введем характеристику $\alpha_i = \frac{size_i}{cap_i}$ и переопределим наш потенциал следующим образом:

$$\Phi(D_i) = \begin{cases} 0, & i = 0 \\ 2size_i - cap_i, & i > 0, \alpha_i \geqslant \frac{1}{2} \\ \frac{cap_i}{2} - size_i, & i > 0, \alpha_i < 0 \end{cases}$$

Тут будут формулы с лекции. Суть в том, что мы для каждого случая расписываем $\hat{c}_i$ и получаем везде константу.

Стек

Рассмотрим еще одну структуру данных под названием *стек*. Она поддерживает всего 4 операции:

Название	Описание	Сложность
push(el)	Положить на стек элемент *el*	$O(1)$
pop()	Вернуть и удалить верхний элемент	$O(1)$
top()	Вернуть верхний элемент	$O(1)$
is_empty()	Вернуть истину, если стек пуст	$O(1)$

Реализуется стек очень просто: у нас есть массив, для него мы храним указатель на последний элемент. При добавлении мы передвигаем указатель на одну позицию вперед и пишем туда элемент.

Иногда может возникнуть желание знать максимальный элемент на стеке. Так как наша структура не поддерживает поиск и произвольную адресацию, нам нужно завести отдельный стек только на хранение максимумов.

Тогда алгоритм будет следующим: при каждом добавлении элемента мы будем класть в другой стек текущий максимум. При удалении мы будем делать pop у этого дополнительного стека.

Лекция 7. Очередь. Приоритетная очередь

Очередь

Одной из полезных структур данных является *очередь*. Эта структура описывает очередь в самом естественном смысле: первый, кто пришел в очередь, первым и уйдет.

Наша структура поддерживает следующие операции:

Название	Описание	Сложность
enque(el)	Добавить в очередь элемент	$O(1)$
deque()	Взять первый элемент и удалить из очереди	$O(1)$
is_empty()	Вернуть истину, если очередь пуста	$O(1)$

Эту структуру можно описать на основе списка. Будем хранить указатели на первый и последний элемент. При удалении возвращаем значение по первому указателю и передвигаем его. При добавлении связываем элемент с последним и передвигаем указатель. Оставим реализацию на упражнение читателю.

Другая реализация основана на массиве. Но при реализации возникает много трудностей. Идея такая: пусть наш массив – закольцованный буфер. Тогда будем поддерживать указатели на первый и на последний элемент. Попробуем описать структуру.

Алгоритм 16 Структура queue_on_array

```
struct queue_on_array:
  Q – array
  head, tail – integer

  function ENQUE(el)
      if head == tail + 1 then
          return error
      Q[tail] := el
      tail := tail + 1 (mod n)

  function DEQUE( )
      x := Q[head]
      head := head + 1 (mod n)
      return x
```

Нужно еще вводить проверку на наличие элементов, проверку на переполнение, проверку еще на много чего, чтобы сохранить структуру в согласованном состоянии.

Очередь над стеком

Допустим, теперь мы хотим знать, какой элемент минимальный. Мы уже умеем хранить минимум на стеке, поэтому попробуем описать очередь над стеком. Идея заключается в следующем:

заведем два стека. В первый будем добавлять элементы тогда, когда будем делать операцию enque. При операции deque будем переливать элементы из первого стека во второй и брать верхний во втором стеке. При этом на каждый стек заведем дополнительный, который будет поддерживать минимум. Приведем реализацию:

Алгоритм 17 Структура queue_on_stack

```
struct queue_on_stack:
  A, B – stack_min

  function ENQUE(el)
    A.push(el)

  function DEQUE( )
    if B.is_empty() then
      transfer(A, B)
    return B.pop()

  function TRANSFER(A, B)
    while not A.is_empty() do
      B.push(A.pop())

  function MIN( )
    return min(A.min(), B.min())
```

Приоритетная очередь

Допустим теперь мы хотим удалять из очереди не первый элемент, а тот элемент, который будет иметь наименьший (или наибольший) *приоритет*. Приоритет - это просто числовая характеристика элемента. Тогда от структуры мы хотим следующие операции:

Название	Описание	Сложность
insert(el, pr)	Добавить в очередь элемент el с приоритетом pr	???
extract_min()	Взять первый элемент и удалить из очереди	???
is_empty()	Вернуть истину, если очередь пуста	$O(1)$

Сложность операций сильно зависит от реализации. Тривиальный подход хранения очереди в кольцевом буфере займет линейное время на взятие элемента и константу на добавление. Можно уменьшить сложность, но нельзя реализовать обе операции за константу.

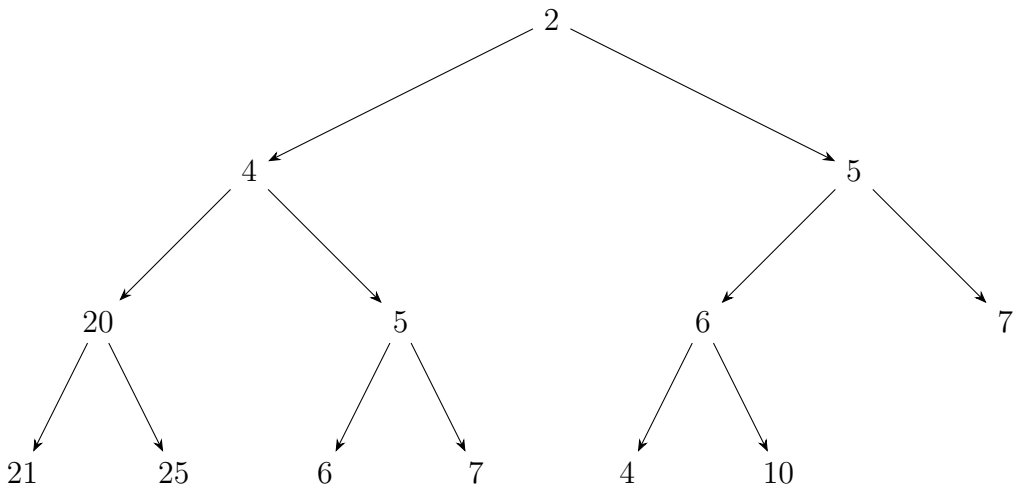
Теорема 7.1. *Нельзя реализовать операции вставки и взятия минимального элемента в приоритетной очереди за константное время.*

Доказательство. Пусть операции константы. Но тогда мы можем реализовать сортировку за линейное время: добавим элементы в виде чисел, и добавим их в приоритетную очередь. Потом будем делать взятие элементов, пока очередь не пуста. Получили $O(n)$, но ведь сортировка имеет сложность минимум $O(n \log n)$, получили противоречие. **[:|||:]**

Куча

Куча, или *двоичная куча*, или *бинарное дерево* – структура данных над деревом, поддерживающая минимальный элемент в корне, логарифмическую вставку и удаление. Она основана на поддержании следующего инварианта: $\forall x \in heap : x.val \geqslant x.par.val$, где $x.par$ - родитель узла. Особенностью этого дерева является то, что из каждого родителя выходит не более двух потомков.

Приведем пример такого дерева:



Дерево можно описывать в виде списка свзяности, введя структуру *node* и поддерживая в каждом узле соответствующие ссылки:

Алгоритм 18 Структура node

struct node:
  $value$ – type
  $parent$ – node
  $left\_child$ – node
  $right\_child$ – node

У корня поле $parent$ будет иметь значение $null$, у листьев это значение будут иметь поля $left\_child$ и $right\_child$.

Для простоты мы будем описывать дерево в массиве. Первым элементом будет корень дерева, затем построчно все уровни дерева. Массивом из примера выше будет

$$[2,\ 4,\ 5,\ 20,\ 5,\ 6,\ 7,\ 21,\ 25,\ 6,\ 7,\ 4,\ 10]$$

При вставке будем записывать элемент в конец массива. При этом может нарушиться инвариант, поэтому придется восстановить справедливость. Эта процедура будет смотреть на родителя нового узла и, при необходимости, менять местами узлы. При удалении будем менять первый и последний элемент массива, удалять последний и восстанавливать справедливость уже с корнем. Эти функции называются $heaptify\_up$ и $heaptify\_down$.

Алгоритм 19 Структура heap_on_array

```
struct heap_on_array:
  A – array
  function PARENT(i)
    return ⌊i/2⌋

  function LEFT_CHILD(i)
    return 2i

  function RIGHT_CHILD(i)
    return 2i + 1

  function HEAPTIFY_UP(i)
    if i > 1 then
      j := parent(i)
      if A[i] < H[j] then
        swap(A[i], A[j])
        heaptify_up(j)

  function HEAPTIFY_DOWN(i)
    if left_child(i) < n or right_child(i) < n then
      j := argmin(H[2i], H[2i + 1])
      if H[i] > H[j] then
        swap(H[i], H[j])
        heaptify_down(j)
```

Утверждается, что после применения функций heaptify_down и heaptify_up свойство кучи будет восстановлено.

Утверждение 7.2. *Свойство кучи может быть нарушено только на одном из ребер.*

Сортировка кучей

С помощью такой структуры данных мы можем относительно быстро отсортировать массив и поддерживать его в таком состоянии: сначала добавим все элементы в кучу, а затем возьмем все, начиная с минимального.

Лекция 8. Двоичные деревья поиска.

Пусть мы хотим реализовать структуру данных, моделирующую множество. Для этого нам надо уметь быстро *вставлять, искать* и *удалять* элементы из множества. В этом нам поможет такая структура, как *двоичное дерево поиска.*

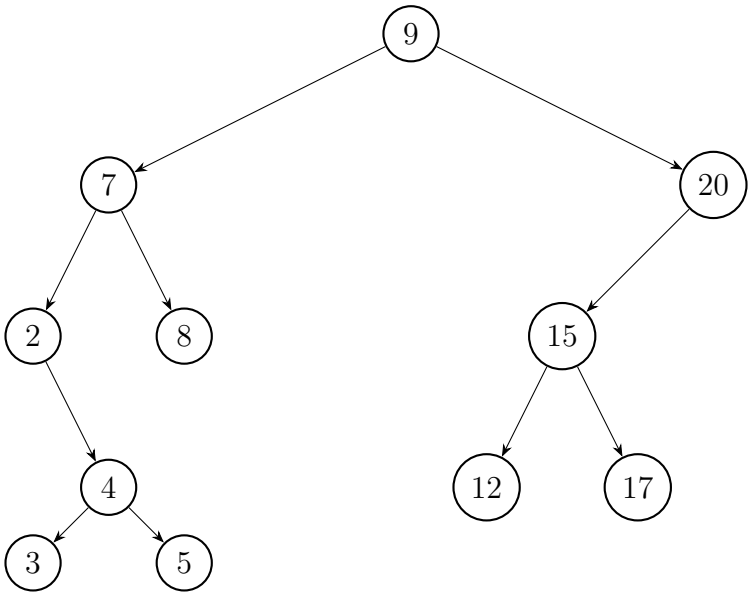
Двоичное дерево поиска

Будем называть двоичное дерево T *деревом поиска*, если для него выполняются следующие условия:

1. У любого узла существует не более двух детей
2. Если узел y лежит в левом поддереве с корнем x, то $y.key \leqslant x.key$; если в правом, то $y.key \geqslant x.key$.

Замечание 8.1. Строгость неравенства зависит от поставленной задачи. Например, при реализации multiset'а неравенство нестрогое, а при реализиации set'а – строгое.

Приведем пример двоичного дерева поиска:



Весь код в этой лекции будет написан в Python-подобном синтаксисе

Для начала опишем поиск в такой структуре:

```
def tree_search(x, k):
    if x:
        if k < x.key:
            return tree_search(x.left, k)
        elif k > x.key:
            return tree_search(x.right, k)
```

```
        else:
            return x
    return None
```

Словами: на каждом шаге сравниваем с текущим и идем в нужную ветку.

Время работы. В худшем случае время работы данного алгоритма займет $O(height(T))$, где T – дерево. [:|||:]

Затем нужно научиться вставлять ноды в дерево. Основная идея: вставляем в корень, проверяем выполнение инварианта, останавливаемся, если все хорошо, и идем дальше иначе.

```
def tree_insert(t, k): # t — tree
    z = Node(k)
    x = t.root
    while x:
            z.parent = x
        if z.key > x.key:
            x = x.right
        else:
            x = x.left

    if z.parent:
        if z.key > z.parent.key:
            z.parent.right = z
        else:
            z.parent.left = z
    else:
        t.root = z
```