МАИ

Лабораторная работа №11

 «Методы поиска приближённого значения трансцендентного уравнения и их программная реализация на основе итерационного и рекурсивного подходов»Вариант №14

Факультет робототехнических и интеллектуальных систем

Кафедра «Системы приводов летательных аппаратов»

Выполнил:

Студент группы М7О-114БВ-24

Фельдман Лев Борисович

Проверил:  
Доцент Кафедры 702 Козлова Н.М.

Ассистент Кафедры 702 Милославский Я.Г.

Москва 2025

**Цель работы: знакомство с пирамидальным методом сортировки, методом быстрой сортировки и сортировкой методом слияния.**

**Рекурсивный алгоритм** — это алгоритм, который решает задачу путём многократного (вложенного) вызова самого себя для решения более простых (подзадач), пока не будет достигнуто некоторое базовое условие (условие выхода), при котором дальнейшие вызовы прекращаются.

Основные компоненты рекурсивного алгоритма

1. Базовое условие (условие выхода)  
   Определяет, при каком входе рекурсия завершается без новых вызовов. Без корректно заданного базового условия рекурсия «уйдёт в бесконечность» и вызовет переполнение стека.
2. Рекурсивный вызов  
   Часть алгоритма, где функция вызывает сама себя с новым (обычно упрощённым) аргументом.

**Пример 1. Факториал**

Факториал числа n определяется как:

n!=

function f = factorial\_rec(n)

% Рекурсивная функция для вычисления факториала n

if n == 0 f = 1; % базовый случай

else f = n \* factorial\_rec(n-1); end % рекурсивный шаг

end

При factorial(3) ход вычислений будет таким:  
3 \* factorial(2) → 3 \* (2 \* factorial(1)) → 3 \* (2 \* (1 \* factorial(0))) → 3 \* 2 \* 1 \* 1 → **6**.

**Пример 2. Числа Фибоначчи**

Последовательность Фибоначчи:

F

function f = fib\_rec(n)

% Рекурсивная функция для n-го числа Фибоначчи

if n == 0 f = 0; % базовый случай 1

elseif n == 1 f = 1; % базовый случай 2

else f = fib\_rec(n-1) + fib\_rec(n-2); % рекурсивный шаг

end

end

>>fib\_rec(10)

ans =

55

**Пример 3. Быстрый подсчёт суммы элементов в массиве**

Допустим, у нас есть вектор AAA, и мы хотим рекурсивно получить сумму всех его элементов:

function s = sum\_rec(A)

% Рекурсивная функция для суммы элементов вектора A

if isempty(A) s = 0; % базовый случай: пустой массив

else s = A(1) + sum\_rec(A(2:end)); % рекурсивный шаг

end

end

>>sum\_rec([4, 7, 1, 3])

ans =

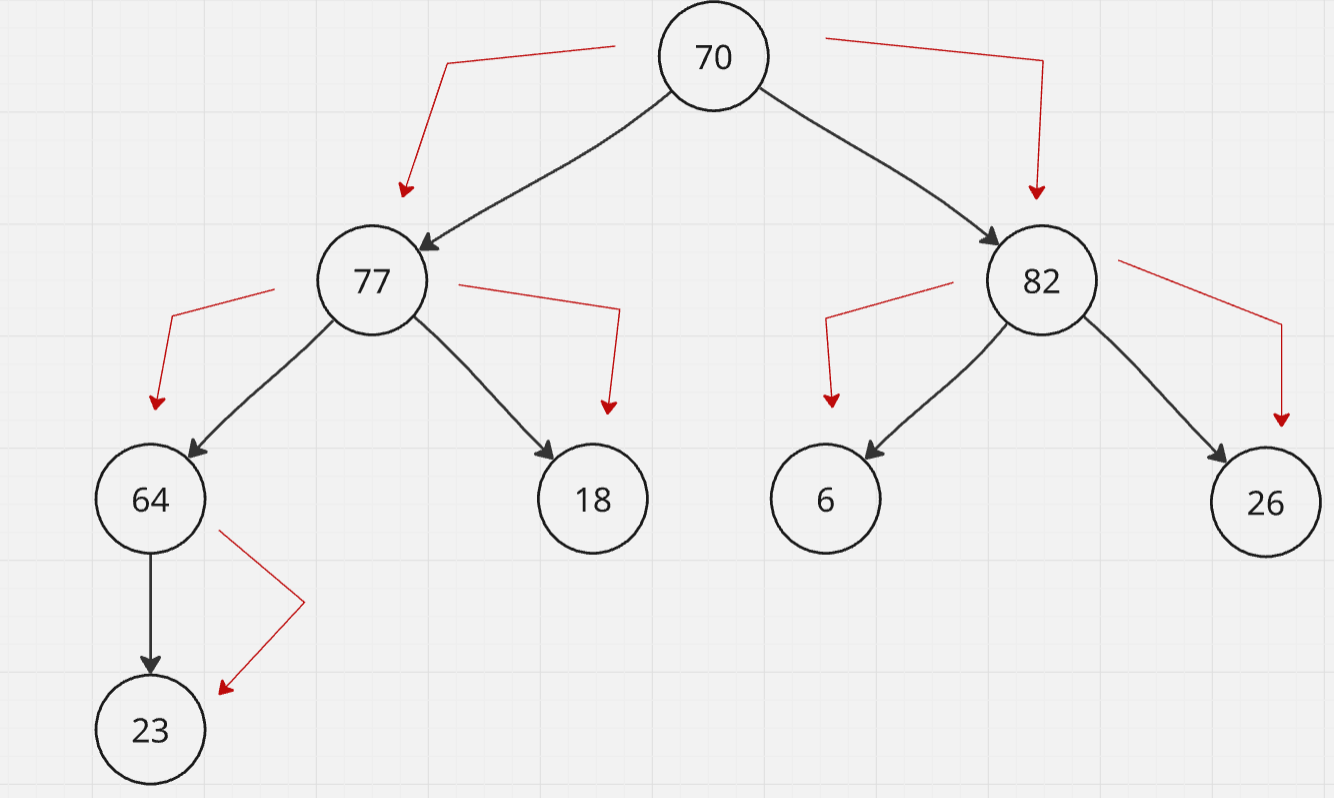
15

## Пирамидальная сортировка

1. **Построение “пирамиды” (кучи)**
   * Воображаем, что наш массив — это куча карточек, сложенных в виде почти полного двоичного дерева: каждая карточка «родитель» лежит над двумя «дочерними», и по правилу кучи её значение не меньше значений под ней.
   * Начинаем с середины массива (половины карточек) и «просеиваем вниз» каждую карточку, если она меньше одной из своих «дочек»: меняем их местами и продолжаем, пока не восстановим правило кучи. В результате получаем именно такую «горку», где самый крупный элемент лежит наверху.
2. **Сортировка через «снятие вершины»**
   * Теперь самая большая карточка (максимум) находится в корне — на первой позиции массива. Чтобы поставить её в конец будущей отсортированной части, меняем местами элемент в начале и элемент в конце массива.
   * После этого «вершина» (сейчас там бывшая последняя карточка) может нарушить правило кучи, поэтому опять «просеиваем её вниз», восстанавливая правильную структуру.
   * Таким образом мы «фиксируем» самую крупную карточку в конце и уменьшаем размер активной части массива на единицу (она остаётся уже отсортированной).
3. **Повторение до конца**
   * Снова берём новую вершину — снова самый большой среди оставшихся — и отправляем её сразу за последний отсортированный элемент.
   * Снова восстанавливаем кучу и повторяем, пока не останется одна карточка.

В результате мы постепенно «снимаем» по одному наибольшему элементу и «укладываем» их в конец массива, а благодаря тому, что структура кучи позволяет быстро (за время пропорциональное логарифму размера) находить и перестраивать максимум, общая скорость — около n·log n операций. А дополнительной памяти почти не нужно, потому что все перестановки происходят внутри самого массива.

Исходный массив [70 77 82 64 18 6 26 23]



**1. Построение max-кучи**

i: 1 2 3 4 5 6 7 8

A: 70, 77, 82, 64, 18, 6, 26, 23

Последний внутренний узел = n/2 = 4.  
Проходим heapify от i = 4 ↓ 1.

**Шаг 1.1: heapify(i=4)**

* Узел A[4]=64, дети: A[8]=23
* 64 ≥ 23 → менять ничего не надо.

Массив остаётся

[70, 77, 82, 64, 18, 6, 26, 23]

**Шаг 1.2: heapify(i=3)**

* Узел A[3]=82, дети: A[6]=6, A[7]=26
* 82 ≥ max(6,26) → без изменений.

**Шаг 1.3: heapify(i=2)**

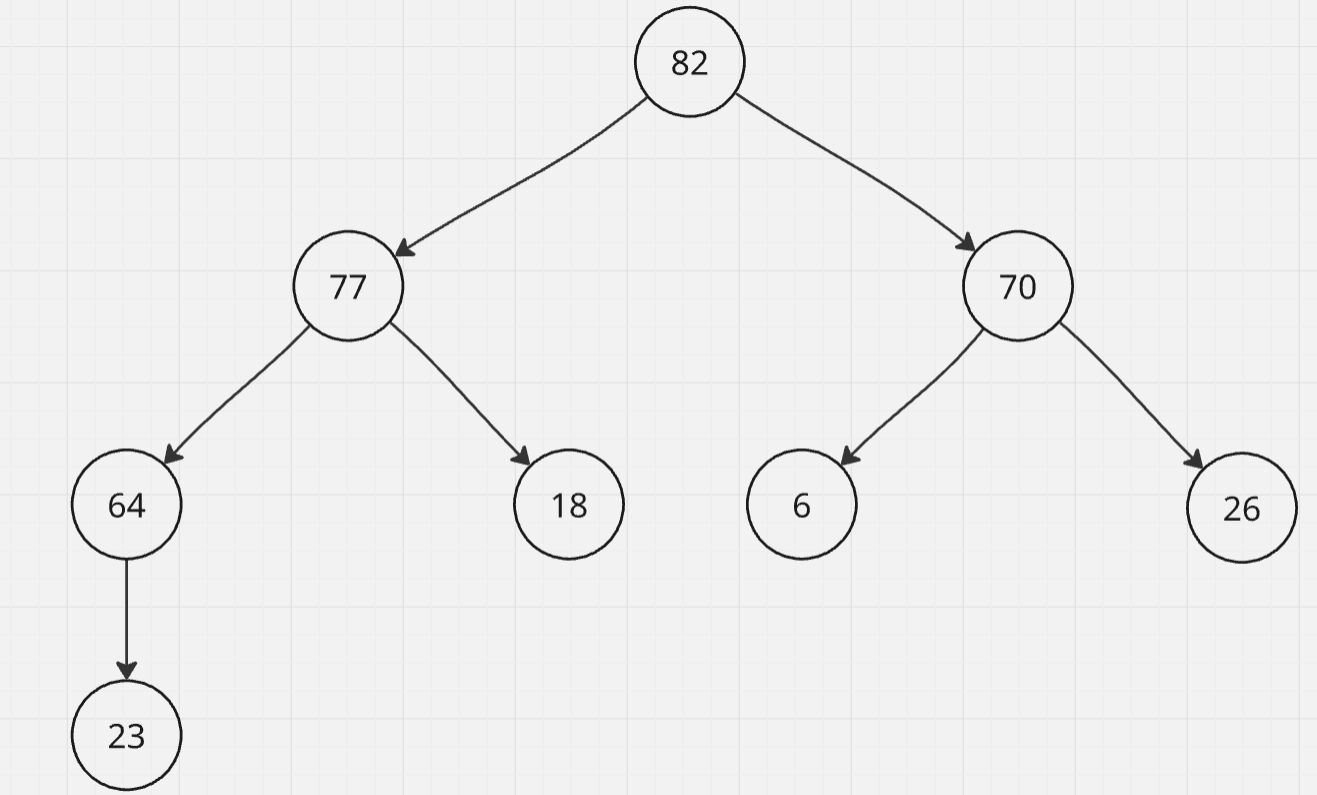
* Узел A[2]=77, дети: A[4]=64, A[5]=18
* 77 ≥ max(64,18) → без изменений.

**Шаг 1.4: heapify(i=1)**

* Узел A[1]=70, дети: A[2]=77, A[3]=82
* Максимум среди (70,77,82) = 82 → меняем A[1]↔A[3]:

Массив →

[82, 77, 70, 64, 18, 6, 26, 23]

**Итоговая max-куча:** 

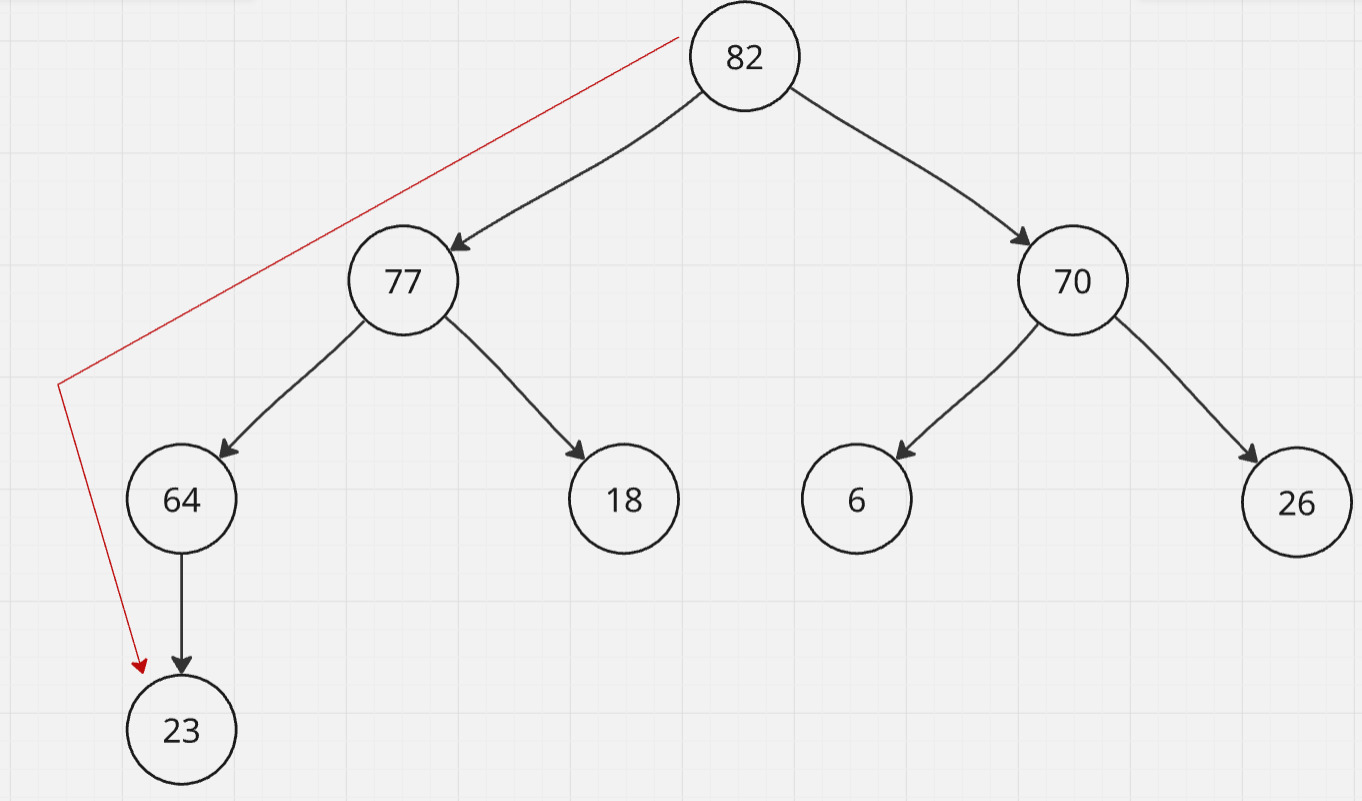
Массив:

[82, 77, 70, 64, 18, 6, 26, 23]

**2. Сортировка (извлечение максимума)**

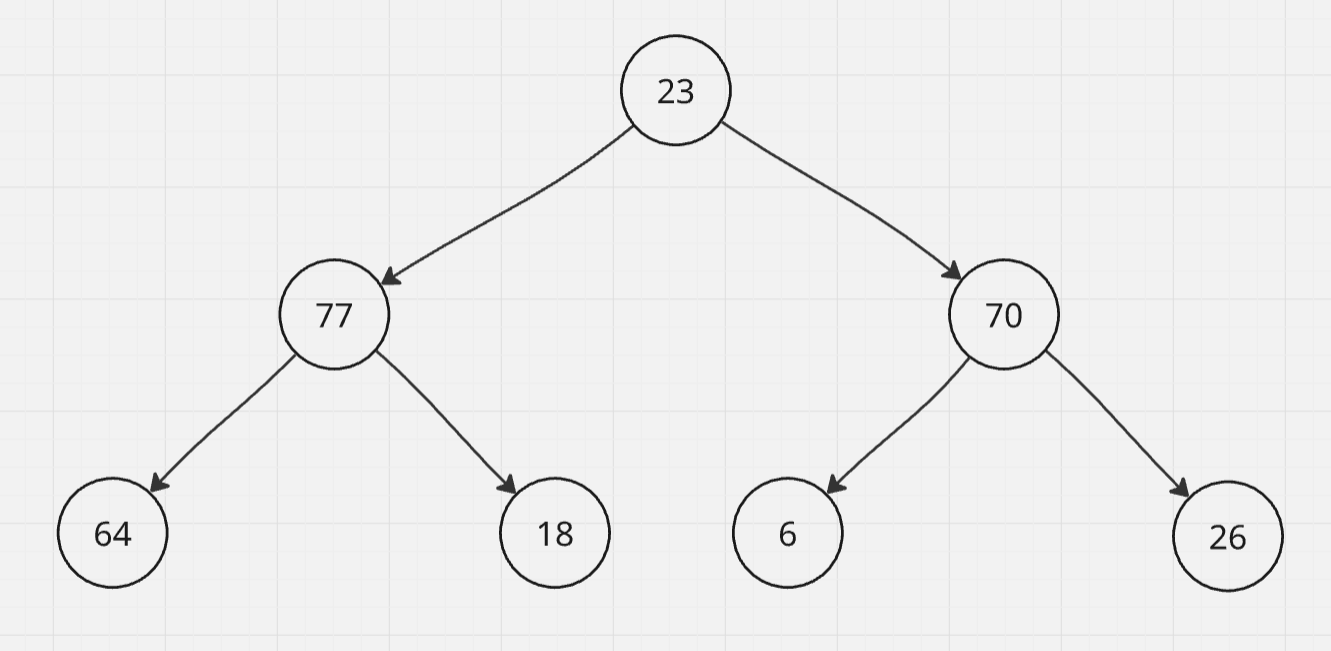
Будем уменьшать «размер кучи» heap\_size с 8 до 2, каждый раз:

1. Меняем A[1] (максимум) ↔ A[heap\_size].
2. Уменьшаем heap\_size на 1.
3. Heapify на корне.

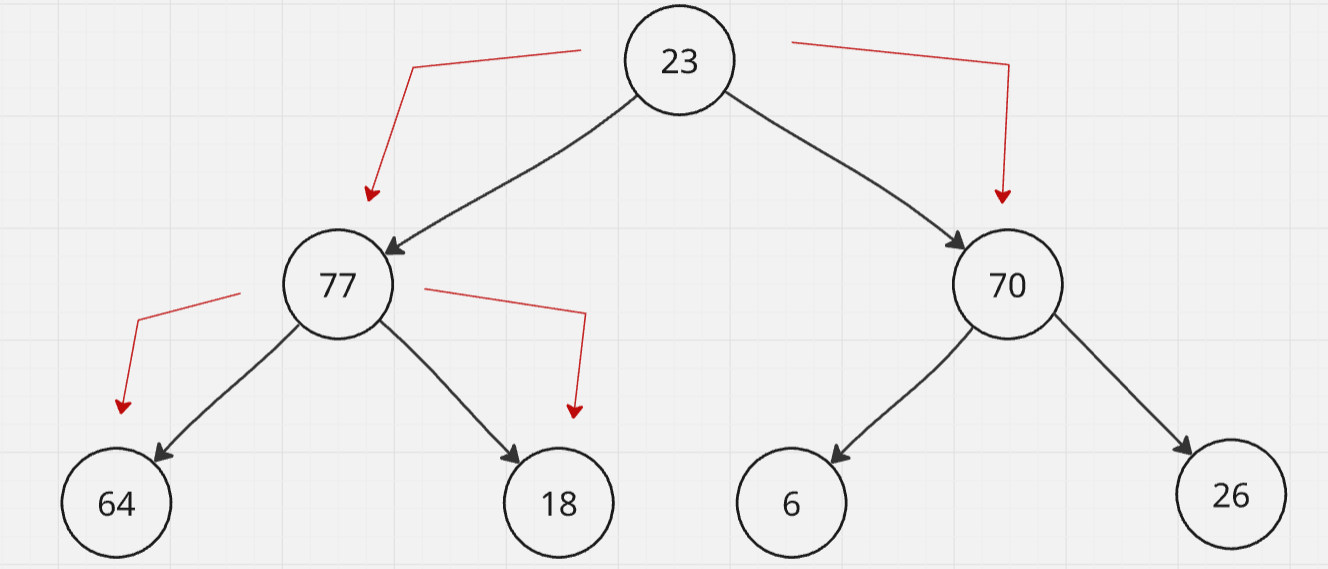


**Шаг 2.1: heap\_size = 8**

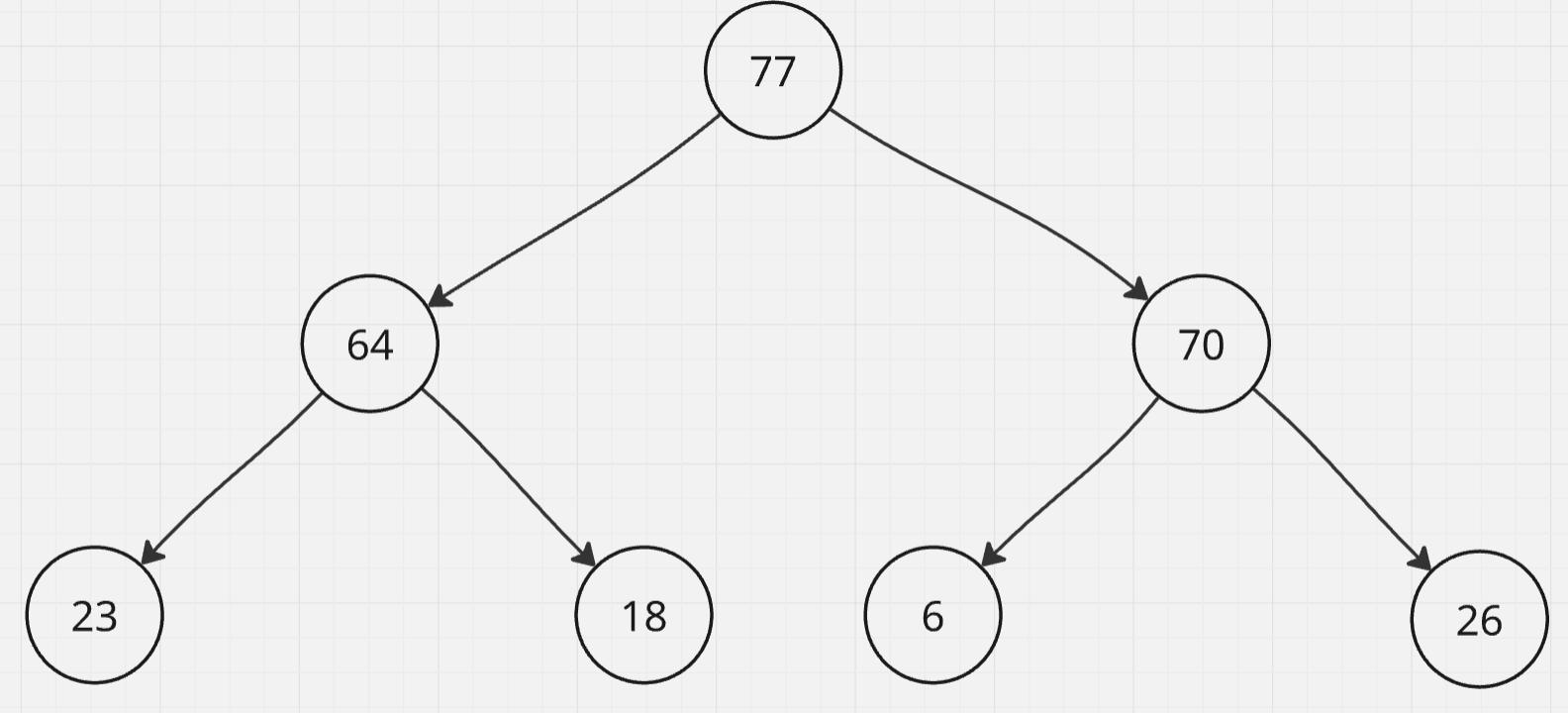
1. swap A[1]=82 ↔ A[8]=23  
   → [23, 77, 70, 64, 18, 6, 26, 82]
2. heap\_size→7



1. heapify i=1:
   * Дети: 77, 70 → max=77 → swap 23↔77 →  
     [77, 23, 70, 64, 18, 6, 26, 82]
   * Теперь i=2 (23): дети 64, 18 → max=64 → swap 23↔64 →  
     [77, 64, 70, 23, 18, 6, 26, 82]



Куча (size 7):

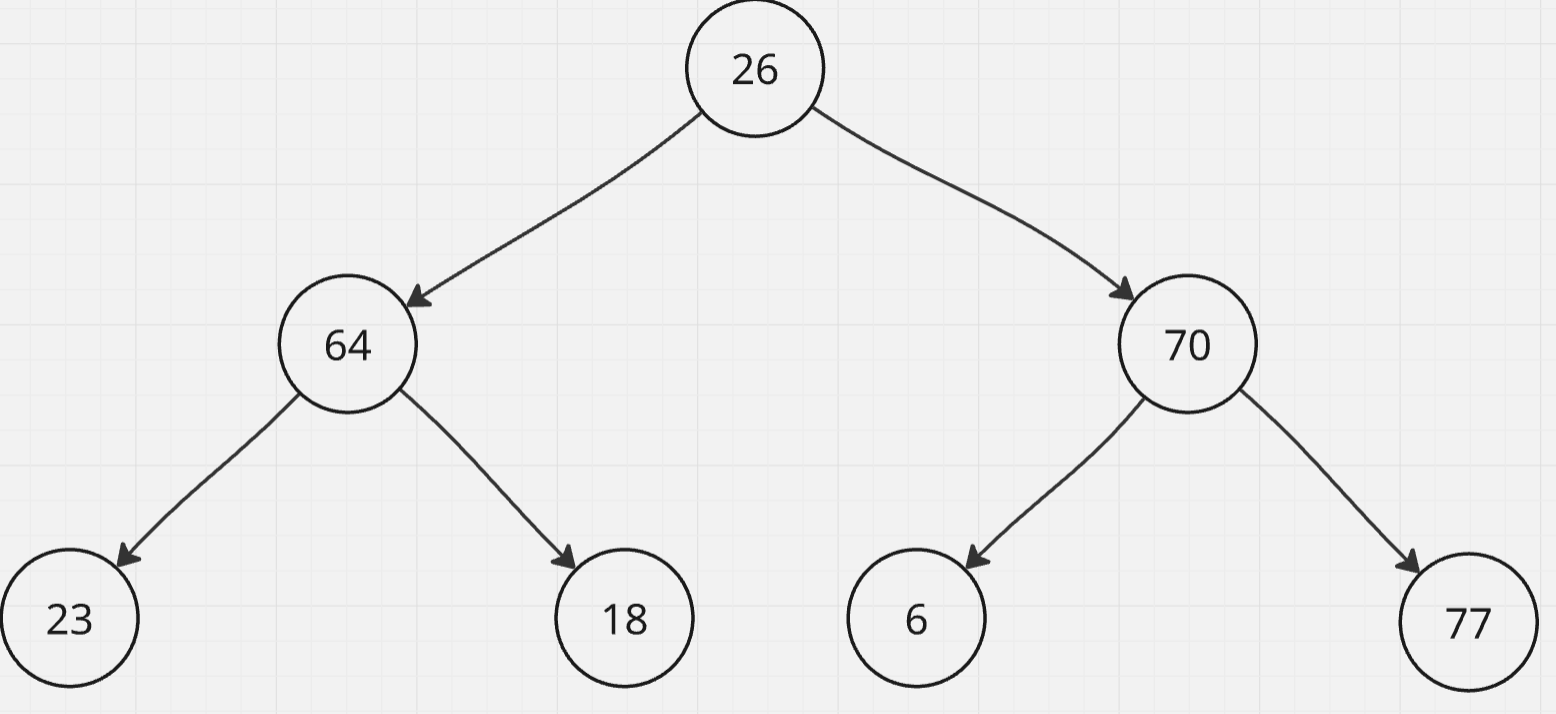


Массив:

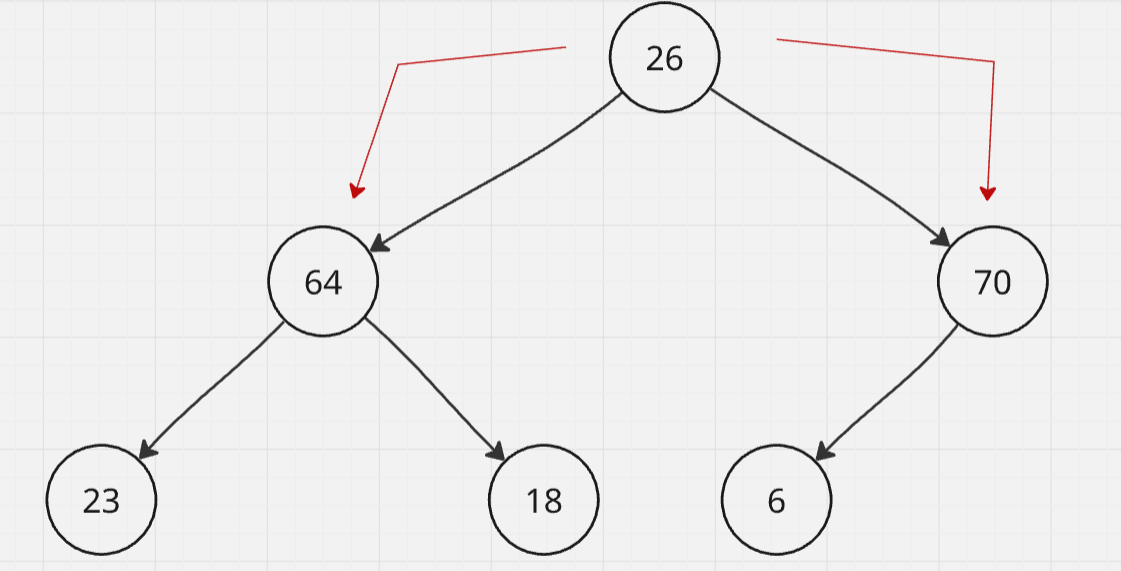
[77, 64, 70, 23, 18, 6, 26, 82]

**Шаг 2.2: heap\_size = 7**

1. swap A[1]=77 ↔ A[7]=26  
   → [26, 64, 70, 23, 18, 6, 77, 82]

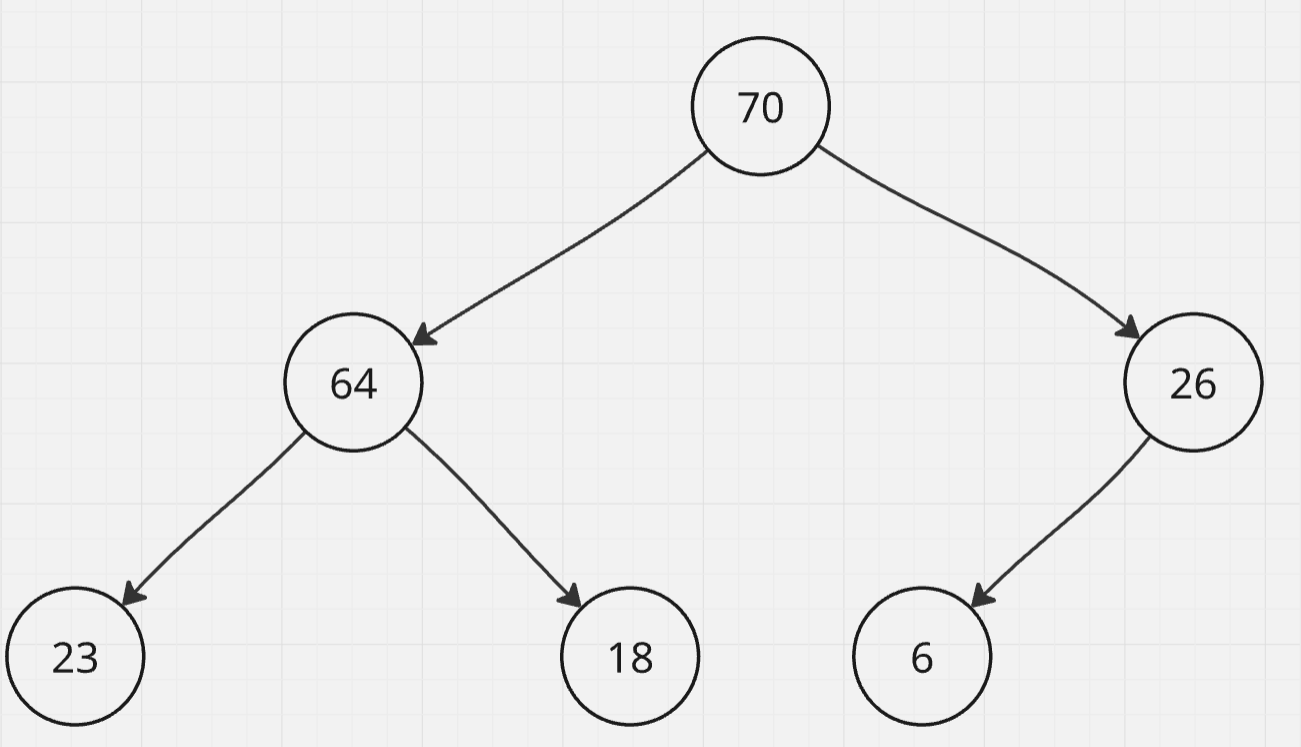


1. heap\_size→6



1. heapify i=1:
   * Дети: 64, 70 → max=70 → swap 26↔70 →  
     [70, 64, 26, 23, 18, 6, 77, 82]
   * Теперь i=3 (26): дети: A[6]=6 (единственный) → 26 ≥ 6 → стоп.

Куча (size 6):

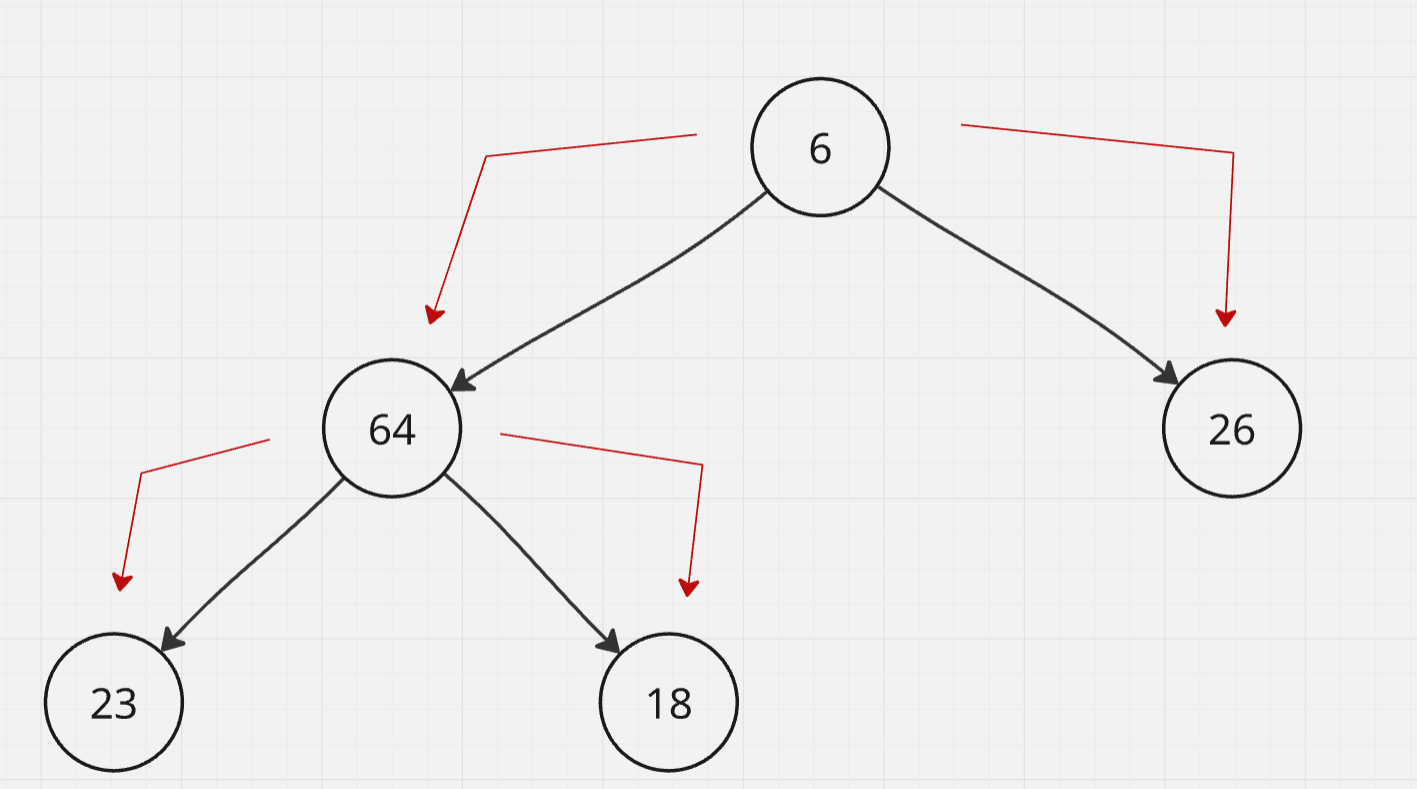


Массив:

[70, 64, 26, 23, 18, 6, 77, 82]

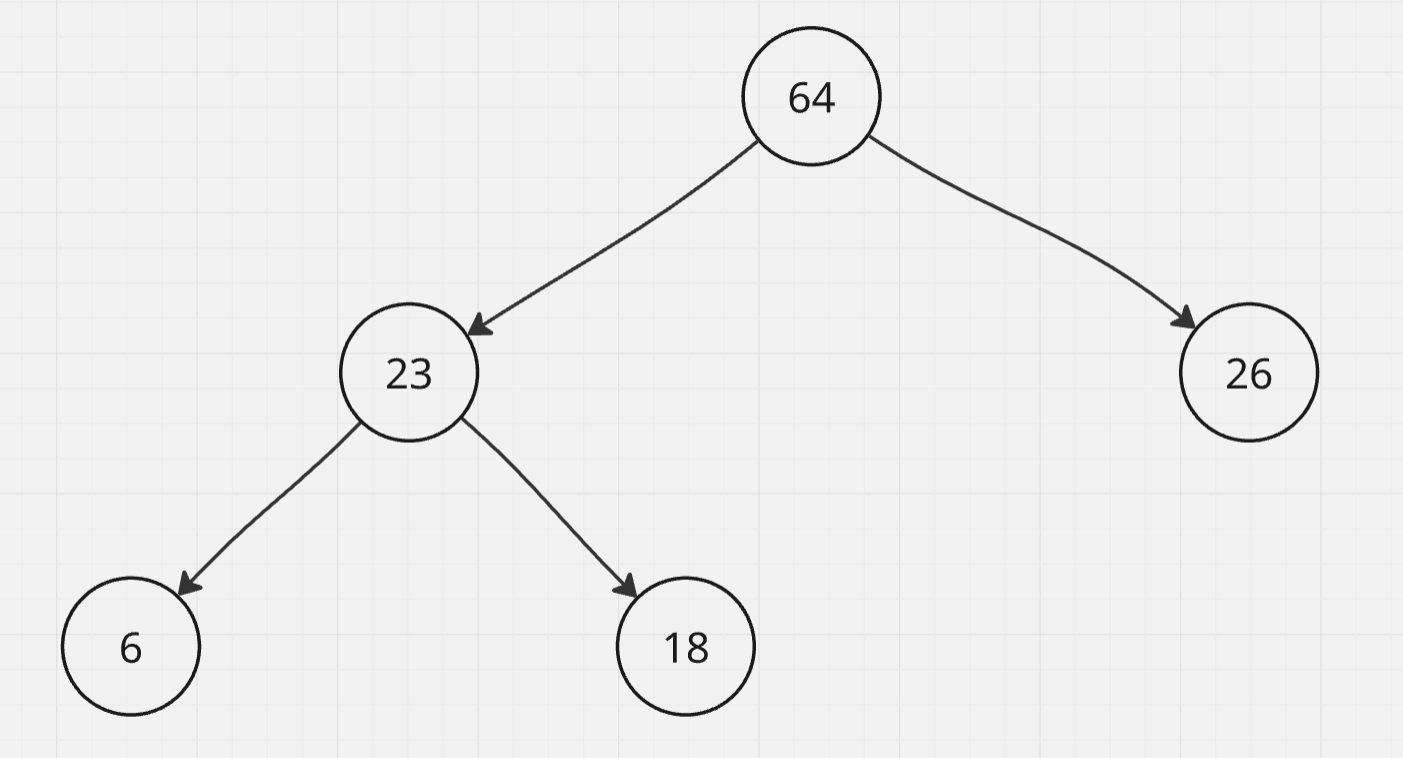
**Шаг 2.3: heap\_size = 6**

1. swap A[1]=70 ↔ A[6]=6  
   → [6, 64, 26, 23, 18, 70, 77, 82]
2. heap\_size→5



1. heapify i=1:
   * Дети: 64, 26 → max=64 → swap 6↔64 →  
     [64, 6, 26, 23, 18, 70, 77, 82]
   * i=2 (6): дети 23, 18 → max=23 → swap 6↔23 →  
     [64, 23, 26, 6, 18, 70, 77, 82]

Куча (size 5):

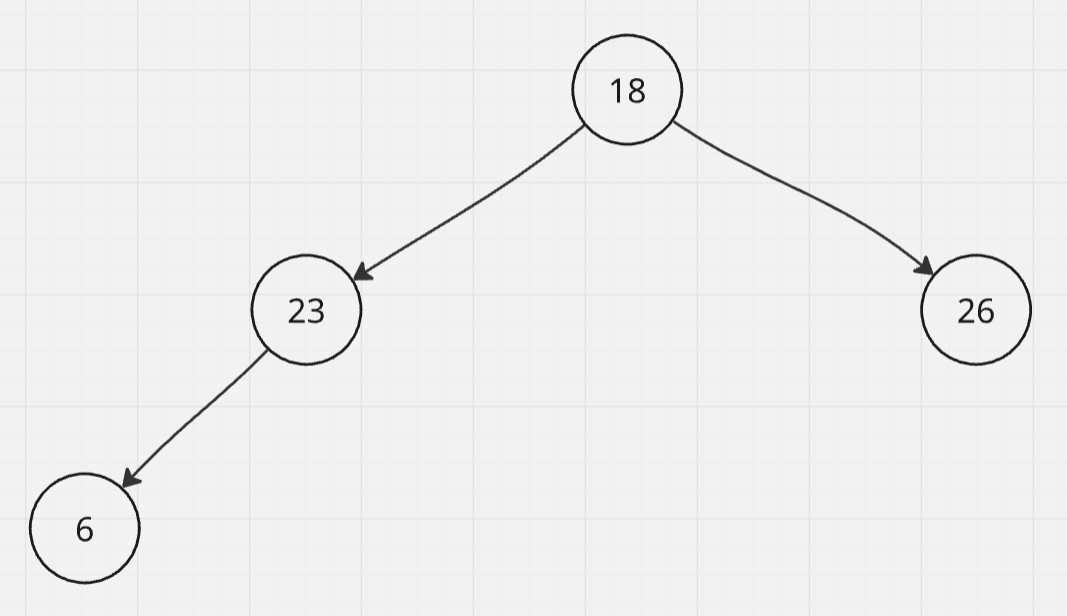


Массив:

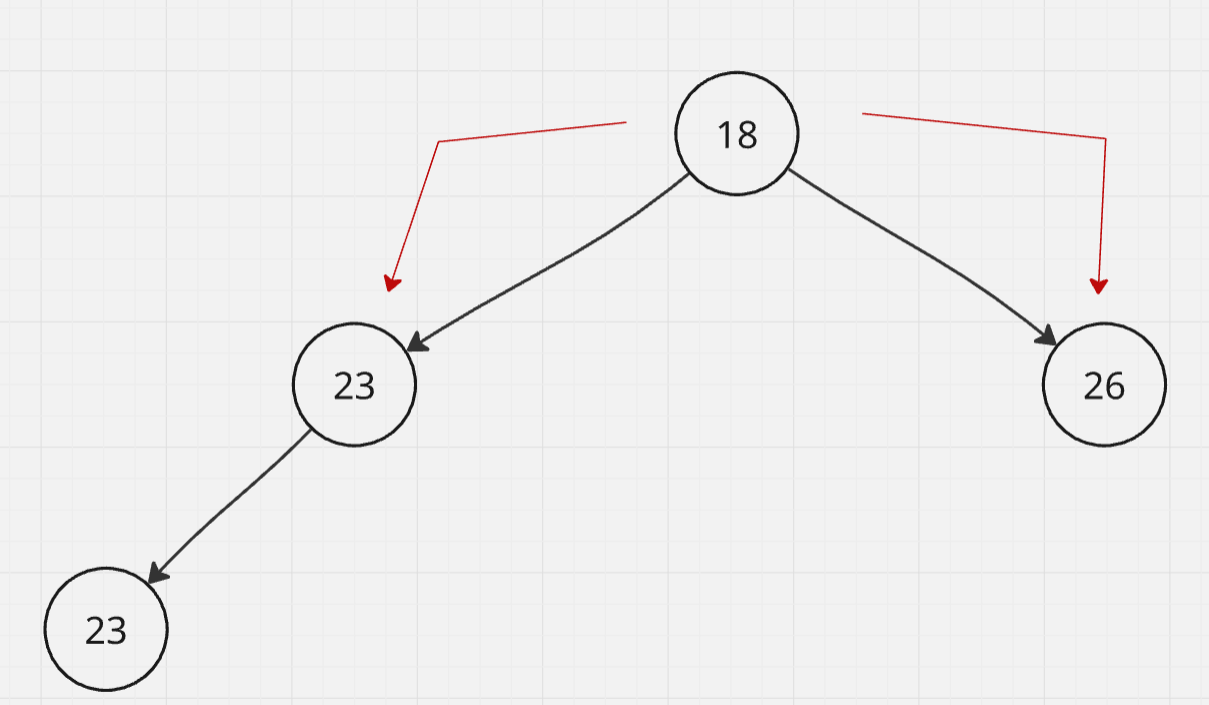
[64, 23, 26, 6, 18, 70, 77, 82]

**Шаг 2.4: heap\_size = 5**

1. swap A[1]=64 ↔ A[5]=18  
   → [18, 23, 26, 6, 64, 70, 77, 82]
2. heap\_size→4

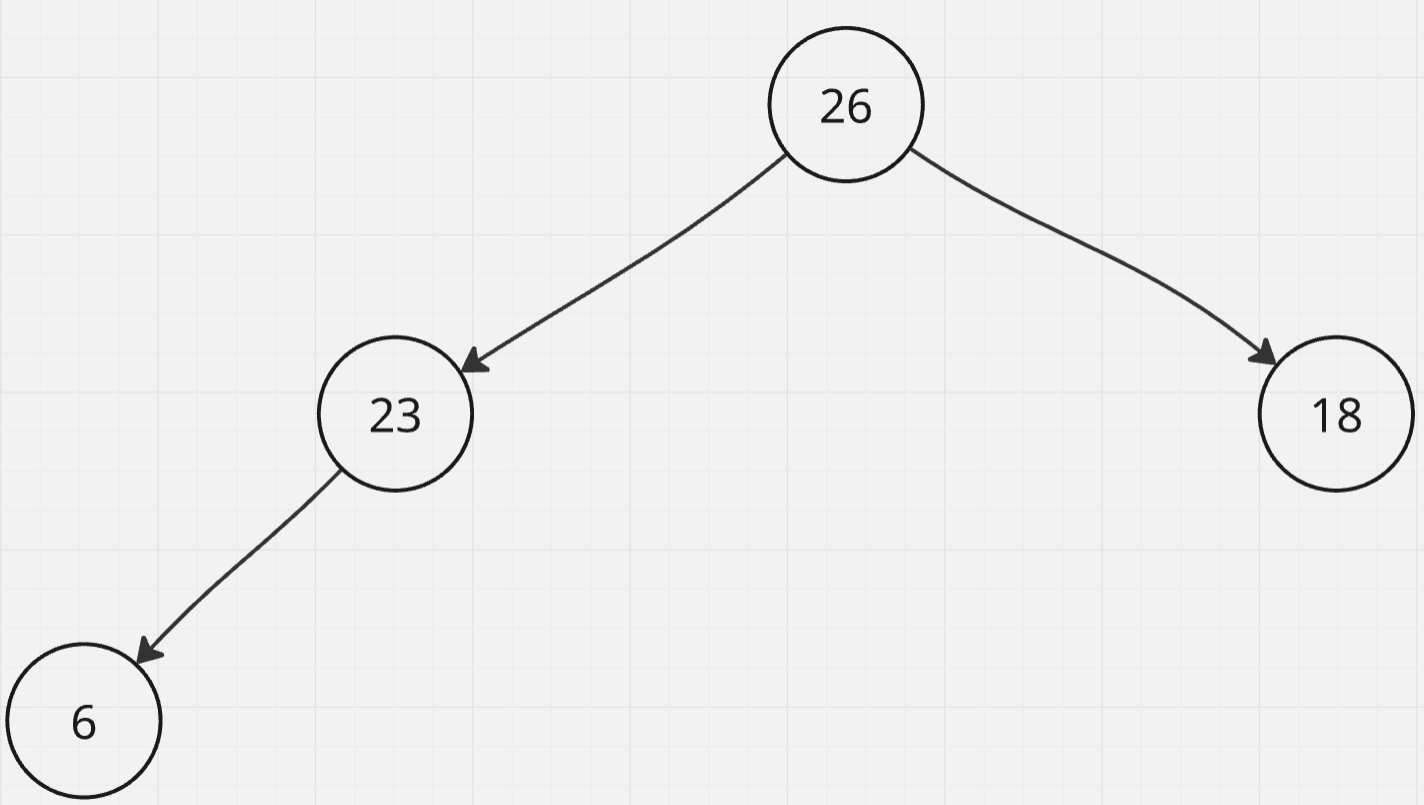


1. heapify i=1:



* + Дети: 23, 26 → max=26 → swap 18↔26 →  
    [26, 23, 18, 6, 64, 70, 77, 82]
  + i=3 (18): нет детей внутри heap\_size=4.

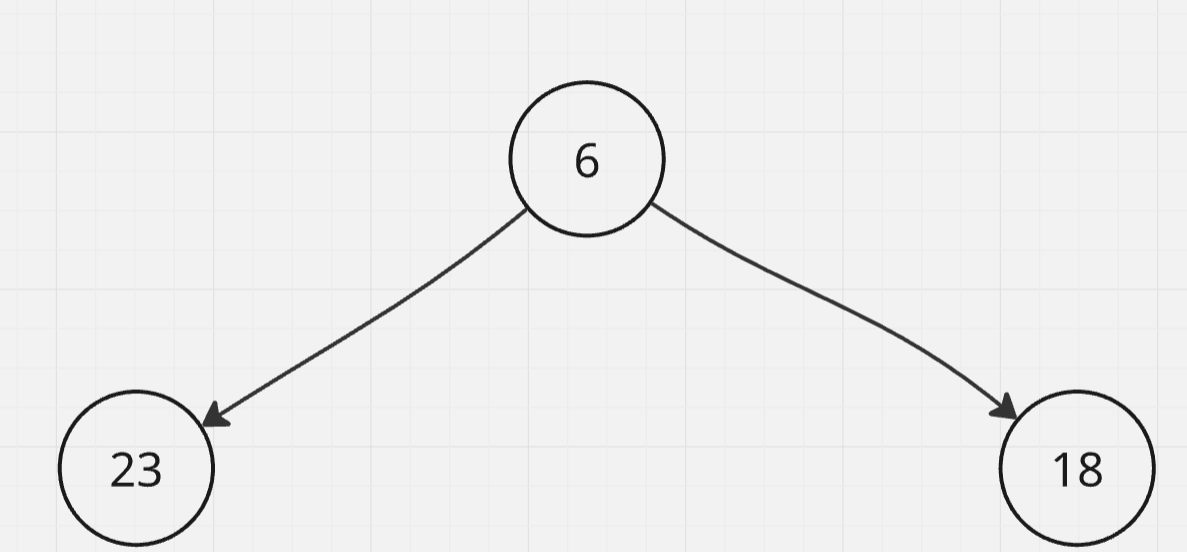
Куча (size 4):



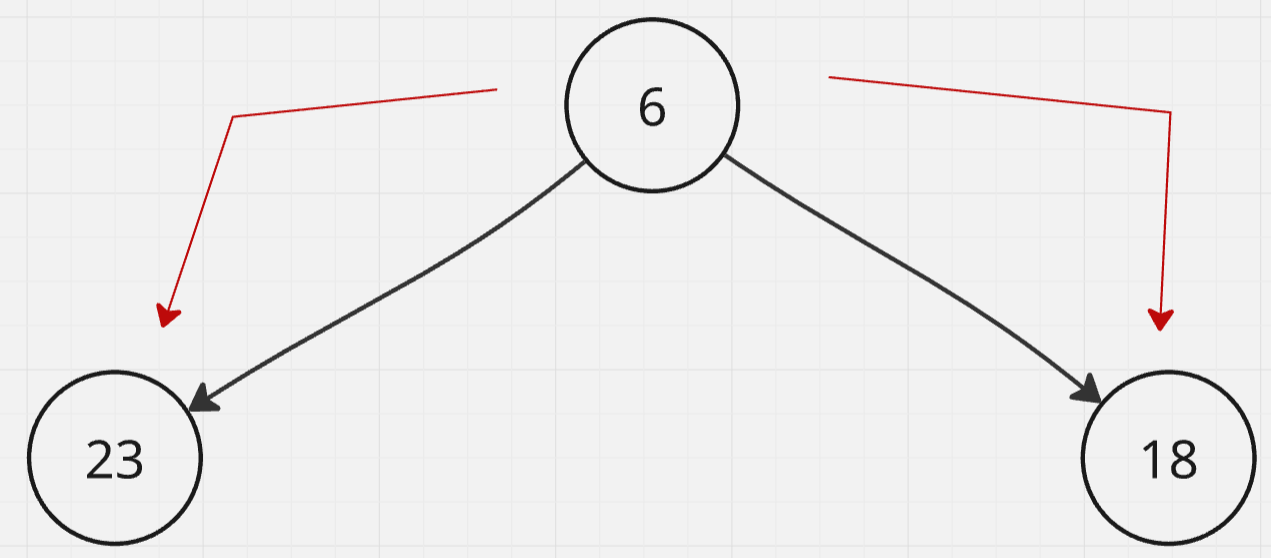
Массив:  
[26, 23, 18, 6, 64, 70, 77, 82]

**Шаг 2.5: heap\_size = 4**

1. swap A[1]=26 ↔ A[4]=6  
   → [6, 23, 18, 26, 64, 70, 77, 82]
2. heap\_size→3

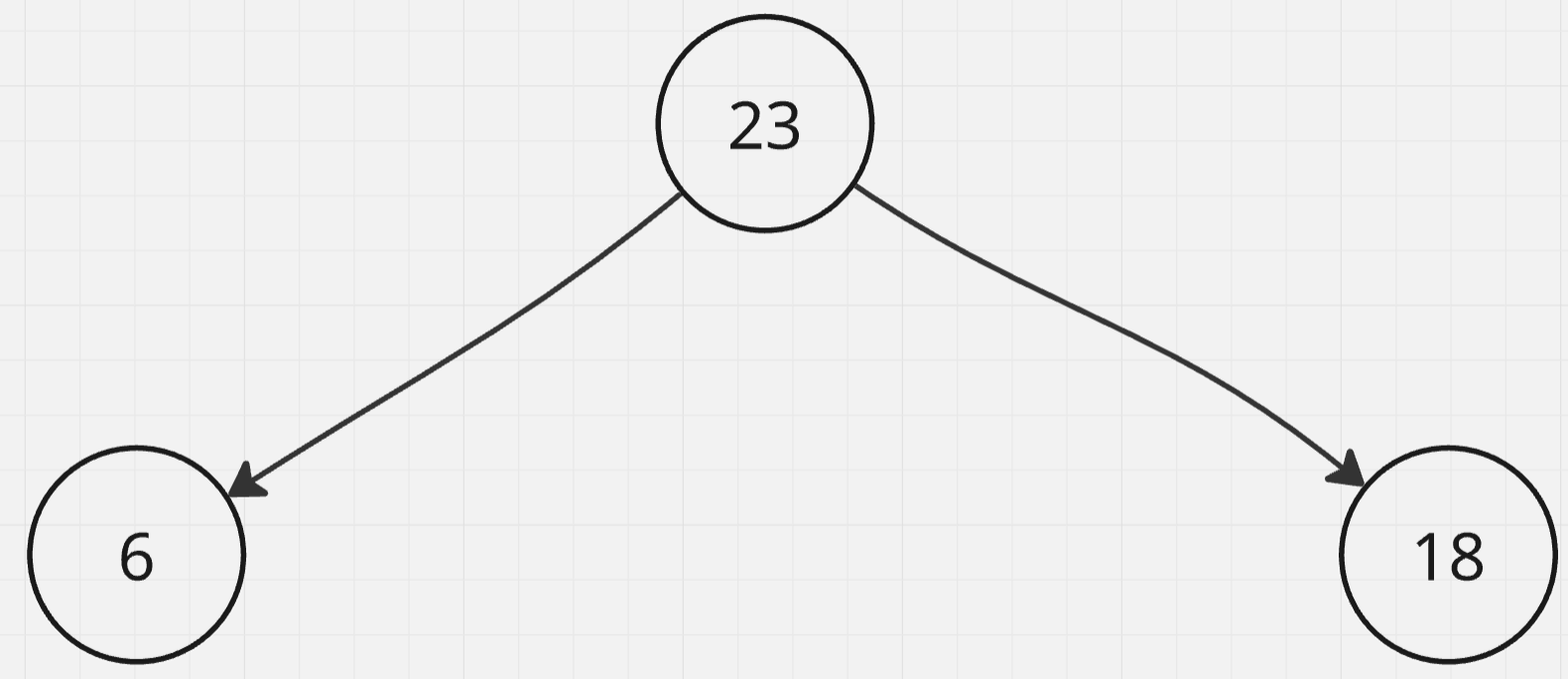


1. heapify i=1:



* + Дети: 23, 18 → max=23 → swap 6↔23 →  
    [23, 6, 18, 26, 64, 70, 77, 82]

Куча (size 3):

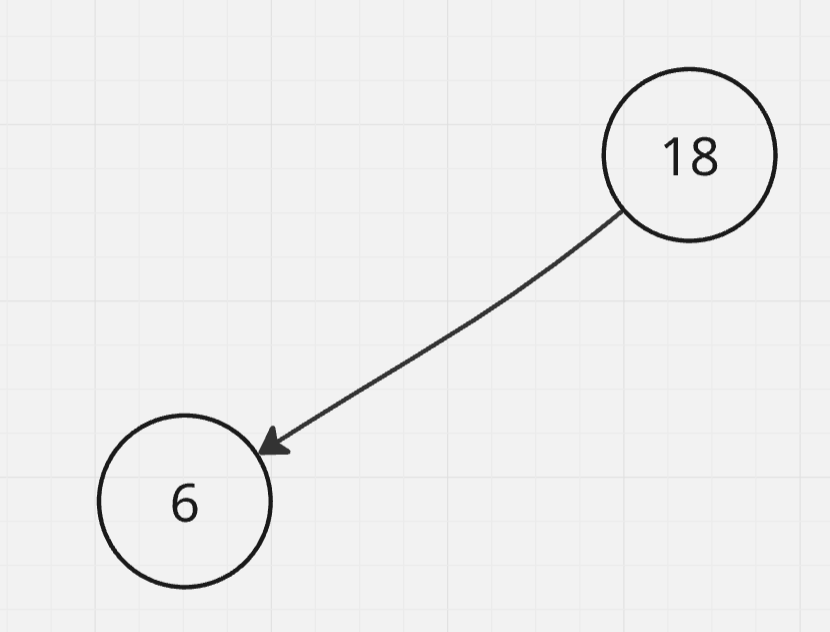


Массив:

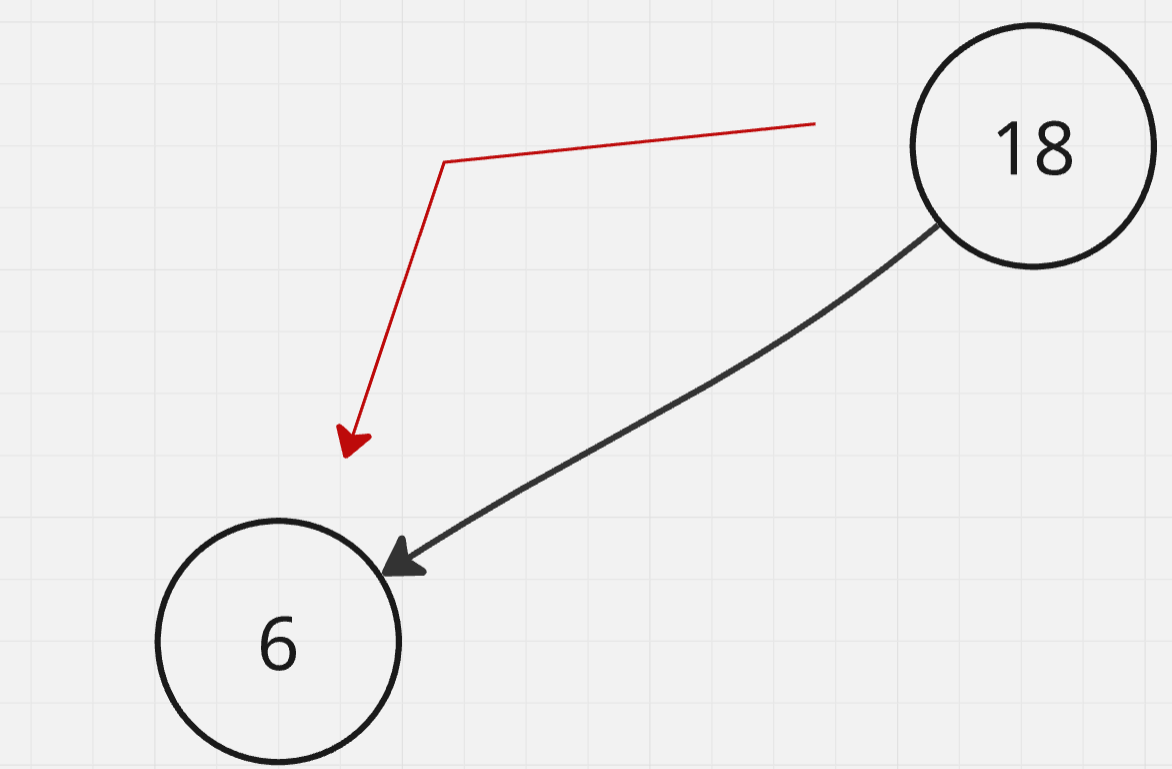
[23, 6, 18, 26, 64, 70, 77, 82]

**Шаг 2.6: heap\_size = 3**

1. swap A[1]=23 ↔ A[3]=18  
   → [18, 6, 23, 26, 64, 70, 77, 82]
2. heap\_size→2

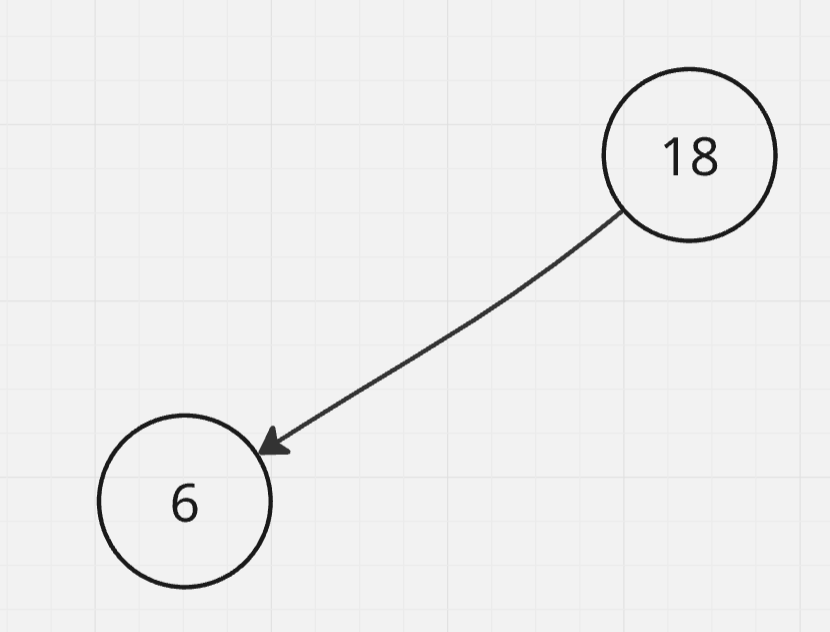


1. heapify i=1:



* + Дети: единственный A[2]=6 → 18 ≥ 6 → без изменений.

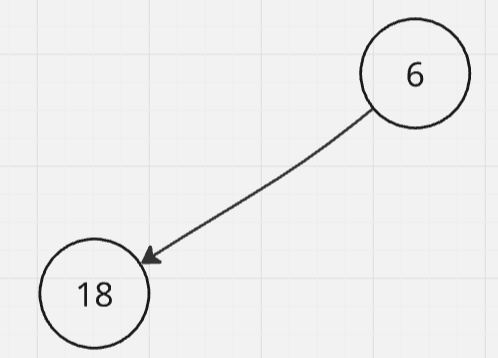
Куча (size 2):



Массив:  
[18, 6, 23, 26, 64, 70, 77, 82]

**Шаг 2.7: heap\_size = 2**

1. swap A[1]=18 ↔ A[2]=6  
   → [6, 18, 23, 26, 64, 70, 77, 82]

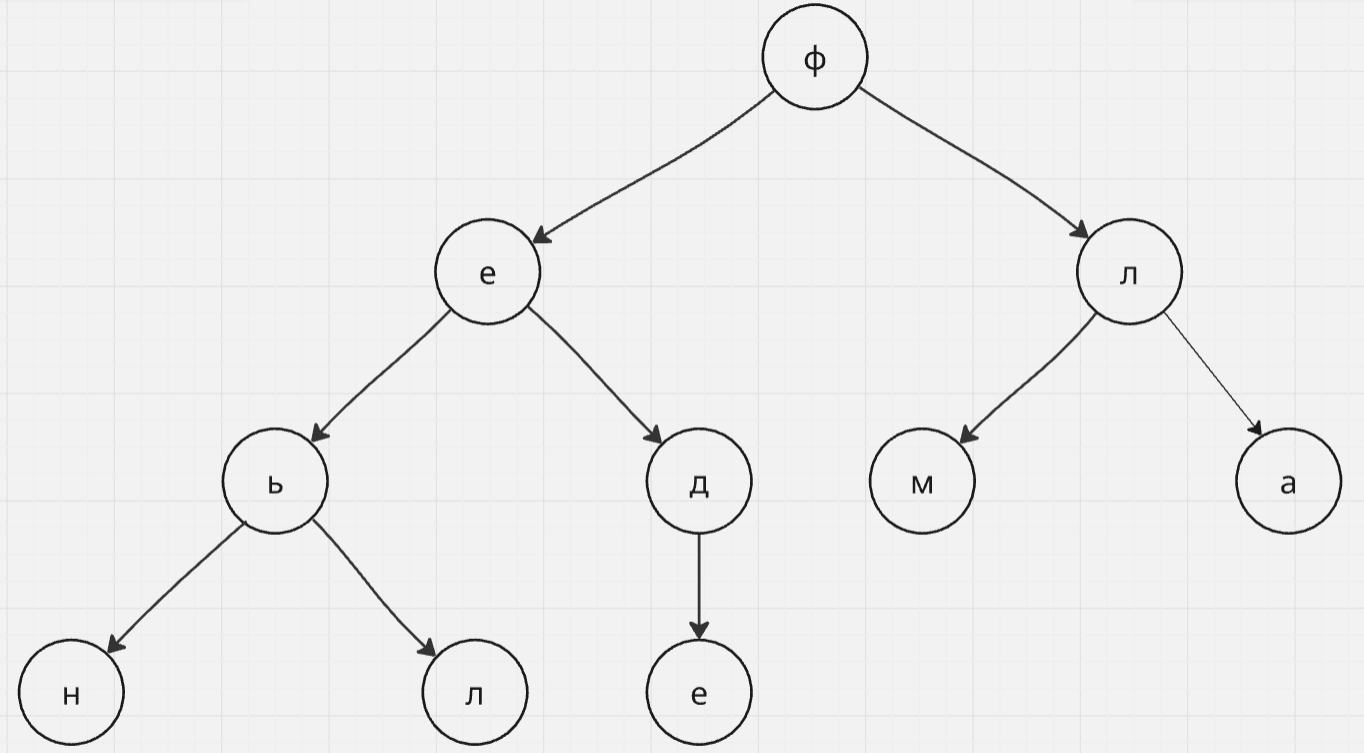
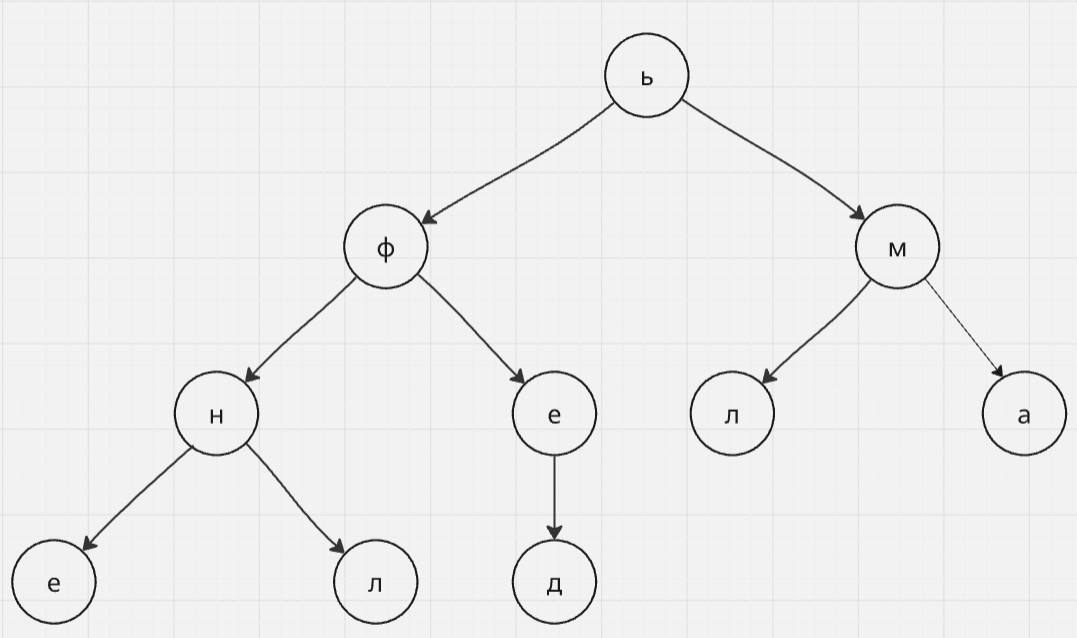
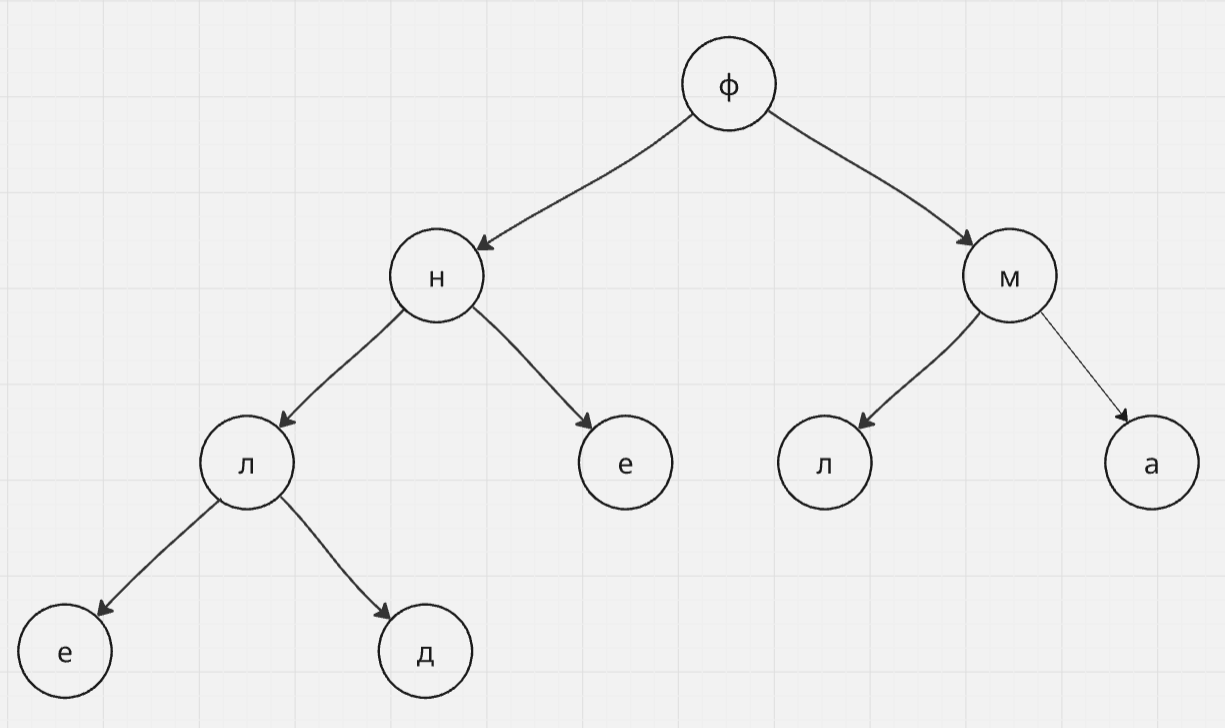


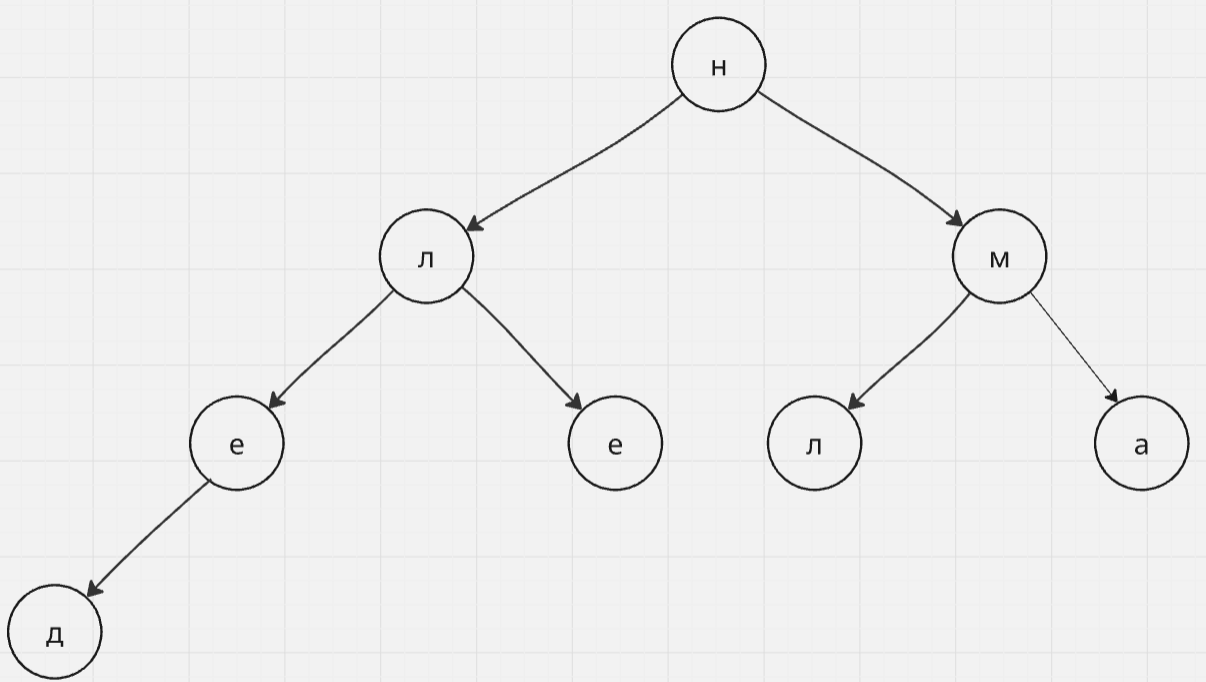
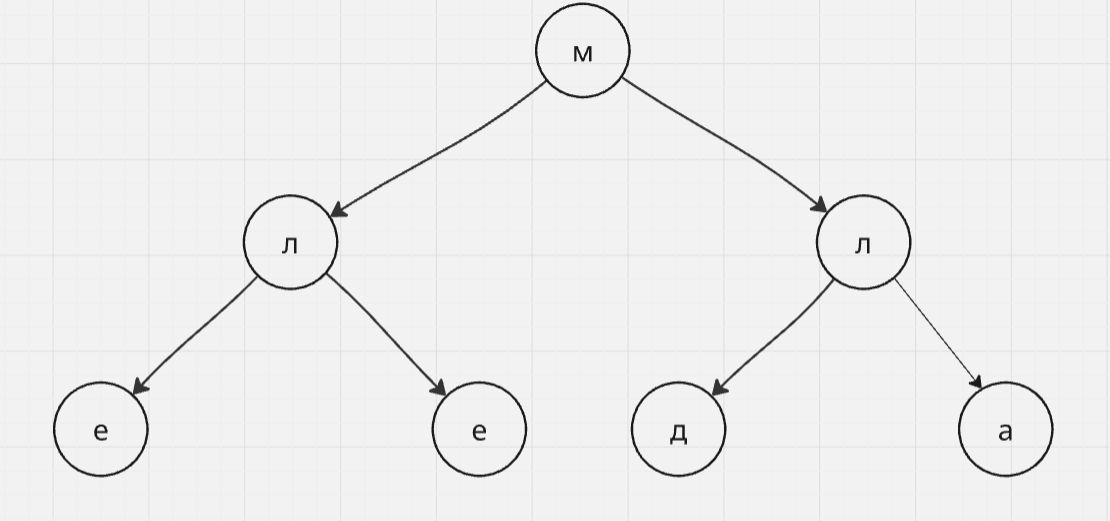
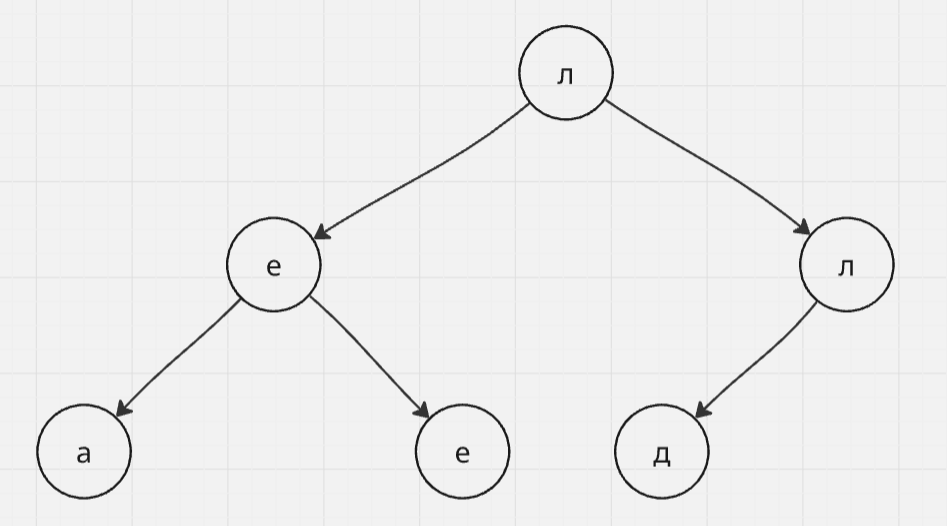
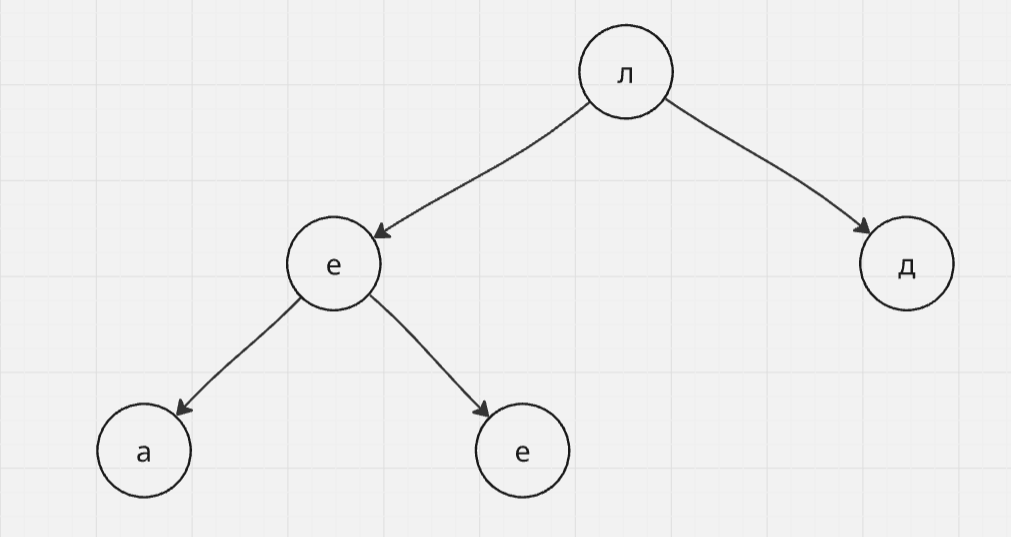
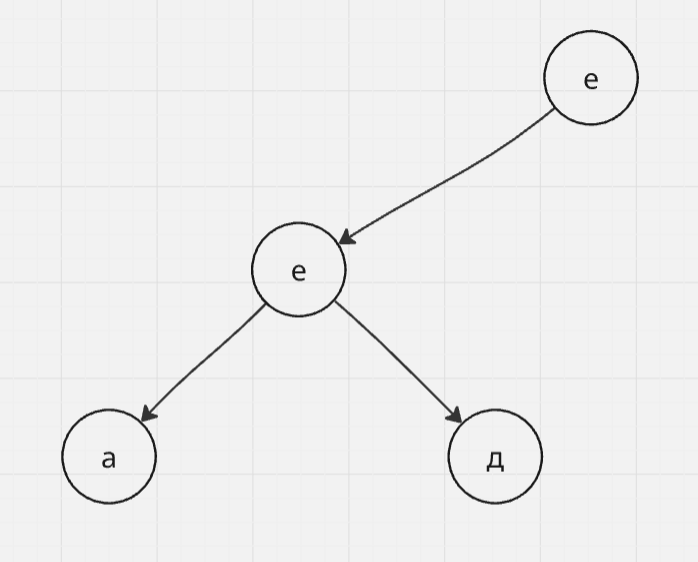
1. heap\_size→1 → сортировка закончена.

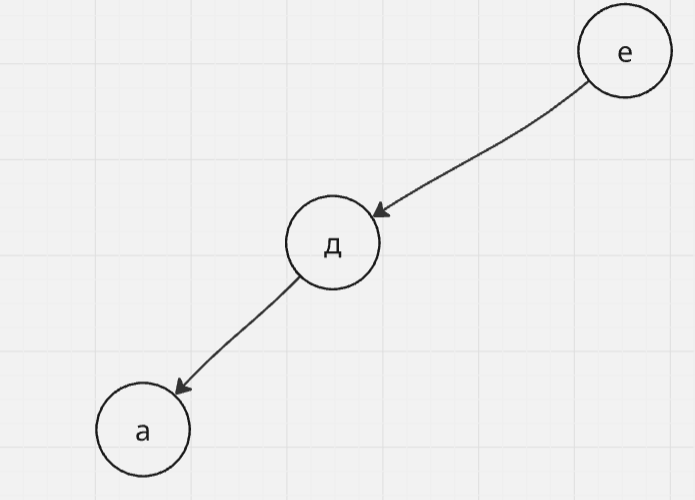
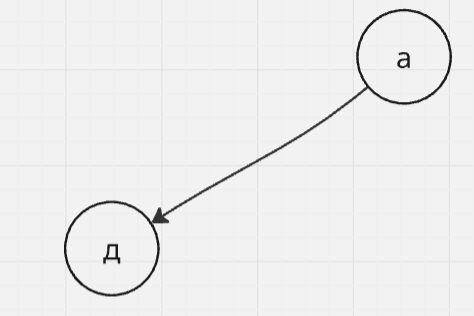
**Итог**[6, 18, 23, 26, 64, 70, 77, 82]

— ровно отсортированный по возрастанию массив.

**Для буквенного массива**

****  

Итог:  
 ['а', 'д', 'е', 'е', 'л', 'л', 'м', 'н', 'ф', 'ь']

function B = heapsort(B)

% HEAPSORT Сортировка массива-матрицы B по первому столбцу методом "пирамидальной сортировки"

n = size(B,1);

% 1) Построение мин-кучи: вставляем по одному элементу

for i = 2 : n

B = heapifyup1(B, i);

end

% 2) Извлечение элементов из кучи в конец массива

for l = n : -1 : 2

% Меняем корень (минимум) с последним элементом текущей кучи

B([1 l], :) = B([l 1], :);

% Восстанавливаем мин-кучу на префиксе [1..l-1]

B = heapifydownN1(B, 1, l-1);

end

function B = heapifydownN1(B, i, l)

% HEAPIFYDOWNN1 Просеивание вниз для мин-кучи

left = 2 \* i;

if left > l

return;

end

right = left + 1;

smallest = i;

if left <= l && B(left,1) < B(smallest,1)

smallest = left;

end

if right <= l && B(right,1) < B(smallest,1)

smallest = right;

end

if smallest ~= i

B([i smallest], :) = B([smallest i], :);

B = heapifydownN1(B, smallest, l);

end

end

function B = heapifyup1(B, i)

% HEAPIFYUP1 Просеивание вверх для мин-кучи

if i == 1

return;

end

parent = floor(i/2);

if B(i,1) < B(parent,1)

B([i parent], :) = B([parent i], :);

B = heapifyup1(B, parent);

end

end

end

% heapsort([70; 77; 82; 64; 18; 6; 26; 23])

# Быстрая сортировка

Быстрая сортировка (Quick Sort) — это рекурсивный алгоритм «разделяй и властвуй», работающий в среднем за O(nlog(n)) времени. Его суть состоит в следующем:

1. **Выбор опорного элемента (pivot).**  
   Из массива выбирают один элемент — «опору». Популярные стратегии:
   * Первый или последний элемент;
   * Случайный элемент;
   * «Медиана трёх» (среднее из первого, среднего и последнего).
2. **Разбиение (partition).**  
   Все элементы массива перегруппировываются так, чтобы:
   * Слева от опоры оказались элементы ;
   * Справа от опоры — элементы .

После этой стадии опорный элемент встаёт на своё окончательное (правильное) место.

1. **Рекурсивная сортировка частей.**  
   Отдельно рекурсивно применяют Quick Sort к левой и правой части (подмассивам) относительно позиции опоры.
2. **Базовый случай.**  
   Если размер подмассива меньше двух, возвращаем его без изменений (одно- или пустой массив уже отсортирован).

**1. Первое разбиение (весь массив)**

* **Подмассив**: low = 0, high = 7
* **Pivot** = A[7] = 23
* Идём по j=0…6, поддерживая индекс i (начинаем i = low – 1 = –1):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| j | A[j] | Сравнение с 23 | i | Массив после операции |
| 0 | 70 | 70 > 23 → skip | -1 | [70, 77, 82, 64, 18, 6, 26, 23] |
| 1 | 77 | 77 > 23 → skip | -1 |  |
| 2 | 82 | 82 > 23 → skip | -1 |  |
| 3 | 64 | 64 > 23 → skip | -1 |  |
| 4 | 18 | 18 ≤ 23 → i=0; swap A[0]⇆A[4] | 0 | [18, 77, 82, 64, 70, 6, 26, 23] |
| 5 | 6 | 6 ≤ 23 → i=1; swap A[1]⇆A[5] | 1 | [18, 6, 82, 64, 70, 77, 26, 23] |
| 6 | 26 | 26 > 23 → skip | 1 |  |

После цикла меняем A[i+1]=A[2] и pivot A[7]:

[18, 6, 23, 64, 70, 77, 26, 82]

Теперь 23 на своём окончательном месте (индекс 2).

Разбили на две части:

* Левый подмассив: [18, 6] (индексы 0…1)
* Правый подмассив: [64, 70, 77, 26, 82] (индексы 3…7)

2. Сортировка левой части [18, 6]

* **low = 0**, **high = 1**, **pivot = A[1] = 6**
* i = –1; j = 0: 18 > 6 → skip.
* Меняем A[i+1]=A[0] и pivot A[1]:  
  [6, 18, 23, 64, 70, 77, 26, 82]

Обе «части» тривиальны (размер < 2).

**3. Сортировка правой части [64, 70, 77, 26, 82]**

**3.1 Разбиение подмассива (индексы 3…7)**

* **pivot = 82**; все 4 элемента ≤ 82, поэтому после разбиения массив не меняется, а 82 переходит в позицию 7 (уже там).
* Две части:
  + Левый: [64, 70, 77, 26] (3…6)
  + Правый: пустой

**3.2 Сортировка [64, 70, 77, 26] (3…6)**

* **pivot = A[6] = 26**
* i = 2; j=3…5: все значения > 26 → i остаётся 2.
* Меняем A[3] и A[6]:  
  [6, 18, 23, 26, 70, 77, 64, 82]
* Две части:
  + Левый: пусто
  + Правый: [70, 77, 64] (4…6)

**3.3 Сортировка [70, 77, 64] (4…6)**

* **pivot = A[6] = 64**
* j=4,5: 70>64, 77>64 → i stays = 3.
* Меняем A[4] и A[6]:  
  [6, 18, 23, 26, 64, 77, 70, 82]
* Левый: пусто; Правый: [77, 70] (5…6)

**3.4 Сортировка [77, 70] (5…6)**

* **pivot = A[6] = 70**
* j=5: 77>70 → skip; swap A[5] и A[6]:

[6, 18, 23, 26, 64, 70, 77, 82]

**4. Результат**

Все рекурсивные вызовы завершены — получаем отсортированный массив:

[6, 18, 23, 26, 64, 70, 77, 82]

Таким образом, Quick Sort за 4 уровня рекурсии и последовательных разбиений упорядочил исходный массив.

**Быстрая сортировка бля буквенного массива**

[ ф е л ь д м а н л е ]

→ (pivot = е)  
[ е д а е ф м л н л ь ]

→ (сортировка левой части [е д а])  
[ а д е е ф м л н л ь ]

→ (pivot правой части = ь; без изменений)  
[ а д е е ф м л н л ь ]

→ (pivot в подмассиве [е ф м л н л] = л)  
[ а д е е л л ф н м ь ]

→ (pivot в подмассиве [ф н м] = м)  
[ а д е е л л м н ф ь ]

**Рекурсивная Функция**

function y = quick\_sort(x)

% QUICK\_SORT Рекурсивная реализация быстрой сортировки (Quick Sort).

n = length(x); % Определяем размер входного массива

% Базовый случай: если в массиве 0 или 1 элемент, он уже отсортирован

if n < 2 y = x; return

end

% Выбираем опорный элемент (pivot): здесь — последний элемент вектора

pivot = x(n);

% Инициализируем два пустых подмассива для элементов < pivot и >= pivot

x1 = []; % для элементов меньше pivot

x2 = []; % для элементов >= pivot

% Разбиение: перебираем все элементы, кроме последнего (pivot)

for i = 1:n-1

if x(i) < pivot

x1 = [x1, x(i)]; % добавляем в «левый» подмассив

else

x2 = [x2, x(i)]; % добавляем в «правый» подмассив

end

end

% Рекурсивно сортируем левый и правый подмассивы

% Объединяем: отсортированные меньшие, затем pivot, затем отсортированные большие

y = [quick\_sort(x1), pivot, quick\_sort(x2)];

end

**>> quick\_sort([70 77 82 64 18 6 26 23])**

**ans =**

**6 18 23 26 64 70 77 82**

**>> unicodeToLetters(quick\_sort(lettersToUnicode(['фельдманле'])))**

**ans =**

**'адееллмнфь'**

function y = quick\_sort\_iter(x)

% y = QUICK\_SORT\_ITER(x) возвращает отсортированный по возрастанию вектор y,

% содержащий те же элементы, что и входной вектор x, без рекурсивных вызовов.

y = x; % Рабочая копия массива

n = length(y); % Размер массива

if n < 2 return % Если элементов меньше 2, сразу выходим

end

% Стек для хранения поддиапазонов [left, right]

stack = zeros(n, 2);

top = 0; % Указатель на вершину стека

% Инициализируем стек полным диапазоном

top = top + 1;

stack(top, :) = [1, n];

% Пока есть диапазоны для обработки

while top > 0

% Извлекаем границы текущего подмассива

range = stack(top, :);

top = top - 1;

left = range(1);

right = range(2);

if left < right

% Выбираем опорный элемент: здесь — последний в поддиапазоне

pivot = y(right);

i = left - 1;

% Перестановка элементов: < pivot влево, >= pivot — вправо

for j = left:right-1

if y(j) < pivot

i = i + 1;

% Обмен y(i) и y(j)

tmp = y(i);

y(i) = y(j);

y(j) = tmp;

end

end

% Поместить pivot на своё место (между «меньшими» и «большими»)

pivotPos = i + 1;

y(right) = y(pivotPos);

y(pivotPos) = pivot;

% Добавляем в стек два новых поддиапазона, которые нужно отсортировать

% (можно добавлять в любом порядке, здесь — сначала левый, потом правый)

% Правый поддиапазон: элементы после pivot

top = top + 1;

stack(top, :) = [pivotPos + 1, right];

% Левый поддиапазон: элементы до pivot

top = top + 1;

stack(top, :) = [left, pivotPos - 1];

end

end

end

**>> quick\_sort\_iter([70 77 82 64 18 6 26 23])**

**ans =**

**6 18 23 26 64 70 77 82**

**>> unicodeToLetters(quick\_sort\_iter(lettersToUnicode(['фельдманле'])))**

**ans =**

**'адееллмнфь'**

## Сортировка слиянием

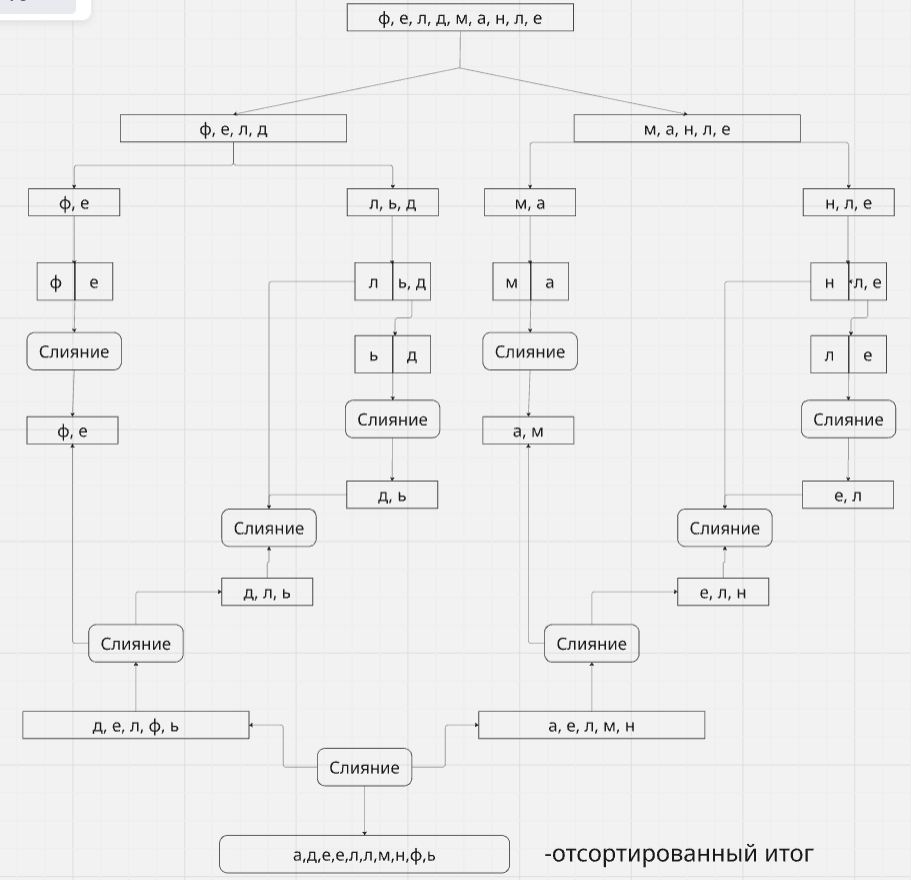
Это классический алгоритм «разделяй и властвуй» с асимптотикой O(n\*log(n)). Основная идея:

1. **Разбиение (Divide).** Рекурсивно делим массив пополам, пока не дойдём до «массивов» длины 1.
2. **Слияние (Conquer).** Два уже отсортированных подмассива сливаем в один отсортированный, сравнивая первые (минимальные) элементы обоих и последовательно формируя новый массив.
3. Повторяем слияние «снизу вверх» по рекурсивному дереву до исходного массива.

Пример 1.

[70 77 82 64 18 6 26 23]

1. Первый уровень разбиения  
   [70 77 82 64] [18 6 26 23]
2. **Второй уровень разбиения**[70 77] [82 64] [18 6] [26 23]
3. **Третий уровень (до единичных массивов)**[70] [77] [82] [64] [18] [6] [26] [23]
4. **Слияние единичных массивов**



function y = MergeSort(x)

n = length(x);

if(n > 2)

% Return Condition

y=x;

return;

end

n\_2 = int32(n/2);

x1=MergeSort(x(1:n\_2));

x2=MergeSort(x(n\_2+1:n));

n1 = length(x1);

n2 = length(x2);

count\_x1 = 1;

count\_x2 = 1;

y = [];

while(count\_x1 <= n1)

if(count\_x2 > n2)

y = [y x1(count\_x1)];

count\_x1 = count\_x1+1;

continue;

end

if(x1(count\_x1) < x2(count\_x2))

y = [y x1(count\_x1)];

count\_x1 = count\_x1+1;

else

y = [y x2(count\_x2)];

count\_x2 = count\_x2+1;

end

end

for j=count\_x2:n2

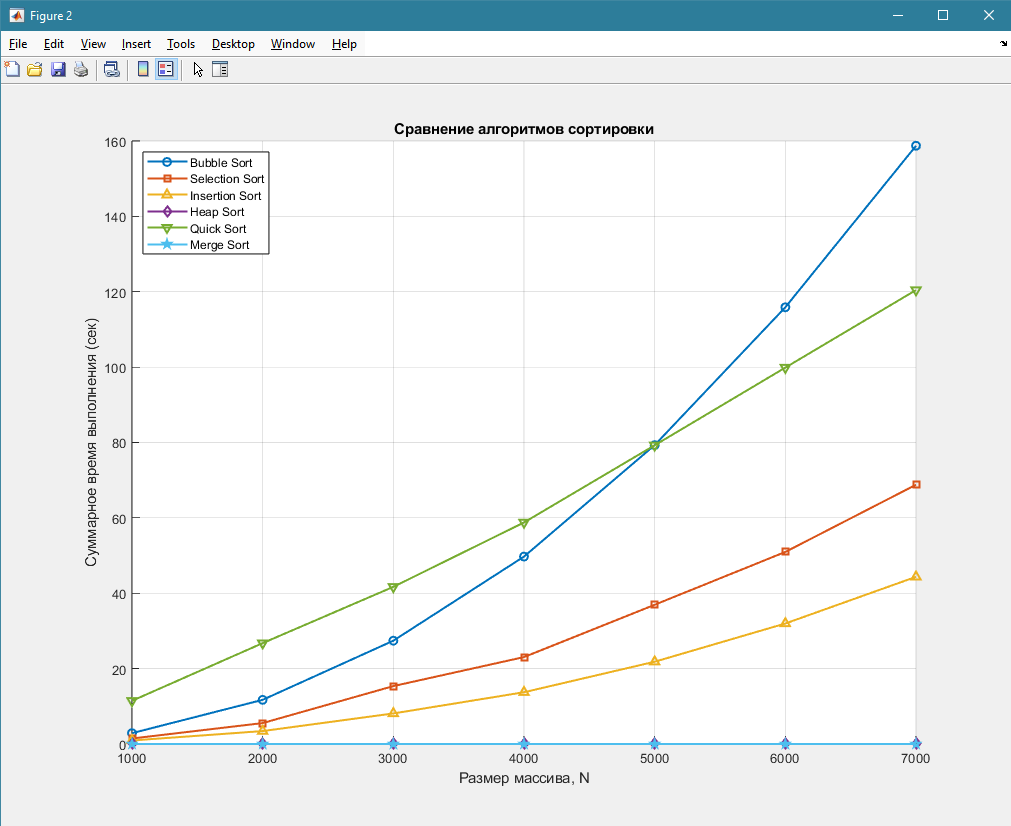
y = [y x2(j)];

end

end

# Анализ

**Для списков размерами 1000, 2000, 3000, 4000, 5000, 6000, 7000**



=== Тестирование Bubble Sort ===

Bubble Sort | N= 1000 : Суммарное время = 2.875727 сек

Bubble Sort | N= 2000 : Суммарное время = 11.720529 сек

Bubble Sort | N= 3000 : Суммарное время = 27.408734 сек

Bubble Sort | N= 4000 : Суммарное время = 49.731046 сек

Bubble Sort | N= 5000 : Суммарное время = 79.305478 сек

Bubble Sort | N= 6000 : Суммарное время = 115.879751 сек

Bubble Sort | N= 7000 : Суммарное время = 158.735647 сек

=== Тестирование Selection Sort ===

Selection Sort | N= 1000 : Суммарное время = 1.458289 сек

Selection Sort | N= 2000 : Суммарное время = 5.550063 сек

Selection Sort | N= 3000 : Суммарное время = 15.363115 сек

Selection Sort | N= 4000 : Суммарное время = 23.060688 сек

Selection Sort | N= 5000 : Суммарное время = 36.945767 сек

Selection Sort | N= 6000 : Суммарное время = 50.961314 сек

Selection Sort | N= 7000 : Суммарное время = 68.790230 сек

=== Тестирование Insertion Sort ===

Insertion Sort | N= 1000 : Суммарное время = 0.930328 сек

Insertion Sort | N= 2000 : Суммарное время = 3.431097 сек

Insertion Sort | N= 3000 : Суммарное время = 8.106391 сек

Insertion Sort | N= 4000 : Суммарное время = 13.732629 сек

Insertion Sort | N= 5000 : Суммарное время = 21.828553 сек

Insertion Sort | N= 6000 : Суммарное время = 31.973750 сек

Insertion Sort | N= 7000 : Суммарное время = 44.357871 сек

=== Тестирование Heap Sort ===

Heap Sort | N= 1000 : Суммарное время = 0.004360 сек

Heap Sort | N= 2000 : Суммарное время = 0.006832 сек

Heap Sort | N= 3000 : Суммарное время = 0.006130 сек

Heap Sort | N= 4000 : Суммарное время = 0.005504 сек

Heap Sort | N= 5000 : Суммарное время = 0.003697 сек

Heap Sort | N= 6000 : Суммарное время = 0.006783 сек

Heap Sort | N= 7000 : Суммарное время = 0.005061 сек

=== Тестирование Quick Sort ===

Quick Sort | N= 1000 : Суммарное время = 11.459996 сек

Quick Sort | N= 2000 : Суммарное время = 26.765093 сек

Quick Sort | N= 3000 : Суммарное время = 41.702844 сек

Quick Sort | N= 4000 : Суммарное время = 58.783769 сек

Quick Sort | N= 5000 : Суммарное время = 79.265751 сек

Quick Sort | N= 6000 : Суммарное время = 99.875159 сек

Quick Sort | N= 7000 : Суммарное время = 120.448698 сек

=== Тестирование Merge Sort ===

Merge Sort | N= 1000 : Суммарное время = 0.000879 сек

Merge Sort | N= 2000 : Суммарное время = 0.000616 сек

Merge Sort | N= 3000 : Суммарное время = 0.000634 сек

Merge Sort | N= 4000 : Суммарное время = 0.000852 сек

Merge Sort | N= 5000 : Суммарное время = 0.000745 сек

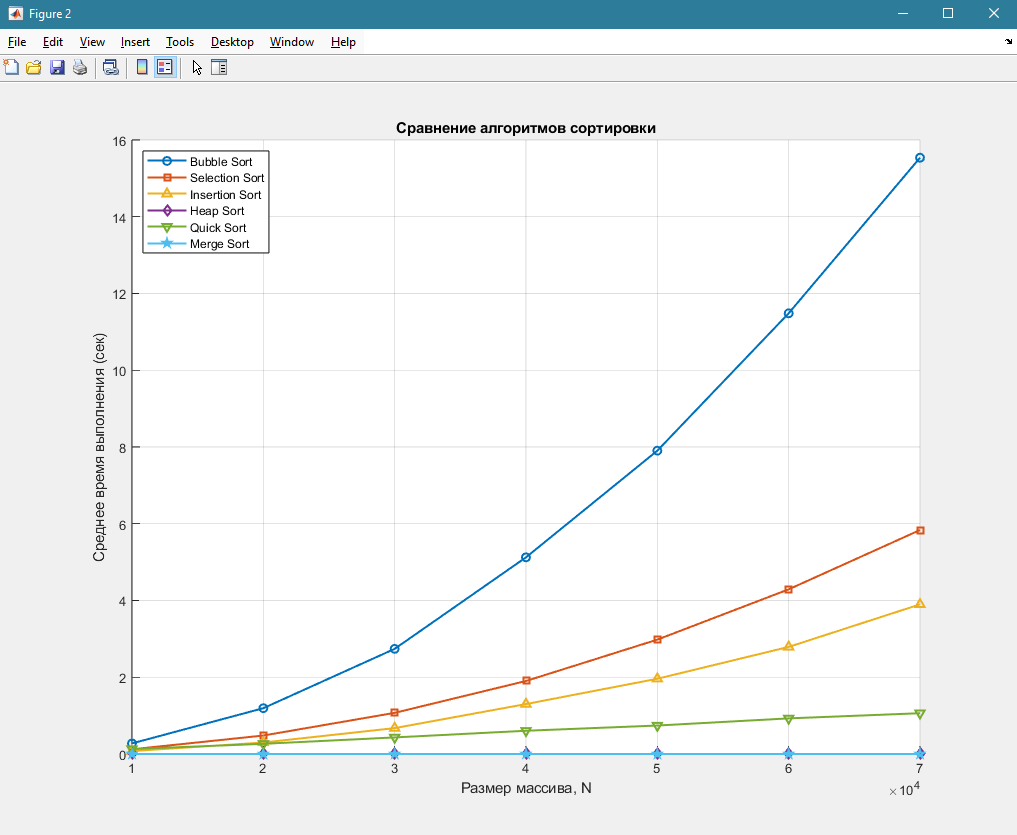
Merge Sort | N= 6000 : Суммарное время = 0.000761 сек

Merge Sort | N= 7000 : Суммарное время = 0.000669 сек

Сводная таблица суммарного времён (строки — алгоритмы):

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | N1000 | N2000 | N3000 | N4000 | N5000 | N6000 | N7000 |
| Bubble Sort | 2.875727 | 11.720529 | 27.4087 | 49.73104 | 79.30547 | 115.87975 | 158.73564 |
| Selection Sort | 1.458289 | 5.550063 | 15.36311 | 23.060688 | 36.945767 | 50.961314 | 68.790230 |
| Insertion Sort | 0.930328 | 3.431097 | 8.106391 | 13.732629 | 21.828553 | 31.973750 | 44.357871 |
| Heap Sort | 0.004360 | 0.006832 | 0.006130 | 0.005504 | 0.003697 | 0.006783 | 0.005061 |
| Quick Sort | 26.76509 | 41.702844 | 58.78376 | 79.265751 | 79.265751 | 99.875159 | 120.448698 |
| Merge Sort | 0.000879 | 0.000616 | 0.000634 | 0.000852 | 0.000745 | 0.000761 | 0.000669 |

**Для списков размерами 10000, 20000, 30000, 40000, 50000, 60000, 70000**



=== Тестирование Bubble Sort ===

Bubble Sort | N=10000 : среднее время = 0.275815 сек

Bubble Sort | N=20000 : среднее время = 1.193870 сек

Bubble Sort | N=30000 : среднее время = 2.739907 сек

Bubble Sort | N=40000 : среднее время = 5.125126 сек

Bubble Sort | N=50000 : среднее время = 7.904384 сек

Bubble Sort | N=60000 : среднее время = 11.484275 сек

Bubble Sort | N=70000 : среднее время = 15.538071 сек

=== Тестирование Selection Sort ===

Selection Sort | N=10000 : среднее время = 0.122146 сек

Selection Sort | N=20000 : среднее время = 0.481131 сек

Selection Sort | N=30000 : среднее время = 1.072239 сек

Selection Sort | N=40000 : среднее время = 1.902363 сек

Selection Sort | N=50000 : среднее время = 2.980438 сек

Selection Sort | N=60000 : среднее время = 4.291067 сек

Selection Sort | N=70000 : среднее время = 5.835943 сек

=== Тестирование Insertion Sort ===

Insertion Sort | N=10000 : среднее время = 0.076035 сек

Insertion Sort | N=20000 : среднее время = 0.299125 сек

Insertion Sort | N=30000 : среднее время = 0.674686 сек

Insertion Sort | N=40000 : среднее время = 1.301519 сек

Insertion Sort | N=50000 : среднее время = 1.960602 сек

Insertion Sort | N=60000 : среднее время = 2.790603 сек

Insertion Sort | N=70000 : среднее время = 3.899104 сек

=== Тестирование Heap Sort ===

Heap Sort | N=10000 : среднее время = 0.000062 сек

Heap Sort | N=20000 : среднее время = 0.000013 сек

Heap Sort | N=30000 : среднее время = 0.000019 сек

Heap Sort | N=40000 : среднее время = 0.000020 сек

Heap Sort | N=50000 : среднее время = 0.000029 сек

Heap Sort | N=60000 : среднее время = 0.000022 сек

Heap Sort | N=70000 : среднее время = 0.000026 сек

=== Тестирование Quick Sort ===

Quick Sort | N=10000 : среднее время = 0.128864 сек

Quick Sort | N=20000 : среднее время = 0.265456 сек

Quick Sort | N=30000 : среднее время = 0.431723 сек

Quick Sort | N=40000 : среднее время = 0.605565 сек

Quick Sort | N=50000 : среднее время = 0.741711 сек

Quick Sort | N=60000 : среднее время = 0.928166 сек

Quick Sort | N=70000 : среднее время = 1.062673 сек

=== Тестирование Merge Sort ===

Merge Sort | N=10000 : среднее время = 0.000048 сек

Merge Sort | N=20000 : среднее время = 0.000004 сек

Merge Sort | N=30000 : среднее время = 0.000004 сек

Merge Sort | N=40000 : среднее время = 0.000006 сек

Merge Sort | N=50000 : среднее время = 0.000006 сек

Merge Sort | N=60000 : среднее время = 0.000008 сек

Merge Sort | N=70000 : среднее время = 0.000011 сек

Сводная таблица средних времён (строки — алгоритмы, столбцы — N):

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | N10000 | N20000 | N30000 | N40000 | N50000 | N60000 | N70000 |
| Bubble Sort | 0.27582 | 1.1939 | 2.7399 | 5.1251 | 7.9044 | 11.484 | 15.538 |
| Selection Sort | 0.0015389 | 0.48113 | 1.0722 | 1.9024 | 2.9804 | 4.2911 | 5.8359 |
| Insertion Sort | 0.12215 | 0.29913 | 0.67469 | 1.3015 | 1.9606 | 2.7906 | 3.8991 |
| Heap Sort | 6.212e-05 | 1.324e-05 | 1.864e-05 | 2.048e-05 | 2.942e-05 | 2.178e-05 | 2.568e-05 |
| Quick Sort | 0.12886 | 0.26546 | 0.43172 | 0.60556 | 0.74171 | 0.92817 | 1.0627 |
| Merge Sort | 4.77e-05 | 3.58e-06 | 4.32e-06 | 5.52e-06 | 6.24e-06 | 7.58e-06 | 1.078e-05 |

# Вывод

**1. Быстрая сортировка (Quick Sort)**

Выбирается опорный элемент (pivot), затем массив разделяется на две части: элементы меньше опорного и элементы больше. Процесс повторяется рекурсивно для каждой части.

* Преимущества: Быстрая на практике для больших массивов и имеет среднюю сложность O(n log n).
* Недостатки: Худший случай O(n^2), но его можно избежать, используя случайный выбор опорного элемента.

**2. Сортировка вставками (Insertion Sort)**

Массив проходит поэлементно, и каждый элемент вставляется на правильное место среди уже отсортированных элементов.

* Преимущества: Простая и эффективная для небольших массивов или почти отсортированных данных. Сложность — O(n^2).
* Недостатки: неэффективна для больших массивов, так как требует много сравнений и перемещений.

**3. Пирамидальная сортировка (Heap Sort)**

При пирамидальной сортировке массив представляется в виде двоичной кучи, и на каждом шаге извлекается наибольший элемент из кучи, перестраивая её.

* Преимущества: имеет гарантированную сложность O(n log n) и не требует дополнительной памяти, как сортировка слиянием.
* Недостатки: Меньше подходит для очень больших объемов данных из-за сложности построения кучи.

**4. Сортировка выбором (Selection Sort)**

На каждом шаге ищется минимальный элемент и перемещается в начало массива, затем процесс повторяется для оставшейся части массива.

* Преимущества: простая реализация, но на практике используется редко. Сложность O(n^2).
* Недостатки: медленная для больших массивов, так как производит много ненужных операций.

**5. Сортировка пузырьком (Bubble Sort)**

Массив проходит несколько раз, каждый раз перемещая наибольший элемент в конец, «всплывая» его на нужное место.

* Преимущества: подходит для обучения и демонстрации работы алгоритмов. Сложность — O(n^2).
* Недостатки: один из самых медленных алгоритмов, обычно не используется на практике для реальных задач.

**6. Сортировка слиянием(Merge Sort)**

Массив проходит несколько раз, каждый раз перемещая наибольший элемент в конец, «всплывая» его на нужное место.

* Преимущества: для больших массивов: временная сложность O(n log n).Стабильная сортировка, сохраняющая порядок равных элементов.
* Недостатки: занимает больше памяти, так как создаются временные массивы для слияния.