Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Сибирский государственный университет телекоммуникаций и

информатики»

(СибГУТИ)

02.03.02 Фундаментальная информатика и информационные технологии

Профиль: Системное программное обеспечение

(очная форма обучения)

ОТЧЕТ ПО учебной ПРАКТИКЕ

в/на кафедре вычислительных систем СибГУТИ

(наименование профильной организации/структурного подразделения СибГУТИ)

Бинарные кучи. Реализация и описание

Выполнил: Куликов Д.А.

студент института ИВТ

гр. ИС-142 / /

«27» мая 2023г. (подпись)

Проверил:

Руководитель от СибГУТИ / /

«27» мая 2023г. (подпись)

Новосибирск 2023

**План-график проведения**  учебной\_\_\_\_\_\_ **практики**

Вид практики

Фамилия Имя Отчество студента

института Информатика и вычислительная техника , 2 курса, гр. ИС-142

Направление: 02.03.02 Фундаментальная информатика и информационные технологии

Код – Наименование направления (специальности)

Профиль: Системное программное обеспечение

Место прохождения практики

Объем практики: **108/3** часов/ЗЕ

Вид практики  ***учебная***

Тип практики ***научно-исследовательская работа (получение первичных навыков научно-исследовательской работы)***

Срок практики с "30" января 2023 г.

по "27" мая 2023 г.

Содержание практики\*:

|  |  |
| --- | --- |
| Наименование видов деятельности | Дата (начало – окончание) |
| 1. Общее ознакомление со структурным подразделением предприятия, вводный инструктаж по технике безопасности | 30.01.2023–01.02.2023 |
| 2. Выдача задания на практику, деление студентов на группы (если необходимо), определение конкретной индивидуальной темы, формирование плана работ | 02.02.2023–04.02.2023 |
| 3. Работа с библиотечными фондами структурного подразделения или предприятия, сбор и анализ материалов по теме практики | 06.02.2023–11.02.2023 |
| 4. Выполнение работ в соответствии с составленным планом:  – Изучил структуру данных «бинарная куча»  – Изучил структуру данных «Турнирное дерево»  – Анализировал алгоритмы и обдумывал реализацию  – Реализация алгоритмов и структур данных  – Экспериментальные замеры времени работ алгоритмов  (перечисление конкретных видов работ, связанных с выполнением поставленных задач) | 13.02.2023 – 20.05.2023 |
| 5. Анализ полученных результатов и произведенной работы  Составление отчета по практике, защита отчета | 22.05.2023–27.05.2023 |

\*В соответствии с программой практики

Руководитель от СибГУТИ / /

«28» \_\_\_\_01\_\_\_\_ 2023г. (подпись)

**ЗАДАНИЕ НА ПРАКТИКУ**

Моя тема на практику: «Бинарные кучи. Реализация и описание». В данной работе я опишу принцип работы бинарных куч и их трудоёмкость.

**ВВЕДЕНИЕ**

В первую очередь нужно разобраться, что из себя представляют деревья, ведь бинарная куча – это бинарное дерево. Корневое дерево – это структура данных, которая представляет собой совокупность связанных узлов, один из которых является корнем дерева. Ребра связывают узлы дерева и устанавливают между ними отношение «родитель-дочерний».

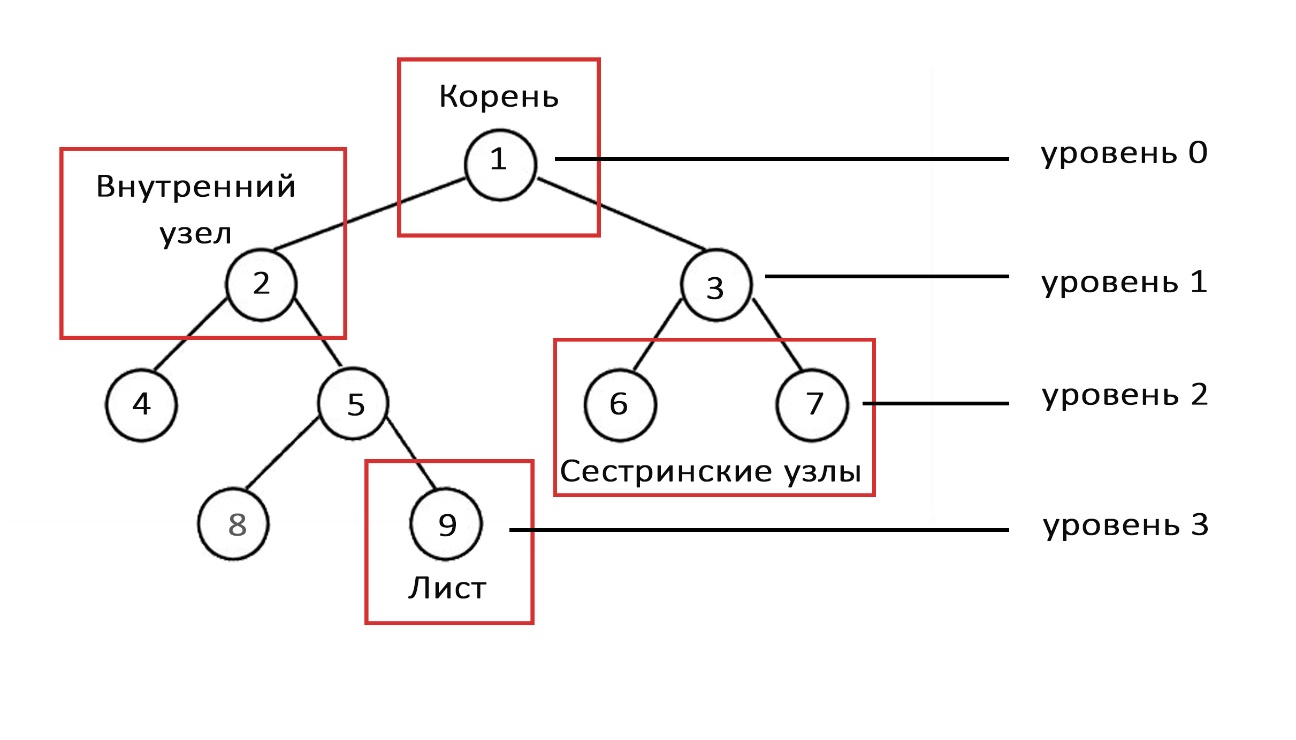


Рис. 1. Пример корневого дерева.

Узел, который не имеет дочерних вершин, называется внешним узлом, или листом. Аналогично, узел дерева, имеющий дочерние вершины называется внутренним узлом. Любой внутренний узел по отношению к своим дочерним вершинам называется родительским. Узлы, имеющие общего родителя, называются родственными, или сестринскими узлами. Любой узел в поддереве некоторой вершины называется ее потомком, а вершина для этих узлов является предком. Степень узла – это количество его дочерних узлов. Высота узла – количество ребер в длиннейшем пути от узла до листа. Высота дерева – количество ребер в длиннейшем пути от корня до листа. Глубина узла или уровень узла – количество ребер в пути от узла до корня.

Бинарное дерево (двоичное дерево) – это корневое дерево, в котором каждый узел имеет не более двух дочерних вершин (0, 1 или 2 вершины).

Полное бинарное дерево – это бинарное дерево, в котором каждый узел имеет 0 или 2 дочерних узла. Пример такого дерева приведен на рис. 1.

Завершенное бинарное дерево – это бинарное дерево, в котором каждый уровень, возможно за исключением последнего, полностью заполнен узлами, а заполнение последнего уровня осуществляется слева направо.

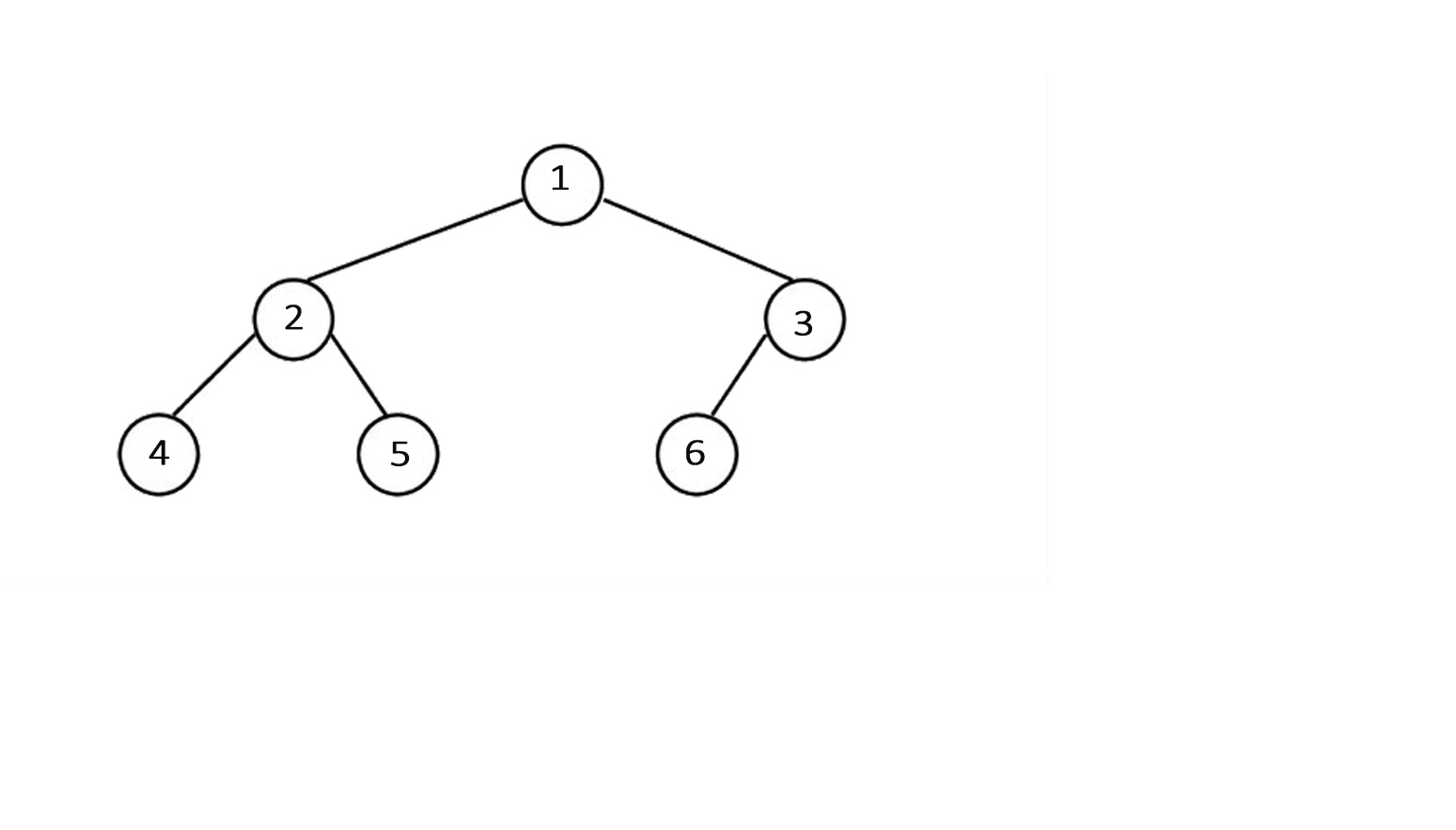


Рис. 2. Завершенное двоичное дерево.

Совершенное двоичное дерево — особый тип бинарного дерева, в котором у каждого внутреннего узла по два ребенка, а листовые вершины находятся на одном уровне (рис. 3).

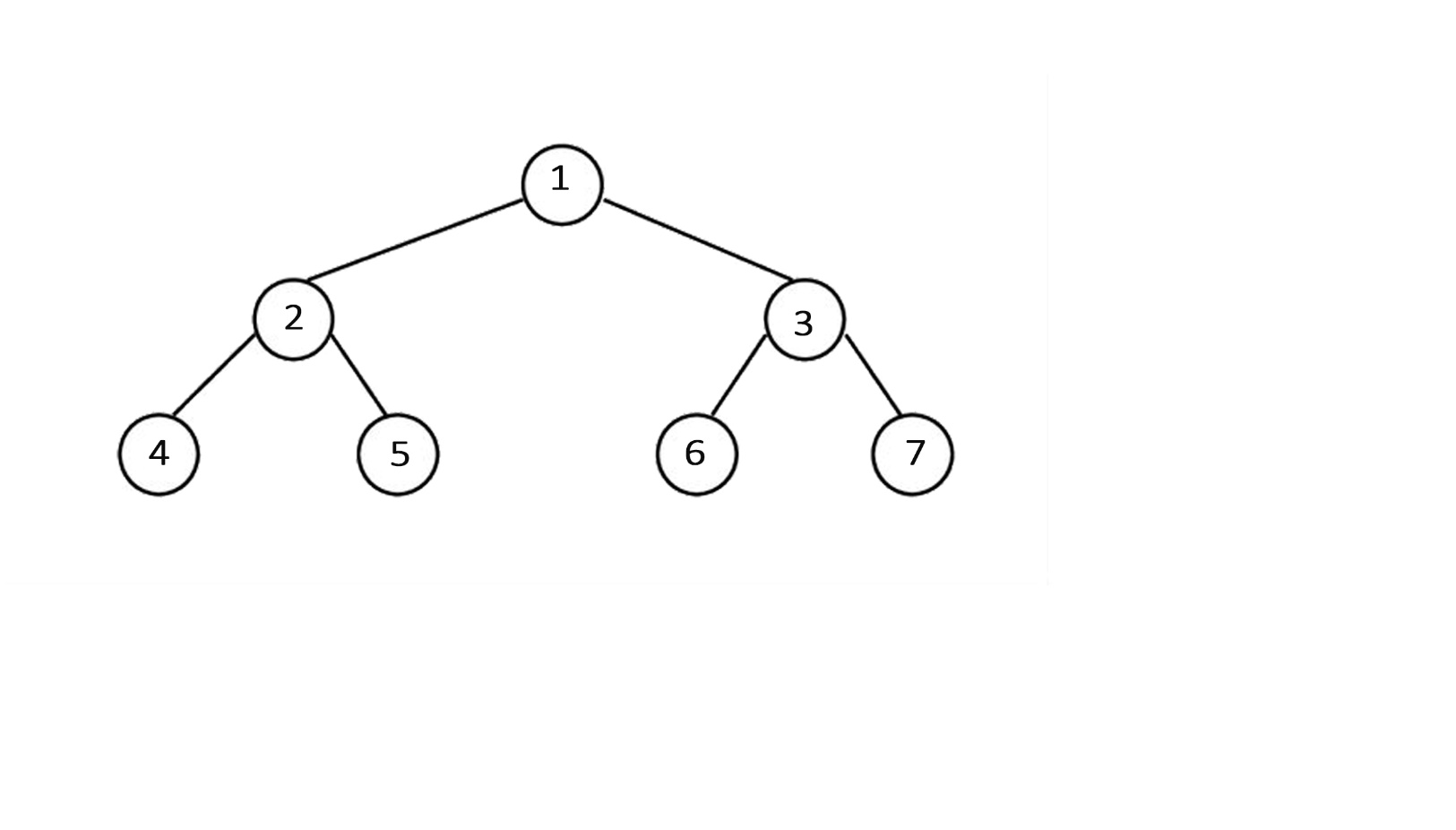


Рис. 3. Совершенное двоичное дерево

Теперь разберёмся, что такое бинарная куча. Бинарной кучей (binary heap) называется завершенное бинарное дерево, в котором каждый узел содержит ключ и значение. Ключ выступает в роли приоритета узла, а значение – это некоторые данные, ассоциированные с ключом. Структура дерева и распределение ключей по нему удовлетворяют следующим требованиям:

* Каждый уровень дерева, возможно за исключением последнего, полностью заполнен узлами, а заполнение последнего уровня осуществляется слева направо
* Ключ любого узла не меньше (не больше) ключей его потомков

Бинарные кучи (binary heap) предназначены для реализации АТД очередь с приоритетом. Они поддерживают операции поиска и удаления элемента с экстремальным значением ключа (либо минимальным, либо максимальным), но не предоставляют возможности поиска элемента по ключу.

Бинарные кучи также называют пирамидами и сортирующими деревьями. Алгоритмы этой структуры данных лежат в основе пирамидальной и турнирной сортировки.

Число уровней в завершенном бинарной куче из 𝑛 ключей равно ⌊log2 𝑛⌋ + 1.

Бинарная куча, в которой ключ любого узла не меньше ключей его потомков, называется невозрастающей бинарной кучей, или невозрастающей пирамидой (max-heap). В таком дереве ключ с максимальным значением расположен в его корне.

Аналогично, бинарная куча, в которой ключ любого узла не больше ключей его потомков, называется неубывающей бинарной кучей, или неубывающей пирамидой (min-heap). В корне дерева находится ключ с минимальным значением.

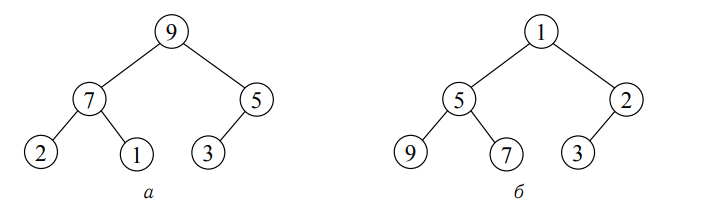


Рис. 4. a - невозрастающая куча (max-heap);

б – неубывающая куча (min-heap).

**ПРЕДСТАВЛЕНИЕ БИНАРНОЙ КУЧИ В ПАМЯТИ**

Завершенные бинарные деревья можно хранить в одномерном массиве. Все узлы дерева размещаются в ячейках массива 𝐴[0..𝑛 - 1] в порядке их обхода в ширину. Корень дерева хранится в первой ячейке массива. Для любой вершины, размещенной в ячейке 𝑖 ∈ {1, 2, . . . , 𝑛 - 1}, левый дочерний узел LEFT(𝑖) расположен в позиции 2𝑖 + 1, а правый дочерний узел RIGHT(𝑖) в позиции 2𝑖 + 2. Родительский узел PARENT(𝑖) вершины 𝑖 находится в ячейке ⌊(𝑖 – 1) /2⌋. Если значения 2𝑖 + 1 или 2𝑖 + 2 больше 𝑛, то у вершины 𝑖 отсутствует соответствующий дочерний узел.

Для хранения элементов бинарной кучи используется массив 𝐴[0..𝑚-1] из 𝑚 ячеек. Каждая ячейка 𝑖 хранит ключ 𝐴[𝑖].𝑘𝑒𝑦 и некоторое значение 𝐴[𝑖].𝑣𝑎𝑙𝑢𝑒. Очевидно, что при таком представлении максимальное число узлов в куче не превышает 𝑚. Для определенности обозначим через 𝑛 количество элементов, реально хранящихся в бинарной куче (𝑛 ≤ 𝑚).

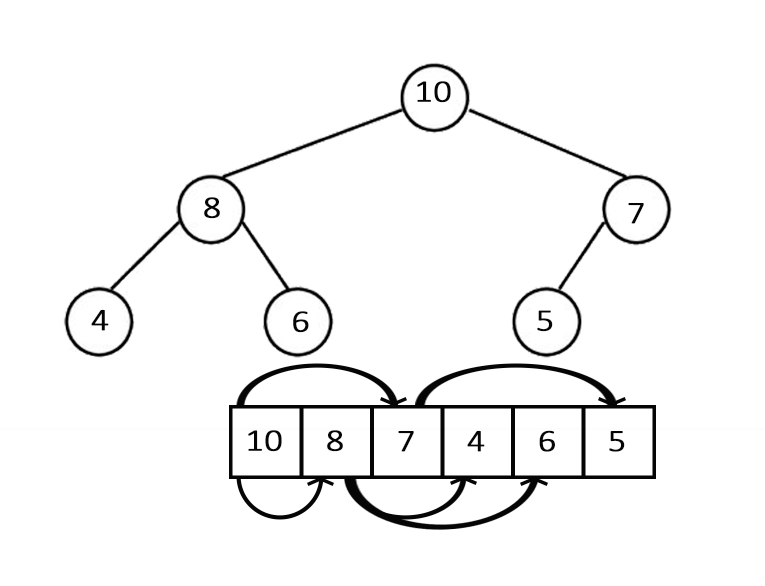


Рис. 5. Представление бинарной кучи из 6 элементов (max-heap)

в виде массива 𝐴[0..5]

**РЕАЛИЗАЦИЯ ОПЕРАЦИЙ НАД БИНАРНОЙ КУЧЕЙ**

Рассмотрим реализации операций невозрастающей бинарной кучи (max-heap). Операции неубывающей бинарной кучи реализуются аналогично.

Структура узлов кучи:

struct binary\_heap\_nodes {

int key;

int value;

};

Структура бинарной кучи:

struct binary\_heap {

int n; // число элементов

int m; // размер массива n <= m

struct binary\_heap\_nodes \*nodes;

};

**Создание кучи.** Для инициализации кучи вызывается функция **heap\_init**. При инициализации задаётся значение m, зануляется n и выделяется динамическая память под массив узлов.

**Вставка элемента.** Новый элемент добавляется в конец массива. В дереве этой позиции соответствует самый правый лист на последнем уровне. Если свободных ячеек в массиве не осталось, функция **heap\_insert** возвращает специальное значение HEAP\_OVERFLOW.

После вставки нового элемента могут быть нарушены свойства невозрастающей бинарной кучи – добавленный ключ может быть больше ключа родительского узла. Поэтому нам необходимо восстановить свойства бинарной кучи. Для этого мы проходим в функции **heapifyup** по дереву от листа 𝐴[𝑛-1] до корня 𝐴[0] и выполняем обмены узлов, если ключ родительского узла меньше ключа текущего элемента. Перебор узлов прерывается, если ключ родителя стал больше ключа текущего узла.

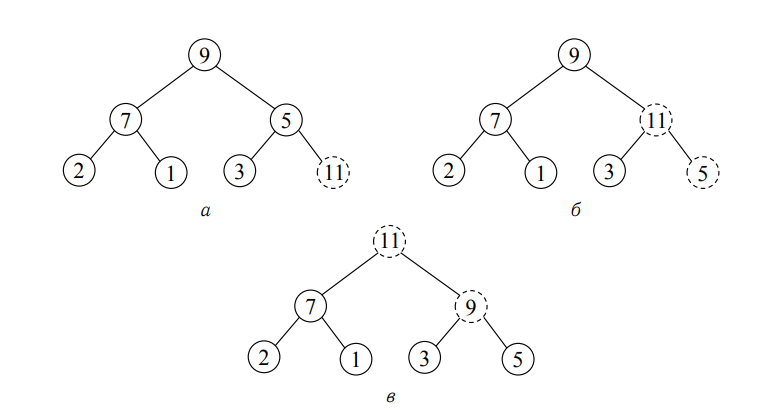


Рис. 6. Добавление ключа 11 в невозрастающую бинарную кучу: а – добавление ключа 11 в конец массива – ячейку 𝐴[6]; б – результат обмена значениями узлов 11 и 5; в – результат обмена значениями узлов 11 и 9.

**Поиск максимального элемента.** По определению элемент с максимальный ключом (приоритетом) хранится в первой ячейке массиве. Если куча не содержит элементов, то функция **heap\_max** возвращает специальное значение HEAP\_EMPTY.

**Удаление максимального элемента.** Для удаления максимального элемента 𝐴[0] мы запоминаем его значение для последующего возврата из функции **heap\_delete\_max**. Далее последний элемент 𝐴[𝑛-1] переносится в ячейку 𝐴[0], число 𝑛 элементов в куче уменьшается на 1. После этого необходимо восстановить свойства невозрастающей кучи, так как в корне дерева теперь может находиться элемент не с максимальным значением.

Восстановление свойств кучи выполняется функцией **heapifydown**, которая в цикле движется от корня к листьям. На каждой итерации цикла среди текущего узла 𝑖 и двух его дочерних элементов 𝑙𝑒𝑓 𝑡 и 𝑟𝑖𝑔ℎ𝑡 выбирается наибольший 𝑙𝑎𝑟𝑔𝑒𝑠𝑡 – тот, который содержит наибольший ключ. Если наибольшим оказался один из дочерних узлов, то выполняется обмен текущего узла 𝑖 и найденного наибольшего. Проход по дереву останавливается, если наибольшее значение ключа оказалось у текущего элемента 𝑖, а не у его дочерних. Это означает, что дальше свойства кучи восстанавливать не требуется.

Время выполнения функции **heapifydown** зависит от высоты ℎ узла 𝑖, для которого она вызвана. Таким образом, время выполнения этой функции есть 𝑂(ℎ). В худшем случае **heapifydown** осуществляет проход по всем уровням завершенного бинарного дерева (от корня до листа), что требует 𝑂(log 𝑛) операций обменов.

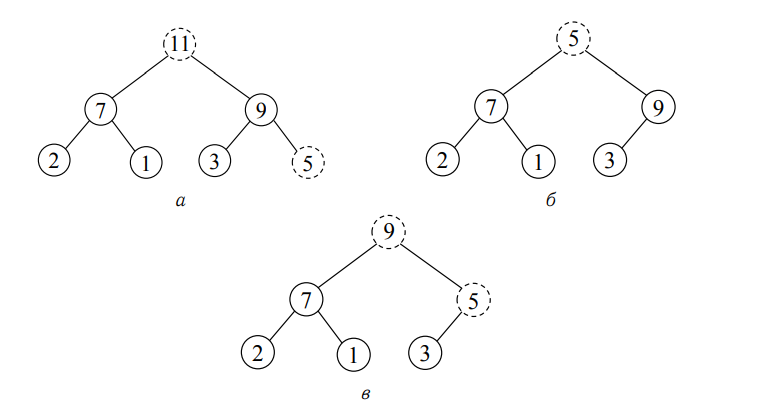


Рис. 7. Удаление ключа 11 с максимальным значением (max-heap): а – исходное состояние кучи; б – последний элемент 5 массива перемещен в корень дерева; в – результат обмена значениями узлов 5 и 9.

**Увеличение ключа**. Операция heap\_increase\_key заменяет у узла, хранящегося в ячейке 𝑖, значение ключа на большее. Если переданное значение нового ключа 𝑘𝑒𝑦 меньше текущего 𝐴[𝑖].𝑘𝑒𝑦, то функция возвращает специальное значение HEAP\_INVALID\_KEY.

После установки нового значения ключа следует восстановить свойства невозрастающей кучи. Новый ключ может быть больше ключа родительского элемента, поэтому мы вызываем функцию **heapifyup** для ячейки с номером 𝑖.

В худшем случае выполняется изменение приоритета листового узла и новый ключ превосходит по значению все имеющиеся ключи. В такой ситуации функция **heapifyup** выполняет порядка 𝑂(log 𝑛) обменов узлов.

**Очистка памяти.** Так как при инициализации кучи выделялась динамическая память, в завершении работы её нужно освободить. Для этого вызывается функция heap\_**free**.

**Скорость операций**

Вставка элементов 𝑂(log 𝑛)

Поиск максимального элемента 𝑂(1)

Увеличение ключа 𝑂(log 𝑛)

Удаление максимального элемента 𝑂(log 𝑛)

**ТУРНИРНОЕ ДЕРЕВО**

Теперь рассмотрим турнирное дерево, которое тоже является бинарной кучей. Она может использоваться для сортировок и для слияния отсортированных массивов.

Особенность турнирного дерева в том, что значения, из которых строится турнирное дерево, находятся на листьях. Во внутренних узлах находится меньшее или большее значение дочерних узлов. В корне содержится минимальное или максимальное значение.

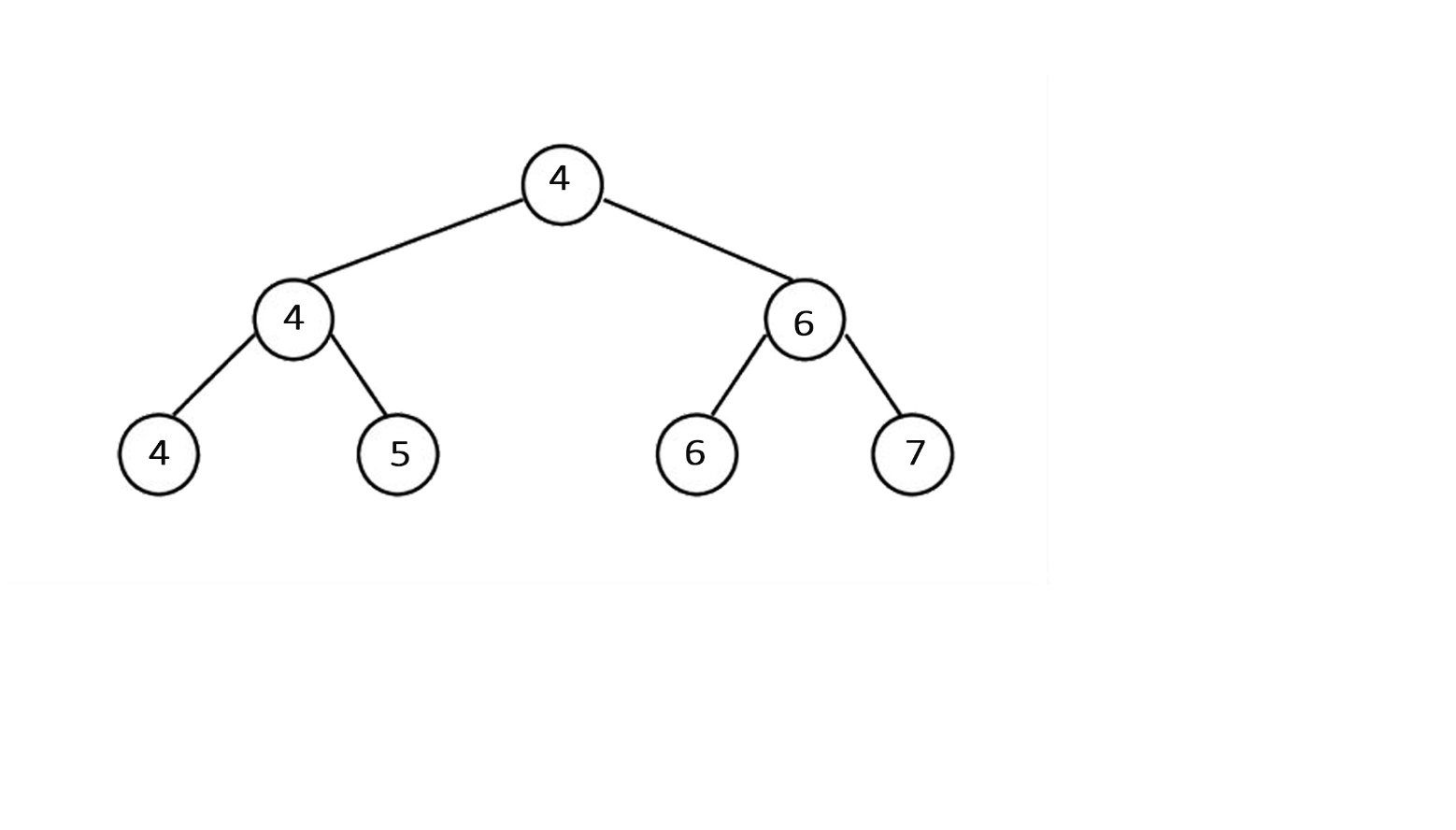


Рис. 8. Турнирное дерево.

**Представление завершенного бинарного дерева в виде одномерного массива**

Завершенные бинарные деревья можно хранить в одномерном массиве. Все узлы дерева размещаются в ячейках массива 𝐴[1..𝑛] в порядке их обхода в ширину. Корень дерева хранится в первой ячейке массива. Для любой вершины, размещенной в ячейке 𝑖 ∈ {1, 2, ..., 𝑛}, левый дочерний узел LEFT(𝑖) расположен в позиции 2𝑖, а правый дочерний узел RIGHT(𝑖) в позиции 2𝑖 + 1. Родительский узел PARENT(𝑖) вершины 𝑖 находится в ячейке ⌊𝑖/2⌋. Если значения 2𝑖 или 2𝑖+ 1 больше 𝑛, то у вершины 𝑖 отсутствует соответствующий дочерний узел.

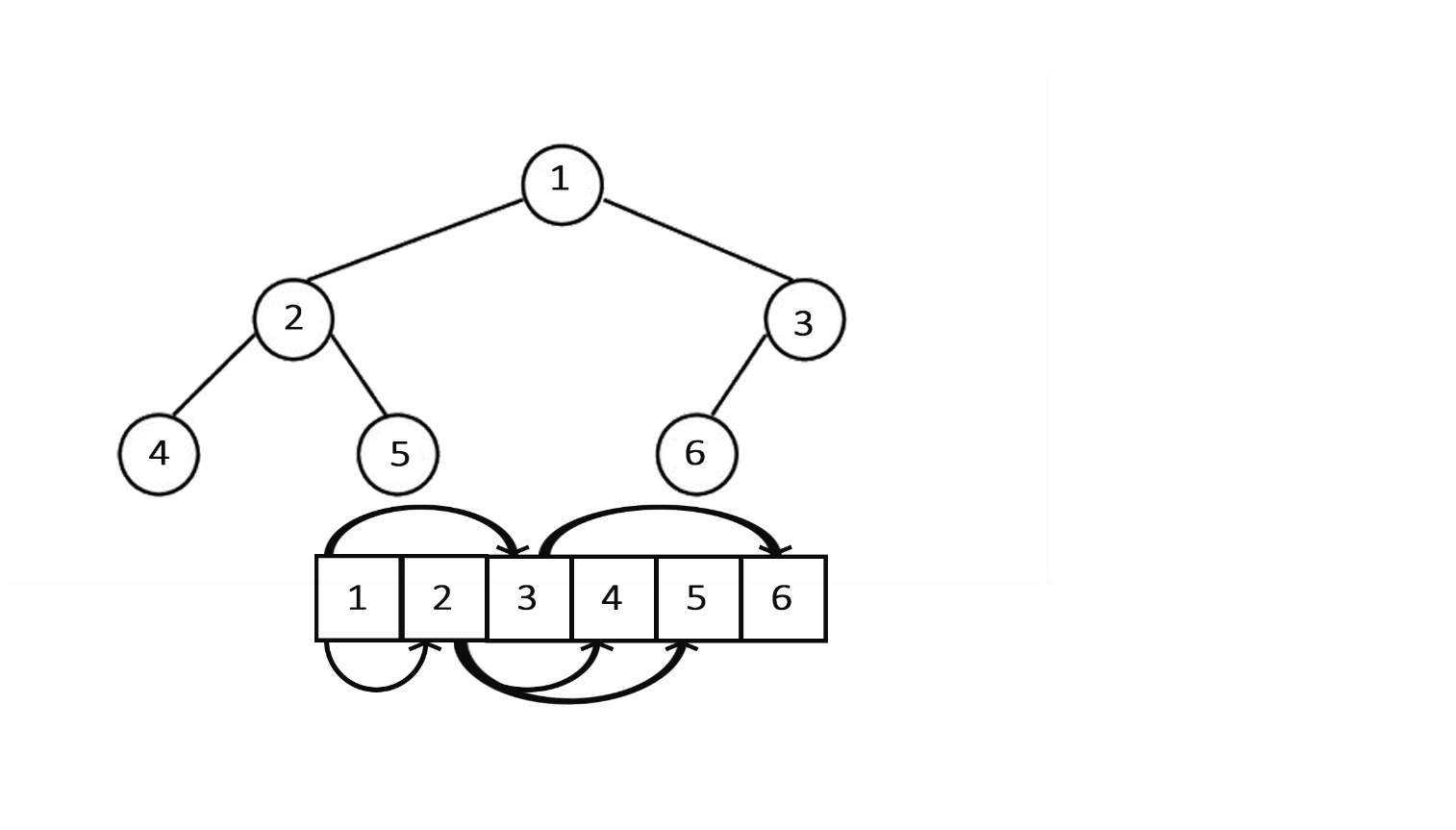


Рис. 9. Представление бинарного дерева в виде массива.

**ПОСТРОЕНИЕ ДЕРЕВА**

Для построения кучи нам нужно узнать, какого размера будет дерево. Для этого создаётся переменные, хранящие размер кучи и количество необходимых листов (изначально равны 0 и 1 соответственно). Далее используется цикл, который каждую итерацию прибавляет к размеру дерева количество листов и умножает их количество в 2 раза. Цикл работает, пока количество листов меньше количества элементов массива, из которого строится дерево. По завершении работы цикла также нужно прибавить к размеру количество листов.

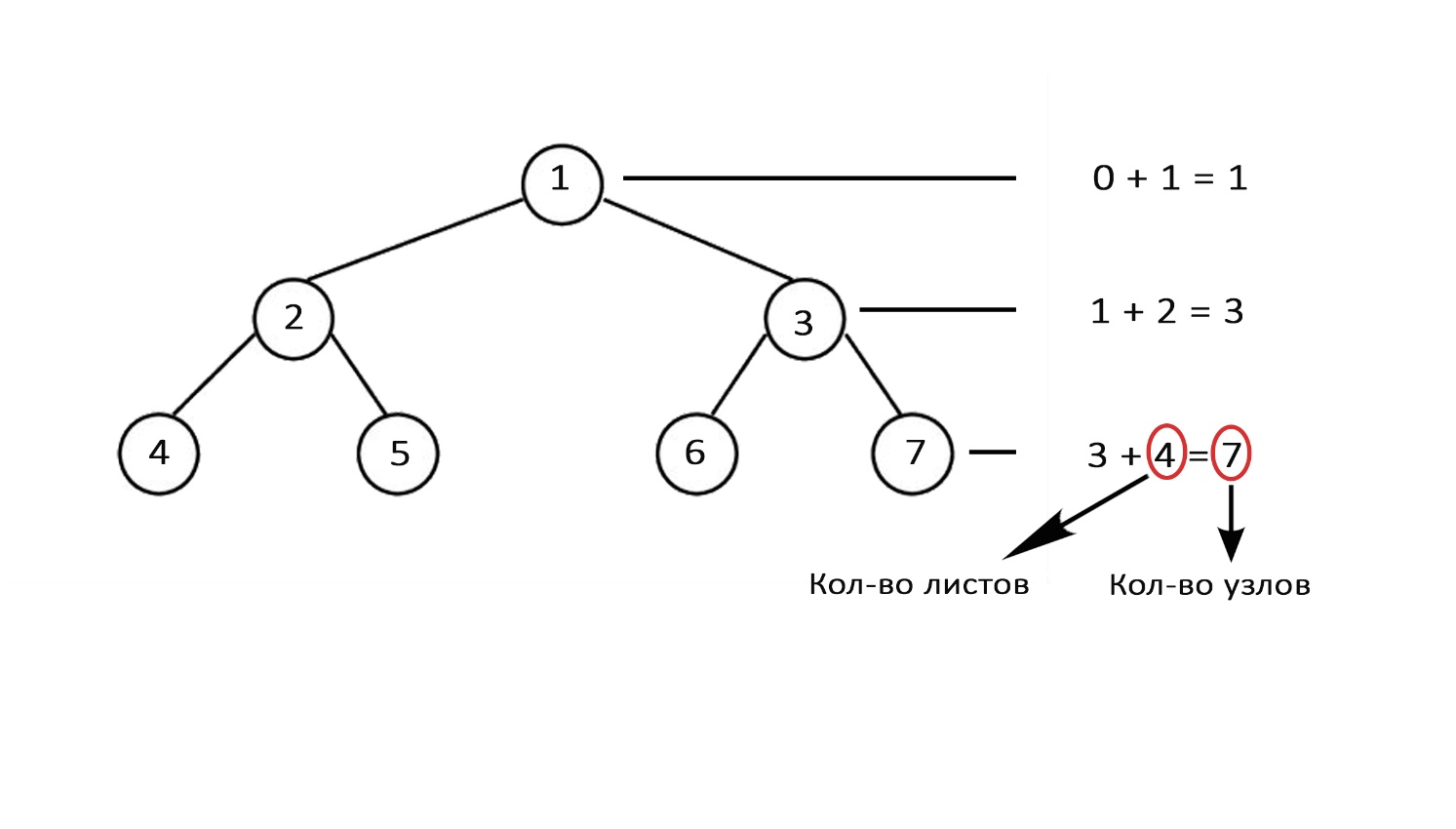


Рис. 10. Пример вычисления количества количество необходимых узлов для турнирного дерева.

Далее выделяется память для дерева. В моей реализации используется дополнительный элемент в массивах и деревьях, в котором записан их размер. Он записывается в начало массивов и деревьев. Это было сделано для упрощения реализации.

После выделения памяти все узлы приравниваются зарезервированному значению INT\_MAX. Элементы равные этому значению означают, что узел пуст. Чтобы опустить какой-либо узел, нужно приравнять его INT\_MAX. Иногда пустые узлы будут обозначаться символом «\*».

Чтобы заполнить листы слева на право, нам понадобится переменная с количеством листов. Она нужна, чтобы найти первый внешний узел.

Затем функция get\_winner\_tree заполняет все остальные узлы. Она начинает работу с самого правого узла предпоследнего уровня, продвигаясь к крайнему левому. После начинает работать аналогично с самым правым узлом, который находится уровнем выше.

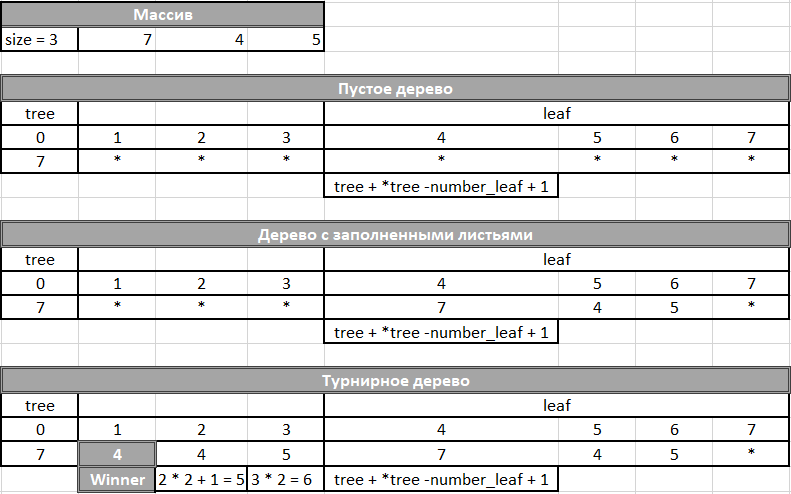


Рис. 11. Процесс создания турнирного дерева.

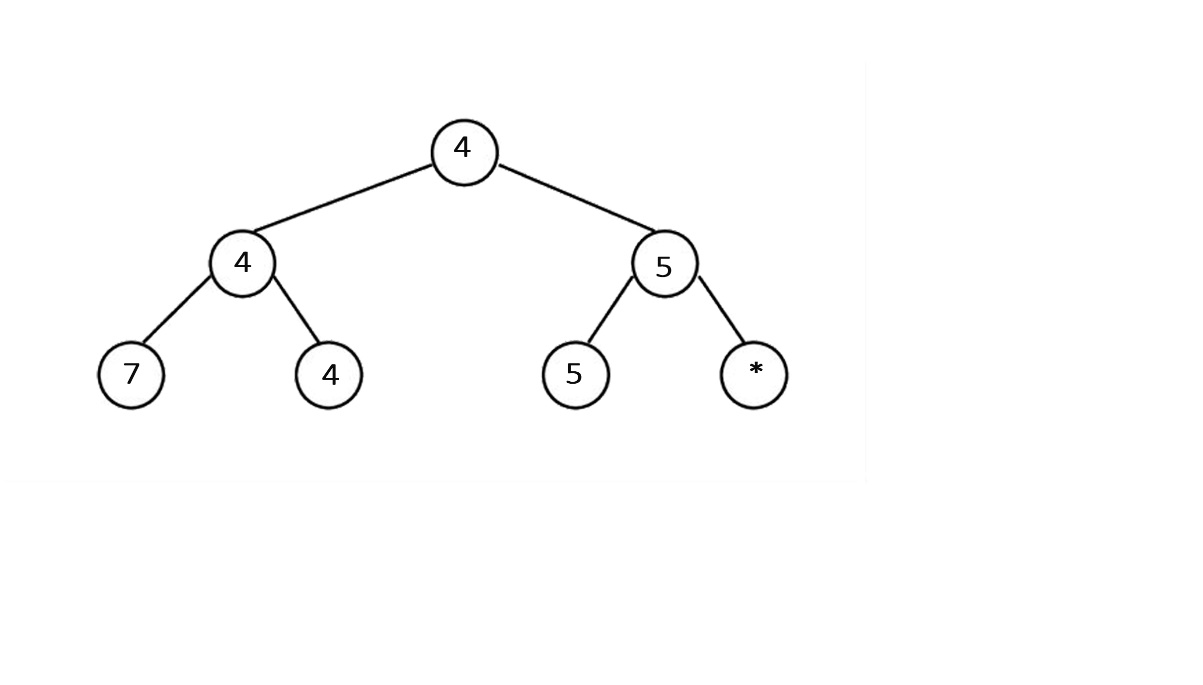


Рис. 12. Графическое представление турнирного дерева.

**АЛГОРИТМ ТУРНИРНОЙ СОРТИРОВКИ**

Турнирное дерево можно использовать для сортировки. После того как построится турнирное дерево, победитель записывается в массив. Затем победитель удаляется из дерева и начинается поиск нового победителя. Этот алгоритм повторяется до тех пор, пока дерево не опустеет. В результате получается отсортированный массив

Для построения турнирного дерева используется алгоритм, описанный выше. А поиск победителей происходит в цикле, который запускает функции HEAPIFYDOWN и HEAPIFYUP. Цикл заканчивает работу, когда дерево опустеет.

Удаление победителя выполняет функция HEAPIFYDOWN. Она устанавливает указатель на корень и ищет расположение победителя среди дочерних узлов. Функция находит победителя и перемещает указатель на его место и продолжает опускаться до листьев. Когда нужный лист будет найден, он очистится, а функция возвратит положение листа.

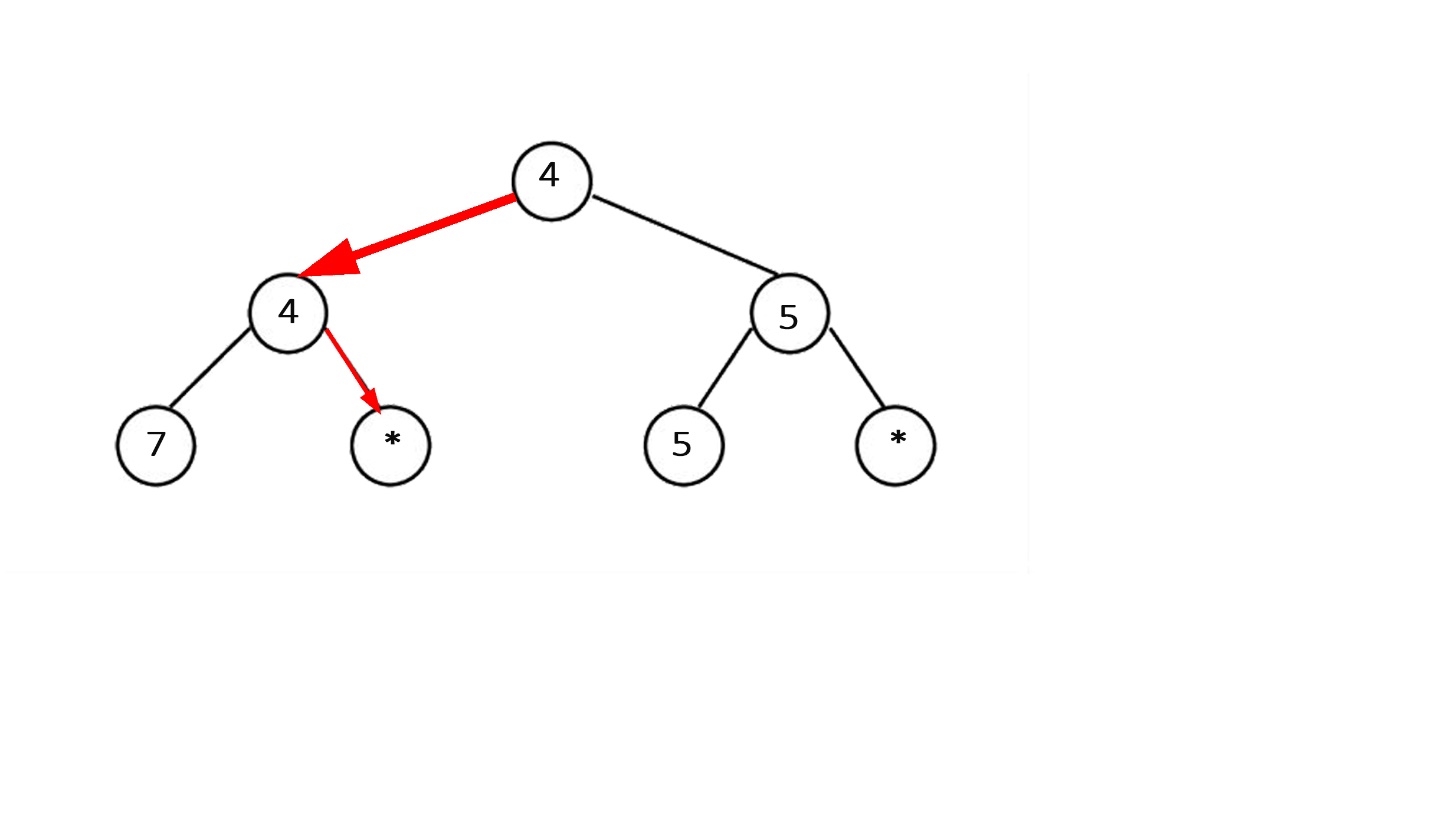
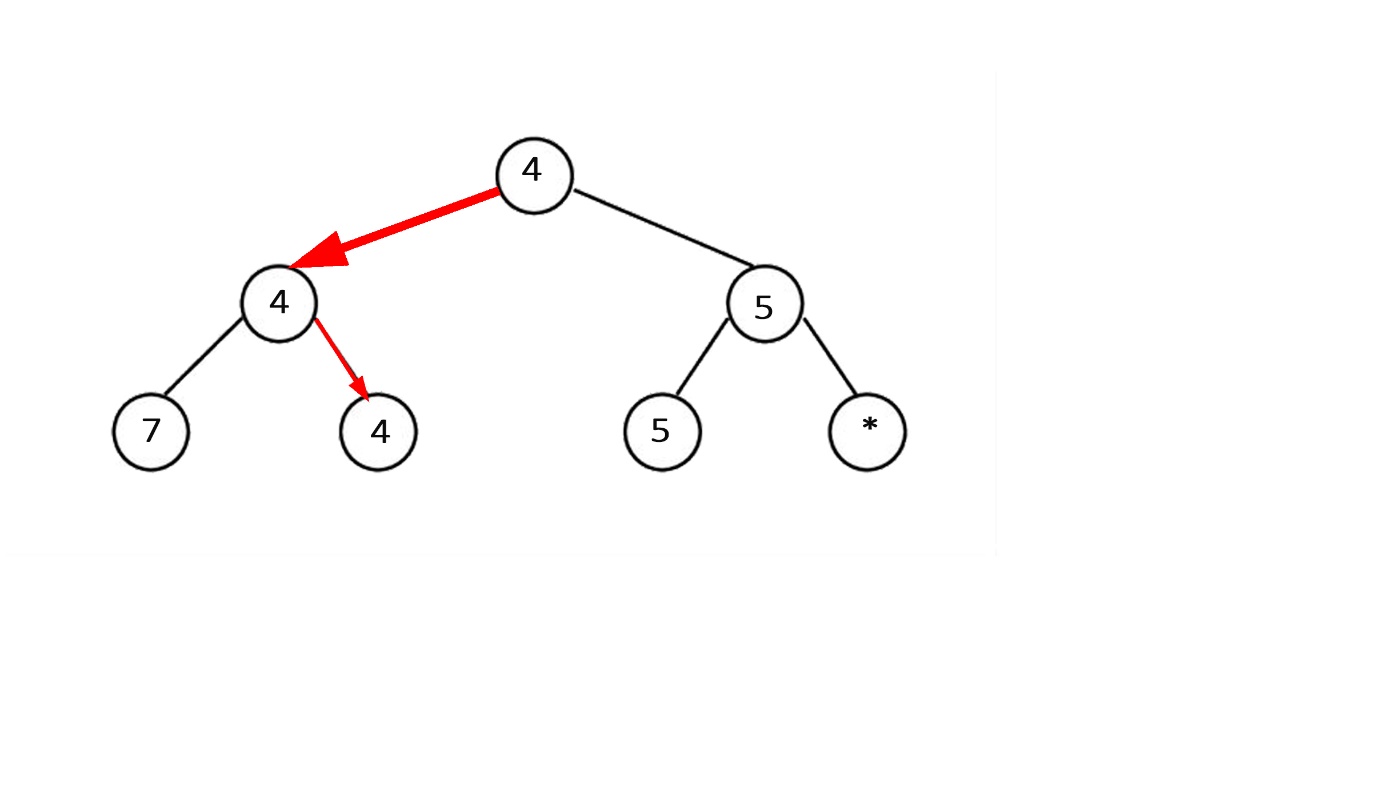


Рис. 13. HEAPIFYDOWN.

Новый победитель находится при помощи функции HEAPIFYUP, в которую передаётся положение откуда нужно начинать повторный поиск победителя. Этим положением является указатель, который возвращает функция HEAPIFYDOWN. При поиске нового победителя нужно обходить только те узлы, которые ведут из изначального узла к корню. Остальная часть дерева уже отсортированная.

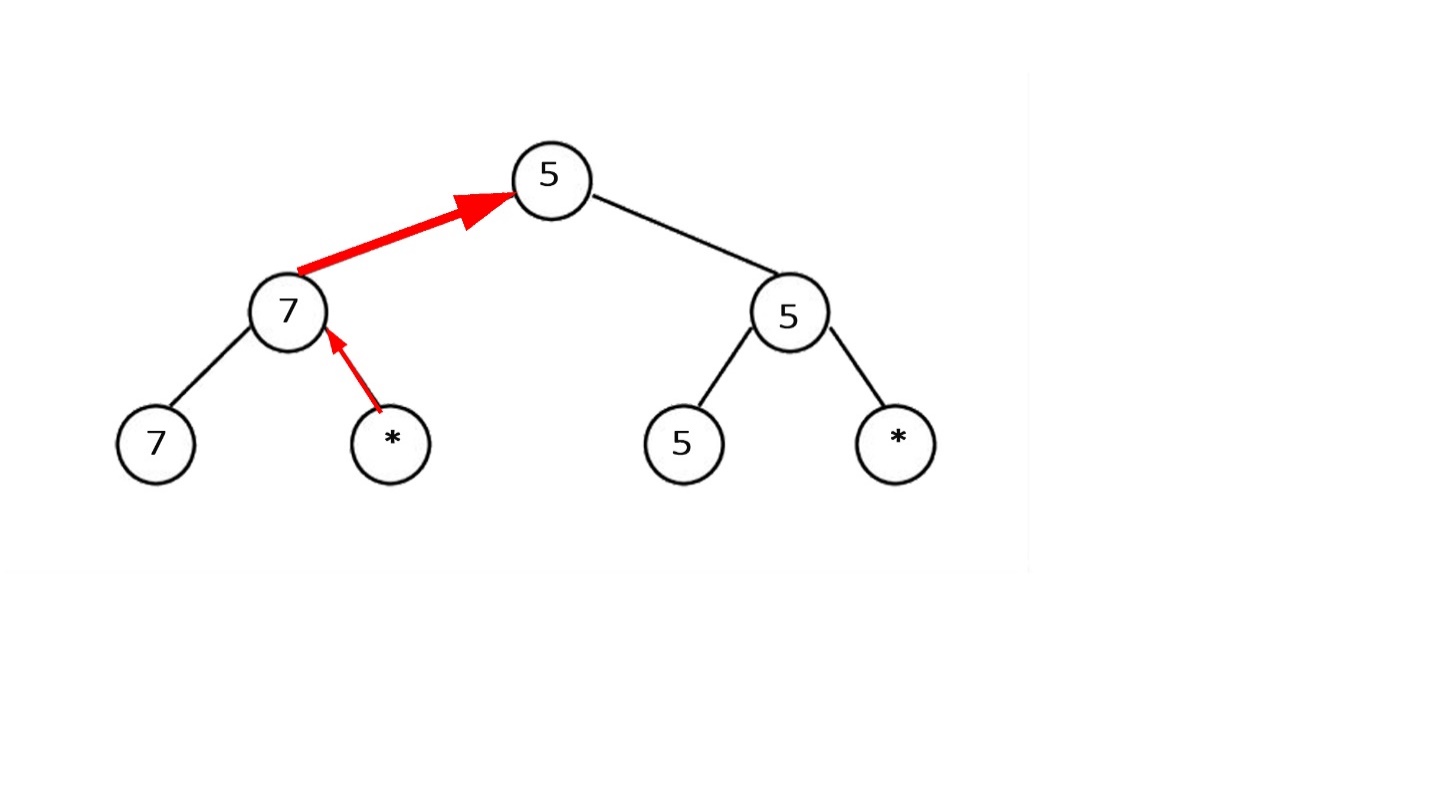


Рис. 14. HEAPIFYUP.

Использование HEAPIFYUP и HEAPIFYDOWN работают на много эффективнее, чем если бы программа обходила все узлы повторно.

Когда дочерние узлы равны, в родительский узел записывается значение левого узла. При поиске листа победителя функция HEAPIFYDOWN так же отдаёт приоритет левому узла. Учитывая, что мы переписываем массив в дерево слева на право, можно сделать вывод, что элементы записываются в том же порядке. Следовательно, сортировка устойчивая.

**АНАЛИЗ СОРТИРОВКИ**

Минусы: сортировка работает не на месте, во время работы сортировки в массив нельзя добавлять новые элементы.

Плюсы: сортировка работает за оптимальное время, сортировка устойчивая.

Сложность: *O*(n ), где n – число элементов.

Каждый элемент должен подняться к корню. Чтобы узел поднялся к корню, число сравнений этого элемента с другими должно быть равно высоте дерева ln(n). Так же после нахождения победителя нам нужно спуститься к его листу. Это занимает такое же количество времени ln(n).

n \* 2 = *O*(n )

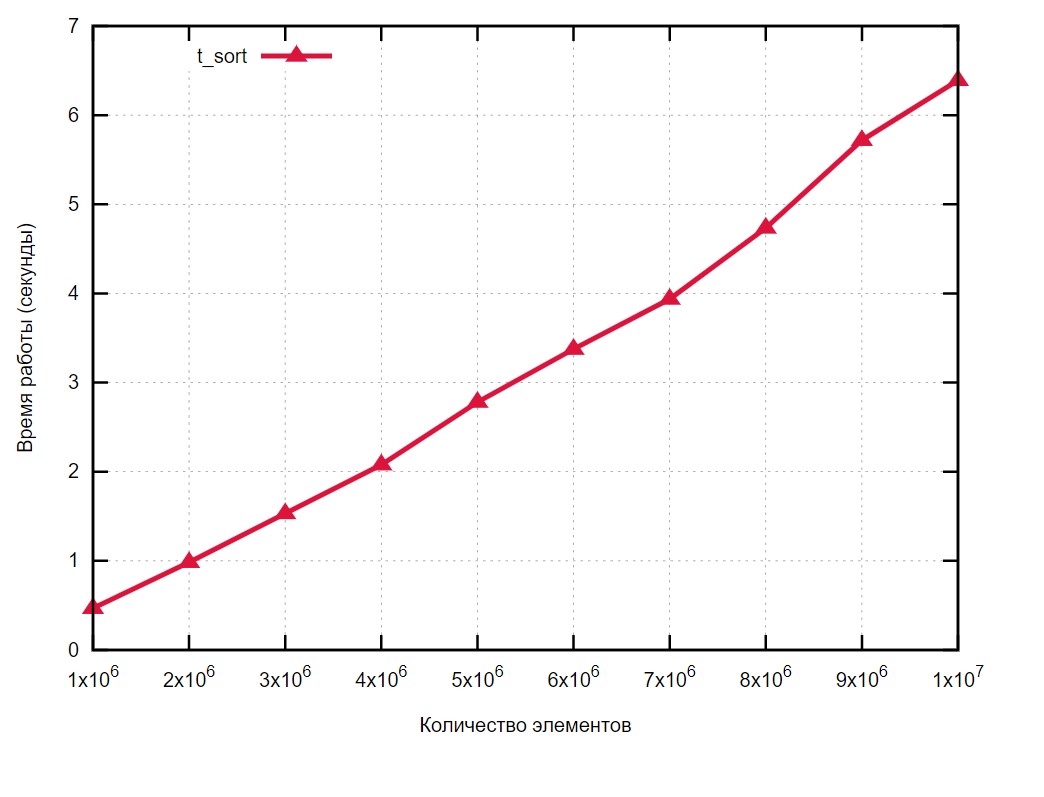


Рис. 15. Времени работы турнирной сортировки.

**ОБЪЕДИНЕНИЕ ОТСОРТИРОВАННЫХ МАССИВОВ**

Турнирное дерево также можно использовать для объединения уже отсортированных массивов. Для слияния понадобится турнирное дерево, в котором каждому массиву достанется лист. Каждый массив будет по необходимости записывать во внешние узлы свои элементы. Когда элементы какого-либо массива закончатся, его лист будет равен INT\_MAX.

В функцию, которая объединяет массивы, передаётся массив массивов. В первом элементе хранятся размеры массивов. В функции создаются дополнительные массивы. Первый для сохранения индексов элементов массивов, которые нужно записывать в дерево. Во второй будет записан результат объединения массивов.

Далее начинает работу цикл, который заполняет пустые листья турнирного дерева и находит победителя. Победитель ищется по тому же алгоритму, что и в первой сортировке. Цикл завершает работу, когда все элементы будут переписаны в новый массив. В результате функция возвращает указатель на этот массив.

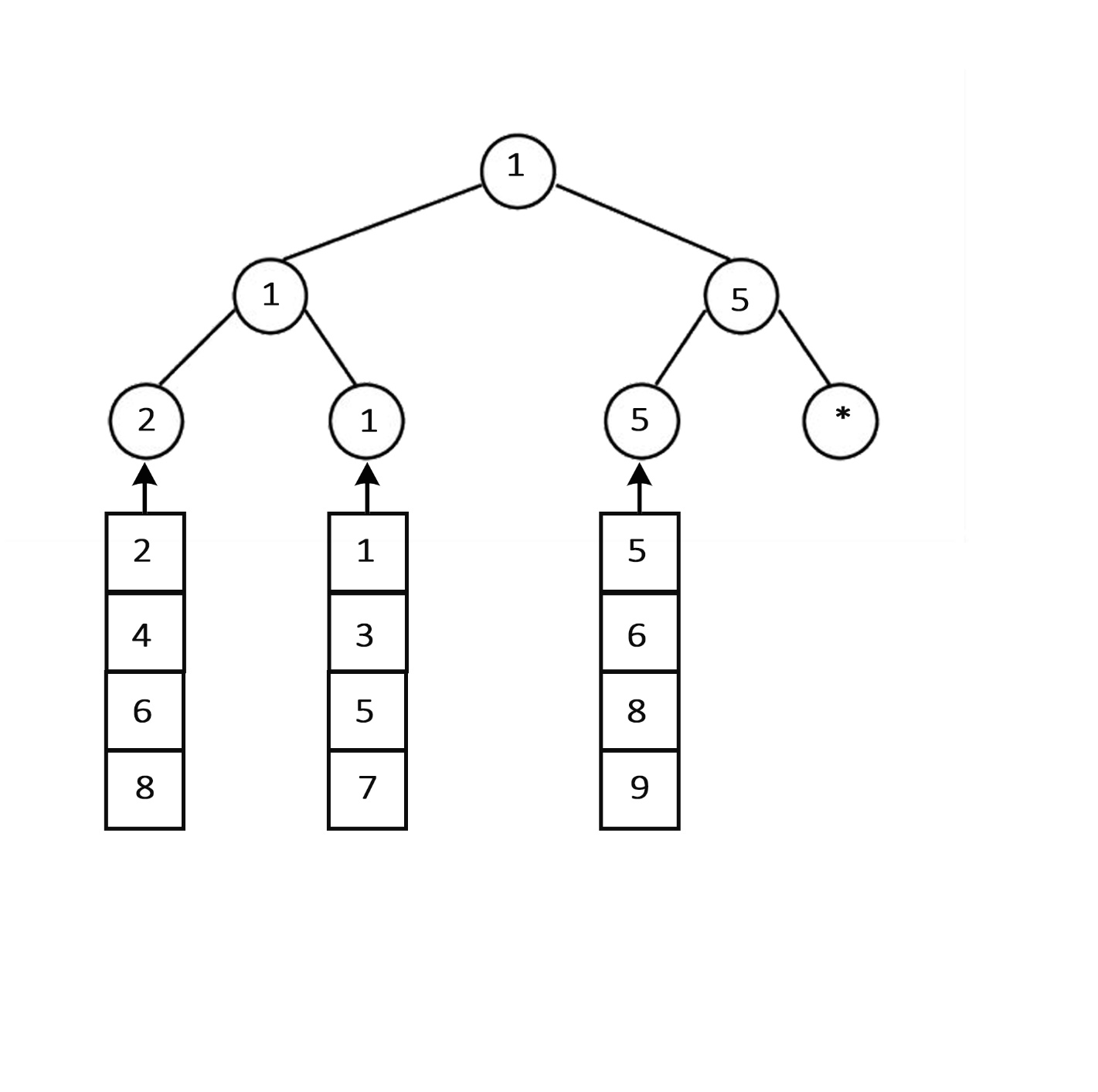


Рис. 16. Графическое представление работы слияния массивов.

**АНАЛИЗ ПРОГРАММЫ**

Минусы: не является сортировкой на месте, после начала слияния в процесс нельзя добавлять новые массивы, такой метод слияния более оптимален при маленькой разнице в размерах массивов.

Плюсы: слияние работает за оптимальное время.

Сложность: *O*(n ), где n – число элементов при условии, что все массивы имеют одинаковый размер.

Каждый элемент должен подняться к корню. Чтобы узел поднялся к корню, число сравнений этого элемента с другими должно быть равно высоте дерева . Так же после нахождения победителя нам нужно спуститься к его листу. Это занимает такое же количество времени .

n \* 2 = *O*(n )

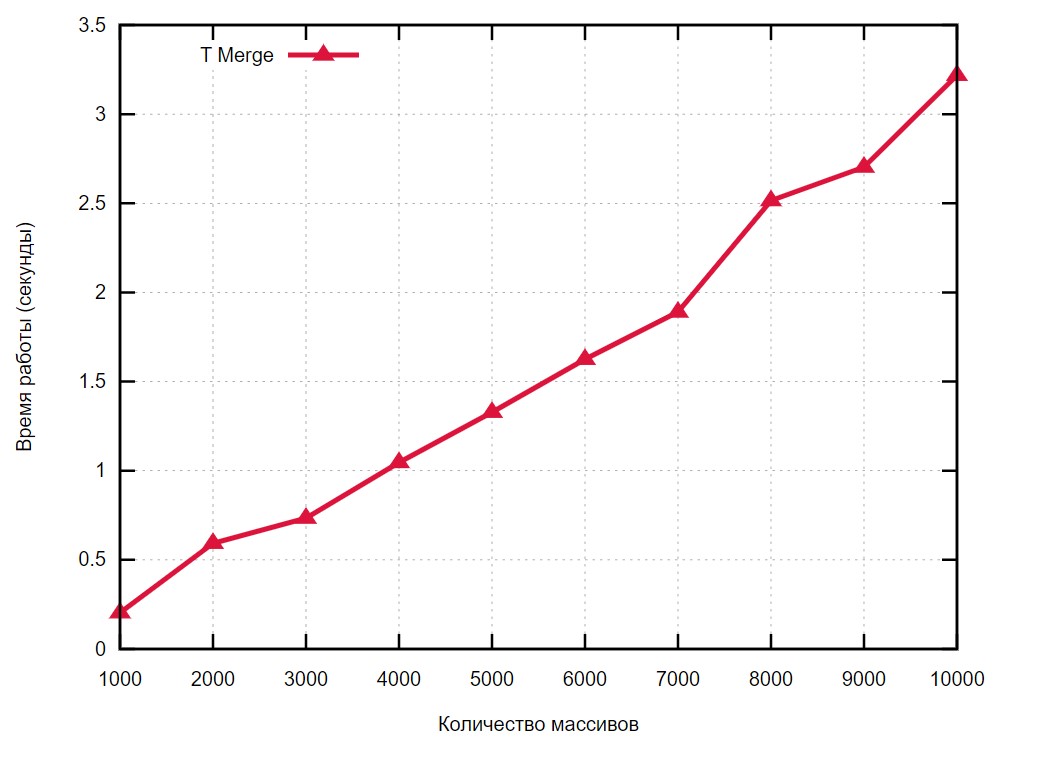


Рис. 17. Времени работы алгоритма слияния массивов (с размером 1000 элементов).

**РЕЗУЛЬТАТ РАБОТЫ**

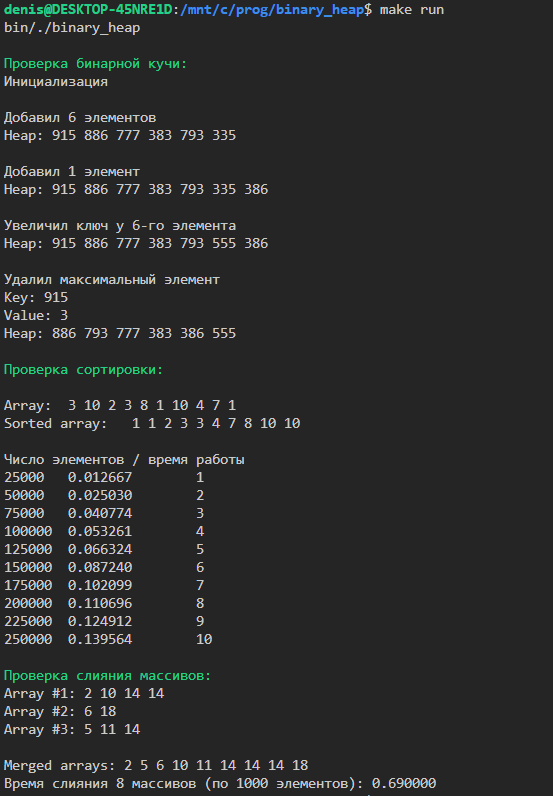


Рис. 18

**ЗАКЛЮЧЕНИЕ**

В ходе работы над проектом я пополнил свои знания об бинарных кучах. Мне удалось написать турнирную сортировку, основанную на принципах бинарной кучи. А также реализовал алгоритм слияния массивов при помощи турнирного дерева.

Как можно заметить, бинарная куча – очень эффективная структура данных, у которой есть много применений. Помимо турнирной сортировки на алгоритмах бинарной кучи основана пирамидальная сортировка.

**СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ**

1.. М. Г. Курносов, Д. М. Берлизов. «Алгоритмы и структуры обработки информации» – Новосибирск: Параллель, 2019. – 227 с

2.Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ. / М.: Издательский дом «Вильямс», 2011.— 1296 с.

**ПРИЛОЖЕНИЯ**

|  |  |
| --- | --- |
| ***main.c***  #include "../lib/lib.h"  **int** main ()  {  printf("\n\033[92mПроверка бинарной кучи:\033[39m\n");  binary\_heap\_test (10);  printf("\n\033[92mПроверка сортировки:\033[39m\n");  tournament\_sort\_test ();  tournament\_sort\_test\_time ();  printf("\n\033[92mПроверка слияния массивов:\033[39m\n");  merge\_array\_test ();  merge\_array\_test\_time ();  **return** 0;  }  ***lib.h***  #include <stdio.h>  #include <stdlib.h>  #include <limits.h>  #include <time.h>  **enum** ERRORS  {  HEAP\_OVERFLOW = 1,  HEAP\_INVALID\_KEY,  };  **int** binary\_heap\_test ();  **int** tournament\_sort\_test ();  **int** tournament\_sort\_test\_time ();  **int** merge\_array\_test ();  **int** merge\_array\_test\_time ();  ***binary\_heap\_test.c***  #include <stdio.h>  #include <stdlib.h>  #include "lib\_heap.h"  **int** binary\_heap\_test (size\_t size)  {  **struct** binary\_heap heap;  **struct** binary\_heap\_nodes \*max\_node;  printf("Инициализация\n");  heap\_init (&heap, size);  printf("\nДобавил 6 элементов\n")***;***  for (int i = 0; i < 6; i++***)***  heap\_insert (&heap, rand() % 1000, i)***;***  if (checking\_max\_heap\_property(heap)) printf("Нарушено свойство кучи\n")***;***  heap\_print (heap)***;***  printf("\nДобавил 1 элемент\n");  heap\_insert (&heap, rand() % 1000, 10);  **if** (checking\_max\_heap\_property(heap)) printf("Нарушено свойство кучи\n");  heap\_print (heap);  printf("\nУвеличил ключ у 6-го элемента\n");  heap\_increase\_key (heap.nodes, 5, 555);  **if** (checking\_max\_heap\_property(heap))  printf("Нарушено свойство кучи\n");  heap\_print (heap);  printf("\nУдалил максимальный элемент\n");  max\_node = heap\_delete\_max (&heap);  printf("Key: %d\nValue: %d\n", max\_node->key, max\_node->value);  heap\_print (heap);  heap\_free (&heap);  free(max\_node);  **return** 0;  } | ***lib\_heap.h***  #**pragma once**  #include <stdio.h>  #include <stdlib.h>  #include <math.h>  #include "../src/lib/lib.h"  #define HEAP\_EMPTY NULL  **struct** binary\_heap\_nodes  {  **int** key;  **int** value;  };  **struct** binary\_heap  {  **int** n; // число элементов  **int** m; // размер массива n <= m  **struct** binary\_heap\_nodes \*nodes;  };  #define SWAP\_NODES(a, b) \  { \  struct binary\_heap\_nodes temp = a; \  a = b; \  b = temp; \  }  **int** checking\_max\_heap\_property (**struct** binary\_heap heap);  **int** heap\_increase\_key (**struct** binary\_heap\_nodes \*nodes, **int** index, **int** key);  **void** heapifydown (**struct** binary\_heap\_nodes \*nodes, **int** index, **int** n);  **struct** binary\_heap\_nodes\* heap\_delete\_max (**struct** binary\_heap \*heap);  **void** heap\_print\_max (**struct** binary\_heap heap);  **struct** binary\_heap\_nodes\* heap\_max (**struct** binary\_heap heap);  **void** heap\_print (**struct** binary\_heap heap);  **void** heapifyup (**struct** binary\_heap\_nodes \*nodes, **int** index);  **int** heap\_insert (**struct** binary\_heap \*heap, **int** key, **int** value);  **int** heap\_free (**struct** binary\_heap \*heap);  **int** heap\_init (**struct** binary\_heap \*heap, size\_t size); |

|  |  |
| --- | --- |
| ***interface.c***  #include "lib\_heap.h"  **int** checking\_max\_heap\_property (**struct** binary\_heap heap)  {  **for** (**int** i = 0; i < heap.n; i++)  {  **if** (( (heap.nodes[i].key < heap.nodes[i \* 2 + 1].key) && (i \* 2 + 1) < heap.n)  || ((heap.nodes[i].key < heap.nodes[i \* 2 + 2].key) && (i \* 2 + 2) < heap.n))  **return** 1;  }  **return** 0;  }  **int** heap\_increase\_key (**struct** binary\_heap\_nodes \*nodes, **int** index, **int** key)  {  **if** (nodes[index].key > key)  **return** HEAP\_INVALID\_KEY;  nodes[index].key = key;  heapifyup(nodes, index);  **return** 0;  }  **void** heapifydown (**struct** binary\_heap\_nodes \*nodes, **int** index, **int** n)  {  **int** largest;  n--;  **while** (index < n)  {  largest = index;  **if** ((index \* 2 + 1 <= n) && (nodes[index \* 2 + 1].key > nodes[largest].key)) // левый потомок  largest = index \* 2 + 1;    **if** ((index \* 2 + 2 <= n) && (nodes[index \* 2 + 2].key > nodes[largest].key)) // правый  largest = index \* 2 + 2;  **if** (&nodes[largest] == &nodes[index])  **break**;  SWAP\_NODES(nodes[index], nodes[largest]);  index = largest;  }  }  index = (index - 1) / 2;  }  }  **int** heap\_insert (**struct** binary\_heap \*heap, **int** key, **int** value)  {  **if** (heap->n == heap->m)  **return** HEAP\_OVERFLOW;    heap->nodes[heap->n].key = key;  heap->nodes[heap->n].value = value;    heapifyup (heap->nodes, heap->n);  heap->n += 1;  **return** 0;  }  **int** heap\_free (**struct** binary\_heap \*heap)  {  **if** (heap == NULL || heap->nodes == NULL)  **return** 1;  free(heap->nodes);  **return** 0;  }  **int** heap\_init (**struct** binary\_heap \*heap, size\_t size)  {  heap->m = size - 1;  heap->n = 0;  heap->nodes = malloc(**sizeof**(**struct** binary\_heap\_nodes) \* size);  **if** (heap->nodes == NULL)  **return** -1;  **return** 0;  } | **struct** binary\_heap\_nodes\* heap\_delete\_max (**struct** binary\_heap \*heap)  {  **struct** binary\_heap\_nodes \*max = malloc(**sizeof**(**struct** binary\_heap\_nodes));  **if** (heap->m == 0)  {  free(max);  **return** HEAP\_EMPTY;  }  max->key = heap->nodes[0].key;  max->value = heap->nodes[0].value;  heap->n -= 1;  heap->nodes[0] = heap->nodes[heap->n];  heapifydown(heap->nodes, 0, heap->n);  **return** max;  }  **void** heap\_print\_max (**struct** binary\_heap heap)  {  **struct** binary\_heap\_nodes \*nodes;  nodes = heap\_max (heap);  **if** (nodes != NULL)  printf("\nKey = %d\nValue = %d\n", nodes->key, nodes->value);  **else**  printf("\nНет узлов\n");  }  **struct** binary\_heap\_nodes\* heap\_max (**struct** binary\_heap heap)  {  **if** (heap.n == 0)  **return** HEAP\_EMPTY;    **return** &heap.nodes[0];  }  **void** heap\_print (**struct** binary\_heap heap)  {  printf("Heap: ");  **for** (**int** i = 0; i < heap.n; i++)  printf("%d ", heap.nodes[i].key);  printf("\n");  }  **void** heapifyup (**struct** binary\_heap\_nodes \*nodes, **int** index)  {  **while** (nodes[index].key > nodes[(index - 1) / 2].key && index > 0)  {  SWAP\_NODES(nodes[index], nodes[(index - 1) / 2]); |

|  |  |
| --- | --- |
| ***tournament\_sort\_test.c***  #**pragma once**  #include <stdio.h>  #include <stdlib.h>  #include <math.h>  #include "../src/lib/lib.h"  #define HEAP\_EMPTY NULL  **struct** binary\_heap\_nodes  {  **int** key;  **int** value;  };  **struct** binary\_heap  {  **int** n; // число элементов  **int** m; // размер массива n <= m  **struct** binary\_heap\_nodes \*nodes;  };  #define SWAP\_NODES(a, b) \  { \  struct binary\_heap\_nodes temp = a; \  a = b; \  b = temp; \  }  **int** checking\_max\_heap\_property (**struct** binary\_heap heap);  **int** heap\_increase\_key (**struct** binary\_heap\_nodes \*nodes, **int** index, **int** key);  **void** heapifydown (**struct** binary\_heap\_nodes \*nodes, **int** index, **int** n);  **struct** binary\_heap\_nodes\* heap\_delete\_max (**struct** binary\_heap \*heap);  **void** heap\_print\_max (**struct** binary\_heap heap);  **struct** binary\_heap\_nodes\* heap\_max (**struct** binary\_heap heap);  **void** heap\_print (**struct** binary\_heap heap);  **void** heapifyup (**struct** binary\_heap\_nodes \*nodes, **int** index);  **int** heap\_insert (**struct** binary\_heap \*heap, **int** key, **int** value);  **int** heap\_free (**struct** binary\_heap \*heap);  **int** heap\_init (**struct** binary\_heap \*heap, size\_t size); | ***lib\_tournament\_merge.h***  #**pragma once**  #include "../src/lib/lib.h"  #define MAX\_VALUE 1000000  #define SIZE\_A 1000  #define ARRAY\_NUMBER 8  #define SIZE\_TREE 7  #define SWAP\_INT(a, b) int temp = a; \  a = b; \  b = temp;  #define PRINT\_TREE(tree) printf("\ntree: "); \  for (int i = 1; i <= \*tree; i++) { \  if (\*(tree + i) != INT\_MAX) \  printf("%d ", \*(tree + i)); \  else \  printf("\* "); \  } \  getchar();  **int**\* find\_winner(**int** \*tree, **int** winner, **int** size);  **int** HEAPIFYUP(**int**\* tree, **int**\* pos);  **int**\* HEAPIFYDOWN(**int** \*tree);  **void** shift(**int** \*tree, **int**\*\* Arrays, **int**\* a\_pos, **int** winner);  **int** get\_winner\_tt(**int**\* tree, **int** size);  **int** get\_winner(**int**\* tree, **int** size);  **int**\*\* create\_arrays();  **int**\*\* create\_arrays\_();  // int\* add\_leaf(int\* tree, int\* t\_pos, int\* a\_pos);  // int\* tournament\_sort\_win\_lose(int\* arr);  **int** tournament\_sort(**int**\* arr);  **int**\* merge\_array(**int**\*\* Arrays); |

|  |  |
| --- | --- |
| ***tournament\_sort.c***  #include "lib\_tournament\_sort.h"  **int**\* add\_leaf(**int**\* tree, **int**\* t\_pos, **int**\* a\_pos)  {  **if** (\*(tree + (t\_pos - tree) / 2) <= \*a\_pos) {  SWAP\_INT(\*t\_pos, \*a\_pos)  } **else** {  \*t\_pos = \*(tree + (t\_pos - tree) / 2);  \*(tree + (t\_pos - tree) / 2) = \*a\_pos;  }    **return** tree + (t\_pos - tree) / 2;  }  **int**\* find\_winner(**int** \*tree, **int** winner, **int** size)  {  **int** \*pos = tree + \*tree - size + 1;  **while** (pos <= tree + \*tree) {  **if** (\*pos == winner)  **return** pos;  pos++;  }  **return** NULL;  }  **int**\* HEAPIFYDOWN(**int** \*tree)  {  **int** \*cur = tree + 1;  **while** (tree + (cur - tree) \* 2 < tree + \*tree) {  **if** (\*(tree + (cur - tree) \* 2) == \*cur)  cur = tree + (cur - tree) \* 2;  **else**  cur = tree + (cur - tree) \* 2 + 1;  }  \*cur = INT\_MAX;  **return** cur;  }  **void** shift(**int** \*tree, **int**\*\* Arrays, **int**\* a\_pos, **int** winner)  {  **for** (**long** **int** i = 1; i <= (**long** **int**)\*Arrays ; i++) {  **if** (\*(\*(Arrays + i) + \*(a\_pos + i - 1)) == winner) {  (\*(a\_pos + i - 1))++;  **break**;  }  }  }  **int** HEAPIFYUP(**int**\* tree, **int**\* pos)  {  pos = tree + (pos - tree) / 2;  **while** (pos > tree) {  **if** (\*(tree + (pos - tree) \* 2) <= \*(tree + (pos - tree) \* 2 + 1))  \*pos = \*(tree + (pos - tree) \* 2);  **else**  \*pos = \*(tree + (pos - tree) \* 2 + 1);  pos = tree + (pos - tree) / 2;  }  **return** \*(tree + 1);  }  **int** get\_winner(**int**\* tree, **int** leaves)  {  **int** \*cur = tree + \*tree - leaves;  **while** (cur > tree) {  **if** (\*(tree + (cur - tree) \* 2) <= \*(tree + (cur - tree) \* 2 + 1))  \*cur = \*(tree + (cur - tree) \* 2);  **else**  \*cur = \*(tree + (cur - tree) \* 2 + 1);  cur--;  }  **return** \*(tree + 1);  }  **int** get\_winner\_(**int**\* tree, **int** size) | {  **int** \*cur = tree + \*tree - size, winner;  **while** (cur > tree) {  **if** (\*(tree + (cur - tree) \* 2) <= \*(tree + (cur - tree) \* 2 + 1)) {  **if** (\*cur > \*(tree + (cur - tree) \* 2)) {  SWAP\_INT(\*cur, \*(tree + (cur - tree) \* 2));  }  } **else** {  **if** (\*cur > \*(tree + (cur - tree) \* 2 + 1)) {  SWAP\_INT(\*cur, \*(tree + (cur - tree) \* 2 + 1));  }  }  cur--;  }  winner = \*(tree + 1);  \*(tree + 1) = INT\_MAX;  **return** winner;  }  **int** tournament\_sort(**int**\* arr)  {  **int** \*tree, number\_leaf = 1, size = 0, \*pos = 0, index = \*arr, winner;  **if** (arr == NULL) **return** 0;  **while** (number\_leaf <= \*arr) {  size += number\_leaf;  number\_leaf \*= 2;  }  size += number\_leaf;  **if** (!(tree = (**int**\*)malloc(size \* **sizeof**(**int**)))) **return** 0;  **for** (**long** **int** i = 1; i <= size; i++)  \*(tree + i) = INT\_MAX;  \*tree = size;  pos = tree + \*tree - number\_leaf;  **for** (**int** i = 1; i <= \*arr; i++)  \*(pos + i) = \*(arr + i);  winner = get\_winner(tree, number\_leaf);  index = 1;  **while** (winner != INT\_MAX && index <= \*arr) {  \*(arr + index) = winner;  index++;  winner = HEAPIFYUP(tree, HEAPIFYDOWN(tree));  }  free(tree);    **return** 1;  } |

|  |  |
| --- | --- |
| ***lib\_tournament\_sort.h***  #**pragma once**  #include "../src/lib/lib.h"  #define MAX\_VALUE 10  #define SIZE\_A 10  #define ARRAY\_NUMBER 7  #define SIZE\_TREE 7  #define SWAP\_INT(a, b) int temp = a; \  a = b; \  b = temp;  #define PRINT\_TREE(tree) printf("\ntree: "); \  for (int i = 1; i <= \*tree; i++) { \  if (\*(tree + i) != INT\_MAX) \  printf("%d ", \*(tree + i)); \  else \  printf("\* "); \  } \  getchar();  **int**\* find\_winner(**int** \*tree, **int** winner, **int** size);  **int** HEAPIFYUP(**int**\* tree, **int**\* pos);  **int**\* HEAPIFYDOWN(**int** \*tree);  **void** shift(**int** \*tree, **int**\*\* Arrays, **int**\* a\_pos, **int** winner);  **int** get\_winner\_tt(**int**\* tree, **int** size);  **int** get\_winner(**int**\* tree, **int** size);  **int**\*\* create\_arrays();  // int\* add\_leaf(int\* tree, int\* t\_pos, int\* a\_pos);  // int\* tournament\_sort\_win\_lose(int\* arr);  **int** tournament\_sort(**int**\* arr);  **int**\* merge\_array(**int**\*\* Arrays); | ***tournament\_merge\_test.c***  #include "lib\_tournament\_merge.h"  **int** merge\_array\_test\_time ()  {  **int** \*\*Arrays = create\_arrays(), \*array;  **for** (**long** **int** i = 1; i <= (**long** **int**)\*Arrays; i++)  tournament\_sort(\*(Arrays + i));  **double** start = (**double**)clock();  array = merge\_array(Arrays);  printf("\n�'ремя слияния 8 массивов (по 1000 элементов): %f\n", ((**double**)clock() - start) / 1000);  **for**(**int** a = 2; a <= \*array; a++) {  **if** (\*(array + a - 1) > \*(array + a)) {  printf("\nСортировка работает неправильно\n");  **break**;  }  }    **return** 0;  }  **int** merge\_array\_test ()  {  **int** \*\*Arrays = create\_arrays\_(), \*array;  **for** (**long** **int** i = 1; i <= (**long** **int**)\*Arrays; i++)  tournament\_sort(\*(Arrays + i));  **for** (**long** **int** i = 1; i <= (**long** **int**)(\*Arrays); i++)  {  printf("Array #%ld: ", i);  **for** (**int** j = 1; j <= \*(\*(Arrays + i)); j++)  {  printf("%d ", \*(\*(Arrays + i) + j));  }  printf("\n");  }  array = merge\_array(Arrays);  printf("\nMerged arrays: ");  **for**(**int** i = 1; i <= \*array; i++)  printf("%d ", \*(array + i));  **for**(**int** a = 2; a <= \*array; a++) {  **if** (\*(array + a - 1) > \*(array + a)) {  printf("\nСортировка работает неправильно\n");  **break**;  }  }    **return** 0;  } |

|  |  |
| --- | --- |
| ***tournament\_merge.c***  #include "lib\_tournament\_merge.h"  #define SWAP\_INT(a, b) int temp = a; \  a = b; \  b = temp;  #define LEAF\_POS -(number\_leaf - (deleted - tree))  **int**\* merge\_array(**int**\*\* Arrays)  {  **int** \*tree, size = 0, \*a\_pos, \*result, winner, \*deleted;  **long** **int** number\_leaf = 1, index = (**long** **int**)\*Arrays;  **if** (Arrays == NULL) **return** NULL;  **while** (number\_leaf < (**long** **int**)\*Arrays) {  size += number\_leaf;  number\_leaf \*= 2;  }  size += number\_leaf;  **if** (!(tree = (**int**\*)malloc(size \* **sizeof**(**int**)))) **return** NULL;  **for** (**long** **int** i = 1; i <= size; i++)  \*(tree + i) = INT\_MAX;  \*tree = size;  **if** (!(a\_pos = (**int**\*)malloc((**long** **int**)\*Arrays \* **sizeof**(**int**)))) **return** NULL;  **for** (**long** **int** i = 0; i < (**long** **int**)\*Arrays; i++)  \*(a\_pos + i) = 1;  size = 0;  **for** (**long** **int** i = 1; i <= (**long** **int**)\*Arrays; i++)  size += \*(\*(Arrays + i));  **if** (!(result = (**int**\*)malloc((size + 1) \* **sizeof**(**int**)))) **return** NULL;  \*result = size;  **for** (**int** i = 1; i <= (**long** **int**)\*Arrays; i++) {  \*(tree + \*tree - number\_leaf + i) = \*(\*(Arrays + i) + 1);  \*(a\_pos + i - 1) += 1;  }  winner = get\_winner(tree, number\_leaf);  index = 1;  **while** (winner != INT\_MAX) {  \*(result + index) = winner;  index++;  deleted = HEAPIFYDOWN(tree);  **if** (\*(a\_pos + LEAF\_POS) <= \*(\*(Arrays + 1 + LEAF\_POS))) {  \*deleted = \*(\*(Arrays + 1 + LEAF\_POS) + \*(a\_pos + LEAF\_POS));  }  \*(a\_pos + LEAF\_POS) += 1;    winner = HEAPIFYUP(tree, deleted);  }  free(tree);  free(a\_pos);  **return** result;  }  **int**\*\* create\_arrays()  {  **int** \*\*Arrays;  Arrays = malloc(**sizeof**(**int**\*) \* ARRAY\_NUMBER);  \*(Arrays) = (**int**\*)ARRAY\_NUMBER;  **for** (**int** i = 1; i <= ARRAY\_NUMBER; i++) {  \*(Arrays + i) = malloc(**sizeof**(**int**\*) \* SIZE\_A + 1);  \*\*(Arrays + i) = SIZE\_A;  **for** (**long** **int** j = 1; j <= (**long** **int**)\*(\*(Arrays + i)); j++)  \*(\*(Arrays + i) + j) = rand() % MAX\_VALUE;  }    **return** Arrays;  }  **int**\*\* create\_arrays\_()  { | **int** \*\*Arrays;  Arrays = malloc(**sizeof**(**int**\*) \* 4);  \*(Arrays) = (**int**\*)3;  \*(Arrays + 1) = malloc(**sizeof**(**int**\*) \* 4);  \*\*(Arrays + 1) = 4;  \*(Arrays + 2) = malloc(**sizeof**(**int**\*) \* 2);  \*\*(Arrays + 2) = 2;  \*(Arrays + 3) = malloc(**sizeof**(**int**\*) \* 3);  \*\*(Arrays + 3) = 3;  **for** (**long** **int** i = 1; i <= (**long** **int**)\*Arrays; i++)  **for** (**long** **int** j = 1; j <= (**long** **int**)\*(\*(Arrays + i)); j++)  \*(\*(Arrays + i) + j) = rand() % 20;  // for (long int i = 1; i <= (long int)\*Arrays; i++) {  // printf("Array %ld: ", i);  // for (int j = 1; j <= \*(\*(Arrays + i)); j++)  // printf("%d ", \*(\*(Arrays + i) + j));  // printf("\n");  // }    **return** Arrays;  } |

**Отзыв о работе студента**

|  |
| --- |
| Куликов Денис Александрович |
| (ФИО студента) |
|  |

Уровень освоения компетенций

|  |  |  |
| --- | --- | --- |
|  |  | |
|  | (ФИО студента) | |
|  |  | |
| Компетенции | | Уровень сформированности  компетенций |
| *ОПК-1 - Способен применять фундаментальные знания, полученные в области математических и (или) естественных наук, и использовать их в профессиональной*  *деятельности* | |  |

отметка о зачете \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Руководитель практики от СибГУТИ:

Должность руководителя подпись ФИО руководителя

"\_\_\_" \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_\_\_ г.