ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 8
BinaryHeap.FibonacciHeap.

Оглавление

Описание задачи.

Реализовать бинарную кучу(мин или макс), а так же 1 из ниже приведенных структур куч:

1. Фибоначиеву кучу
2. Биноминальную кучу

Для реализованных куч выполнить следующие действия:

1. Наполнить кучу N кол-ва элементов (где N = 10 ^ i, i от 3 до 7).
2. После заполнения кучи необходимо провести следующие тесты:
    a. 1000 раз найти минимум/максимум
    b. 1000 раз удалить минимум/максимум
    c. 1000 раз добавить новый элемент в кучу

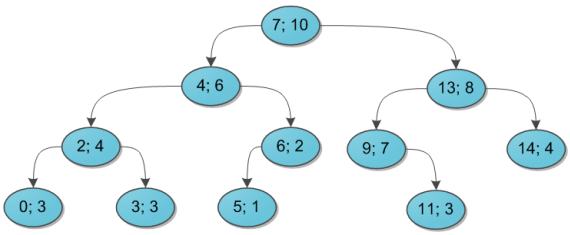
Для всех операция требуется замерить время на выполнения всей 1000 операций и рассчитать время на одну операцию, а также запомнить максимальное время, которое требуется на выполнение одной операции если язык позволяет его зафиксировать, если не позволяет воспользоваться хитростью и расчитывать усредненное время на каждые 10,25,50,100 операций, и выбирать максимальное из полученных результатов, чтобы поймать момент деградации структуры и ее перестройку.

3. По полученным в задании 2 данным построить графики времени выполнения операций для усреднения по 1000 операций, и для максимального времени на 1 операцию..

Описание метода/модели.

Куча

Куча – это особая структура, которая является деревом удовлетворяющее основному правилу кучи: все узлы потомки текущего узла по значению ключа меньше чем значение ключа текущего узла



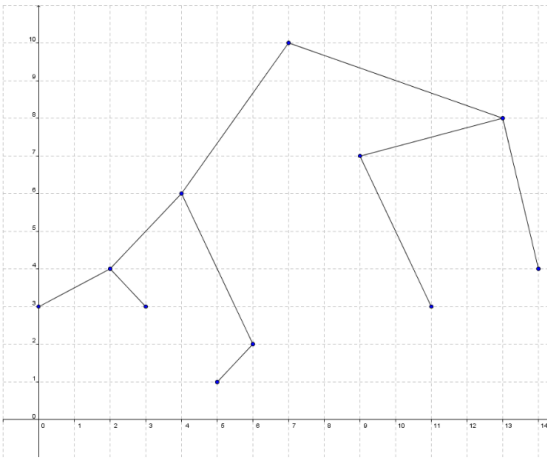


Рис.1. Декартово дерево.

Над кучами обычно проводятся следующие операции:

- найти максимум или найти минимум: найти максимальный элемент в max-куче или минимальный элемент в min-куче, соответственно
- удалить максимум или удалить минимум: удалить корневой узел в max- или min-куче, соответственно
- увеличить ключ или уменьшить ключ: обновить ключ в max- или min-куче, соответственно
- добавить: добавление нового ключа в кучу.
- слияние: соединение двух куч с целью создания новой кучи, содержащей все элементы обеих исходных.

Варианты куч

Какие кучи бывают?

- 2-3 куча
- Двуродительская куча
- Двоичная куча
- Биномиальная куча
- Очередь Бродала
- Куча с D потомками
- Фибоначчиева куча
- Куча с приоритетом самого левого
- Спаренная куча
- Асимметричная куча
- Мягкая куча
- Тернарная куча
- Декартово дерево

Операция	Двоичная	Биномиальная	Фибоначчиева	Спаренная[2]	Бродал
найти минимум	Θ(1)	Θ(log *n*) or Θ(1)	Θ(1)[1]	Θ(1)	Θ(1)
удалить минимум	Θ(log *n*)	Θ(log *n*)	O(log *n*)*	O(log *n*)*	O(log *n*)
добавить	Θ(log *n*)	O(log *n*)	Θ(1)	O(1)*	Θ(1)
уменьшить ключ	Θ(log *n*)	Θ(log *n*)	Θ(1)*	O(log *n*)*	Θ(1)
слияние	Θ(*n*)	O(log *n*)**	Θ(1)	O(1)*	Θ(1)

Рис.1. Сложность операций различных куч.

Двоичная куча

Двоичная куча (binary heap) – просто реализуемая структура данных, позволяющая быстро (за логарифмическое время) добавлять элементы и извлекать элемент с максимальным приоритетом (например, максимальный по значению).

Основной особенностью двоичной кучи является то, что каждый из узлов кучи не может иметь более чем двух потомков.

Двоичная куча отлично представляется в виде одномерного массива, при этом: нулевой элемент массива всегда является вершиной кучи, а первый и второй потомок вершины с индексом i получают свои положения на основании формул: 2 * i + 1 левый, 2 * i + 2 правый.

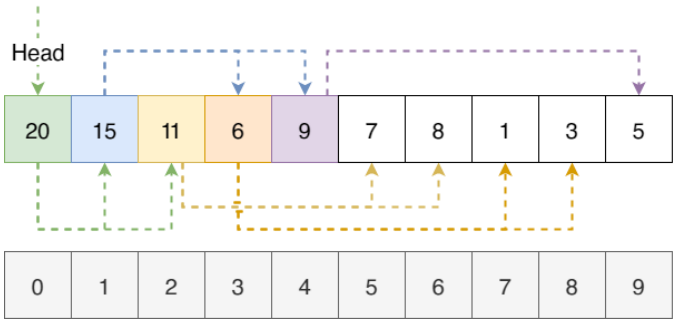


Рис.2.Реализация двоичной кучи через одномерный массив массив

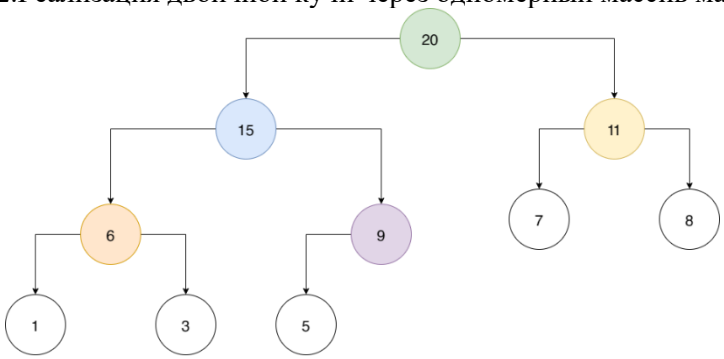


Рис.3.Двоичная куча.

Вставка

Вставка в двоичную кучу производится в крайнюю левую позицию, т.е. в элемент i равный размеру кучи. При этом может нарушаться основное правило кучи, что приводит к операции балансировки. Суть операции балансировки заключается в том, чтобы постепенно поднимать вверх значение постоянно сравнивая его с родительским, до тех пор, пока оно либо не станет корнем, либо не будет на положенном месте. Данная операция имеет логарифмическую асимптотическую сложность, но большую константу, чем аналогичная вставка для AVL дерева. Что будет заметно только на больших значениях.

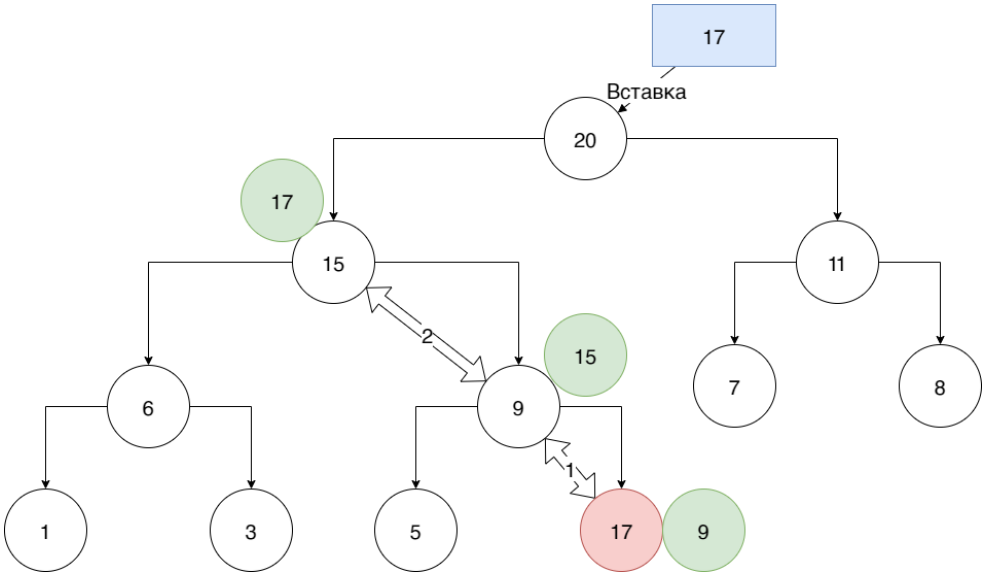


Рис.4.Вставка в двоичную кучу

Общая характеристика этого процесса может быть описана следующей формулировкой: больший элемент после вставки всплывает на ту позицию, на которой ему логично находится исходя из основного правила кучи.

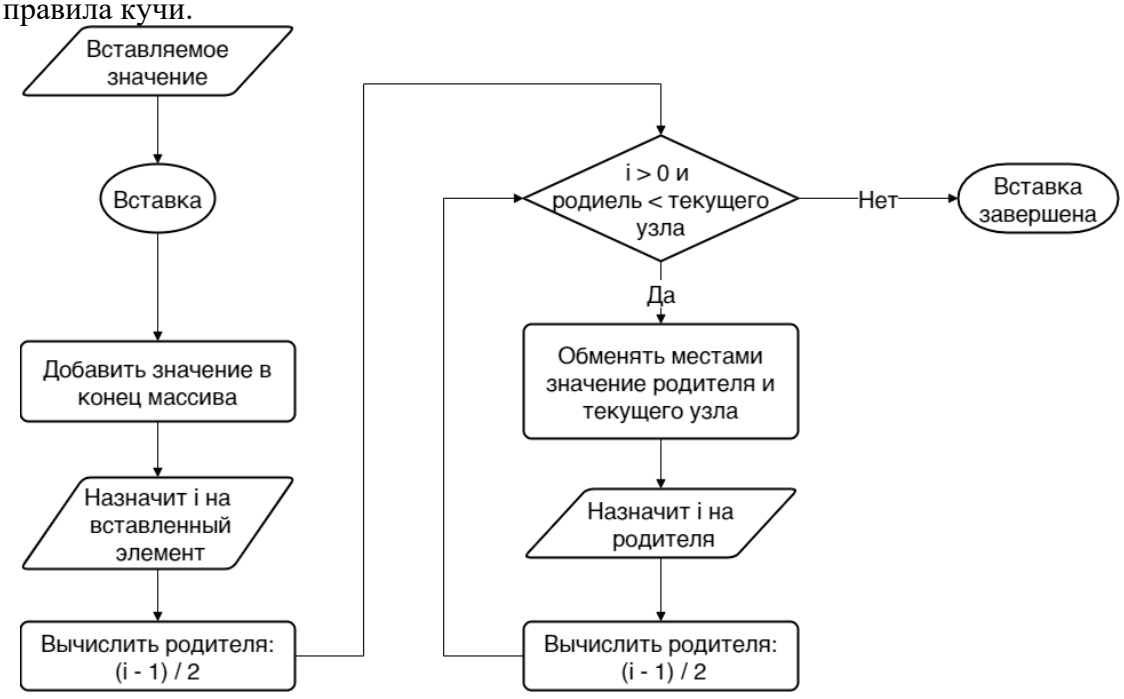


Рис.5.Алгоритм вставки в двоичную кучу

Упорядочивание

В ходе прочих операций с двоичной кучей может возникать ситуация, в которой ее придется переупорядочивать, причем такая ситуация считается возникающей только в том случае, если какой-либо корень поддерева кучи меньше чем его потомки и он не удовлетворяет основному правилу кучи.

В рамках процесса упорядочивания потребуется вычислять потомков каждого проверяемого узла, для чего используются следующие формулы:

Левый: 2 * i + 1

Правый: 2 * i + 2

Общая характеристика этого процесса может быть описана следующей формулировкой: меньший элемент во время упорядочивания тонет до той позиции на которой ему логично находится исходя из основного правила кучи.

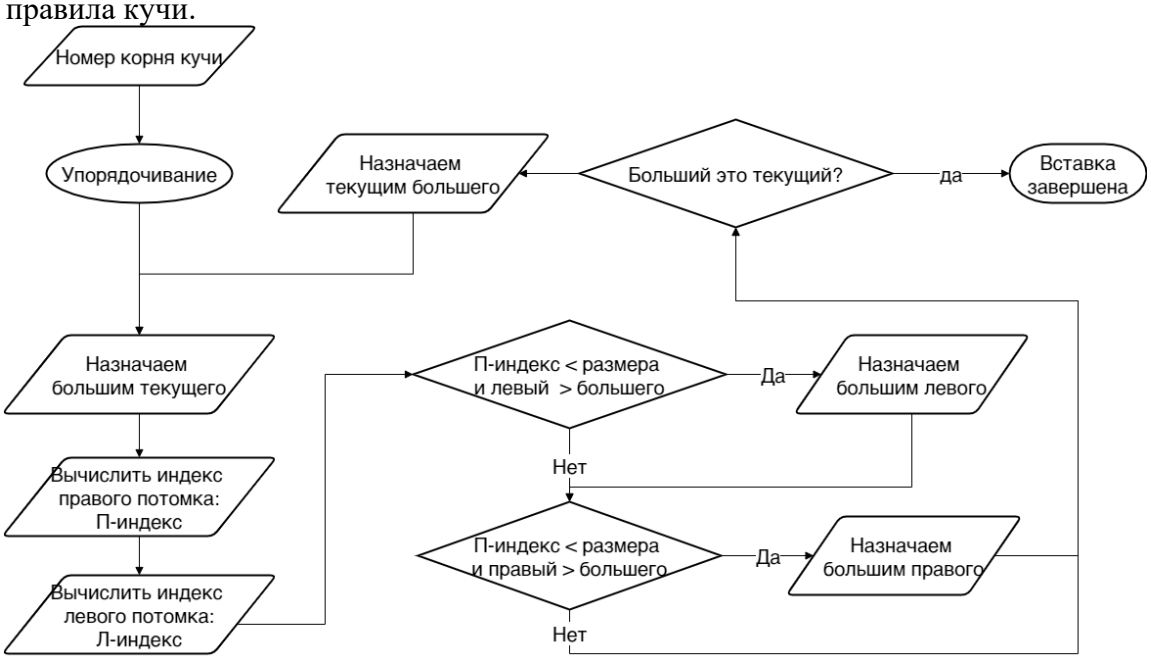


Рис.6.Алгоритм упорядочивания в двоичной куче

Удаление максимума

Для удаления максимума из кучи требуется извлечь элемент с индексом 0, а на его место поставить элемент из конца кучи, после чего вызвать упорядочивание кучи.

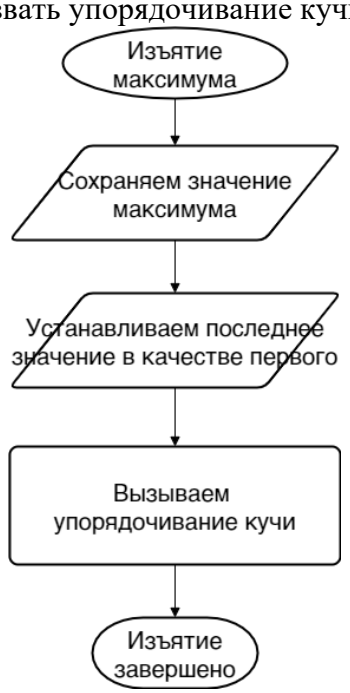


Рис.7.Алгоритм удаление максимума в двоичной куче

Выполнение задачи.

Программа для получения тестовых данных была написана на C#.

Код Бинарной кучи:

```
namespace Binar_Hip
{
    public class Bin_Hip
    {
        private List<int> root;
        /// <summary>
        /// Упорядочивание при инициализации
        /// </summary>
        /// <param name="el"></param>
        public Bin_Hip(int[] el = null)
        {
            if (el != null)
            {
                this.root = new List<int>(el);
                for (int i = el.Length / 2; i >= 0; i--)
                {
                    this.SiftDown(i);
                }
            }
            else
            {
                this.root = new List<int>();
            }
        }

        /// <summary>
        /// Изъятие минимального
        /// </summary>
        /// <returns>Изъятиый минимальный элемент</returns>
        public int RemoveMin()
        {
            var min = this.root[0];//Сохраняем первое значение
            this.root[0] = this.root[this.root.Count - 1];//Устанавливаем на первое значение последнее
            this.root.RemoveAt(this.root.Count - 1);//извлекаем последнее

            if (this.root.Count > 0)
            {
                this.SiftDown(0);//упорядочиваем
            }

            return min;
        }
        /// <summary>
        /// Нахождение минимума
        /// </summary>
        /// <returns>Минимум</returns>
        public int FindMin()
        {
            var min = this.root[0];

            return min;
        }
        /// <summary>
        /// Вставка элемента
        /// </summary>
        /// <param name="x">Вставляемый элемент</param>
        public void Insert(int x)
        {
            this.root.Add(x);//добавляем в конец новый элемент

            this.SiftUp(this.root.Count - 1);//сдвигаем его вверх
        }
        /// <summary>
```

```
        /// Упорядочивание
        /// </summary>
        /// <param name="i">индекс, с которого стартует упорядочивание</param>
        private void SiftDown(int i)
        {
            var leftChild = (i * 2) + 1;//левый потомок
            var rightChild = (i * 2) + 2;//правый потомок
            var least = i;//текущий назначается меньшим
            //если левый потомок меньше текущего, назначаем меньшим потомка
            if (leftChild < this.root.Count && this.root[leftChild] < this.root[least])
            {
                least = leftChild;
            }
            //если правый потомок меньше текущего, назначем меньшим потомка
            if (rightChild < this.root.Count && this.root[rightChild] < this.root[least])
            {
                least = rightChild;
            }
            //пока текущий не стал меньшим
            if (least != i)
            {
                (this.root[i], this.root[least]) = (this.root[least], this.root[i]);
                this.SiftDown(least);
            }
        }
        /// <summary>
        /// Сдвиг вверх
        /// </summary>
        /// <param name="i">Индекс, с которого стартует сдвиг</param>
        private void SiftUp(int i)
        {
            var parent = (i - 1) / 2;//находим родителя
            while (i > 0 && this.root[i] < this.root[parent])//проверяем меньше ли родитель, чем сравниваемый элемент
            {
                (this.root[parent], this.root[i]) = (this.root[i], this.root[parent]);//меняем местами
                i = parent;//сдвигаем наш индекс на наш элемент наверх
                parent = (i - 1) / 2;//выбираем нового родителя
            }
        }
    }

}}
```

Код кучи Фибоначчи:

```
namespace Fib_Hip
{

    /// <summary>
    /// Узлы Кучи Фибоначчи
    /// </summary>
    public class Node
    {
        public int key;//ключ
        public Node right;//правый узел
        public Node left;//левый узел
        public bool mark;//был ли удален в процессе изменения ключа ребенок этой вершины)
        public int degree;//степень вершины
        public Node child;//ребенок
        public Node parent;//родитель
        public Node(int key)
        {
            right = this;
            left = this;
            this.key = key;
        }

    }

    /// <summary>
    /// Фибоначчиева куча
    /// </summary>
    public class FibonacciHeap
    {
        private int min_key;//значения минимума
        private Node min_roote;//стартовый(минимальный) узел
        private int size;

        /// <summary>
        /// Инициализация нового дерева кучи
        /// </summary>
        /// <param name="x">Первый ключ</param>
        public FibonacciHeap(int x)
        {
            this.min_key = x;
        }
        /// <summary>
        /// Интерфейс для вставки
        /// </summary>
        /// <param name="x">Вставляемое значение</param>
        public void Insert(int x)
        {
            Insert(new Node(x));//создание нового узла x
        }


        /// <summary>
        /// Вставка нового значения
        /// </summary>
        /// <param name="node">Вставляемое дерево</param>
        public void Insert(Node node)
        {
            // добавка данного поддерева в min_node
            if (min_roote != null)//проверка кучи на пустоту
            {
                node.left = min_roote;
                node.right = min_roote.right;
                min_roote.right = node;
                node.right.left = node;

                if (node.key < min_roote.key)//меняем минимум, если новый ключ меньше
                {
                    min_roote = node;
```

```
            }
        }
        else//если да, вставляем в min_node
        {
            min_roote = node;
        }
        size++;
    }

    /// <summary>
    /// Нахождение минимума
    /// </summary>
    /// <returns>Минимальный узел</returns>
    public Node FindMin()
    {
        return min_roote;
    }

    /// <summary>
    /// Изъятия минимума
    /// </summary>
    /// <returns>Изъятый минимум</returns>
    public Node RemoveMin()
    {
        Node min_node = this.min_roote;

        if (min_node != null)
        {
            int num_kids = min_node.degree;//сохраняем степень
            Node old_min_child = min_node.child;//сохраняем детей удаляемоего узла

            // Добавляем всех детей в верхний root
            while (num_kids > 0)
            {
                Node temp_right = old_min_child.right;

              // изымаем old_min_child из списка детей, приравнивая друг-друга
                old_min_child.left.right = old_min_child.right;
                old_min_child.right.left = old_min_child.left;

                // добавляем old_min_child к верхнему списку
                old_min_child.left = this.min_roote;
                old_min_child.right = this.min_roote.right;
                this.min_roote.right = old_min_child;
                old_min_child.right.left = old_min_child;


                old_min_child.parent = null;//учитываем, что родителя теперь нет
                old_min_child = temp_right;
                num_kids--;
            }

            // изымаем min_node из списка
            min_node.left.right = min_node.right;
            min_node.right.left = min_node.left;
            //если один элемент
            if (min_node == min_node.right)
            {
                this.min_roote = null;
            }
            else// пока что перекинем указатель min на правого сына, а далее consolidate() скорректирует min в
процессе выполнения
            {
                    this.min_roote = min_node.right;
                    Consolidate();
                }
                size--;
            }

        return min_node;
```

```
}

/// <summary>
/// Уплотнение кучи
/// </summary>
private void Consolidate()
{
    //кол-во вершин в новой куче
    int arraySize = (int)Math.Floor(Math.Log(size) * 1.0 / Math.Log((1.0 + Math.Sqrt(5.0)) / 2.0)) + 1;

    var array = new List<Node>(arraySize);

    // Инициализция массива array
    for (var i = 0; i < arraySize; i++)
    {
        array.Add(null);
    }

    // поиск кол-ва вершин
    var num_roots = 0;
    Node x = min_roote;

    if (x != null)
    {
        num_roots++;
        x = x.right;

        while (x != min_roote)
        {
            num_roots++;
            x = x.right;
        }
    }

    // Для каждого узла в списка
    while (num_roots > 0)
    {

        int d = x.degree;//степень данной вершины
        Node next = x.right;

        // поиск других с такой же степенью
        for (; ; )
        {
            Node y = array[d];
            if (y == null)//если нет
            {
                break;
            }

            //Делаем того, кто меньше, ребенком того, кто больше
            if (x.key > y.key)
            {
                (x, y) = (y, x);
            }

            //связываем родителя и ребенка
            Link(y, x);

            // Переходим к другой степени
            array[d] = null;
            d++;
        }

        // Сохраяем найденный узел
        array[d] = x;

        // Сдвигаемся по списку вправо.
        x = next;
        num_roots--;
```

```
    }

    // собираем min_roote из array[], зануляя перед этим оригинальный min_roote
    min_roote = null;

    for (var i = 0; i < arraySize; i++)
    {
        Node y = array[i];
        if (y == null)
        {
            continue;
        }

        if (min_roote != null)
        {
            // Изымаем старые узлы из root
            y.left.right = y.right;
            y.right.left = y.left;

            // вставляем
            y.left = min_roote;
            y.right = min_roote.right;
            min_roote.right = y;
            y.right.left = y;

            // Проверяем не меньше ли
            if (y.key < min_roote.key)
            {
                min_roote = y;
            }
        }
        else
        {
            min_roote = y;
        }
    }
}



/// <summary>
/// Присоединяем родителя к ребенку
/// </summary>
/// <param name="new_child">Новый ребенок</param>
/// <param name="new_parent">Новый родитель</param>
private static void Link(Node new_child, Node new_parent)
{
    // изъятие new_child из списка детей
    new_child.left.right = new_child.right;
    new_child.right.left = new_child.left;

    // делаем new_child ребенком new_parent
    new_child.parent = new_parent;

    if (new_parent.child == null)
    {
        new_parent.child = new_child;
        new_child.right = new_child;
        new_child.left = new_child;
    }
    else
    {
        new_child.left = new_parent.child;
        new_child.right = new_parent.child.right;
        new_parent.child.right = new_child;
        new_child.right.left = new_child;
    }

    new_parent.degree++;
```

```
            new_child.mark = false;
        }
    }

}
```

Реализация:

1)Бинарное дерево реализовано в кач-ве отдельного класса с методами RemoveMin(), FindMin(), Insert, SiftDown, SiftUp и полями root(для хранения списка с значениями дерева)

- RemoveMin ()-Извлекает из дерева минимальный элемент
  - Сохраняет первое значение и заменят его на последнее, удаляя последний элемент массива
  - Если размер массива позволяет упорядочиваем в методе SiftDown(), который в кач-ве параметра принимает индекс в массиве.
  - В SiftDown() получаем левых и правых потомков ((i * 2) + 1 и (i * 2) + 2), создаем переменную для хранения индекса минимального элемента и приравниваем к текущему, после чего сравниваем его последовательно с левым и правым и, если один из них меньше, приравниваем переменную least к нему. После сравнений проверяем, не равен ли least текущему элементу, если равен, значит упорядочили, если нет, меняем местами элементы под индексами least и i и снова выполняем упорядочивание, но уже отправляя в кач-ве индекса least
- Добавление элемента реализуется через Insert(), который в конец массива добавляет новый элемент, после чего запускаем метод SiftUp, принимающим в кач-ве параметра индекс последнего элемента.
  - В этом методе мы сравниваем только что вставленный элемент с родителем и если он меньше, то поднимаем наверх.
- FindMin-возвращает минимальный элемент.

2)Фибоначчиеву куча я реализую в классе FibonacciHeap с полями min_roote(стартовый минимальный узел), который принадлежит классу Node с полями: key-ключ, right-правый узел, left-левый узел, mark был ли удален узел, degree-степень узла, child ребенок, parent-родитель. Также у класса кучи есть поля size для хранения размера кучи

- Insert-вставка нового значения
  - Сначала проверяем кучу на пустоту, а потом переставляем указателя на верхней уровне списка так, чтобы справа от нашего минимального и слева от старого правого помещался новый элемент
  - Если новый элемент меньше старого минимума, делаем его новым минимумом.

- RemoveMin()-отвечает за удаление минимального элемента
  - Первым делом мы переносим всех детей текущего минимального дерева на верхний уровень списка, естественно вместе с их уже детьми.
  - После изымаем минимум из верхнего списка и если у нас не всего один элемент в списке упорядочиваем нашу кучу, используя Consolidate()
  - В Consolidate воспользовавшись формулой узнаем оптимальную ширину списка верхнего и создаем с такой шириной список, который будут хранить пересобранные вершины, дальше проходимся по всем узлам и сравниваем их степень, те, у которых она одинаковая объединяются в одно дерево, где ребёнком будет тот, чей ключ меньше. Найденное сохраняем в array.
  - После чего обнуляем нашу кучу и пересобираем её из array в уже упорядоченном виде

3)Вфункции Testing тестируем программу на поиск, изъятие и вставку у обоих куч

Результаты:

1) Замерил и сравнил среднее время поиска в обоих кучах

Таблица 1

Время среднего поиска в обоих кучах

2^	Bin	Фибоначчи
13	0.739	0.581
14	0.051	0.032
15	0.05	0.032
16	0.05	0.032
17	0.051	0.032

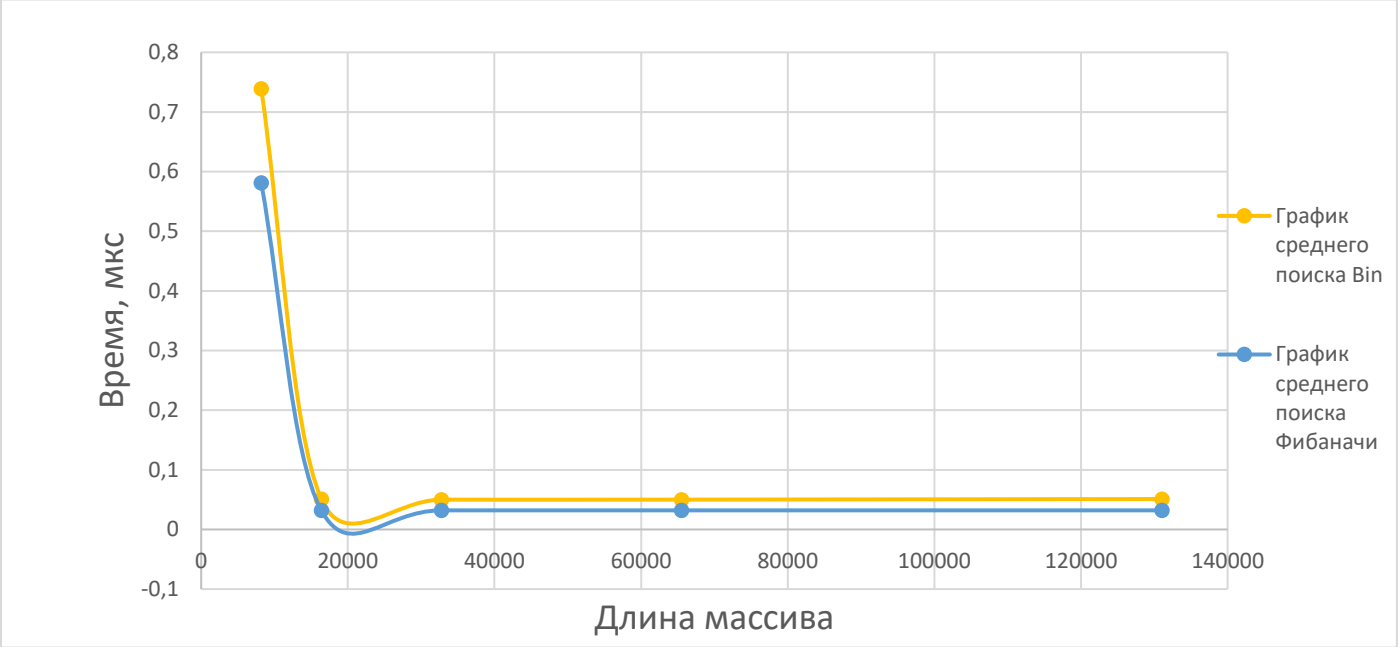


Рис.8. График сравнения среднего времени, затрачиваемого на поиск в обоих кучах

2) Замерил и сравнил среднее время удаления в обоих кучах

Таблица 2

Время среднего удаления в обоих кучах

2^	Bin	Фибоначчи
13	6.713	16.793
14	4.093	12.148
15	4.413	17.779
16	5.186	33.269
17	5.858	59.873

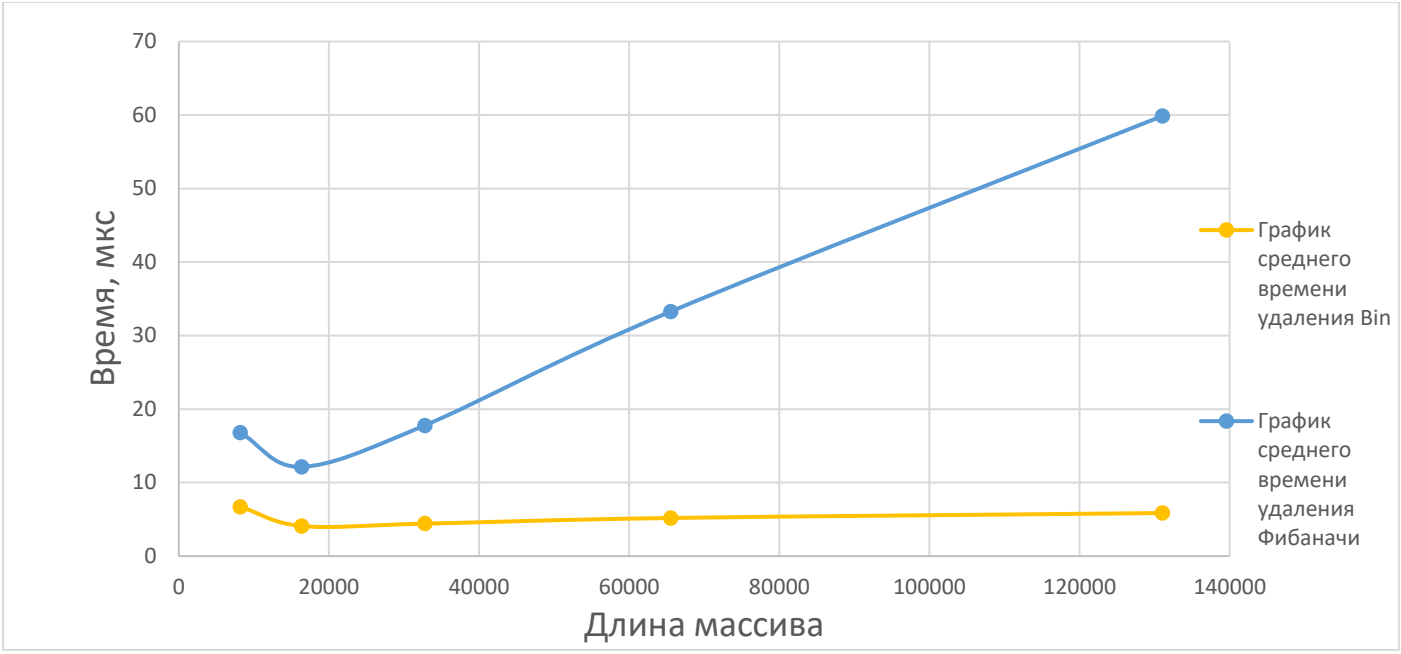


Рис.9. График сравнения среднего времени, затрачиваемого на удаление в обоих кучах

3) Замерил и сравнил среднее время вставки в обоих кучах

Таблица 3

Время средней вставки в обоих кучах

^	Bin	Фибоначчи
13	0.622	0.535
14	0.566	0.579
15	0.514	0.364
16	0.494	0.549
17	0.5	0.369

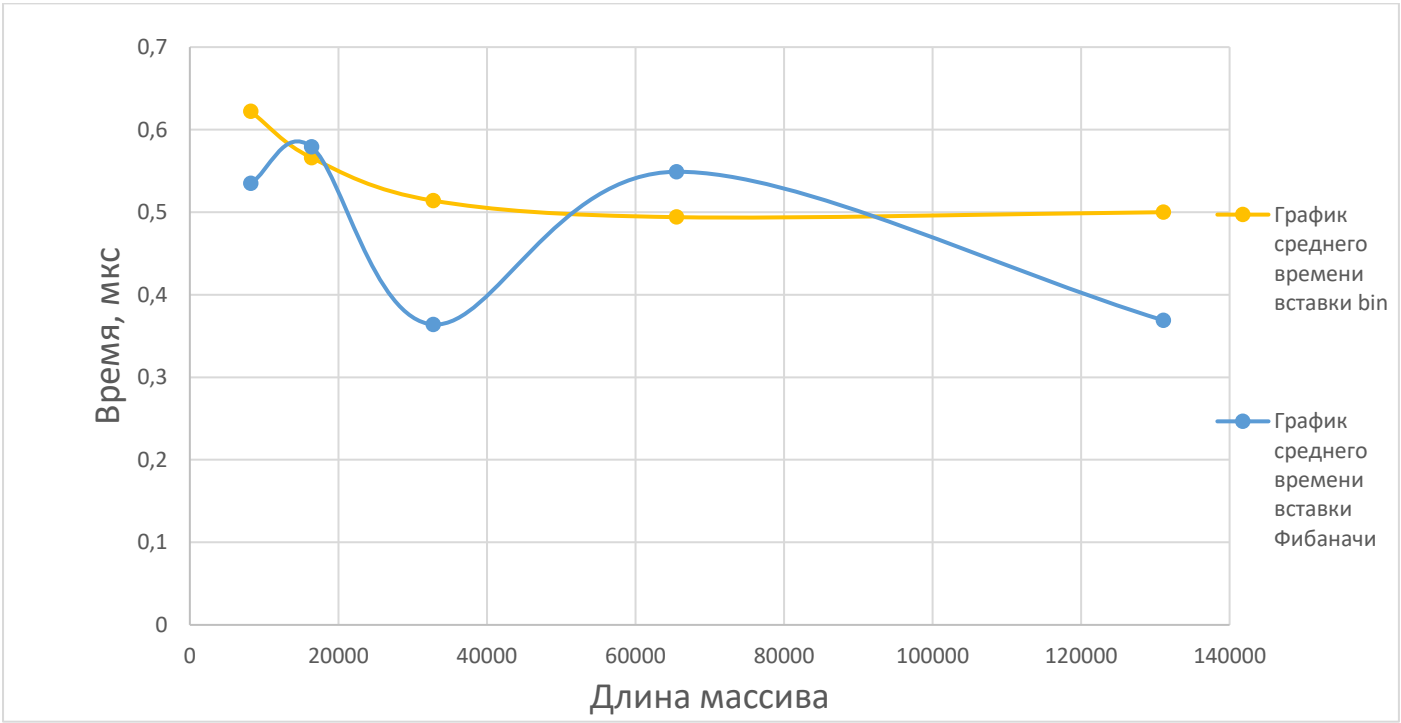


Рис.10. График сравнения среднего времени, затрачиваемого на вставку в обоих кучах

4) Замерил максимальное время поиска в обоих кучах

Таблица 4

Время максимальное время поиска в обоих кучах

2^	Bin	Фибоначчи
13	3	1
14	1	1
15	1	1
16	1	1
17	1	1

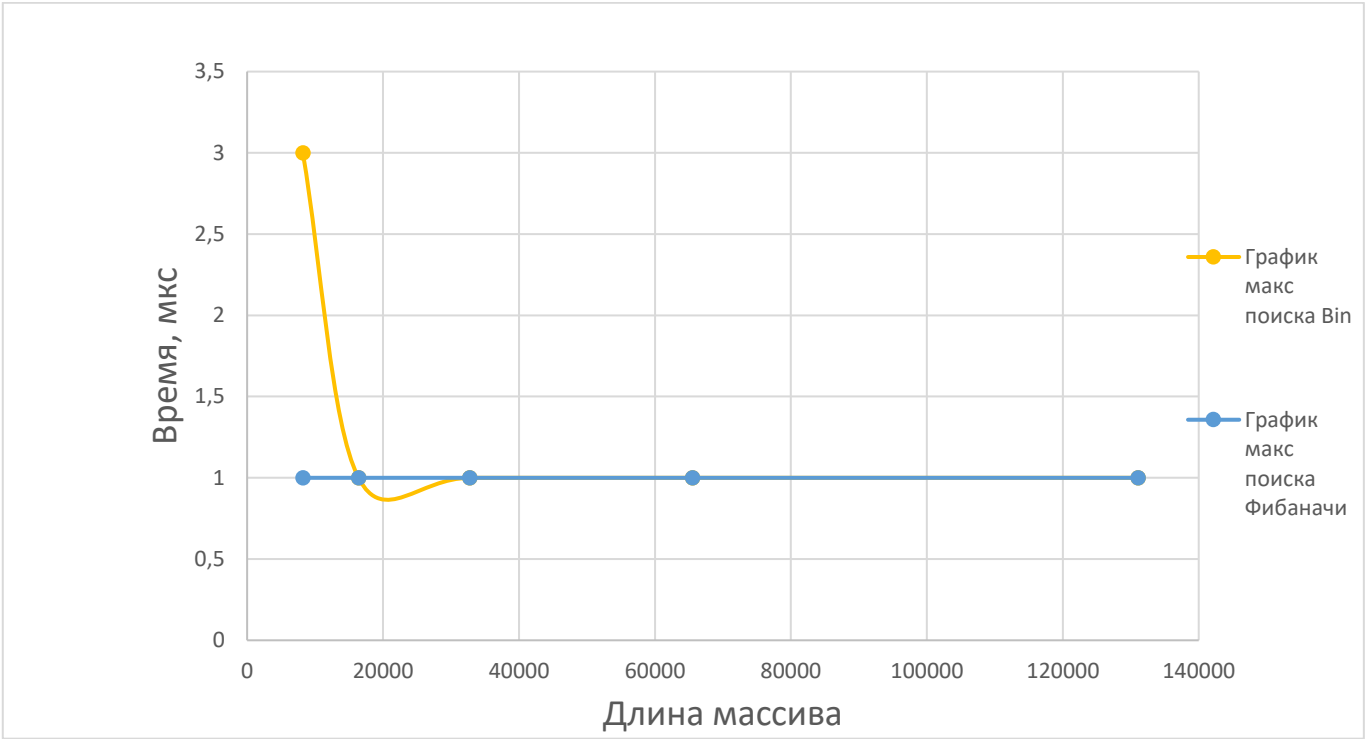


Рис.11. График сравнения максимального времени, затрачиваемого на поиск в обоих кучах

5) Замерил и сравнил макс. время удаления в обоих кучах

Таблица 5

Время максимальное время удаления в обоих кучах

2^	Bin	Фибоначчи
13	16	51
14	14	76
15	12	16
16	15	47
17	27	15

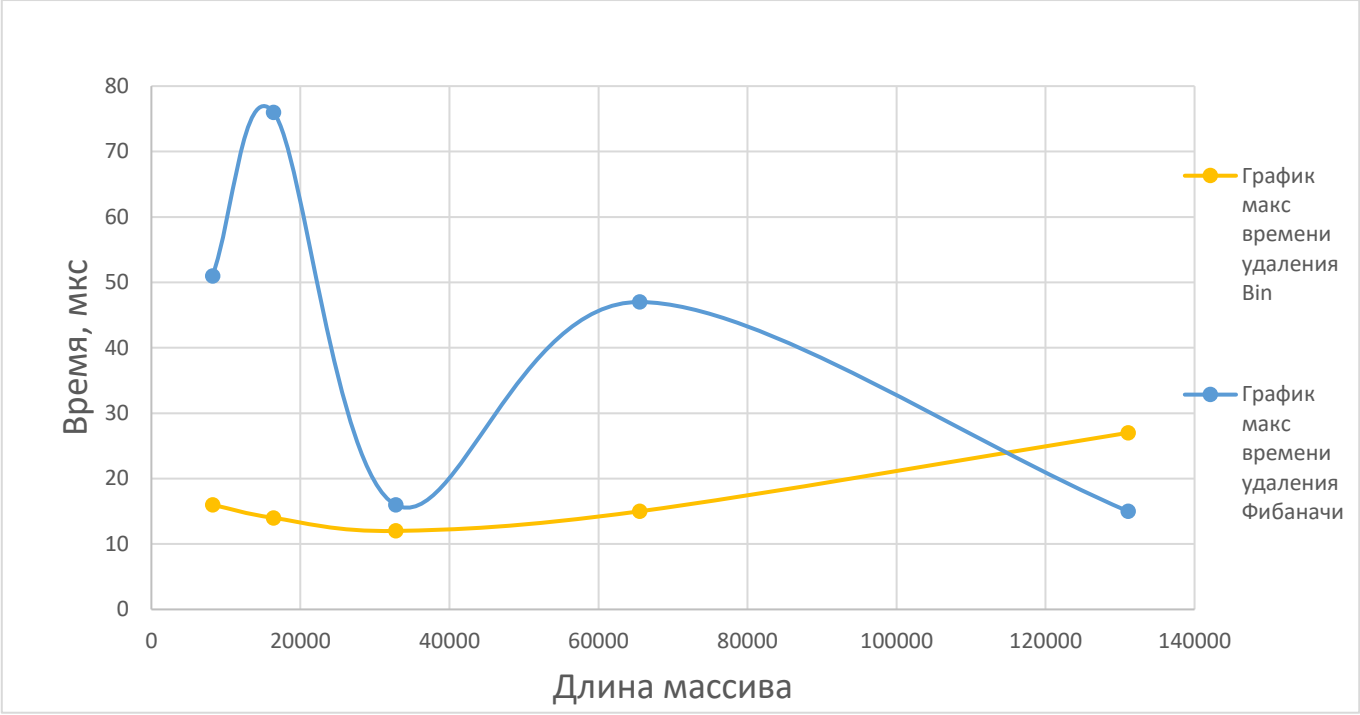


Рис.12. График сравнения макс времени, затрачиваемого на удаление в обоих кучах

6) Замерил и сравнил максимальное время вставки в обоих кучах

Таблица 6

Время максимальной вставки в обоих кучах

2^	Bin	Фибоначчи
13	3	29
14	2	28
15	3	6
16	3	31
17	2	6

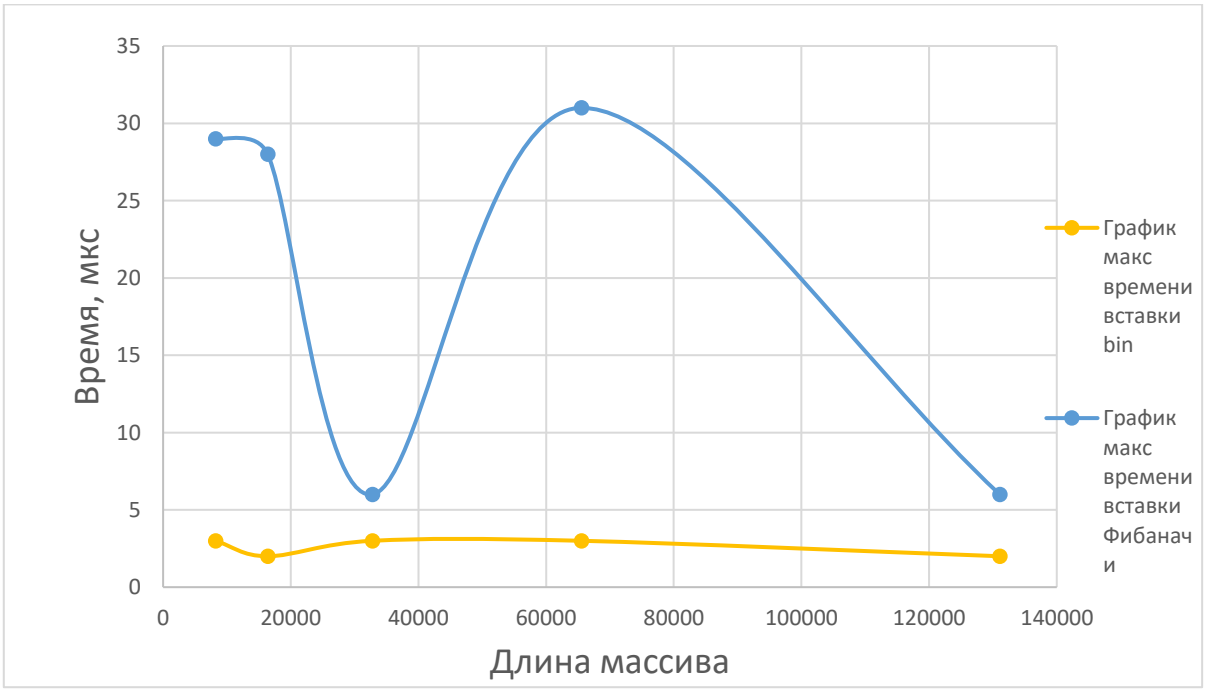


Рис.13. График сравнения среднего времени, затрачиваемого на вставку в обоих кучах

По графикам видно, что, во-первых, фибоначиева куча в среднем (за ту самое амортизированное время) показывает себя лучше бинарной, её характер ближе к единице, но удаляется она также за log, как и в бинарном.
А вот максимально время на одну операцию на всех опытах было выше у кучи Фибаначи, что говорит нам о большем влиянии на нее входных данных и показывает наглядно, что за амортизированная оценка

Заключение.

Реализация двоичной кучи оказалось достаточно простым, а вот куча Фибоначчи, особенно в моментах с упорядочиванием дерева при удалении, напротив. Из опытов стало понятно, что куча Фибоначи в среднем работает очень хорошо и эффективна, но в некоторых случаях возможен ухудшение по времени.