

Санкт-Петербургский политехнический университет Петра Великого

Институт машиностроения, материалов и транспорта

Высшая школа автоматизации и робототехники

Курсовая работа

Дисциплина: Программирование на языках высокого уровня

Тема: Heap sort

Выполнил

студент гр. 3331506/70401

Самарин А.С.

Преподаватель

Ананьевский М. С.

« » _____ 2020 г.

Санкт-Петербург

2020 г.

1.Формулировка задачи, которую решает алгоритм

Heap sort – это метод сортировки сравнением, основанный на такой структуре данных как двоичная куча.

2.Словесное описание алгоритма

Для описания алгоритма сначала необходимо дать определение двоичной куче. Двоичная куча – это такая структура данных, которая удовлетворяет следующим условиям:

- 1) Значение в любом корне больше значения двух его потомков
- 2) Заполнение слоев идет слева направо без «дырок».

Пример двоичной кучи представлен на рисунке 1.

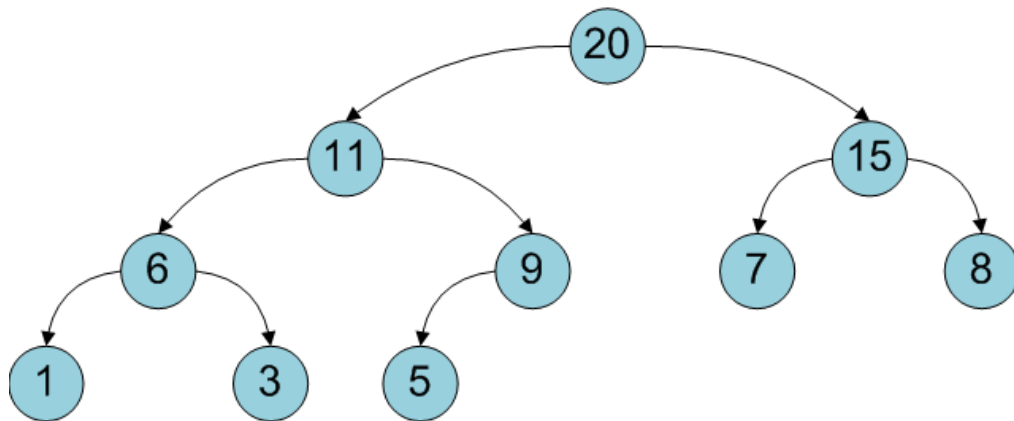


Рисунок 1 – пример двоичной кучи

Алгоритм heap sort работает следующим образом:

- 1) Из входных данных строится двоичная куча
- 2) Корень меняется местами с последним элементом кучи и размер кучи уменьшается на 1
- 3) Повторение 1 и 2 шага до тех пор, пока размер кучи не станет равен 1.

3.Реализация алгоритма

Алгоритм был реализован на языке C++. Алгоритм состоит из двух основных функций *heapify* и *heapSort*(рисунок 2).

```
//функция для преобразования в двоичную кучу поддерева с корневым узлом i
void heapify(int arr[], int n, int i)
{
    int largest = i; //корень
    int l = 2 * i + 1; //левый
    int r = 2 * i + 2; //правый

    if (l < n && arr[l] > arr[largest]) //если левый элемент больше корня
    {
        largest = l;
    }

    if (r < n && arr[r] > arr[largest]) //если правый элемент больше корня
    {
        largest = r;
    }

    if (largest != i) // если самый большой элемент не корень
    {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

void heapSort(int arr[], int n)
{
    for (int i = n / 2 - 1; i >= 0; i--) //построение кучи
        heapify(arr, n, i);

    for (int i = n - 1; i >= 0; i--)
    {
        swap(arr[0], arr[i]); //перемещение текущего элемента в конец

        heapify(arr, i, 0); //вызов на уменьшенной куче
    }
}
```

Рисунок 2 – функции

4. Анализ алгоритма

Оценим время работы функции *heapify*. На каждом шаге требуется произвести $O(1)$ действий, не считая рекурсивного вызова. Пусть $T(n)$ – время работы для поддерева, содержащего n элементов, то поддерева с корнями $left(i)$ и $right(i)$ содержат не более чем по $2n/3$ элементов каждое (наихудший случай – когда последний уровень в поддереве заполнен наполовину). Таким образом,

$$T(n) \leq T\left(\frac{2n}{3}\right) + O(1)$$

Найдем верхнюю оценку с помощью индуктивного метода. Можно предположить что, $T(n) = \log n$, то есть что $T(n) \leq cn \log n$ для подходящего $c > 0$. Пусть эта оценка верна для $(2n/3)$, то есть $T(2n/3) \leq c \log(2n/3)$. Подставив ее в соотношение, получим:

$$\begin{aligned} T(n) &\leq c \log(2n/3) + O(1) \leq c (\log(2n) - \log(3)) + O(1) \leq \\ &c (\log 2 - \log 3 + \log n) + O(1) \leq c \log n \end{aligned}$$

Последний переход закончен при $c \geq 1$.

Эту же оценку можно получить следующим образом: на каждом шаге мы опускаемся по дереву на один уровень, а высота дерева есть $O(\log n)$.

Функция *heapify* вызывается n раз, таким образом сложность алгоритма $n \log n$.

Проведем временную оценку алгоритма для трех характерных случаев:

- 1) Массив заполняется случайными значениями
- 2) Массив отсортирован в порядке уменьшения

3) В конце отсортированного массива один неотсортированный элемент

Количество элементов в массиве и время сортировки для трех случаев представлено в таблице 1.

Таблица 1 – время сортировки

	250000	300000	350000	500000	750000	1000000	1500000	2000000	3000000	5000000	7500000	10000000
1 случай	544	705	824	1198	1739	2511	3596	4847	7505	12691	18853	26880
2 случай	458	613	703	1042	1523	2062	3250	4267	6446	10724	17376	23162
3 случай	515	767	818	1072	1794	2304	3553	4509	6706	11499	17750	24015

Графики зависимости времени сортировки от количества элементов представлены на рисунке 3, 4 и 5.

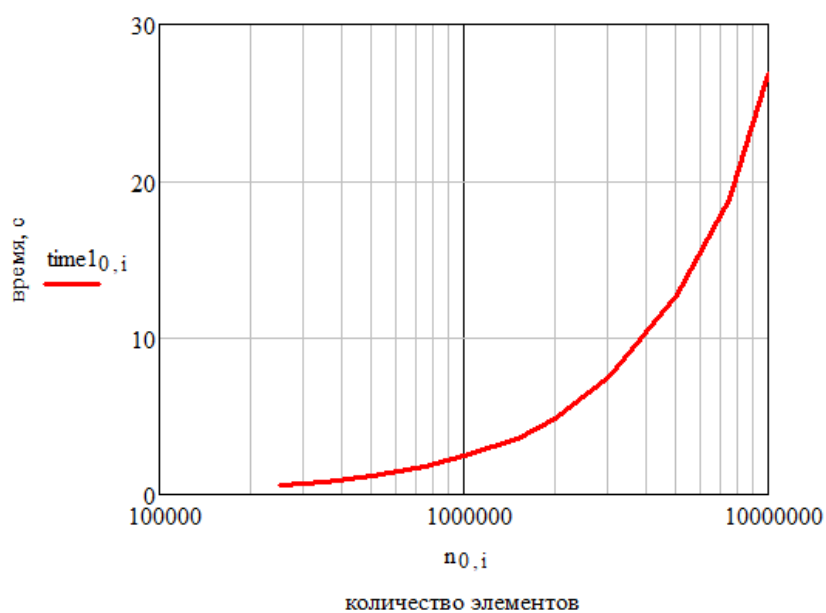


Рисунок 3 – график для первого случая

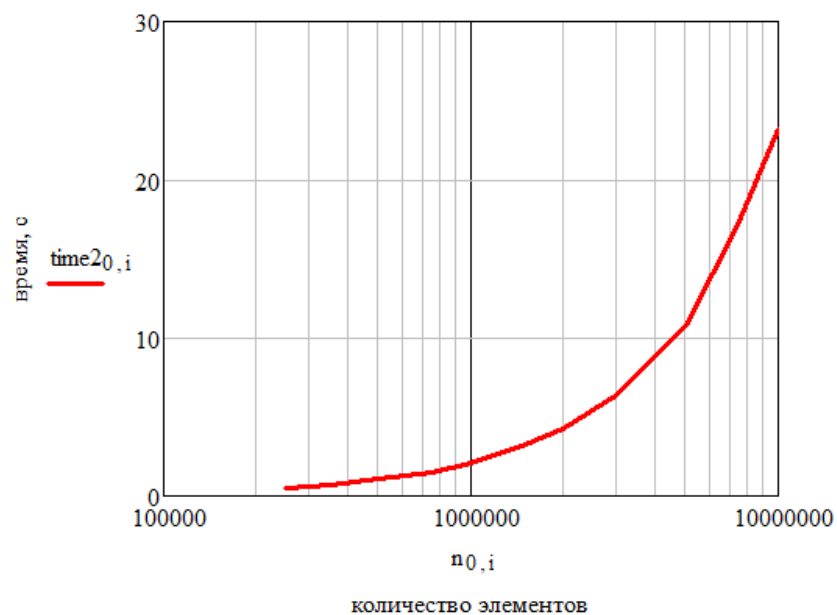


Рисунок 4 – график для второго случая

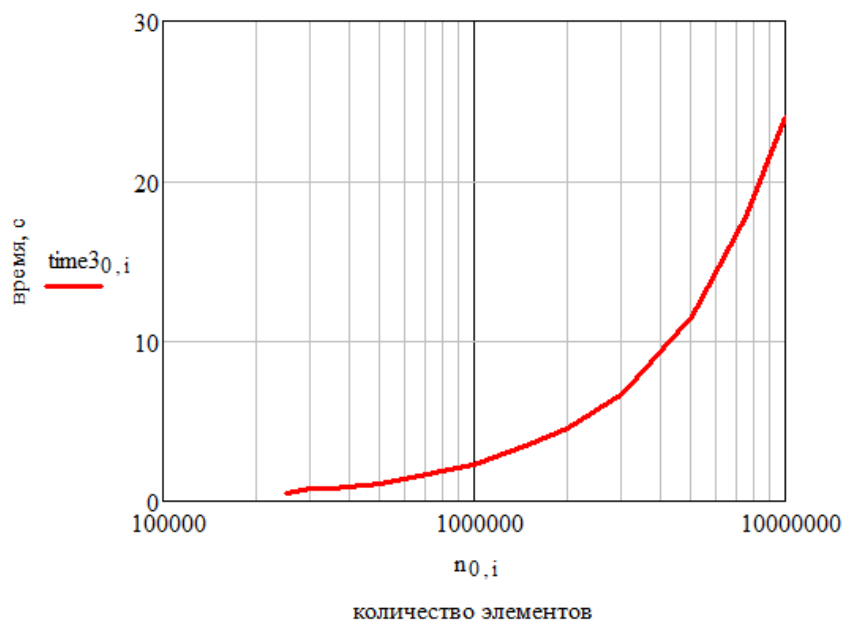


Рисунок 5 – график для третьего случая

Как можно увидеть из графиков данный алгоритм сортировки не имеет лучшего и худшего случая, как бы не был заполнен массив сортировка всегда занимает примерно одинаковое время.

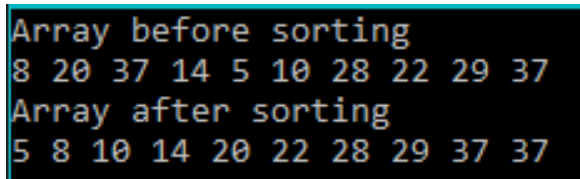
Применение алгоритма

Данный алгоритм является весьма популярным и распространенным из-за своей стабильности и применяется, в первую очередь, для сортировки.

Алгоритм «сортировка кучей» активно применяется в ядре Linux.

Может быть очень удобным если требуется отсортировать не весь массив, а получить, например, только несколько максимальных элементов. После шага построения сортирующего дерева далее следуют только несколько шагов извлечения максимального элемента из корня кучи, именно так работает `std::partial_sort()` из стандартной библиотеки C++.

Пример работы представлен на рисунке 6.



```
Array before sorting
8 20 37 14 5 10 28 22 29 37
Array after sorting
5 8 10 14 20 22 28 29 37 37
```

Рисунок 6 – пример работы

Заключение

В данной работе был рассмотрен алгоритм *heap sort*. Рассмотрена его сложность и проведена временная оценка его работы.

Список литературы

