Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Российский химико-технологический университет имени Д.И. Менделеева»
Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 8

Выполнил студент группы КС-30 Суханова Евгения Валерьевна
Ссылка на репозиторий:
https://github.com/MUCTR-IKT-CPP/EVSuhanova_30/blob/main/Algoritms/laba8.cpp

Приняли: Пысин Максим Дмитриевич
Краснов Дмитрий Олегович

Дата сдачи: 03.04.2023

Оглавление

Описание задачи.

В рамках лабораторной работы необходимо реализовать бинарную кучу(мин или макс), а так же 1 из ниже приведенных структур куч:

- Фибоначчиеву кучу
- Биномиальную кучу

Для реализованных куч выполнить следующие действия:

- Наполнить кучу N кол-ва элементов (где N = 10 ^ i, i от 3 до 7).
- После заполнения кучи необходимо провести следующие тесты:
- 1000 раз найти минимум/максимум
- 1000 раз удалить минимум/максимум
- 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 рассчитывать усредненное время на каждые 10, 25, 50, 100 операций, и выбирать максимальное из полученных результатов, чтобы поймать момент деградации структуры и ее перестройку.

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

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.

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.

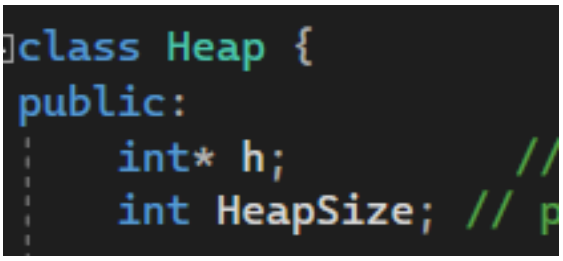
Биноминальное куча, это биномиальное дерево которое подчиняется правилу кучи, т.е. любой родитель всегда больше любого его ребенка.

При этом, биномиальное дерево всегда содержит в себе 2^k вершин для дерева BK. Т.е. если у нас имеется элементов меньше или больше чем некое 2^k, мы не можем использовать одно биномиальное дерево и нам нужно разложить количество вершин так, чтобы оно состояло из суммы 2^l(i). Важной особенностью биномиальной кучи является то, что она не должна содержать в себе деревьев одного порядка.

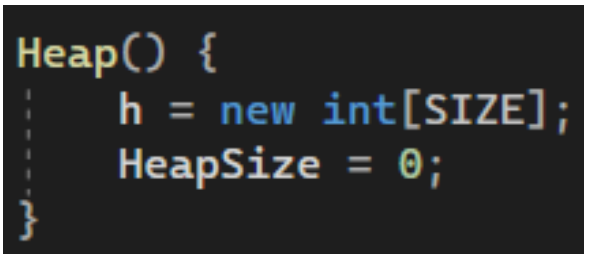
Выполнение задачи.

Бинарная куча:

1. Класс кучи. Бинарная куча имеет параметр массива, в котором хранятся все элементы кучи, и параметр размера. В конструкторе на массив выделяется максимальное количество памяти и устанавливается размер 0.

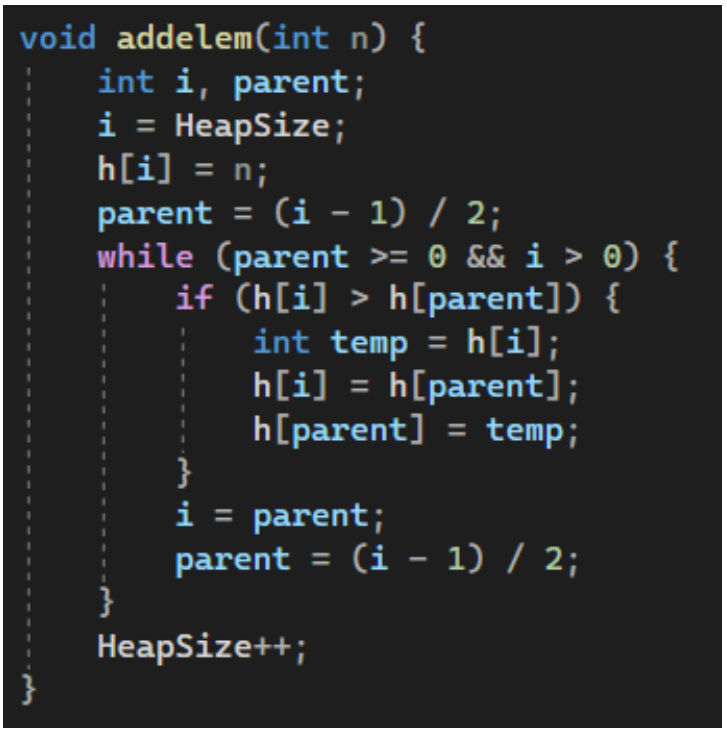


```
class Heap {
public:
    int* h;          //
    int HeapSize; // p
```



```
Heap() {
    h = new int[SIZE];
    HeapSize = 0;
}
```

2. Добавление элемента в кучу.На вход подается значение добавляемого элемента. Добавляем этот элемент в конец массива. Определяем родителя вставленного элемента. Далее путем перестановок проверяем является ли потомок меньше своего родителя. Если это не так, то переставляем их местами. Данный цикл продолжается пока мы не дойдем до верха кучи.



```
void addelem(int n) {
    int i, parent;
    i = HeapSize;
    h[i] = n;
    parent = (i - 1) / 2;
    while (parent >= 0 && i > 0) {
        if (h[i] > h[parent]) {
            int temp = h[i];
            h[i] = h[parent];
            h[parent] = temp;
        }
        i = parent;
        parent = (i - 1) / 2;
    }
    HeapSize++;
}
```

3. Упорядочивание кучи. На вход подается номер элемента в куче. Положение его правого и левого потомков находятся по формулам, описанным выше. Если полученное правое или левое значение меньше размера кучи, то происходит упорядочивание. Идет проверка того, что родитель меньше своего потомка. В зависимости от того правый потомок больше или левый, происходит перестановка этого потомка и родителя. Затем для нового потомка вызывается рекурсия этой функции.

```
void heapify(int i) {
    int left, right;
    int temp;
    left = 2 * i + 1;
    right = 2 * i + 2;
    if (left < HeapSize) {
        if (h[i] < h[left]) {
            temp = h[i];
            h[i] = h[left];
            h[left] = temp;
            heapify(i: left);
        }
    }
    if (right < HeapSize) {
        if (h[i] < h[right]) {
            temp = h[i];
            h[i] = h[right];
            h[right] = temp;
            heapify(right);
        }
    }
}
```

4. Поиск максимального элемента. В данной куче максимальный элемент всегда булет корневым.

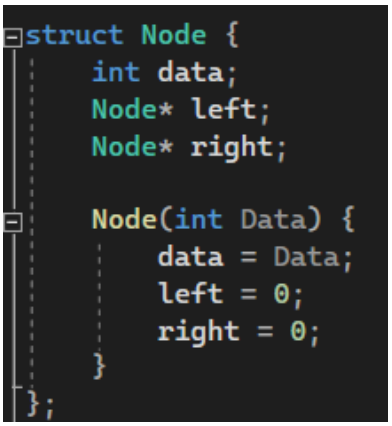
```
int findmax() {
    return h[0];
}
```

5. Удаление максимального элемента. На место корневого элемента ставится последний элемент и вызывается упорядочивание кучи относительно корня.

```
int getmax() {
    int x;
    x = h[0];
    h[0] = h[HeapSize - 1];
    HeapSize--;
    heapify(i: 0);
    return(x);
}
```

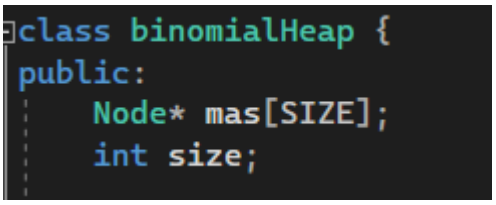
Биномиальная куча:

1. Структура для элемента кучи. Один элемент кучи имеет значение, а также ссылки на правого и левого потомков.



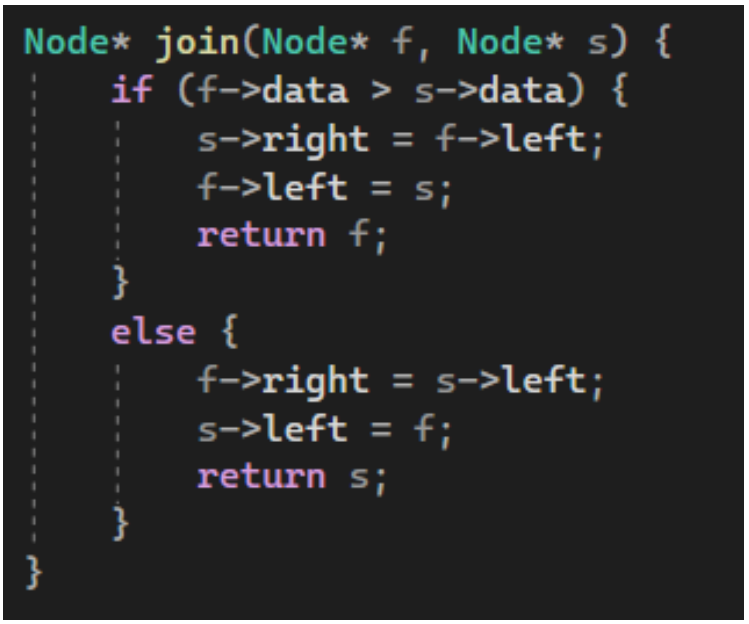
```
struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int Data) {
        data = Data;
        left = 0;
        right = 0;
    }
};
```

2. Класс кучи. Куча имеет параметр общего размера, а также массив всех элементов кучи.



```
class binomialHeap {
public:
    Node* mas[SIZE];
    int size;
```

3. Функция объединения двух куч одинакового размера. На вход подаются два элемента кучи. Далее проверяется чье значение больше. Больший элемент встает в корень, меньший становится его левым потомком. Далее потомки большего элемента уходят вправо, а меньше влево.



```
Node* join(Node* f, Node* s) {
    if (f->data > s->data) {
        s->right = f->left;
        f->left = s;
        return f;
    }
    else {
        f->right = s->left;
        s->left = f;
        return s;
    }
}
```

4. Добавление элемента в кучу. На вход подается значение элемента. Создается элемент кучи с этим значением. Далее происходит попытка вставки этого элемента. Если в массиве на этом месте нет элемента, то мы записываем наш новый элемент. Если там что-то есть, то вызывает функция объединения двух куч одинакового размера для нашего нового элемента и элемента, на котором мы находимся.

```
void add(int val) {
    Node* newNode = new Node(val);
    Node* curNode = newNode;
    for (int i = 0; i < SIZE; i++) {
        if (mas[i] == NULL) {
            mas[i] = curNode;
            break;
        }
        else {
            curNode = join(f: curNode, mas[i]);
            mas[i] = NULL;
        }

    }
    size++;
}
```

5. Функция удаления максимального элемента. Так как это биномиальная куча, то мы не можем просто удалить корневой элемент, так как не факт, что он максимальный. Поэтому для начала мы находим этот элемент и его позицию в куче.

```
int getMax() {
    int res = 0;
    int resPos = 0;
    for (int i = 0; i < SIZE; i++) {
        if (mas[i] && mas[i]->data > res) {
            res = mas[i]->data;
            resPos = i;
        }
    }
```

Далее мы выделяем временную кучу. После мы проходим по всем элемент,которые находятся слева от максимального, и записываем их в нашу новую кучу. После его мы удаляем наш максимальный элемент из старой кучи и вызываем функцию объединения двух деревьев для нашей оставшейся кучи и временной, в которую мы записали часть элементов. Удаляем временную кучу.

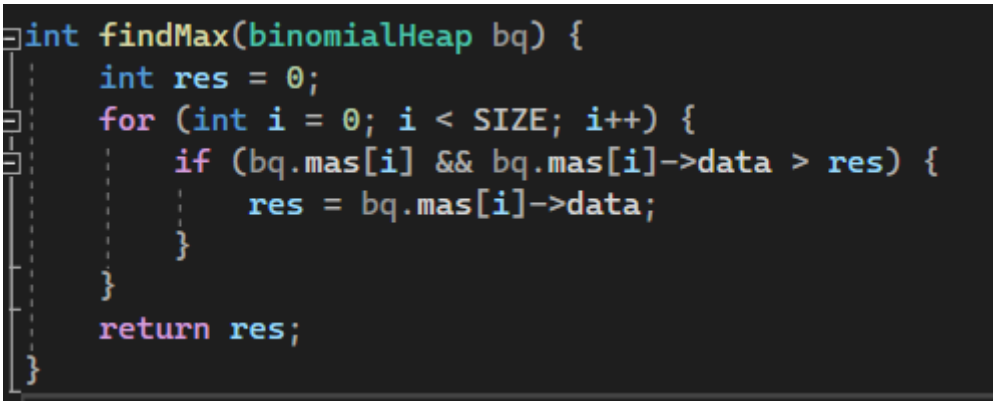
```
Node** tmp = new Node * [SIZE];
memset( _Dst: tmp, _Val: 0, _Size: sizeof(tmp) * SIZE);
Node* cur = 0;
if (!mas[resPos])
    cur = 0;
else
    cur = mas[resPos]->left;
for (int i = resPos - 1; i >= 0; i--) {
    tmp[i] = new Node(*cur);
    cur = cur->right;
    tmp[i]->right = 0;
}
delete mas[resPos];
mas[resPos] = 0;

joinBQ(mas, b: tmp);
delete[] tmp;
size--;
return res;
```

6. Функция объединения двух куч разного размера. На вход подаются две кучи разного размера. Далее выполняется оператор switch. В функцию num подаются значения 1 или 0, что является результатов проверки на пустоту каждой кучи (c - новая куча). В зависимости от результата данной функции мы выполняем либо запись в новую кучу с объединения, либо приравнивание к элементу первой кучи какого-либо значения, либо, как в случае с 6, ничего не делаем. Данная функция это аналогия сложения двоичных чисел.

```
int num(int c, int b, int a) {
    return 4 * c + 2 * b + a;
}
```

```
void joinBQ(Node* a[], Node* b[]) {
    Node* c = 0;
    for (int i = 0; i < SIZE; i++) {
        switch (num(c != 0, b[i] != 0, a[i] != 0)) {
        case 2:
            a[i] = b[i];
            break;
        case 3:
            c = join( f: a[i], s: b[i]);
            a[i] = 0;
            break;
        case 4:
            a[i] = c;
            c = 0;
            break;
        case 5:
            c = join( f: a[i], s: c);
            a[i] = 0;
            break;
        case 6:
        case 7:
            c = join( f: b[i], s: c);
            break;
        }
    }
}
```

7. Поиск максимального элемента кучи. Проходим по всем элементам кучи и находим максимальное значение.

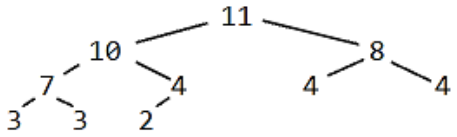


```
int findMax(binomialHeap bq) {
    int res = 0;
    for (int i = 0; i < SIZE; i++) {
        if (bq.mas[i] && bq.mas[i]->data > res) {
            res = bq.mas[i]->data;
        }
    }
    return res;
}
```

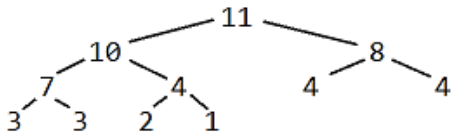
Результаты работы алгоритма

Для демонстрации работоспособности алгоритма в отчете были проведены тесты на меньших значениях куч.

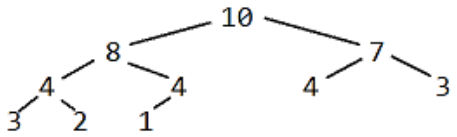
Сгенерированная куча из 10 элементов



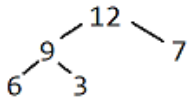
Добавление в кучу элемента



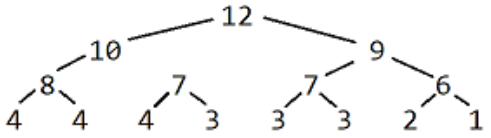
Удаление максимального элемента



Сгенерированная куча по-меньше



Объединение двух куч разного размера

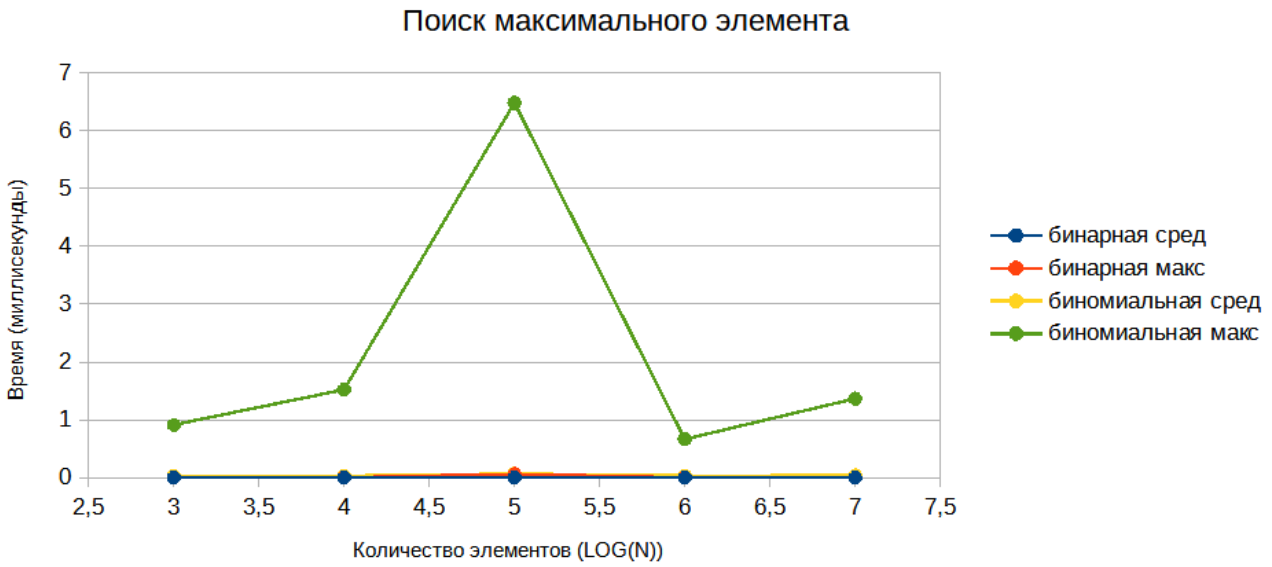


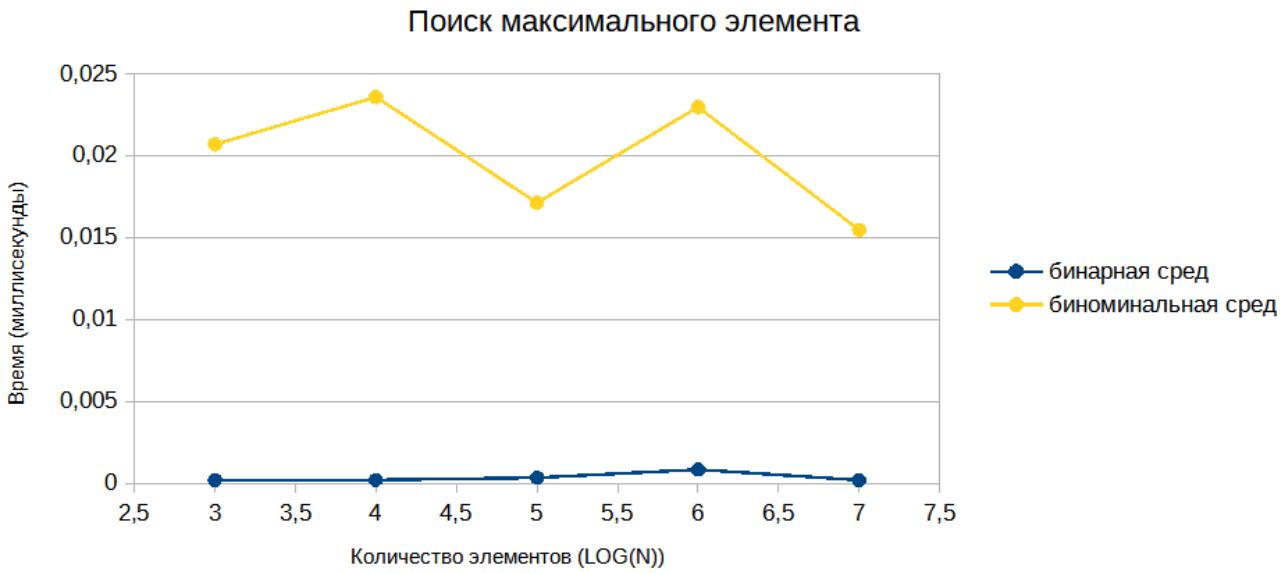
Вывод максимального элемента
12

Полученные результаты

Поиск максимального элемента кучи:

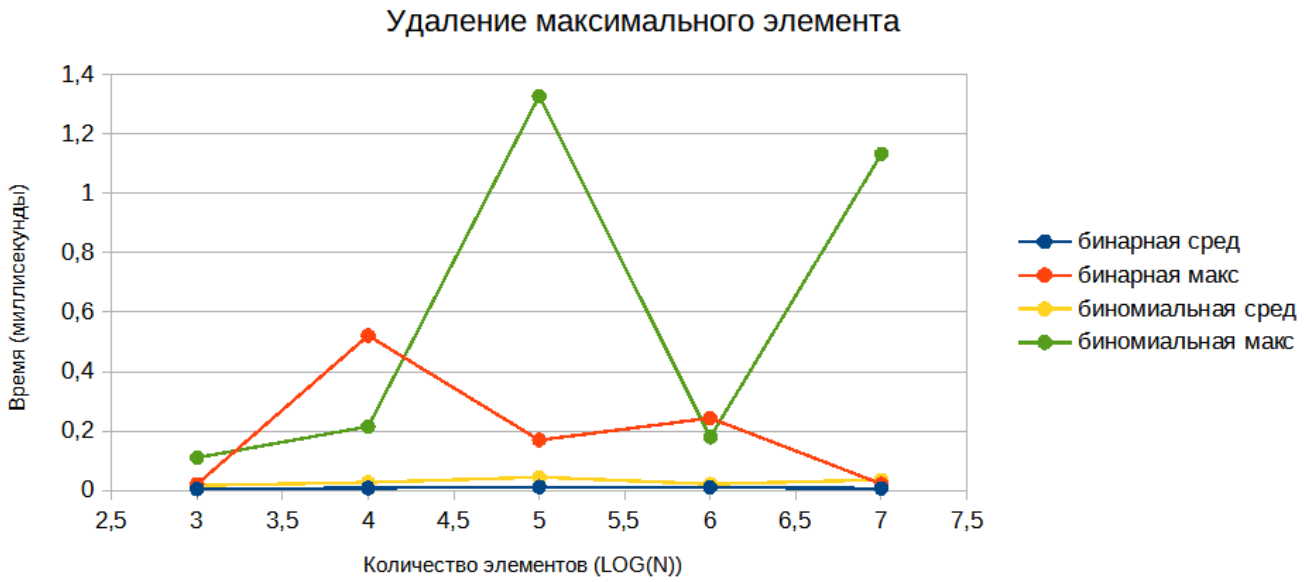
Поиск максимального					
Кол-во LOG(N)	**3**	**4**	**5**	**6**	**7**
бинарная сред	0,0000859	0,0000900	0,0001685	0,0000575	0,0000471
бинарная макс	0,0005000	0,0017000	0,0612000	0,0004000	0,0003000
биномиальная сред	0,0332799	0,0310494	0,066341	0,033811	0,0365261
биномиальная макс	0,9128	1,5216	6,4779	0,6635	1,366

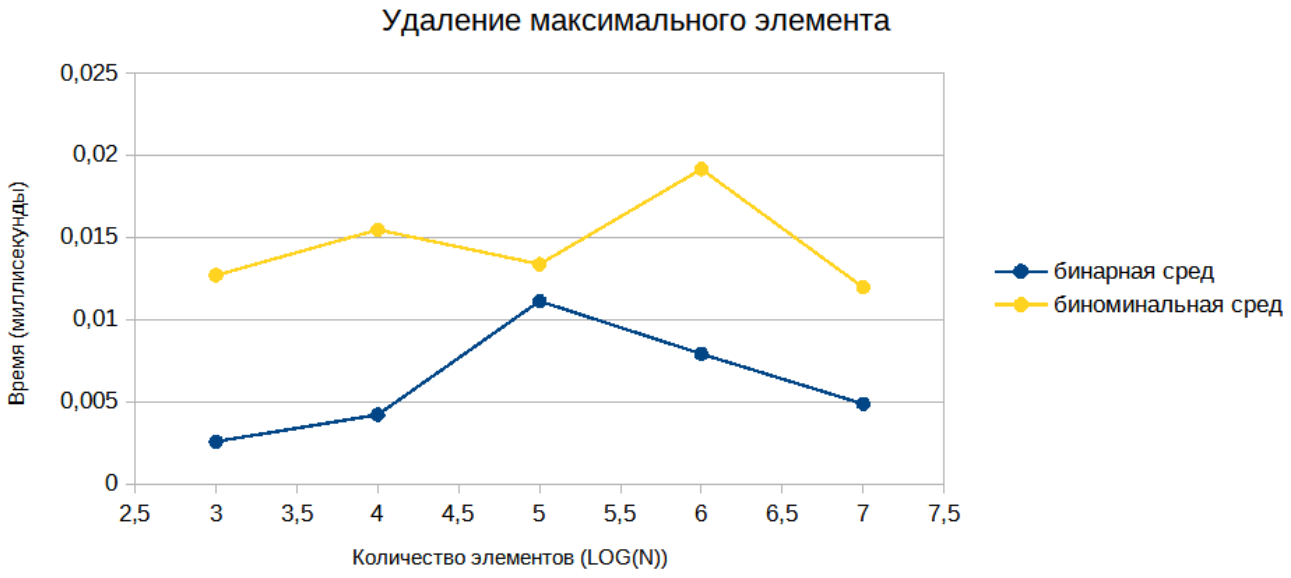




Удаление максимального элемента кучи:

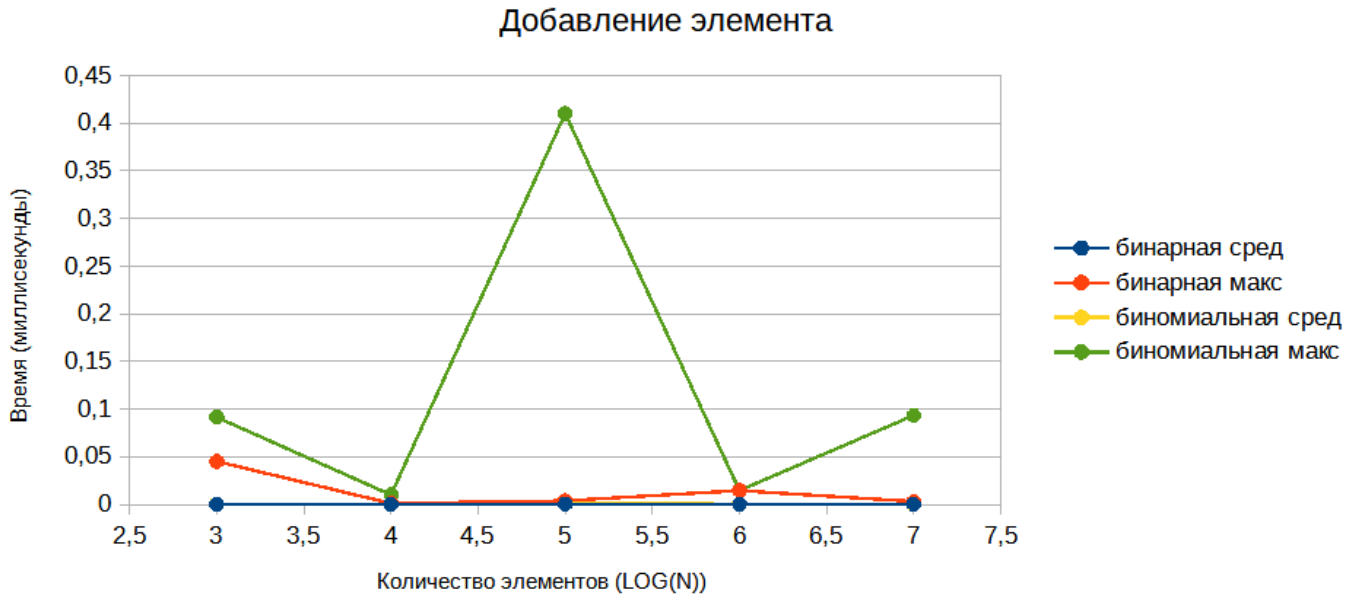
Удаление максимального					
Кол-во LOG(N)	**3**	**4**	**5**	**6**	**7**
бинарная сред	0,0039937	0,0082139	0,0118351	0,0107724	0,0063584
бинарная макс	0,0210000	0,5212000	0,1692000	0,2434000	0,0219000
биномиальная сред	0,0165758	0,0267627	0,0444552	0,021436	0,0355076
биномиальная макс	0,1102	0,2149	1,3271	0,1793	1,132

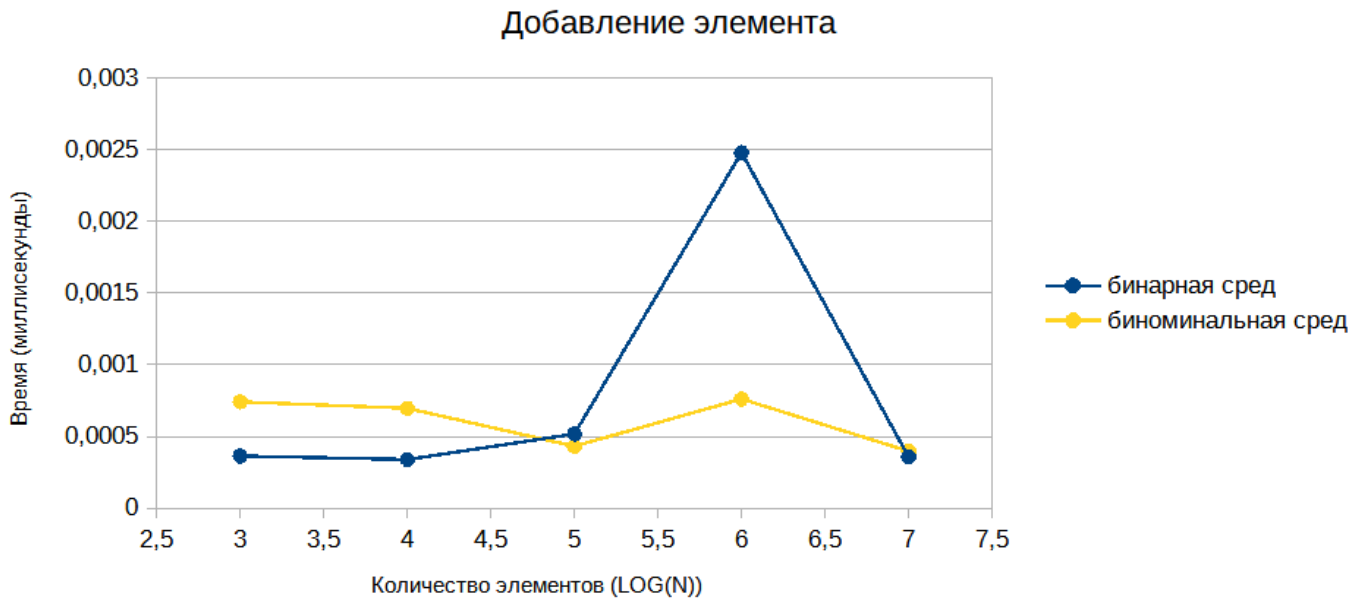




Добавление элемента в кучу:

Добавление					
Кол-во LOG(N)	**3**	**4**	**5**	**6**	**7**
бинарная сред	0,0002797	0,0002599	0,0004083	0,0003635	0,0002533
бинарная макс	0,0453000	0,0012000	0,0040000	0,0147000	0,0032000
биномиальная сред	0,0005714	0,0004979	0,0011895	0,0006843	0,000564
биномиальная макс	0,0916	0,0102	0,41	0,0147	0,0936





Заключение.

Из полученных результатов можно увидеть, что операции над обычной бинарной кучей занимают меньше времени нежели с биномиальной кучей. Также при увеличении количества элементов кучи возрастал риск появления сильной деградации алгоритма, то есть максимальное время на операцию сильно увеличивалось.

Также очевидно, что при увеличении количества элементов, занималось больше памяти, что также сильно влияло на скорость работы алгоритма.

Сложность обеих куч одинаковая (log(N)). Однако стоит заметить, что биномиальная куча имеет отличительную особенность. Она дает возможность производить объединение двух куч (как одинакового, так и разного размеров). В бинарной куче данный функционал реализовать сложнее. Поэтому стоит отметить, что хоть биномиальная куча и проигрывает по времени во всех операциях, но она дает возможность объединять кучи, что довольно часто может пригодится при работе с ними.