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 № 7

Выполнил студент группы КС-30 Суханова Евгения Валерьевна
Ссылка на репозиторий:
https://github.com/MUCTR-IKT-CPP/EVSuhanova_30/blob/main/Algoritms/laba7.cpp

Приняли: Пысин Максим Дмитриевич
Краснов Дмитрий Олегович

Дата сдачи: 03.04.2023

Оглавление

Описание задачи.

В рамках лабораторной работы необходимо изучить одно из двух деревьев поиска:

Рандомизированное дерево (https://habr.com/ru/post/145388/)

Для этого его потребуется реализовать и сравнить в работе с реализованным ранее AVL-деревом. Для анализа работы алгоритма понадобиться провести серии тестов:

1. В одной серии тестов проводится 50 повторений
2. Требуется провести серии тестов для N = 2^i элементов, при этом i от 10 до 18 включительно.

В рамках одной серии понадобится сделать следующее:

- Генерируем N случайных значений.
- Заполнить два дерева N количеством элементов в одинаковом порядке.
- Для каждого из серий тестов замерить максимальную глубину полученного деревьев.
- Для каждого дерева после заполнения провести 1000 операций вставки и замерить время.
- Для каждого дерева после заполнения провести 1000 операций удаления и замерить время.
- Для каждого дерева после заполнения провести 1000 операций поиска.
- Для каждого дерева замерить глубины всех веток дерева.

Для анализа структуры потребуется построить следующие графики:

- График зависимости среднего времени вставки от количества элементов в изначальном дереве для вашего варианта дерева и AVL дерева.
- График зависимости среднего времени удаления от количества элементов в изначальном дереве для вашего варианта дерева и AVL дерева.
- График зависимости среднего времени поиска от количества элементов в изначальном дереве для вашего варианта дерева и AVL дерева.
- График максимальной высоты полученного дерева в зависимости от N.
- Гистограмму среднего распределения максимальной высоты для последней серии тестов для AVL и для вашего варианта.
- Гистограмму среднего распределения высот веток в AVL дереве и для вашего варианта, для последней серии тестов.

Описание метода/модели.

Бинарное дерево — это иерархическая структура данных, в которой каждый узел имеет значение (оно же является в данном случае и ключом) и ссылки на левого и правого потомка.

Можно подобрать такую последовательность операций с бинарным деревом поиска в наивной реализации, что его глубина будет пропорциональна количеству ключей, а следовательно операции будут выполняться за O(n). Поэтому, если поддерживать инвариант "случайности" в дереве, то можно добиться того, что математическое ожидание глубины дерева будет небольшим. Дадим рекурсивное определение рандомизированного бинарного дерева поиска.

Необходимой составляющей рандомизации является применение специальной вставки нового ключа, в результате которой новый ключ оказывается в корне дерева (полезная функция во многих отношениях, т.к. доступ к недавно вставленным ключам оказывается в результате очень быстрым). Для реализации вставки в корень нам потребуется функция поворота, которая производит локальное преобразование дерева.

Сбалансированное AVL-дерево поиска — это бинарное дерево поиска с логарифмической высотой. Данное определение скорее идейное, чем строгое. Строгое определение оперирует разницей глубины самого глубокого и самого неглубокого листа (в AVL-деревьях) или отношением глубины самого глубокого и самого неглубокого листа. В сбалансированном бинарном дереве поиска операции поиска, вставки и удаления выполняются за логарифмическое время (так как путь к любому листу от корня не более логарифма). В вырожденном случае несбалансированного бинарного дерева поиска, например, когда в пустое дерево вставлялась отсортированная последовательность, дерево превратится в линейный список, и операции поиска, вставки и удаления будут выполняться за линейное время. Поэтому балансировка дерева крайне важна. Технически балансировка осуществляется поворотами частей дерева при вставке нового элемента, если вставка данного элемента нарушила условие сбалансированности.

Выполнение задачи.

Описание всех функций, с помощью которых будем проводить различные операции над бинарными деревьями:

1. Для начала зададим структуру одного узла дерева (где key - значение узла, size - высота, на которой находится узел, left - левый потомок, right - правый потомок).

```
struct node // структура для представления узлов дерева
{
    int key;
    int size;
    node* left;
    node* right;
    node(int k) { key = k; left = right = 0; size = 1; }
};
```

2. Добавление узла в дерево. Если узел не конечный, то мы определяем в какую сторону необходимо двигаться для правильной постановки нового узла. Если узел конечный, то ставится новый узел. Для AVL-дерева вызывается балансировка в конце рекурсии. Для рандомизированного дерева вызывается функция поворота относительно левого или правого узла.

```
/* Добавление узла */
node* addnode(int k, node* p, bool AVL) {
    if (!p) return new node(k); // пока не найдем пустое место
    if (k <= p->key) { // Если значение меньше или равно текущему узлу
        p->left = addnode(k, p->left, AVL);  // идем влево
        return AVL ? balance(p) : rotateright(p);
    }
    else {
        p->right = addnode(k, p->right, AVL); // идем вправо
        return AVL ? balance(p) : rotateleft(q: p);
    }
}
```

4. Рандомизированная вставка узла. С некоторой вероятностью в рандомизированном дереве может быть случайная вставка.

```
node* insert_random(node* p, int k) // рандомиз
{
    if (!p) return new node(k);
    if (rand() % (p->size + 1) == 0)
        return addnode(k, p, AVL: 0);
    if (p->key > k)
        p->left = insert_random(p->left, k);
    else
        p->right = insert_random(p->right, k);
    fixsize(p);
    return p;
}
```

3. Поиск узла в бинарном дереве. Определяем меньше ли необходимое значение текущего узла. Если да, то двигаемся в левую сторону дерева, иначе в правую сторону.

```
void reverse(node* p, int k) {
    if (p->key == k) cout << "Найден узел со значением " << k << endl;
    else {
        if (k < p->key) reverse(p->left, k);        //Рекурсивная функц
        else reverse(p->right, k);                  //Рекурсивная функц
    }
}
```

4. Вывод дерева в консоль. Выводим с самого левого узла к самому правому с помощью рекурсий.

```
// Вывод бинарного дерева
void print_Tree(node* p, int level) {
    if (p) {
        print_Tree(p->right, level + 1);
        for (int i = 0; i < level; i++) cout << "    ";
        cout << p->key << endl;
        print_Tree(p->left, level + 1);
    }
}
```

5. Удаление узла из дерева. Проверяем есть ли потомки у данного узла. Определяем в каком потомке находится узел для удаления. С помощью рекурсивной перестановки удаляем нужный узел и поднимаем на уровень правого потомка.

```
node* remove(node* p, int k, bool AVL) // удаление из
{
    if (!p) return p;
    if (p->key == k)
    {
        node* q;
        if (AVL) q = join_AVL(q: p->left, p->right);
        else q = join(p->left, q: p->right);
        delete p;
        return q;
    }
    else if (k < p->key)
        p->left = remove(p->left, k, AVL);
    else
        p->right = remove(p->right, k, AVL);
    if (AVL) return balance(p); // балансировка при н
    else return p;
}
```

6. Функция объединения деревьев (рандомизированное). Проверяем существование обоих деревьев. С некоторой вероятностью выбираем правое или левое дерево. Фиксируем глубину.

```
node* join(node* p, node* q) // объединение двух д
{
    if (!p) return q;
    if (!q) return p;
    if (rand() % (p->size + q->size) < p->size) {
        p->right = join(p->right, q);
        fixsize(p);
        return p;
    } else {
        q->left = join(p, q->left);
        fixsize(p: q);
        return q;
    }
}
```

7. Функция объединения деревьев (AVL). Проверяем существование правого потомка. Ищем минимальное значение в правом потомке. Удаляем минимальный узел. Вызываем балансировку.

```
node* join_AVL(node* q, node* r) {
    if (!r) return q; // если право
    node* min = findmin(p: r); // и
    min->right = removemin(p: r); /
    min->left = q;
    return balance(p: min);
}
```

8. Функция поиска узла с минимальным значением. Данная функция находит минимальный по значению узел в правом потомке узла.

```
node* findmin(node* p) // поиск узла с мин
{
    return p->left ? findmin(p->left) : p;
}
```

9. Удаление узла с минимальным значением. Данная функция удаляет найденный в ранее описанной функции узел и ставит на его место правый узел. При необходимости вызывается балансировка дерева.

```
node* removemin(node* p) // удален
{
    if (p->left == 0)
        return p->right;
    p->left = removemin(p->left);
    return balance(p);
}
```

10. Балансировка. Изначально мы определяем глубину каждой ветки относительно заданного узла. Далее, если у нас в правом дереве больше потомков, то идет та же проверка для правого узла. Если левый потомок больше, то происходит правый поворот вокруг этого правого узла. После чего, будет левый поворот вокруг изначального узла. Аналогично будет происходить, если у изначального узла левых потомков больше.

```
node* balance(node* p) // балансировка узла p
{
    fixheight(p);
    if (bfactor(p) == 2)
    {
        if (bfactor(p->right) < 0)
            p->right = rotateright(p->right);
        return rotateleft(q: p);
    }
    if (bfactor(p) == -2)
    {
        if (bfactor(p->left) > 0)
            p->left = rotateleft(q: p->left);
        return rotateright(p);
    }
    return p; // балансировка не нужна
}
```

11. Функция для фиксации размера бинарного дерева. Мы определяем глубину по правому и по левому потомкам.

```
void fixsize(node* p) // установление корректного размер
{
    p->size = getsize(p->left) + getsize(p->right) + 1;
}
```

12. Функция определения разницы между глубиной правого и левого потомков.

```
/* Разница между высотой правого и левого потомко
int bfactor(node* p) {
    return getsize(p->right) - getsize(p->left);
}
```

13. Функция определения глубины потомка.

```
int getsize(node* p) {
    if (!p) return 0;
    return p->size;
}
```

14. Правый поворот вокруг узла. Переставляем местами левого и правого потомков. Фиксируем их размер. Возвращаем нового правого потомка.

```
node* rotateright(node* p) {
    node* q = p->left;
    if (!q) return p;
    p->left = q->right;
    q->right = p;
    q->size = p->size;
    fixsize(p);
    return q;
}
```

15. Левый поворот вокруг узла. Переставляем местами левого и правого потомков. Фиксируем их размер. Возвращаем нового левого потомка.

```
node* rotateleft(node* q) {
    node* p = q->right;
    if (!p) return q;
    q->right = p->left;
    p->left = q;
    p->size = q->size;
    fixsize(p: q);
    return p;
}
```

16. Проход по дереву. Проходимся по всему дереву. Когда достигаем конца ветки, записываем текущую глубину ветки в вектор.

```
vector<int> treewalk(node* p, int level, vector<int>& a) {
    if (p) {
        if(p->left == NULL && p->right == NULL)
            a.push_back(_Val: level);
        treewalk(p->right, level + 1, &: a); // правый пото
        treewalk(p->left, level + 1, &: a); // левый потомо
    }
    return a;
}
```

17. Вывод максимальной глубины дерева. Из вектора глубин всех веток выводим самую максимальную.

```
void maxsize(vector<int> a) {
    ofstream out1;
    out1.open(_Filename: "C:\\Users\\evgen\\Desktop\\Алгоритм
    out1 << *max_element(_First: a.begin(), _Last: a.end());
    out1 << endl;
    out1.close();
}
```

18. Вывод глубин всех веток.

```
void printTreeWalk(vector<int> a) {
    ofstream out1;
    out1.open( _Filename: "C:\\Users\\evgen\
    out1 << "\n--------------------\n";
    for (int i = 0; i < a.size(); i++) {
        out1 << a[i] << endl;
    }
    out1 << "\n--------------------\n";
    out1.close();
}
```

19. Освобождение памяти из под дерева. Рекурсивно удаляем каждый узел у дерева.

```
//Освобождение памяти дерева
void freemem(node* tree) {
    if (tree != NULL) {
        freemem(tree->left);
        freemem(tree->right);
        delete tree;
    }
}
```

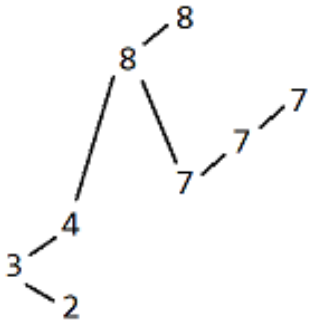
Результаты работы алгоритма

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

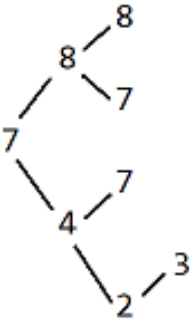
Таким образом, была проведена всего одна серия для одного цикла. Массив заполнялся случайно. Также на каждую операцию было проведено по 5 тестов.

С помощью функции вставки мы создали рандомизированное и AVL бинарное дерево из массива случайных чисел. Далее продемонстрирована операция вставки и удаления:

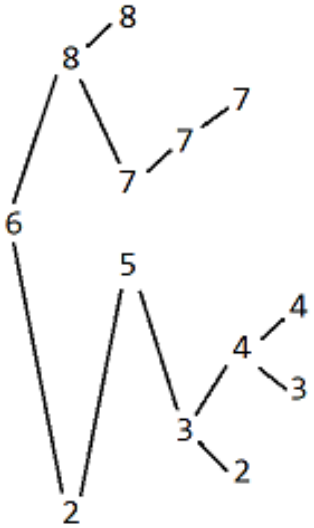
Рандомизированное дерево



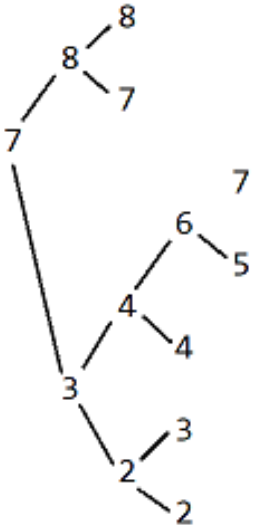
AVL-дерево



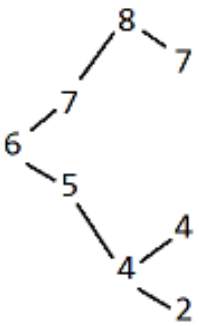
Рандомизированное дерево после вставки



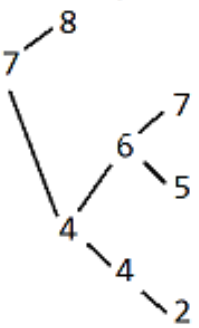
AVL-дерево после вставки



Рандомизированное дерево после удаления



AVL-дерево после удаления

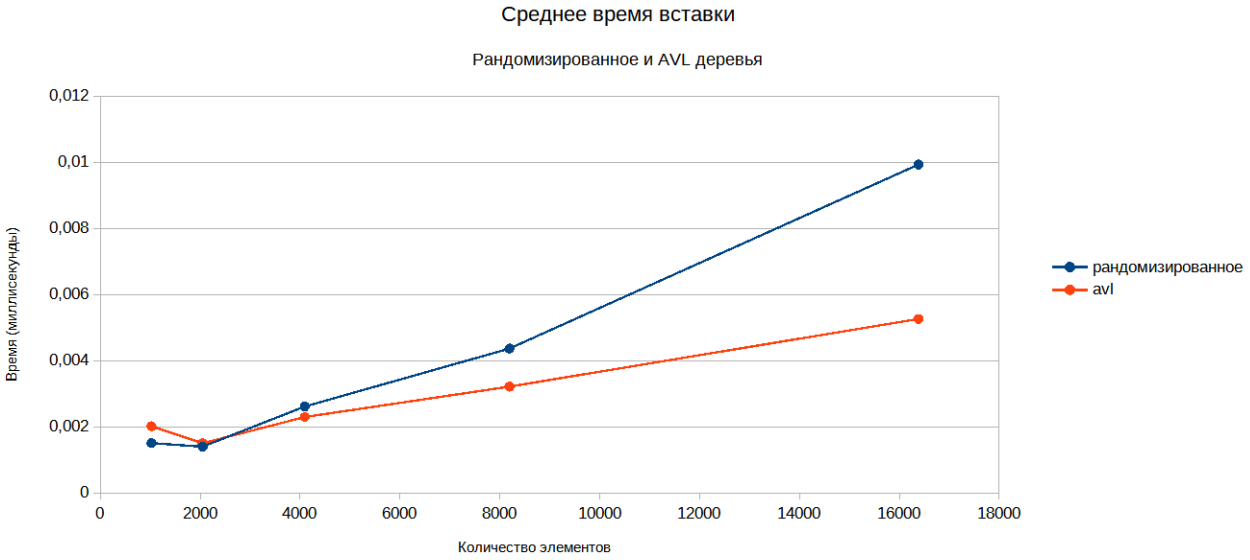


Полученные результаты

Вставка:

	Вставка в рандомизированное дерево				
	1024	2048	4096	8192	16384
1	0,000999	0,001913	0,001835	0,005070	0,008425
2	0,001491	0,001697	0,003436	0,003845	0,013279
3	0,002198	0,001290	0,002295	0,002932	0,009048
4	0,002045	0,001010	0,003418	0,003216	0,010387
5	0,002636	0,001268	0,001720	0,003743	0,008649
6	0,002127	0,001500	0,002337	0,005366	0,011532
7	0,002704	0,001113	0,001502	0,002403	0,010632
8	0,001981	0,000950	0,003789	0,002788	0,004788
9	0,001267	0,000927	0,002695	0,006131	0,006535
10	0,003417	0,001314	0,002141	0,007036	0,014236
11	0,003250	0,001061	0,001749	0,003106	0,021939
12	0,002104	0,001059	0,001438	0,006290	0,018527
13	0,002194	0,001146	0,002125	0,002766	0,013849
14	0,002113	0,001660	0,001330	0,011233	0,008368
15	0,001318	0,001174	0,001372	0,003278	0,008998

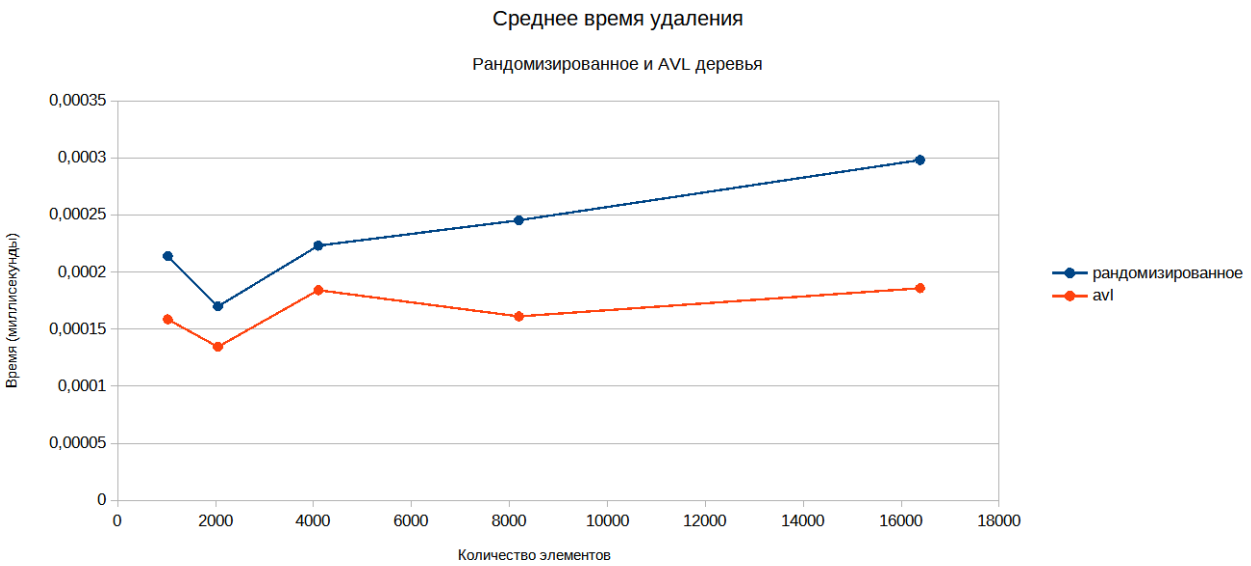
	Вставка в AVL дерево				
	1024	2048	4096	8192	16384
1	0,001200	0,001886	0,001624	0,002088	0,002540
2	0,001452	0,001397	0,003093	0,002421	0,008379
3	0,002342	0,001783	0,002197	0,001951	0,003836
4	0,002714	0,001710	0,002976	0,003190	0,003566
5	0,004011	0,001496	0,001873	0,003402	0,006265
6	0,007683	0,001201	0,002090	0,002079	0,009568
7	0,002908	0,001316	0,001361	0,001566	0,003293
8	0,003158	0,001098	0,002414	0,001929	0,003740
9	0,001938	0,001077	0,002990	0,003653	0,002931
10	0,007797	0,002465	0,002043	0,003973	0,004185
11	0,002742	0,001185	0,002078	0,002046	0,008471
12	0,001874	0,001188	0,001340	0,004641	0,013015
13	0,002496	0,001179	0,002714	0,001974	0,006450
14	0,002719	0,002396	0,001729	0,007003	0,003709
15	0,001622	0,001965	0,001183	0,001710	0,005546



Удаление:

	Удаление из рандомизированного дерева				
	1024	2048	4096	8192	16384
1	0,000172	0,000200	0,000150	0,000213	0,000237
2	0,000173	0,000121	0,000164	0,000183	0,000235
3	0,000255	0,000127	0,000177	0,000149	0,000187
4	0,000251	0,000157	0,000161	0,000189	0,000498
5	0,000172	0,000116	0,000205	0,000171	0,000370
6	0,000402	0,000145	0,000172	0,000145	0,000290
7	0,000269	0,000196	0,000307	0,000192	0,000145
8	0,000248	0,000166	0,000235	0,000241	0,000393
9	0,000287	0,000114	0,000317	0,000346	0,000151
10	0,000247	0,000138	0,000130	0,000333	0,000215
11	0,000327	0,000132	0,000175	0,000406	0,000194
12	0,000189	0,000193	0,000153	0,000184	0,000419
13	0,000392	0,000167	0,000174	0,000196	0,000217
14	0,000199	0,000163	0,000171	0,000220	0,000439
15	0,000265	0,000180	0,000155	0,000216	0,000245

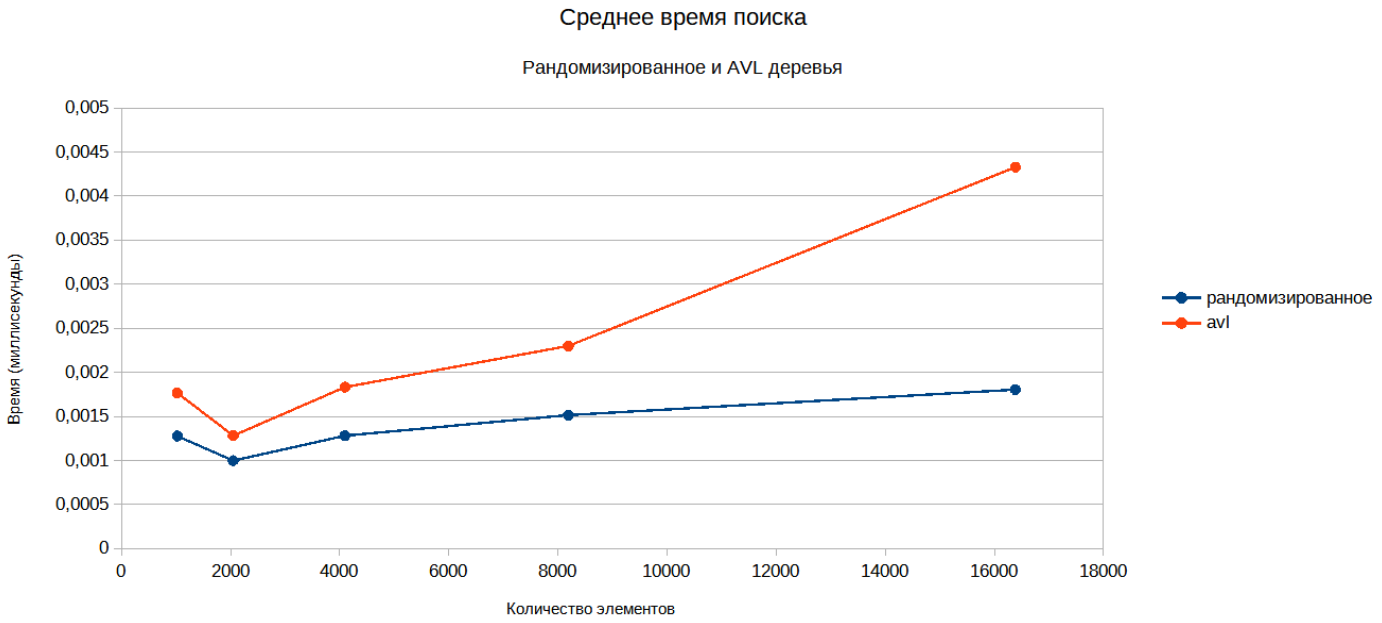
	Удаление из AVL-дерева				
	1024	2048	4096	8192	16384
1	0,000151	0,000137	0,000113	0,000131	0,000178
2	0,000130	0,000087	0,000122	0,000123	0,000144
3	0,000202	0,000094	0,000131	0,000138	0,000119
4	0,000187	0,000116	0,000136	0,000143	0,000153
5	0,000148	0,000106	0,000119	0,000131	0,000279
6	0,000204	0,000134	0,000121	0,000104	0,000160
7	0,000203	0,000116	0,000223	0,000134	0,000106
8	0,000197	0,000105	0,000182	0,000136	0,000242
9	0,000218	0,000109	0,000216	0,000249	0,000102
10	0,000199	0,000093	0,000102	0,000190	0,000147
11	0,000204	0,000105	0,000113	0,000306	0,000138
12	0,000152	0,000129	0,000109	0,000139	0,000191
13	0,000324	0,000109	0,000115	0,000153	0,000163
14	0,000182	0,000099	0,000136	0,000143	0,000243
15	0,000201	0,000118	0,000100	0,000131	0,000176



Поиск:

	Поиск в рандомизированном дереве				
	1024	2048	4096	8192	16384
1	0,001229	0,001088	0,000818	0,001010	0,001274
2	0,001145	0,000676	0,000926	0,000963	0,001058
3	0,001310	0,000849	0,001294	0,001067	0,001039
4	0,003044	0,000737	0,001265	0,001119	0,000919
5	0,001173	0,000765	0,000787	0,001183	0,001787
6	0,002048	0,001052	0,000836	0,000721	0,001977
7	0,001502	0,000863	0,001814	0,001086	0,001087
8	0,002150	0,000888	0,001039	0,001359	0,004627
9	0,002164	0,000848	0,000964	0,001725	0,000947
10	0,002562	0,000734	0,000893	0,000944	0,001547
11	0,001699	0,000882	0,001112	0,001811	0,001505
12	0,001259	0,001773	0,000722	0,001068	0,001555
13	0,001887	0,000808	0,000865	0,001111	0,001924
14	0,001112	0,000752	0,000827	0,001835	0,002150
15	0,001192	0,000687	0,000818	0,001460	0,001330

	Поиск в AVL-дереве				
	1024	2048	4096	8192	16384
1	0,001775	0,001308	0,001194	0,002091	0,004333
2	0,001419	0,000920	0,001284	0,001588	0,008101
3	0,004017	0,000977	0,001406	0,001602	0,007383
4	0,002202	0,001086	0,001231	0,001708	0,002652
5	0,001545	0,000891	0,001198	0,001658	0,005310
6	0,002767	0,001138	0,001158	0,001358	0,008029
7	0,003289	0,001062	0,002958	0,001738	0,002143
8	0,002104	0,001112	0,001434	0,002109	0,004756
9	0,003361	0,000890	0,002111	0,003066	0,002213
10	0,002287	0,000957	0,001134	0,001638	0,003265
11	0,004147	0,001003	0,001157	0,003125	0,002719
12	0,002146	0,001457	0,001225	0,001959	0,004237
13	0,002396	0,001206	0,001331	0,001984	0,002769
14	0,001617	0,001090	0,001359	0,003594	0,009881
15	0,001856	0,001214	0,001115	0,002021	0,006620



Глубина веток в последней серии:

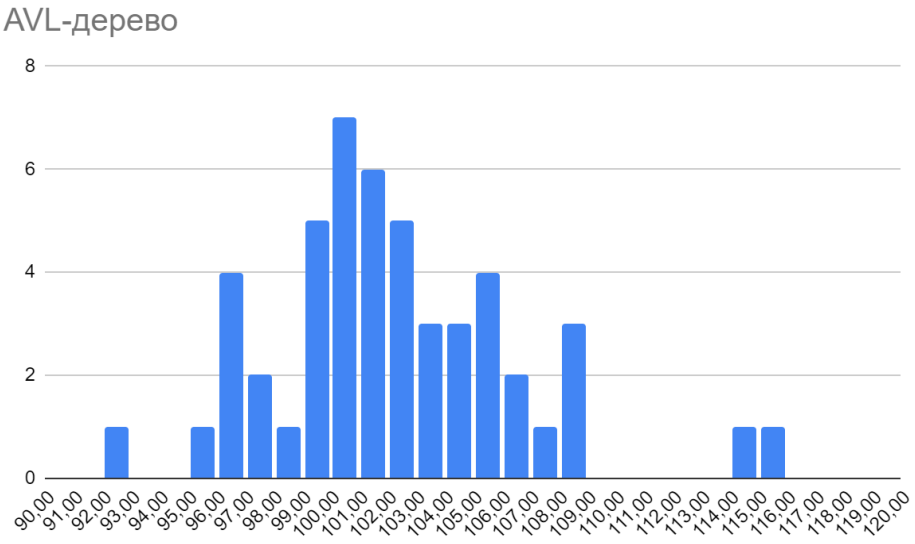
RANDOM	
Число	Кол-во
45	2
47	2
48	4
49	4
50	7
51	6
52	6
53	15
54	25
55	33
56	41
57	53
58	90
59	78
60	117
61	143
62	161
63	167
64	244
65	262
66	315
67	372
68	362

AVL	
Число	Кол-во
4	4
5	28
6	170
7	613
8	1709
9	3837
10	6764
11	10743
12	14852
13	19115
14	22521
15	25382
16	27423
17	28667
18	29471
19	29289
20	28966
21	27280
22	25006
23	21866
24	18508
25	14681
26	11086

Максимальная глубина деревьев в последней серии:

RANDOM	Кол-во
47	3
48	4
49	9
50	7
51	7
52	8
53	6
54	4
56	1
57	1

AVL	Кол-во
92	1
95	1
96	4
97	2
98	1
99	5
100	7
101	6
102	5
103	3
104	3
105	4
106	2
107	1
108	3
114	1
115	1



Заключение.

Из полученных результатов можно заметить, что операции с рандомизированным деревом проходят медленнее, чем с обычным AVL-деревом. Однако поиск по такому дереву все же быстрее. Это связано с тем, что при случайном выборе необходимого значения мы можем попасть в достаточно короткую ветку, в которой мы быстро найдем необходимый узел.

Быстрое выполнение остальных операций связано с тем, что сбалансированное дерево позволяет нам равномерно распределить все узлы, что ускоряет процесс вставки и удаления узлов.

Из гистограмм, полученных из глубин веток можно сделать вывод, что у AVL-дерева пик распределения находится в центре, в то время как у рандомизированного оно смещено в сторону. Это связано с тем, что у AVL-дерева все ветви приблизительно одной глубины. Если смотреть на максимальные значения глубины деревьев, то у AVL-дерева оно намного меньше, все по той же причине. А у рандомизированного дерева с какой-либо вероятностью может получится очень длинная ветка.