ОТЧЁТ О СРАВНЕНИИ БЫСТРОДЕЙСТВИЯ ИСПОЛЬЗУЕМЫХ АБСТРАКТНЫХ ТИПОВ ДАННЫХ: АВЛ TREE, RED-BLACK TREE, SPLAY TREE, TREAP, BINARY HEAP и FIBONACCI HEAP ПРИ ФУНКЦИОНИРОВАНИИ RS1 COARSENING ФУНКЦИИ SETUPPHASE КЛАССИЧЕСКОГО АЛГЕБРАИЧЕСКОГО МНОГОСЕТОЧНОГО МЕТОДА.

К. А. Иванов

kirill7785@mail.ru , г. Москва

Оглавление

Введение

Для нахождения температурных полей в модулях усилителей мощности в данном отчёте используется конечно-разностный метод контрольного объёма. Его применение приводит к необходимости решать большие разреженные системы линейных алгебраических уравнений. В данном отчёте

для этого используется алгебраический многосеточный метод [1],[2]. Работа алгебраического многосеточного метода содержит процедуру автоматизированного построения иерархии грубых сеток – однопроходовое Руге-Стубен огрубление сетки (RS1 coarsening). Для работы функции RS1 coarsening используется абстрактный тип данных (АТД) очереди по приоритетам (см. ПРИЛОЖЕНИЕ А). Очередь по приоритетам это АТД, поддерживающий эффективное выполнение четырёх операций:

1. извлечение элемента с максимальным значением ключа, 2. добавление нового элемента, если его ещё не существовало или модификация существующего в противном случае, 3. удаление элемента с заданным ключом. 4. найти элемент по ключу или сказать, что его нет.	(1)

Операции 2 и 3 используют внутри себя операцию поиска 4 поэтому 4 выделена, как самостоятельная операция.

Данный функционал может иметь разные алгоритмические воплощения. В данном отчёте мы рассматриваем шесть реализаций:

1. на основе АВЛ дерева (наследия шахматного кода КАИССА, 1962 г.)
2. на основе скошенного дерева (Андре Тарьян и Даниел Слитор, 1985г.)
3. на основе двоичной кучи (вероятно, возникла при изобретении пирамидальной сортировки массива, Дж. Уильямс 1964г.).
4. на основе дерамиды (дерево в сочетании с пирамидой, Treap). Изобретены Сиделем (Siedel) и Арагоном (Aragon) в 1996г. Дерамида положена в основу высокоэффективной библиотеки LEDA.
5. на основе Красно-Чёрного дерева (Байер, Гибас, Седжвик 1978г.).
6. на основе Фибоначчиевой кучи (Майкл Фридман и Роберт Тарьян 1984г.).

Порядок, в котором перечислены данные реализации АТД, соответствует порядку их внедрения в программу регистрационный № 2013660267 [1],[2]. Они перечислены в порядке простоты их освоения. АВЛ реализованы с использованием [3],[4], (см. ПРИЛОЖЕНИЕ Б). Скошенные (SPLAY) деревья реализованы согласно [5],[6] (см. ПРИЛОЖЕНИЕ В). Куча-наиболее подходящая структура данных для RS1 алгоритма, реализована согласно [6],[7] (см. ПРИЛОЖЕНИЕ Г - двоичная куча). В перспективе авторы планируют внедрить (2017г.) в [1],[2] Фибоначчиеву кучу [7],[12],[13](встроена в 2018г. См. ПРИЛОЖЕНИЕ Ж) через освоение биномиальной [7]. Деамида[9],[10] – это структура данных взявшая всё лучшее и от сбалансированного дерева и от двоичной кучи. В её основе

лежит факт, что отношение порядка, используемое при поиске ключа в дереве организовано независимо от специального поля, хранящего случайное число, которое используется для поддержания сбалансированности двоичного дерева. Для балансировки по-прежнему используются повороты. (см. ПРИЛОЖЕНИЕ Д).

Выбор правильной структуры данных может повлиять на производительность кода RS1 функции. На удивление было мало эмпирических исследований взаимосвязи между алгоритмами, используемыми для управления функционалом очереди по приоритетам, и характеристиками производительности алгебраического многосеточного метода на реальных задачах. В этой статье предпринята попытка восполнить этот пробел и тщательно проанализировать производительность шести разных алгоритмических воплощений очереди по приоритетам в условиях реальных нагрузок классического алгебраического многосеточного метода.

В таблице 2 представлен ряд вычислительных экспериментов по расчёту температурного поля с использованием программы [1],[2]. В каждом случае нагрузки на очередь по приоритетам взяты из реального мира ситуаций и некоторые нагрузки отражают как наиболее благоприятные, так и наиболее тяжелые случаи, по критерию быстродействие, расположения входных данных для каждой из рассмотренных нами реализаций АТД.

Прежде чем приводить результаты реальных замеров производительности мы опишем теоретические сведения о гарантированной производительности реализованных структур данных. Дадим словесное толкование, почему одна структура данных эффективнее другой и при каких гипотезах о распределении входных данных это справедливо [8]. Обоснуем, почему необходимы эмпирические замеры быстродействия каждой реализованной структуры данных. И наконец, приведем результаты реальных замеров времени расчёта.

Теоретические гарантии производительности

Теоретическая стоимость выполнения операций для четырёх из шести реализаций представлена в таблице 1.

Таблица 1.

	АВЛ дерево	Скошенное дерево	Двоичная куча на массиве	Фибонач-чиева куча
Извлечение максимума	O(log(n))	Амортизационно O(log(n))	O(log(n))	O(log(n))
Вставка	O(log(n))	Амортизационно O(log(n))	O(log(n))	O(1)
Удаление	O(log(n))	Амортизационно O(log(n))	O(log(n))	O(log(n))
Балансировка	ДА	ДА	НЕТ	ДА при операции извлечение максимума
Накладные расходы	Простой и большой поворот вправо (влево)	Zig, zig-zig, zig-zag	fixUp, fixDown или (HeapifyUp, HeapifyDown)	при операции извлечение максимума
Частые malloc/free	ДА	ДА	НЕТ	ДА
	рекурсия	рекурсия	итерация	
	ПРИЛО-ЖЕНИЕ Б	ПРИЛОЖЕНИЕ В	ПРИЛОЖЕ-НИЕ Г	ПРИЛОЖЕНИЕ Ж

Здесь n количество элементов. Реальное быстродействие определяет пока неизвестная константа перед O – большим. Для вычисления данной константы требуются замеры производительности на реальных данных.

Гипотеза производительности

В статье сравнивается шесть вариантов реализации очереди по приоритетам, перечисленные выше (см. таблица 1). Априорно каждая реализация АТД наиболее предпочтительна при определённой гипотезе о структуре входных данных. Сначала рассмотрим сбалансированные деревья поиска: АВЛ, Красно-Черное дерево и скошенное (SPLAY). АВЛ дерево наиболее предпочтительно, когда входные данные поступают с ключами в отсортированном порядке. АВЛ дерево имеет более строгое правило балансировки чем, скажем красно-чёрное дерево (RB tree), поэтому его высота меньше ($1.44*\log_2 n$ у АВЛ против $2*\log_2 n$ у RB), оно лучше сбалансировано и лучше пригодно по быстродействию, чем RB дерево в случае патологического ввода ключей в порядке сортировки. Также вставка в

АВЛ дерево не требует более одной ротации, но удаление может потребовать до $\log_2 n$ ротаций. С другой стороны, вставка в RB дерево может потребовать до двух ротаций, но удаление требует не более трёх. Скошенные (SPLAY) деревья при каждом запросе, например, поиск по ключу, передвигают запрашиваемый элемент в корень дерева, что сильно ускоряет последующие обращения к этому элементу и близким к нему в смысле значения ключа. Однако, теоретический анализ эффективности SPLAY деревьев достаточно сложен и требует привлечения аппарата амортизационного анализа. На практике известны случаи отличной производительности SPLAY деревьев [8]. Гарантировать хорошую производительность скошенных деревьев можно только в амортизационном смысле. Известна самая лучшая теоретически достижимая-линейная производительность для некоторых шаблонов последовательных запросов в скошенных деревьях, что означает, что скошенные деревья могут быть очень эффективны, но это сложно предсказать теоретически, как их производительность соотносится с производительностью АВЛ или красно-чёрных деревьев на реальных данных.

Помимо АВЛ, Красно-Чёрных и SPLAY деревьев существуют и другие разновидности сбалансированных деревьев: рандомизированные, Деамиды и др. Рандомизация (её принудительное введение в структуру дерева) призвана гарантировать, что в среднем получающиеся деревья будут неплохо сбалансированы даже при патологическом вводе ключей в порядке сортировки, а случай сильно несбалансированного дерева крайне маловероятен. [6] Структура сбалансированного двоичного дерева (BBST) предоставляет гораздо больший функционал, чем нам требуется. Например, BBST хранящие строковые ключи (фамилии персон) позволяют эффективно вывести на печать фамилии всех персон фамилии которых начинаются, например, с буквы П – по букву Т, если для этого в дереве поддерживаются ссылки на родительский узел. Ни хеш таблица ни куча на это не способны. Кроме того BBST имеют существенные накладные расходы для того чтобы поддерживать дерево сбалансированным : простой и большой поворот в АВЛ деревьях [3], повороты для балансировки Красно-Черных деревьев [7], операции zig, zig-zig, zig-zag в SPLAY деревьях [5]. Дерево двоичной кучи (Binary Heap) не требуется поддерживать сбалансированным по самому определению двоичной кучи, т.к. двоичная куча на массиве это по определению полное двоичное дерево у которого имеются вакансии для ставки лишь на самом последнем уровне. Для двоичной кучи требуется поддерживать лишь порядок кучи за O(log(n)) время с помощью операций

fixUp и fixDown [6]. Также двоичная куча в отличие от BBST не содержит частых malloc/free над небольшими фрагментами ОЗУ, т.к. динамическая память под двоичную кучу выделяется лишь единожды при вызове конструктора. Это создаёт предпосылки считать, что двоичная куча должна быть более быстрой, чем BBST при больших размерах входных данных.

Однако с кучей связана проблема поиска элемента по ключу и как следствие удаление элемента заданным значением ключа. Это справедливо и для сливаемых куч. Справиться с данной проблемой позволяет объединение кучи и хеш таблицы. Суть состоит в том, что поиск элемента в куче по ключу осуществляется через хеш таблицу преобразующую значение ключа в ссылку на соответствующий узел сливаемой кучи или в индекс массива при реализации кучи на массиве (см. приложение Г). Однако на поддержание такой хеш таблицы при операциях fixUp, fixDown в двоичной куче на массиве тратится быстродействие, поэтому теоретический анализ утверждения что двоичная куча быстрее BBST дерева представляется затруднительным и спасают лишь эмпирические замеры производительности, чему и посвящён данный отчёт.

В данном отчёте также представлена Фибоначчиева куча. Асимптотически она лучше, чем двоичная куча на массиве т.к. операция вставки делается за O(1). Особенность нашего алгоритма такова, что довольно значительная последовательность операций вставки чередуется небольшим числом удалений, при которых как раз и вызывается балансировка (окучивание). Таким образом, мы имеем очень быструю вставку и не даем куче сильно разбалансироваться – иметь сильно вытянутый корневой двусвязный список с малым числом дочерних деревьев. Всё это говорит о том, что Фибоначчиева куча может быть очень быстрой, так как размерности входных данных в многосеточном методе имеют большой размер и хорошая асимптотика Фибоначчиевой кучи должна сказаться.

Фибоначчиева куча исповедует принцип не откладывай на завтра то, что можно отложить на послезавтра. Т.е. она делает балансировку только в последний момент, только когда это действительно требуется и из-за этого нет избыточности операций - отсюда скорость

Оборудование и программное обеспечение

Замеры быстродействия реализованных для данной статьи алгоритмов произведены на оборудовании со следующими комплектующими:

1. процессор: intel core i7 6850K, материнская плата: msi goodlike gaming carbon x99 chipset, ОЗУ: 128Gb 2133mhz, жёсткий диск: Samsung 960EVO M.2 NVMe SSD 1Tb.
2. ASUS ultrabook: процессор 2.2ГГц, ОЗУ : 12Гб ddr3 sodimm памяти 1233мгц. Жёсткий диск: ssd Samsung 500Gb.
3. процессор core i7 3930K, 3.2ГГц, ОЗУ: 64Гб ddr3 1600мгц, Жёсткий диск: 1tb ssd samsung.

4. серверный процессор: XEON, ОЗУ: 256гб.

Замеры времени производились в следующем программном окружении: ОС (1. и 4. Win10), (2. Win 8.1), (3. Win 7) x64, visual studio community edition 2017, режим компиляции Realese x64 без OpenMP. Модель целочисленной арифметики int64_t – 64 битное целое, double – для вещественной арифметики.

Настройка алгебраического многосеточного метода для решения уравнения конвекции -диффузии имела вид, представленный на рис. 1:

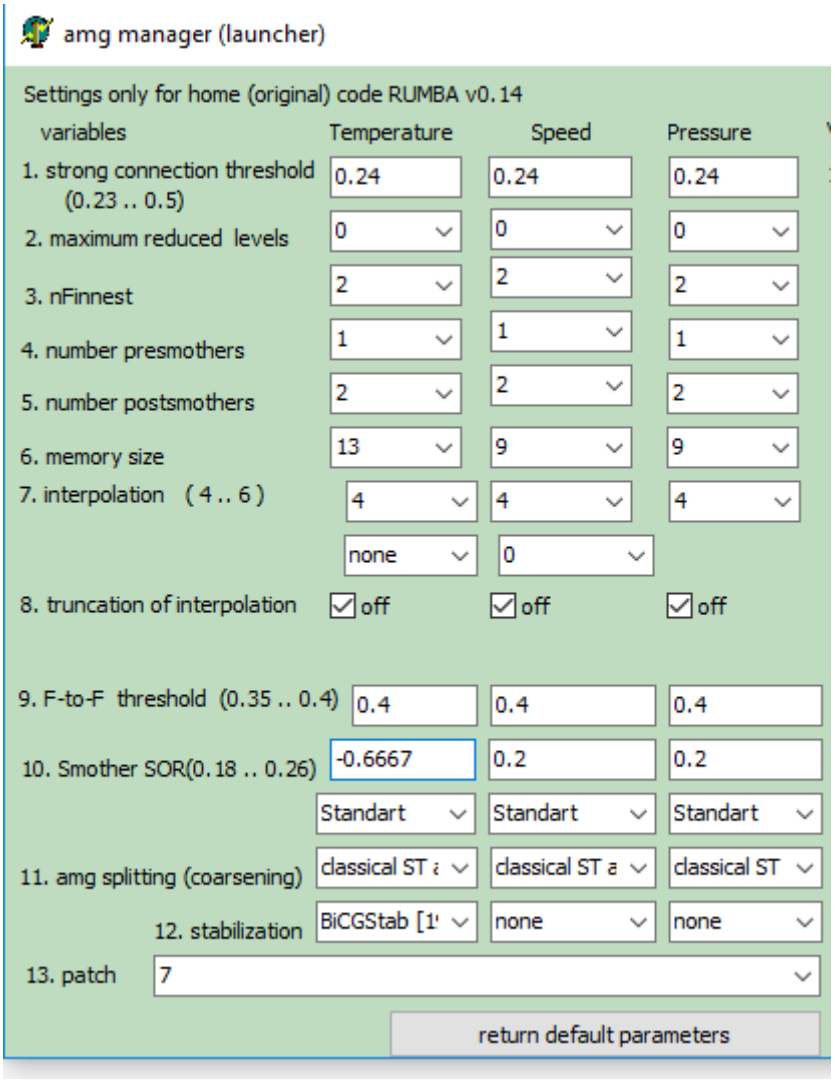


Рис. 1.а. Настройки реализованного и используемого алгебраического многосеточного метода при нахождении поля температур через решение уравнения конвекции-диффузии.

Для решения уравнения теплопередачи в только в твёрдом теле нижняя релаксация в сглаживателе была заменена на верхнюю.

Результаты

В таблице 2 представлено время нахождения поля температур с помощью реализованного алгебраического многосеточного метода [1],[2] внутри которого лежит та или иная алгоритмическая реализация АТД очереди с приоритетом.

Таблица 2.

Задача	Число незвестных, млн (M)	Потребление ОЗУ, Гб	Используемый алгоритм/код				
			АВЛ [RED – BLACK Tree]	SPLAY	BINARY HEAP и хеш таблица	TREAP	Фибоначчиева куча и хеш таблица
ПИОНЕР x1[1]	2.1	4.6	2min 29s	2min 32s	2min 40s	2min 33s	2min 26s
ПИОНЕР x1[2]	2.1	4.6	9min 17s	7min 22s	7min 15s		
ПИОНЕР x1[3]	2.1	4.6	2min 30s	2min 28s	2min 31s		
ПИОНЕР x1[1A]	2.1	4.6	2min 12s	2min 10s	1min 45s		
ПИОНЕР x8[1]	17.6	38.1	89min 56s	86min 48s	93min 36s	88min 10s	
ПИОНЕР x8[1A]						58min 19s	
ПИОНЕР x8[3]	17.6	38.1	97min 8s	94min 21s	101min 25s		
ПИОНЕР x24[4B]	51.5	114.5	386min 42s	378min 7s	414min 0s		
МОДУЛЬ O[1]	10.7	23.5	21min 28s	21min 4s	20min 45s		
МОДУЛЬ O cfd[1C]	0.022	0.1	19min 6s 19min 22s 17min 39s	20min 8s 17min 14s 19min 44s	18min 40s 19min 17s 19min 14s		
КРИСТАЛЛ-C-24[1]	3.9	10.3	14min 40s 14min 31s	14min 24s 14min 15s	13min 59s 13min 57s		
КРИСТАЛЛ-C - 24[1D]	34.5	212.2	127min 33s	127min 24s	136min 50s		
12mm hfet[2]	3.7	6	9min 34s	9min 6s	9min 1s		
12mm heft[1]	3.7	6	7min 14s	6min 37s	6min 39s		
12mm hfet[1]	25.7	39.5	31min 37s	29min 15s	30min 42s	31min 37s	28min 28s
12mm hfet[1]	2.1	3	2min 11s [2min 8s]	2min 1s	2min 1s	2min 8s	1min 58s
Четвертинка detailed	19.4	36.7	35min 30s [35min 5s]	33min 31s	31min 45s	35min 31s	32min 50s

[A)] SOLUTION PHASE распараллелена с помощью ***OpenMP*** на 8 потоках исполнения центрального процессора.

[B)] Осуществлен выход за пределы 32 битного типа int в 2147M значений. Для хранения иерархии матриц используется массив длиной более чем 2246 млн ячеек. Это стало возможно после перехода в программе [1] на 64 битный тип int64_t. Операторная сложность =6.85.

[C)] В данной задаче находятся гидродинамические характеристики радиатора скорость-давление с помощью **SIMPLE [1972]** алгоритма. Эта задача характеризуется относительно невысоким числом уравнений в матрицах и огромным количеством разнообразных матриц подаваемых на вход многосеточному методу – порядка 5К матриц каждая размером около 22.6К уравнений. Этот пример важен, потому что показывает истинно ожидаемое время расчёта каждой реализации АТД на небольших данных, т.к. число матриц велико, а также они достаточно разнообразны, следовательно, можно говорить о достоверности среднего результата.

[D)] Осуществлён выход за пределы 32 разрядного типа int. Число ячеек в массиве для хранения иерархии матриц превышает 2424M индексов при

максимальном размере типа int 2147M. Операторная сложность Acomp=10.24;

При использовании SPLAY дерева увеличение быстродействия решения задачи теплопередачи по сравнению с АВЛ деревом в нашем АТД представлено на графике Рис. 1.б.

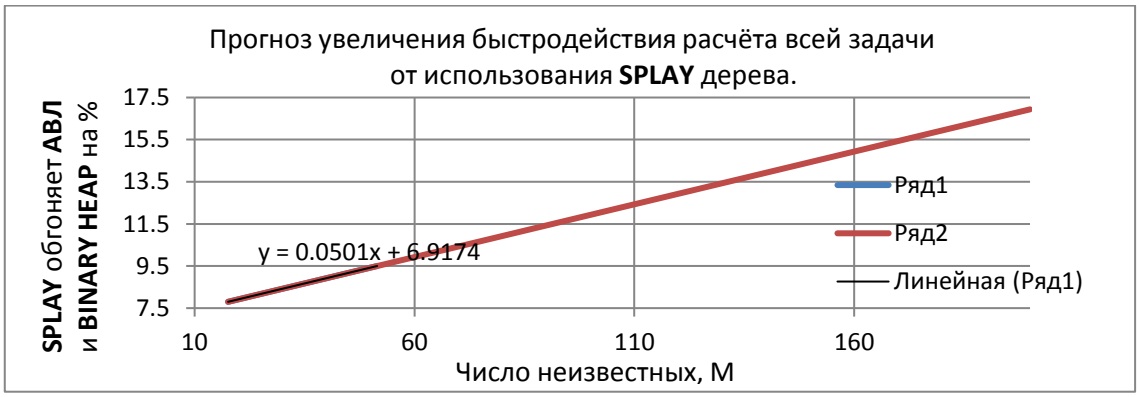
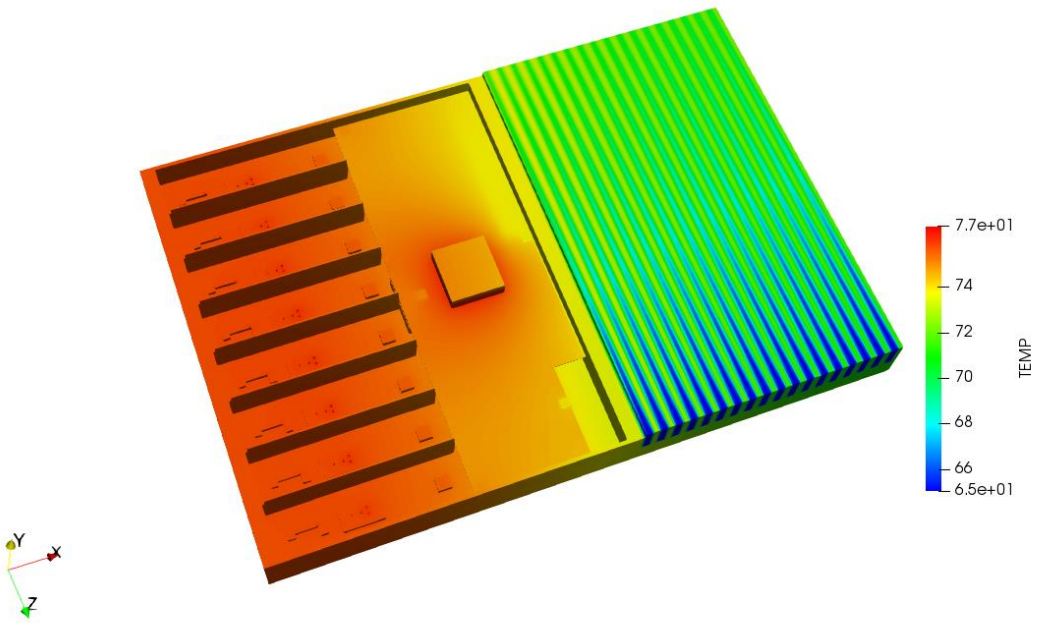
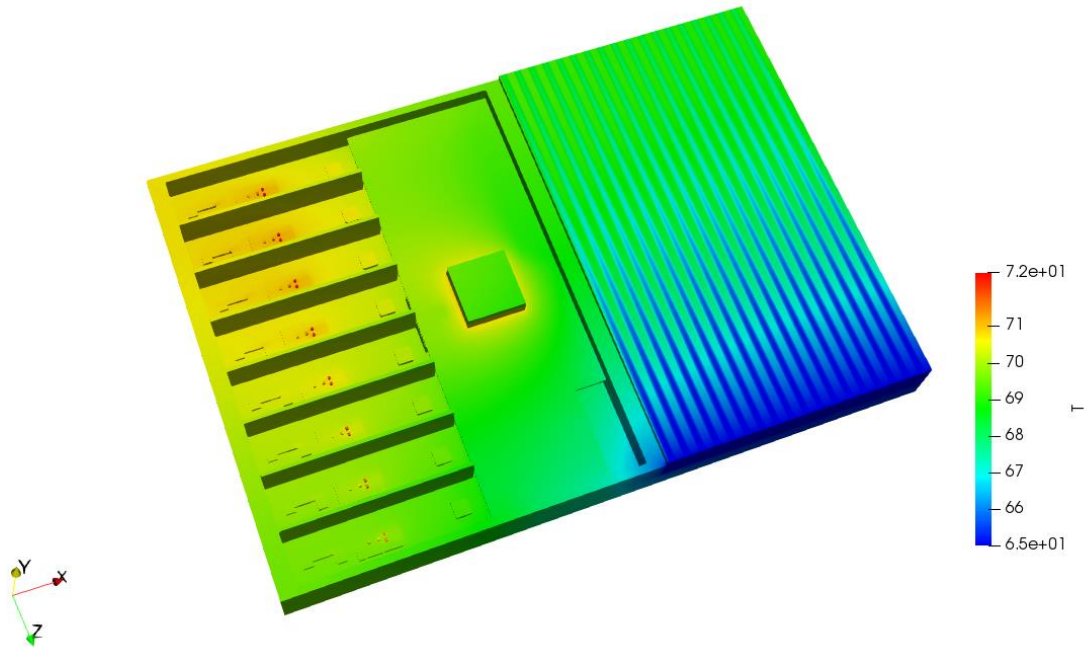


Рис. 1.б. Теоретическое увеличение быстродействия от использования SPLAY дерева в АТД по отношению к использованию АВЛ дерева в АТД.



2.а поузловая сборка матрицы.



2.б. поячеечная сборка матрицы.

Рис. 2. ПИОНЕР x1 потребление ОЗУ 4.6гб. Построено 28 уровней иерархии сетки. 76.63^{O}C. Скорость между рёбрами 2м/с. Число неизвестных 2.1М.

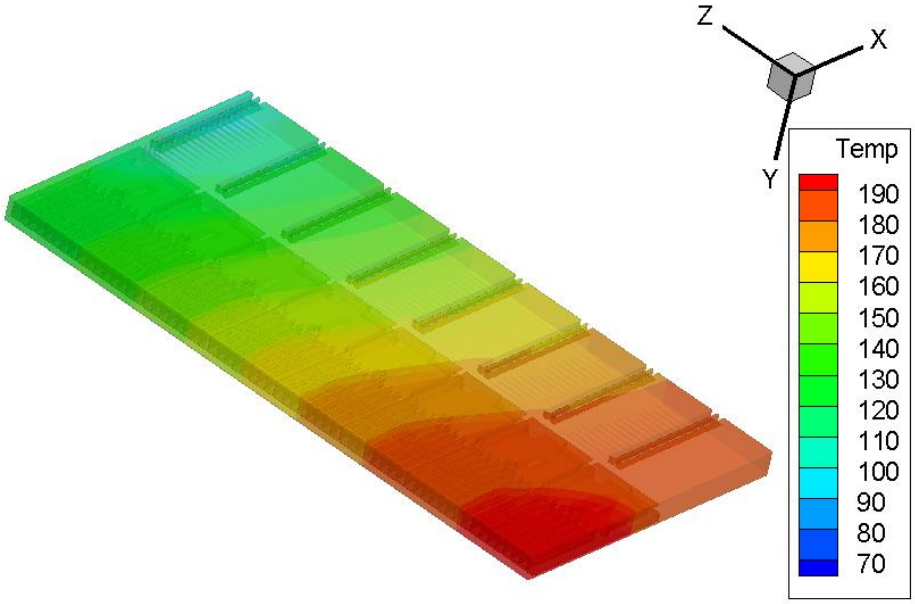
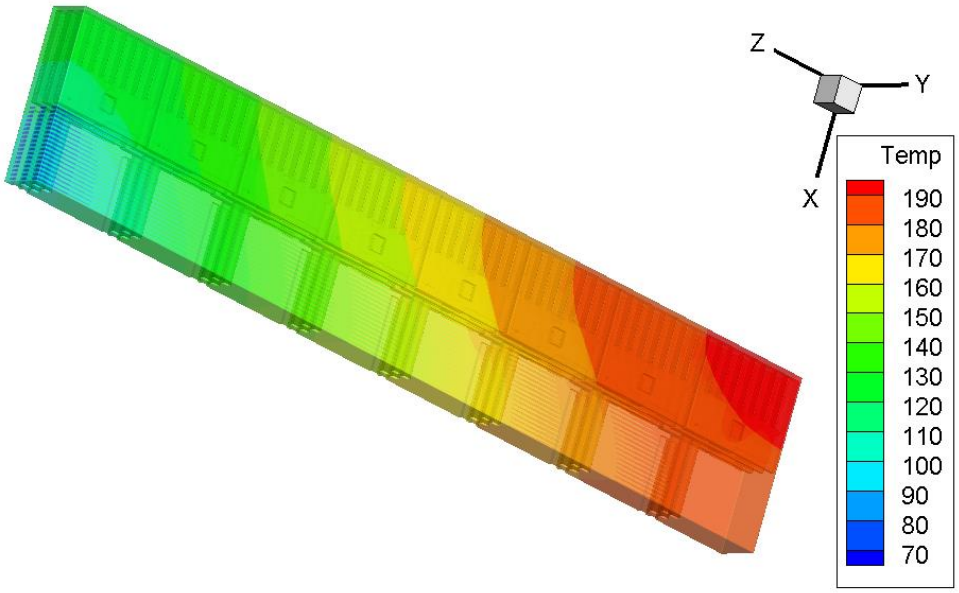
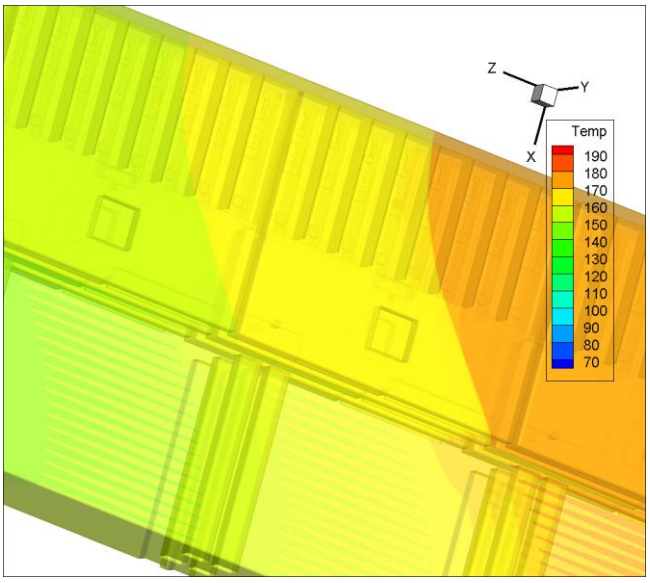


Рис. 3. ПИОНЕР x8 потребление ОЗУ 38.1гб. Построено 33 уровней иерархии сетки. 195.34^{O}C. Скорость между рёбрами 2м/с. Число неизвестных 17.6М.



а). Целиком 24 модуля.



б). Фрагмент.

Рис. 4. ПИОНЕР x3*8 потребление ОЗУ 114.5гб. Построено 36 уровней иерархии сетки. 195.38^{O}C. Operator complexity=6.85. Файл с температурными полями и их градиентами весит 25Гб на жестком диске. Скорость между рёбрами 2м/с.

Построенные в автоматическом режиме иерархии уровней расчётной сетки для задачи «ПИОНЕР» представлены в таблице 3. В таблице 3 можно проследить сеточное отличие задач «ПИОНЕР» с одним, восемью и 24 модулями.

Таблица 3.

	ПИОНЕР 1 модуль			ПИОНЕР 8 модулей			ПИОНЕР 24 модуля		
	n	N_{nz}	N_{nz}/n	n	N_{nz}	N_{nz}/n	n	N_{nz}	N_{nz}/n
1	2109304	13048636	6	17607236	108982680	6	51492076	329193346	6
2	1054652	13216980	12	8803618	110767270	12	25746048	334544838	12
3	401315	15050165	37	3327405	125894388	37	9644720	384610317	39
4	172875	13532213	78	1422170	114743204	80	4078010	369523749	90
5	97679	7282665	74	800795	61117777	76	2290631	243568414	106
6	66650	4495352	67	552741	38112979	68	1564967	165356327	105
7	49748	3510912	70	409479	32882161	80	1151874	146894958	127
8	37902	1676302	44	309947	21659369	69	864040	107500342	124
9	31299	1156627	36	247333	15245743	61	674716	67317380	99
10	25263	504149	19	204107	9828875	48	561490	39520442	70
11	20889	403193	19	169165	5376915	31	459281	18576781	40
12	18289	353745	19	146861	4256045	28	391625	14535187	37
13	14223	276995	19	112195	3588627	31	331553	11926495	35
14	10525	169247	16	81477	2992443	36	226568	9271302	40
15	7372	87458	11	56559	2097867	37	144103	6154465	42
16	4982	46010	9	37053	1106423	29	82602	2891318	35
17	3341	28105	8	23643	542267	22	44904	1169184	26
18	2265	17705	7	15163	277993	18	25023	511369	20
19	1544	11442	7	9980	160750	16	14782	246346	16
20	1054	7226	6	6713	93529	13	9208	134114	14
21	717	4535	6	4539	55761	12	5894	76520	12
22	477	2745	5	3119	39605	11	3842	45294	11
23	329	1795	5	2155	22031	10	2555	28245	11
24	236	1230	5	1508	14372	9	1742	16942	9
25	173	877	5	1076	9642	8	1224	11272	9
26	131	647	4	766	6546	8	863	7809	9
27	97	465	4	543	4775	8	614	5232	8
28	68	318	4	383	3093	8	436	3278	7
29	44	186	4	257	1929	7	319	2327	7
30				163	1155	7	238	1734	7
31				98	670	6	173	1153	6
32				64	408	6	124	822	6
33				41	239	5	90	570	6
34							66	380	5
35							51	295	5
36							39	217	5

Здесь n – число уравнений (неизвестных), N_{nz} - число ненулевых элементов в матрице, N_{nz}/n – средний размер шаблона дискретизации на данном уровне.

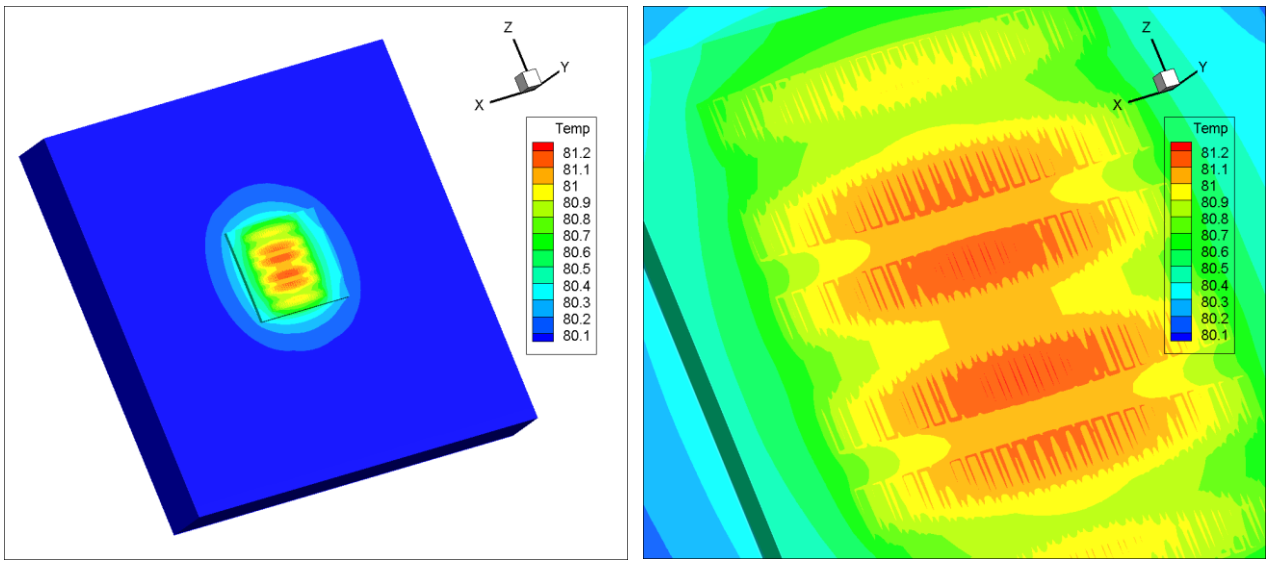


Рис. 5. Температурное поле на поверхности модели КРИСТАЛЛ-С в МД40 корпусе при токе стока = 15А. Используется finnest сетка в 34.5М неизвестных. Pdiss=1.23Вт. Тепловое сопротивление θ=1 ^{O}C/Вт.

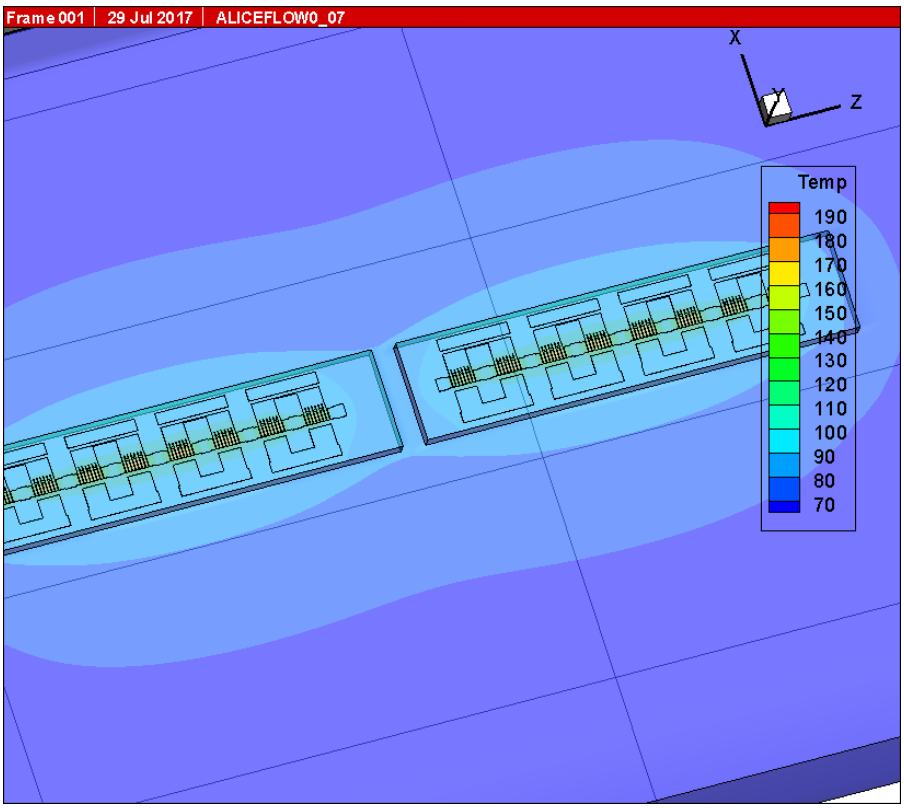
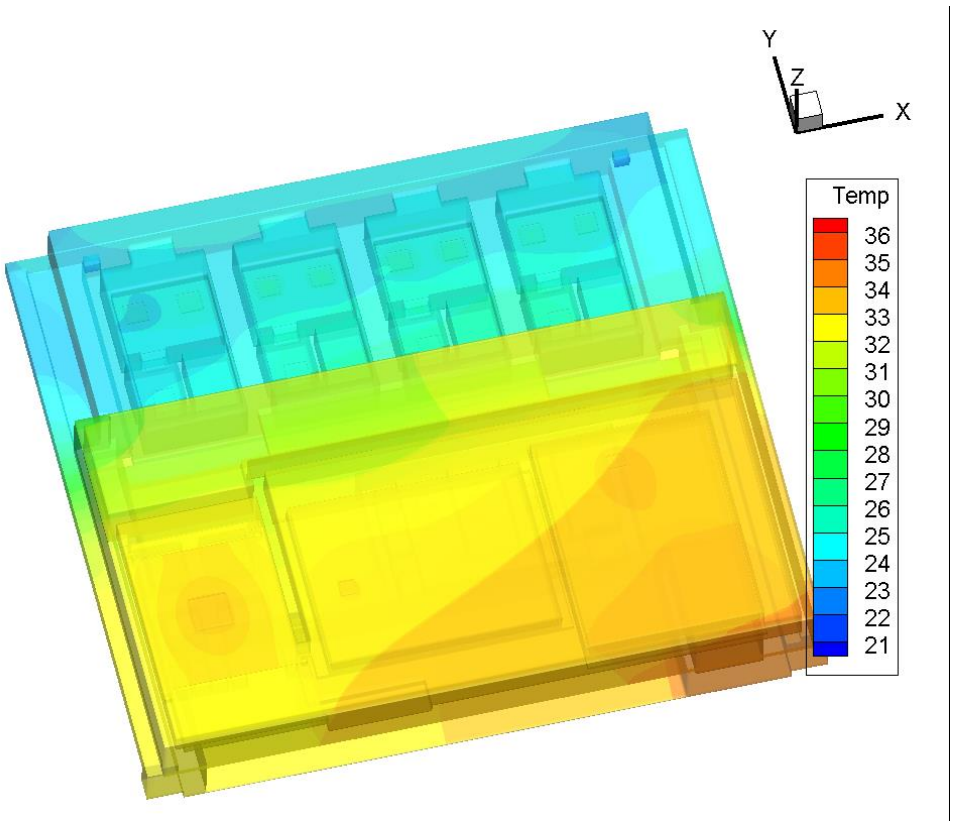
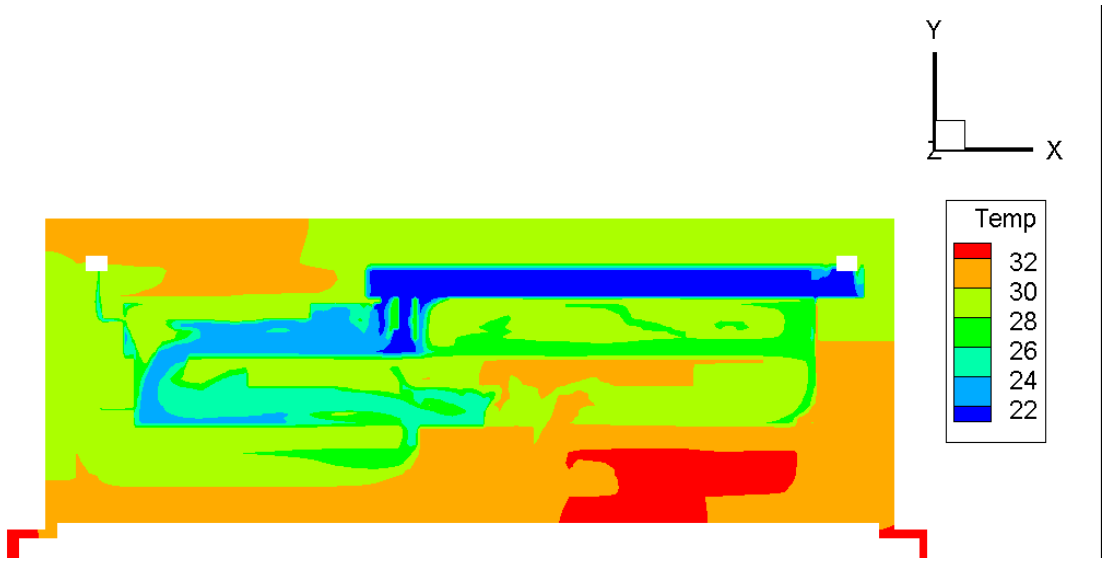


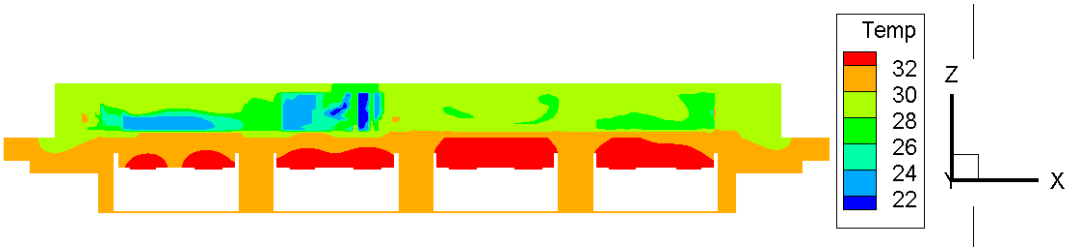
Рис. 6. Температурное поле на поверхности модели 12mm heft. Алгебраический многосеточный метод построил 12 уровней иерархии сеток.



a) Модуль-О целиком.



b) Температура теплоносителя в радиаторе.



c) В сечении под транзисторами выходного каскада.

Рис. 7. Температурное поле на поверхности модуля - О при расходе теплоносителя через радиатор 3л/мин.

Выводы

***пункты 1-3 выводов получены в 2017 году без участия Фибоначчиевой кучи**.

1. При числе неизвестных в матрице СЛАУ менее 3М отличий в скорости работы различных реализаций очереди по приоритетам проследить не представляется возможным.
2. При числе неизвестных в матрице СЛАУ от 3М до 15М выигрывает по скорости BINARY HEAP. Вторым по быстродействию идёт SPLAY дерево. Наиболее медленной является реализация на АВЛ дереве. Медленнее на 3.4%.
3. При числе неизвестных в матрице СЛАУ более 15М SPLAY дерево обгоняет АВЛ и BINARY HEAP. Причём BINARY HEAP становится самой медленной среди рассмотренных реализаций.
4. Одна рассчитанная точка для 25М неизвестных подтверждает, что Фибоначчиева куча даже лучше (быстрее) чем SPLAY дерево при числе неизвестных более 15М.
5. В таблице 2 приведены времена расчёта задачи теплопередачи, реализованной программой [1],[2]. Время, которое тратит алгоритм на собственно саму очередь по приоритетам, относительно невелико, поэтому отличие в скорости исполнения всего процесса решения равное 7.4-7.8% из-за использования той или иной реализации очереди по приоритетам является не таким малым при числе неизвестных более 15М на задачу.
6. В приложениях Б, В, Г, Д, Е и Ж приведены адаптированные к использованию в [1],[2] реализации АТД на языке С работоспособность которых проверена авторами данного отчёта.

ПРИЛОЖЕНИЕ А

RS1 функция алгебраического многосеточного метода. Ход работы RS1 функции проиллюстрирован на рис. 8 на примере 2D задач при переходе с finnest сетки на следующий уровень грубости.

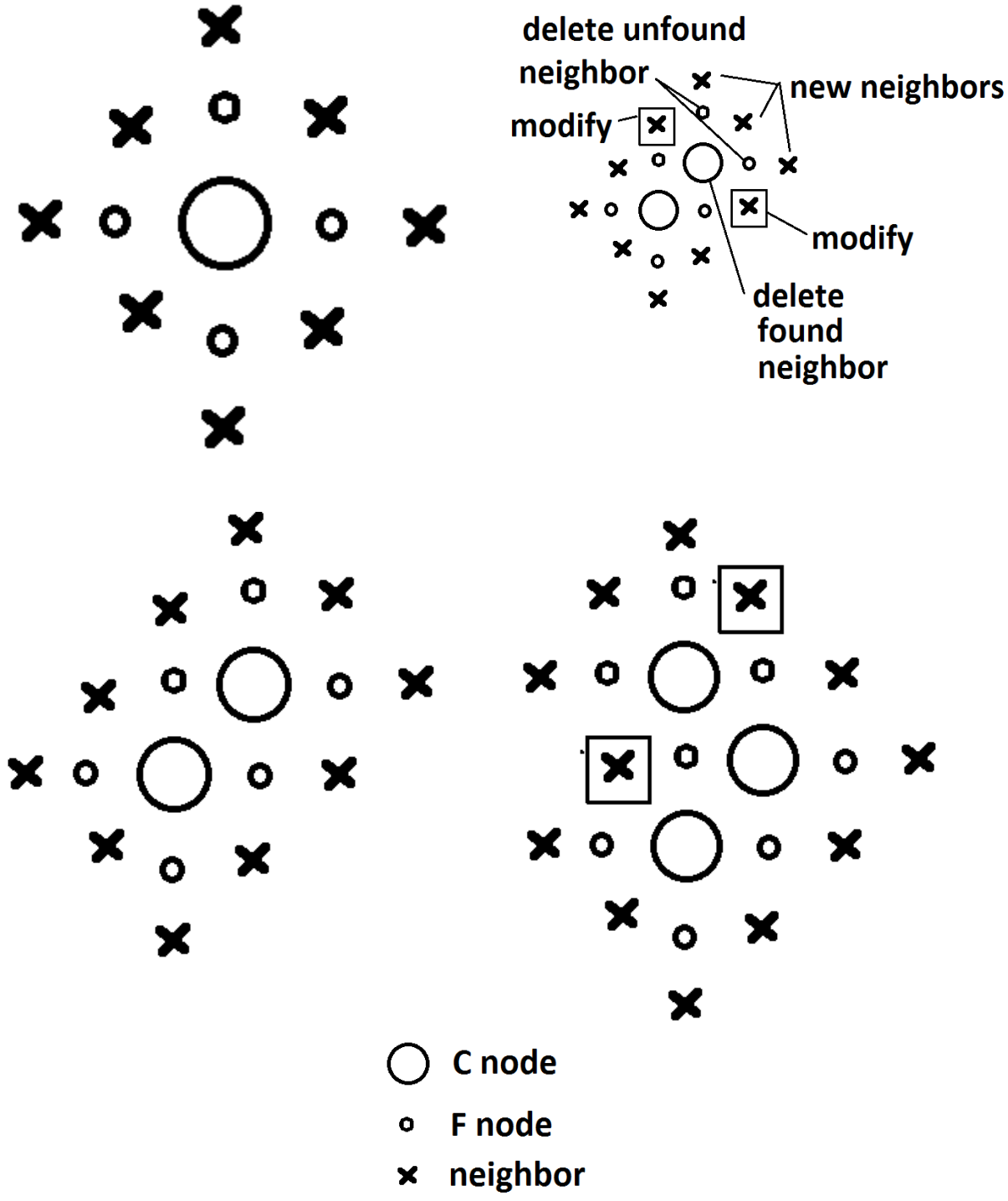


Рис. 8. Формирование одного из грубосеточных уровней вложенности.

Алгоритм используемой RS1 функции на языке теории множеств представлен на рис. 9.

Пусть $N_i=\{j \in \Omega^k: j \neq i, \alpha_{ij} \neq 0\}$.
$S_i=\{j \in N_i : \lvert\alpha_{ij}\rvert \geq 0.25 \max\lvert\alpha_{ik}\rvert \}$.
AMG cl_agl_amg_v0_14 coarsening:
1. [initialization] F=Ø, C=Ø, U=Ω.
$\lambda_i=\lvert S_i \cap \Omega\rvert$.
2. [update weight coefficients]
λ_i += $\lvert N_i \cap$ new F$\rvert$.
3. if λ_i==0 then stop. Else go to step 4.
4. pick i∈U with max λ_i : C=C U {i},
U:=U\{i}.
5. for all j ∈ N_i∩U: F=FU{j}, U=U\{j}.
Goto update step 2.
A.3. Алгоритм огрубления сетки из РУМБА0.14. см. [1].
	Пусть $N_i=\{j \in \Omega^k: j \neq i, \alpha_{ij} \neq 0\}$.
	$S_i=\{j \in N_i : \lvert\alpha_{ij}\rvert \geq 0.25 \max\lvert\alpha_{ik}\rvert$, ~~∀~~k≠i}.
	S_i^T = transpose S_i. Обратная к S_i - зависимость (обратное преобразование). При формировании S_i в ячейку j множества S_i^T добавляется индекс i. (учёт связи обратной к S_i). $S_i^T=\{j: i \in S_j\}$.
	1. [initialization] F=Ø, C=Ø, U=Ω.
	$\lambda_i=\lvert S_i^T \cap \Omega\rvert$. F=FU{i: λ_i=0}.
	2. [update weight coefficients]
	For {k: $\alpha_{kj} \neq 0$, для каждого {j}= {new F}, k ∈ U (без повторных посещений в цикле)} {
	2.1. Found m=max{$\lvert a_{kp}\rvert$: a_{kp} <0,p≠k }.
	2.2. {Для j: a_{kj}<0 && $\lvert a_{kj}\rvert$>0.2375*m&&j∩(new F, причём new F полученные только из S_i^T связей) делаем λ_k++.}
	} // конец цикла по к.
	3. if U= Ø(эквивалентно max λ_i==0), then stop. Else go to step 4.
	4. pick i∈U with max λ_i : C=C U {i},
	U:=U\{i}.
	5. for all j ∈ $\{S_i U S_i^T\}$∩U: F=FU{j}, U=U\{j}.
	Goto update step 2.
	A.3. Алгоритм огрубления сетки из РУМБА0.14. см. [1]. Версия 2017 г.

Версия опубликованная в статье [1] на основе которой функционировала версия программы 2016 года.	Текущая версия RS1 алгоритма функционирующая в данный момент 2017г.

Рис. 9. RS1 функция.

ПРИЛОЖЕНИЕ Б

Реализация АВЛ деревьев заимствована из книги Никлауса Вирта [3] и переведена с языка Modula2 на язык C вручную для использования в программе [1],[2]. Также использовалась статья Николая Ершова [4].

```
// Для АВЛ дерева
// 12.12.2015
// По видимому в 3D очень большое количество соседей и простой
// линейный двухсвязный список не справляется с таким большим
// количеством элементов по параметру быстродействие.
// Для сравнения в 2D при 1M неизвестных видно что линейный
// список справляется прекрасным образом со своим 2D числом соседей
// и даже по результатам
// профайлинга не видно чтобы он испытывал сколько-нибудь ощутимую
// нагрузку.

// АВЛ дерево это структура данных позволяющая производить все
// операции за log_2 (number of sosed count) операций.

// Адельсон-Вельский и Ландис 1962 (взято из интернета).
// узел с максимальным значением ключа крайний правый.

// для поля данных надо определить перегруженные операции,
// меньше, больше и равно.

// Поле данных в АВЛ дереве.
// перенесена в файл priority_queue.cpp
//struct data_BalTree
//{
// --> high priority --> for operation <,>
//integer ii;
//integer  i, countsosed;
// countsosed есть key.
//};

//отношения порядка только в insert и remove

// Узел АВЛ дерева.
struct node_AVL
{
	data_BalTree key;
	// Высота поддерева с корнем в данном узле.
	unsigned char height;
	node_AVL* left;
	node_AVL* right;
	// Конструктор.
	node_AVL(data_BalTree k) { key = k; left = right = 0; height = 1; }
};

// Работает также и с пустыми деревьями.
// Обёртка поля height
unsigned char height(node_AVL* p)
{
	return p ? p->height : 0;
};

// Вычисляет balance factor заданного узла
// работает только с ненулевыми указателями.
integer bfactor(node_AVL* p)
{
	return height(p->right) - height(p->left);
};

// Восстанавливает корректное значение поля height
// заданного узла (при условии, что значения этого поля
// в правом и левом дочерних узлах являются корректными).
void fixheight(node_AVL* p)
{
	unsigned char hl = height(p->left);
```

```
	unsigned char hr = height(p->right);
	p->height = (hl > hr ? hl : hr) + 1;
};

// Балансировка узлов.
node_AVL* rotateright(node_AVL* p)
{
	// правый поворот вокруг p
	node_AVL* q = p->left;
	p->left = q->right;
	q->right = p;
	fixheight(p);
	fixheight(q);
	return q;
};

// Левый поворот является симметричной копией правого :
node_AVL* rotateleft(node_AVL* q)
{
	// левый поворот вокруг q
	node_AVL* p = q->right;
	q->right = p->left;
	p->left = q;
	fixheight(q);
	fixheight(p);
	return p;
};

// Код выполняющий балансировку сводится к проверке условий и выполнению поворотов
node_AVL* balance(node_AVL* p) // балансировка узла p
{
	fixheight(p);
	if (bfactor(p) == 2)
	{
		if (bfactor(p->right) < 0)
			p->right = rotateright(p->right);
		return rotateleft(p);
	}
	if (bfactor(p) == -2)
	{
		if (bfactor(p->left) > 0)
			p->left = rotateleft(p->left);
		return rotateright(p);
	}
	return p; // балансировка не нужна
}

// Вставка ключей в дерево.
// Возвращает новое значение корня АВЛ дерева.
node_AVL* insert(node_AVL* &p, data_BalTree k)
{
	// Вставка ключа k в дерево с корнем p
	if (p == NULL) {
		node_AVL* r1 = NULL;
		r1 = new node_AVL(k);
		if (r1 == NULL) {
			// недостаточно памяти на данном оборудовании.
			printf("Problem : not enough memory on your equipment for r1 in
insert my_agregat_amg...\n");
			printf("Please any key to exit...\n");
			//getchar();
			system("pause");
			exit(1);
		}
```

```
                return r1;
        }
        if (k.countsosed < p->key.countsosed)
                p->left = insert(p->left, k);
        else if (k.countsosed > p->key.countsosed)
                p->right = insert(p->right, k);
        else if (k.i < p->key.i)
                p->left = insert(p->left, k);
        //else if (k.i > p->key.i)
        else
                p->right = insert(p->right, k);
        //else if (k.ii < p->key.ii)
        //p->left = insert(p->left, k);
        //else
        //p->right = insert(p->right, k);
        return balance(p);
} // insert

  // Возвращает true если узел найден в дереве
bool isfound(node_AVL* p, data_BalTree k)
{
        if (p == 0) return false; // ненайден.
        if (k.countsosed < p->key.countsosed)
                return isfound(p->left, k);
        else if (k.countsosed > p->key.countsosed)
                return isfound(p->right, k);
        else if (k.i < p->key.i)
                return isfound(p->left, k);
        else if (k.i > p->key.i)
                return isfound(p->right, k);
        //else if (k.ii < p->key.ii)
        //return isfound(p->left, k);
        //else if (k.ii > p->key.ii)
        //return isfound(p->right, k);
        else return true; // найден.
}

// Полное удаление бинарного дерева.
void clear_AVL(node_AVL* p)
{
        if (p != 0) {
                clear_AVL(p->left);
                clear_AVL(p->right);
                // удаляем лист.
                delete p;
                p = 0;
        }
} // clear_AVL

  // Удаление узла с занными свойства с сохранением сбалансированности.
node_AVL* findmin(node_AVL* p)
{
        // поиск узла с минимальным ключём в дереве p
        //if (!p) {
        return p->left ? findmin(p->left) : p;
        //}
        //else {
        // на поиск минимума подан нулевой указатель.
        //return 0;
        //}
} // findmin

node_AVL* findmax(node_AVL* p)
{
```

```
	// поиск узла с максимальным ключём в дереве p
	if (p != 0) {
		return p->right ? findmax(p->right) : p;
		/*
		#if doubleintprecision == 1
		if (p->right == 0) {
		//printf("%lld %lld %lld\n", p->key.countsosed, p->key.i, p->key.ii);
		//system("pause");
		return p;
		}
		else {
		//printf("%lld %lld %lld\n",p->key.countsosed,p->key.i,p->key.ii);
		findmax(p->right);
		}
		#else
		if (p->right == 0) {
		//printf("%d %d %d\n", p->key.countsosed, p->key.i, p->key.ii);
		//system("pause");
		return p;
		}
		else {
		//printf("%d %d %d\n",p->key.countsosed,p->key.i,p->key.ii);
		findmax(p->right);
		}
		#endif

		*/
	}
	else {
		// На поиск максимума подан нулевой указатель.
		return 0;
	}
} // findmax

data_BalTree get_max_AVL(node_AVL* p)
{
	// возвращение максимального узла в дереве.
	return p->right ? get_max_AVL(p->right) : p->key;
}

node_AVL* removemin(node_AVL* p)
{
	// удаление узла с минимальным ключом из дерева p
	if (p->left == 0)
		return p->right;
	p->left = removemin(p->left);
	return balance(p);
}

// Удаление заданного элемента из AVL дерева
// с полным сохранением балансировки.
// на возвращаемое значение можно не обращать внимания.
node_AVL* remove_AVL(node_AVL* p, data_BalTree k)
{
	// Отношение порядка определено
	// для структуры из трёх целых чисел.
	// удаление ключа k из дерева p
	if (p == 0) return 0;
	// Двоичный поиск нужного элемента.
	if (k.countsosed < p->key.countsosed)
		p->left = remove_AVL(p->left, k);
	else if (k.countsosed>p->key.countsosed)
		p->right = remove_AVL(p->right, k);
	else if (k.i < p->key.i)
```

```
		p->left = remove_AVL(p->left, k);
	else if (k.i>p->key.i)
		p->right = remove_AVL(p->right, k);
	//else if (k.ii < p->key.ii)
	//p->left = remove_AVL(p->left, k);
	//else if (k.ii>p->key.ii)
	//p->right = remove_AVL(p->right, k);
	else // k==p->key
	{
		node_AVL* q = p->left;
		node_AVL* r = p->right;
		delete p;
		p = 0;
		if (r == 0) return q;
		node_AVL* min = findmin(r);
		min->right = removemin(r);
		min->left = q;
		return balance(min);
	}

	// При выходе из рекурсии делаем балансировку.
	return balance(p);
}

// Вставка ключа K в дерево если ключа k_search
// еще нет в дереве или модификация ключа K_search на K.
// Возвращает новое значение корня АВЛ дерева.
node_AVL* insert_and_modify(node_AVL* p, data_BalTree k, data_BalTree k_search)
{
	if (isfound(p, k_search) == false) {
		//   узла в дереве нет.
		p = insert(p, k);
		return p;
	}
	else {
		// удаление k_search
		p = remove_AVL(p, k_search); // необходимое действие
                                                    //remove_AVL(p, k_search);
// приводит к ошибке.
		p = insert(p, k);                                   // вставка k.
		return p;
	}
}

// print_AVL for debug.
void print_AVL(node_AVL* p)
{
	if (p != 0) {
		print_AVL(p->left);
		for (integer i = 0; i <= p->height; i++) {
			printf(" ");
		}
#if doubleintprecision == 1
		//printf("%lld %lld %lld\n", p->key.countsosed, p->key.i, p->key.ii);
		printf("%lld %lld\n", p->key.countsosed, p->key.i);
#else
		//printf("%d %d %d\n", p->key.countsosed, p->key.i, p->key.ii);
		printf("%d %d\n", p->key.countsosed, p->key.i);
#endif

		print_AVL(p->right);
	}
} // print_AVL
```

```
  // тестирование АВЛ дерева
  // 12 декабря 2015 удовлетворительно.
void test_AVL()
{
	node_AVL* root = 0;
	data_BalTree d3;
	d3.countsosed = rand();
	d3.i = rand();
	//d3.ii = rand();
	root = remove_AVL(root, d3);
	for (integer i = 0; i < 10; i++)
	{
		data_BalTree d;
		d.countsosed = rand();
		d.i = rand();
		//d.ii = rand();
		root = insert_and_modify(root, d, d);
		print_AVL(root);
		if (i == 5) {

			//41 18467 6334


			data_BalTree d1;
			d1.countsosed = 41;
			d1.i = 18467;
			//d1.ii = 6334;
			data_BalTree d2;
			d2.countsosed = rand();
			d2.i = rand();
			//d2.ii = rand();
			root = insert_and_modify(root, d2, d1);
			print_AVL(root);
			printf("remove 41 18467 6334\n");
			root = remove_AVL(root, d1);
			print_AVL(root);
			printf("found max\n");
			node_AVL* emax;
			emax = findmax(root);
#if doubleintprecision == 1
			printf("maximum id %lld\n", emax->key.countsosed);
#else
			printf("maximum id %d\n", emax->key.countsosed);
#endif

			emax = 0;
			//clear_AVL(root);
			//root = 0;
			print_AVL(root);
		}

		system("pause");
	}
	system("pause");
}
```

ПРИЛОЖЕНИЕ В

Используемая реализация скошенного (SPLAY) дерева заимствована

отсюда:

```
/*
//
An implementation of top-down splaying
D. Sleator <sleator@cs.cmu.edu>
March 1992

"Splay trees", or "self-adjusting search trees" are a simple and
efficient data structure for storing an ordered set.  The data
structure consists of a binary tree, without parent pointers, and no
additional fields.  It allows searching, insertion, deletion,
deletemin, deletemax, splitting, joining, and many other operations,
all with amortized logarithmic performance.  Since the trees adapt to
the sequence of requests, their performance on real access patterns is
typically even better.  Splay trees are described in a number of texts
and papers [1,2,3,4,5].

The code here is adapted from simple top-down splay, at the bottom of
page 669 of [3].  It can be obtained via anonymous ftp from
spade.pc.cs.cmu.edu in directory /usr/sleator/public.

The chief modification here is that the splay operation works even if the
item being splayed is not in the tree, and even if the tree root of the
tree is NULL.  So the line:

t = splay(i, t);

causes it to search for item with key i in the tree rooted at t.  If it's
there, it is splayed to the root.  If it isn't there, then the node put
at the root is the last one before NULL that would have been reached in a
normal binary search for i.  (It's a neighbor of i in the tree.)  This
allows many other operations to be easily implemented, as shown below.

[1] "Fundamentals of data structures in C", Horowitz, Sahni,
and Anderson-Freed, Computer Science Press, pp 542-547.
[2] "Data Structures and Their Algorithms", Lewis and Denenberg,
Harper Collins, 1991, pp 243-251.
[3] "Self-adjusting Binary Search Trees" Sleator and Tarjan,
JACM Volume 32, No 3, July 1985, pp 652-686.
[4] "Data Structure and Algorithm Analysis", Mark Weiss,
Benjamin Cummins, 1992, pp 119-130.
[5] "Data Structures, Algorithms, and Performance", Derick Wood,
Addison-Wesley, 1993, pp 367-375.

The following code was written by Daniel Sleator, and is released
in the public domain.
      */
```

И лишь незначительно подверглась адаптации в программе [1],[2]. Также использованы материалы [5],[6].

```
integer size_splay_Tree;  /* number of nodes in the Tree_splay */
                                        /* Not actually needed for any of the
operations */

typedef struct Tree_splay_node Tree_splay;
struct Tree_splay_node {
      Tree_splay * left, *right;
      data_BalTree item;
};

Tree_splay* findmax(Tree_splay* p)
```

```
{
	// поиск узла с минимальным ключём в дереве p
	if (p != 0) {
		return p->right ? findmax(p->right) : p;
	}
	else {
		// На поиск максимума подан нулевой указатель.
		return 0;
	}
} // findmax

  // Возвращает true если узел найден в SPLAY дереве
bool isfound_recursive(Tree_splay* p, data_BalTree k)
{
	if (p == 0) return false; // ненайден.
	if (k.countsosed < p->item.countsosed)
		return isfound_recursive(p->left, k);
	else if (k.countsosed > p->item.countsosed)
		return isfound_recursive(p->right, k);
	else if (k.i < p->item.i)
		return isfound_recursive(p->left, k);
	else if (k.i > p->item.i)
		return isfound_recursive(p->right, k);
	//else if (k.ii < p->item.ii)
	//return isfound_recursive(p->left, k);
	//else if (k.ii > p->item.ii)
	//return isfound_recursive(p->right, k);
	else return true; // найден.
} // isfound_recursive in splay tree.

  // Возвращает true если узел найден в SPLAY дереве
bool isfound(Tree_splay* p1, data_BalTree k)
{
	//if (p == 0) return false; // не найден.

	Tree_splay* p;
	p = p1;

	for (;;) {
		if (p == 0) return false; // не найден.
		else if (k.countsosed < p->item.countsosed) {
			p = p->left;
		}
		else if (k.countsosed > p->item.countsosed) {
			p = p->right;
		}
		else if (k.i < p->item.i) {
			p = p->left;
		}
		else if (k.i > p->item.i) {
			p = p->right;
			//else if (k.ii < p->item.ii)
			//return isfound(p->left, k);
			//else if (k.ii > p->item.ii)
			//return isfound(p->right, k);
		}
		else {
			p = 0;
			return true; // найден.
		}
	}
} // isfound in splay tree.
```

```
Tree_splay * splay(data_BalTree i, Tree_splay * t) {
	/* Simple top down splay, not requiring i to be in the Tree_splay t.  */
	/* What it does is described above.
*/
	Tree_splay Nbuf, *l, *r, *y;
	if (t == NULL) return t;
	Nbuf.left = Nbuf.right = NULL;
	l = r = &Nbuf;

	for (;;) {
		if (i.countsosed < t->item.countsosed) {
			if (t->left == NULL) break;
			if ((i.countsosed < t->left->item.countsosed) || ((i.countsosed < t-
>left->item.countsosed) && (i.i < t->left->item.i)))
			{
				y = t->left;                                          /* rotate
right */
				t->left = y->right;
				y->right = t;
				t = y;
				if (t->left == NULL) break;
			}
			r->left = t;                                              /* link
right */
			r = t;
			t = t->left;
		}
		else if (i.countsosed > t->item.countsosed) {
			if (t->right == NULL) break;
			if ((i.countsosed > t->right->item.countsosed) || ((i.countsosed ==
t->right->item.countsosed) && (i.i > t->right->item.i))) {
				y = t->right;                                         /* rotate
left */
				t->right = y->left;
				y->left = t;
				t = y;
				if (t->right == NULL) break;
			}
			l->right = t;                                             /* link left
*/
			l = t;
			t = t->right;
		}
		else if (i.i < t->item.i) {
			if (t->left == NULL) break;
			if ((i.countsosed < t->left->item.countsosed) || ((i.countsosed == t-
>left->item.countsosed) && (i.i < t->left->item.i)))
			{
				y = t->left;                                          /* rotate
right */
				t->left = y->right;
				y->right = t;
				t = y;
				if (t->left == NULL) break;
			}
			r->left = t;                                              /* link
right */
			r = t;
			t = t->left;
		}
		else if (i.i > t->item.i) {
			if (t->right == NULL) break;
			if ((i.countsosed > t->right->item.countsosed) || ((i.countsosed ==
t->right->item.countsosed) && (i.i > t->right->item.i)))
```

```
                        {
                                y = t->right;                                   /* rotate
left */
                                t->right = y->left;
                                y->left = t;
                                t = y;
                                if (t->right == NULL) break;
                        }
                        l->right = t;                                           /* link left
*/
                        l = t;
                        t = t->right;
                }
                else {
                        break;
                }
        }
        l->right = t->left;                                                 /* assemble */
        r->left = t->right;
        t->left = Nbuf.right;
        t->right = Nbuf.left;
        return t;
}

// Данный метод не используется.
/* Here is how sedgewick would have written this.                                */
/* It does the same thing.
*/
Tree_splay * sedgewickized_splay(data_BalTree i, Tree_splay * t) {
        Tree_splay Nbuf, *l, *r, *y;
        if (t == NULL) return t;
        Nbuf.left = Nbuf.right = NULL;
        l = r = &Nbuf;

        for (;;) {
                if (i.countsosed < t->item.countsosed) {
                        if (t->left != NULL && i.countsosed < t->left->item.countsosed) {
                                y = t->left; t->left = y->right; y->right = t; t = y;
                        }
                        else if (t->left != NULL && i.countsosed == t->left->item.countsosed
&& i.i < t->left->item.i) {
                                y = t->left; t->left = y->right; y->right = t; t = y;
                        }
                        if (t->left == NULL) break;
                        r->left = t; r = t; t = t->left;
                }
                else if (i.countsosed > t->item.countsosed) {
                        if (t->right != NULL && i.countsosed > t->right->item.countsosed) {
                                y = t->right; t->right = y->left; y->left = t; t = y;
                        }
                        else if (t->right != NULL && i.countsosed == t->right-
>item.countsosed && i.i > t->right->item.i) {
                                y = t->right; t->right = y->left; y->left = t; t = y;
                        }
                        if (t->right == NULL) break;
                        l->right = t; l = t; t = t->right;
                }
                else if (i.i < t->item.i) {
                        if (t->left != NULL && i.countsosed < t->left->item.countsosed) {
                                y = t->left; t->left = y->right; y->right = t; t = y;
                        }
                        else if (t->left != NULL && i.countsosed == t->left->item.countsosed
&& i.i < t->left->item.i) {
                                y = t->left; t->left = y->right; y->right = t; t = y;
```

```
                        }
                        if (t->left == NULL) break;
                        r->left = t; r = t; t = t->left;
                }
                else if (i.i > t->item.i) {
                        if (t->right != NULL && i.countsosed > t->right->item.countsosed) {
                                y = t->right; t->right = y->left; y->left = t; t = y;
                        }
                        else if (t->right != NULL && i.countsosed == t->right-
>item.countsosed && i.i > t->right->item.i) {
                                y = t->right; t->right = y->left; y->left = t; t = y;
                        }
                        if (t->right == NULL) break;
                        l->right = t; l = t; t = t->right;
                }
                else break;
        }
        l->right = t->left; r->left = t->right; t->left = Nbuf.right; t->right =
Nbuf.left;
        return t;
}

Tree_splay * insert(data_BalTree i, Tree_splay * t) {
        /* Insert i into the Tree_splay t, unless it's already there.       */
        /* Return a pointer to the resulting Tree_splay.                            */
        Tree_splay * new_splay_node = NULL;

        new_splay_node = (Tree_splay *)malloc(sizeof(Tree_splay));
        if (new_splay_node == NULL) {
                printf("Ran out of space in SPLAY tree\n");
                system("pause");
                exit(1);
        }
        new_splay_node->item = i;
        if (t == NULL) {
                new_splay_node->left = new_splay_node->right = NULL;
                size_splay_Tree = 1;
                return new_splay_node;
        }
        t = splay(i, t);
        if (i.countsosed < t->item.countsosed) {
                new_splay_node->left = t->left;
                new_splay_node->right = t;
                t->left = NULL;
                size_splay_Tree++;
                return new_splay_node;
        }
        else if (i.countsosed > t->item.countsosed) {
                new_splay_node->right = t->right;
                new_splay_node->left = t;
                t->right = NULL;
                size_splay_Tree++;
                return new_splay_node;
        }
        else if (i.i < t->item.i) {
                new_splay_node->left = t->left;
                new_splay_node->right = t;
                t->left = NULL;
                size_splay_Tree++;
                return new_splay_node;
        }
        else if (i.i > t->item.i) {
                new_splay_node->right = t->right;
                new_splay_node->left = t;
```

```
                t->right = NULL;
                size_splay_Tree++;
                return new_splay_node;
        }
        else { /* We get here if it's already in the Tree_splay */
                   /* Don't add it again                                    */
                free(new_splay_node);
                return t;
        }
}

Tree_splay * delete_splay_Tree(data_BalTree i, Tree_splay * t) {
        /* Deletes i from the Tree_splay if it's there.                  */
        /* Return a pointer to the resulting Tree_splay.                       */
        Tree_splay * x;
        if (t == NULL) return NULL;
        t = splay(i, t);
        if ((i.countsosed == t->item.countsosed) && (i.i == t->item.i)) {
/* found it */
                if (t->left == NULL) {
                        x = t->right;
                }
                else {
                        x = splay(i, t->left);
                        x->right = t->right;
                }
                size_splay_Tree--;
                free(t);
                t = NULL;
                return x;
        }
        return t;                                       /* It wasn't there */
} // delete_splay_Tree

  // Вставка ключа K в дерево если ключа k_search
  // еще нет в дереве или модификация ключа K_search на K.
  // Возвращает новое значение корня АВЛ дерева.
Tree_splay* insert_and_modify(Tree_splay* &p, data_BalTree k, data_BalTree k_search)
{
        if (isfound(p, k_search) == false) {
                //   узла в дереве нет.
                p = insert(k, p);
                return p;
        }
        else {
                // удаление k_search
                p = delete_splay_Tree(k_search, p); // необходимое действие
                p = insert(k, p);                                               // вставка k.
                return p;
        }
} // insert_and_modify splay tree

  // Полное удаление бинарного splay дерева.
void clear_SPLAY(Tree_splay* p)
{
        if (p != 0) {
                clear_SPLAY(p->left);
                clear_SPLAY(p->right);
                // удаляем лист.
                // delete p;
                free(p);
                p = 0;
        }
        } // clear_SPLAY
```

ПРИЛОЖЕНИЕ Г

Реализация очереди по приоритетам на не сливаемой двоичной куче для поиска элемента по ключу в которой, а также для удаления элемента по ключу используется быстродействующая хеш таблица.

Данная программная реализация на языке C++ есть развитие варианта Р. Седжвика [5] в котором дополнительно реализована операция поиска и удаления по ключу. Также использовалось [6].

```
// Используется в алгебраическом многосеточном методе.
// Используются только следующие функции:
	// clear, remove, insert, readkeymaxelm.

// Соединяем с быстродействующей хеш таблицей.
template <class Item>
void exch(integer i, integer j, Item* &pq, integer* &qp, integer* &hash) {
	// exchange


	Item t;

	t = pq[j];
	pq[j] = pq[i];
	pq[i] = t;

	integer p;

	p = hash[qp[i]];
	hash[qp[i]] = hash[qp[j]];
	hash[qp[j]] = p;

	p = qp[j];
	qp[j] = qp[i];
	qp[i] = p;

}

template <class Item>
void fixUp(Item* &a, integer* &inda, integer* &hash, integer k)
{
	while (k > 1 && a[k / 2] < a[k])
	{
		integer kdiv2 = k / 2;

		exch(k, kdiv2, a, inda, hash);

		k = kdiv2;
	}
}

// Нисходящая установка структуры сортирующего дерева.
template <class Item>
void fixDown(Item* &a, integer* &inda, integer* &hash, integer k, integer N)
{
	while (2 * k <= N)
	{
```

```
                integer j = 2 * k;
                if (j < N&&a[j] < a[j + 1]) j++;
                if (!(a[k] < a[j])) break;


                exch(k, j, a, inda, hash);

                k = j;
        }
}

// Ключи должны быть уникальны, целочисленны и различны.
// Двух одинаковых ключей быть недолжно, иначе коллизия в хеш таблице.

template <class Item>
class PQ
{
private:
        // Хранение binary heap.
        Item *pq;
        // Обратный доступ по номеру в qp на ячейку в hash.
        integer *qp; // Ссылка на хеш таблицу.
        // Доступ по ключу к полю в pq.
        integer *hash; // Хеш таблица !!!
        integer N;
        integer isize;
        integer ihash_size;

public:
// Конструктор
        PQ(integer maxN, integer max_key_size)
        {
                isize = maxN;
                pq = new Item[maxN + 1];
                qp = new integer[maxN + 1];
                for (integer i_1 = 0; i_1 < maxN + 1; i_1++) {
                        // Инициализация: таблица пуста т.к. поле 0
                        // в массиве pq никогда не используется.
                        qp[i_1] = 0;
                }
                N = 0;
                // Хеш таблица !!!
                ihash_size = max_key_size;
                hash = new integer[max_key_size+2];
                for (integer i_1 = 0; i_1 < max_key_size + 2; i_1++) {
                        // Инициализация: таблица пуста т.к. поле 0
                        // в массиве pq никогда не используется.
                        hash[i_1] = 0;
                }
        }

// Деструктор
        ~PQ()
        {
                if (pq != NULL) delete[] pq;
                N = 0;
                if (qp != NULL) delete[] qp;
                if (hash != NULL) delete[] hash;
        }

        // Очищаем содержимое и она снова готова к использованию.
        void clear()
        {
```

```
        for (integer i_1 = 0; i_1 < N + 1; i_1++) {
                // Ускоренная очистка хеш таблицы.
                hash[qp[i_1]] = 0;
        }
        for (integer i_1 = 0; i_1 < isize + 1; i_1++) {
                // Инициализация: таблица пуста т.к. поле 0
                // в массиве pq никогда не используется.
                qp[i_1] = 0;
        }
        N = 0;
}

bool empty() const
{
        return N == 0;
}

// Есть ли элемент с данным ключём в таблице ?
bool isfound(integer key) {
        if (hash[key] == 0) {
                // Элемент отсутствует в хеш таблице.
                return false;
        }
        return true;
}

// Вернуть элемент с заданным ключём:
// Обязательно предполагается что ключ существует внутри таблицы.
Item get(integer key) {
        if (hash[key] == 0) {
                // Элемент отсутствует в хеш таблице.
                printf("priority queue get ERROR: get element not found.\n");
                system("pause");
                exit(1);
        }
        return pq[hash[key]];
}

// Просто прочитать масимальный элемент.
Item readmax()
{
        return pq[1];
}

integer readkeymaxelm() {
        return qp[1];
}

// Вставить элемент item в очередь по
// приоритетам если элемент item имеет ключ key.
template <class Item>
void insert(Item item, integer key)
{
        if (N + 1 > isize) {
                printf("ERROR!!! priority_queue stack overflow...\n");
                printf("N=%lld\n",N);
                system("pause");
                exit(1);
        }
        else {
                pq[++N] = item;
                hash[key] = N;
                qp[N] = key;
                fixUp(pq, qp, hash, N);
```

```
	}
}

// Возвратить максимальный элемент
// и удалить его.
Item getmax()
{
	exch(1, N, pq, qp, hash);

	fixDown(pq, qp, hash, 1, N - 1);
	return pq[N--];
}

// Заменяет элемент с ключём key на элемент val с тем же ключём key.
// При этом ключ key должен быть уникальным.
void modify(integer key, Item val)
{
	if (hash[key] == 0) {
		// Элемент отсутствует в хеш таблице.
		printf("priority queue modify ERROR: get element not found.\n");
		system("pause");
		exit(1);
	}

	pq[hash[key]] = val;
	// Теперь необходимо восстановить порядок кучи.
	integer i = hash[key];
	fixUp(pq, qp, hash, i);
	fixDown(pq, qp, hash, i, N);
}

// Удаление элемента с заданным значением ключа.
void remove(integer key)
{
	if (hash[key] == 0) {
		// Элемент отсутствует в хеш таблице.
		// Ничего не делаем т.к. элемента уже нет.
	}
	else {

		// Удаление.
		if (hash[key] == N) {
			N--;
			hash[key] = 0;
			qp[N + 1] = 0;
			// Ключ исключён из таблицы.
		}
		else {

			integer i = hash[key];

			exch(hash[key], N, pq, qp, hash);

			hash[qp[N]] = 0;
			qp[N] = 0;
			N--;

			// Теперь необходимо восстановить порядок кучи.
			fixUp(pq, qp, hash, i);
			fixDown(pq, qp, hash, i, N);



		}
```

```
        }
        //print_log('r');

}

// У элемента изменть значение старого ключа на новый ключ
// при этом меняется и само содержимое элемента.
void change(integer key_serch, integer key_new, integer item_new)
{
        if (hash[key_serch] == 0) {
                // Элемент отсутствует в хеш таблице.
                if (hash[key_new] != 0) {
                        // Элемент присутствует в хеш таблице.
                        pq[hash[key_new]] = item_new;
                        // Теперь необходимо восстановить порядок кучи.
                        fixUp(pq, qp, hash, hash[key_new]);
                        fixDown(pq, qp, hash, hash[key_new], N);
                }
                else {
                        // Вставка нового ключа с новыми данными.
                        insert(item_new, key_new);
                }
        }
        else {
                if (hash[key_new] != 0) {
                        // удаление старого ключа со всем его содержимым.
                        remove(key_search);
                        // Элемент присутствует в хеш таблице.
                        pq[hash[key_new]] = item_new;
                        // Теперь необходимо восстановить порядок кучи.
                        fixUp(pq, qp, hash[key_new]);
                        fixDown(pq, qp, hash[key_new], N);

                }
                else {
                        // key_new отсутствует.

                        hash[key_new] = hash[key_search];
                        hash[key_search] = 0; // исключение из дерева.
                        pq[hash[key_new]] = item_new;
                        qp[hash[key_new]] = key_new;
                        // Теперь необходимо восстановить порядок кучи.
                        fixUp(pq, qp, hash, hash[key_new]);
                        fixDown(pq, qp, hash, hash[key_new], N);

                }
        }
}


};
```

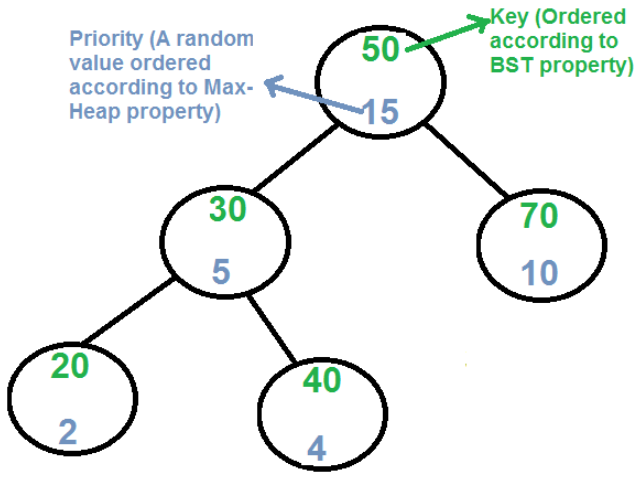
ПРИЛОЖЕНИЕ Д

Описание Дерамиды (Treap) заимствовано из [9],[10].

Описание Рандомизированного Двоичного Дерева Поиска (RBST) описанное у Р. Седжвика и по его книге на ХабраХабре не является рабочим и оно так и не заработало. Дерамида наиболее близка к RBST по духу и является 100% рабочей простой идеей. Чтобы это подтвердить приведем описание алгоритма и реализацию на языке C++.

Treap (A Randomized Binary Search Tree)

Like Red-Black and AVL Trees, Treap is a Balanced Binary Search Tree, but not guaranteed to have height as O(Log n). The idea is to use Randomization and Binary Heap property to maintain balance with high probability. The expected time complexity of search, insert and delete is O(Log n).



Every node of Treap maintains two values.
1) **Key** Follows standard BST ordering (left is smaller and right is greater)
2) **Priority** Randomly assigned value that follows Max-Heap property.

Basic Operation on Treap:
Like other self-balancing Binary Search Trees, Treap uses rotations to maintain Max-Heap property during insertion and deletion.

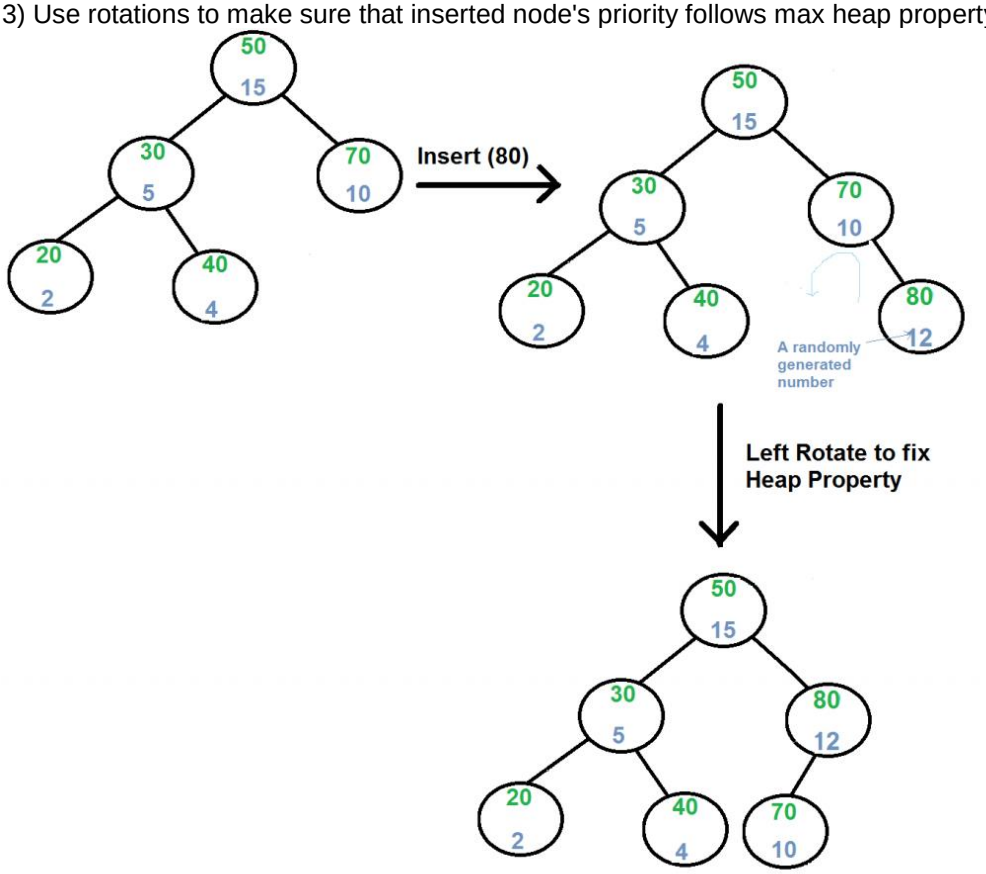
```
T1, T2 and T3 are subtrees of the tree rooted with y (on left side)
or x (on right side)
               y                               x
              / \     Right Rotation          /  \
             x   T3   - - - - - - - >        T1   y
            / \       < - - - - - - -            / \
           T1  T2     Left Rotation            T2  T3
Keys in both of the above trees follow the following order
      keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)
So BST property is not violated anywhere.
```

search(x)

Perform standard BST Search to find x.

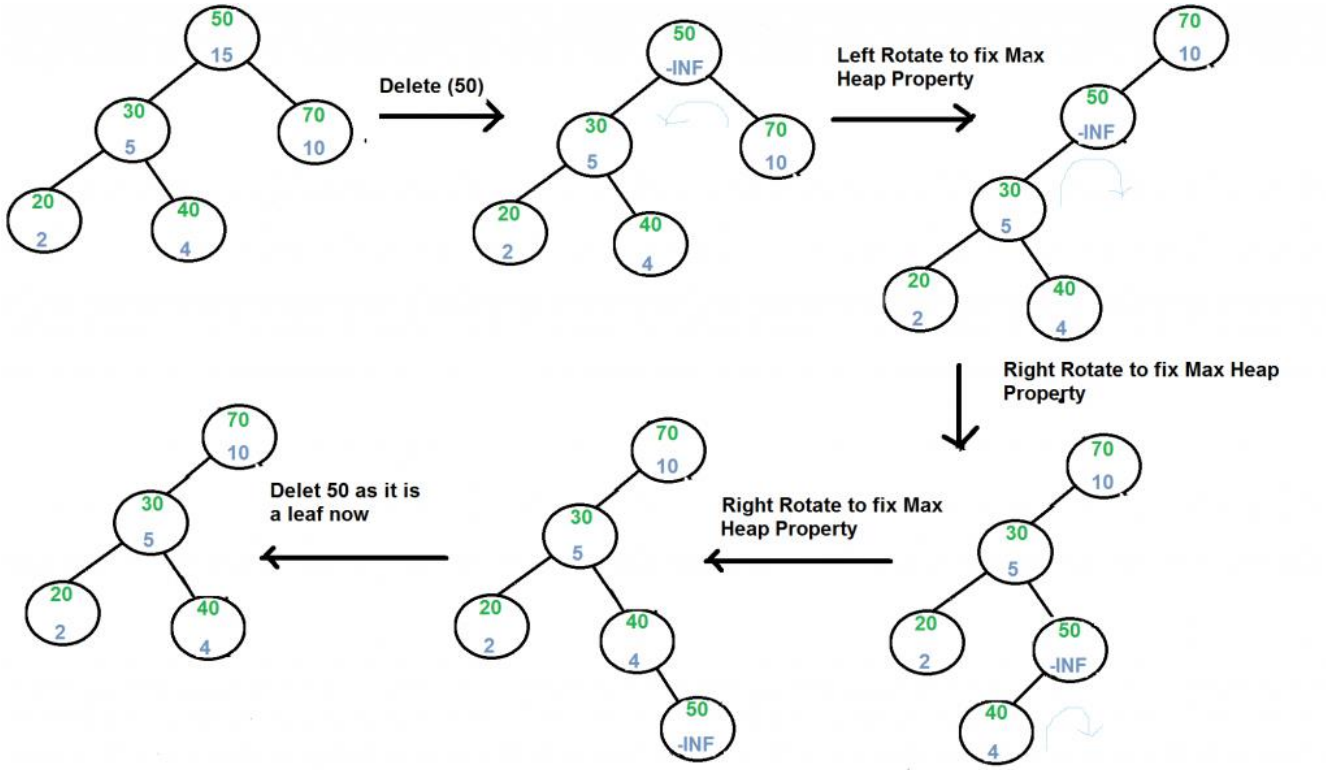
Insert(x):

1) Create new node with key equals to x and value equals to a random value.
2) Perform standard BST insert.
3) Use rotations to make sure that inserted node's priority follows max heap property.



Delete(x):

1) If node to be deleted is a leaf, delete it.
2) Else replace node's priority with minus infinite (-INF), and do appropriate rotations to bring the node down to a leaf.



Refer Implementation of Treap Search, Insert and Delete for more details.

References:

https://en.wikipedia.org/wiki/Treap

https://courses.cs.washington.edu/courses/cse326/00wi/handouts/lecture19/sld017.htm

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

```
// Рандомизированное дерево двоичного поиска.
// Достоинства: Простота и понятность реализации при обеспечении логарифмической
сложности операций.
// 24 august 2017
// По материалам статьи с хабрахабра: 7 июня 2012 в 23:43 Рандомизированные деревья
поиска

//Дерамида (объединение дерева и двоичной кучи). Сидель (Siedel) и Арагон (Aragon) 1996г.
// Имеет опыт использования в качестве базовой структуры в тщательно проработанной
библиотеке LEDA.
// Описание заимствовано с сайта GeeksForGeeks.
// Amat Treap Node
struct TreapNode
{
	integer  priority;
	data_BalTree key;
	TreapNode *left, *right;
};

/* T1, T2 and T3 are subtrees of the tree rooted with y
(on left side) or x (on right side)
y                               x
/ \     Right Rotation          /  \
x   T3   - - - - - - - >        T1   y
/ \       < - - - - - - -            / \
T1  T2     Left Rotation            T2  T3 */

// Amat utility function to right rotate subtree rooted with y
// See the diagram given above.
TreapNode *rightRotate(TreapNode *y)
```

```
{
	TreapNode *x = y->left, *T2 = x->right;

	// Perform rotation
	x->right = y;
	y->left = T2;

	// Return new root
	return x;
}

// Amat utility function to left rotate subtree rooted with x
// See the diagram given above.
TreapNode *leftRotate(TreapNode *x)
{
	TreapNode *y = x->right, *T2 = y->left;

	// Perform rotation
	y->left = x;
	x->right = T2;

	// Return new root
	return y;
}

/* Utility function to add a new key */
TreapNode* newNode(data_BalTree key)
{
	TreapNode* temp = new TreapNode;
	temp->key = key;
	//temp->priority = rand() % 100;
	//temp->priority = rand() % 200;
	//temp->priority = rand() % 65536;
	temp->priority = rand();
	temp->left = temp->right = NULL;
	return temp;
}

// C function to search a given key in a given BST
TreapNode* search_recursive(TreapNode* root, data_BalTree key)
{
	// Base Cases: root is null or key is present at root
	if (root == NULL || (root->key.i == key.i&&root->key.countsosed ==
key.countsosed))
		return root;

	// Key is greater than root's key
	if (root->key.countsosed < key.countsosed)
		return search_recursive(root->right, key);

	// Key is smaller than root's key
	if (root->key.countsosed > key.countsosed)
		return search_recursive(root->left, key);

	// Key is greater than root's key
	if (root->key.i < key.i)
		return search_recursive(root->right, key);

	// Key is smaller than root's key
	return search_recursive(root->left, key);
}

// Итеративный вариант должен бфыть быстрее.
// C function to search a given key in a given BST
```

```
TreapNode* search(TreapNode* root, data_BalTree key)
{

	TreapNode* scan = root;

	for (;;) {

		// Base Cases: root is null or key is present at root
		if (scan == NULL || (scan->key.i == key.i&&scan->key.countsosed ==
key.countsosed)) {
			return scan;
		}


		// Key is greater than root's key
		if (scan->key.countsosed < key.countsosed) {
			scan = scan->right;
		}

		// Key is smaller than root's key
		else if (scan->key.countsosed > key.countsosed)
		{
			scan = scan->left;
		}

		// Key is greater than root's key
		else if (scan->key.i < key.i) {
			scan = scan->right;
		}

		// Key is smaller than root's key
		else {
			scan = scan->left;
		}
	}
}


/* Recursive implementation of insertion in Treap */
TreapNode* insert(TreapNode* root, data_BalTree key)
{
	// If root is NULL, create a new node and return it
	if (!root)
		return newNode(key);

	// If key is smaller than root
	if (key.countsosed < root->key.countsosed)
	{
		// Insert in left subtree
		root->left = insert(root->left, key);

		// Fix Heap property if it is violated
		if (root->left->priority > root->priority)
			root = rightRotate(root);
	}
	else if (key.countsosed > root->key.countsosed) // If key is greater
	{
		// Insert in right subtree
		root->right = insert(root->right, key);

		// Fix Heap property if it is violated
		if (root->right->priority > root->priority)
			root = leftRotate(root);
	}
```

```
	else if (key.i <= root->key.i)
	{
		// Insert in left subtree
		root->left = insert(root->left, key);

		// Fix Heap property if it is violated
		if (root->left->priority > root->priority)
			root = rightRotate(root);
	}
	else  // If key is greater
	{
		// Insert in right subtree
		root->right = insert(root->right, key);

		// Fix Heap property if it is violated
		if (root->right->priority > root->priority)
			root = leftRotate(root);
	}
	return root;
}

/* Recursive implementation of Delete() */
TreapNode* deleteNode(TreapNode* root, data_BalTree key)
{
	if (root == NULL)
		return root;

	if (key.countsosed < root->key.countsosed)
		root->left = deleteNode(root->left, key);
	else if (key.countsosed > root->key.countsosed)
		root->right = deleteNode(root->right, key);
	else if (key.i < root->key.i)
		root->left = deleteNode(root->left, key);
	else if (key.i > root->key.i)
		root->right = deleteNode(root->right, key);

	// IF KEY IS AT ROOT

	// If left is NULL
	else if (root->left == NULL)
	{
		TreapNode *temp = root->right;
		delete(root);
		root = temp;  // Make right child as root
	}

	// If Right is NULL
	else if (root->right == NULL)
	{
		TreapNode *temp = root->left;
		delete(root);
		root = temp;  // Make left child as root
	}

	// If ksy is at root and both left and right are not NULL
	else if (root->left->priority < root->right->priority)
	{
		root = leftRotate(root);
		root->left = deleteNode(root->left, key);
	}
	else
	{
		root = rightRotate(root);
		root->right = deleteNode(root->right, key);
```

```
	}

	return root;
}

// Amat utility function to print tree
void inorder(TreapNode* root)
{
	if (root)
	{
		inorder(root->left);
		printf("key: %d  | priority: %d ", root->key, root->priority);
		if (root->left)
			printf(" | left child: %d", root->left->key);
		if (root->right)
			printf(" | right child: %d\n", root->right->key);
		inorder(root->right);
	}
}


// Полное удаление бинарного радномизированного дерева поиска.
void clear_random_tree(TreapNode* root)
{
	if (root != NULL) {
		clear_random_tree(root->left);
		clear_random_tree(root->right);
		// удаляем лист.
		delete root;
		//free(p);
		root = NULL;
	}
} // clear_random_tree

TreapNode* findmax_random_tree(TreapNode* &p)
{
	TreapNode* p1 = p;
	// поиск узла с минимальным ключём в дереве p
	if (p1 != NULL) {
		//return p1->right ? findmax_random_tree(p1->right) : p1;
		while (p1->right != NULL) p1 = p1->right;
		TreapNode* q = new TreapNode;
		q->key = p1->key;
		p1 = NULL;
		return q;
	}
	else {
		// На поиск максимума подан нулевой указатель.
		return NULL;
	}
} // findmax
```

ПРИЛОЖЕНИЕ Е

Описание используемого красно-черного дерева (RB – tree). См. [7] глава 13 с. 341-371. Интересное сравнение АВЛ и Красно-Черных деревьев содержится в [11] с. 278-290. (с. 289). Код программы реализующей Красно-Чёрное дерево найден в интернете http://www.cyberforum.ru/cpp-

beginners/thread1009501.html и немного адаптирован для применения в многосеточном методе.

```
// ссылка
// красно-черное дерево с++ реализация в поисковике yandex.
// http://www.cyberforum.ru/cpp-beginners/thread1009501.html
// 22.06.2018.
class RBtree {
	struct node_st { node_st *p1, *p2; data_BalTree value; bool red; }; // структура узла
	node_st *tree_root;			//!< корень
	integer nodes_count;			//!< число узлов дерева
private:
	node_st * NewNode(data_BalTree value);	//!< выделение новой вершины
	void DelNode(node_st*);		//!< удаление вершины
	void Clear(node_st*);		//!< снос дерева (рекурсивная часть)
	node_st *Rotate21(node_st*);	//!< вращение влево
	node_st *Rotate12(node_st*);	//!< вращение вправо
	void BalanceInsert(node_st**);	//!< балансировка вставки
	bool BalanceRemove1(node_st**);	//!< левая балансировка удаления
	bool BalanceRemove2(node_st**);	//!< правая балансировка удаления
	bool Insert(data_BalTree, node_st**);	//!< рекурсивная часть вставки
	bool GetMin(node_st**, node_st**);	//!< найти и убрать максимальный узел поддерева
	bool Remove(node_st**, data_BalTree);	//!< рекурсивная часть удаления
public: // отладочная часть
	enum check_code { error_balance, error_struct, ok }; // код ошибки
	void Show();			//!< вывод дерева
		//check_code Check();		//!< проверка дерева
		//bool TreeWalk(bool*, integer);	//!< обход дерева и сверка значений с массивом
private: // отладочная часть
	void Show(node_st*, integer, char);	//!< вывод дерева, рекурсивная часть
	//check_code Check(node_st*, integer, integer&);//!< проверка дерева (рекурсивная часть)
	//bool TreeWalk(node_st*, bool*, integer);	//!< обход дерева и сверка значений с массивом (рекурсивная часть)
```

```
public:
	RBtree();
	~RBtree();
	void Clear();			//!< снести дерево
	bool Find(data_BalTree);		//!< найти значение
	void Insert(data_BalTree);		//!< вставить значение
	void Remove(data_BalTree);		//!< удалить значение
	void InsertAndModify(data_BalTree, data_BalTree); //!< вставить если нет элемента,
модифицировать существующий элемент если он уже есть.
	integer GetNodesCount();	//!< узнать число узлов
	data_BalTree GetMaxElm(); //!< возвращает максимальный элемент в дереве.
};


//!< возвращает максимальный элемент в дереве.
data_BalTree RBtree::GetMaxElm() {
	if (tree_root) {
		node_st *node = tree_root;
		node_st *p2_loc = tree_root->p2;
		if (p2_loc) {
			while (p2_loc) {
				node = p2_loc;
				p2_loc = p2_loc->p2;
			}
			data_BalTree ir = node->value;
			node = NULL;
			p2_loc = NULL;
			return ir;
		}
		else {
			data_BalTree ir = node->value;
			node = NULL;
			return ir;
		}
```

```
        }
        else {
                //printf("Red - Black tree is empty...\n");
                //getchar();
                //exit(1);
        }

} // GetMaxElm

// Конструктор.
RBtree::RBtree()
{
        tree_root = 0;
        nodes_count = 0;
}

// деструктор.
RBtree::~RBtree()
{
        Clear(tree_root);
}

integer RBtree::GetNodesCount()
{
        return nodes_count;
}

// выделение новой вешины
RBtree::node_st *RBtree::NewNode(data_BalTree value)
{
        nodes_count++;
        node_st *node = new node_st;
        node->value = value;
```

```
        node->p1 = node->p2 = 0;
        node->red = true;
        return node;
}

// удаление вершины
void RBtree::DelNode(node_st *node)
{
        nodes_count--;
        delete node;
}

// снос дерева (рекурсивная часть)
void RBtree::Clear(node_st *node)
{
        if (!node) return;
        Clear(node->p1);
        Clear(node->p2);
        DelNode(node);
}

// вывод дерева, рекурсивная часть
//! \param node узел
//! \param depth глубина
//! \param dir   значёк
//! \code Show(root,0,'*'); \endcode
void RBtree::Show(node_st *node, integer depth, char dir)
{
        integer n;
        if (!node) return;
        for (n = 0; n<depth; n++) putchar(' ');
        printf("%c[%d %d] (%s)\n", dir, node->value.i, node->value.countsosed, node->red ?
"red" : "black");
```

```
        Show(node->p1, depth + 2, '-');
        Show(node->p2, depth + 2, '+');
}



// вращение влево
//! \param index индеск вершины
//! \result новая вершина дерева
RBtree::node_st *RBtree::Rotate21(node_st *node)
{
        node_st *p2 = node->p2;
        node_st *p21 = p2->p1;
        p2->p1 = node;
        node->p2 = p21;
        return p2;
}

// вращение вправо
//! \param index индеск вершины
//! \result новая вершина дерева
RBtree::node_st *RBtree::Rotate12(node_st *node)
{
        node_st *p1 = node->p1;
        node_st *p12 = p1->p2;
        p1->p2 = node;
        node->p1 = p12;
        return p1;
}



// балансировка вершины
void RBtree::BalanceInsert(node_st **root)
{
```

```
node_st *p1, *p2, *px1, *px2;
node_st *node = *root;
if (node->red) return;
p1 = node->p1;
p2 = node->p2;
if (p1 && p1->red) {
        px2 = p1->p2;             // задача найти две рядом стоящие крастные вершины
        if (px2 && px2->red) p1 = node->p1 = Rotate21(p1);
        px1 = p1->p1;
        if (px1 && px1->red) {
                node->red = true;
                p1->red = false;
                if (p2 && p2->red) { // отделаемся перекраской вершин
                        px1->red = true;
                        p2->red = false;
                        return;
                }
                *root = Rotate12(node);
                return;
        }
}
// тоже самое в другую сторону
if (p2 && p2->red) {
        px1 = p2->p1;             // задача найти две рядом стоящие красные вершины
        if (px1 && px1->red) p2 = node->p2 = Rotate12(p2);
        px2 = p2->p2;
        if (px2 && px2->red) {
                node->red = true;
                p2->red = false;
                if (p1 && p1->red) { // отделаемся перекраской вершин
                        px2->red = true;
                        p1->red = false;
                        return;
```

```
                        }
                        *root = Rotate21(node);
                        return;
                }
        }
}


bool RBtree::BalanceRemove1(node_st **root)
{
        node_st *node = *root;
        node_st *p1 = node->p1;
        node_st *p2 = node->p2;
        if (p1 && p1->red) {
                p1->red = false; return false;
        }
        if (p2 && p2->red) { // случай 1
                node->red = true;
                p2->red = false;
                node = *root = Rotate21(node);
                if (BalanceRemove1(&node->p1)) node->p1->red = false;
                return false;
        }
        unsigned int mask = 0;
        node_st *p21 = p2->p1;
        node_st *p22 = p2->p2;
        if (p21 && p21->red) mask |= 1;
        if (p22 && p22->red) mask |= 2;
        switch (mask)
        {
        case 0:     // случай 2 - if((!p21 || !p21->red) && (!p22 || !p22->red))
                p2->red = true;
                return true;
```

```
        case 1:
        case 3:     // случай 3 - if(p21 && p21->red)
                p2->red = true;
                p21->red = false;
                p2 = node->p2 = Rotate12(p2);
                p22 = p2->p2;
        case 2:     // случай 4 - if(p22 && p22->red)
                p2->red = node->red;
                p22->red = node->red = false;
                *root = Rotate21(node);
        }
        return false;
}

bool RBtree::BalanceRemove2(node_st **root)
{
        node_st *node = *root;
        node_st *p1 = node->p1;
        node_st *p2 = node->p2;
        if (p2 && p2->red) { p2->red = false; return false; }
        if (p1 && p1->red) { // случай 1
                node->red = true;
                p1->red = false;
                node = *root = Rotate12(node);
                if (BalanceRemove2(&node->p2)) node->p2->red = false;
                return false;
        }
        unsigned int mask = 0;
        node_st *p11 = p1->p1;
        node_st *p12 = p1->p2;
        if (p11 && p11->red) mask |= 1;
        if (p12 && p12->red) mask |= 2;
        switch (mask) {
```

```
        case 0:     // случай 2 - if((!p12 || !p12->red) && (!p11 || !p11->red))
                p1->red = true;
                return true;
        case 2:
        case 3:     // случай 3 - if(p12 && p12->red)
                p1->red = true;
                p12->red = false;
                p1 = node->p1 = Rotate21(p1);
                p11 = p1->p1;
        case 1:     // случай 4 - if(p11 && p11->red)
                p1->red = node->red;
                p11->red = node->red = false;
                *root = Rotate12(node);
        }
        return false;
}



bool RBtree::Find(data_BalTree value)
{
        node_st *node = tree_root;
        while (node) {
                if ((node->value.i == value.i) && (node->value.countsosed == value.countsosed))
return true;
                //node = node->value>value ? node->p1 : node->p2;
                if (value.countsosed < node->value.countsosed)
                        node = node->p1;
                else if (value.countsosed >node->value.countsosed)
                        node = node->p2;
                else if (value.i < node->value.i)
                        node = node->p1;
                else if (value.i > node->value.i)
                        node = node->p2;
```

```
	}
	return false;
}



// рекурсивная часть вставки
//! \result true если изменений небыло или балансировка в данной вершине не нужна
bool RBtree::Insert(data_BalTree value, node_st **root)
{
	node_st *node = *root;
	if (!node) *root = NewNode(value);
	else {
		if ((node->value.i == value.i) && (node->value.countsosed == value.countsosed))
return true;
		//if (Insert(value, value<node->value ? &node->p1 : &node->p2)) return true;
		if (value.countsosed < node->value.countsosed) {
			if (Insert(value, &node->p1))  return true;
		}
		else if (value.countsosed > node->value.countsosed) {
			if (Insert(value, &node->p2))  return true;
		}
		else if (value.i < node->value.i) {
			if (Insert(value, &node->p1)) return true;
		}
		else if (value.i > node->value.i) {
			if (Insert(value, &node->p2))  return true;
		}

		BalanceInsert(root);
	}

	return false;
}
```

```
// найти и убрать максимальный узел поддерева
//! \param root корень дерева в котором надо найти элемент
//! \retval res эелемент который был удалён
//! \result true если нужен баланс
bool RBtree::GetMin(node_st **root, node_st **res)
{
	node_st *node = *root;
	if (node->p1) {
		if (GetMin(&node->p1, res)) return BalanceRemove1(root);
	}
	else {
		*root = node->p2;
		*res = node;
		return !node->red;
	}
	return false;
}



// рекурсивная часть удаления
//! \result true если нужен баланс
bool RBtree::Remove(node_st **root, data_BalTree value)
{
	node_st *t, *node = *root;
	if (!node) return false;
	if ((node->value.countsosed<value.countsosed) || ((node->value.countsosed ==
value.countsosed) && (node->value.i<value.i))) {
		if (Remove(&node->p2, value)) return BalanceRemove2(root);
	}
	else if ((node->value.countsosed>value.countsosed) || ((node->value.countsosed ==
value.countsosed) && (node->value.i>value.i))) {
		if (Remove(&node->p1, value)) return BalanceRemove1(root);
```

```
        }
        else {
                bool res;
                if (!node->p2) {
                        *root = node->p1;
                        res = !node->red;
                }
                else {
                        res = GetMin(&node->p2, root);
                        t = *root;
                        t->red = node->red;
                        t->p1 = node->p1;
                        t->p2 = node->p2;
                        if (res) res = BalanceRemove2(root);
                }
                DelNode(node);
                return res;
        }
        return 0;
}



// вывод дерева
void RBtree::Show()
{
        printf("[tree]\n");
        Show(tree_root, 0, '*');
}

// функция вставки
void RBtree::Insert(data_BalTree value)
{
        Insert(value, &tree_root);
```

```
        if (tree_root) tree_root->red = false;
}

// удаление узла
void RBtree::Remove(data_BalTree value)
{
        Remove(&tree_root, value);
}

// снос дерева
void RBtree::Clear()
{
        Clear(tree_root);
        tree_root = 0;
}

//!< вставить если нет элемента, модифицировать существующий элемент если он уже есть.
void RBtree::InsertAndModify(data_BalTree value, data_BalTree ksearch) {

        if (Find(ksearch)) {
                // Обект уже существует
                Remove(ksearch);
                // Замена ksearch на value.
                Insert(value);
        }
        else {
                // Объекта не существует
                Insert(value);
        }
}

/*
// проверка дерева (рекурсивная часть)
```

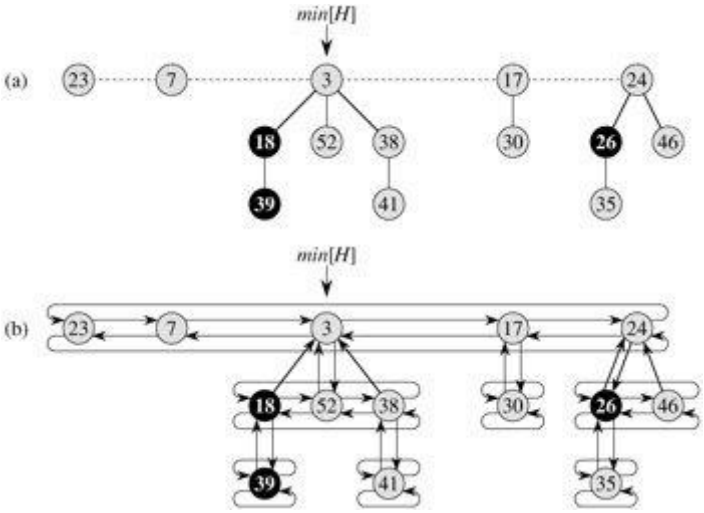
```
//! \param tree дерево
//! \param d    текущая чёрная глубина
//! \param h    эталонная чёрная глубина
//! \result 0 или код ошибки
RBtree::check_code RBtree::Check(node_st *tree, integer d, integer &h)
{
if (!tree) {
// количество чёрных вершин на любом пути одинаковое
if (h<0) h = d;
return h == d ? ok : error_balance;
}
node_st *p1 = tree->p1;
node_st *p2 = tree->p2;
// красная вершина должна иметь чёрных потомков
if (tree->red && (p1 && p1->red || p2 && p2->red)) return error_struct;
if (p1 && tree->value<p1->value || p2 && tree->value>p2->value) return error_struct;
if (!tree->red) d++;
check_code n = Check(p1, d, h); if (n) return n;
return Check(p2, d, h);
}



// проверка дерева
RBtree::check_code RBtree::Check()
{
integer d = 0;
integer h = -1;
if (!tree_root) return ok;
if (tree_root->red) return error_struct;
return Check(tree_root, d, h);
}

// обход дерева и сверка значений с массивом (рекурсивная часть)
```

```
//! \param node  корень дерева
//! \param array массив для сверки
//! \param size  размер массива
bool RBtree::TreeWalk(node_st *node, bool *array, integer size)
{
if (!node) return false;
integer value = node->value;
if (value<0 || value >= size || !array[value]) return true;
array[value] = false;
return TreeWalk(node->p1, array, size) || TreeWalk(node->p2, array, size);
}

// обход дерева и сверка значений с массивом
//! \param array массив для сверки
//! \param size  размер массива
bool RBtree::TreeWalk(bool *array, integer size)
{
if (TreeWalk(tree_root, array, size)) return true;
for (integer n = 0; n<size; n++) if (array[n]) return true;
return false;
}
*/

//=================================================================


void test_Red_Black_Tree() {
	RBtree root;
	data_BalTree d3;
	d3.countsosed = rand();
	d3.i = rand();
	root.Insert(d3);
	//root.Show();
```

```
        d3.countsosed = rand();
        d3.i = rand();
        root.Insert(d3);
        //root.Show();
        d3.countsosed = rand();
        d3.i = rand();
        root.Insert(d3);
        d3.countsosed = rand();
        d3.i = rand();
        root.Insert(d3);
        root.Show();
        d3 = (root.GetMaxElm());
        printf("i==%d countsosed==%d\n", d3.i, d3.countsosed);
        d3.countsosed = rand();
        d3.i = rand();
        root.Insert(d3);
        root.Show();
        d3.i = rand();
        root.Insert(d3);
        d3.i = rand();
        root.Insert(d3);
        d3.i = rand();
        root.Insert(d3);
        d3.i = rand();
        root.Insert(d3);
        d3.i = rand();
        d3 = (root.GetMaxElm());
        printf("i==%d countsosed==%d\n", d3.i, d3.countsosed);
        root.Remove(d3);

        d3 = (root.GetMaxElm());
        printf("i==%d countsosed==%d\n", d3.i, d3.countsosed);
        root.Show();
        getchar();
}
```

ПРИЛОЖЕНИЕ Ж.

Реализация Фибоначчиевой кучи.



См. Кормен [6, Третье издание 2017 год] глава 19 Фибоначчиевы пирамиды с. 542-568. См. [6, Второе издание 2007год] глава 19 Биномиальные пирамиды с.537-558. Глава 20 Фиббоначчиевы пирамиды с. 558-581. Лекционный курс на **youtube** [11], [12] обсуждает филосовски и приводит реализации многих куч, в частности двоичной и фибоначчиевой.

```
// Наиболее адаптированная к программе AliceFlow версия фибоначчиевой кучи.
// 15.07.2018
/* Copyright(c) 2010, Robin Message <Robin.Message@cl.cam.ac.uk>
Все права защищены.
Распространение и использование в виде исходного и двоичного кода, с или без
модификация допускается при соблюдении следующих условий :
*При распространении исходного кода должны сохраняться вышеуказанные авторские права
обратите внимание, этот список условий и следующий отказ от ответственности.
* При повторном распространении двоичного кода должна сохраняться
обратите внимание, этот список условий и следующий отказ от ответственности в
документация и / или другие материалы, поставляемые вместе с дистрибутивом.
* Ни название Univsersity Кембриджа, ни
имена его участников могут использоваться для поддержки или продвижения продуктов
производные от данного программного обеспечения без предварительного письменного
разрешения.
ЭТО ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ПРЕДОСТАВЛЯЕТСЯ ВЛАДЕЛЬЦАМИ АВТОРСКИХ ПРАВ И УЧАСТНИКАМИ "КАК
ЕСТЬ" И
ЛЮБЫЕ ЯВНЫЕ ИЛИ ПОДРАЗУМЕВАЕМЫЕ ГАРАНТИИ, ВКЛЮЧАЯ, НО НЕ ОГРАНИЧИВАЯСЬ, ПОДРАЗУМЕВАЕМЫЕ
ГАРАНТИИ ТОВАРНОЙ ПРИГОДНОСТИ И ПРИГОДНОСТИ ДЛЯ ОПРЕДЕЛЕННОЙ ЦЕЛИ
ОТКАЗАВШИЙСЯ.НИ В КОЕМ СЛУЧАЕ КЕМБРИДЖСКИЙ УНИВЕРСИТЕТ
*/

template <class V> class FibonacciHeap;


template <class V> struct node {
private:
	// Указатель на левый сестринский узел.
	node<V>* prev;
	// указатель на правый сестринский узел.
	node<V>* next;
	// указатель на один из дочерних узлов.
	node<V>* child;
```

```
        // указатель на родительский узел.
        node<V>* parent;
        V value;

        // количество дочерних узлов.
        int degree;


        //логическое значение, которое указывает,
        //были ли потери узлом x дочерних узлов,
        //начиная с момента, когда  x стал дочерним
        //узлом какого-то другого узла.
        //FIBONNACCI_HEAP
        bool marked;
public:
        friend class FibonacciHeap<V>;
        node<V>* getPrev() { return prev; }
        node<V>* getNext() { return next; }
        node<V>* getChild() { return child; }
        node<V>* getParent() { return parent; }
        V getValue() { return value; }
        bool isMarked() { return marked; }

        bool hasChildren() { return child; }
        bool hasParent() { return parent; }
};

template <class V> struct FiboHashNode {
        node<V>* link;
        integer count_sosed;
};

template <class V> class FibonacciHeap {
protected:
        node<V>* heap;
        FiboHashNode<V>* hash_index; // Хеш таблица !!!
        integer isize;
public:

        FibonacciHeap() {
                heap = _empty();

                isize = 0;//future n_a+1
                hash_index = NULL;
        }

        void WakeUp2(integer isize_loc) {
                isize = isize_loc;
                hash_index = new FiboHashNode<V>[isize];
                for (integer i = 0; i < isize; i++) {
                        hash_index[i].link = NULL;
                        hash_index[i].count_sosed = 0;
                }
        }

        void WakeUp(integer isize_loc) {
                heap = _empty();

                isize = isize_loc;
                hash_index = new FiboHashNode<V>[isize];
                for (integer i = 0; i < isize; i++) {
                        hash_index[i].link = NULL;
                        hash_index[i].count_sosed = 0;
                }
```

```
}

void Clear() {
	if (heap) {
		if (hash_index != NULL) {
			for (integer i = 0; i < isize; i++) {
				hash_index[i].link = NULL;
				hash_index[i].count_sosed = 0;
			}
		}
		_deleteAll(heap);
	}
}

void UpdateSize(integer isize_loc) {
	isize = isize_loc;
}

virtual ~FibonacciHeap() {
	if (heap) {
		for (integer i = 0; i < isize; i++) {
			hash_index[i].link = NULL;
			hash_index[i].count_sosed = 0;
		}
		delete[] hash_index;
		hash_index = NULL;
		_deleteAll(heap);
	}
	else {
		if (hash_index != NULL) {
			for (integer i = 0; i < isize; i++) {
				hash_index[i].link = NULL;
				hash_index[i].count_sosed = 0;
			}
			delete[] hash_index;
			hash_index = NULL;
		}
	}
}
node<V>* insert(V value) {
	node<V>* ret = _singleton(value);
	heap = _merge(heap, ret);

	integer i = ((-value) % (isize));
	integer countsosed = ((-value) / (isize));
	hash_index[i].link = ret;
	hash_index[i].count_sosed = countsosed;

	return ret;
}
void merge(FibonacciHeap& other) {
	heap = _merge(heap, other.heap);
	other.heap = _empty();
}

bool isEmpty() {
	return heap == NULL;
}

V getMinimum() {
	return heap->value;
}

V removeMinimum() {
```

```
        node<V>* old = heap;
        heap = _removeMinimum(heap);
        V ret = old->value;
        delete old;
        old = NULL;
        return ret;
}

void decreaseKey(node<V>* n, V value) {
        heap = _decreaseKey(heap, n, value);
}

void deleteKey(V value) {
        //node<V>* find_ = find(value);
        integer i = ((-value) % (isize));
        integer countsosed = ((-value) / (isize));

        if (hash_index != NULL) {
                node<V>* find_ = hash_index[i].link;
                if (find_ != NULL) {
                        hash_index[i].link = NULL;
                        hash_index[i].count_sosed = 0;

                        decreaseKey(find_, -4294967296);
                        removeMinimum();
                }
        }
}

void deleteKey(data_BalTree ddel) {

        if (hash_index != NULL) {
                node<V>* find_ = hash_index[ddel.i].link;
                if (find_ != NULL) {
                        hash_index[ddel.i].link = NULL;
                        hash_index[ddel.i].count_sosed = 0;

                        decreaseKey(find_, -4294967296);
                        removeMinimum();
                }
        }
}

//fibo_n = fibo_heap.find(-veb_dsearch_key);
//if (fibo_n == NULL) {
//      fibo_heap.insert(-veb_dadd_key);
//}
//else {
//      fibo_heap.decreaseKey(fibo_n, -veb_dadd_key);
//}
//fibo_n = NULL;
void insert_and_modify(V value_search, V value_add) {
        //node<V>* find_ = find(value_search);
        integer i = ((-value_search) % (isize));
        integer countsosed = ((-value_search) / (isize));

        if (hash_index == NULL) {
                insert(value_add);
        }
        else {
                node<V>* find_ = hash_index[i].link;
                if (find_ == NULL) {
                        insert(value_add);
                }
```

```
                        else {
                                // меняем позицию указателя на find_.
                                hash_index[i].link = NULL;
                                hash_index[i].count_sosed = 0;

                                i = ((-value_add) % (isize));
                                countsosed = ((-value_add) / (isize));

                                hash_index[i].link = find_;
                                hash_index[i].count_sosed = countsosed;
                                decreaseKey(find_, value_add);
                        }
                        find_ = NULL;
                }
        }

        node<V>* find(V value) {
                //return _find(heap, value);
                return hash_index[-value];
        }
private:
        node<V>* _empty() {
                return NULL;
        }

        node<V>* _singleton(V value) {
                node<V>* n = new node<V>;
                n->value = value;
                n->prev = n->next = n;
                n->degree = 0;
                n->marked = false;
                n->child = NULL;
                n->parent = NULL;
                return n;
        }

        node<V>* _merge(node<V>* a, node<V>* b) {
                if (a == NULL)return b;
                if (b == NULL)return a;
                if (a->value>b->value) {
                        node<V>* temp = a;
                        a = b;
                        b = temp;
                }
                node<V>* an = a->next;
                node<V>* bp = b->prev;
                a->next = b;
                b->prev = a;
                an->prev = bp;
                bp->next = an;
                return a;
        }

        void _deleteAll(node<V>* n) {
                if (n != NULL) {
                        node<V>* c = n;
                        do {
                                node<V>* d = c;
                                c = c->next;
                                _deleteAll(d->child);
                                delete d;
                                d = NULL;
                        } while (c != n);
                }
```

```
}

void _addChild(node<V>* parent, node<V>* child) {
	child->prev = child->next = child;
	child->parent = parent;
	parent->degree++;
	parent->child = _merge(parent->child, child);
}

void _unMarkAndUnParentAll(node<V>* n) {
	if (n == NULL)return;
	node<V>* c = n;
	do {
		c->marked = false;
		c->parent = NULL;
		c = c->next;
	} while (c != n);
}

node<V>* _removeMinimum(node<V>* n) {
	if (n == NULL)return n;
	_unMarkAndUnParentAll(n->child);
	if (n->next == n) {
		n = n->child;
	}
	else {
		n->next->prev = n->prev;
		n->prev->next = n->next;
		n = _merge(n->next, n->child);
	}
	if (n == NULL)return n;
	node<V>* trees[64] = { NULL };

	while (true) {
		if (trees[n->degree] != NULL) {
			node<V>* t = trees[n->degree];
			if (t == n)break;
			trees[n->degree] = NULL;
			if (n->value<t->value) {
				t->prev->next = t->next;
				t->next->prev = t->prev;
				_addChild(n, t);
			}
			else {
				t->prev->next = t->next;
				t->next->prev = t->prev;
				if (n->next == n) {
					t->next = t->prev = t;
					_addChild(t, n);
					n = t;
				}
				else {
					n->prev->next = t;
					n->next->prev = t;
					t->next = n->next;
					t->prev = n->prev;
					_addChild(t, n);
					n = t;
				}
			}
			continue;
		}
		else {
			trees[n->degree] = n;
```

```
                        }
                        n = n->next;
                }
                node<V>* min = n;
                node<V>* start = n;
                do {
                        if (n->value<min->value)min = n;
                        n = n->next;
                } while (n != start);
                return min;
        }

        node<V>* _cut(node<V>* heap, node<V>* n) {
                if (n->next == n) {
                        n->parent->child = NULL;
                }
                else {
                        n->next->prev = n->prev;
                        n->prev->next = n->next;
                        n->parent->child = n->next;
                }
                n->next = n->prev = n;
                n->marked = false;
                return _merge(heap, n);
        }

        node<V>* _decreaseKey(node<V>* heap, node<V>* n, V value) {
                if (n->value<value)return heap;
                n->value = value;
                if (n->parent) {
                        if (n->value<n->parent->value) {
                                heap = _cut(heap, n);
                                node<V>* parent = n->parent;
                                n->parent = NULL;
                                while (parent != NULL && parent->marked) {
                                        heap = _cut(heap, parent);
                                        n = parent;
                                        parent = n->parent;
                                        n->parent = NULL;
                                }
                                if (parent != NULL && parent->parent != NULL)parent->marked =
true;
                        }
                }
                else {
                        if (n->value < heap->value) {
                                heap = n;
                        }
                }
                return heap;
        }

        // Дьявольски медленный поиск _find.
        // В следующей версии мы заменим операцию
        //поиска на быстродействующую хеш таблицу.
        // Недостаток в том что хеш таблица получается слишком большой,
        // много времени на выделение памяти и на её освобождение.
        node<V>* _find(node<V>* heap, V value) {
                node<V>* n = heap;
                if (n == NULL)return NULL;
                do {
                        if (n->value == value)return n;
                        node<V>* ret = _find(n->child, value);
                        if (ret)return ret;
```

```
                n = n->next;
            } while (n != heap);
            return NULL;
        }
};
```

Список литературы

1. К.А. Иванов. Однопоточная реализация многосеточного метода с magic интерполяцией. Электронная техника. Серия 2. Полупроводниковые приборы. Выпуск 1 (240) 2016, с. 19-36.
2. https://github.com/kirill7785/algebraic-multigrid-method
3. Никлаус Вирт. Алгоритмы и структуры данных. На языке Modula2. Санкт Петербург,; Невский Диалект, 2001. С.261-274.
4. Николай Ершов. АВЛ-деревья tutorial. 3 сентября 2012. Хабрахабр.
5. Splay-деревья/ Хабрахабр. SeptiM 22 февраля 2014.
6. Роберт Седжвик, Алгоримы на C++. DiaSoft, 2002.
7. Кормен, Т., Лейзерсон, Ч., Ривест, Р, Штайн, К. Алгоритмы: построение и анализ = Introduction to Algorithms.
8. Performance Analysis of BSTs in System Software. Ben Pfaff. Stanford University.
9. http://www.geeksforgeeks.org/treap-a-randomized-binary-search-tree/
10. http://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/
11. Ричард Хэзфилд, Лоуренс Кирби. Искусство программирования на C. Фундаментальные алгоритмы, структуры данных и примеры приложений. M. Dia-Soft 2001.
12. https://www.youtube.com/watch?v=7m7af2MNAGY (Лекция 1 июнь-июль 2018)
13. https://www.youtube.com/watch?v=39s4xAEgT8M (Лекция 2 июнь-июль 2018)