



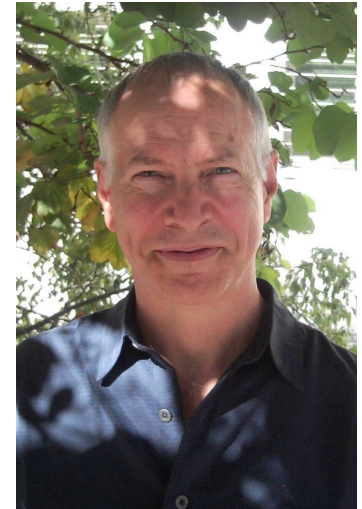
Фибоначчиева куча

История создания

Фибоначчиева куча была изобретена в 1984 году американскими учёными Майклом Фредманом и Робертом Тарьяном. Они искали структуру данных, которая позволила бы улучшить асимптотическую сложность приоритетных операций, особенно операции уменьшения ключа — узкого места в классических кучах.



Майкл Фредман



Роберт Тарьян



Актуальность Фибоначчиевой кучи

Фибоначчиева куча стала ключевым прорывом в теории структур данных и алгоритмов, так как позволила существенно ускорить алгоритмы на графах (например, алгоритм Дейкстры) и дала мощный инструмент для решения широкого круга задач, где важна работа с приоритетами и динамическими изменениями ключей.

Однако в реальных задачах она практически не применяется из-за больших констант в асимптотике. Она теоретически может дать прирост на очень больших данных, например, если надо хранить граф для карты местности всей Земли.



Математическая формулировка задачи

Фибоначчиева куча — это структура данных, поддерживающая мультимножество элементов с ключами из упорядоченного множества (обычно чисел) и обеспечивающая следующие операции:

- `insert(x)` - вставка элемента с ключом `x`
- `getMin()` - получение элемента с минимальным ключом
- `extractMin()` - удаление элемента с минимальным ключом
- `decreaseKey(h, delta)` - уменьшение ключа элемента, на который указывает дескриптор `h`, на значение `delta`
- `merge(fh1, fh2)` - объединение двух куч `fh1` и `fh2`



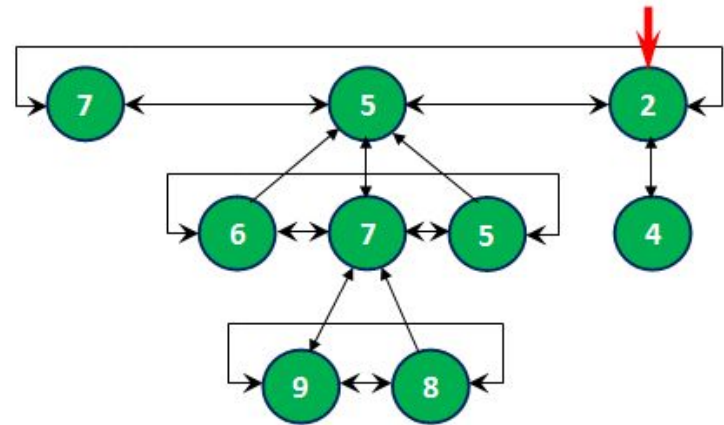
Асимптотика

Фибоначчиева куча основана на биномиальной. Однако в отличие от биномиальной, куча Фибоначчи позволяет делать операции merge за чистое $O(1)$ и decreaseKey за амортизированное $O^*(1)$. Асимптотика decreaseKey является основной причиной, по которой Фибоначчиева куча появилась.

	Binary	Binomial	Fibonacci
Insert	$O(\log N)$	$O(\log N)$	$O(1)$
GetMin	$O(1)$	$O(1)$	$O(1)$
Merge	$O(N)$	$O(\log N)$	$O(1)$
DecreaseKey	$O(\log N)$	$O(\log N)$	$O^*(1)$
ExtractMin	$O(\log N)$	$O(\log N)$	$O^*(\log N)$

Структура Фибоначчиевой кучи

Фибоначчиева куча представляет из себя множество деревьев, которые удовлетворяют основному требованию кучи — ключ родителя не больше ключей детей. Таким образом минимальный элемент в каждом дереве это его корень. А минимальный элемент в куче — один из корней.



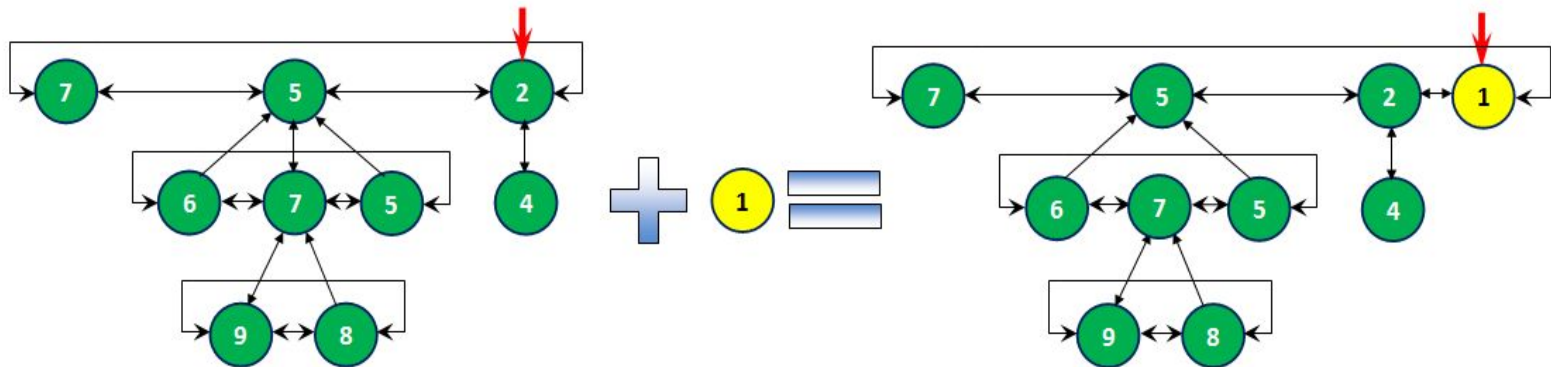


GetMin

Это самая простая операция, т.к. в мы постоянно поддерживаем указатель на минимальный элемент, так что для getMin надо лишь вернуть этот элемент.

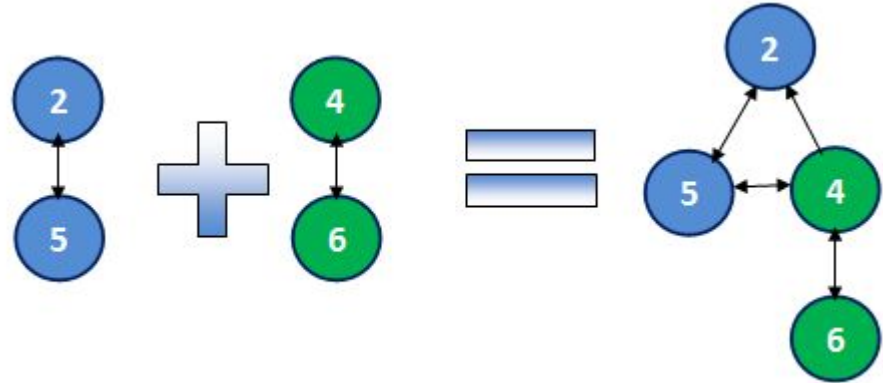
Insert

При добавлении нового элемента, просто добавляем его в двусвязный список корней и обновляем указатель на минимальный элемент.



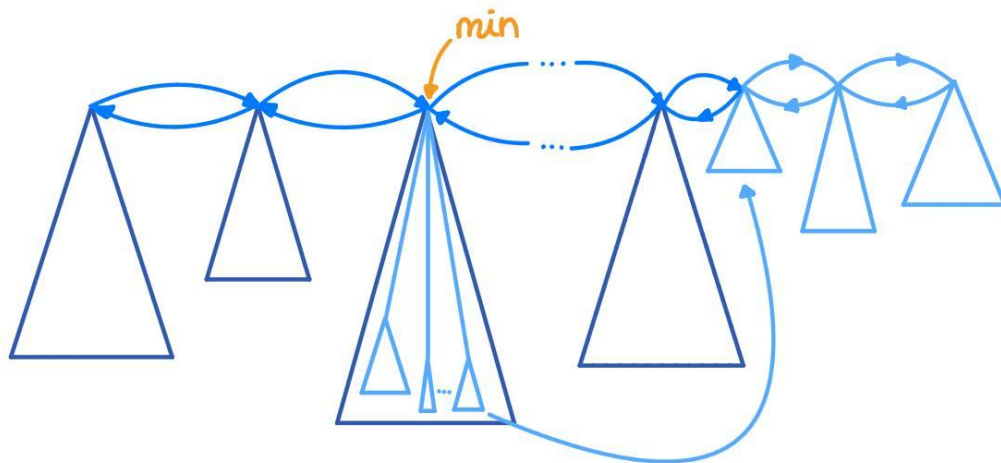
Merge

При объединении двух Фибоначчиевых куч просто сливаем списки корней. Более содержательно объединение двух Фибоначчиевых деревьев. Это объединение — вспомогательный метод в других операциях, поэтому обычно он вызывается только для деревьев одинакового ранга, где ранг Фибоначчиева дерева - количество детей у корня.



ExtractMin

Минимальный элемент — всегда один из корней. Если у этого корня нет детей мы просто удаляем его из двусвязного списка. Если дети есть, то в силу того, что дети корня образуют Фибоначчиеву кучу H' мы делаем merge исходной кучи и кучи H' . Далее вызываем операцию consolidate, о которой будет рассказано на следующем слайде.





consolidate

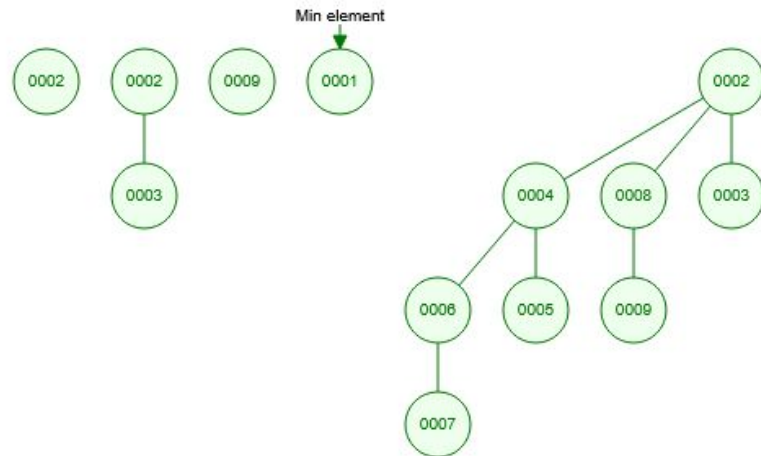
Одна из самых важных операций. Именно она обеспечивает обещанную асимптотику. Её суть в том, чтобы “уплотнить кучу” путём слияния Фибоначчиевых деревьев одинакового ранга.

Заводим массив $[0, D(N)]$. Где $D(N)$ — максимально возможный ранг дерева в куче на N элементах. Проходим циклом по деревьям кучи. Допустим, на очередной итерации мы имеем дерево ранга k . Есть два случая:

- 1) k -ая ячейка массива пуста — добавляем указатель на дерево в эту ячейку
- 2) k -ая ячейка массива занята — делаем merge текущего дерева и дерева, которое лежит в этой ячейке. Пытаемся добавить полученное дерево ранга $k + 1$ а соответствующую ячейку массива. Если она пустая — добавляем, если нет повторяем пункт 2.

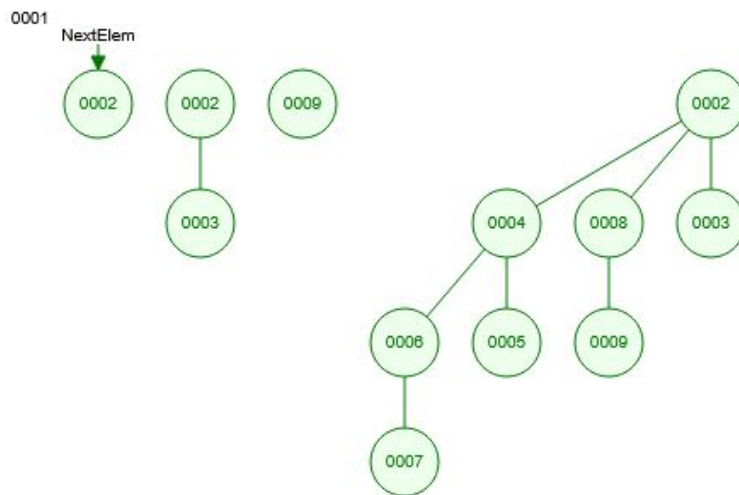
consolidate

Допустим, мы хотим сделать ExtractMin для такой кучи.



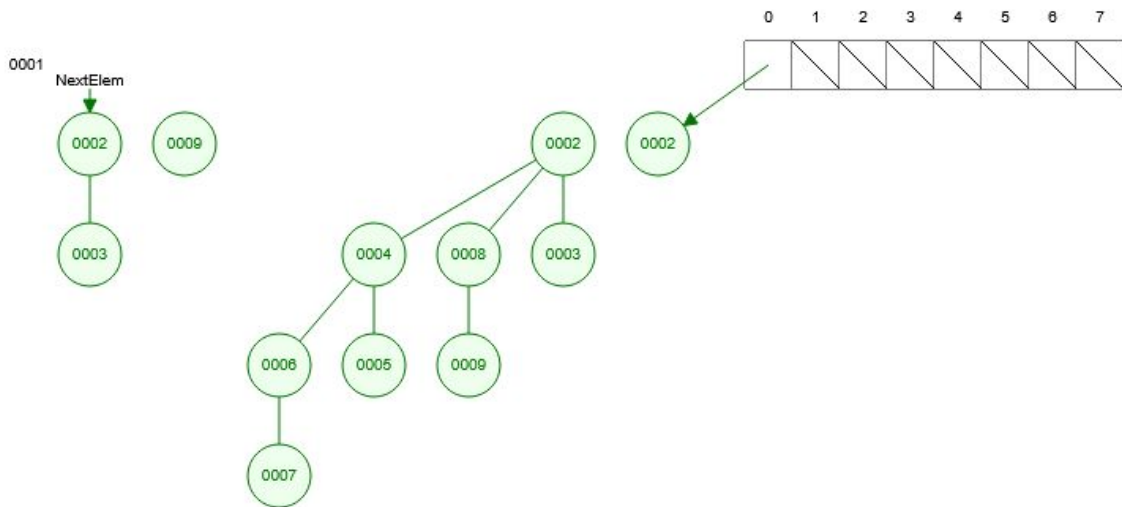
consolidate

Создали массив. Начали цикл по деревьям кучи. На нулевой итерации обрабатываем дерево ранга ноль.



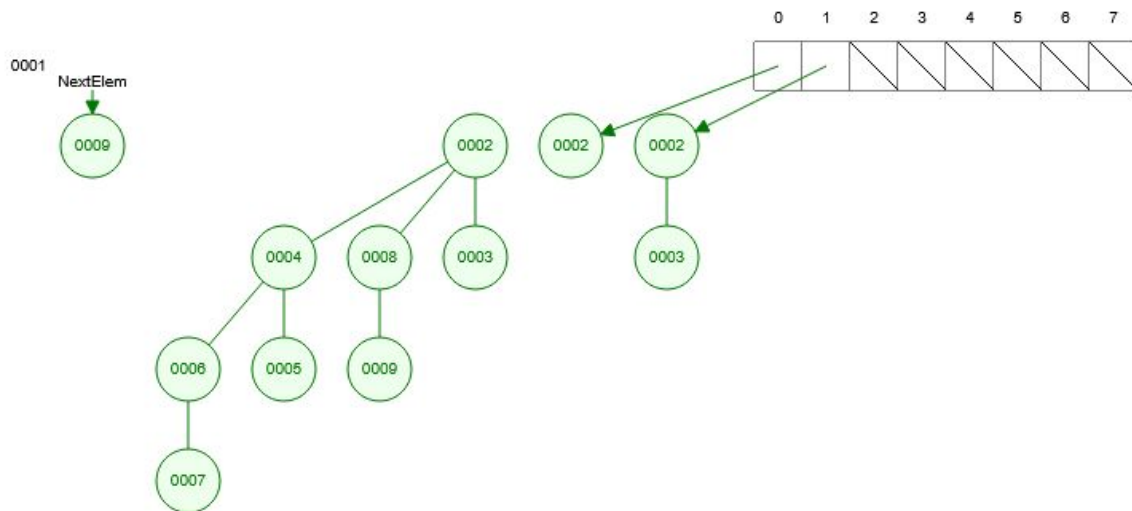
consolidate

Привязали 0-ую ячейку массива
с первым деревом.
Обрабатываем следующее
дерево — оно ранга 1.



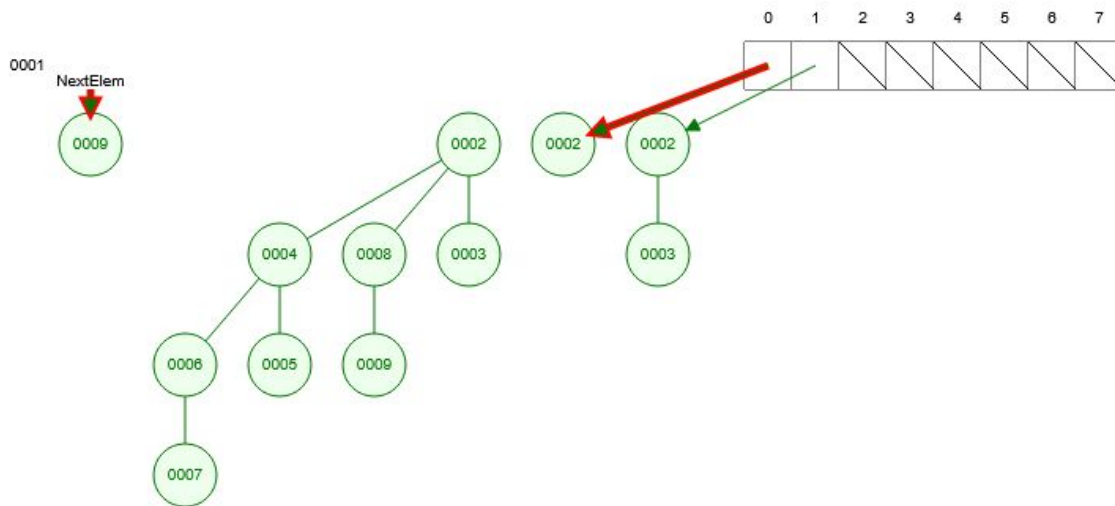
consolidate

1 ячейка массива пуста,
поэтому мы без проблем
связываем ее с деревом.
Теперь нужно обработать
дерево ранга 0.



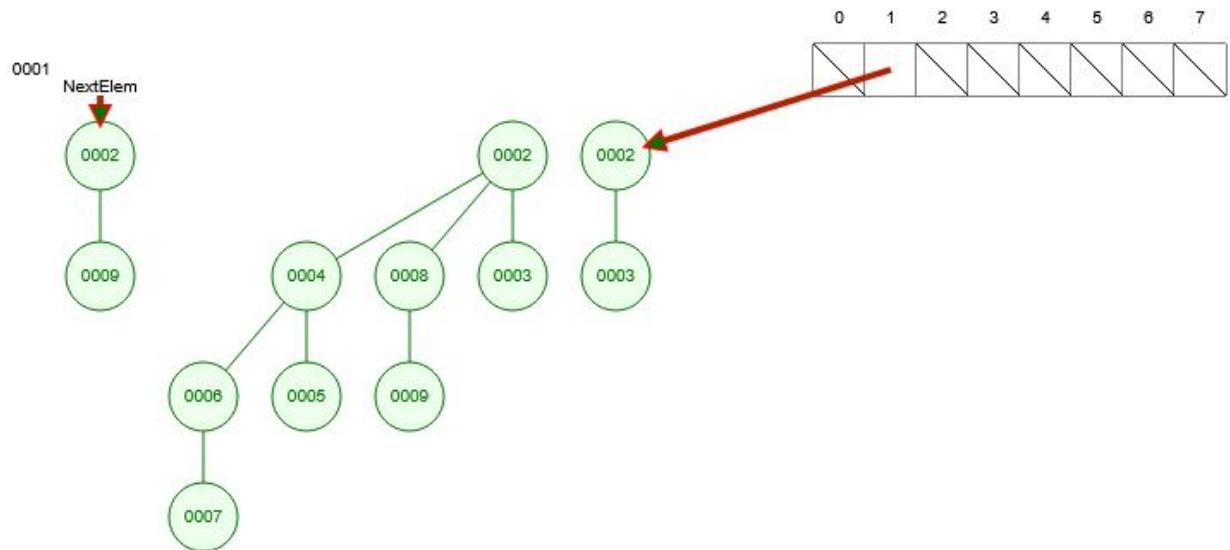
consolidate

Вот тут начинаются сложности. Ячейка 0 занята, поэтому нужно мерджить деревья.



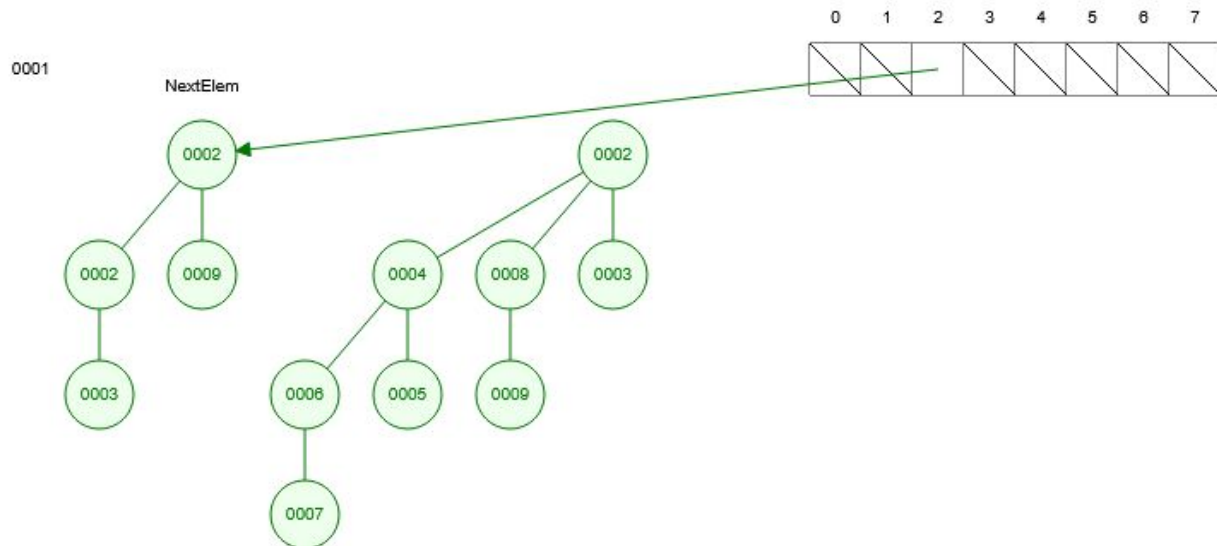
consolidate

Смерджив деревья ранга 0 получили дерево ранга 1. Пытаемся добавить его в массив, но ячейка 1 занята. Мерджим деревья ранга 1.



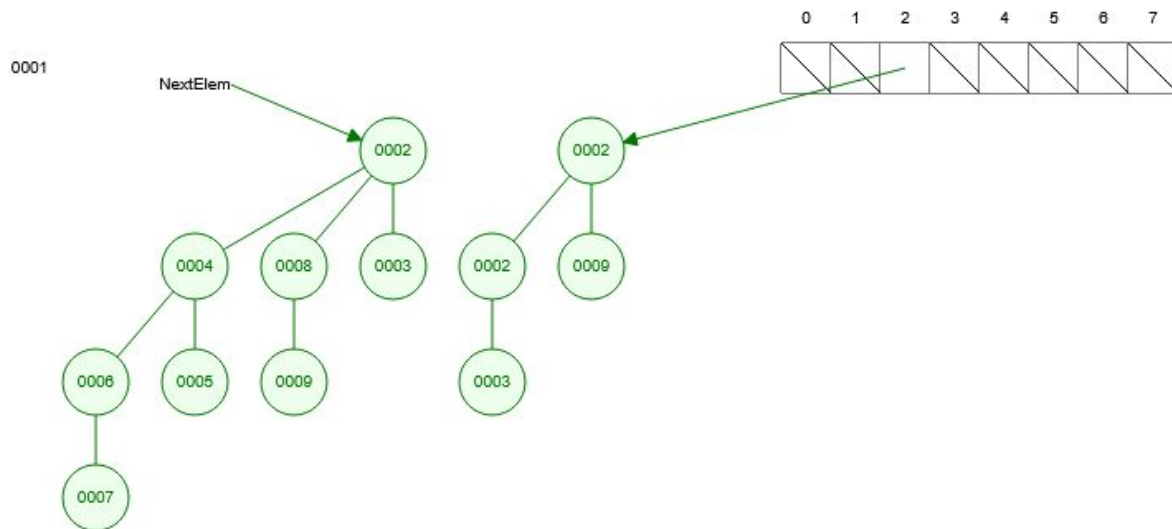
consolidate

Смерджив деревья ранга 1 получили дерево ранга 2. На этот раз ячейка, в которую, мы хотим добавить дерево пустая, поэтому обходимся без слияний.



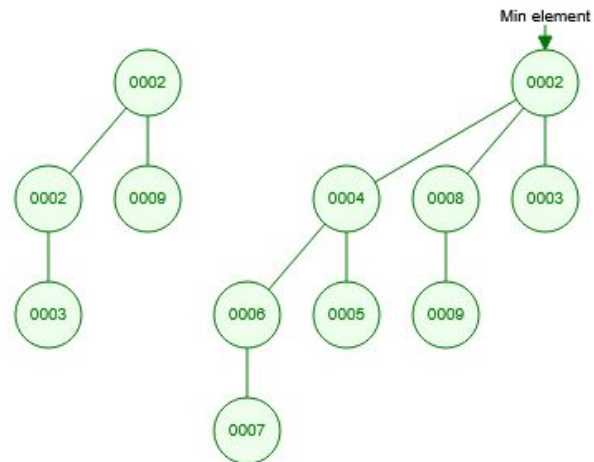
consolidate

Теперь обрабатываем
дерево ранга 3. Ячейка 3
пустая — просто добавляем
текущее дерево в массив.



consolidate

Цикл закончен. Операция consolidate выполнена.





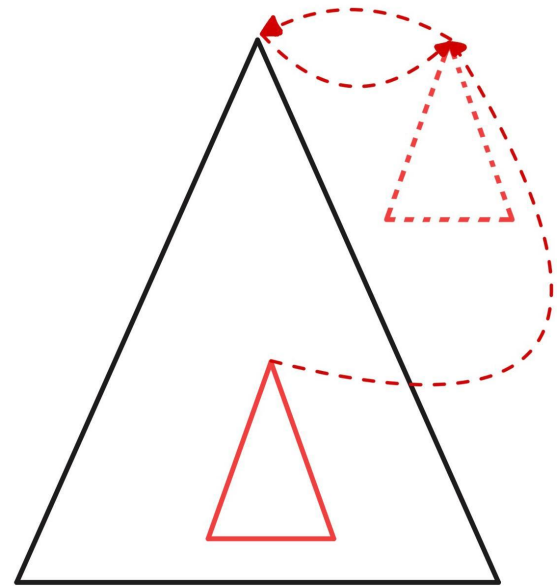
DecreaseKey

Для данной операции в каждой вершине хранится предикат `mark`, который равен `false` у всех корней деревьев и выставляется в `true`, если ребенка вершины удалили. Удаление может возникать в операции `DecreaseKey` или быть отдельным методом, который реализуется очень просто: методом `DecreaseKey` присваиваем вершине минимально возможное значение, после чего делаем `ExtractMin`.

Начинается операция `DecreaseKey(h, delta)` с того, что у вершины с дескриптором `h` ключ уменьшается на значение `delta`. Дальше есть два случая.

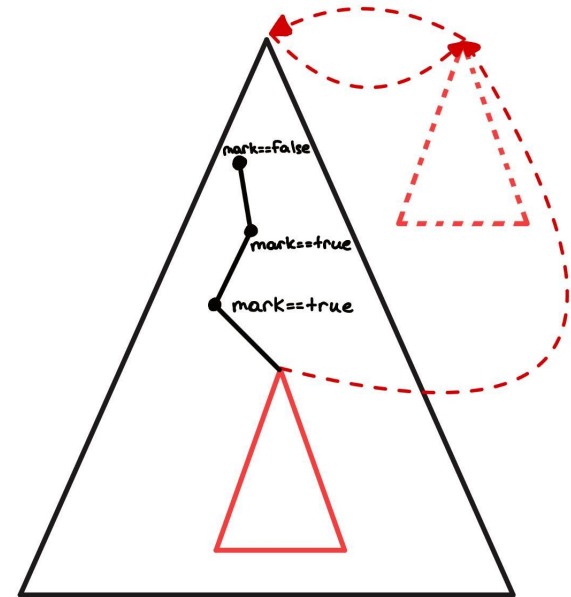
DecreaseKey

- 1) Требование кучи в дереве, в котором находится вершина не нарушено. Тогда операция завершается.
- 2) Требование нарушено. В таком случае, поддереву с корнем в этой вершине вырезается и добавляется в список корней. Дальше есть два варианта развития событий.



DecreaseKey

- 1) У родителя вершины `mark` выставлен в `false`. Тогда мы просто выставляем его в `true` и завершаем операцию.
- 2) У родителя вершины `mark` выставлен в `true`. Тогда мы тем же способом вырезаем поддерево родителя и вставляем его в список корней. Если у родителя `mark == false`, то завершаем операцию, иначе повторяем пункт 2.





Анализ сложности операций

Докажем методом бухчёта, что амортизированная сложность `DecreaseKey` - $O^*(1)$.

Каждому корню в куче приписываем 1 монету, помеченному узлу 2 монеты, остальным 0. Очевидно, что если не происходит каскадного вырезания (цепочка родителей с `mark = true`), то операция выполняется за константное время. Иначе на каждом новом шаге вырезания мы получаем 2 монеты за то, что `mark = true` снимается, тратим 1 на перекидывание узла в список корней и ещё одну на то, чтобы сделать вершину корнем. Таким образом выходим в 0 на каждой итерации каскадного вырезания, что означает, что амортизированная сложность `DecreaseKey` - $O^*(1)$.



Анализ сложности операций

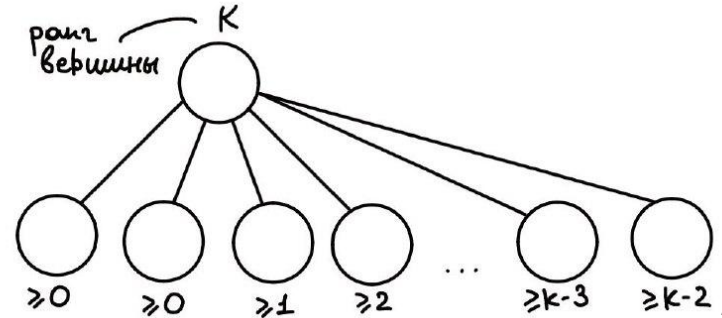
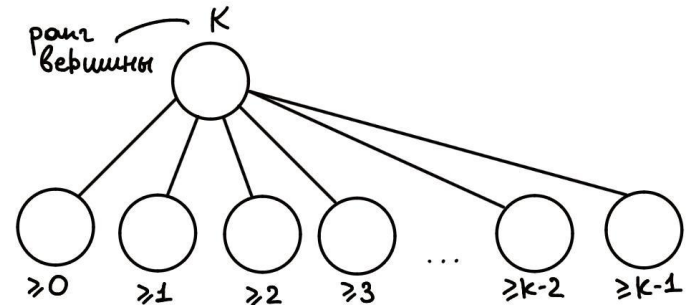
Теперь мы наконец-то узнаем почему структура данных имеет название Фибоначчиева куча. Докажем, что асимптотическая сложность ExtractMin — $O(\log N)$.

Понятно, что сложность ExtractMin это сложность consolidate . Так как привязывание вершины к списку корней выполняется за константу. Амортизационная сложность consolidate — $O^*(\text{количество корней} + D(N))$. Докажем, что $O^*(\text{количество корней}) = 0$. Также каждому корню в куче приписываем 1 монету, помеченному узлу 2 монеты, остальным 0. При слиянии двух деревьев мы тратим по одной монете на каждое и еще две на новый корень. А потратить мы должны 4. Таким образом вышли в ноль. То есть надо лишь доказать, что $O(D(N)) = O(\log N)$.

Анализ сложности операций

Пусть $S(k)$ — минимальное число вершин в дереве ранга k . В момент слияния дерево имеет вид как на картинке сверху. После некоторого времени, могли произойти удаления из поддеревьев, но т.к. у вершины может быть удален только один ребенок, минимальный случай — картинка снизу. Тогда имеем следующую формулу:

$$S(k) = 2 + \sum_{i=0}^{k-2} S(i)$$





Анализ сложности операций

Имеем следующее

$$2 + \sum_{i=0}^{k-2} S(i) = 2 + \sum_{i=0}^{k-3} S(i) + S(k-2) = S(k-1) + S(k-2)$$

Получаем, что

$$S(k) = S(k-1) + S(k-2)$$

То есть $S(k)$ - это числа Фибоначчи!



Анализ сложности операций

Для n -го числа Фибоначчи есть следующая формула

$$F_n = \frac{\phi^n - (-\phi)^{-n}}{\sqrt{5}}$$

То, есть

$$F_n = O(\phi^n) \Rightarrow S(k) = O(\phi^k) \Rightarrow D(n) = k = S^{-1}(n) = O(\log_{\phi} k)$$

Где $n = S(k)$ - число вершин в дереве ранга k . Это и требовалось доказать.



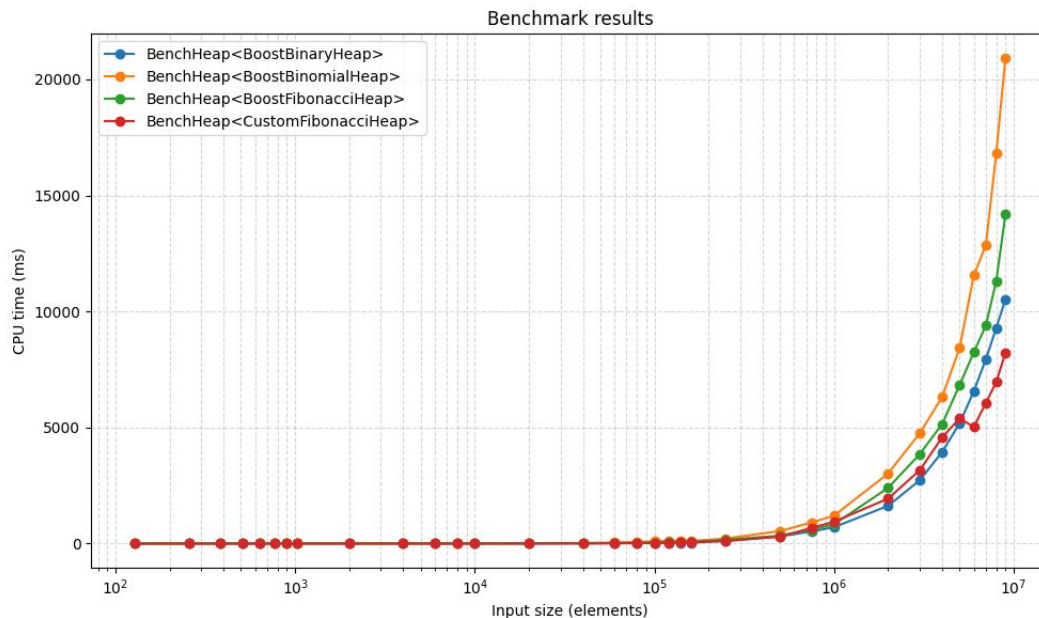
Сравнительный анализ

Сравним реализации бинарной, биномиальной и Фибоначчиевой куч из библиотеки boost и собственную реализацию Фибоначчиевой кучи. Для оценки реализаций проведем тест, который упрощенно симулирует логику алгоритма Дейкстры. Сначала в кучу будет добавляться много элементов. А потом в цикле делаться GetMin и ExtractMin. Размеры данных будем варьировать в пределах степени 10 - от 2 до 7.

Сравнительный анализ

Результаты теста можно наблюдать на графике. По оси X отложено начальное количество элементов в куче по логарифмической оси, а по оси Y время выполнения теста в миллисекундах.

Как видно с ростом количества элементов фибоначчиева куча является более предпочтительной.





Особенности реализации на вычислителях

Фибоначчиева куча демонстрирует прекрасные амортизированные оценки в теории, но на реальных вычислителях с реальными объемами данных ее производительность часто оказывается хуже, чем у биномиальной или двоичной кучи.

1. Cache-unfriendly. Структура кучи — множество узлов, разбросанных по памяти.
2. Большой размер узла — 21 дополнительных байт против 12 у биномиальной.
3. Аллокации на каждый узел.
4. Сложные операции со списками приводят к большим константам в асимптотике

Ее создатели писали в своей статье: “They are complicated when it comes to coding them. Also they are not as efficient in practice when compared with the theoretically less efficient forms of heaps, since in their simplest version they require storage and manipulation of four pointers per node, compared to the two or three pointers per node needed for other structures”.

Реализация Фибоначчиевой кучи

<https://github.com/sevaphasol/fibonacci-heap>

GitHub репозиторий с реализацией Фибоначчиевой кучи
и юнит тестами.





Список используемой литературы

- [1] [Приоритетная очередь на основе бинарной, биномиальной и фибонначиевой куч и ее применение в многоагентных поисковых системах](#)
- [2] [Фибонначиева куча. ИТМО.](#)
- [3] [Визуализатор Фибонначиевой кучи](#)
- [4] [Fredman, M. L.; Tarjan \(1987\). Fibonacci heaps and their uses in improved network optimization algorithms.](#)
- [5] [АиСД 7. Фибонначиева куча. Лекторий ФПМИ.](#)