Больше куч!

	binaryheap	*binomialheap*	*fibonacciheap*
insert	$O(\log n)$	$O(\log n)$	$O(1)$
extract_min	$O(\log n)$	$O(\log n)$	$\tilde{O}(\log n)$
decrease_key	$O(\log n)$	$O(\log n)$	$\tilde{O}(1)$
increase_key	$O(\log n)$	$O(\log n)$	$\tilde{O}(\log n)$
merge	$\tilde{O}(\log^2 n)$	$O(\log n)$	$O(1)$
get_min	$O(\log n)$ *or* $O(1)$	$O(1)$	

Биномиальная куча

Храним биномиальные деревья. Каждому дереву сопоставим ранг. Ранг дерева полностью определяет его структуру. Дерево ранга 0 — одна вершина. Дерево ранга 1 — одно ребро. В общем случае, дерево ранга n содержит корень и полное двоичное дерево размера 2^{n-1}. Дерево ранга $n+1$ — это два слитых вместе дерева ранга n — корень второго указывает на корень первого, а корень первого теперь будет указывать на оба бинарных дерева.

Биномиальная куча — это набор из логарифма биномиальных куч.

Слияние двух деревьев мы научились делать за $O(1)$ — меньшая вершина по ключу становится новым корнем, а дальше перекидываем указатели.

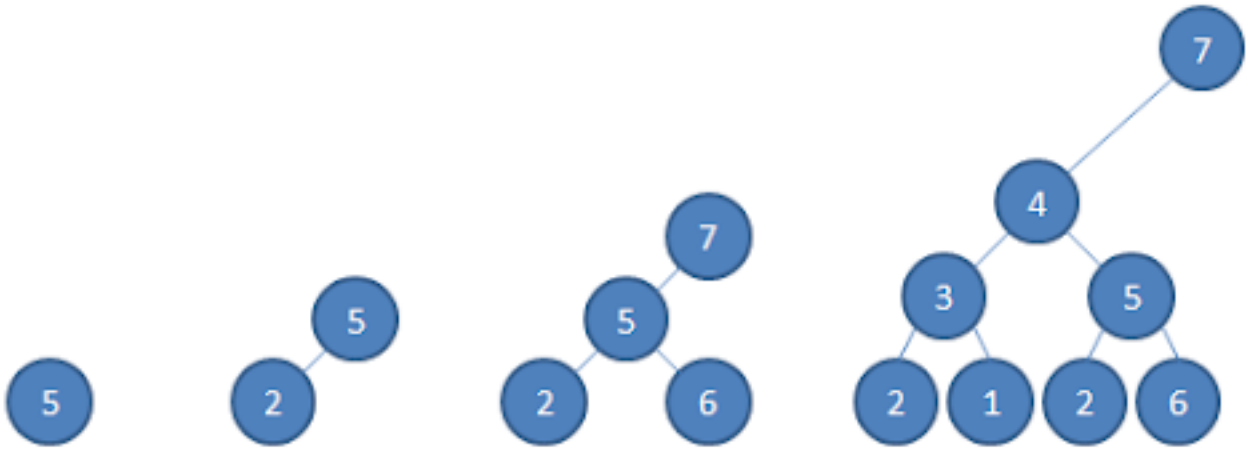
Слияние двух куч — это алгоритм сложения двоичных чисел — при сливании двух деревьев ранга n мы «переносим» прибавление кучи ранга $n+1$

Как добавлять элемент? Создать кучу на 0 элементов и слить их вместе.

Уменьшение ключа — напишем $sift\_up$ на нашей новой куче

Удаление минимума — Заметим, что если в правом дереве пройти по правым детям и обозначим их за корни, а их левых детей за полные бинарные деревья, то мы получим набор деревьев рангов $0, 1, \ldots, n-1$. Обозначим их за новую кучу, и сольем все вместе.

Увеличение ключа — удалим соответствующий элемент и добавим другой.



Фибоначчиева куча

Хотим сделать биномиальную кучу с послаблаблениями — делать операции в самый последний момент, менее четкую структуру, etc

Есть деревья, их корни храним в двусвязном закольцованном списке.

Всех детей для всех вершин храним в двусвязном закольцованном списке.

Новый ранг — это количество вершин в списке детей.

На каждом дереве выполнена куча, а также поддерживаем глобальный минимум.

Улучшение ключа делается так — удаляем вершину из своего списка, вместе с поддеревом. Добавляем в корневой список. Если мы удалили уже вторую вершину в поддереве родителя, то делаем каскадное вырезание — прыгаем по предкам с $mark = 1$, и вырезаем их в корневой список, причем все вырезания делаются по очереди.

Удаление делается так — мы приписываем всех детей к корневому списку, а потом вызываем *compact*, которая должна спасти наше дерево и навести порядок.

compact

- Сбрасываем пометки корневого списка в 0
- Переводим дерево в состояние, где все ранги разлиичны
- Храним ранги, мерджим одинаковые
- Как мерджим? Берем меньший корень, и записываем в его детей второй корень

Обозначим $R = \max rank$, $t(H) = root\ list\ size$, $m(H) = \sum_v mark(v)$
compact работает за $O(R + t(H))$.

Анализ времени работы $extract\_min$ &$increase\_key$ — $\tilde{O}(R)$.

$\Phi(H) = t(H) + 2m(H) < 3n$

Пусть каскадное вырезание сделало t_i действий. $m: 1 \to 0$, $t: +1$. Тогда $\Phi'(H) = \Phi(H) - 1$ за каждое вырезание. Тогда амортизированно вырезание работает за $O(1)$.

Compact:

$t(H) \le R$, $t'(H) = t(H) - R$

Амортизационно работает за $2R + 1$

Псевдокод тупых операций:

```
struct Node{
    Node *child;
    Node *left;
    Node *right;
    Node *parent;
    int rank;
    bool mark;
    int value;
}

list<Node *> roots;

void insert(int x) {
    Node *node = new Node(x);
    roots.insert(node);
}
```

```
void merge(list<Node *> a, list<Node *> b) {
    merge(a, b); // O(1) haha super easy
}

int getmin() {
    return argmin->value;
}
```