В. М. КОТОВ
Е. П. СОБОЛЕВСКАЯ
А. А. ТОЛСТИКОВ

«АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ»

ГЛАВА 3. СТРУКТУРЫ ДАННЫХ

Допущено
Министерством образования Республики Беларусь
в качестве учебного пособия для студентов
учреждения высшего образования
по специальностям «Прикладная математика»,
«Информатика», «Актуарная математика»

Минск
БГУ
2011

УДК 519.712(075.8)
ББК 22.18я73
К73

Рецензенты:
Кафедра математического и информационного обеспечения
Экономических систем ГГУ имени Янки Купалы
(заведующий кафедрой кандидат физико-математических наук, доцент
В.И. Ляликова);
доктор физико-математических наук, профессор С.В. Колосов

Котов, В.М.
К73 Алгоритмы и структуры данных: учеб. пособие / В.М. Котов, Е.П. Соболевская, А.А. Толстиков. – Минск : БГУ, 2011. – 267 с. – (Классическое университетское издание).
ISBN 978-985-518-530-8.

В учебном пособии изложены фундаментальные понятия, используемые при разработке алгоритмов и оценке их трудоемкости. Теоретический материал дополнен примерами и рисунками, облегчающими самостоятельное изучение материала, а также перечнем задач для самостоятельного решения. В приложении разбираются алгоритмы решения творческих задач повышенной сложности.

Предназначено для студентов высших учебных заведений, обучающихся по специальтностям «Прикладная математика», «Информатика», «Актуарная математика».

Структура данных куча

Биномиальные кучи

Определение. Биномиальный лес – это семейство биномиальных деревьев (обозначим биномиальное дерево высотой k через B_k). Биномиальное дерево высотой B_0 состоит из одной единственной вершины; биномиальное дерево B_k образуется присоединением биномиального дерева высотой $k-1$ к корню другого биномиального дерева высотой $k-1$ (во время присоединения корень одного из деревьев полагается одним из сыновей корня другого дерева).

Биномиальные деревья B_0, B_1, B_2 и B_3 показаны на рис. 1.

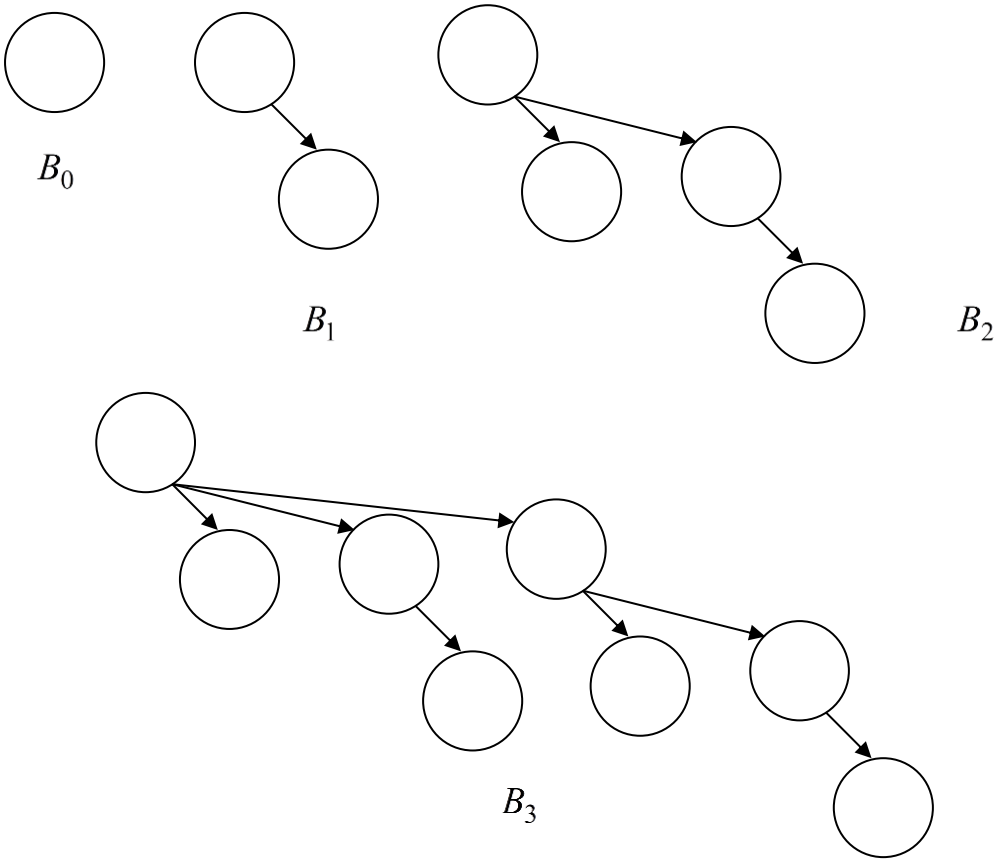

Рис.1

Название «биномиальное дерево» определил тот факт, что у любого биномиального дерева высотой *h* на глубине *d* находится ровно C_h^d вершин, где C_h^d – биномиальный коэффициент:

$$(x+a)^h = x^h + C_h^1 x^{h-1} a + \ldots + C_h^k x^{h-k} a^k + C_h^k x a^{h-1} + a^h .$$

Так, для биномиального дерева B_3, изображенного на рис. 3.22, справедливо:

$$d = 3; \quad C_3^1 = 3;$$
$$d = 2; \quad C_3^2 = 3;$$
$$d = 3; \quad C_3^3 = 1.$$

Для корня биномиального дерева B_k сыновьями являются биномиальные деревья B_0, B_1, B_2,…, B_{k-1}. Биномиальное дерево B_k содержит ровно 2^k вершин. Если в биномиальном дереве *m* вершин, то его высота $h = \log_2 m$.

Напомним, что *ранг вершины* определяется как количество ее сыновей, ранг дерева равен рангу корневой вершины. Для биномиального дерева его ранг совпадает с высотой дерева.

Определение. Биномиальная куча – это биномиальный лес, для которого выполняются следующие свойства (*инварианты*):

1) (*инвариант 1*): каждая вершина удовлетворяет основному свойству кучи: приоритет отца не ниже приоритета каждого из его сыновей;

2) (*инвариант 2*): в семействе биномиальных деревьев нет двух деревьев с корнями одинакового ранга.

Любая последовательность из *n* элементов может быть представлена единственным образом как семейство биномиальных деревьев, в котором не более одного дерева каждого ранга (каждому элементу соответствует вершина в биномиальном дереве). Предположим, что $n = 13$. Переведем это число в двоичную систему счисления:

$$13 = 2^3 + 2^2 + 2^0 .$$

Следовательно, для $n = 13$ деревья B_0, B_2, B_3 присутствуют в представлении, а дерево B_1 – нет.

Предположим, что в биномиальной куче *k* деревьев. Тогда минимальное количество вершин $n_{\min}$ в таком лесу:

$$n_{\min} = 2^{k-1} + 2^{k-2} + \cdots + 2^0 = \frac{2^k - 1}{2-1} = 2^k - 1.$$

Для любой другой биномиальной кучи из k деревьев количество вершин n в ней:

$$n \geq n_{\min} = 2^k - 1,$$
$$k \leq \log_2(n-1).$$

Следовательно, количество биномиальных деревьев в биномиальной куче есть $O(\log n)$, где n – количество элементов в куче.

Рассмотрим семейство корневых деревьев (каждой вершине дерева поставлено в соответствие некоторое ключевое значение) и для него определим следующие процедуры:

- процедура *link*(*i*, *j*) состоит в объединении двух деревьев одинакового ранга, корнями которых являются вершины *i* и *j*; при объединении корень дерева с большим ключевым значением становится одним из сыновей корня дерева с меньшим ключевым значением;
- процедура *cut*(*i*) состоит в том, что поддерево с корнем в вершине *i* удаляется из исходного дерева, образуя при этом самостоятельное дерево с корнем в вершине *i*.

Основные операции с биномиальными кучами

1. *Поиск минимального элемента в биномиальной куче*

Эта процедура требует проверки корневых значений всех биномиальных деревьев кучи. Для эффективной реализации данной процедуры необходимо, чтобы структура данных, используемая для хранения биномиальной кучи, обеспечивала быстрый доступ к корневым значениям всех биномиальных деревьев (например, корневые вершины всех биномиальных деревьев объединены в связный список).

Поскольку для биномиальной кучи из n элементов количество деревьев порядка $O(\log n)$, то трудоемкость алгоритма поиска минимального элемента есть $O(\log n)$.

Если хранить указатель на корень дерева с минимальным ключевым значением, то трудоемкость алгоритма поиска минимального элемента есть $O(1)$.

2. *Объединение двух биномиальных куч*

Объединение предполагает получение из двух биномиальных куч H_1 и H_2 новой биномиальной кучи H_3. Будем предполагать, что мы поддерживаем для каждой биномиальной кучи упорядоченность биномиальных деревьев по их рангам, что позволит выполнить процесс объединения более эффективно.

Рассмотрим процесс обединения двух куч, представленных на рис. 2.

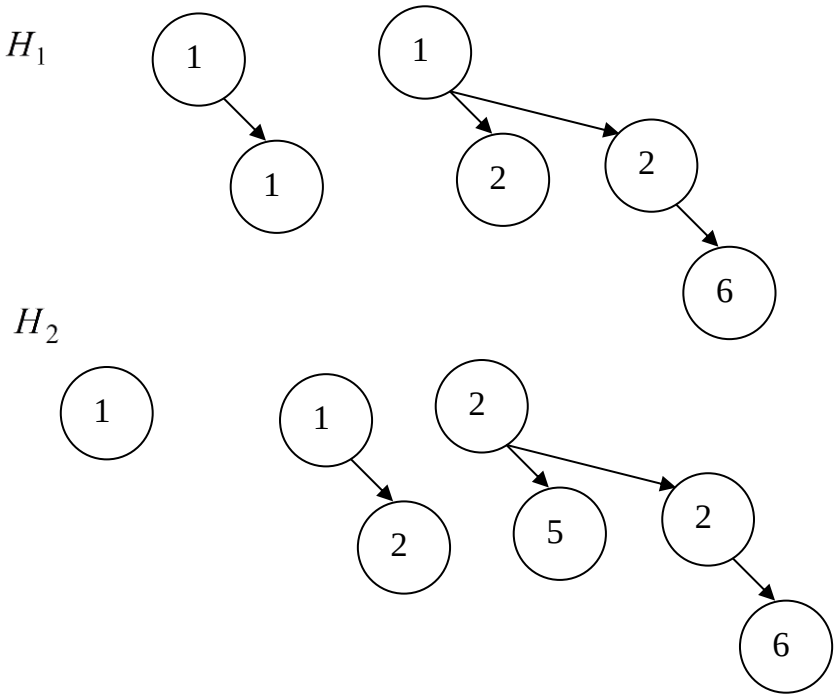

Рис. 2

Биномиальная куча H_2 содержит дерево ранга 0, а биномиальная куча H_1 – нет, следовательно, добавляем это дерево в кучу H_3 (рис. 3).

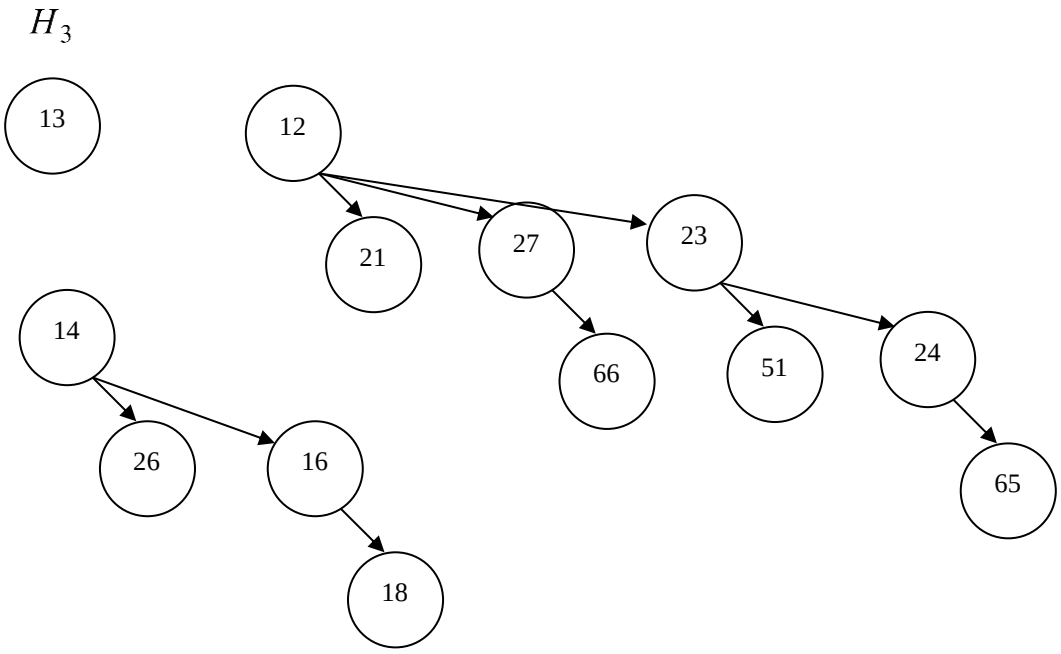

Рис. 3

Обе биномиальные кучи H_1 и H_2 содержат деревья ранга 1 (нарушение *инварианта 2*), следовательно, объединяем эти деревья в $link(14,16)$ (поддержка *инварианта 1*). В результате получаем биномиальное дерево ранга $h=2$. Таким образом, новая куча не будет иметь биномиального дерева ранга 1. Имеется три биномиальных дерева высотой $h=2$: по одному в H_1 и H_2, а также дерево, полученное на предыдущем этапе объединения. Одно из деревьев с корнем 14 помещаем в кучу H_3, а два других объединяем в $link(12, 23)$, получая при этом дерево ранга $h=3$. Поскольку ни одна из куч H_1 и H_2 не имеет дерева ранга $h=3$, то помещаем полученное дерево в кучу H_3 и завершаем процесс объединения (рис. 3).

Для эффективного выполнения процедуры объединения двух биномиальных куч необходимо, чтобы структура данных, разработанная для хранения кучи, имела следующий вид (рис. 4):

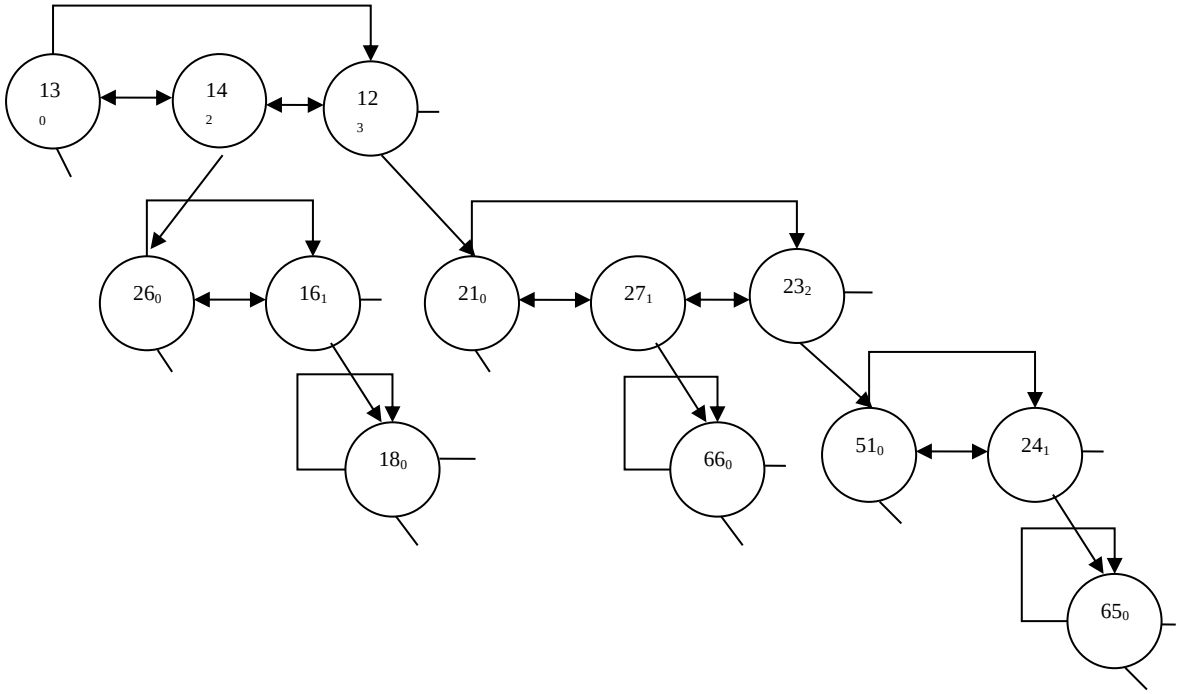

Рис. 4

- каждая вершина содержит информацию о ее ранге;
- биномиальные деревья упорядочены по рангам;
- каждая вершина имеет указатель на первого сына (*nil*, если сыновей нет), указатели на левого и правого братьев;
- сыновья каждой вершины представлены в виде двунаправленного списка и упорядочены по рангам их поддеревьев; указатель левого брата первого сына есть последний сын; указатель правого брата равен *nil*, если правого брата нет;
- указатель на биномиальную кучу – указатель на корень дерева минимального ранга.

На рис. 4 приведено представление биномиальной кучи, изображенной на рис. 3. Нижним индексом значения ключа каждой вершины является ее ранг. Линия без стрелки, отходящая от вершины, соответствует указателю *nil*.

С учетом разработанной структуры данных время, необходимое для объединения двух биномиальных деревьев, равно константе. Поскольку в биномиальной куче из n элементов существует порядка $O(\log n)$ биномиальных деревьев, то трудоемкость процедуры слияния двух биномиальных куч есть $O(\log n)$.

3. *Добавление нового элемента в биномиальную кучу*

Очевидно, что данная процедура может рассматриваться как частный случай процедуры объединения двух биномиальных куч. Создадим сначала новую кучу, состоящую из единственного биномиального дерева B_0 (ключ вершины – значение добавляемого элемента), а затем выполним процедуру объединения кучи, состоящей из единственного биномиального дерева B_0, и исходной кучи. Трудоемкость процедуры добавления нового элемента в кучу есть $\mathrm{O}(\log n)$.

4. *Создание биномиальной кучи*

Предположим, что нам необходимо создать биномиальную кучу из n элементов. Одним из возможных вариантов создания кучи является вызов n раз процедуры добавления элемента в кучу. В этом случае трудоемкость процедуры создания кучи есть $\mathrm{O}(n)$, хотя следовало бы ожидать оценку порядка $\mathrm{O}(n\log n)$. Докажем это утверждение.

Рассмотрим последовательность из n операций добавления нового элемента. Предположим, что в куче было z_0 биномиальных деревьев (если в куче деревьев не было, то $z_0 = 0$). Обозначим через t_i число процедур *link*, выполненных на i-й операции добавления нового элемента. После выполнения очередной итерации добавления нового элемента количество деревьев в биномиальной куче сначала увеличится на 1 (образуется новое биномиальное дерево высотой 0, ключевое значение корня которого равно значению добавляемого элемента), а затем в результате поддержки *инварианта 2* количество деревьев уменьшится на число, равное количеству выполненных процедур *link* .

После первой операции добавления количество деревьев в семействе станет равным

$$z_0 + 1 - t_1.$$

После второй операции добавления количество деревьев в семействе станет равным

$$z_0 + 2 - (t_1 + t_2).$$

После n-й операции добавления количество деревьев в семействе станет равным

$$z_0 + n - \sum_{i=1}^{n} t_i \ .$$

Поскольку не может остаться менее, чем одно дерево, то справедливо следующее неравенство:

$$z_0 + n - \sum_{i=1}^{n} t_i \geq 1.$$

Следовательно,

$$\sum_{i=1}^{n} t_i \leq z_0 + n - 1.$$

Из последнего неравенства следует, что трудоемкость всех операций слияния, выполненных за *n* шагов, есть $O(n)$ (при условии, что процедура *link* выполняется за время $O(1)$). Трудоемкость всех процедур создания биномиальных деревьев $B_0 - O(n)$. Следовательно, трудоемкость процедуры создания биномиальной кучи из *n* элементов равна $O(n)$.

Оценим **усреднённую оценку** одной операции добавления элемента в биномиальную кучу. Для этого выполним *k* раз операцию добавления элемента, подсчитаем суммарное затраченное время и разделим на число выполненных операций добавления, т.е. на *k*. За *k* шагов было потрачено время $O(k)$ на создание биномиальных деревьев высоты 0. Как было доказано выше, суммарное время, потраченное за *k* шагов на все операции слияния равно $O(k)$ (при условии, что процедура *link* выполняется за время $O(1)$). Таким образом получаем, что усреднённая оценка одной операции добавления элемента в биномиальную кучу:

$$\frac{O(k)+O(k)}{k} = O(1).$$

5. *Удаление минимального элемента из биномиальной кучи*

Будем предполагать, что биномиальная куча реализована в виде структуры, описанной ранее для операции объединения двух биномиальных куч.

- Для выполнения процедуры удаления минимального элемента необходимо сначала найти биномиальное дерево с минимальным корневым значением. Предположим, что это дерево B_k. Тогда трудоемкость этого

шага алгоритма есть $O(\log n)$; если хранить указатель на корень дерева с минимальным ключевым значением, то $O(1)$.

- Удаляем дерево B_k из кучи H, формируя новую биномиальную кучу H' (куча H без дерева B_k). Трудоемкость этого шага алгоритма $O(1)$.
- Выполняем серию операций *cut* к сыновьям корня дерева B_k, в результате чего бинарное дерево B_k распадается на деревья B_0, B_1, B_2,…, B_{k-1}, из которых мы образуем новую биномиальную кучу H''. Поскольку для каждого узла храним ссылки на первого сына, левого и правого брата, то трудоемкость этого шага алгоритма равна $O(1)$.
- Выполняем операцию слияния куч H' и H''. Трудоемкость слияния двух куч есть $O(\log n)$, где n – общее количество элементов в двух кучах.

Таким образом, трудоемкость процедуры удаления минимального элемента из биномиальной кучи есть $O(\log n)$.

На рис. 5 показана биномиальная куча, полученная из кучи H_3, изображенной на рис. 3, в результате выполнения операции удаления минимального элемента (удаляется элемент с ключом 12).

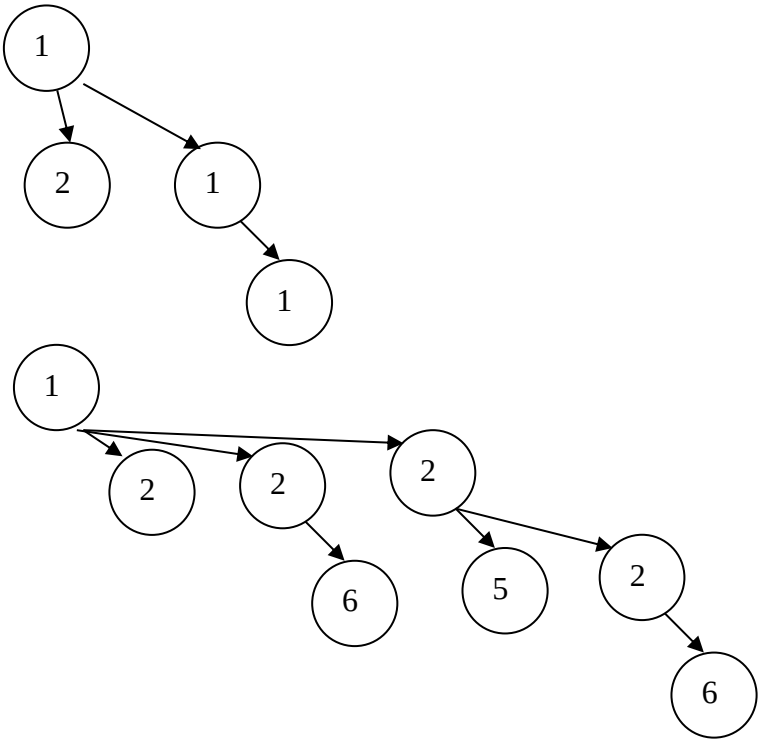

Рис. 5

Приведем псевдокод функции слияния двух биномиальных деревьев merge_tree (T_1, T_2).

function merge_tree (T_1, T_2 : *Priority_Queue*) : *Priority_Queue*;

{ если $T_1.element > T_2.element$ то *swap* (T_1, T_2);

если $T_1.rank = 0$ то $T_1.f\_child = T_2$;

иначе

{ $T_2.l\_sib = T_1.f\_child.l\_sib$

$T_2.l\_sib.r\_sib = T_2, T_1.f\_child.l\_sib = T_2$}

$T_1.rank = T_1.rank + 1$

merge_tree $= T_1$}

Приведем псевдокод процедуры слияния двух биномиальных куч merge (H_1, H_2), которая использует функцию слияния двух биномиальных деревьев merge_tree (T_1, T_2) и предполагает, что существует такая функция extract(*T*, *H*), которая удаляет дерево T из биномиальной кучи *H*.

function merge (H_1, H_2 : *Priority_Queue*) : *Priority_Queue*;

{ если $H_1 = null$ то merge $= H_2$

иначе если $H_2 = null$ то merge $= H_1$

иначе если $H_1\text{^}.rank < H_2\text{^}.rank$ то

{ extract (T_1, H_1)

H_3 = merge (H_1, H_2), $T_1.l\_sib = H_3.l\_sib$

$H_3.l\_sib.r\_sib = null$, $T_1.r\_sib = H_3$

$H_3.l\_sib = T_1$, merge $= T_1$}

иначе если если $H_2\text{^}.rank < H_1\text{^}.rank$ то

merge = merge (H_2, H_1)

иначе

{ extract (T_1, H_1), extract (T_2, H_2)

H_3 = merge (H_1, H_2)

T_3 = merge_tree (T_1, T_2)

merge=merge(H_3, T_3)}

}

Кучи Фибоначчи

Название «кучи Фибоначчи» обусловлено тем, что для доказательства оценок трудоемкости операций используются числа Фибоначчи. Напомним, что последовательность чисел Фибоначчи определяется следующим образом:

$$\begin{cases} F(k) = F(k-1) + F(k-2), & k \geq 3, \\ F(1) = F(2) = 1. \end{cases}$$

Свойство 1. Для последовательности чисел Фибоначчи ($k \geq 3$) справедливы следующие соотношения:

$$F(k) \geq 2^{\frac{k-1}{2}}; \qquad (1)$$

$$F(k) = 1 + F(1) + F(2) + \ldots + F(k-2). \qquad (2)$$

Доказательство. Докажем сначала соотношение (1).

По определению чисел Фибоначчи при $k \geq 3$

$$F(k) = F(k-1) + F(k-2),$$

и с учетом неравенства

$$F(k-1) \geq F(k-2)$$

получим

$$F(k) \geq 2 \cdot F(k-2). \qquad (3)$$

Если *k* – нечетно, то, используя неравенство (3), получим соотношение (1):

$$F(k) \geq 2 \cdot F(k-2 \cdot 1) \geq 2^2 \cdot F(k-2 \cdot 2) \geq \ldots \geq 2^z \cdot F(k-2 \cdot z) \geq$$

$$\geq \left[k - 2 \cdot z = 1,\ z = \frac{k-1}{2} \right] \geq 2^{\frac{k-1}{2}} \cdot F(1) = 2^{\frac{k-1}{2}}.$$

Если *k* – четно, то применим для доказательства метод математической индукции. Пусть $k = 4$. Тогда неравенство

$$F(4) \geq 2^{\frac{3}{2}}$$

выполняется, так как $F(4) = 4$.

Предположим, что соотношение (1) истинно для всех четных чисел, меньших *k*, и с учетом его верности для нечетных значений *k* имеем

$$F(k)=F(k-1)+F(k-2)\geq 2^{\frac{k-2}{2}}+2^{\frac{k-3}{2}}=2^{\frac{k-3}{2}}\cdot\left(2^{\frac{1}{2}}+1\right)\geq$$

$$\geq 2^{\frac{k-3}{2}}\cdot 2=2^{\frac{k-3}{2}+1}=2^{\frac{k-1}{2}}.$$

Следовательно, соотношение (1) выполняется и для нечетных значений *k*.

Докажем сейчас соотношение (2). Для доказательства будем использовать метод математической индукции. Проверим истинность равенства (2) для $k=3$. Получаем

$$F(3)=F(2)+F(1)=1+F(1),$$

следовательно, равенство (2) выполняется. Предположим, что

$$F(k-1)=1+F(1)+F(2)+\ldots+F(k-3),$$

и докажем верность соотношения (2) для *k*.

Действительно,

$$F(k)=F(k-1)+F(k-2)=1+F(1)+F(2)+\ldots+F(k-3)+F(k-2).$$

Поэтому соотношение (2) выполняется, и доказательство свойства 1 завершено.

Свойство 2. Если последовательность чисел $G(k)$ удовлетворяет соотношениям:

$$G(1)=G(2)=1,$$
$$G(k)\geq 1+G(1)+G(2)+\ldots+G(k-2),\quad k\geq 3,$$

то

$$G(k)\geq F(k). \qquad (4)$$

Доказательство. Проведем доказательство по индукции. Если $k=3$, то

$$G(3)\geq 1+G(1)=2=F(3).$$

Предположим, что $G(k)\geq F(k)$ для всех значений k до $q-1$ и докажем его верность для $k=q$. Действительно, с учетом равенства (2) получаем

$$G(q)\geq 1+G(1)+\ldots+G(q-2)\geq 1+F(1)+F(2)+\ldots+F(q-2)=F(q).$$

Следовательно, неравенство (4) выполняется, и доказательство свойства 2 завершено.

Определение. Куча Фибоначчи – это семейство корневых деревьев, для которого выполняются следующие свойства (инварианты):

1) (*инвариант 1*) каждая вершина в куче Фибоначчи удовлетворяет основному свойству кучи: приоритет отца не ниже приоритета каждого из его сыновей;

2) (*инвариант 2*) в семействе корневых деревьев нет двух деревьев с корнями одинакового ранга;

3) (*инвариант 3*) каждая некорневая вершина в куче Фибоначчи может потерять не более одного сына при выполнении процедуры *cut*.

Рассмотрим все деревья кучи ранга *k*: $T_1^k, T_2^k, \ldots, T_l^k$. Каждое из деревьев T_i^k содержит некоторое число вершин $G\left(T_i^k\right)$. Пусть

$$G(k) = \min_{1 \le i \le l} G\left(T_i^k\right) -$$

минимальное количество вершин дерева ранга *k*.

Свойство 3. Справедливо следующее неравенство: $G(k) \ge F(k),\ k \ge 1$.

Доказательство. Поскольку $G(1) = 2 \ge F(1)$ и $G(2) = 3 \ge F(2)$, то для $k = 1, 2$ свойство 3 верно. Пусть ω – вершина в куче Фибоначчи с рангом $k \ge 3$. Упорядочим сыновей вершины ω в том же порядке, в котором предыдущие операции присоединяли их к вершине ω (от ранних к поздним). Докажем, что ранг *i*-го сына вершины ω как минимум $(i-2)$.

Пусть y_i есть *i*-й сын вершины ω, и рассмотрим момент, когда вершина y_i присоединилась к вершине ω. Перед моментом присоедиения вершина ω должна была иметь как минимум $(i-1)$ сына (вершина ω могла ранее иметь и больше сыновей, но они были удалены при выполнении процедуры *cut*). Поскольку в момент присоединения корни y_i и ω имели один и тот же ранг, то вершина y_i должна была иметь как минимум $(i-1)$ сына. Заметим, что после присоединения вершины y_i к вершине *w* вершина y_i стала некорневой и могла потерять не более одного сына (*инвариант 3*). Следовательно, вершина y_i должна иметь как минимум $(i-2)$ сына.

Таким образом, поддерево с корнем в вершине *w* содержит как минимум

$$G(k) \ge 1 + G(1) + G(2) + \ldots + G(k-2), \quad k \ge 3,$$

вершин, и по свойству 2 заключаем, что $G(k) \geq F(k)$. Доказательство свойства 3 завершено.

Лемма 1. Максимально возможный ранг любого узла в куче Фибоначчи равен $2 \cdot \log_2 n + 1$.

Доказательство. Учитывая тот факт, что никакое поддерево не может содержать больше чем *n* узлов, и свойства 1, 2, 3, получаем следующие неравенства:

$$n \geq G(k) \geq F(k) \geq 2^{\frac{k-1}{2}}.$$

Логарифмируя, получим $2\log_2 n + 1 \geq k$.

Таким образом, максимально возможный ранг любого узла в куче Фибоначчи не превосходит $2\log_2 n + 1$.

Доказательство леммы завершено.

ТЕОРЕМА 1. Куча Фибоначчи состоит не более чем из $2\log_2 n + 2$ корневых деревьев.

Доказательство. В силу леммы 1 максимальный ранг корня любого дерева не превосходит величины $2 \cdot \log_2 n + 1$. Поскольку в куче Фибоначчи не существует двух деревьев одинакового ранга, то остальные деревья могут иметь ранги $2\log_2 n, 2\log_2 n - 1, \ldots, 1, 0$.

Следовательно, куча Фибоначчи состоит не более чем из $2\log_2 n + 2$ корневых деревьев. Доказательство теоремы завершено.

Сейчас опишем основные операции для кучи Фибоначчи и покажем, что для их реализации достаточно использовать последовательность процедур *link* и *cut.*

Для выполнения этих процедур необходимо для каждой вершины *i* дерева хранить следующую информацию:

- отец вершины: *pred*(*i*);
- ключевое значение вершины: *key*(*i*) (в дальнейшем будем предполагать, что $key(i) = i$;
- множество сыновей (это множество может быть задано в виде двунаправленного списка);
- ранг вершины: $rank(i)$.

Также нам понадобятся следующие данные:

- указатель на корень дерева с минимальным ключевым значением *minkey*;

- массив *bucket* (от 0 до $2\log_2 n+1$); *k*-й элемент массива равен корню дерева ранга *k* в куче Фибоначчи; если в куче не существует дерева ранга *k*, то $bucket(k)=0$;

- вспомогательный массив $list$ (от 0 до $2\log_2 n+1$); $list(k)$ – список корней деревьев ранга *k*;

для каждой вершины *i* дерева зададим величину $lost(i)$, которая будет принимать одно из трех значений: 0, 1 или 2.

Свойство 4. Процедуры *link* и *cut* требуют $O(1)$ времени для выполнения.

Основные операции с кучей Фибоначчи

1. *Поиск минимального элемента в куче Фибоначчи*

Поскольку мы храним указатель на корень дерева с минимальным ключевым значением *minkey*, то трудоемкость поиска минимального элемента равна $O(1)$.

2. *Добавление нового элемента с ключом x в кучу Фибоначчи*

Для выполнения этой операции сначала формируем новое корневое дерево, состоящее из единственной вершины с ключом *x*. После этого может быть нарушен *инвариант 2*, и для его восстановления может потребоваться последовательность процедур *link* (см. далее процедуру восстановления *инварианта 2*). Корректируем, если необходимо, значение *minkey*. Трудоемкость операции добавления нового элемента равна трудоемкости восстановления *инварианта 2*.

3. *Удаление минимального элемента из кучи Фибоначчи*

Удаляем с помощью *начальных процедур cut* всех сыновей корня дерева, на которое указывает значение *minkey* (каждый из сыновей *x* корня *minkey* порождает новое дерево семейства с корнем в вершине *x*). Восстанавливаем, если необходимо, *инвариант 2*. Следует заметить, что так как исходные процедуры *cut* применялись для сыновей корня дерева, то *инвариант 3* не нарушается (корневая вершина может терять в результате выполнения процедур *cut* любое количество сыновей). Поскольку трудоемкость выполнения всех исходных процедур *cut* равна $O(\log n)$, то трудоемкость операции удаления равна трудоемкости восстановления *инварианта 2* плюс $O(\log n)$.

4. *Уменьшение значения ключа вершины в куче Фибоначчи*

После того как мы уменьшили ключ вершины *i*, каждая вершина в поддереве с корнем в вершине *i* удовлетворяет *инварианту 1*, но для вершины

$j = pred(i)$ (отца вершины i) этот инвариант может нарушиться. Если это произошло, то мы выполняем одну *начальную процедуру cut*(i), модифицируем, если необходимо, значение *minkey*. Выполнение начальной процедуры *cut*(i) могло привести к нарушению *инварианта* 3 для вершины j, если это так, то восстанавливаем *инвариант 3*. После того, как восстановлен *инвариант 3*, может нарушиться *инвариант 2* (для восстановления *инварианта 3* выполнялась серия *порожденных операций cut*), если это так, то восстанавливаем его. Трудоемкость операции уменьшения ключа пропорциональна количеству выполненных процедур *cut*, необходимых для поддержки *инварианта 3*, плюс трудоемкость восстановления *инварианта 2*.

5. *Увеличение значения ключа вершины в куче Фибоначчи*

После того как мы увеличили ключ вершины i, для отца $pred(i)$ вершины i *инвариант 1* выполняется, но для некоторых сыновей j вершины i этот инвариант может нарушиться. Если это произошло, то мы выполняем *исходные процедуры* $cut(j)$ к этим сыновьям. Выполнение *начальных процедур* $cut(j)$ могло привести к нарушению *инварианта 3* для вершины i. Восстанавливаем *инвариант 3*. После того как восстановлен *инвариант 3* (для восстановлении *инварианта 3* выполнялась серия *порожденных операций cut*), может нарушиться *инвариант 2*, если это так, то восстанавливаем его.

Трудоемкость операции увеличения ключа пропорциональна количеству *начальных процедур cut* (их количество $O(\log n)$) плюс количество *порожденных процедур cut*, необходимых для поддержки *инварианта 3*, и плюс трудоемкость восстановления *инварианта 2*.

Опишем теперь последовательность действий, которые необходимо выполнить, чтобы восстановить инварианты кучи Фибоначчи.

Восстановление инварианта 2

(никакие две корневые вершины в куче Фибоначчи
не должны иметь одинакового ранга)

Если *инвариант 2* нарушился при выполнении процедуры *cut*, то занесем в массив *list* указатели на корни всех деревьев, полученных в результате выполнения *начальных* и *порожденных процедур cut*. Записывая корни в массив *list*, будем заботиться о том, чтобы каждый элемент списка массива *list* содержал не более одного элемента. Для этого, например, можно поступить следующим образом: если мы хотим занести в массив *list* корень дерева x ранга k, а список $list(k)$ не пуст, то выполняем

процедуру *link* для дерева с корнем x и дерева, на которое указывает значение $list(k)$. После чего список $list(k)$ полагаем пустым, а корень полученного в результате слияния дерева ранга $k+1$ пытаемся добавить в список $list(k+1)$ и т. д., пока не найдем подходящий пустой список. Поскольку каждая процедура *link* уменьшает количество деревьев на 1, то трудоемкость формирования массива *list* пропорциональна количеству *начальных* и *порожденных* процедур *cut*. Заметим, что если нарушение инварианта произошло при выполнении процедуры удаления минимального элемента, то у нас только *начальные процедуры cut* и их количество не превосходит $O(\log n)$. Если же нарушение инварианта произошло при выполнении операций уменьшения (увеличения) ключа, то у нас будут как *начальные*, так и *порожденные процедуры cut*, и об их количестве мы пока ничего сказать не можем.

Если *инвариант 2* нарушился при выполнении операции добавления нового элемента x в кучу Фибоначчи, то занесем в массив *list* единственный элемент, который соответствует корню дерева ранга нуль с ключевым значением x. Теперь у нас есть два массива: *list* и *bucket* размерности $O(\log n)$, для каждого из которых выполняется *инвариант 2*. Выполним слияние этих массивов в массив *bucket* таким образом, что для него также будет выполнен *инвариант 2*. Это можно сделать, например, используя следующую процедуру.

```
l =0
для k = 0 до (2 · log n +1)
  если l = 0 то
    {если (bucket(k) = 0) and (list(k) ≠ 0) то bucket(k) = list(k)
     если (bucket(k) ≠ 0) and (list(k) ≠ 0) то
                    { bucket(k) = 0, l = link(bucket(k), list(k)) }
     }
     иначе
            { если (bucket(k) = 0) and (list(k) = 0) то
                    {bucket(k) = l, l = 0}
              если (bucket(k) ≠ 0) and (list(k) = 0) то
                     {bucket(k) = 0 , l = link(l, bucket(k))}
              если list(k) ≠ 0 то l = link(l, list(k))
            }
```

Трудоемкость слияния двух массивов пропорциональна их размерностям и равна $O(\log n)$.

ТЕОРЕМА 2. Для основных операций с кучей Фибоначчи справедливы следующие оценки трудоемкости в худшем случае:

1) поиск минимального элемента – $O(1)$;
2) добавление нового элемента – $O(\log n)$;
3) удаление минимального элемента – $O(\log n)$.

Восстановление инварианта 3

(каждая некорневая вершина в куче Фибоначчи
может потерять не более одного сына
при выполнении процедуры *cut*)

Инвариант 3 мог нарушиться при выполнении операции уменьшения (увеличения) ключа некоторой вершины. Для восстановления этого инварианта будем хранить для каждой вершины i дополнительный индекс $lost(i)$:

- для корневой вершины $lost(i) = 0$;
- для некорневой вершины значение $lost(i)$ равено числу сыновей, которые вершина i потеряла после того, как стала некорневой вершиной.

Если $lost(i) = 2$, то к вершине i применим *порожденную процедуру cut*(i). В свою очередь, *порожденная процедура cut*(i) может породить еще одну *порожденную процедуру cut*(*pred*(i)) для отца вершины i, если $lost(pred(i)) = 2$, и т. д. до тех пор, пока не будет выполнен *инвариант 3*.

Поскольку трудоемкость выполнения одной операции *cut* равна $O(1)$, то трудоемкость восстановления *инварианта 3* пропорциональна количеству *порожденных операций cut*.

Свойство 5. Общее число *порожденных операций cut* не превышает общего числа *начальных операций cut*.

Доказательство. Предположим, что мы выполнили некоторое число *начальных операций cut*, а они привели нас к выполнению *порожденных операций cut*. Рассмотрим функцию

$$\Phi = \sum_i lost(i).$$

Предположим, что мы выполнили *cut*(i) и j – отец вершины i. Операция *cut*(i) устанавливает *lost*(i) в нуль и увеличивает *lost*(j) на единицу, если j не корень. Если мы рассматриваем *начальную операцию cut*(i), то *lost*(i) равно нулю или единице до операции, а если это порожденная *cut*(i), то

lost(*i*) равно двум до операции. Поэтому *начальная операция cut*(*i*) увеличивает $lost(i)+lost(j)$ (а, следовательно, и функцию Ф) как максимум на единицу, в то время как *порожденная операция cut*(*i*) уменьшает значение $lost(i)+lost(j)$ как минимум на единицу. Если мы начнем со значения $\Phi = 0$, то общее число уменьшений функции ограничим общим числом увеличений. Доказательство свойства 5 завершено.

Существует зависимость между числом процедур *link* и *cut* при работе со структурой данных *куча Фибоначчи*.

Свойство 6. Число процедур *link* равно, как максимум, *m* плюс число процедур *cut*, где *m* – начальное число корневых деревьев.

Доказательство. Рассмотрим *функцию потенциалов*, определяемую как количество корневых деревьев. Каждая процедура *link* уменьшает число корневых деревьев на 1, и каждая процедура *cut* увеличивает их число на 1. Общее число уменьшений числа корневых деревьев в куче Фибоначчи ограничено начальным числом корневых деревьев (*m*) плюс общее число их увеличений. Доказательство свойства 6 завершено.

Таким образом, если количество *начальных процедур cut* не превосходит некоторого числа *k*, то количество *порожденных процедур cut* также не будет превосходить этого числа *k* (свойство 5). Следовательно, общее число всех процедур *cut* не превосходит величины $2 \cdot k$, и общее число процедур *link* не превосходит величины $m + 2 \cdot k$, где *m* – начальное число корневых деревьев (свойство 6).

Для оценки трудоемкости в худшем случае некоторой группы из *n* операций (например, *n* операций добавления элемента в кучу Фибоначчи) мы могли бы умножить максимальную длительность операции на общее число операций. В этом случае трудоемкость группы из *n* операций добавления нового элемента (удаления минимального элемента) была бы равна $O(n \log n)$.

Иногда удается получить более точную оценку времени выполнения последовательности операций (или, что равносильно, среднего времени выполнения одной операции: *усредненную оценку трудоемкости одной операции в худшем случае*). Оценки такого рода в литературе называются *амортизационным анализом* некоторой реализации для рассматриваемой структуры данных. Для вычисления усредненной оценки трудоемкости одной операции суммарная трудоемкость последовательности операций делится на количество операций, т. е. получаем время в расчете на одну операцию. Получим сейчас усредненные оценки трудоемкости для основных операций с кучей Фибоначчи.

ТЕОРЕМА 3. Для кучи Фибоначчи справедливы следующие усредненные оценки трудоемкости:

1) усредненная оценка трудоемкости операции добавления нового элемента для худшего случая равна $\mathrm{O}(1)$;

2) усредненная оценка трудоемкости операции уменьшения ключа некоторого элемента для худшего случая равна $\mathrm{O}(1)$;

3) усредненная оценка трудоемкости операции удаления минимального элемента для худшего случая равна $\mathrm{O}(\log n)$;

4) усредненная оценка трудоемкости операции увеличения ключа некоторого элемента для худшего случая равна $\mathrm{O}(\log n)$.

Доказательство. Предположим, что первоначально куча состояла из z_0 корневых деревьев.

1. *Добавление нового элемента*

Рассмотрим последовательность из *n* операций добавления элемента. Обозначим через t_i число процедур *link*, выполненных на *i*-й операции добавления нового элмента.

После выполнения первой операции добавления количество деревьев в куче Фибоначчи сначала увеличится на 1, а затем в результате поддержки *инварианта 2* уменьшится на число, равное количеству выполненных процедур:

$$z_0 + 1 - t_1.$$

После второй операции добавления количество деревьев в семействе станет равным

$$z_0 + 2 - (t_1 + t_2).$$

После *n*-й операции добавления количество деревьев в семействе станет равным

$$z_0 + n - \sum_{i=1}^{n} t_i .$$

Поскольку не может остаться менее чем одно дерево, то справедливо следующее неравенство:

$$z_0 + n - \sum_{i=1}^{n} t_i \geq 1.$$

Следовательно,

$$\sum_{i=1}^{n} t_i \leq z_0 + n - 1. \qquad (5)$$

Для получения усредненной оценки трудоемкости одной операции добавления нужно найти сумму трудоемкостей всех процедур формирования новых деревьев и всех процедур *link*, которые были выполнены за n операций добавления нового элемента, и разделить эту сумму на количество операций добавления ($=n$). Тогда с учетом неравенства (5) и свойства 4 получим

$$\frac{\mathrm{O}(n)+\sum_{i=1}^{n} t_i}{n} \le \frac{\mathrm{O}(n)+z_0+n-1}{n} = \mathrm{O}(1).$$

2. *Уменьшение значения ключа для некоторого элемента*

После завершения n-й операции уменьшения ключа будут выполнены: n *начальных процедур cut*, n' *порожденных процедур cut* (поддержка *инварианта 3*) и y процедур *link* (восстановление *инварианта 2*).

В силу свойств 5 и 6 справедливы следующие неравенства:

$$n' \le n; \quad y \le z_0 + n + n' \le z_0 + 2 \cdot n. \qquad (6)$$

С учетом неравенств (6) и свойства 4 получим, что усредненная оценка одной операции уменьшения ключа равна

$$\frac{n+n'+y}{n} \le \frac{4n+z_0}{n} = \mathrm{O}(1).$$

3. *Удаление минимального элемента* (*увеличение значения ключа для некоторого элемента*)

После выполнения n операций удаления минимального элемента (увеличение значения ключа для некоторого элемента) будет выполнено не более чем $n\log n$ *начальных процедур cut* (лемма 1), не более чем $n' \le n\log n$ *порожденных процедур cut* для поддержки *инварианта 3* (свойство 5) и $y \le z_0 + 2n\log n$ процедур *link* для поддержки *инварианта 2* (свойство 6). С учетом этих неравенств и свойства 4 получим, что усредненная оценка одной операции удаления минимального элемента (увеличения значения ключа для некоторого элемента) равна

$$\frac{n+n'+y}{n} \le \frac{n\log n + n\log n + 2n\log n + z_0}{n} = \mathrm{O}(\log n).$$

Доказательство теоремы завершено.

Рассмотрим сейчас на примере операцию добавления нового элемента в кучу Фибоначчи.

Пример. Добавить в кучу Фибоначчи, изображенную на рис. 6, элемент с ключом 10.

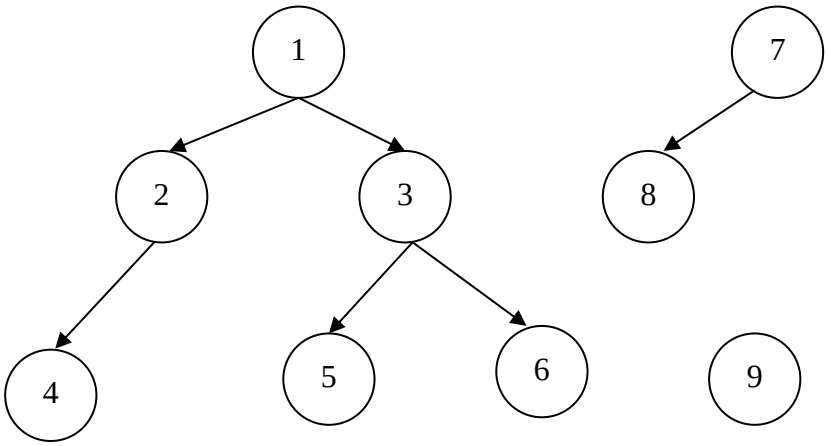

Рис. 3.6

Формируем корневое дерево нулевого ранга с ключевым значением корня, равным 10. Формируем список *list*. В результате получаем

i	**0**	**1**	**2**
bucket	9	7	1
list	10	*nil*	*nil*

Так как нарушен *инвариант 2*, то выполняем процедуру его восстановления. Для этого сначала будет выполнена процедура *link*(9, 10) (рис. 7).

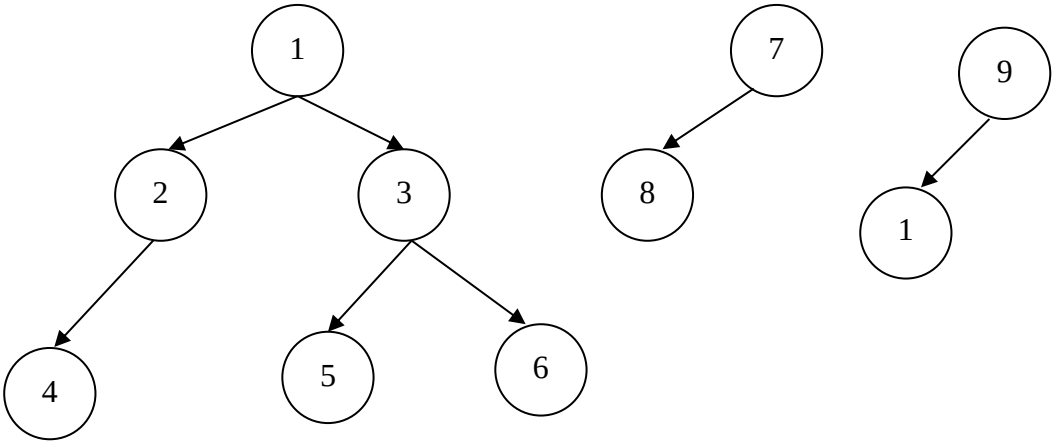

Рис. 7

Потом будет выполнена процедура *link*(7, 9) (рис. 8).

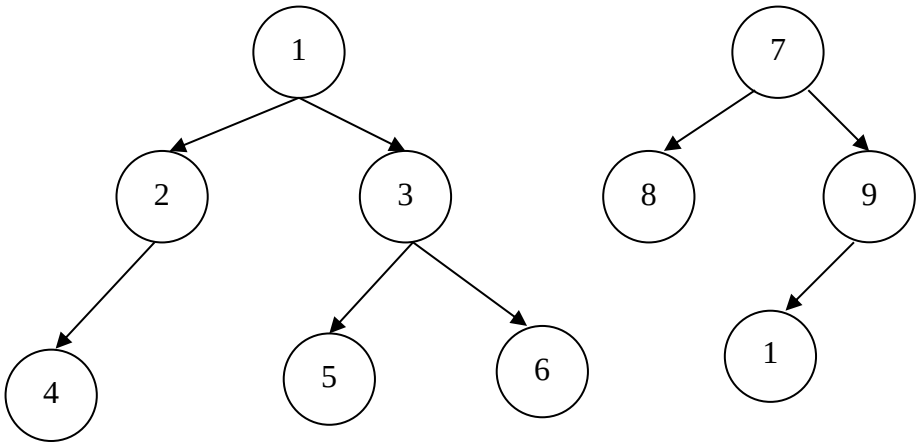

Рис. 8

И в конце будет выполнена процедура *link*(1, 7) (рис. 9).

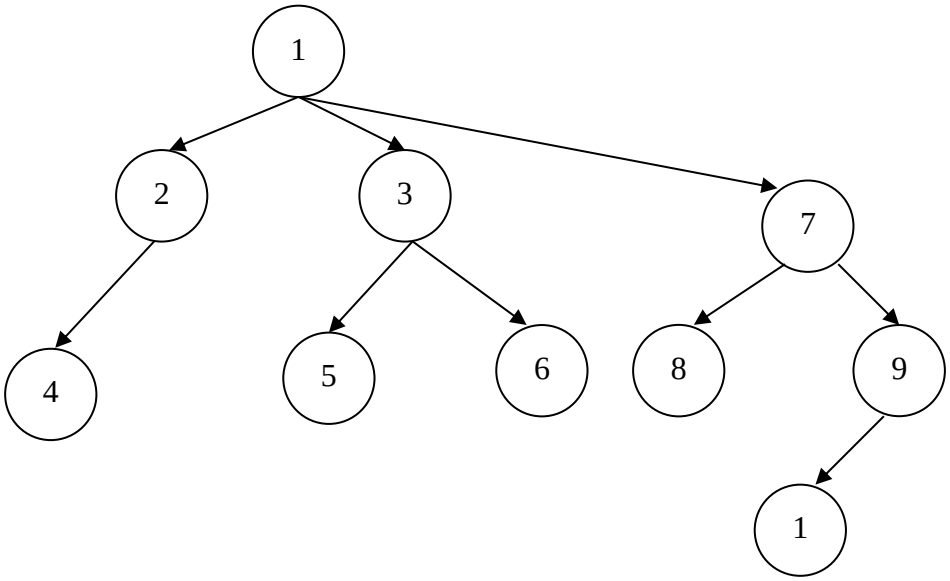

Рис. 9

Теперь *инвариант 2* выполнен, и операция добавления элемента завершена.

Пример. Рассмотрим кучу Фибоначчи, изображенную на рис. 10. Возле каждой вершины в квадратных скобках указано значение индекса *lost* (количество сыновей, которые потеряла вершина, когда стала некорневой).

Уменьшим значение ключа вершины $i = 24$ ($key(i) = 24$) на число 22. После выполнения операции получаем, что новое значение ключа вершины *i* равно 2.

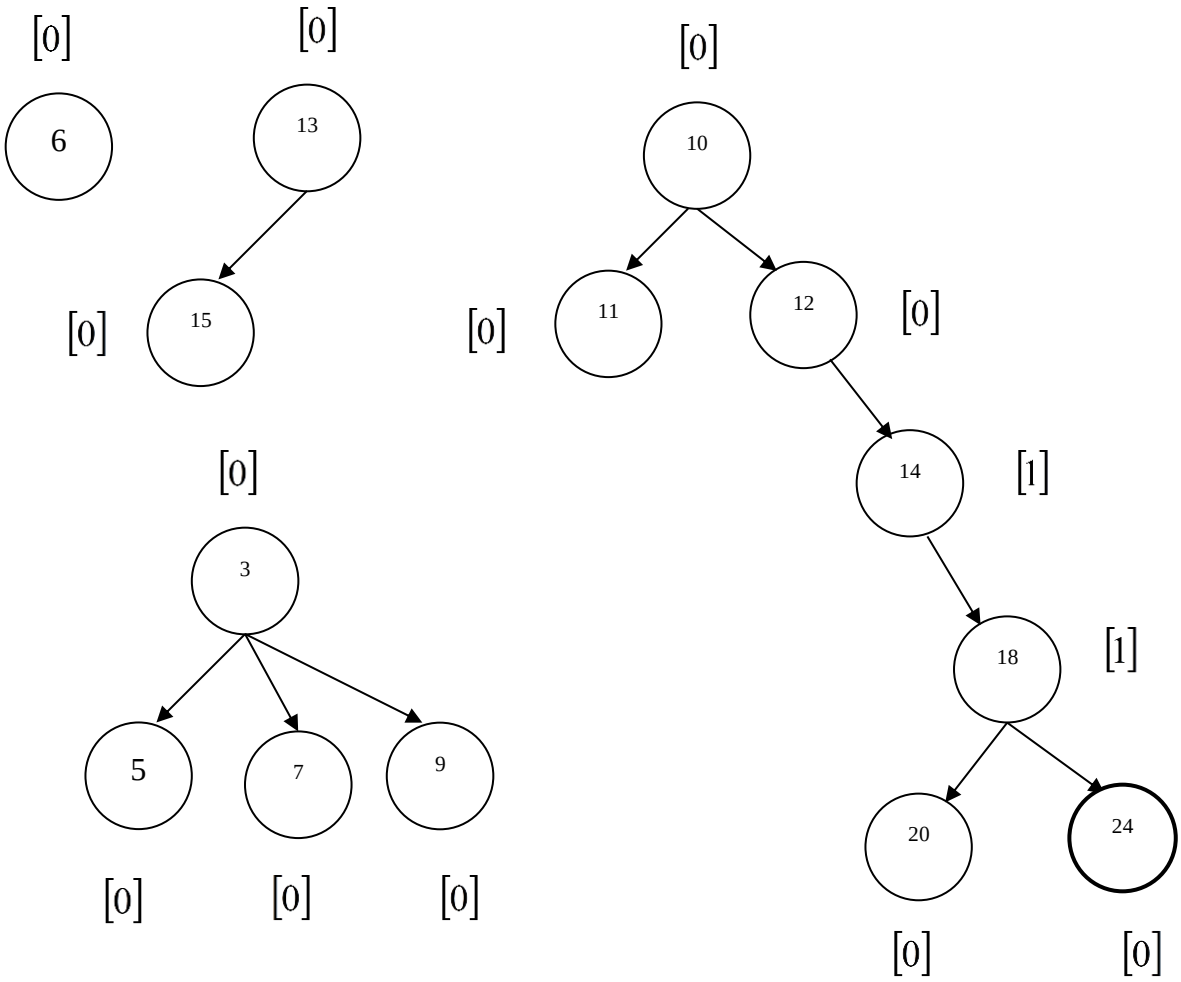

Рис. 10

Модифицируем значение минимального ключа: $minkey := 2$.

Для вершины *i* находим ее отца: $j = pred(i) = 18$.

Поскольку $2 < 18$, т. е. нарушен *инвариант 1*, то выполним одну *начальную процедуру cut*(2).

В результате имеем коллекцию корневых деревьев, изображенную на рис. 11.

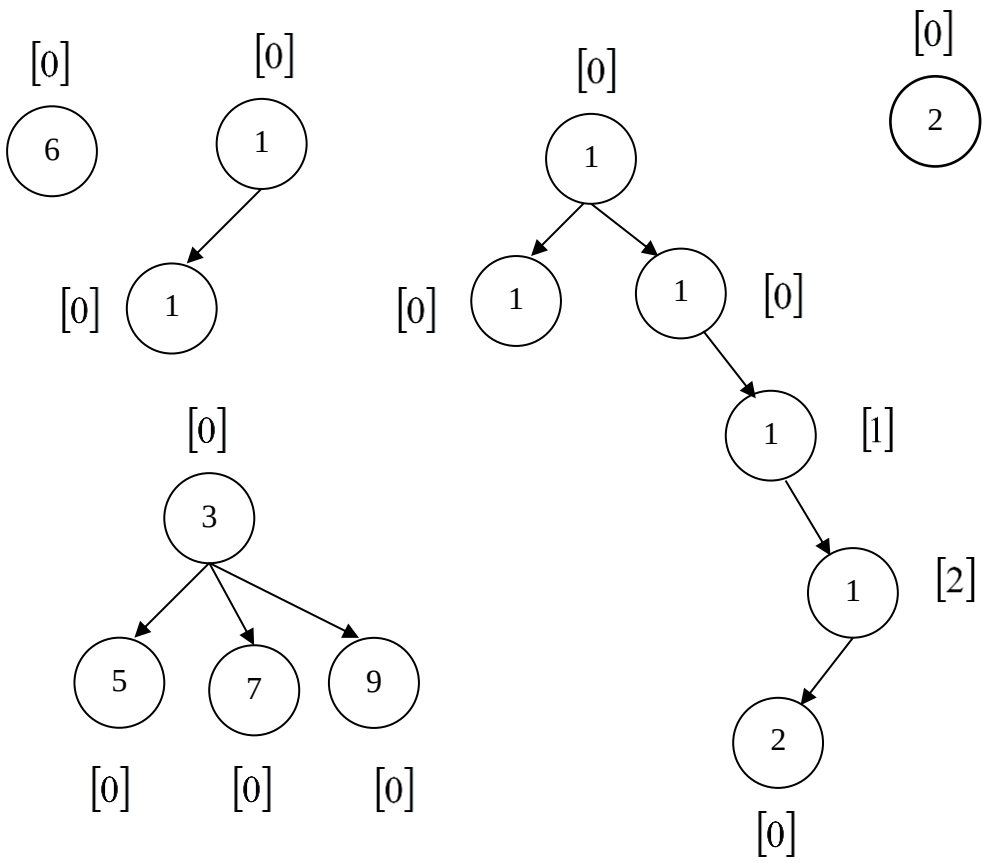

Рис. 11

Помещаем в список *list*(0) вершину с ключом 2

i	**0**	**1**	**2**	**3**
bucket	6	13	10	3
list	2	*nil*	*nil*	*nil*

Поскольку $lost(18)=2$ (вершина 18 потеряла двух сыновей), то необходимо восстановление *инварианта 3*. Это осуществляется с помощью *порожденной процедуры cut*(18).

i	**0**	**1**	**2**	**3**
bucket	6	13	10	3
list	2	*18*	*nil*	*nil*

В свою очередь, порожденная процедура *cut*(18) породит процедуру *cut*(14), и поскольку $lost(12)=1$, то инвариант 3 выполнен (рис. 12).

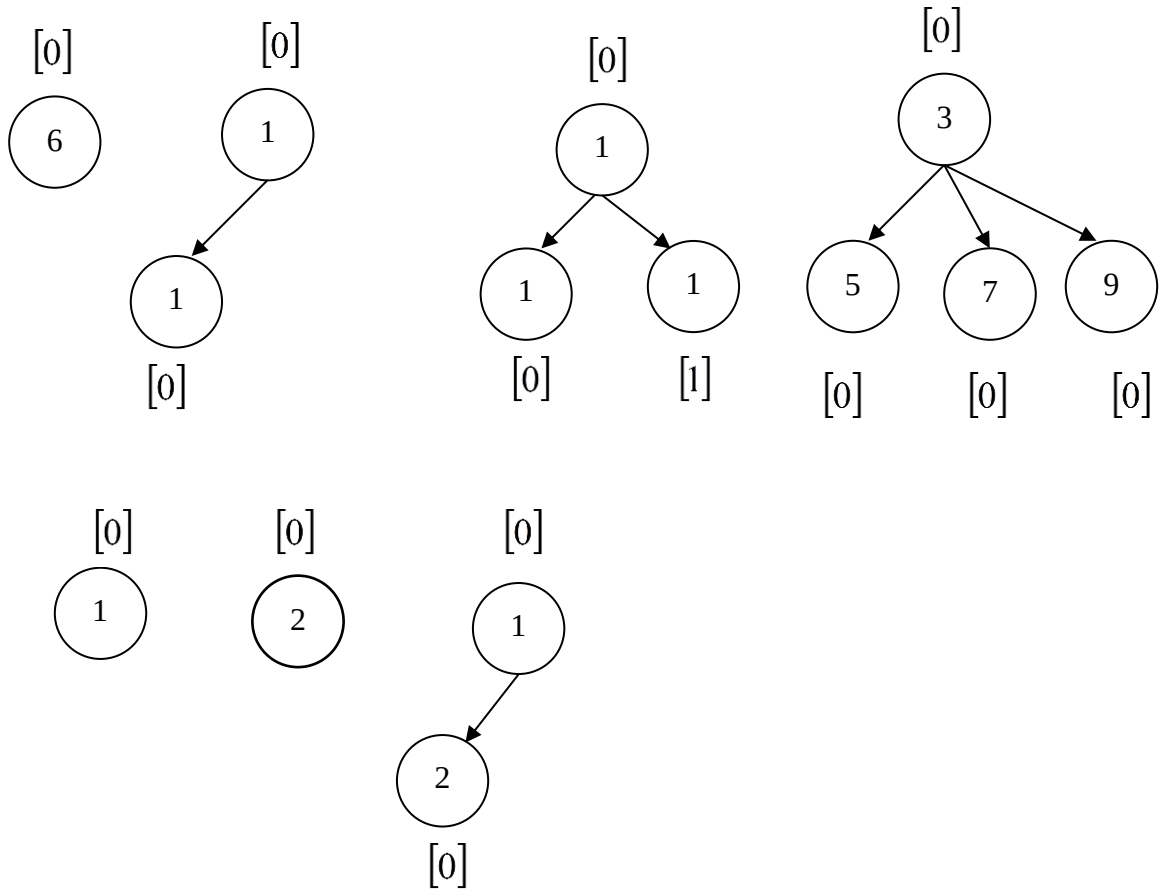

Рис. 12

Поскольку ранг узла 14 равен 0, а в массиве *list* список *list*(0) не пуст, то выполняем процедуру *link* (2, 14) и получаем корневое дерево с корнем 2 ранга 1.

Поскольку список *list*(1) не пуст, то выполняем процедуру *link* (2, 18), и на этом формирование списка list завершено.

i	**0**	**1**	**2**	**3**
bucket	6	13	10	3
list	*nil*	*nil*	2	*nil*

На рис. 13 изображены: вверху – семейство корневых деревьев, которые остались в куче Фибоначчи после выполнения *начальной* и *порожденных процедур cut*; внизу – корневое дерево, соответствующее списку *list*.

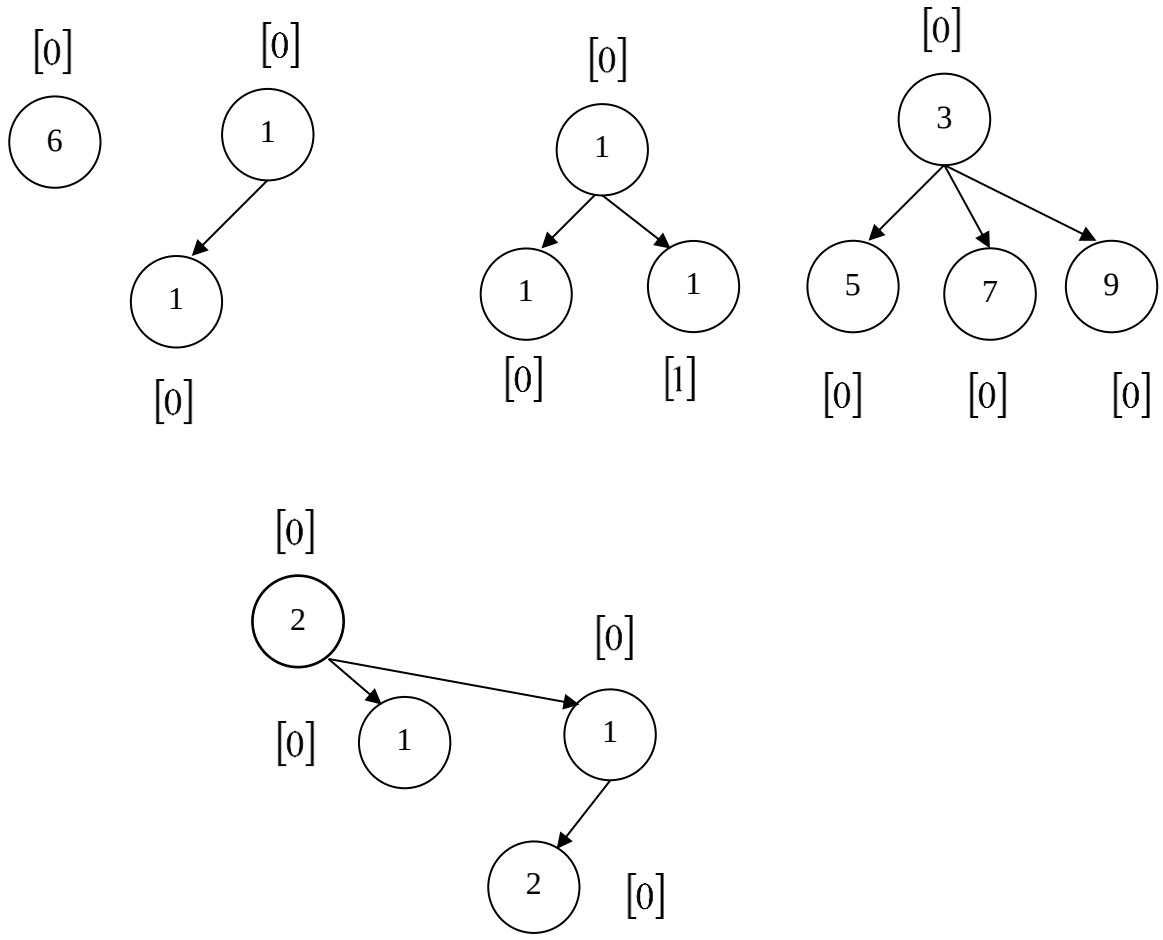

Рис. 13

Теперь необходимо провести слияние корневых деревьев из массивов *list* и *bucket*.

Для слияния этих двух массивов будет сначала выполнена процедура *link* (2, 10), а затем *link* (2, 3).

i	**0**	**1**	**2**	**3**	**4**
bucket	6	13	*nil*	*nil*	2

На рис. 14 изображена куча Фибоначчи, полученная после выполнения операции уменьшения значения ключа и восстановления инвариантов.

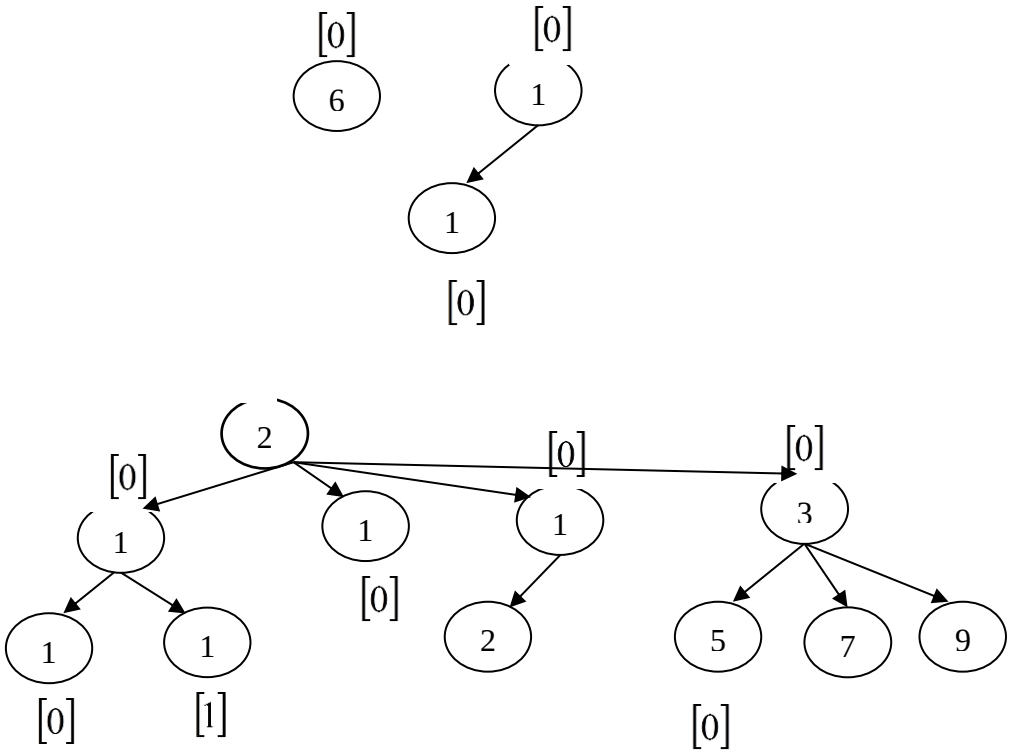

Рис. 14

Упражнение. Реализовать операцию удаления произвольной вершины из кучи Фибоначчи и получить усредненную оценку ее трудоемкости.

Применение структуры данных куча

Два алгоритма решения задачи нахождения k-го наименьшего элемента массива были рассмотрены ранее. Эту задачу также можно решить с помощью структуры данных *куча*. Для этого нужно поместить все n элементов массива в кучу ($\mathrm{O}(n)$), а затем удалить из нее k элементов ($\mathrm{O}(k\log n)$). Последний из удаленных элементов и будет искомым. Заметим, что если k – константа, то трудоемкость данного алгоритма равна $\mathrm{O}(n)+\mathrm{O}(k\log n)$.

Структуру данных *куча* можно использовать для внутренней сортировки элементов массива (*heap sort*). Для этого достаточно вначале добавить каждый из элементов в кучу, а затем поочередно удалить. Трудоемкость сортировки, реализованной таким образом, определяется трудоемкостью операций добавления элемента и удаления минимального элемента и равна $\mathrm{O}(n)+\mathrm{O}(n\log n)=\mathrm{O}(n\log n)$. Таким образом, данный алгоритм имеет ту же трудоемкость, что и лучшие алгоритмы внутренней сортировки.

Структура данных куча может использоваться при реализации алгоритма Дейкстры поиска кратчайшего маршрута в графе с положительными длинами ребер, а также в алгоритме Краскала постоения минимального остовного дерева.