Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ Информатика и системы управления

КАФЕДРА Информационная безопасность (ИУ8)

ОТЧЕТ ПО ДОМАШНЕЙ РАБОТЕ

ПО АЛГОРИТМАМ И СТРУКТУРАМ ДАННЫХ

НА ТЕМУ:
Сравнение биноминальной кучи и двоичной кучи

Студент ИУ8-53 _____ С. А. Митяков
(Группа) (Подпись, дата) (И.О.Фамилия)

Преподаватель _____ В. О. Чесноков
(Подпись, дата) (И.О.Фамилия)

Москва 2017

Оглавление

Теоретическая часть

Двоичная куча

Деревом называется неориентированный, связный, ациклический граф. Двоичное дерево – это конечное множество вершин, которое либо пусто, либо состоит из трех непересекающихся подмножеств:

- вершины, которая называется корнем;
- левого поддерева, которое является двоичным деревом;
- правого поддерева, которое является двоичным деревом.

Двоичная куча – такое двоичное дерево, для которого выполняются следующие три условия:

1. Значение в каждой вершине не меньше (если куча невозрастающая) или не больше (если куча неубывающая), чем значения её потомков.
2. На i-ом слое 2^i вершин, кроме последнего. Слои нумеруются с нуля, начиная с корня.
3. Последний слой заполнен слева направо.

Пример неубывающей двоичной кучи представлен на рисунке 1.

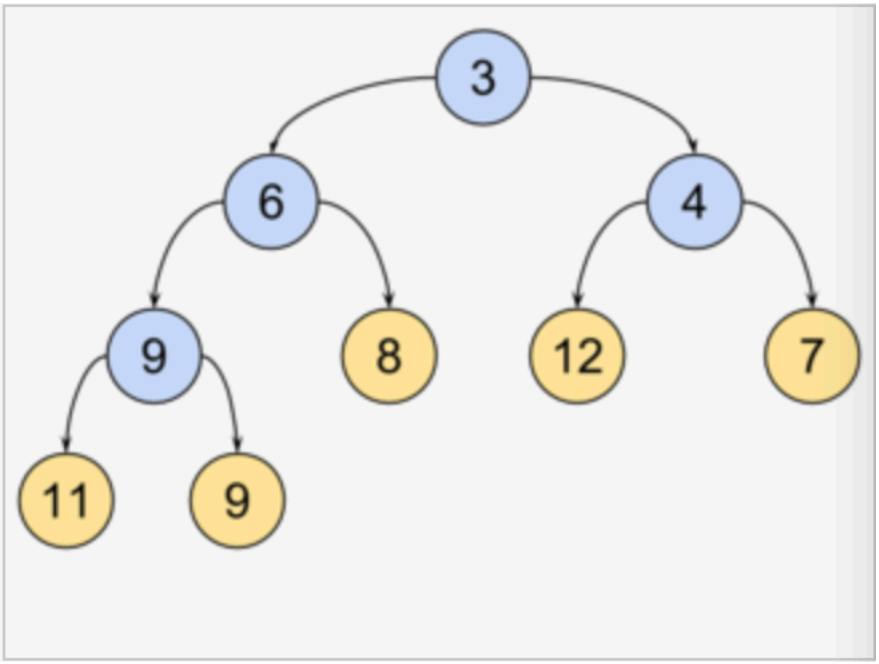

Рисунок 1

Виды двоичной кучи в формализованном виде (key(i) – ключ (значение) элемента с индексом i; left(i) – индекс левого сына элемента с индексом i; right(i) – индекс правого сына элемента с индексом i):

- MaxHeap – невозрастающая куча:
  - $\text{key(i)} \geq \text{key}\big(\text{left(i)}\big)$
  - $\text{key(i)} \geq \text{key}\big(\text{right(i)}\big)$
- MinHeap – неубывающая куча:
  - $\text{key(i)} \leq \text{key}\big(\text{left(i)}\big)$
  - $\text{key(i)} \leq \text{key}\big(\text{right(i)}\big)$

Двоичную кучу из n элементов удобно хранить в виде одномерного массива A[0..n-1], причем A[0] – элемент в корне, A[i] – i-ый элемент, A[2i+1] – левый сын i-го элемента, A[2i+2] – правый сын i-го элемента.

Пример представления двоичной кучи с рисунка 1 в виде массива представлен на рисунке 2 (красная стрелка – левый сын, зеленая – правый).

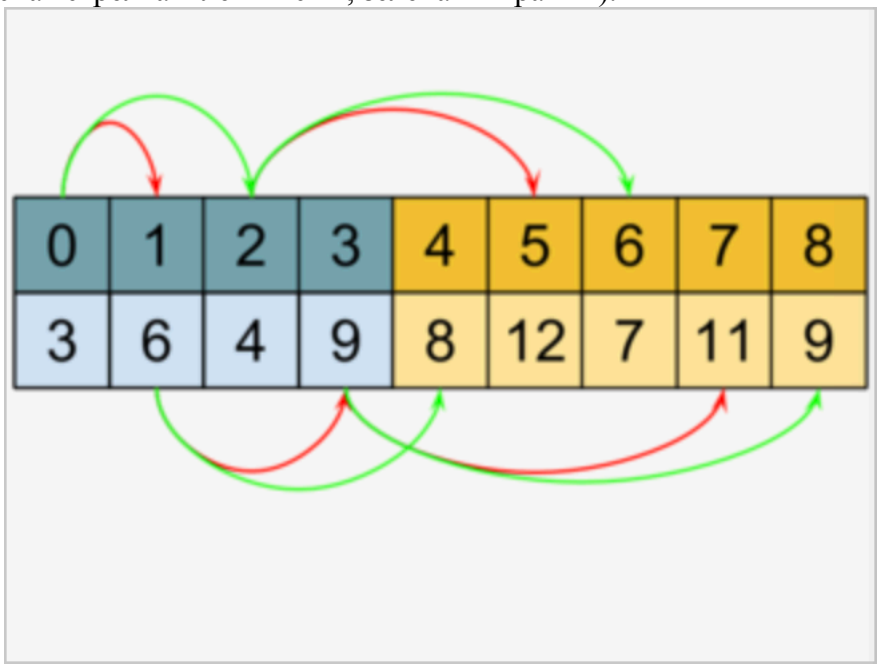

Рисунок 2

Высота двоичной кучи определяется как высота двоичного дерева: она равна количеству рёбер в самом длинном простом пути, соединяющим корень кучи с одним из её листьев (вершин, у которых нет сыновей). Высота двоичной кучи есть $\lfloor \log_2 n \rfloor$, где n – количество элементов в куче.

Базовые процедуры (для невозрастающей кучи):

- **Восстановление свойств кучи**

Если в куче изменяется один из элементов или добавляется новый, то она может перестать удовлетворять свойству упорядоченности (первому условию). Для восстановления этого свойства служат процедуры siftDown (просеивание вниз) и siftUp (просеивание вверх).

siftDown

Если значение измененного элемента увеличивается, то свойства кучи восстанавливаются функцией siftDown. Она работает следующим образом:

Если i-ый элемент меньше или равен его сыновьям, то все поддерево уже является кучей и делать ничего не надо. В противном случае i-ый элемент меняется местами с наименьшим своим сыном, после чего выполняется siftDown для этого сына.

Данная процедура выполняется за время $O(\log_2 n)$, т.к. в худшем случае процедура должна спустить корень на самый последний слой, т.е. пройтись по всей куче сверху вниз, а высота кучи равна $\log_2 n$.

siftUp

Если значение измененного элемента уменьшается, то свойства кучи восстанавливаются функцией siftUp. Она работает следующим образом:

Если i-ый элемент больше или равен своему отцу, то все дерево уже является кучей и делать ничего не надо. В противном случае i-ый элемент меняется местами со своим отцом, после чего выполняется siftUp для этого отца.

Данная процедура выполняется за время $O(\log_2 n)$, т.к. в худшем случае процедура должна поднять лист в корень, т.е. пройтись по всей куче снизу вверх, а высота кучи равна $\log_2 n$.

- **Добавление нового элемента**

Процедура **insert** добавляет новый элемент в кучу. Она работает следующим образом:

Происходит добавление элемента в конец кучи, после чего восстанавливается свойство упорядоченности с помощью процедуры siftUp, применяемой к этому элементу (концу кучи).

Выполнение процедуры insert показано на рисунке 3.

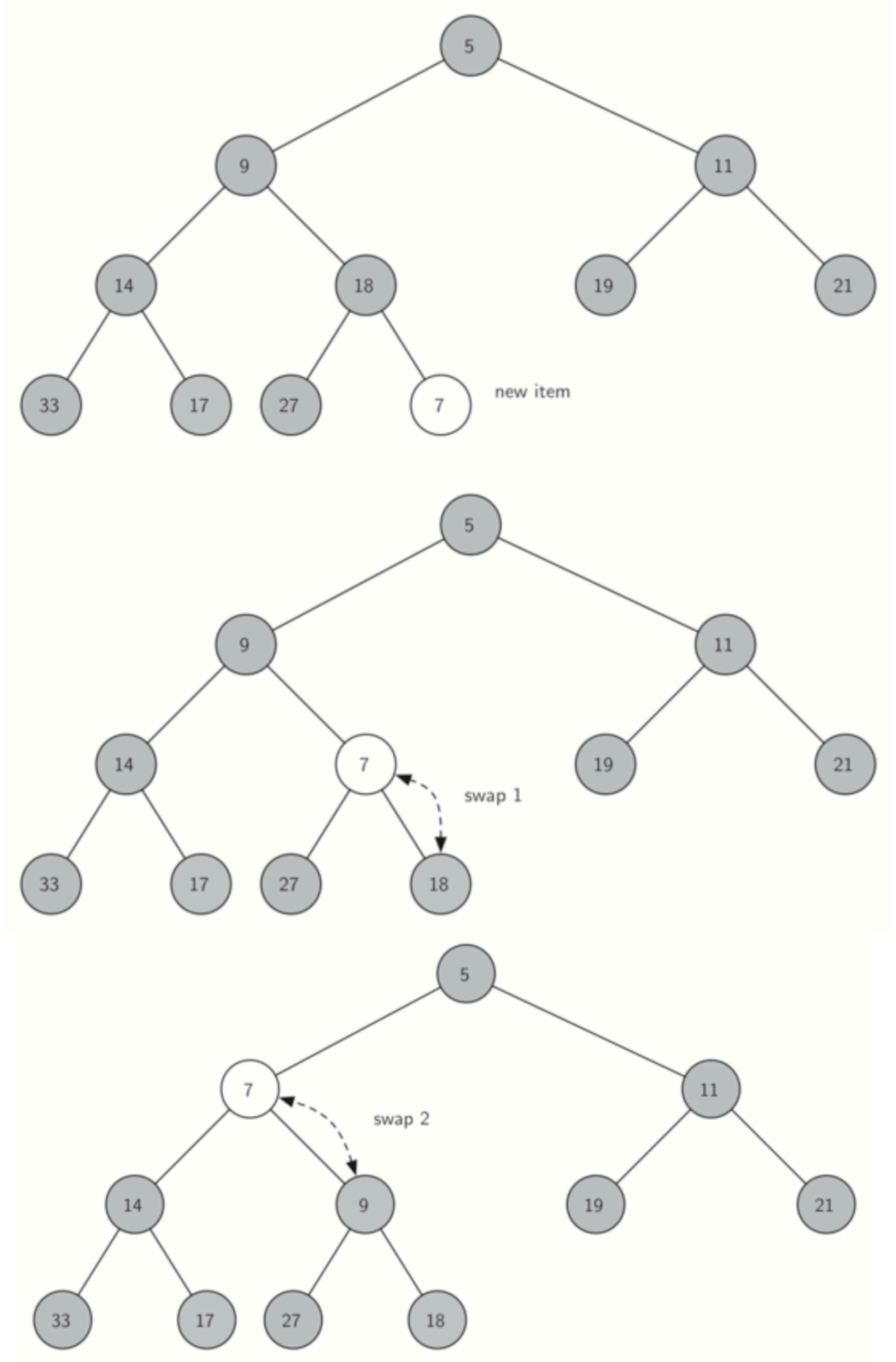

Рисунок 3

Данная процедура выполняется за время $O(\log_2 n)$, т.к. худший случай для неё – это худший случай для процедуры siftUp, а само добавление в конец массива кучи выполняется за O(1).

- **Извлечение минимального элемента**

Процедура **extractMin** извлекает минимальный элемент из кучи. Она работает следующим образом:

Значение корневого элемента (он для неубывающей кучи является минимальным) сохраняется для последующего возвращения. Затем последний элемент копируется в корень, после чего удаляется из кучи. Далее восстанавливается свойство упорядоченности кучи с помощью процедуры siftDown, применяемой к корню, и возвращается сохраненный элемент.

Выполнение процедуры extractMin показано на рисунках 4 и 5.

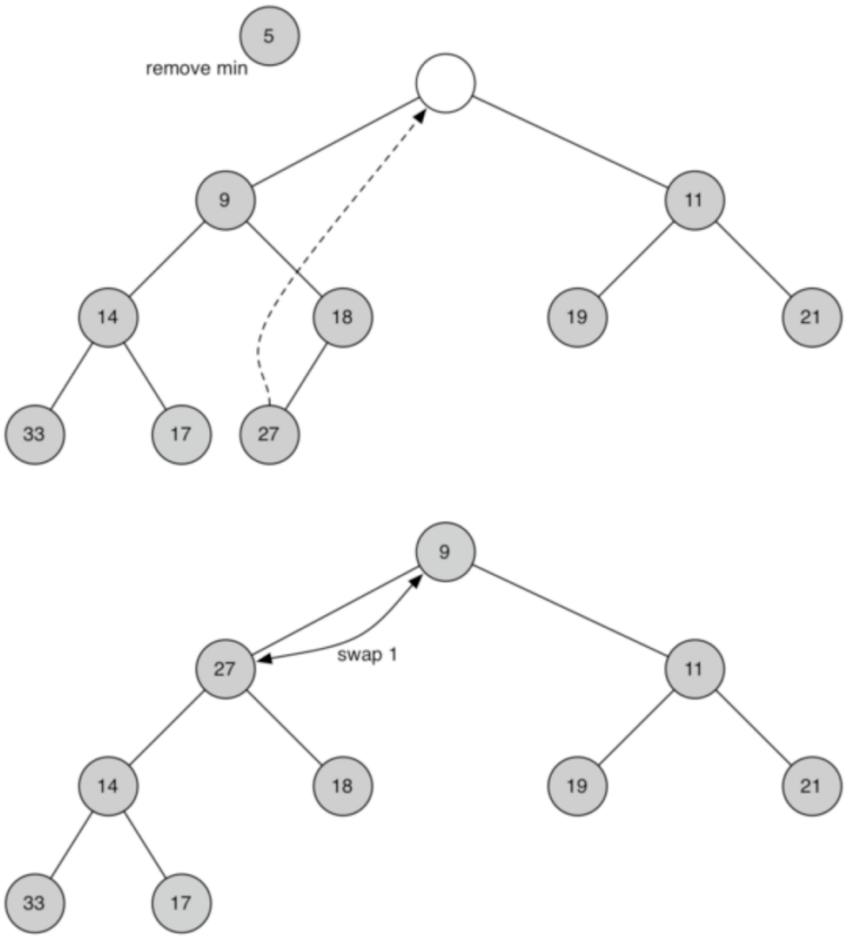

Рисунок 4

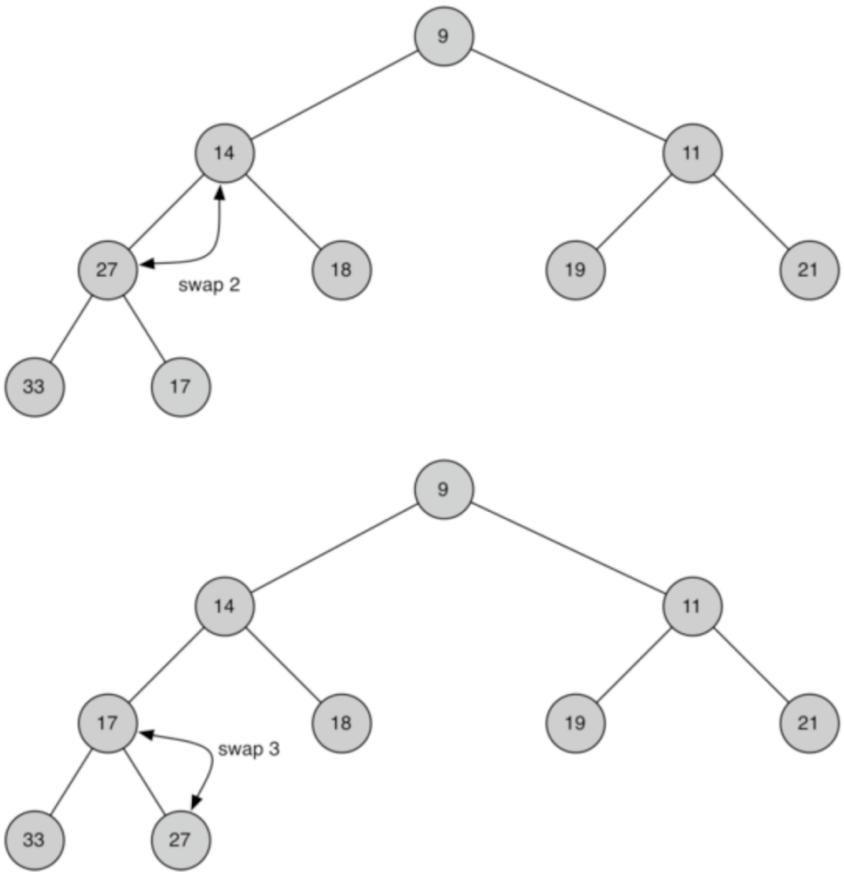

Рисунок 5

Данная процедура выполняется за время $O(\log_2 n)$, т.к. худший случай для неё – это худший случай для процедуры siftDown, а сохранение, копирование и возвращение значения выполняются за O(1).

- **Построение кучи** (из неупорядоченного массива)

Процедура **buildHeap** строит двоичную кучу из массива A[0..n-1].

Наиболее очевидный способ построить кучу в данном случае – сделать нулевой элемент массива корнем, а дальше по очереди добавить все его элементы в кучу с помощью процедуры insert. Тогда данная процедура выполнялась бы за время $O(n \log_2 n)$, т.к. в худшем случае имеем n худших случаев операции insert. Однако можно построить кучу быстрее – за O(n). Это связано со свойством двоичной кучи – число вершин высоты h в куче из n элементов не превышает $\left\lceil \frac{n}{2^{h+1}} \right\rceil$. Значит, число листьев в куче не превышает $\left\lceil \frac{n}{2} \right\rceil$, а число элементов, не являющихся листьями не превышает $\left\lfloor \frac{n}{2} \right\rfloor$. Поддеревья, состоящие из одного элемента (листья), уже являются кучами. При вызове процедуры siftDown для вершин высоты 1, их поддеревья (листья) уже являются кучами. После выполнения siftDown эти вершины с их поддеревьями будут являться кучами. Значит, после выполнения всех siftDown, применяемых ко всем вершинам, у которых есть хотя бы один потомок, в обратном порядке их индексов получится куча. В итоге процедура работает следующим образом:

К каждому элементу с индексом в диапазоне от $\left\lfloor \frac{n}{2} \right\rfloor - 1$ до 0 (в порядке убывания), где n – количество элементов в куче, применяется процедура siftDown.

Выполнение процедуры buildHeap показано на рисунке 6.

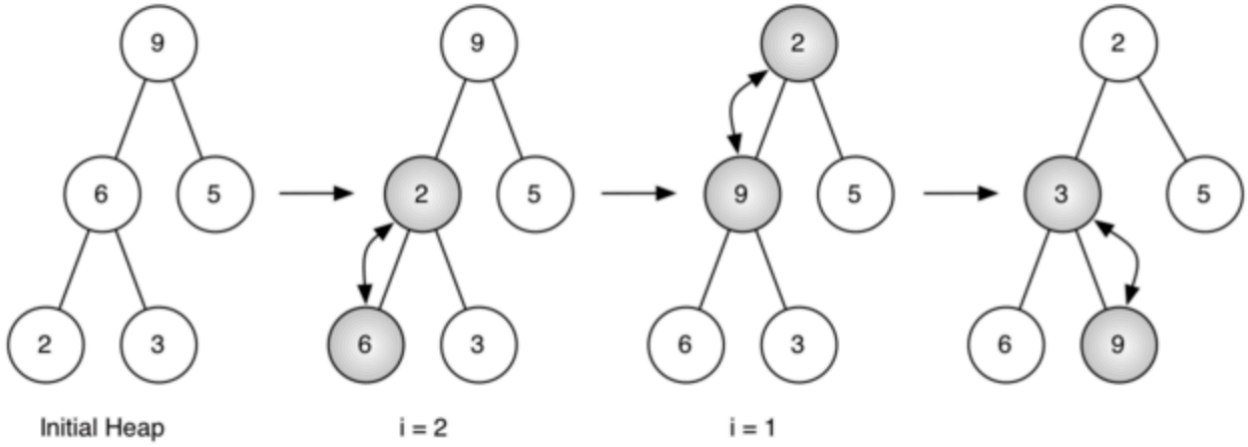

Рисунок 6

Доказательство времени выполнения процедуры:

Число вершин высоты h в куче из n элементов не превышает $\left\lceil \frac{n}{2^{h+1}} \right\rceil$, высота h кучи не превышает $\lfloor \log_2 n \rfloor$, а время выполнения процедуры siftDown – $O(\log_2 n) = O(h)$. Тогда время работы процедуры buildHeap не превышает $\sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(\frac{n}{2} \sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h}\right)$. Сумма бесконечно убывающей геометрической прогрессии $\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$, $|x| < 1$. Дифференцируя и умножая на x, получаем $\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2}$. Подставим x=1/2, k=h, получим $\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$. Подставим результат в исходное выражение и получим $O\left(\frac{n}{2} \times 2\right) = O(n)$ ■.

- **Удаление произвольного элемента**

Процедура **delete** удаляет произвольный элемент из кучи. Она работает следующим образом:

Последний элемент копируется в удаляемый, после чего удаляется из кучи. Далее восстанавливается свойство упорядоченности кучи с помощью процедуры siftUp, применяемой к извлекаемому элементу. Если она восстановила свойство кучи, то задача выполнена, в противном случае к извлекаемому элементу применяется процедура siftDown. В конце возвращается сохраненное значение.

Данная процедура выполняется за время $O(\log_2 n)$, т.к. худший случай для неё – это худший случай для процедуры siftUp, либо для процедуры siftDown, а сохранение, копирование и возвращение значения выполняются за O(1).

- **Слияние двух куч**

Процедура **merge** объединяет две двоичные кучи. Она работает следующим образом:

Все элементы массива одной кучи добавляются в конец массива второй кучи, после чего к полученному массиву применяется процедура buildHeap.

Данная процедура выполняется за время O(n+m), где n и m – число элементов первой и второй кучи соответственно, т.к. перемещение элементов из одного массива в другой выполняется за время O(n) или O(m) (в зависимости от того, из какого в какой массив перемещаются элементы), а процедура buildHeap выполняется за время O(n+m).

Биноминальная куча

Биноминальное дерево B_k – дерево, определяемое для каждого k = 0, 1, 2,… следующим образом: B_0 – дерево, состоящее из одного узла; B_k состоит из двух биноминальных деревьев B_{k-1}, связанных вместе таким образом, что корень одного из них является самым левым ребенком корня другого.

На рисунке 7а показано определение биноминального дерева, а на 7б – деревья $B_0, \dots, B_4$.

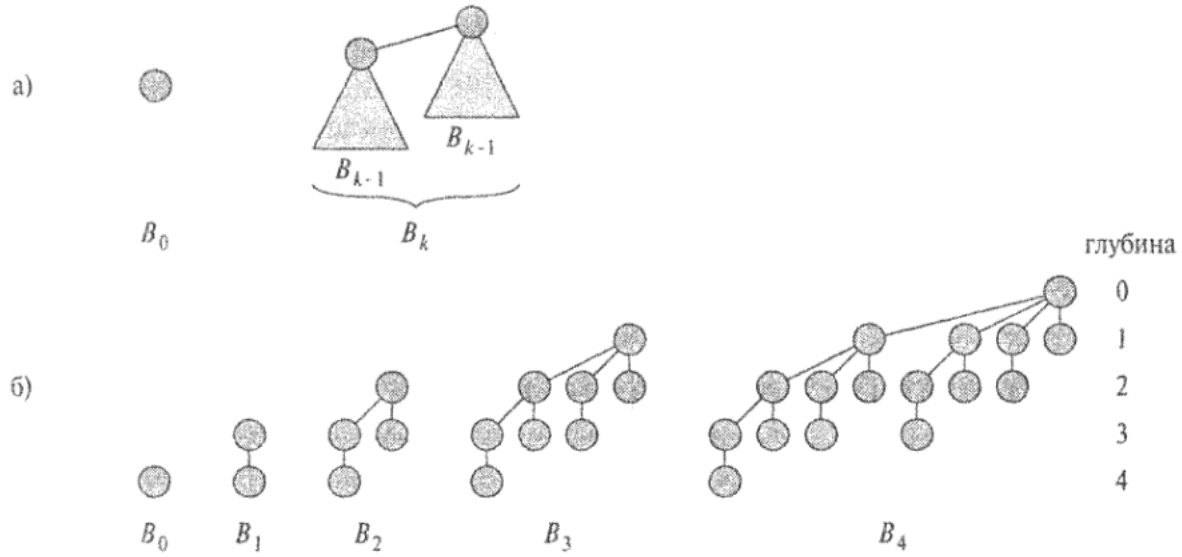

Рисунок 7

Свойства биноминального дерева B_k:

1. содержит 2^k вершин;
2. имеет высоту k;
3. имеет C_k^i вершин глубины i;
4. имеет корень, являющийся вершиной степени k; все остальные вершины имеют меньшую степень; дети корня являются корнями поддеревьев B_{k-1}, B_{k-2},…, B_1, B_0 (слева направо).

Следствие из свойств 1 и 4: максимальная степень вершины в биноминальном дереве с n вершинами равна $\log_2 n$.

Биноминальная куча представляет собой множество биноминальных деревьев, которые удовлетворяют следующим свойствам:

1. каждое биноминальное дерево в куче подчиняется свойству неубывающей кучи: ключ узла не меньше ключа его родительского узла;
2. в куче нет двух деревьев одинакового размера (с одинаковой степенью корня).

Первое свойство гарантирует, что корень каждого из деревьев имеет наименьший ключ среди его вершин.

Из второго свойства следует, что суммарное количество вершин в биноминальной куче однозначно определяет размеры входящих в неё деревьев. В самом деле, общее число вершин, равное n, есть сумма размеров отдельных деревьев, которые равны различным степеням двойки, а такое представление единственно (двоичная система счисления). Отсюда также вытекает, что куча с n элементами состоит не более, чем из $\lfloor \log_2 n \rfloor - 1$ биноминальных деревьев.

На рисунке 8а показана биноминальная куча с 13 узлами.

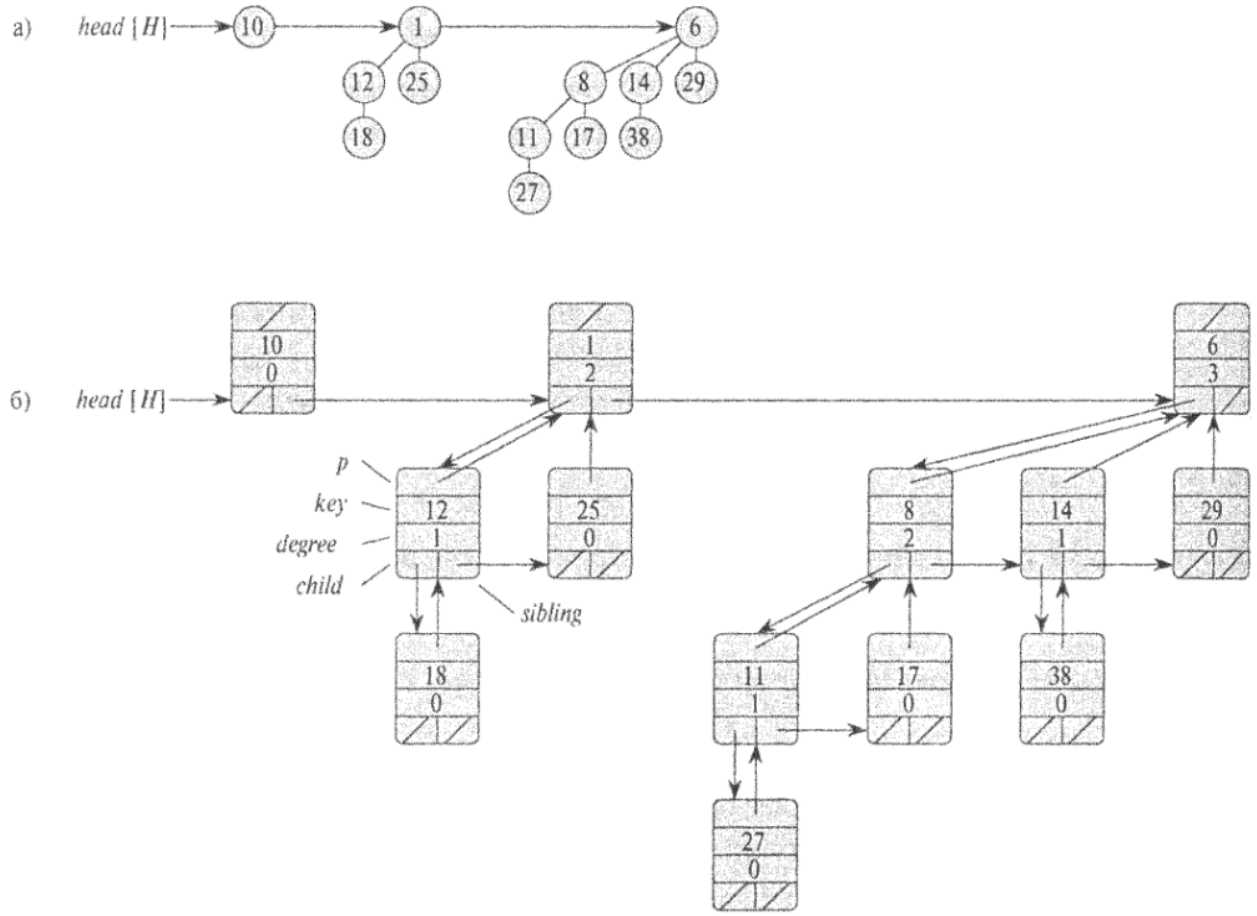

Рисунок 8

Представление биноминальной кучи

Каждое биноминальное дерево в биноминальной куче хранится в представлении «левый ребенок, правый сосед». Каждая вершина имеет поле key (ключ), а также может хранить дополнительную информацию. Кроме того, каждая вершина x хранит указатели p[x] (родитель), child[x] (самый левый из детей), и sibling[x] («следующий по старшинству брат»). Если x – корень, то p[x] = NIL; если x не имеет детей, то child[x] = NIL, а если x является самым правым ребенком своего родителя, то sibling[x] = NIL. Каждая вершина x содержит также поле degree[x], в котором хранится степень (число детей) вершины x.

Корни биноминальных деревьев, составляющих кучу, связаны в список, называемый корневым списком (root list) в порядке возрастания степеней. Для построения корневого списка используется поле sibling: для корневой вершины оно указывает на следующий элемент корневого списка. Степени корневых вершин образуют подмножество множества $\{0, 1, \ldots, \lfloor \log_2 n \rfloor\}$.

Доступ к биноминальной куче H осуществляется с помощью поля head[H] – указателя на первый корень в корневом списке кучи H. Если куча H пуста, то head[H] = NIL.

На рисунке 8б показано представление биноминальной кучи с 13 узлами в описанном выше виде.

Базовые процедуры:

- **Создание новой кучи**

Процедура **buildHeap** создает новую пустую кучу. Она работает следующим образом: Создается и возвращается объект H, для которого head[H] = NIL.

Данная процедура работает за время O(1).

- **Поиск минимального ключа**

Процедура **minimum** возвращает указатель на вершину с минимальным ключом в биноминальной куче H. Она работает следующим образом:

Просматривается корневой список кучи, храня в переменной min минимальное из просмотренных значений, а в переменной y – вершину, где оно достигается. В конце выполнения возвращается значение y.

Данная процедура работает за время $O(\log_2 n)$, т.к. длина корневого списка не превосходит $\lfloor \log_2 n \rfloor + 1$.

- **Объединения двух куч**

Процедура **union** соединяет две биноминальные кучи в одну. Она работает следующим образом:

Процедура имеет две фазы. В первой фазе объединяются корневые списки биноминальных куч H_1 и H_2 в единый связный список H, который отсортирован по степеням корней в монотонно возрастающем порядке. Однако в списке может оказаться по два (и более) корня каждой степени, так что во второй фазе связываются корни одинаковой степени до тех пор, пока все корни не будут иметь разную степень.

Выполнение процедуры union показано на рисунках 9 и 10.

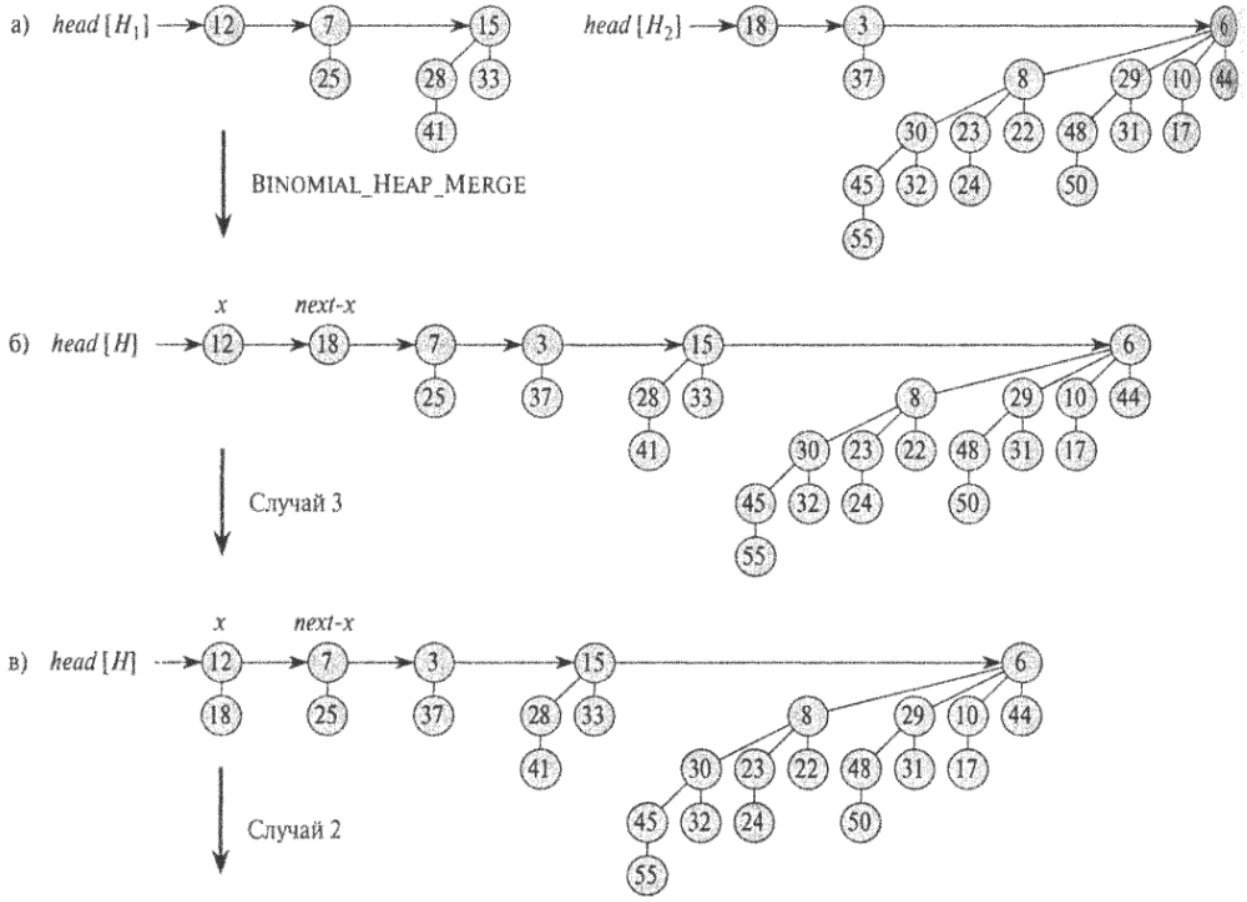

Рисунок 9

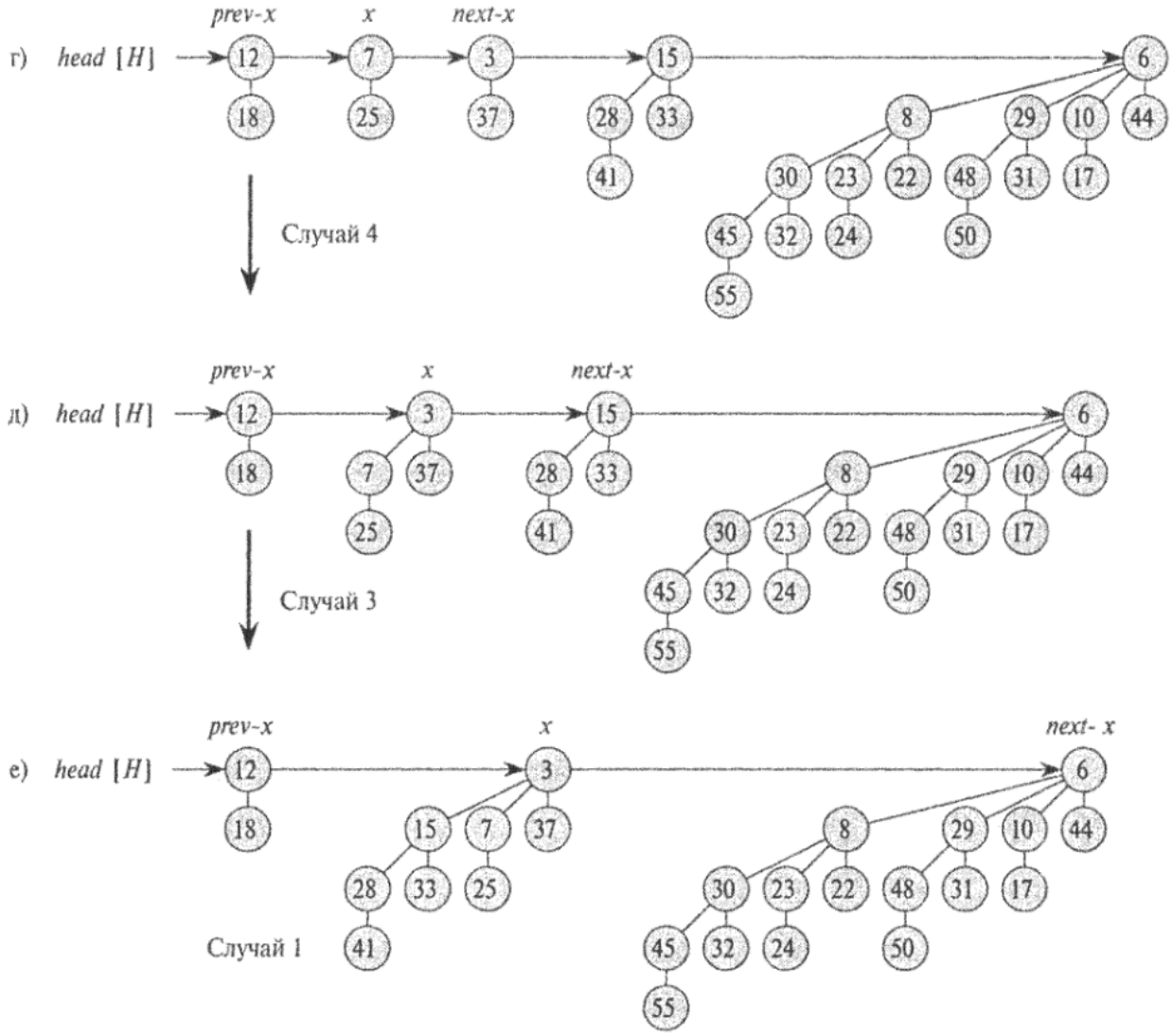

Рисунок 10

Данная процедура выполняется за время $O(\log_2 n)$, где n – общее количество узлов в биноминальных кучах H_1 и H_2.

Доказательство:

Пусть H_1 содержит n_1 узлов, а $H_2 - n_2$ узлов, так что $n_1 + n_2 = n$. Тогда H_1 содержит не более $\lfloor \log_2 n_1 \rfloor + 1$ коБрней, а H_2 – не более $\lfloor \log_2 n_2 \rfloor + 1$ корней. Таким образом, после объединения корневых списков H содержит не более $\lfloor \log_2 n_1 \rfloor + \lfloor \log_2 n_2 \rfloor + 2 \leq 2\lfloor \log_2 n \rfloor + 2 = O(\log_2 n)$ корней. Следовательно, на объединение затрачивается $O(\log_2 n)$ времени. Время работы вторая фазы процедуры тоже пропорциональна начальной длине корневого списка H и составляет $O(\log_2 n)$. В итоге получаем, что общее время выполнения процедуры есть $O(\log_2 n)$.

- **Добавление нового элемента**

Процедура **insert** добавляет новый элемент в биноминальную кучу. Она работает следующим образом:

Создается новая биноминальная куча H’ из одной вершины – добавляемого элемента. Затем она объединяется с уже имеющейся биноминальной кучей с помощью процедуры union.

Данная процедура работает за время $O(\log_2 n)$, т.к. создание новой кучи H’ происходит за O(1), а объединение – за $O(\log_2 n)$.

- **Извлечение минимального элемента**

Процедура **extractMin** извлекает вершину с минимальным ключом из биноминальной кучи и возвращает указатель на неё. Она работает следующим образом:

В корневом списке биноминальной кучи с помощью процедуры minimum находится вершина с минимальным ключом, сохраняется указатель на неё, и вершина удаляется из кучи. После удаления дерево, корнем которого являлась вершина, распадается на поддеревья меньшего размера. Происходит объединение этих поддеревьев с оставшейся частью биноминальной кучи с помощью процедуры union. В конце работы возвращается сохраненный указатель на извлеченную вершину.

Выполнение процедуры extractMin показана на рисунке 11.

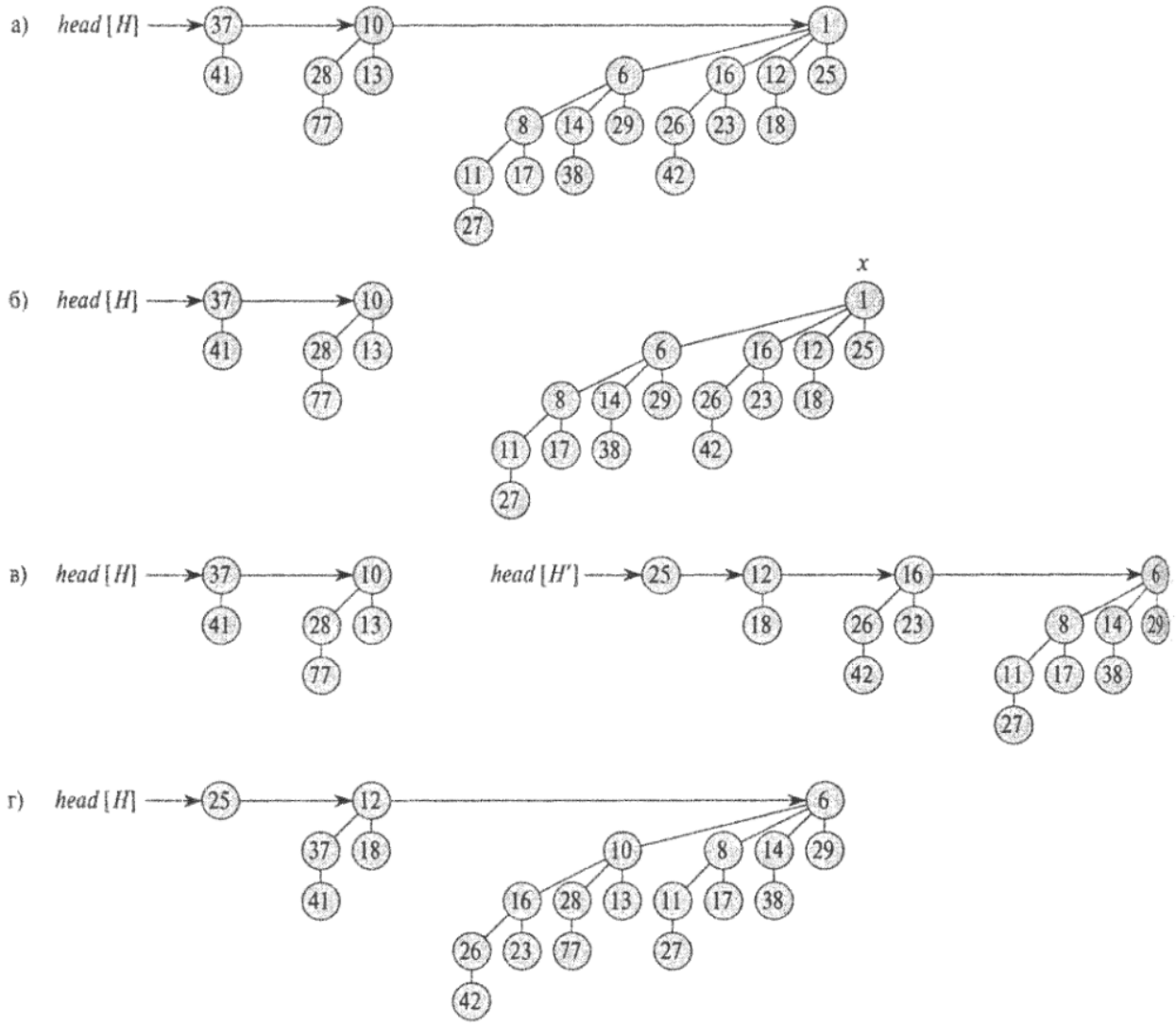

Рисунок 11

Данная процедура работает за время $O(\log_2 n)$, т.к. процедура minimum работает за время $O(\log_2 n)$, сохранение, удаление и возвращение – за O(1), а процедура union – за $O(\log_2 n)$.

- **Уменьшение ключа элемента**

Процедура **decreaseKey** уменьшает ключ заданного элемента биноминальной кучи, присваивая ему новое заданное значение. Она работает следующим образом:

Если значение заданного ключа больше ключа элемента, то выдается сообщение об ошибке. В противном случае элементу присваивается новое значение ключа, затем

происходит попытка «поднять» элемент вверх по дереву (восстановление свойства неубывающей кучи).

Выполнение процедуры decreaseKey показано на рисунке 12.

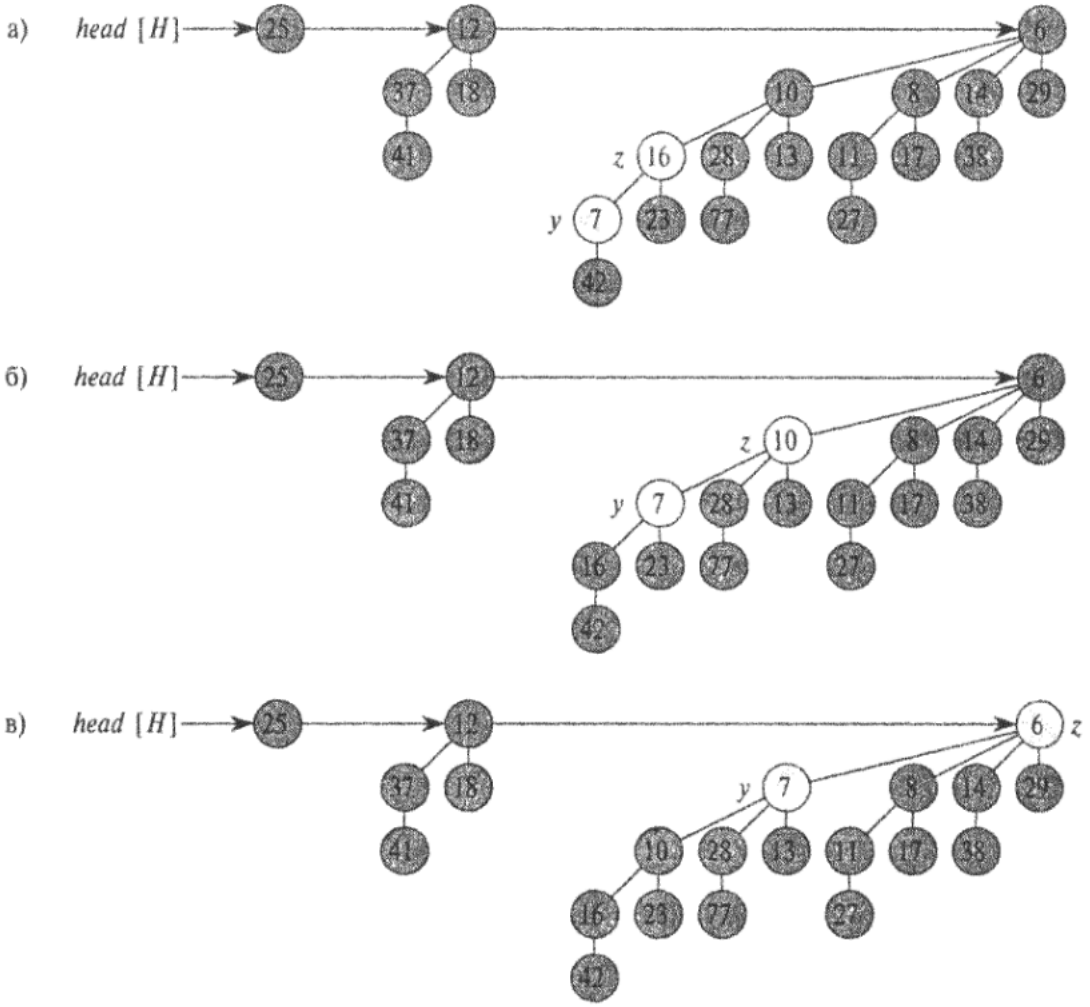

Рисунок 12

Данная процедура работает за время $O(\log_2 n)$, т.к. изменение значения ключа происходит за время O(1), а «подъем» вершины – за $O(\log_2 n)$ (в худшем случае: вершина – лист с глубиной $O(\log_2 n)$, и её нужно «поднять» в корень дерева).

- **Удаление произвольного элемента**

Процедура **delete** удаляет вершину с произвольным ключом из биноминальной кучи. Она работает следующим образом:

Значение ключа удаляемой вершины с помощью процедуры decreaseKey заменяется на специальное значение, которое меньше значения любого элемента в куче, благодаря чему эта вершина перемещается в корень своего биноминального дерева. Затем с помощью процедуры exctractMin происходит её извлечение из биноминальной кучи.

Данная процедура работает за время $O(\log_2 n)$, т.к. время работы двух составляющих её процедур decreaseKey и extractMin - $O(\log_2 n)$.

Предварительное сравнение и план решения задачи

Если нам не нужна операция объединения куч, вполне хорошо работают обычные двоичные кучи. Все остальные операции выполняются в двоичной куче в худшем случае за $O(\log_2 n)$. Однако при необходимости поддержки операции объединения привлекательность двоичных куч резко уменьшается, поскольку реализация этой операции путем объединения двух массивов, в которых хранятся двоичные кучи, с последующим выполнением операции построения новой кучи требует O(n) времени, тогда как для биноминальных куч операция объединения требует $O(\log_2 n)$ времени, где n – общее количество элементов в двух объединяемых кучах.

План решения задачи:

1. Реализовать оба вида куч с их процедурами на языке C++.
2. С помощью тестов, представляющих собой набор различных команд для двух видов куч, получить информацию о скорости работы каждой структуры данных.
3. Сравнить полученные результаты.
4. Сделать выводы о результатах сравнения.

Практическая часть

Ссылка на программу, размещенную в публично доступном репозитории сервиса GitHub: https://github.com/MityakovSA/HeapsComparison.

Описание программы, формат входных и выходных данных, а также инструкция по использованию и компилированию программы находятся в файле README.md в репозитории.

Помимо операций, перечисленных и описанных в теоретической части, были также добавлены специфичные для данных структур данных операции, требуемые в задании.

Дополнительные операции для двоичной кучи:

- **Поиск минимального элемента**

Процедура **min()** возвращает указатель на элемент кучи с минимальным ключом. Она работает следующим образом:

Так как элементы с минимальным ключом находится в корне кучи, т.е. является первым элементом массива, процедура возвращает указатель, на элемент массива с индексом 0.

Данная процедура работает за время O(1), т.к. для произвольного доступа к элементу массива по индексу требуется O(1) времени.

- **Поиск максимального элемента**

Процедура **max()** возвращает указатель на элемент кучи с максимальным ключом. Она работает следующим образом:

В неубывающей двоичной куче не предусмотрен быстрый поиск максимума, т.к. он может находится в любом её месте. Поэтому процедура просто перебирает элементы массива кучи в порядке возрастания индексов, сохраняя указатель на максимальный элемент по пути от начала к концу массива. Затем она возвращает этот указатель.

Данная процедура работает за время O(n), т.к. ей необходимо перебрать все n элементов массива кучи для того, чтобы найти максимальный среди них.

- **Поиск элемента по ключу**

Процедура **find(**key**)** возвращает указатель на элемент кучи со значением ключа, равным key. Она работает следующим образом:

В двоичной куче не предусмотрен быстрый поиск элемента, т.к. он может находится в любом её месте. Поэтому процедура просто перебирает элементы массива кучи в порядке возрастания индексов, пока не найдет элемент с ключом, равным key. Затем она возвращает указатель на этот элемент. Если же элемент не был найден, процедура возвращает нулевой указатель.

Данная процедура работает за время O(n), т.к. в худшем случае искомый элемент находится в конце массива кучи, и процедуре придется перебрать все n элементов массива кучи для того, чтобы найти его.

Дополнительные операции для биномиальной кучи:

- **Поиск максимального элемента**

Процедура **max()** возвращает указатель на элемент кучи с максимальным ключом. Она работает следующим образом:

В биномиальной куче не предусмотрен быстрый поиск максимума, т.к. он может находится в любом её месте. Поэтому процедура просто перебирает элементы кучи в порядке «текущий -> ребенок -> правый брат», сохраняя указатель на максимальный элемент на протяжении всего пути. Затем она возвращает этот указатель.

Данная процедура работает за время O(n), т.к. ей необходимо перебрать все n элементов массива кучи для того, чтобы найти максимальный среди них.

- **Поиск элемента по ключу**

Процедура **find(**key**)** возвращает указатель на элемент кучи со значением ключа, равным key. Она работает следующим образом:

В биномиальной не предусмотрен быстрый поиск элемента, т.к. он может находится в любом её месте. Поэтому процедура просто перебирает элементы кучи в порядке «текущий -> ребенок -> правый брат», пока не найдет элемент с ключом, равным key. Затем она возвращает указатель на этот элемент. Если же элемент не был найден, процедура возвращает нулевой указатель.

Данная процедура работает за время O(n), т.к. в худшем случае искомый элемент находится в конце кучи, и процедуре придется перебрать все n элементов кучи для того, чтобы найти его.

Операция print()

В классе обоих видов структур данных также определены операции print(), которые отражают внутреннее строение кучи. Их формат вывода описан в файле README.md, находящемся в репозитории.

Каждая операция над каждым видом кучи работает на месте, т.е. не создает дополнительных копий объектов и больших по объему данных, а работает только с текущим объектом, либо с указателями, поэтому работает с O(1) памяти.

Итоговое теоретическое сравнение времени работы всех основных операций над двоичной и биномиальной кучами представлено в таблице 1.

Таблица 1. Сравнение времени работы различных операций над структурами данных

Операция	Двоичная куча	Биномиальная куча
Добавление нового элемента	$O(\log_2 n)$	$O(\log_2 n)$
Извлечение минимального элемента	$O(\log_2 n)$	$O(\log_2 n)$
Слияние двух куч	$O(n)$	$O(\log_2 n)$
Поиск	$O(n)$	$O(n)$
Поиск максимального элемента	$O(n)$	$O(n)$
Поиск минимального элемента	$O(1)$	$O(\log_2 n)$
Удаление	$O(\log_2 n)$	$O(\log_2 n)$

Исходя из данных таблицы можно сделать вывод, что существенная разница во времени работы у двоичной и биномиальной куч будет в двух сценариях их использования:

1. Если будет часто использоваться операция слияния куч.
2. Если будет часто использоваться операция поиска минимального элемента.

Также стоит сравнить эти две структуры данных в сценариях, в которых часто используются операции удаления элементов, извлечения минимальных элементов и добавления новых элементов, т.к. формально времена работы этих процедур одинаковы, но на практике из-за разительно разных алгоритмов работы может оказаться, что теоретические ожидания неточные.

Сравнение проводилось с помощью файлов-тестов, содержащих набор соответствующих тесту команд. Эти файлы подавались на вход программы в качестве

аргументов, затем программа создавала файл, в который она записывала вывод всех поданных команд (если таковой имелся) и в конце файла выводила время своей работы в микросекундах.

Результаты сравнения средних времен работы отражены в таблице 2.

Таблица 2. Сравнение средних времен работы тестов структур данных

Тест	Двоичная куча	Биномиальная куча
test9, test10 (много вставок)	256 мкс	245 мкс
test11, test12 (много извлечений минимумов)	720 мкс	718 мкс
test13, test14 (много слияний)	305 мкс	258 мкс
test15, test16(много поисков минимумов)	460 мкс	461 мкс
test17, test18 (много удалений)	252 мкс	258 мкс

В колонке «Тесты» название первого теста – для двоичной кучи, второго – для биномиальной.

Исходя из данных таблицы можно сделать следующие выводы:

1. Операция **вставки** работает **примерно за одинаковое время** у обоих структур данных (подтверждение теоретического ожидания).
2. Операция **извлечения минимального элемента** работает **примерно за одинаковое время** у обоих структур данных (подтверждение теоретического ожидания).
3. Операция **слияния** двух куч работает **заметно быстрее у биномиальной кучи** (подтверждение теоретического ожидания).
4. Операция **поиска минимального элемента** работает **примерно за одинаковое время** у обоих структур данных (опровержение теоретического ожидания).
5. Операция **извлечения минимального элемента** работает **примерно за одинаковое время** у обоих структур данных (подтверждение теоретического ожидания).

Выводы

Практическое сравнение двух структур данных, двоичной кучи и биномиальной кучи, с помощью реализации таковых на языке c++ и тестирования различными сценариями подтвердило теоретические ожидания времен работы всех процедур, применяемых к данным структурам, за исключением поиска минимума, время работы которой оказалось примерно равно у обоих структур данных. Возможно, это связано с тем, что в тестах с множеством таких поисков также использовалось много процедур вставки элементов, т.к. кучи необходимо было сначала сформировать, а затем проводить в них поиск. Также повлиять на результат мог тот факт, что корневой список в биномиальной куче получился не такой большой, а значит, поиск в таком списке производился примерно за O(1), как и в двоичной куче, т.е. имел место лучший случай для данной процедуры.

В результате проведенной работы мной были изучены теоретические сведения о таких структурах данных, как двоичная куча и биномиальная куча, а также получены практические навыки по их реализации на объектно-ориентированном языке программирования c++.