



Горшенин Юрий Васильевич

Студент Московского физико-технического института (МФТИ), II курс, факультет инноватики высоких технологий (ФИБТ).

Бинарная пирамида

Цель этой статьи – познакомить читателя с одной из простых структур данных – бинарной пирамидой (её также называют бинарной кучей или **binary heap**). На её примере мы познакомимся с понятием интерфейса.

Задача. На вход программы подаётся N целых чисел. Необходимо найти K наибольших ($1 \leq N \leq 10^8$, $1 \leq K \leq \min\{1000, N\}$). Предполагается, что число N настолько большое, что нет возможности хранить все N чисел в памяти – они всё поступают и поступают на вход, и необходимо на каждый момент иметь ответ на вопрос задачи.

Самое простое решение следующее. Вы создаёте массив из K элементов и записываете в него первые K чисел из входных N . Теперь рассмотрим $(K+1)$ -е число. Если оно больше, чем минимальное из хранимых K чисел, то ставим его на место этого минимального. Аналогично поступаем с $K+2, K+3, \dots, N$ -м числами.

На каждом шаге нам, возможно, потребуется искать наименьшее среди хранимых K чисел. Если их хранить в неупорядоченном виде, то на поиск наименьшего будет тратиться порядка K элементарных действий.

Нам придётся сделать $(N - K)$ шагов, на каждом из которых в худшем случае будет сделано K элементарных действий. Значит, число элементарных действий можно оценить как $N \cdot K$. Посмотрите на ограничения, которым удовлетворяют N и K . Представьте, что миллион элементарных действий выполняется на компьютере за 1 секунду. Всего получается $10^8 \cdot 1000 / 10^6 = 10^5$ секунд, то есть порядка одного дня. Это достаточно долго. Даже если вы будете хранить упорядоченный массив K чисел, всё равно алгоритм будет неэффективным. Дело в том, что на вставку в упорядоченный массив требуется время порядка размера массива. Например, чтобы вставить элемент в середину массива размера K , необходимо $K/2$ последних чисел последовательно перенести на одну ячейку вперёд и в образовавшуюся свободную ячейку поместить добавляемый элемент.

Таким образом, как бы мы ни хранили наши K чисел, в упорядоченном или неупорядоченном массиве, в любом случае на решение задачи в худшем случае уйдёт время примерно пропорциональное $N \cdot K$.

Необходимо K максимальных чисел хранить в специальном хранилище, которое позволяло бы быстро узнавать минимальное из этих чисел, извлекать это минимальное и добавлять новое число. То есть нам хотелось бы иметь три функции для работы с нашим хранилищем:

- **min()** – возвращает текущий минимум;

insert(x) – добавляет в хранилище новое число x ;

- **extract_min(x)** – возвращает текущий минимум и удаляет его из хранилища.

Эти три функции и есть *интерфейс*, который должно *реализовывать* наше хранилище. О том, что такое интерфейс вообще, мы поговорим ниже.

Желательно, чтобы все три указанные функции работали быстрее, чем за время, пропорциональное K .

Бинарная пирамида и является подходящей структурой, которая предоставляет указанный интерфейс.

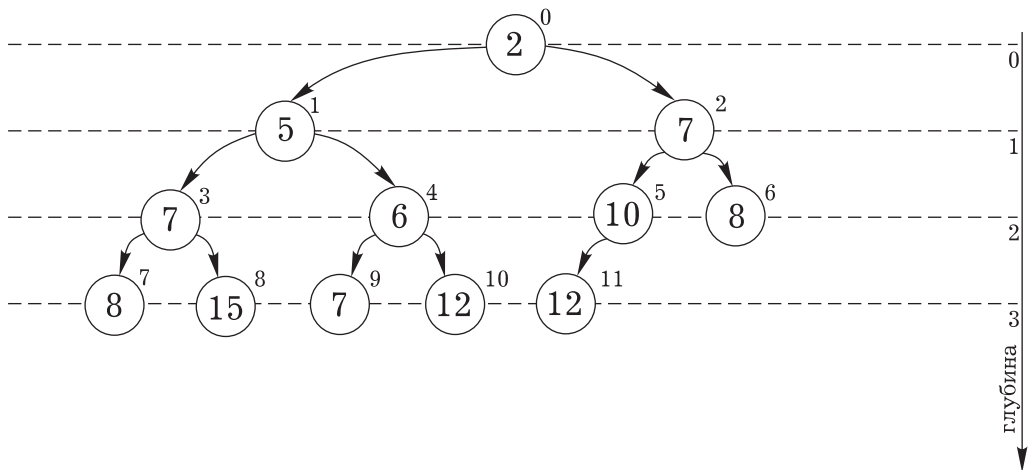


Рис. 1. Изображение бинарной пирамиды

Итак, посмотрите на рисунок 1. На нём изображено **дерево** – нечто, что если перевернуть вверх ногами, весьма похоже на обычное дерево. Кружочки будем называть **узлами**. Числа, содержащиеся в кружках – **ключами** данного узла. Узлы занумеруем целыми неотрицательными числами в порядке возрастания – слева направо, сверху вниз. Номера узлов написаны справа сверху от кружка. Узел, имеющий номер 0, будем называть **корневым**.

Обратите внимание, что из некоторых узлов идут стрелки к другим узлам. Узел является **родителем** по отношению к тем узлам, к которым от него идут стрелки. Эти узлы называются **детьми** этого узла. Из каждого узла исходит не более двух стрелок. Дерево, у каждого узла которого не более двух детей, называется **бинарным**.

Примеры: узел с номером 4 является родителем по отношению к узлам с номерами 9 и 10; в свою оче-

редь, узел под номером 1 является родителем узла с номером 4.



Узлы, не имеющие детей, будем называть **листьями**. Листьями являются узлы 7, 8, 9, 10 и 11. Обратите внимание на штриховые линии – каждая линия обозначает **глубину** узлов, через которые она проходит. То есть узел с номером 0 имеет глубину 0, узлы с номерами 1 и 2 имеют глубину 1, узлы 3, 4, 5, 6 – глубину 2 и т.д.

А теперь самое главное – то, что,

Хранение бинарной пирамиды

Далее при реализации функций **insert(x)** и **extract_min()** мы добьёмся того, что уровни дерева заполняются последовательно, и новый уровень не будет начат, если ещё не заполнен предыдущий.

Для дерева, в котором уровни последовательно заполняются до конца, достаточно всего двух вещей – массива и целого числа, указывающего, сколько узлов у дерева на данный момент. Элементами массива будут узлы этого дерева – номер узла будет индексом массива. В нашем примере, изображённом на рисунке 1, по индексу 0 в массиве

наверное, вы уже заметили, – для каждого узла ключ, содержащийся в нём, не больше, чем ключи, содержащиеся в его детях. Это и есть то ограничение, которое мы накладываем на двоичную пирамиду.

Свойство бинарной пирамиды: ключ родителя меньше либо равен ключам детей.

Из этого непосредственно следует, что в корне бинарной пирамиды находится наименьший ключ!

Чтобы узнать минимальное число из хранимых (функция **min()**), необходимо просто заглянуть в корень.

Нам предстоит решить следующие вопросы.

- Как организовать хранение этого бинарного дерева в программе? Красивое дерево легко нарисовать на бумаге, а как его хранить в памяти компьютера, как его описать в программе?

- Как при выполнении операции добавления нового элемента в кучу поддерживать выполнение указанного выше свойства?

ве будет храниться значение 2, по индексу 1 – значение 5, по индексу 2 – значение 7 и т. д.

Пусть мы знаем номер текущего узла. Как получить номера **левого** и **правого** детей? И как по номеру текущего узла получить номер его родителя? Ответить на эти вопросы вы сможете сами, если попытаетесь найти закономерность в индексах на рисунке с изображением дерева: индексы детей можно получить из индекса родителя, умножив его на 2 и прибавив к результату 1 или 2. Представленные ниже функции **left** и

right возвращают индексы детей по индексу родителя. Функция **parent**

возвращает индекс родителя по индексу ребёнка.

```
inline int left(int cur) { return 2 * cur + 1; }  
inline int right(int cur) { return 2 * cur + 2; }  
inline int parent(int cur) { return (cur - 1) / 2; }
```

Эти функции объявлены как встраиваемые (**inline**), т.е. в местах их вызова компилятор подставит их содержимое. Обратите внимание, что для корневого узла индекс родителя будет равен 0, то есть он же сам, а дети корневого узла – это узлы с индексами 1 и 2. Для узла 1, в свою очередь, детьми будут узлы с индексами 3 и 4, а родителем – узел с индексом 0. Для узла 2 детьми будут 5 и 6, а родителем – 0 и т.д. Проверьте работу этих функций по рисунку.

Теперь вам должен быть уже ясен смысл рисунка 2 – дерево из рисунка 1 «упаковано» в линейный

вид. Сверху стрелки от родителя к левому потомку, снизу – к правому. Теперь мы обладаем возможностью перемещаться по пирамиде. Воспользуемся этим и напишем функцию «красивого» вывода пирамиды на экран. Это важно, чтобы при отладке кода иметь возможность быстро проверить, что всё хорошо. Одна из популярных концепций разработки кода заключается в том, что прежде чем что-либо разрабатывать, необходимо создать функции для отладки и тестирования и визуализации (анализа) результатов.

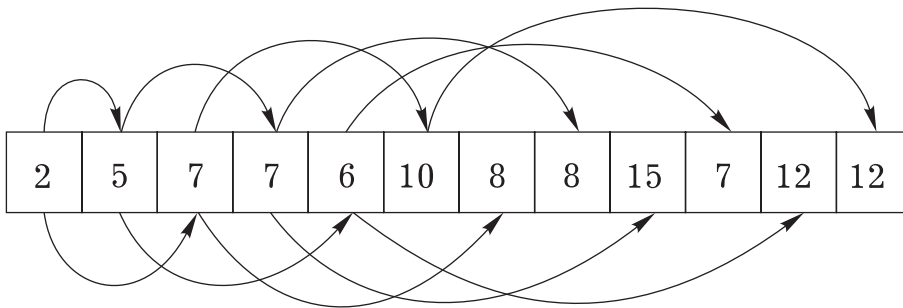


Рис. 2. Линейное представление бинарной пирамиды

```
void print_heap(int *heap, int size, int cur, int level)  
{  
    int i;  
    if(cur >= size)  
        return;  
    print_heap(heap, size, right(cur), level + 1);  
    for(i = 0; i < level; i++)  
        printf("\t");  
    printf("%d\n", heap[cur]);  
    print_heap(heap, size, left(cur), level + 1);  
}
```

Это рекурсивная функция. Параметры – сама пирамида (указатель на начало массива), её размер (число хранимых в ней элементов), текущее положение – индекс текущего узла, и – искусственный параметр «уровня» (**level**) – количество отступов для текущего узла при выводе его на экран. В начале

```
print_heap(heap, size, 0, 0);
```

где **heap** – массив, **size** – его размер в элементах. В языке Си имя массива является указателем на его первый элемент. Этого вполне достаточно для доступа по индексу. Теперь вы сможете вывести произвольный массив целых чисел в древовидном виде. Для массива {2, 3, 4, 4, 5, 6, 12, 17} такой вывод будет следующим:

```

          12
        4
       6
      2
     5
    3
   4
  17
```

Поверните свою голову на 90 градусов по часовой стрелке и, возможно,



идёт проверка, не вышли ли мы за пределы массива. Затем рекурсивный вызов для левого поддерева, вывод **level** пробелов на экран, вывод значения в текущем узле и, наконец, рекурсивный вызов для правого поддерева. Пользоваться этой функцией Вы должны следующим образом:

вы найдёте сходство с рисунком 1.

Обратите внимание, что какой бы массив вы ни выводили на экран с такой функцией, он получается как ровно подстриженный куст. Идеальные кусты получаются, если число элементов равно $2^n - 1$, где n – натуральное число. **Высотой** пирамиды будем называть самую большую глубину среди глубин его узлов. На рисунке пирамида имеет **высоту** 2. Пирамида из $2^n - 1$ узлов будет иметь высоту $n - 1$ (убедитесь в этом). Оценить высоту пирамиды из n узлов можно как $\log_2(n)$ ($\log_2(n)$ – логарифмическая функция, которая равна степени, в которую нужно возвести 2, чтобы получить n).

Хорошо. Пусть у нас есть двоичная пирамида. Пусть в каком-то из узлов значение ключа изменилось – уменьшилось или увеличилось, или в конец пирамиды (в конец представляющего её массива) добавлен новый элемент. В результате изменений упомянутое свойство пирамиды может оказаться нарушенным, а именно, изменённый узел может оказаться 1) больше своих детей или 2) легче своего родителя.

Для того чтобы исправлять нарушения первого типа, определим функцию **heapify**:

```
void heapify(int *heap, int cur, int size)
{
    int best = cur;
    if (left(cur) < size && heap[left(cur)] < heap[best])
        best = left(cur);
    if (right(cur) < size && heap[right(cur)] < heap[best])
        best = right(cur);
    if (best != cur) {
        int tmp = heap[cur];
        heap[cur] = heap[best];
        heap[best] = tmp;
        heapify(heap, best, size);
    }
}
```

Аргументы функции – указатель на начало массива пирамиды, номер узла, в котором значение ключа увеличилось, размер пирамиды в элементах – для контроля выхода за границы массива пирамиды. Среди родителя и его детей выбирается узел с наименьшим значением ключа. Если этот узел не есть родитель, то он «всплывает» на место родительского, а родитель опускается на его место. Затем нужно перейти в узел, где оказался родитель, и проверить, вдруг он снова тяжелее одного из своих детей, и так далее.

Эти повторные проверки мы осуществляем с помощью рекурсивных вызовов. При каждом новом рекурсивном вызове мы спускаемся всё глубже и глубже – ведь каждый ребёнок ровно на 1 находится глубже своего родителя. Значит, оценочно число рекурсивных вызовов не более, чем высота пирамиды. Это самый наихудший

случай – если «утяжелелся» корневой узел и начал тонуть до самого «дна» пирамиды. На рисунке 3 показано, как должен тонуть в изначально правильной пирамиде «утяжелевшийся» узел.

Аналогично можно решить проблему уменьшения ключа. Если узел становится легче, чем его родитель, то он «всплывает» на его место – меняется с ним местами. Причём родитель дальше «тонуть» не должен, так как до изменения он был легче ребёнка, а значит, легче и детей своего ребёнка. После выполнения этого шага возможно только одно нарушение – в узле, куда переместился наш облегчённый элемент. Возможно, он легче, чем его новый родитель, и нужно рекурсивно продолжить «всплывание». Функция **heap_decrease_key** будет уменьшать значение ключа с индексом **cur** и осуществлять его «всплывание» по мере необходимости.

```
void heap_decrease_key(int *heap, int cur, int newkey)
{
    while (cur && heap[parent(cur)] > newkey) {
        heap[cur] = heap[parent(cur)];
        cur = parent(cur);
    }
    heap[cur] = newkey;
}
```

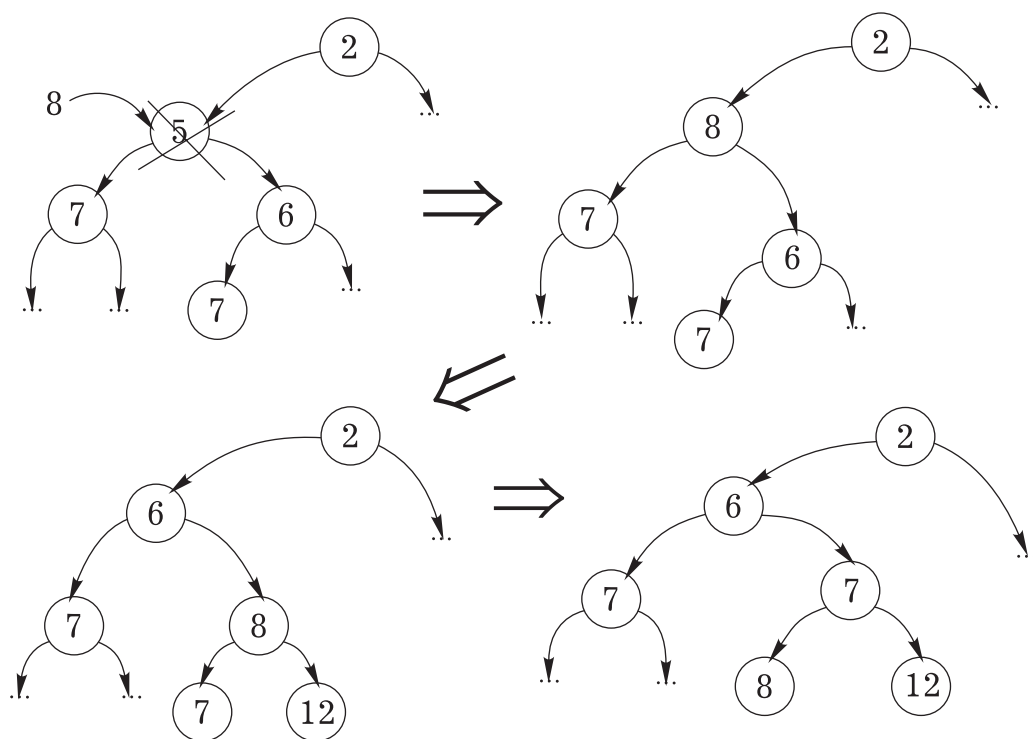


Рис. 3

Аргументы функции – указатель на начало массива, индекс узла, в котором **нужно** произвести изменение (уменьшение) ключа, и новое значение ключа. Здесь вместо рекурсии используется метод итераций.

Итак, мы умеем исправлять в существующей пирамиде нарушения свойства пирамиды. Но дело в том, что пока на руках у нас нет метода конструирования пирамиды. Ну что же, не беда. Сейчас будет. Пустая пирамида соответствует пустому массиву. Пусть у нас есть некоторая пирамида. Как добавить в неё очередной новый элемент? В самый конец массива дописываем элемент со значением ключа, заведомо превосходящим все значения, уже хранящиеся в куче. Например, $+\infty$. Ясно,

что никакого нарушения свойств кучи это не сделает. А теперь изменим значение этого ключа на то, которое нам нужно при помощи **heap_decrease_key**. Надо также не забыть увеличить размер кучи на 1.

В случае удаления наименьшего значения (которое находится в корневом узле) мы просто на его место перемещаем самый последний элемент массива и уменьшаем на один переменную, в которой хранится размер пирамиды. Далее нужно восстановить свойство бинарной пирамиды, которое возможно было нарушено, после перемещения последнего узла в вершину. Для этого нужно вызвать **heapify** для индекса 0. Все эти идеи представлены ниже в виде функций:

```
void heap_insert(int *heap, int *size, int newkey)
{
```

```
heap[*size] = INT_MAX;  
heap_decrease_key(heap, *size++, newkey);  
}  
  
int heap_extract_min(int *heap, int *size)  
{  
    int res  
    if(*size < 1)  
        return;  
    res = heap[0];  
    heap[0] = heap[--*size];  
    heapify(heap, 0, *size);  
    return res;  
}
```

Таким образом, мы научились добавлять новый элемент и извлекать из пирамиды самый наименьший элемент, причём и то, и другое за время, пропорциональное высоте пирамиды.

Возвратимся к исходной задаче. Для решения достаточно вместо массива организовать бинарную пирамиду из первых K элементов. Рассматривая каждый новый элемент, мы сравниваем его с наименьшим элементом в пирамиде на данный момент. Если больше, то этот наименьший элемент из пирамиды удаляем и вставляем новый со значением, равным значению рассматриваемого элемента. Иначе – ничего не делаем. Продолжаем выполнение алгоритма. Так как на вставку одного элемента затрачивается число шагов, пропорциональное высоте кучи, то есть $\log(K)$, и число шагов равно N , то мы получаем оценочно число элементарных шагов $N\log(K)$, то есть что-то порядка 10^9 , или 17 минут.

Однако это была весьма специфичная задача. А вот сортировка как таковая применяется широко в качестве одного из шагов для многих алгоритмов. Двоичная пирамида позволяет нам разработать простой алгоритм сортировки. Создав кучу из N элементов, мы можем последовательно забирать наименьший из элементов кучи. Другими словами, серия



из N вставок в кучу и N извлечений минимума приведёт к тому, что мы получим ту же последовательность, но в отсортированном по возрастанию порядке. Причём на все эти действия в конечном итоге мы затратим порядка $N\log(N)$ элементарных шагов. Причём эта оценка верна как для среднего, так и для худшего случая. Доказано, что $N\log(N)$ является лучшей асимптотикой для задачи сортировки произвольных чисел на однопроцессорной машине.

Пусть f и g – функции натурального аргумента n . Будем говорить, что $f(n) = O(g(n))$, если

$$\exists C \in \mathbb{R}, \exists n_0 \in \mathbb{N} : \forall n \in \mathbb{N},$$

$$n \geq n_0 \rightarrow f(n) \leq Cg(n).$$

Строго говоря, знак равенства не следует воспринимать действительно

как знак равенства (иначе, можно придти к уникальным результатам вроде $n = O(n) = \log(n)!$). Вместо знака равенства следовало бы писать знак принадлежности к множеству – действительно, $O(g(n))$ – это целое множество функций. Однако это договорённость среди математиков, не будем её нарушать. В силу того, что « $=$ » на самом деле вовсе не « $=$ », а « $\in$ », не следует писать символы O в левых частях равенств. Это всё необходимо для более простой оценки эффективности алгоритмов. В качестве n может быть размерность задачи. Например, при сортировке n – это количество сортируемых чисел, для работы с графами появляются две размерности – число вершин и рёбер в графе и т.д. Мы делаем упрощённую оценку, так как нас, как правило, интересует поведение времени выполнения алгоритма при увеличении размерностей задачи. Строго говоря, **точно** оценить время весьма сложно. Для этого пришлось бы писать программу на машинном языке, знать, сколько тактов процессора уходит на конкретную инструкцию, проводить

весьма сложный анализ поведения алгоритма и так далее. Это нетривиальная задача. Впрочем, для большинства задач нам достаточно лишь приблизительно оценить время работы алгоритма.

Теперь мы можем пользоваться «более математическими» терминами при описании пирамиды. Скажем, время работы функции **heapify** для кучи из n элементов можно записать как $O(\log(n))$. Такая же оценка времени для функции **heap_decrease_key**, **heap_insert**, **heap_extract_min**. Двоичную пирамиду для заданного набора чисел можно создать последовательными вставками. Оценка всех этих действий – $O(n \log(n))$. Однако если элементы поступают не последовательно, а, скажем, уже расположены в памяти в виде массива, то есть более эффективный алгоритм получения из них кучи. А именно, существует алгоритм, который за время $O(n)$ переставляет элементы в массиве n элементов так, что он становится бинарной пирамидой. Ниже он представлен в виде функции **make_heap**:

```
void make_heap(int *heap, int size)
{
    int i;
    if (!size) return;
    for (i = parent(size - 1); i >= 0; i--)
        heapify(heap, i, size);
}
```

Казалось бы – $O(n)$ вызовов функции **heapify**, каждый вызов обходится в $O(\log(n))$. Т.е. время работы этого алгоритма сверху можно оценить как $O(n \log(n))$. На самом деле есть более точная оценка – $O(\log(n) \cdot d)$, где d – глубина узла, начиная с которого происходит вызов

heapify. Кстати, в худшем случае на глубине d находится 2^d узлов. Доказательство **линейности** времени работы достаточно поучительно – для этого достаточно лишь просуммировать временную сложность **heapify** для всех узлов:

$$\begin{aligned}
T(n) &\leq \sum_{d=0}^{\log(n)} 2^d (\log(n) - d) = \log(n) \sum_{d=0}^{\log(n)} 2^d - \sum_{d=0}^{\log(n)} d 2^d; \\
\sum_{d=0}^{\log(n)} 2^d &= \frac{1(2^{\log(n)+1} - 1)}{2 - 1} = 2n - 1; \\
\sum_{d=0}^{\log(n)} d 2^d &= 2 \sum_{d=0}^{\log(n)} d 2^{d-1}; \quad P_n(x) = \sum_{i=0}^n x^i = \frac{x^{n+1} - 1}{x - 1}; \\
\sum_{i=0}^n i x^{i-1} &= \frac{d}{dx} P_n(x) = \frac{(n+1)x^n(x-1) - (x^{n+1} - 1)}{(x-1)^2} = \frac{nx^{n+1} - (n+1)x^n + 1}{(x-1)^2}; \\
2 \sum_{d=0}^{\log(n)} d 2^{d-1} &= 2(2n \log(n) - n(\log(n) + 1) + 1) = 2n \log(n) - 2n + 2. \\
T(n) &\leq \log(n)(2n - 1) - (2n \log(n) - 2n + 2) = 2n - \log(n) - 2 \leq 2n = O(n).
\end{aligned}$$

Напоследок хочется отметить, что двоичная пирамида лишь одна из возможных реализаций очереди с приоритетом. Очередь с приоритетом можно рассматривать как абстрактную структуру данных, предоставляющую функции **insert** (добавление элемента), **extract_min** (извлечения элемента с минимальным ключом), а также функции создания новой пустой очереди, удаление элемента из очереди по ключу, уменьшение приоритета элемента с заданным ключом, объединение двух очередей в одну.

У всех этих реализаций очереди с приоритетом один и тот же набор публичных функций, то есть функций, предоставляемых программисту, пользующемуся структурой данных.

Существуют различные реализации очереди с приоритетами: бинарная пирамида, биномиальная пирамида, фибоначиева пирамида.

Разные реализации этого интерфейса могут давать различные асимптотики для средних времён работы функций. Если вы используете в программе очередь с приоритетом, то правильно выделите её в отдельный модуль (например, абстрактный класс в языке C++ или интерфейс в языке Java). Далее можно сделать так, чтобы соответствующие функции в разных реализациях имели одинаковые имена, одинаковые аргументы, возвращаемые значения и имена. Программе, использующей очередь с приоритетом, станет всё равно, какую из реализаций интерфейса очереди с приоритетом вы ей «подсунете». Ей важно, что выполняют эти функции, а что скрывается «под капотом» этих модулей – всё равно. Понятие *интерфейса* позволяет программисту абстрагироваться от реализации отдельных модулей и работать с более высокоуровневыми сущностями.

Рекомендуемая литература

1. «Алгоритмы: построение и анализ» Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штейн. Здесь вы найдёте подробные описания всех трёх реализаций очереди с приоритетом.

2. «Искусство программирования» Д. Кнут, т. 1–3 – бесценный вклад в Computer Science – замечательно рассмотрены методы анализа алгоритмов, плюс сами алгоритмы. Особенно полезен 3-й том, посвящённый сортировке и поиску.