Двоичная куча (пирамида). Пирамидальная сортировка. Приоритетная очередь

И. Григорьев

Рекомендуемый порядок изучения

По мере изучения теоретического материала рекомендуется решать включенные в него упражнения. Если упражнение предполагает написание программы, то её правильность можно проверить с помощью автоматической тестирующей системы. Для этого надо найти соответствующую задачу в разделе «Задачи на реализацию кучи», решить её (в случае затруднений можно посмотреть раздел «Подсказки») и сдать решение в тестирующую систему. Лишь после этого рекомендуется читать решение этого упражнения.

Решения некоторых упражнений содержат важные дополнения к теоретическому материалу, поэтому их рекомендуется прочитать даже в том случае, если тестирующая система не нашла ошибок в вашем решении.

1. Введение

Вероятно, читателю известны такие простые методы сортировки массива, как *сортировка вставками* и *сортировка выбором*. Сложность обоих этих алгоритмов – $O(n^2)$, где n – размер массива. Возможно, читатель знаком и с *сортировкой слиянием* (работающей за $O(n \log n)$), которую можно рассматривать как результат усовершенствования (с применением принципа балансировки) *сортировки вставками*.

Здесь мы получим другой метод сортировки за $O(n \log n)$, который можно считать усовершенствованной *сортировкой выбором*. Для его реализации понадобится структура данных, называемая *двоичной (бинарной) кучей* или *пирамидой*. Одновременно с этим окажется удобно изучить также основанную на куче *приоритетную очередь*.

Для определённости будем сортировать массив по возрастанию. (Точнее, по неубыванию, так как в массиве могут быть совпадающие элементы).

2. От сортировки выбором к пирамидальной сортировке

2.1. Основная схема сортировки выбором сохраняется. Массив разделен на две части, одна из которых (правая) уже отсортирована, причем любой элемент отсортированной части не меньше любого элемента неотсортированной:

```
пока размер неотсортированной части > 1 выполнять
| найти максимальный элемент
| переставить его в конец отсортированной части
конец
```

2.2. Что здесь можно ускорить? Только поиск максимального элемента: ведь перенос элемента и так занимает $O(1)$ (в расчёте на весь массив – $O(n)$).

В простой сортировке выбором мы за k-1 сравнение (где k – длина неотсортированной части) всего лишь находили максимальный (минимальный) элемент. При этом мы часто получали информацию про относительные величины некоторых пар элементов, но «забывали» её. В пирамидальной сортировке эта информация будет сохраняться.

Другими словами: неотсортированная часть будет не совсем беспорядочной — она будет иметь внутреннюю структуру, которая позволит каждый раз быстро выбирать максимальный элемент. (Как быстро? Чтобы сложность сортировки была $O(n \log n)$, достаточно, чтобы время выбора было $O(\log n)$). Забегая вперёд, отметим, что эту внутреннюю структуру нам обеспечит *двоичная пирамида,* или *куча*[1]. Основная идея кучи состоит в том, что мы рас-

[1] По-английски эта структура данных называется *heap*, дословный перевод – *куча*, но термин *пирамида*, который тоже иногда используется, лучше соответствует сути де-

сматриваем массив как представление двоичного дерева и вводим некоторый порядок на узлах этого дерева.

2.3. Тогда схема алгоритма сортировки будет такой:

```
преобразовать массив в кучу (пирамиду)
пока размер кучи > 1 выполнять
| выбрать максимальный элемент и перенести его в
| отсортированную часть
конец
```

3. Приоритетная очередь

Легко заметить, что задача реализации кучи родственна задаче эффективной реализации **приоритетной очереди**. (Это очередь, в которой важно не то, кто «встал» раньше, а кто «главнее». Более точно: при помещении в очередь указывается приоритет помещаемого элемента, а при взятии из очереди выбирается один из элементов с наибольшим приоритетом. В учебных задачах для простоты приоритетом обычно служит само значение элемента).

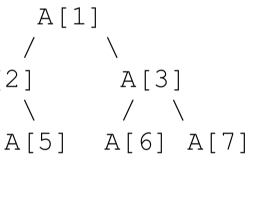
Ведь и в сортировке выбором, и в реализации приоритетной очереди требуется быстро извлекать максимальный элемент. Однако приоритетная очередь, сверх того, должна уметь быстро добавлять новый элемент, не нарушая своей внутренней структуры.

Будет удобно на время отвлечься от сортировки и разобрать подробно реализацию приоритетной очереди на основе кучи. К сортировке мы вернёмся в пункте 10.

Определение кучи будет дано в пункте 5, а пункт 4 подготовит нас к нему.

4. Массив и двоичное дерево

Следующая картинка показывает, как можно отобразить элементы массива A на вершины двоичного дерева.



Заметим, что при этом получается дерево вполне определённой формы: все его уровни, кроме, возможно, самого нижнего, заполнены целиком. А на самом нижнем уровне все пустые места, если они есть, располагаются правее имеющихся вершин. Будем называть такое двоичное дерево *почти полным*.

ла. Рассматриваемый здесь алгоритм сортировки называют *сортировкой с помощью кучи*, *пирамидальной сортировкой* или, реже, сортировкой деревом.

Из рисунка нетрудно видеть, что родитель вершины с индексом i имеет индекс $[i/2]$, а левая и правая дочерние вершины имеют индексы $2i$ и $2i+1$ соответственно[2].

Высотой дерева будем называть число уровней в дереве (или, что то же самое, длину пути от корня до самого дальнего листа плюс один). Так, для дерева на рисунке длина пути от корня до листа $A[8]$ равна 3, а высота равна 4.

Полезно установить связь между высотой h и числом вершин дерева s.

Упражнение 1. Сколько вершин может иметь почти полное двоичное дерево высоты h?

Упражнение 2. Выразите высоту почти полного двоичного дерева через число вершин.

5. Двоичная куча (пирамида)

Чтобы быстро искать максимум в этом дереве, расположим элементы массива так, чтобы значение любого элемента было не меньше значений всех его потомков (если они существуют).

Для этого достаточно, чтобы для каждого i выполнялись два неравенства: $A[i] \ge A[2i]$ (если $2i \le s$), и $A[i] \ge A[2i+1]$ (если $2i+1 \le s$).

Вообще говоря, часто будет удобно, чтобы этому правилу подчинялись не обязательно все элементы массива, а лишь элементы его некоторого начального участка. Этот участок и будем называть кучей. Будем обозначать размер всего массива n, а размер кучи – s. Размер кучи может меняться от 0 до n включительно.

Теперь можно дать **определение кучи.**

Двоичной максимальной кучей будем называть некоторое *начало* A[1]…A[s] массива A [1]…A[n], $(0 \le s \le n)$ если каждый его элемент обладает ***основным свойством максимальной кучи***: его значение *не меньше* значений всех его потомков.

Здесь все неравенства нестрогие: в куче могут встречаться равные элементы.

Нетрудно доказать, что максимальный элемент в такой куче имеет индекс 1 в массиве (находится в корне дерева).

Если потребовать, чтобы элемент был *не больше* своих потомков, получим *минимальную кучу*. (Мы будем рассматривать, в основном, максимальные кучи).

Обратите внимание, что «физически» куча – это участок массива. А «логически» её удобно рассматривать как двоичное дерево.

[2] Здесь квадратные скобки обозначают целую часть числа, то есть округление вниз до ближайшего целого.

6. Основные операции с кучей

Как уже отмечалось, куча будет применяться нами в двух задачах:
1) Сортировка массива (без дополнительной памяти, за время $O(n \log n)$) — процедура *HeapSort* («пирамидальная сортировка», «сортировка деревом», «сортировка с помощью кучи»).
2) Реализация *очереди с приоритетами* — процедуры *Extract_Max* (извлечь максимальный) и *Insert* (добавить элемент). (За время $O(\log n)$).

Понятно, что если будет реализована приоритетная очередь, то с её помощью можно будет отсортировать массив в два этапа. На первом этапе – поместить все сортируемые элементы в кучу многократными вызовами процедуры *Insert*, а на втором – извлечь их все многократными вызовами *Extract_Max*. При этом каждый этап будет выполняться не дольше, чем за $O(n \log n)$).

Впрочем, преобразование массива в кучу (назовём эту операцию *Build_Heap*) можно (и нужно!) делать и по-другому. Мы обсудим это в п. 9.

6.1. Реализация *Extract_Max*

Пусть имеется куча размера $s \in [1; n]$. Найти максимальный очень просто – это $A[1]$. Для его удаления требуется переставить остальные элементы кучи так, чтобы выполнялись три условия:
1) освободилось место $A[s]$ (поскольку размер кучи должен уменьшиться);
2) оказалось занято место $A[1]$
3) для всех $A[1], \ldots, A[s\text{-}1]$ по-прежнему выполнялось основное свойство кучи.

Если временно забыть про условие 3, то первые два выполнить очень просто: надо всего лишь переставить последний элемент на 1-е место: $A[1]:=A[s]$ и уменьшить на 1 размер кучи s. Тогда мы получим «немного испорченную кучу», которую затем починим. Таким образом, процедура *Extract_Max* будет выглядеть так:

```
ответ := A[1]
A[1] := A[s];
s := s - 1;
Восстановить кучу
```

О том, как восстанавливать кучу – ниже.

6.2. Реализация *Insert*

Пусть имеется куча размера $s \in [0; n\text{-}1]$. Добавлять элемент x в кучу будем так:

```
s := s + 1;       (увеличили размер кучи
A[s] = x;          и поставили x в её конец )
Восстановить кучу
```


7. Восстановление «немного испорченной» кучи

Заметим, что после выполнения как первых трёх строк процедуры *Extract_Max,* так и первых двух строк процедуры *Insert* мы получаем кучу, испорченную совсем немного. В обоих случаях есть не больше одного «неправильного» элемента. В первом случае это $A[1]$, а во втором – $A[s]$. В первом случае его значение слишком мало для занимаемого им места, а во втором – слишком велико[3]. В первом случае «неправильный» элемент нужно спустить вниз, а во втором – поднять вверх. Некоторым другим элементам при этом, естественно, тоже придётся подвинуться, но нам будет удобно следить, в первую очередь, именно за перемещением «неправильного» элемента.

Соответственно, нам понадобятся две процедуры для «починки» кучи, которые назовём *Sift_Down* (просеивание вниз) и *Sift_Up* (просеивание вверх).

Каждую из них будем реализовывать для более общего случая, чем вроде бы требуется. А именно, будем считать, что «неправильным» может оказаться любой элемент кучи, а не только первый или последний (зачем это нужно, станет ясно из дальнейшего). Однако по-прежнему будем требовать, чтобы такой элемент был только один.

Вот спецификация процедуры *Sift_Down*:
Дано: «немного испорченная куча» $A[1]\ldots A[s]$, ($s \in [1; n]$), индекс $i \in [1; s]$. Известно, что куча испорчена тем, что элемент $A[i]$, возможно, имеет *меньшее* значение, чем требуется для занимаемого им места[4].
Надо: переставить некоторые из элементов $A[1]\ldots A[s]$ так, чтобы для них всех стало выполняться основное свойство кучи.

Спецификация *Sift_Up* будет отличаться лишь заменой слова «меньшее» на «большее».

Обе процедуры должны работать за $O(h)$, где h – высота кучи. Поскольку $h = O(\log n)$ (см. упражнение 2), то это обеспечит работу операций *Extract_Max* и *Insert* за $O(\log n)$.

Читателю предлагается самостоятельно реализовать эти процедуры (рекомендуется начать с *Sift_Up*, поскольку она проще).

Упражнение 3. Напишите реализацию процедуры *Sift_Up*. (Для её автоматического тестирования предназначена задача «Увеличение приоритета»).

Упражнение 4. Напишите реализацию процедур *Sift_Down* и *Extract_Max*. (Для автоматического тестирования предназначены задачи «Уменьшение

---

[3] Точнее, следовало бы сказать «возможно, слишком мало», «возможно, слишком велико». Потому что не исключен и случай «везения», когда после описанных выше манипуляций куча совсем не будет испорчена.

[4] Если угодно, более формально это можно сформулировать так. Найдётся такое значение $b \geq A[i]$, что если выполнить присваивание $A[i] := b$, то основное свойство кучи будет выполняться для всех $A[1]\ldots A[s]$.

приоритета» и «*Извлечение максимального*». Рекомендуется также решить задачу «*Приоритетная очередь*»).

8. Дополнительные операции

На основе *Sift_Up* и *Sift_Down* могут быть реализованы ещё несколько операций, которые тоже бывают полезны в некоторых задачах.

8.1. *Change_Priority*. Изменяет приоритет указанного элемента. В качестве входных параметров получает индекс элемента *i* и его новое значение *x*. Реализация очевидна:

```
if x > A[i] then begin
  A[i] := x;      Sift_Up(i);
end else begin
  A[i] := x;      Sift_Down(i);
end;
```
[5]

Иногда для увеличения и уменьшения приоритета используют две отдельные процедуры: *Increase_Priority* (увеличить) и *Decrease_Priority* (уменьшить).

8.2. *Delete*. Удаляет указанный элемент из кучи. Входной параметр – индекс удаляемого элемента. Реализовать предлагается самостоятельно. В качестве подсказки см. реализацию *Extract_Max*.

Упражнение 5. Напишите процедуру *Delete*. (Для автоматического тестирования предназначена задача «Приоритетная очередь с удалением».)

9. Построение кучи из массива

9.1. Напомним, что первый этап сортировки с помощью кучи состоит в преобразовании всего массива в кучу. Не исключено, что эта же операция может оказаться полезной и для начального построения приоритетной очереди, если требуется сразу поместить в очередь много элементов: вместо многократного повторения операции *Insert* можно записать все эти элементы в начальный участок массива и преобразовать его в кучу. Вопрос в том, можно ли это преобразование выполнить быстрее, чем выполняется простое многократное повторение *Insert*?

До сих пор все операции с кучей были основаны на двух базовых – просеивании вверх (*Sift_Up*) и просеивании вниз (*Sift_Down*). Давайте попробуем с

[5] Как правило, реализации *Sift_Up* и *Sift_Down* устроены так, что ничего не делают, если значение просеиваемого элемента отклоняется в другую сторону, чем требуется по спецификации. Это позволяет упростить *ChangePriority*:

```
A[i] := x;
Sift_Up(i);
Sift_Down(i);
```

Вызовы *Sift_Up* и *Sift_Down* могут идти здесь в любом порядке, причём, по крайней мере один из них никакой полезной работы не делает. Но, строго говоря, при этом следует внести соответствующие изменения в их спецификации.

помощь каждой из них написать и процедуру *Build_Heap*.

Рекомендуется сначала попытаться сделать это самостоятельно, решив три следующих упражнения (решения первых двух можно проверить, сдав их в тестирующую систему), а потом прочитать их разбор в разделах «Подсказки» и «Ответы и решения».

Упражнение 6. *Build_Heap*1. Реализуйте процедуру преобразования массива в кучу с помощью просеивания вверх *Sift_Up* (для тестирования – задача 6).

Упражнение 7. *Build_Heap*2. Реализуйте процедуру преобразования массива в кучу с помощью просеивания вниз *Sift_Down* (для тестирования – задача 7).

Упражнение 8. Какая из этих реализаций *Build_Heap* работает быстрее? Почему? (Качественно это можно понять, даже если ваши знания математики не позволяют количественно оценить разницу в их времени работы).

Упражнение 9*. (Для хорошо знающих математику) Довольно очевидно, что каждая из реализаций *Build_Heap* работает не дольше, чем за $O(n \log n)$, поскольку выполняется $O(n)$ вызовов, каждый из которых работает за $O(h)$.

Однако не исключено, что если посчитать точнее, то можно найти лучшую оценку времени – ведь только последние вызовы происходят на куче размера, близкого к *n*, а первые выполняются на куче меньшего размера и, соответственно, занимают меньше времени.

Чтобы это выяснить, надо для каждой реализации сделать одно из двух:

- Доказать, что, по крайней мере в некоторых случаях, время работы действительно имеет порядок $n \log n$ (Это принято записывать $\Theta(n \log n)$).
- Найти и доказать более точную оценку времени, которая меньше, чем $\Theta(n \log n)$.

10. Пирамидальная сортировка

Общая схема алгоритма была приведена в пункте 2.3. Теперь её можно конкретизировать:

```
Build_Heap2;   (после этого s = n);
пока s ≠ 1 выполнять
  поменять местами A[s] и A[1]
  s := s - 1;
  Sift_Down(1);
конец
```

Заметим, что вторая реализация *Build_Heap* (т.е. на основе *Sift_Down*) предпочтительна здесь по двум причинам. Во-первых, она быстрее. Во-вторых, она позволяет при написании сортировки обойтись вообще без реализации *Sift_Up*.

Для автоматического тестирования программы сортировки предназначены две соответствующие задачи раздела «Задачи на реализацию кучи».

11. Литература

1. Т. Кормен, Ч. Лейзерсон, Р. Ривест. Алгоритмы: построение и анализ. М.: МЦНМО, 2000. § 7. *Примечание: процедура, которая выше названа* `Sift_Down`*, в этой книге называется* `Heapify`.

Задачи на реализацию кучи

Решения этих задач рекомендуется сдавать в тестирующую систему.

Во всех задачах рассматриваются максимальные кучи.

В задачах 1 и 2 входные данные устроены следующим образом.

В первой строке задан размер кучи $N \in [1; 10^5]$. Во второй строке вводится сама куча – N различных целых чисел, каждое из диапазона $[-10^9; 10^9]$. (Гарантируется, что эти числа составляют корректную максимальную кучу).

В третьей строке вводится число M – количество запросов, $M \in [0; 10^5]$.

В следующих M строках вводятся сами запросы – по одному в строке. (Как устроен запрос, указано в условии каждой задачи).

Для простоты в первых трёх задачах будем иметь дело с кучей, где все элементы различны (однако на практике обеспечить выполнение этого условия почти нереально: при добавлении или изменении элемента невозможно за разумное время проверить его уникальность средствами самой кучи).

1. Увеличение приоритета

Запрос задаётся двумя целыми числами i и x. Требуется увеличить значение i-го элемента кучи на x и выполнить *Sift_Up* для восстановления кучи. Гарантируется, что $i \in [1; N]$, $x \geq 0$, новое значение $A[i]+x$ не превышает 10^9 и отличается от текущих значений всех остальных элементов кучи.

Формат входных данных – см. выше.

Формат выходных данных. В качестве ответа на запрос требуется вывести одно число: сообщить, на каком месте массива оказался изменённый элемент после выполнения *Sift_Up*. (Вывести в отдельной строке одно число – соответствующий индекс).

Кроме того, после выполнения всех запросов требуется вывести кучу в её конечном состоянии.

Пример:

Входные данные	Результат	Куча в виде дерева
6 12 6 8 3 4 7 2 5 11 3 6	1 3 15 12 14 3 6 7	Начальная Конечная 12 15 / \ / \ 6 8 12 14 /\ / /\ / 3 4 7 3 6 7

2. Уменьшение приоритета

Запрос задаётся двумя целыми числами i и x. Требуется уменьшить значение i-го элемента кучи на x и выполнить *Sift_Down* для восстановления кучи. Гарантируется, что $i \in [1; N]$, $x \geq 0$, новое значение $A[i]-x$ не превышает по модулю 10^9 и отличается от текущих значений всех остальных элементов кучи.

Формат входных данных – см. выше.

Формат выходных данных. В качестве ответа на запрос требуется вывести одно число: сообщить, на каком месте массива оказался изменённый элемент после выполнения *Sift_Down*. (Вывести в отдельной строке одно число – соответствующий индекс).

Кроме того, после выполнения всех запросов требуется вывести кучу в её конечном состоянии.

Пример:

Входные данные	Результат	Куча в виде дерева
6 12 6 8 3 4 7 2 2 5 1 2	5 1 10 4 8 3 1 7	Начальная Конечная 12 10 / \ / \ 6 8 4 8 /\ / /\ / 3 4 7 3 1 7

3. Извлечение максимального

Дана куча размера $N > 1$. Требуется N-1 раз выполнить извлечение максимального элемента. Как рассказано выше, в процессе выполнения процедуры *Extract_Max* последний элемент кучи помещается в её корень, а затем просеивается вниз вызовом *Sift_Down*. После каждого выполнения процедуры *Extract_Max* нужно будет вывести индекс конечного положения этого элемента после просеивания, а также значение извлечённого максимального элемента.

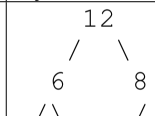
Формат входных данных

В первой строке задан размер кучи $N \in [2; 10^5]$. Во второй строке вводится сама куча – N различных целых чисел, каждое из диапазона $[-10^9; 10^9]$. (Гарантируется, что эти числа составляют корректную максимальную кучу).

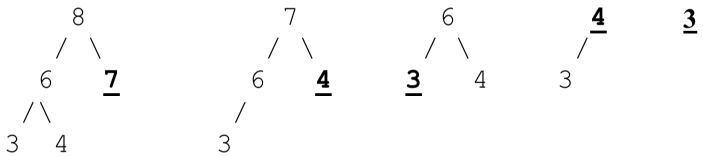
Формат выходных данных

Требуется вывести N-1 строку, в каждой – два числа. Первое – индекс конечного положения элемента после его просеивания; второе – значение извлечённого элемента.

Пример:

Входные данные	Результат	Куча в виде дерева
6 12 6 8 3 4 7	3 12 3 8 2 7 1 6 1 4	

Куча в виде дерева после каждого выполнения *Extract_Max* (выделено положение только что просеянного элемента):



Просеивание и равенство элементов

В следующих задачах в куче могут встречаться равные элементы. Это создаёт некоторый произвол при выполнении процедур *Sift_Up* и *Sift_Down*. Для однозначности ответа его придётся ограничить. Поэтому введём два правила:

1) Процедуры просеивания не должны перемещать элемент дальше, чем это действительно необходимо. Например, если $A[i] = A[2*i]$, то вызов *Sift_Up*(2**i*) не должен менять местами эти два элемента (хотя их обмен и не испортит кучу, он бесполезен).

2) Если при просеивании вниз можно перемещать рассматриваемый элемент как влево вниз, так и вправо вниз (это бывает, когда он меньше двух равных дочерних), то следует выбирать направление влево.

Второе правило довольно произвольно и введено лишь для обеспечения однозначности ответа. Первому правилу, напротив, должна удовлетворять любая разумная реализация кучи.

4. Приоритетная очередь

Требуется реализовать с помощью кучи приоритетную очередь, поддерживающую две операции: добавить элемент и извлечь максимальный элемент.

См. также пункт «Просеивание и равенство элементов».

Формат входных данных

В первой строке вводятся два числа – максимальный размер приоритетной очереди N и количество запросов M. ($1 \le M, N \le 10^5$).

Далее идут M строк, в каждой строке – по одному запросу.

Первое число в запросе задаёт его тип, остальные числа (если есть) – параметры запроса.

Тип 1 – извлечь максимальный (без параметров),

Тип 2 – добавить данный элемент в очередь. Запрос имеет один параметр – число из диапазона $[-10^9; 10^9]$

Формат выходных данных

В ответ на запрос типа 1 следует вывести:

- Если извлекать было нечего (очередь пуста), то -1.
- Иначе, как и в предыдущей задаче – два числа: первое – индекс конечного положения элемента после его просеивания (если же удалён был последний элемент и просеивать осталось нечего, вывести 0); второе – значение извлечённого элемента.

В ответ на запрос типа 2 следует вывести:

- Если добавить нельзя (нет места, поскольку в очереди уже N элементов), то вывести -1. (При этом куча не должна измениться).
- Иначе – индекс добавленного элемента.

Кроме того, после выполнения всех запросов требуется вывести кучу в её конечном состоянии.

Пример:

Входные данные	Результат
4 7 1 2 9 2 4 2 9 2 9 2 7 1	 -1 1 2 3 2 -1 2 9 9 4 9

5. Приоритетная очередь с удалением

Условие этой задачи отличается от условия предыдущей лишь наличием ещё одного типа запроса – запроса на удаление заданного (не обязательно максимального) элемента. Это будет запрос типа 3 с единственным параметром, задающим индекс элемента, который требуется удалить из кучи.

В ответ на запрос типа 3 следует вывести:

- -1, если элемента с таким индексом нет и удаление невозможно. (При этом куча не должна измениться).
- Иначе – значение удаленного элемента.

Гарантируется, что параметр является неотрицательным целым не больше 10^9.

Пример:

Входные данные	Результат
4 10 1 2 9 2 4 2 9 2 9 2 7 1 3 4 2 1 3 3	 -1 1 2 3 2 -1 2 9 -1 4 9 9 4 1

6. Построение кучи просеиванием вверх (*Build_Heap1*)

Дан массив. Требуется преобразовать его в кучу с помощью процедуры просеивания вверх.

Формат входных данных. В первой строке вводится длина массива N. В следующей строке идут элементы массива – N целых чисел, каждое из которых не превышает по модулю 10^9. ($0 \le N \le 10^5$).

Формат выходных данных. N целых чисел – элементы кучи по порядку.

Пример:

Входные данные	Результат
6 1 2 3 4 5 6	6 4 5 1 3 2

7. Построение кучи просеиванием вниз (*Build_Heap2*)

Дан массив. Требуется преобразовать его в кучу с помощью процедуры просеивания вниз. Ввод-вывод устроен так же, как в предыдущей задаче. См. также пункт «Просеивание и равенство элементов».

Пример:

Входные данные	Результат
6 1 2 3 4 5 6	6 5 3 4 2 1

8. Сортировка - подробно

Требуется отсортировать по неубыванию с помощью изученного алгоритма

целочисленный массив размера N, выводя также некоторые промежуточные результаты работы. А именно, должны быть выведены:

- первоначальная куча, построенная вызовом *Build_Heap2*;
- куча после удаления каждого элемента (то есть после каждой итерации внешнего цикла);
- отсортированный массив.

См. также пункт «Просеивание и равенство элементов».

Формат входных данных. В первой строке вводится длина массива N. В следующей строке идут элементы массива – N целых чисел, каждое из которых не превышает по модулю 10^9. ($1 \le N \le 500$).

Формат выходных данных. В первой строке должна быть выведена куча, построенная вызовом *Build_Heap2*, а в каждой из следующих N-1 строк должно быть выведено состояние кучи после удаления очередного элемента. (Таким образом, в i-й строке должно быть выведено N+1-i чисел). В последней (N+1-й) строке нужно вывести отсортированный массив (N чисел).

Пример:

Входные данные	Результат
6 1 2 3 4 5 6	6 5 3 4 2 1 5 4 3 1 2 4 2 3 1 3 2 1 2 1 1 1 2 3 4 5 6

9. Просто сортировка

Требуется отсортировать по неубыванию с помощью изученного алгоритма целочисленный массив размера N.

Формат входных данных. В первой строке вводится длина массива N. В следующей строке идут элементы массива – N целых чисел, каждое из которых не превышает по модулю 10^9. ($1 \le N \le 10^5$).

Формат выходных данных. N чисел – элементы исходного массива в порядке неубывания.

Пример:

Входные данные	Результат:
6 10 4 2 2 1 6	1 2 2 4 6 10

Упражнения. Подсказки

Упражнение 7. Это упражнение может оказаться труднее предыдущего потому, что процедуру *Sift_Down* придётся вызывать в других условиях, чем мы делали это до сих пор. Заметим, что разумная реализация *Sift_Down* удовлетворяет не только спецификации, которая была сформулирована выше (см. п. 7), но также и следующей спецификации:

Дано: Вершина i, такая, что основное свойство кучи выполнено во всех вершинах поддерева с корнем i, кроме, возможно, самой вершины i.

Надо: Основное свойство кучи выполнено во всех вершинах этого поддерева.

(Эта спецификация отличается от предыдущей тем, что о вершинах вне данного поддерева ничего не известно).

Упражнения. Ответы и решения

Упражнение 1. Если высота дерева равна h, то число вершин s может принимать значение от 2^{h-1} до 2^h-1 включительно.

Упражнение 2. Дерево, в котором s вершин, имеет высоту $h = [\log_2 s]+1$.

Упражнение 3. *Sift_Up*(i). Проще всего алгоритм описать так. Если $A[i]$ является корнем дерева или не превосходит своего родителя, то ничего делать не надо. В противном случае надо поменять местами его с родителем, после чего снова получим «немного испорченную кучу», но теперь элемент, который, возможно, имеет большее значение, чем требуется на его месте, находится по индексу $[i/2]$. Соответственно надо выполнить *Sift_Up* для этого элемента.

Однако при реализации полезно сделать два усовершенствования. Во-первых, нетрудно избавиться от рекурсии. Во-вторых, нет необходимости на каждом шаге просеивания действительно копировать просеиваемый элемент на место его родителя – его можно записать лишь в самом конце просеивания на его окончательное место.

Кроме того, можно немного упростить программу и уменьшить число проверок, если поместить значение $+\infty$ в $A[0]$ в качестве барьерного элемента.

Полезно заметить сходство этого алгоритма с вставкой очередного элемента в подходящее место отсортированной части массива, которая происходит при простой сортировке вставками.

Для оценки времени работы заметим, что число шагов просеивания не превышает высоты кучи, а время выполнения каждого шага ограничено константой.

Упражнение 4. *Sift_Down*(i). Проще всего алгоритм описать так. Если у $A[i]$ нет сыновей в дереве или если они не превосходят его по величине, то ничего делать не надо. В противном случае надо поменять местами его с максимальным из сыновей. (Если эти сыновья равны, то, вообще говоря, менять можно с любым из них. Но чтобы все последующие задачи были приняты

тестирующей системой, надо менять непременно с левым). После чего снова получим «немого испорченную кучу», но теперь элемент, который, возможно, имеет меньше значение, чем требуется, находится на новом месте. Соответственно надо выполнить *Sift_Down* для этого элемента.

При реализации можно избавиться от рекурсии. Оценка времени работы такая же, как для *Sift_Up*. Типичная ошибка – неправильная обработка ситуации, когда имеется только левый сын (то есть когда n чётно, а $i=n/2$).

Упражнение 5. По аналогии с *Extract_Max* извлечение элемента с индексом i можно реализовать следующим образом:

```
ответ := A[i]
A[i] := A[s];
s := s - 1;
Восстановить кучу
```

Типичная ошибка – предположить, что последний элемент $A[s]$ не превосходил удаляемого $A[i]$ и пытаться выполнить восстановление кучи просто вызовом *Sift_Down*.

Упражнение 6. Можно считать, что с самого начала имеется «немного испорченная куча» размером $s = 2$, в которой последний элемент, возможно, имеет большее значение, чем требуется для занимаемого им места. Исправив её вызовом *Sift_Up*(2), получим «немного испорченную кучу» размером $s = 3$, которая, в свою очередь, тоже может быть исправлена вызовом *Sift_Up*(3). И так далее. Заметим, что это, по существу, ничем не отличается от последовательного добавления элементов A[2], A[3], … A[n] по одному в приоритетную очередь.

Упражнение 7. См. раздел «Подсказки» выше.

Основная идея состоит в следующем. В соответствии с новой спецификацией *Sift_Down* мы можем сразу вызвать её для любой вершины, чьи сыновья являются листьями. Вызвав её для всех таких вершин, получим поддеревья высоты 2, в каждом из которых выполняется основное свойство кучи. И так далее – новые вызовы *Sift_Down* будут обеспечивать выполнение основного свойства кучи в поддеревьях всё большего и большего размера.

Реализовать это проще всего с помощью цикла for, который вызывает *Sift_Down* для всех элементов массива, кроме листьев дерева, справа налево.

Упражнение 8. В первой реализации (на основе *Sift_Up*) выполняется много просеиваний на большую глубину, и мало просеиваний на маленькую глубину. Во второй реализации, наоборот, просеиваний на большую глубину выполняется мало, а на маленькую – много. Поэтому разумно предположить (и можно доказать), что вторая реализация работает быстрее.

Упражнение 9*. Можно доказать (см. например, книгу Кормена и др. из списка литературы), что вторая реализация (на основе *Sift_Down*) всегда работает за время $\Theta(n)$, а первая в худшем случае – за $\Theta(n \log n)$.