Лекция 3

Представление данных (II).

Массив (сортировка, поиск k-го элемента).

3.1 Массив.

Абстрактное понятие массива. Линейный (полный) порядок. Сортировка массива.

ПРОБЕЛ В КОНСПЕКТЕ.

3.1.1 HeapSort

Алгоритм HeapSort получил свое название от английского слова *heap* — куча. Неформально говоря, в этом алгоритме данные из массива организуются в виде «*кучи*»: двоичного дерева, в каждой вершине которого хранится элемент, не превосходящий элемента, хранящегося в родителе этой вершины.

Как нетрудно видеть, в таком представлении легко найти наибольший элемент массива: он находится в корне дерева. Удалив его из дерева и восстановив структуру кучи, мы сможем так же легко найти следующий по убыванию элемент, и т. д.

Таким образом, для достижения цели нам достаточно научиться строить кучу и восстанавливать правильность ее структуры после удаления ее корня. И то, и другое мы будем делать при помощи рекурсивной операции «*утапливания*» вершины: если в некоторой вершине хранится элемент, строго меньший элемента, хранящегося в одном из сыновей этой вершины, то этот элемент надо поменять местами с его сыном (с тем из двух сыновей, в котором ключ наибольший), а затем, если необходимо, продолжить его «утапливание».

Для того, чтобы описать этот алгоритм более строго, зафиксируем способ представления нашей кучи. Будем ее поддерживать в том же самом массиве, который нам дан. Укладка дерева в массив a производится следующим образом. Занумеруем дерево по уровням: корень — это $a[1]$, вершины следующего уровня — это $a[2]$ и $a[3]$, и т. д. При такой нумерации массив a содержит все элементы этого дерева, причем сыновья вершины $a[i]$ расположены в $a[2i]$ и $a[2i+1]$. Нам достаточно такого представления дерева, поскольку нам нужны лишь две операции: чтение конкретного элемента $a[i]$ и перестановка *содержимого* двух его вершин: $\mathrm{swap}(a[i], a[j])$.

«Утопим» вершину:

```
procedure pushnode ( i : integer );
begin
    if i — лист then return;

    выбрать из потомков i наибольший (назовем j, это 2i или 2i+1);
    if a[j] > a[i]                              (* что неправильно! *)
    then begin swap(a[i], a[j]); pushnode(j) end
end;
```

Замечание 3.1. Количество вершин в нашем дереве будет сокращаться. Процедура pushnode должна отслеживать это; в частности, правильно определять, сколько в текущий момент времени потомков у i (ноль, один или два). Заметим, что мы используем массив a, количество элементов в нем и текущее количество элементов в дереве как глобальные переменные. □

Построим правильную кучу (пусть всего в ней n вершин):

```
procedure pushall;
begin
    for i:=n downto 1 do pushnode(i)
end;
```

Лемма 3.1. *В дереве, построенном процедурой* pushall*, никакой потомок не превосходит родителя.*

Доказательство. Индукция по *убыванию* номера вершины (от n до 1). Иначе говоря, по построению дерева (добавлению корня к двум поддеревьям). На первом же шаге новая вершина становится больше всех своих потомков, на втором — единственная вершина, в которой что-то могло испортиться, также становится больше всех своих потомков, и т. д. □

Лемма 3.2. *Процедура* pushall *(вместе с вызовами процедуры* pushnode*) использует $O(n \log n)$ операций обмена (*swap*).*

Доказательство. Для каждой из n вершин вызывается процедура pushnode. Она делает не более $\log n$ рекурсивных вызовов (поскольку такова высота дерева), в каждом из них происходит лишь константное число обращений к элементам массива. □

Упражнение 3.1. Показать, что на самом деле используется лишь $O(n)$ операций (хотя для дальнейших рассуждений нам это не будет важно). □

Наконец, отсортируем массив:

```
procedure heapsort;
begin
    pushall;

    for i:=n downto 1 do
    begin
        swap(a[1], a[i]); (*a[1] — наибольший из оставшихся — в конец!*)
        pushnode(1);                        (*ведь a[1] «испортился»*)
    end
end;
```

Теорема 3.1. *Процедура* heapsort *правильно сортирует массив и затрачивает на это лишь $O(n \log n)$ операций с элементами массива.*

Доказательство. **Время работы** складывается из времени работы pushall (см. лемму 3.2) и времени работы процедуры pushnode (в доказательстве леммы 3.2 мы уже видели, что это $O(\log n)$ операций), вызванной n раз.

Корректность *построения* кучи доказана в лемме 3.1. То, что на каждом шаге после отправки $a[1]$ в конец куча восстанавливается правильно, можно доказать аналогично индуктивному шагу в доказательстве леммы 3.1. Наконец, благодаря основному свойству кучи, на каждом шаге мы действительно «вытаскиваем» из нее (отправляем в конец массива) наибольший из оставшихся элементов.

□

Замечание 3.2. Теорема 3.1 справедлива для *любого* массива a (с *любыми* значениями). Таким образом, мы оценили время работы алгоритма *в наихудшем случае.* □

Упражнение 3.2. Точное время работы зависит от того, какие элементы мы сортируем. Какое время займет сортировка массива целых чисел на RAM-машине при помощи алгоритма heapsort? □

3.1.2 QuickSort

Алгоритм QuickSort: возьмем какой-нибудь (скажем, первый в массиве) элемент, поставим его на нужное место i (так что все меньшие его элементы находятся слева, все бóльшие — справа) и рекурсивно отсортируем полученные массивы, состоящие из $i-1$ и $n-i$ элементов соответственно.

Теорема 3.2. *В алгоритме QuickSort количество операций над элементами массива в наихудшем случае составляет $O(n^2)$.*

Упражнение 3.3. Доказать теорему 3.2. □

Замечание 3.3. Говорят, что $f = \Omega(g)$, если $g = O(f)$. Однако, есть и другое определение (часто используемое в теории сложности): $f = \Omega(g)$, если $f \neq o(g)$, т.е. $f(n)$ бесконечно часто бывает больше $cg(n)$ для некоторой константы $c > 0$.

Теорема 3.3. *В алгоритме QuickSort количество операций над элементами массива в наихудшем случае составляет $\Omega(n^2)$.*

Доказательство. Рассмотрим поведение алгоритма на уже отсортированном массиве. □

Теорема 3.4. *В алгоритме QuickSort количество операций над элементами массива* в среднем *составляет $O(n \log n)$.*

Доказательство. Пусть $t(\alpha)$ обозначает количество операций, затрачиваемое на массив, исходное упорядочение которого задано перестановкой α (как легко заметить, количество операций зависит только от этой перестановки, а не от конкретных элементов массива: $(9, 5, 7)$ и $(3, 1, 2)$ сортируются за одно и то же время). Количество операций, затрачиваемых в среднем на массивы размера n, обозначим через $T(n)$. Размер массива (или соответствующей перестановки) α обозначим через $|\alpha|$. Часть массива (или перестановки) α с индексами от j до k обозначим через $\alpha[j:k]$.

$$
\begin{aligned}
T(n) &= \frac{1}{n!}\sum_{|\alpha|=n} t(\alpha) = \\
&= \frac{1}{n!}\sum_{i=1}^{n}\sum_{|\alpha|=n,\,\alpha[1]=i} t(\alpha) \leq \\
&\leq \frac{1}{n!}\sum_{i=1}^{n}\sum_{|\alpha|=n,\,\alpha[1]=i} (cn + t(\alpha[1:i-1]) + t(\alpha[i+1:n])) = \\
&= cn + \frac{1}{n!}\sum_{i=1}^{n}\left(i(i{+}1)\dots(n{-}1)\sum_{|\beta|=i-1} t(\beta) + \right. \\
&\qquad \left. +(n{-}i{+}1)(n{-}i{+}2)\dots(n{-}1)\sum_{|\gamma|=n-i} t(\gamma)\right) = \\
&= cn + \frac{1}{n}\sum_{i=1}^{n}\left(\frac{1}{(i-1)!}\sum_{|\beta|=i-1} t(\beta) + \frac{1}{(n-i)!}\sum_{|\gamma|=n-i} t(\gamma)\right) = \\
&= cn + \frac{1}{n}\sum_{i=1}^{n}(T(i-1) + T(n-i)) = \\
&= cn + \frac{2}{n}\sum_{i=0}^{n-1} T(i). \qquad (3.1)
\end{aligned}
$$

Остается показать, что решение этого рекуррентного неравенства удовлетворяет условию $T(n) = O(n\log n)$. Предварительно убедимся, что

$$\sum_{i=2}^{n-1} i\ln i \leq \int_2^n x\ln x \, \mathrm{d}x \leq \frac{n^2\ln n}{2} - \frac{n^2}{4}$$

(в этом можно убедиться при помощи интеграла — по выпуклости функции $x\ln x$; или, вместо интеграла, по индукции).

Пусть $b = \max\{T(0), T(1)\}$, $k = 2b + 2c$. Покажем по индукции, что для всех $n \geq 2$ выполняется $T(n) \leq kn\ln n$. База ($n = 2$) очевидна из

(3.1). Для $n \geq 3$ имеем

$$T(n) \leq cn + \frac{2}{n}\sum_{i=0}^{n-1} T(i) \leq cn + \frac{4b}{n} + \frac{2}{n}\sum_{i=2}^{n-1} T(i) \leq$$
$$\leq cn + \frac{4b}{n} + \frac{2}{n}\sum_{i=2}^{n-1} ki\ln i \leq cn + \frac{4b}{n} + \frac{2k}{n}\left(\frac{n^2\ln n}{2} - \frac{n^2}{4}\right)$$
$$= kn\ln n + cn + \frac{4b}{n} - (b+c)n \leq kn\ln n.$$

□

3.1.3 Randomized QuickSort

Алгоритм Randomized QuickSort отличается от QuickSort тем, что на каждом шаге элемент выбирается случайным образом. Оценим время его работы *в наихудшем случае.* Оно будет зависеть от того, какие нам достанутся случайные числа.

Теорема 3.5. *Для* любого *входного массива с вероятностью не менее* $1/2$ *в алгоритме Randomized QuickSort количество операций над элементами массива составляет* $O(n \log n)$.

Упражнение 3.4. Доказать теорему 3.5. □

Замечание 3.4. В исходном алгоритме QuickSort для некоторых массивов соответствующая вероятность равна нулю. □

3.1.4 Поиск k-го элемента

Аналогично QuickSort — все так же, как и в QuickSort, но рекурсивный вызов делаем только для той половины массива, в которой содержится интересующий нас элемент.

Упражнение 3.5. Оценить время работы этого алгоритма в наихудшем случае и в среднем.

Упражнение 3.6. Оценить время работы алгоритма, работающего наподобие *Randomized* QuickSort.

За линейное время в наихудшем случае. Разобьем массив на пятерки; возьмем медианы (третьи элементы) полных пятерок и вычислим их медиану. Полученным элементом и разобьем массив «пополам» (как в предыдущем алгоритме). Время работы (количество операций над элементами массива) в наихудшем случае составляет время на поиск медианы + время на поиск искомого элемента в одной из полученных «половинок»: $T(n) \le T(\frac{n}{5}) + T(n - \lfloor\frac{3n}{10}\rfloor) + cn$. При $n \ge 50$ это выражение $\le T(\frac{n}{5}) + T(\frac{3n}{4}) + cn$. Нетрудно по индукции доказать, что $T(n) \le cn$, где c выбрана так, чтобы $T(n) \le cn$ при $n \le 49$. Мы доказали следующую теорему.

Теорема 3.6. *Приведенный алгоритм затрачивает в наихудшем случае лишь $O(n)$ операций на поиск k-го элемента в массиве из n элементов.*