Лекция 4

Представление данных (III).

Файл (сортировка на четырех лентах). Списки. Хеш-таблицы. Деревья.

4.1 Файл

Файл последовательного доступа — это структура данных, к которой применимы следующие элементарные операции:

- READNEXT — считать следующий элемент;
- WRITENEXT — записать следующий элемент;
- REWIND — вернуться к первому элементу (т.е. следующая операция READNEXT или WRITENEXT будет обращаться к первому элементу).

Прямого доступа к i-му элементу нет (точнее, он не является элементарной операцией и занимает не константное время).

Размер дисковой памяти (а тем более — магнитных лент) может превышать размер оперативной памяти. Поэтому при сортировке файла может случиться так, что мы не сможем полностью считать файл в оперативную память (и отсортировать полученный массив). Сейчас мы предъявим алгоритм сортировки, который будет использовать лишь *константное* число ячеек оперативной памяти (правда, будет пользоваться четырьмя файлами, а не одним). Файлы будем называть *лентами.*

Сортировка на четырех лентах. Разобьем исходный файл пополам на две ленты (последовательно считывая элементы, будем нечетные записывать на первую ленту, а четные — на вторую). Далее будем из двух лент, состоящих из отсортированных блоков по i элементов, составлять две ленты, состоящие из отсортированных блоков по $2i$ элементов (*распространенный прием*: для простоты будем считать, что количество элементов является степенью двойки 2^k, — в противном случае время работы вырастет заведомо не более, чем в константу раз, поскольку размер входа вырастет не более, чем в константу раз, даже если его округлить до степени двойки в большую сторону).

Делается это так: читаем поэлементно блоки с обеих лент (назовем эти ленты A и B), пишем блок удвоенной длины на одну ленту (назовем ее C) (а следующий — на другую, назовем ее D); при этом каждый раз на ленту C мы пишем наименьший элемент v из двух считанных (с ленты A и с ленты B) и читаем следующий элемент с той ленты, с которой взяли v (если текущий блок на ней еще не закончился).

Лемма 4.1. *Если ленты A, B длины 2^{k-1} состояли из блоков, отсортированных по i элементов, то после этой операции ленты C, D будут состоять из блоков, отсортированных по $2i$ элементов (и по-прежнему будут иметь длину 2^{k-1}). Эта процедура займет $O(n)$ операций считывания/записи элементов и $O(1)$ ячеек оперативной памяти.*

После этого ленты (A,B) и (C,D) меняются местами (читаем C и D, пишем на A и B). Очевидно, за t итераций ленты, отсортированные по 1 элементу, превратятся в ленты, отсортированные по 2^t элементов. Таким образом, мы доказали следующую теорему.

Теорема 4.1. *Приведенный алгоритм сортирует исходный файл за $O(n \log n)$ обращений к файлам.*

4.2 Списки

Однонаправленные списки. Однонаправленный список состоит из элементов[1], каждый из которых содержит полезные данные и указатель на следующий элемент (пустой указатель, если следующего элемента нет). На RAM-машине список чисел можно хранить в виде пар регистров (число; номер регистра, содержащего следующий элемент списка).

Очевидно, над однонаправленными списками легко (за константное число операций над элементами) реализуются следующие операции:

- NEXT — вернуть указатель на следующий элемент списка;
- INSERT_AFTER — вставить новый элемент после заданного;
- DELETE_AFTER — удалить элемент, следующий за заданным;
- DELETE_ROOT — удалить первый элемент списка;
- ROOT — вернуть указатель на первый элемент списка.

[1] На Паскале — записей.

Поиск элемента, занимающего позицию i, занимает линейное время. То же верно для нахождения элемента, предшествующего в списке заданному.

Упражнение 4.1. Списки удобно сортировать сортировкой на четырех лентах. Заметим, что для этого не потребуется дополнительной памяти (даже файла). □

Двунаправленные списки. Отличаются от однонаправленных тем, что каждый элемент списка содержит также и указатель на *предыдущий* элемент. Отдельно хранится указатель на последний элемент списка. Благодаря этому легко реализуются следующие дополнительные операции:

- LAST — вернуть указатель на последний элемент;
- PREVIOUS — вернуть указатель на предыдущий элемент;
- INSERT_BEFORE — вставить элемент перед заданным;
- DELETE — удалить элемент (заменяет операции DELETE_AFTER и DELETE_ROOT над однонаправленными списками).

Skip-lists. Снабдим список дополнительной структурой, облегчающей поиск элемента по ключу из линейно упорядоченного множества. Над каждым элементом надстроим список из нескольких (для разных элементов — из разного количества) элементов, объединив надстроенные элементы по этажам. Теперь можно сначала искать элемент на верхнем (самом маленьком) этаже, затем спуститься в нужное место следующего этажа, и т. д.

Замечание 4.1. В нескольких экземплярах надо хранить только ключи; остальные данные достаточно хранить только на нижнем уровне.

Один из способов построить эффективный в среднем skip-list — строить его случайным образом, т.е. надстраивать над данным элементом следующий этаж с вероятностью 1/2 (аналогично — добавлять новые элементы). Точные формулировки и доказательства опустим, т.к. они потребовали бы знания теории вероятностей.

4.3 Хеш-таблицы

Представим, что нам нужно хранить словарь. Проиндексировать строки строками (т.е. составить массив, к которому можно было бы обращаться A[строка]) невозможно — их слишком много. Проиндексировать строки последовательными числами — неудобно (будет трудно найти заданное слово — его придется искать во всем словаре; к тому же, очень неудобно вставлять новое слово).

Хеш-функция f — это функция, отображающая множество объектов в множество ключей. *Хеш-таблица* — это массив, проиндексированный возможными значениями хеш-функции. В каждой ячейке этого массива хранится список объектов с соответствующим значением хеш-функции.

Важно выбрать хеш-функцию так, чтобы в разных списках было примерно одинаковое количество элементов.

Имеется очевидный компромисс между временем работы (которое зависит от длин списков) и занимаемой памятью (которая зависит от мощности образа хеш-функции).

4.4 Деревья

Будем говорить о деревьях с корнем.

4.4.1 Представление деревьев в компьютере

Имеется много разновидностей структур данных, называемых деревьями. Соответственно, и набор операций над деревьями может быть разным. Поэтому *представление зависит от требуемых операций.* (Например, мы уже видели нетрадиционное представление двоичного дерева в виде массива в ситуации, когда перестановки поддеревьев и даже вставка/удаление не нужны.)

Как представить двоичное дерево, ясно: элемент, соответствующий каждой вершине, содержит хранимые данные и два указателя: на левого и правого сына; если нужно — указатель на родителя. Для дерева с произвольной степенью вершин имеется несколько вариантов: например, список сыновей (полезно указать их количество, а указатель наверх хранить в каждом из них).

Замечание 4.2. Динамический массив вместо списка не подойдет — трудно вставлять!

4.4.2 Деревья поиска

Дерево поиска — структура данных, для которой эффективно реализуемы операции INSERT, DELETE и FIND (поиск по ключу). («Эффективно» в данном случае означает «не больше, чем за $O(\log \text{количества элементов})$ операций с элементами». Даже если исходно данные не представлены в виде дерева, их может быть полезно представить в таком виде, если они упорядочены и над ними часто приходится выполнять указанные операции.

Свойство

$$\text{«все элементы левого поддерева меньше ключа,} \atop \text{ключ больше всех элементов правого поддерева»} \tag{4.1}$$

позволяет реализовать FIND за O(высоты дерева) операций. Если дерево «идеально» (высота h, количество вершин $2^{h+1}-1$), имеем $O(\log n)$ операций. Ниже мы изучим разновидности деревьев поиска, позволяющие поддерживать себя в «почти идеальном» виде, при этом ограничиваясь логарифмическим количеством операций с элементами при реализации операций INSERT и DELETE.

4.4.3 АВЛ-деревья[2]

Определение 4.1. *АВЛ-дерево (сбалансированное дерево)*: двоичное дерево поиска (удовлетворяющее свойству (4.1)), для любой вершины которого высоты левого и правого поддеревьев отличаются не более, чем на единицу.

Лемма 4.2. *Высота АВЛ-дерева составляет $O(\log n)$.*

Доказательство. Покажем по индукции, что в АВЛ-дереве высоты h имеется не менее

$$\frac{5+2\sqrt{5}}{5}\left(\frac{1+\sqrt{5}}{2}\right)^h + \frac{5-2\sqrt{5}}{5}\left(\frac{1-\sqrt{5}}{2}\right)^h - 1 \qquad (= \Omega(\phi^h))$$

вершин[3]. Очевидно, в дереве высоты h имеется не менее $G_{h-1}+G_{h-2}+1$ вершин, где G_i — наименьшее количество вершин в АВЛ-дереве высоты i. По предположению индукции доказательство завершается (NB: $\left(\frac{1\pm\sqrt{5}}{2}\right)^2 = \left(\frac{3\pm\sqrt{5}}{2}\right)^2$). □

Следовательно, поиск элемента отнимает лишь $O(\log n)$ операций. Покажем, что то же самое относится и к операциям INSERT и DELETE. Для поддержания сбалансированности нам понадобится выяснять высоты поддеревьев. Для этого будем хранить в каждой из вершин разность

[2]Сокращение произошло от фамилий авторов этой конструкции (Адельсон-Вельский и Ландис).

[3]Имеется в виду обозначение Ω, принятое в (непрерывном) математическом анализе.

высот левого и правого поддеревьев (очевидно, высота любого поддерева тогда вычисляется за $O(\log n)$ операций — но в большинстве случаев это даже не нужно).

INSERT. Попробуем вставить вершину с ключом i. Найдем место, где она должна находиться, и вставим ее туда. Если это второй потомок какой-то вершины, то высота не изменилась, и все OK. В противном случае, посмотрим на высоты всех (пра)родителей вершины i. По ним мы сможем найти самую нижнюю из разбалансированных вершин (назовем ее v); пусть w — первая вершина на пути из v в i. Если i была вставлена во «внешнее» поддерево вершины w, мы можем совершить следующее «вращение»: w становится корнем, а v — ее сыном. «Внутреннее» поддерево вершины w становится «внутренним» поддеревом вершины v.

```
        v                              w
       / \                            / \
      A   w              --->        v   I
         / \                        / \
        B   I (в нем i)            A   B
```

Если же i была вставлена во «внутреннее» поддерево вершины w, то сначала надо произвести другое «вращение» поддерева с корнем w, чтобы попасть в только что рассмотренную ситуацию:

```
        v                              v
       / \                            / \
      A   w              --->        A   x
         / \                            / \
        x   C                          I   w
       / \                                / \
      I   I'                             I'  C
```

Теперь правое поддерево вершины x (играющей роль w) заведомо выше левого, так что вершина, разбалансирующая v, расположена именно в правом поддереве.

Замечание 4.3. Заметим, что все эти операции затрагивают только поддерево с корнем v! □

DELETE. Аналогично. Однако, заметим, что после «вращений» высота всего дерева с корнем v может уменьшиться. При добавлении это не мешало, так как получалась в точности высота этого дерева до добавления, а остальные вершины уже были сбалансированы при этом условии. При удалении же может разбалансироваться другая вершина (выше) и операции надо будет повторить (и т. д. — вплоть до $O(\log n)$ раз).

4.4.4 2-3-4-деревья

В 2-3-4-дереве внутренняя вершина может иметь от 2 до 4 сыновей и от 1 до 3 ключей соответственно. В листьях может храниться от 2 до 4 ключей. Все листья 2-3-4-дерева находятся на одной и той же глубине.

Ключи, хранящиеся в вершине, разделяют (в смысле операции сравнения) ключи, лежащие в соответствующих поддеревьях:

```
   ________________
  |_10_|_20_|__30__|
  /     |      \       \
 /\     /\      /\      /\
/  \   /  \    /  \    /  \
1..9 11..19 21..29 31..40
```

Таким образом, поиск занимает $O(\log n)$ операций.

INSERT. Чтобы вставить ключ в 2-3-4-дерево, найдем лист, в котором он должен был бы находиться. Вставим его туда. Если вершина переполнилась (5 ключей), разобьем ее на две, а средний (третий) ключ используем в качестве нового разделяющего ключа в родительской вершине. Если и она переполнилась (4 ключа, 5 поддеревьев), разделим и ее пополам (1 ключ/2 поддерева и 2 ключа/3 поддерева, разделяющий их ключ отправляется в родительскую вершину). Так будем продолжать, пока вершины не перестанут переполняться. Если дойдем до корня, разделим его пополам, увеличив высоту дерева.

DELETE.

Упражнение 4.2. Аналогично (но вершины не разделяются, а сливаются). При этом может понадобиться заимствовать ключи из соседних вершин, в том числе, при помощи «вращений». □

Следующая теорема теперь очевидна.

Теорема 4.2. *Операции INSERT, DELETE и FIND над 2-3-4-деревьями можно реализовать за $O(\log n)$ операций над их элементами.*

4.4.5 B-деревья

Это обобщение 2-3-4-деревьев. Мотивировка: хранение базы данных на диске; заодно с нужными данными с диска автоматически (так работают диски) считывается сразу много других (целый *блок* — физическая единица информации на диске); хорошо бы, чтобы они в дальнейшем тоже были небесполезны.

Степень вершин теперь между $t/2$ и t (кроме корня: его степень ≥ 2) — так, чтобы запись всей вершины в точности поместилась в блок. Соответственно, в каждой вершине хранится больше ключей.

Для четных t операции аналогичны.

Упражнение 4.3. Реализовать операции для нечетных t. □

4.4.6 B^+-деревья

Можно не хранить данные во внутренних вершинах, а повторять там ключи из нижних уровней дерева. Тогда, объединив листья в двунаправленный список, мы сможем быстро находить предыдущий и следующий (в смысле упорядочения) элемент.

Упражнение 4.4. Реализовать операции INSERT и DELETE. □