Основные алгоритмы, комментарии по седьмому листку (Структуры данных II)

Шибаев Иннокентий

March 21, 2021

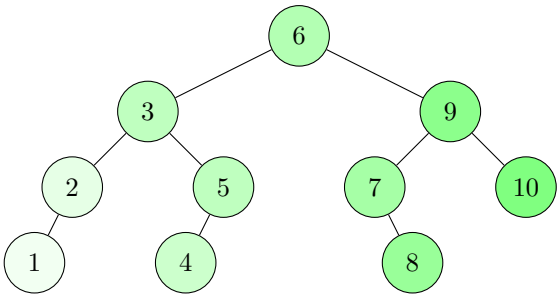
1 По поводу теории

1.1 О сбалансированности деревьев поиска

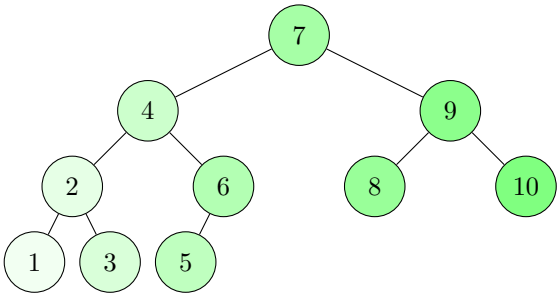
На прошлом семинаре мы рассматривали двоичные деревья поиска (BST). Благодаря тому что они упорядочены по ключам (ключи в левом поддереве меньше либо равны ключу в вершине-родителе, а этот ключ в свою очередь меньше либо равен ключей в правом поддереве) в этой структуре данных можно осуществлять поиск элемента по ключу.

Но если мы рассматриваем произвольное BST то в худшем случае такой поиск может занимать $O(n)$ операций (т.к. дерево может выродиться в граф-путь). Однако если поддеревья примерно равны по размеру, то такой поиск займет уже $O(\log n)$. Это свойство называется сбалансированностью, и может формулироваться немного по-разному:

- *Сбалансированные по числу вершин деревья* – деревья для которых у любой вершины число вершин в левом и правом поддереве отличается не более чем на 1, к примеру



- *Сбалансированные по высоте деревья* – деревья для которых у любой вершины высота левого и правого поддерева отличается не более чем на 1, к прмеру



Это условие более слабое чем предыдущее (можно показать по индукции используя тот факт что это условие эквивалентно тому что в таком дереве заполнены все слои кроме последнего).

Второе условие гарантирует, что высота дерева не превосходит $\log 2n$, и, таким образом, операции поиска по ключу будут выполнятся за $O(\log n)$.

Понятно, что мы как раз хотим чтобы деревья были сбалансированы, чтобы все операции выполнялись как можно быстрее. Поэтому и существует различные самобалансирующиеся деревья, к примеру АВЛ-деревья (сбалансированы по высоте) и красно-черные деревья. Обсудим последние.

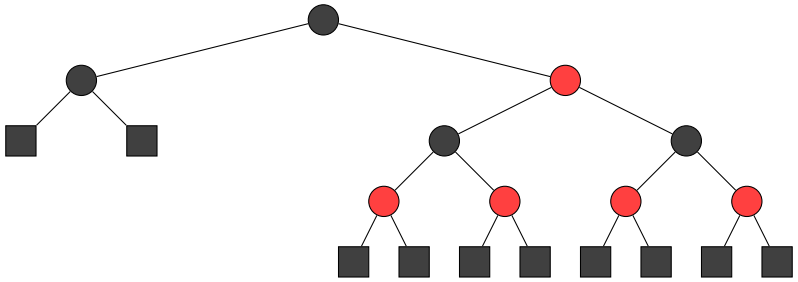
1.2 Красно-черное дерево

Это структура данных на основе бинарного дерева, для которой выполняются следующие 5 свойств:

- Все вершины покрашены в красный либо черный цвет;
- Корень покрашен в черный цвет;
- Все листья, не содержащие данных (NIL-вершины) — чёрные;
- Обе дочерних вершины красного узла — чёрные;
- У всех листовых вершин одинаковая черная высота h_b (b-height) – одинаковое число черных вершин на пути из корня в лист.

Оказывается, что выполнение вот этих пяти свойств (из которых существенными являются последние 2) влечет за собой следующее свойство – высота этого дерева не превышает $2\log n$.

Важно отметить, что это не означает ту же сбалансированность что мы обсуждали выше, к примеру посмотрите на следующий пример



где слева и справа $h_b = 2$ (не считая NIL), но при этом высота правого поддерева вдвое больше высоты левого (причем поддеревья полные!). Т.е. хоть высота дерева и ограничена (доказывалась на лекции, и следует из последних двух условий), но про такую же сбалансированность как раньше мы говорим не можем.

С другой стороны – нам это и не нужно! С точки зрения сложности что с высотой $\log 2n$, что с высотой $2\log n$ операции поиска будут иметь сложность $O(\log n)$.

Сохранение этих свойств достигается за счет всяческих операций поворотов и переподвешиваний, которые мы здесь обсуждать не будем. Что важно отметить, так это то что мы собственно сохранение какой-то разумной высоты подменяем сохранением каких-то странных свойств (перечисленных выше), а уже из них следует сохранение высоты не слишком большой.

В АВЛ-деревьях в вершинах пишется (дополнительно) высота дерева с корнем в этой вершине, и далее при обновлении дерева эти значения обновляются и, при необходимости, происходят повороты которые исправляют коллизии (т.е. случаи когда в левом и правом поддереве высоты отличаются больше чем на 1).

1.3 Амортизационный анализ

То что дальше – краткая компиляция с Википедии и neerc.ifmo.

Рассмотрим некоторый алгоритм, который совершает последовательность из N действий. Некоторые из этих действие простые (занимают мало времени), другие сложные (вызовы каких-то подпрограмм, сложность которых большая). Самое простое что мы можем сделать в оценке сложности – это оценить ее как

$$\sum_{i=1}^{n} c_i \leqslant N \max_{i \in \{1,\dots,N\}} c_i,$$

где сумма c_i – это сумма сложностей отдельных шагов алгоритма.

Но часто бывает так что сложные шаги совершаются намного реже простых, и тогда такая оценка становится очень грубой.

Идея *амортизационного анализа* примерно в следующем – если мы сможем показать что редкие сложные операции как-то равномерно перемешаны с многочисленными простыми, то мы можем как бы *сгладить* их сложность, перераспределить ее при учете и, тем самым, облегчить получение более точной оценки. Т.е. вместо того чтобы пытаться понять в какие именно моменты i алгоритм будет совершать операции с большими c_i мы хотим перейти к стоимостям

$$\tilde{c}_i \approx \frac{\sum_{i=1}^{N} c_i}{N}$$

используя которые уже оценивать исходную сумму (или даже остановится на этом – лчень часто нас интересует то сколько времени будет занимать какая-то одна операция, к примеру при работе со структурами данных). Теперь вопрос – а как получать такую *амортизационную* стоимость. Есть несколько способов:

0) *Метод усреднения* заключается в прямом вычислении суммы выше (поэтому я и записываю его под номером 0), т.е. получении $\tilde{c}_i$ как

$$\tilde{c}_i = \frac{\sum_{i=1}^{N} c_i}{N}$$

и тогда можно оценивать всю сложность как $N\tilde{c}_i$. Это (очевидно) имеет смысл применять когда нас интересует именно то сколько времени тратится на одну операцию в среднем. К примеру в случае динамических массивов они увеличивают длину в два раза когда достигают предела – и в процессе приходится копировать очень много данных. Таким образом иногда операции занимают больше времени чем нужно. К примеру, считая что мы начинаем с массива размера один, имеем такую последовательность

Число элементов в массиве

0	1	2	3	4	5	6	7	8	9	10	11	12	
1	1	2	4	4	8	8	8	8	16	16	16	16	← Размер массива
1	2	3	1	5	1	1	1	9	1	1	1	1	

Сложность добавления нового элемента в массив

что приводит нас к сумме вида

$$\tilde{c}_i = \frac{\sum_{i=1}^{N} 1 + \sum_{i=1}^{\lfloor \log_2 N \rfloor} 2^i}{N} \leqslant \frac{N + 2N}{N} = 3$$

и мы получаем что амортизационная стоимость добавления элемента в такой динамический массив есть $O(1)$.

1) *Метод предоплаты* по сути своей представляет собой перенос сложности из редких сложных операций в частые простые. Мы увеличиваем сложность простых операций, откладывая "лишнее" время (в виде монет) в банк для будущих сложных операций – а когда мы приходим к ним важно лишь чтобы в банке было достаточно монет чтобы оплатить эту сложную операцию. На том же примере динамического массива мы могли разделить операции на две группы – операции собственно добавления элемента в массив ($c_{add} = 1$ действие) и редкие операции копирования массива (и вот тут число действий равно размеру массива).

   Затем мы говорим что пусть сложность добавления $\tilde{c}_{add}$ будет равна 3, а сложность операций копирования обнулим. Утверждается что при таком перераспределении все будет корректно. Обоснуем это.

   Идея в том что когда мы платим 3 за операцию мы на самом деле платим 1 за добавление элемента в массив, 1 за **один** будущий перенос этого элемента, и еще 1 за перенос одного из элементов из левой половины массива, у которых денег на копирование себя уже не осталось (т.к. мы дали каждому элементу деньги на перенос себя только **один** раз).

   Т.е. в любой момент копирования в банке будет достаточно монет чтобы оплатить эту операцию, тем самым средняя сложность операции опять же $O(1)$, а в целом заполнение массива занимает $O(N)$ операций.

2) Наконец *метод потенциалов* похож на тот что мы использовали для построения нижних оценок – мы вводим потенциальную функцию (зависящую, к примеру, от структуры данных) и переходим к стоимостям вида

$$\tilde{c}_i = c_i + \Phi(i) - \Phi(i-1)$$

   и тогда из

$$\sum_{i=1}^{N} \tilde{c}_i = \sum_{i=1}^{N} c_i + \Phi(N) - \Phi(0) \Rightarrow \sum_{i=1}^{N} \tilde{c}_i + \Phi(0) - \Phi(N) \geqslant \sum_{i=1}^{N} c_i,$$

   т.е. получив оценку для суммы новых стоимостей операций мы сможем получить тем самым оценку на исходную сумму .

   Как и раньше, придумать такую функцию – самая сложная задача. Однако тут можно применить идеи метода предоплаты – там будет легко посчитать это если сложные операции мы как-то выровняем с простыми, к примеру как-то убрав влияние размера массива в задаче с динамическим массивом.

   Давайте, к примеру, будем рассматривать

$$\Phi i = -\#\text{ свободных ячеек в массиве после i-го шага}$$

   (именно с минусом), тогда в случае операций когда копирования не происходит получаем $\tilde{c}_i = 1 - (k-1) + k = 2$ (было k свободных, стало на 1 меньше), а для операций когда копирование происходит имеем

$$\tilde{c}_i = \overbrace{(1+k)}^{c_i} - \overbrace{(k-1)}^{k \text{ добавили, 1 заняли}} + 0 = 2$$

   и тогда

$$\sum_{i=1}^{N} c_i \leqslant \sum_{i=1}^{N} \tilde{c}_i + \Phi(0) - \Phi(N) \leqslant 2N + 1 + \overbrace{(N-1)}^{\text{макс. незаполненных}} = 3N$$

   таким образом амортизационная стоимость равна 3, это $O(1)$