Домашняя работа №8

Бредихин Александр

12 апреля 2020 г.

Задача 1

Задача: в языке C++ структура данных **`std::set`** *реализована с помощью красно-чёрных деревьев. В ней хранится множество ключей, при этом считается, что ключи упорядочены. Помимо стандартных операций (вставки, поиска, удаления), которые стоят $O(\log n)$, где n — число элементов в структуре данных, эта структура данных позволяет вывести все элементы в отсортированном порядке за $O(n)$. Опишите, как модифицировать красно-чёрное дерево так, чтобы сделать возможным такой вывод и не изменить стоимость стандартных операций. Считайте, что балансировка дерева после добавления и удаления элемента стоит $O(\log n)$.*

Красно-чёрное дерево делает дерево поиска сбалансированным, то есть высота красно-чёрного поиского дерева не превышает $\log(n)$, где n – количество вершин в дереве. Построим рекуривный алгоритм, который выводит все элементы в порядке возрастания:

Запускаемся от вершины дерева, если есть левая ветка, то рекурсивно запускаемся от неё, если её нет, то запускаемся от правой (если она есть). После этого выводим элемент в вершине, где мы находимся.

В результате этого мы напечатаем в порядке возрастания все элементы дерева (так как по определению поиского дерева в проекции на ось элементы находятся в порядке возрастания, то есть в самом левом листе наименьший, его мы и выведем первым и т.д.)
Докажем, что в случае сбалансированного дерева такой алгоритм работает за $O(n)$. Составим рекуренту: $T(n) = T(k)+T(n-k-1)+O(1)$, где k – количество элементов в левом поддереве.(вывод конкретной вершины производится за константу). Также мы знаем, что высота дерева не пре-

вышает $\log(n)$ (по определению красно-чёрного дерева). Следовательно, сумма всех операций не превысит сумму операций в полном бинарном дереве. Для него мы уже много раз считали ассимптотику и она оценивается суммой геометрической прогрессии:

$$O(1) \cdot (1 + 2 + 2^2 + \dots + 2^{\log(n)}) = \frac{1 \cdot \left(1 - 2^{(\log(n)+1)}\right)}{1 - 2} = O(1) \cdot 2^{\log(n)} = O(n)$$

Получается, вывод всех элементов в отсортированном порядке производится за $O(n)$, остальные операции также сохраняют свою сложность.

Задача 2

Задача: семейство хэш-функций $H = \{f, g, h\}$, отображающих множество ключей $\{0, 1, 2, 3\}$ в множество хэшей $\{0, 1, 2\}$, задано таблицей. Является ли оно универсальным?

ключи	значения f	значения g	значения h
0	0	1	0
1	0	2	2
2	1	2	2
3	1	1	0

В нашем случае мощность множества хэшей равняется 3, следовательно, в определении, значение $\frac{1}{m} = \frac{1}{3}$. Рассмотрим ключи 0 и 3. Для них: $f(0) \neq f(3)$, но $g(0) = g(3)$ и $h(0) = h(3)$.
То есть вероятность, для этой пары чисел, что $H(x) = H(y)$ равняется $\frac{2}{3} \neq \frac{1}{3}$, поэтому по определению семейство не является универсальным.

Задача 3

Задача: в структуре данных «динамический массив» (в C++ `std::vector`) удаление k-го элемента стоит $O(n-k)$, где n — количество элементов в массиве. Это происходит за счёт сдвигов: необходимо, чтобы первый элемент динамического массива находился в первой ячейке выделенной под массив памяти, а также нужно, чтобы между соседними элементами не было пустых ячеек памяти. Поэтому после очищения k-ой ячейки в результате удаления требуется последовательно сдвинуть $n-k$ последних элементов на единицу влево.

Так, удаление последнего элемента стоит $O(1)$, поэтому с помощью динамических масивов стандартно реализует стеки, а удаление первого элемента стоит $O(n)$, поэтому использовать динамические массивы для наивной реализации очереди (при извлечении из очереди первый элемент извлекают из массива, а потом удаляют, а новые элементы добавляются в конец массива) — не лучшее решение.

Однако, с помощью динамического массива можно эффективно реализовать очередь поплатившись двукратным увеличением памяти. Как и в случае наивной реализации, элементы будут извлекаться из начала массива A, а добавляться будут в его конец. Заведём указатель (целочисленная перменная b), который сначала указывает на первый элемент массива ($b = 0$). Когда требуется извлечь элемент из очереди, сам элемент физически удаляться не будет, а будет увеличиваться лишь указатель: при извлечении из очереди возвращается элемент $A[b]$, после чего b увеличивается на единицу. В каждый момент, когда $b \geq \lfloor \frac{n}{2} \rfloor$ (n — число элементов в массиве A), происходит удаление b элементов из массива, после чего b присваивается значение 0.

Уточните описанный выше алгоритм так, чтобы в результате n любых запросов к структуре данных «очередь» выполнялось $O(n)$ операций и при этом выделяемая под динамический массив память не превосходила удвоенную память, требуемую для хранения очереди.

Решение: Используем амортизационные сложности следующим образом: теперь удаление будет «стоить» 4 операции (1 за перемещение указателя b и 3 как бы кладём в банк на запас). Получается, когда $b \geq \lfloor \frac{n}{2} \rfloor$, мы делаем следующие операции: очищаем память до b (то есть делаем b операций), затем перенумеровываем наши элементы, то есть bый станет 0ым элементом b+1ый 1ым и так далее. Элементов, который нужно перенумеровать $b+1$. Получается, мы затратим операций: $b+b+1$. Заметим, что если $b \geq \lfloor \frac{n}{2} \rfloor$, то мы произвели b удалений, то есть «накопили» $3b$ операций, а тратим $2b + 1$. То есть нам хватает.

Покажем, что в результате n любых запросов выполнялось $O(n)$ операций. Эти запросы содержат k удалений и $n - k$ добавлений. Удаление занимает 4 операции, а добавление 1, следовательно, всего операций $4k + n - k = 3k + n$. Знаем, что $k \leq n$, поэтому количество операций $\leq 4n = O(n)$.

Доказали, что в результате n любых запросов к структуре данных «очередь» выполнялось $O(n)$ операций, докажем, что при этом выделяемая под динамический массив память не превосходила удвоенную память, требуемую для хранения очереди. То есть если L – длина «очередь» (количество элементов после b), то $n \leq 2L$.

1) если $b < \left[\frac{n}{2}\right]$, то $n - b > n - \left[\frac{n}{2}\right] = \left[\frac{n}{2}\right] \longrightarrow n < 2L$ – выполнено.

2) если $b = \left[\frac{n}{2}\right]$. Если n – x, то получаем, что $L = \frac{n}{2} \longrightarrow n = 2L$ – выполнено, аналогично если n нечётно, то $n = 2L - 2$.

Получается, доказали что выделяемая память не превосходит удвоенную память для хранения очереди.