

Федеральное государственное бюджетное образовательное учреждение высшего
профессионального образования

Московский государственный технический университет имени Н.Э.Баумана

(МГТУ им. Н.Э.Баумана)

ДОМАШНЕЕ ЗАДАНИЕ

ПО ДИСЦИПЛИНЕ «АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ»

***Сравнение декартового дерева (treap) и AVL-
дерева.***

Выполнил: Редченко Дмитрий Михайлович

ИУ8-52

Проверил: Чесноков Владислав Олегович

« ____ » _____ 2017 г.

Москва, 2017

Теоретическая часть

Декартово дерево (treap)

Декартово дерево - это структура данных, объединяющая в себе бинарное дерево поиска и бинарную кучу (отсюда и второе её название: treap (tree+heap) или же иногда в русских источниках встречается название дерамида (дерево+пирамида)¹. Думаю в этимологиях происхождения всех названий данной структуры данный мы разобрались с лихвой, настало время углубиться в её суть.

Описать данную структуру можно как структуру, которая хранит пары (x, y) в виде бинарного дерева таким образом, что она является бинарным деревом поиска по x и бинарной пирамидой по y . Предполагая, что все x и все y являются различными, получаем, что если некоторый элемент дерева содержит (x_0, y_0) , то у всех элементов в левом поддереве $x < x_0$, у всех элементов в правом поддереве $x > x_0$, а также и в левом, и в правом поддереве имеем: $y < y_0$.

Говоря о применении декартовых деревьев, хочется подчеркнуть сравнительную несложность их реализации, отсюда перечислю сферы их применения:

- Задачи, в которых требуется достаточно сбалансированное дерево поиска (например, эффективная реализация множеств или словарей);
- Поиск порядковых статистик (определение k -го по величине элемента) за $O(\log N)$. В этом случае нужно поддерживать поля `cnt`, и далее по этой информации определять, в каком поддереве будет располагаться нужный элемент;
- Решение обратной задачи (определение номера элемента в отсортированной последовательности) за $O(\log N)$. Требуется разрезать дерамиду по заданному ключу и определить количество элементов в левой части;
- Декартово дерево служит основой для ещё более мощной структуры данных, именуемой декартовым деревом по неявному ключу, которую можно рассматривать как массив, позволяющий производить большое количество различных операций с логарифмическим временем выполнения.

Описание

Итак у нас есть данные дерева — ключи x (здесь и далее предполагается, что ключ и является той самой информацией, которую мы храним в дереве). Второй элемент пары — y , назовем его *приоритетом*. Теперь построим такое волшебное дерево, которое хранит в каждой вершине по два параметра, и при этом *по ключам является деревом поиска, а по приоритетам — кучей*. Вот теперь, немного разобравшись мы поняли, какое дерево будем далее называть

¹ В ходе изучения литературы «всплывало» еще уж совсем смешное название — «дуча», очевидно «дерево» + «куча».

декартовым.

Встает сразу вопрос, почему же дерево называется декартовым. Получить ответ можно наглядно проиллюстрировав нашу структуру. Возьмем набор пар «ключ-приоритет» и расставим на координатной сетке соответствующие точки (x, y). А потом соединим соответствующие вершины линиями, образуя дерево. Таким образом, декартово дерево отлично укладывается на плоскости благодаря своим ограничениям, а два его основных параметра — ключ и приоритет — в некотором смысле, координаты. Результат построения показан на **рис.1** : слева в стандартной нотации дерева, справа — на декартовой плоскости.

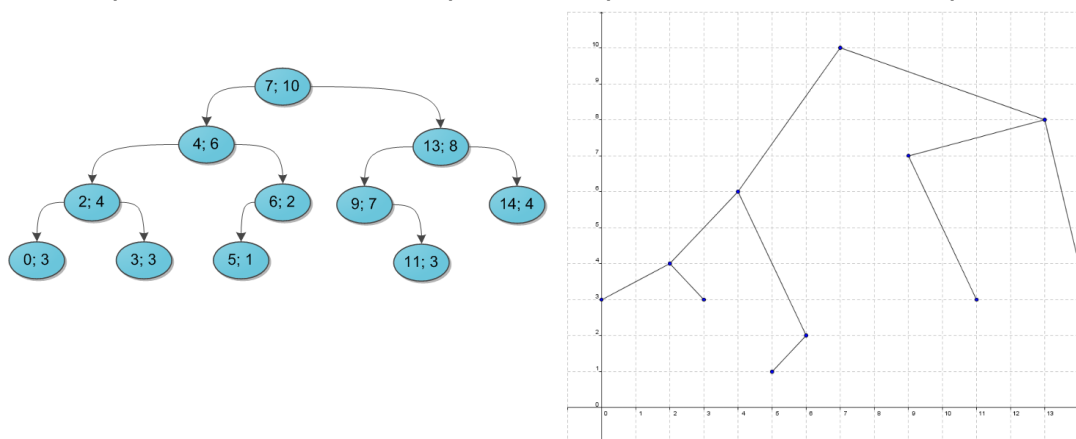


Рис 1. Декартово дерево в виде дерева и на координатной плоскости.

Достоинства

После наглядной иллюстрации, стало ясно только причина названия декартового дерева, однако вопрос «зачем такое нужно» остается открытым. Надо признать ответ на данный вопрос довольно тривиален, и заключается в следующих утверждениях.

Во-первых, пусть дано множество ключей: корректных деревьев поиска из них можно построить много различных, в том числе и спископодобное. А вот после добавления к ним приоритетов дерево из данных ключей можно построить уже лишь одно-единственное, вне зависимости от порядка поступления ключей. Во-вторых, давайте теперь сделаем наши приоритеты случайными. То есть просто ассоциируем с каждым ключом случайное число из достаточно большого диапазона, и именно оно и будет служить соответствующим игреком. Тогда полученное декартово дерево с очень высокой, стремящейся к 100% вероятностью, будет иметь высоту порядка $\log_2 N$. И даже если оно и не будет идеально сбалансировано, время поиска ключа в таком дереве все равно будет порядка $O(\log_2 N)$, что является достоинством(см. Приложение №1).

Не хочу выделять отдельный пункт на недостатки данной структуры данных, так как в целом ознакомившись с ней я испытал некую симпатию, поэтому здесь коротко напишу основные недостатки декартового дерева (и сразу перейду к достоинству со знаком *):

- Большие накладные расходы на хранение: вместе с каждым элементом хранятся два-три указателя и случайный ключ y.

- Скорость доступа $O(n)$ в худшем, хотя и маловероятном, случае. Поэтому декартово дерево недопустимо, например, в ядрах ОС.
- Много названий одной и той же структуры данных☺.

Достоинство со знаком *.

Знак звездочка в данном разделе означает ничто иное как дополнительную или в коей мере интересную информацию. Существует такие ситуации, когда есть возможность не делать приоритеты случайными, а использовать некую доп-информацию, которую мы храним в вершинах нашего дерева. При условии, что эту информацию можно считать случайной в контексте решения той или иной задачи, мы можем в качестве приоритета использовать некую (обязательно обратимую) функцию этой доп-информации. Однако, такие методы стоит применять опытным программистам, которые уверены, что такие действия не приведут к разбалансировке дерева.

Операции

Итак, treap предоставляет следующие операции:

- **Insert (x, y)**

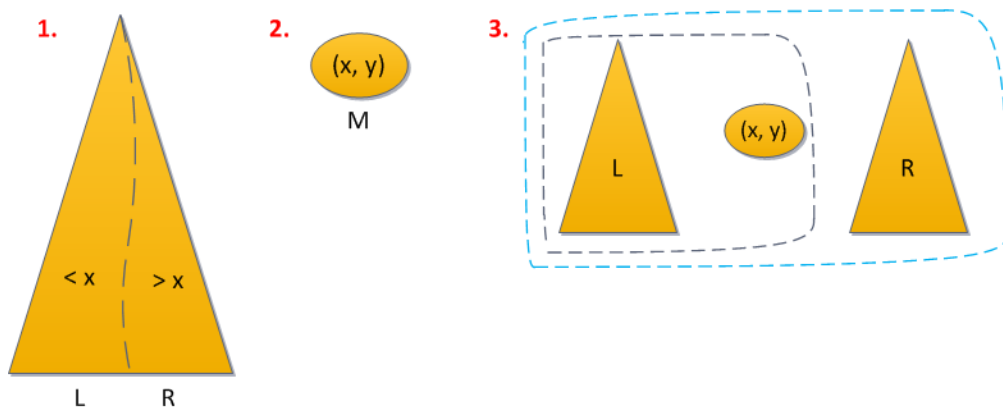
Выполняет добавление в дерево нового элемента.

Возможен вариант, при котором значение приоритета **y** не передаётся функции, а выбирается случайно (правда, нужно учесть, что оно не должно совпадать ни с каким другим **y** в дереве)².

Говоря о конкретной задаче вставки, то она решается реализацией подобных шагов.

- 1) Разделим (split) дерево по ключу **x** на дерево **L**, с ключами меньше **x**, и дерево **R**, с большими.
- 2) Создадим из данного ключа дерево **M** из единственной вершины (**x**, **y**), где **y** — только что сгенерированный случайный приоритет.
- 3) Объединим (merge) по очереди **L** с **M**, то что получилось — с **R**.

² Нужно быть осторожным в тех платформах, где `rand()` возвращает 15 случайных бит. В дальнейших рассуждениях мы рассчитываем на то, что все приоритеты различны. Нарушения этого правила редки, если генерировать случайные числа порядка 10^9 , несколько случайных совпадений на практике не влияют на быстродействие. Однако, если генерировать числа порядка 10^4 , то коллизии случаются часто и это может повлиять на время работы.



Сложность: $O(\log N)$ в среднем (1 split и два merge)

- **Contains (x)**

Ищет элемент с указанным значением ключа x . Реализуется абсолютно так же, как и для обычного бинарного дерева поиска. Так как дерамида является по ключам двоичным деревом поиска, то алгоритм поиска тривиален.

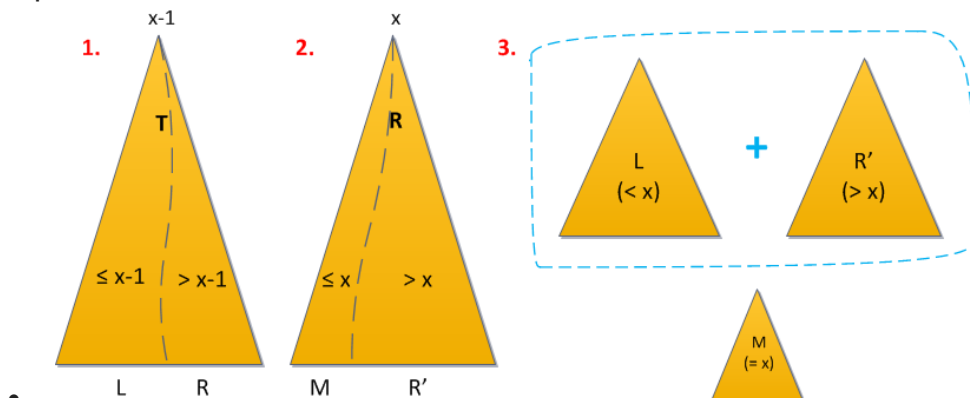
Сложность: $O(\log N)$ в среднем

- **Erase(x)**

Ищет элемент и удаляет его из дерева.

Пусть нас просят удалить из декартова дерева элемент с ключом x . Тогда совершим следующую последовательность действий:

- 1) Разделим сначала дерево по ключу $x-1$. Все элементы, меньшие либо равные $x-1$, отправились в левый результат, значит, искомым элемент — в правом.
- 2) Разделим правый результат по ключу x (здесь стоит быть аккуратным с равенством!). В новый правый результат отправились все элементы с ключами, большими x , а в «средний» (левый от правого) — все меньшие либо равные x . Но поскольку строго меньшие после первого шага все были отсеяны, то среднее дерево и есть искомым элемент.
- 3) Теперь просто объединим снова левое дерево с правым, без среднего, и дерамида осталась без ключей x .



Сложность: $O(\log N)$ в среднем

- **Split(T, x)**

Разделяет дерево T на два дерева L и R (которые являются возвращаемым значением) таким образом, что L содержит все элементы, меньшие по ключу x, а R содержит все элементы, большие x.

Сложность: $O(\log N)$

- **Merge(T₁, T₂)**

Объединяет два поддерева T₁ и T₂, и возвращает это новое дерево. Она работает в предположении, что T₁ и T₂ обладают соответствующим порядком (все значения X в первом меньше значений X во втором). Таким образом, нам нужно объединить их так, чтобы не нарушить порядок по приоритетам Y. Для этого просто выбираем в качестве корня то дерево, у которого Y в корне больше, и рекурсивно вызываем себя от другого дерева и соответствующего сына выбранного дерева.

Сложность: $O(N)$

- **Print(T)**

Вывод данного дерева планируется реализовать в подобном (см. Рис. 2) и\или формате вывода на координатную плоскость.

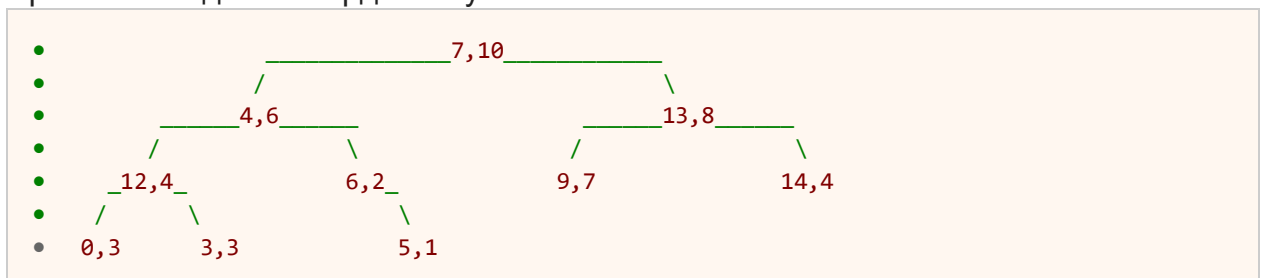


Рис 2. Вывод print

АВЛ-дерево

Определение

АВЛ-дерево — это прежде всего двоичное дерево поиска, ключи которого удовлетворяют стандартному свойству: ключ любого узла дерева не меньше любого ключа в левом поддереве данного узла и не больше любого ключа в правом поддереве этого узла.

Особенностью АВЛ-дерева является то, что оно является сбалансированным в следующем смысле: для любого узла дерева высота его правого поддерева отличается от высоты левого поддерева не более чем на единицу. AVL-деревья обычно используются в учебной практике, также на практике предлагается применять AVL-деревья в управлении динамической памятью. В целом сложность балансировки ограничивает область применимости данной структуры сильно фактически применение ограничивается теми местами, где поиск в дереве является доминирующей операцией). Еще возьмем такой пример: если мы храним в базе данных некоторые связанные данные (т.е. записи, состоящие из нескольких значений, возможно, различного типа), то можно выделить какое-то из этих значений (которое более-менее уникально характеризует запись или же мы будем часто ссылаться на эту запись по этому значению) в качестве ключа. Таким образом, зная ключ, пользователь нашей базы сможет достаточно быстро получить все данные, ассоциированные с ним.

Балансировка

Относительно АВЛ-дерева балансировкой вершины называется операция, которая в случае разницы высот левого и правого поддеревьев $= 2$, изменяет связи предок-потомок в поддереве данной вершины так, что разница становится ≤ 1 , иначе ничего не меняет. Указанный результат получается вращениями поддерева данной вершины (Вращения показаны наглядно в Таблице 1):

- Малое левое вращение
- Большое левое вращение
- Малое правое вращение
- Большое правое вращение

Операция приводит к нужному результату и что полная высота уменьшается не более чем на 1 и не может увеличиться. Также можно заметить, что большое левое вращение это композиция правого малого вращения и левого малого вращения. Из-за условия сбалансированности высота дерева $O(\log(N))$, где N - количество вершин, поэтому добавление элемента требует $O(\log(N))$ операций.

k - ключ

p – корень

Тип вращения	Условие применения	Графическое изображение вращений в АВЛ
--------------	--------------------	--

в AVL дереве		дереве
Малое левое вращение	Разница между высотой b- поддерева и L равна 2 и высота C меньше либо равна высоте R	
Большое левое вращение	Разница между высотой b- поддерева L равна 2 и высота c- поддерева больше высоты R.	
Малое правое вращение	Разница между высотой b- поддерева и R равна 2 и высота C меньше либо равна высоте L.	
Большое правое вращение	Разница между высотой b- поддерева и R равна 2 и высота c-поддерева больше высоты L.	

Таблица 1. Вращения в AVL-дереве.

Операции

- **insert(p, k)**

Вставка ключа **k** в дерево с корнем **p**. Вставка нового ключа в AVL-дерево выполняется, так же, как это делается в простых деревьях поиска: спускаемся вниз по дереву, выбирая правое или левое направление движения в зависимости от результата сравнения ключа в текущем узле и вставляемого ключа. Единственное отличие заключается в том, что при возвращении из рекурсии (т.е. после того, как ключ вставлен либо в правое, либо в левое поддерево, и это дерево сбалансировано) выполняется балансировка текущего узла.

Сложность: $O(\log(n))$

- **search(p, k)**

Поиск ключа **k** в дереве **p**. Поиск выполняется так же как в деревьях поиска. Для каждого узла сравниваем значение его ключа с искомым ключом. Если ключи одинаковы, то функция возвращает текущий узел, в противном случае функция вызывается рекурсивно для левого или правого поддерева.

Сложность: $O(\log(n))$

- **remove(p, k)**

Удаление ключа **k** из дерева **p**. Удаление происходит сложнее. Находим узел **p** с заданным ключом **k** (если не находим, то делать ничего не надо), в правом поддереве находим узел **min** с наименьшим ключом и заменяем удаляемый узел **p** на найденный узел **min**.

При реализации возникает несколько нюансов. Прежде всего, если у найденный узел **p** не имеет правого поддерева, то по свойству АВЛ-дерева слева у этого узла может быть только один единственный дочерний узел (дерево высоты 1), либо узел **p** вообще лист. В обоих этих случаях надо просто удалить узел **p** и вернуть в качестве результата указатель на левый дочерний узел узла **p**.

Сложность: $O(\log(n))$

- **print(p)**

Вывод данного дерева планируется реализовать в подобном стиле как и вывод «дучи» (см. Рис. 2)

Формат данных

Формат входных данных

На стандартном потоке ввода задаётся последовательность команд. Пустые строки игнорируются.

Команды: insert, delete, find, print, min, max

Каждая строка содержит ровно одну команду `command [key] [data]`, где `command` — заданные операции для данной структуры.

`key` — ключ, целое число;

`data` — данные, целое число.

Формат выходных данных

На выходе программы получаем выходной файл `out.txt` с приведенной структурой данных или же время выполнения сценариев работы.

План выполнения работы

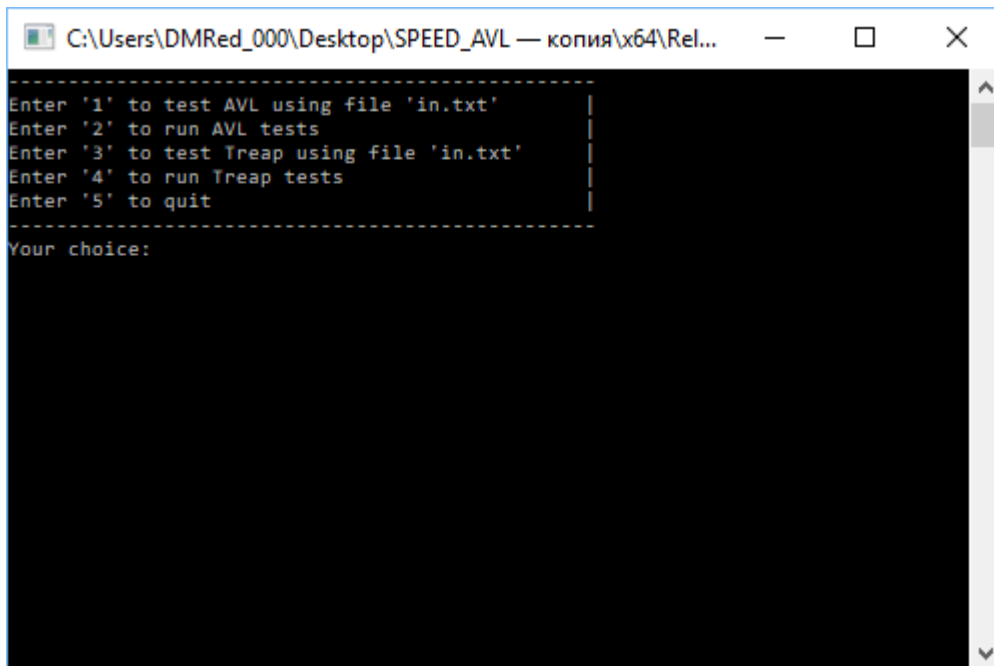
1. Реализовать AVL-дерево.
2. Реализовать «дучу».
3. BUG-FIXxXING или «доведение до ума».
4. Написание сценариев тестирования.
5. Анализ полученных результатов и написание вывода³.
6. Оформление отчета.

³ Время выполнения каждой операции будет вычислено при помощи стандартных библиотек, путем замера времени на проведение той или иной операции. Операция продельвается несколько раз. Сравнение проводится для абсолютно идентичных данных для первого и второго дерева. Далее, можно проделать операцию для разной длины данных и построить график зависимостей и времени от количества элементов.

Практическая часть

Логика программы

Программа состоит из 5 файлов, 4 из которых являются файлами классов Treap и AVL(2 header'a: crtsn.h и avl.h и 2 cpp'шника: crtsn.cpp и avl.cpp). Последний файл AVLvsTREAP.cpp содержит исполняемую функцию main() и функции тестов скорости операций над заданными структурами. С него начинается исполнение программы. При запуске нашей программы выводится меню (рис. 3) Там предлагаются варианты создания структуры данных из входного файла in.txt (согласно заданным в условии паттернам). Если строка соответствует паттерну, из нее, при необходимости, извлекается параметр (индекс искомого символа и т.д.) и вызывается метод, соответствующий требуемой операции.



```
-----
Enter '1' to test AVL using file 'in.txt'
Enter '2' to run AVL tests
Enter '3' to test Treap using file 'in.txt'
Enter '4' to run Treap tests
Enter '5' to quit
-----
Your choice:
```

Описание тестов

Для описания тестов смоделируем ситуацию(сценарий их использования). Для этого представлю себя организатором лотерейных розыгрышей. В данном случае в наших структурах планируется хранить номера лотерейных билетов.

Тест №1:

Описание: Итак представим главную ситуацию. Мои лотерейные билеты я хочу продавать на всей территории нашей протяженной страны. Чтобы не возникало путаницы я организовал производство (то бишь печать билетов) "на местах", и чтобы не запутаться у меня номера напечатанных лотерейных билетов идут по возрастанию по мере перемещения по карте нашей страны с запада на восток. Но продажи я хочу начинать в одно и то же время независимо от региона. Для этого хочется провести "стресс" тест и представить как я буду заполнять базу активированных билетов в случае, если они покупаются то в Калининграде то во Владивостоке. Этим собственно и занимается первый тест - заполнением структур данных чисел с большим разбросом.

Конкретная реализация: вставка 1 000 000 элементов в дерево с диапазоном значений 1 500 000 000

Тест №2:

Описание: Посидев, подумав, я решил посмотреть, а вдруг мне выгоднее проводить не один большой тираж по всей стране а равное им число тиражей по регионам. Так, собственно, и второй тест создает такое же число лотерейных билетов как и Тест №1, однако разбивает его по регионам(делит на мелкие структуры данных).

Конкретная реализация: вставка 1 000 000 элементов в 10 деревьев с диапазоном значений 1 500 000 000

Тест №3:

Описание: Дела мои с лотерейками пошли в гору, я понял, что сидеть на месте это не мое и пора выходить на новый уровень. Газеты пестрят заголовками что моя сеть расширяется и теперь я организую розыгрыши по всему СНГ - значит мне нужна большая структура. Тест 3 выясняет скорость работы большой структуры данных.

Конкретная реализация: вставка 15 000 000 элементов в дерево

Тест №4:

Описание: Этот тест показывает как определяются и сколько ищутся среди всех участников победители: 5 000 000 обладателей совсем мелких призов, 1 000 000 обладателей мелких призов, 50 000 - обладателей "нестыдных призов", и 3 безоговорочных суперультрамега победителя. Стоит сказать, что билеты выбранных победителей могут и не быть куплены или разыграны - как любят делать организаторы настоящих розыгрышей.

Конкретная реализация: 4 независимых поиска по большому дереву из 3-го теста. В общей сумме 6 000 050 003 поиска.

Тест №5:

Описание: Форс-мажорные обстоятельства(!!!). Я узнаю об ошибке печати лотереек на конкретном производстве - нужно срочно снять их с розыгрыша(и, возможно, вернуть деньги покупателям;).

Конкретная реализация: Удаление 1 500 000 элементов структуры данных.

Тест №6:

Описание: -

Конкретная реализация: Время min/max

Результаты тестов:

Примечание: тесты были произведены на моноблоке LENOVO 10150 в версии RELEASE x64 (в DEBUG медленнее).

ТЕСТ СТРУКТУРА	TEST №1, ms	TEST №2	TEST №3	TEST №4	TEST №5	TEST №6
AVL	1351	1120	34332	134	99	1
TREAP	414	1017	5022	125	21	0

Результаты тестов:

Проанализировав результаты тестов, можно сделать вывод о том, что на больших данных, структурах TREAP имеет внушительное преимущество в операциях вставки и удаления (тогда как время поиска эквивалентно), однако если необходимо создать несколько средних структур одинакового размера, то в данном случае эффективнее использовать AVL.

Приложение №1

Теорема:⁴

В декартовом дереве из n узлов, приоритеты y которого являются случайными величинами с равномерным распределением, средняя глубина вершины $O(\log n)$.

Доказательство:

▷

Будем считать, что все выбранные приоритеты y попарно различны.

Для начала введем несколько обозначений:

- x_k — вершина с k -ым по величине ключом;
- $A_{i,j} = \begin{cases} 1, & x_i \text{ is ancestor of } x_j \\ 0, & \text{otherwise} \end{cases}$
- индикаторная величина
- $d(v)$ — глубина вершины v ;

В силу обозначений глубину вершины можно записать как количество предков:

$$d(x_k) = \sum_{i=1}^n A_{i,k}.$$

Теперь можно выразить математическое ожидание глубины конкретной вершины:

$$E(d(x_k)) = \sum_{i=1}^n Pr[A_{i,k} = 1]$$

— здесь мы использовали линейность математического ожидания, и то что $E(X) = Pr[X = 1]$ для индикаторной величины X ($Pr[A]$ — вероятность события A).

Для подсчёта средней глубины вершин нам нужно сосчитать вероятность того, что вершина x_i является предком вершины x_k , то есть $Pr[A_{i,k} = 1]$.

Введем новое обозначение:

- $X_{i,k}$ — множество ключей $\{x_i, \dots, x_k\}$ или $\{x_k, \dots, x_i\}$, в зависимости от $i < k$ или $i > k$. $X_{i,k}$ и $X_{k,i}$ обозначают одно и тоже, их мощность равна $|k - i| + 1$.

Лемма:

Для любых $i \neq k$, x_i является предком x_k тогда и только тогда, когда x_i имеет наибольший приоритет среди $X_{i,k}$.

Доказательство:

▷

Если x_i является корнем, то оно является предком x_k и по определению имеет максимальный приоритет среди всех вершин, следовательно, и среди $X_{i,k}$.

С другой стороны, если x_k — корень, то x_i — не предок x_k , и x_k имеет максимальный приоритет в декартовом

⁴ Источник - https://neerc.ifmo.ru/wiki/index.php?title=Декартово_дерево#, доступно на 03.11.2017

дереве; следовательно, x_i не имеет наибольший приоритет среди $X_{i,k}$.

Теперь предположим, что какая-то другая вершина x_m — корень. Тогда, если x_i и x_k лежат в разных поддеревьях, то $i < m < k$ или $i > m > k$, следовательно, x_m содержится в $X_{i,k}$. В этом случае x_i — не предок x_k , и наибольший приоритет среди $X_{i,k}$ имеет вершина с номером m .

Наконец, если x_i и x_k лежат в одном поддереве, то доказательство применяется по индукции: пустое декартово дерево есть тривиальная база, а рассматриваемое поддерево является меньшим декартовым деревом.
◁

Так как распределение приоритетов равномерное, каждая вершина среди $X_{i,k}$ может иметь максимальный приоритет, мы немедленно приходим к следующему равенству:

$$Pr[A_{i,j} = 1] = \begin{cases} \frac{1}{k-i+1}, & k > i \\ 0, & k = i \\ \frac{1}{i-k+1}, & k < i \end{cases}$$

Подставив последнее в нашу формулу с математическим ожиданием получим:

$$E(d(x_k)) = \sum_{i=1}^n Pr[A_{i,k} = 1] = \sum_{i=1}^{k-1} \frac{1}{k-i+1} + \sum_{i=k+1}^n \frac{1}{i-k+1} \leq \sum_{i=1}^n \frac{1}{i} \leq \ln(n) + 1$$

(здесь мы использовали неравенство $\sum_{i=1}^n \frac{1}{i} \leq \ln(n) + 1$)

$\log(n)$ отличается от $\ln(n)$ в константу раз, поэтому $\log(n) = O(\ln(n))$.

В итоге мы получили что $E(d(x_k)) = O(\log(n))$.
◁

Таким образом, среднее время работы операций `split` и `merge` будет $O(\log(n))$.

Список литературы

1. Н. Вирт, Алгоритмы и структуры данных — *сбалансированные деревья по Вирту*
2. Д. Кнут, Искусство программирования — *раздел 6.2.3*
3. <https://habrahabr.ru/post/101818/>
4. Личный сайт программиста: <http://e-maxx.ru/algo/treap>
5. Олимпиадное программирование в УлГТУ:
<https://acm.khpnets.info/w/index.php?title=%D0%94%D0%B5%D0%BA%D0%B0%D1%80%D1%82%D0%BE%D0%B2%D0%BE%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE>
6. Викиконспекты университета ИТМО:
<https://neerc.ifmo.ru/wiki/index.php?title=%D0%90%D0%92%D0%9B-%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE#.D0.9F.D1.80.D0.B8.D0.BC.D0.B5.D1.80.D1.8B>
7. Иващенко Д., Семенов К. Декартово дерево
8. Сравнение АВЛ и Красно-черного дерева. Опрышко А. В., студент.