Курс kiev-clrs – Лекция 11. Расширение структур данных. Деревья отрезков.

Иван Веселов

2009 г.

Содержание

1 План лекции

- Динамические порядковые статистики
- Методология расширения структур данных
- Деревья отрезков

2 Введение

Сегодня мы познакомимся с расширением структур данных. Это весьма распространённый и общепринятый подход: в процессе решения задачи мы обычно не создаём новые структуры данных “с нуля”, а скорее дополняем или расширяем уже существующие структуры в своих целях. Это и называется расширение структур данных.

3 Динамические порядковые статистики

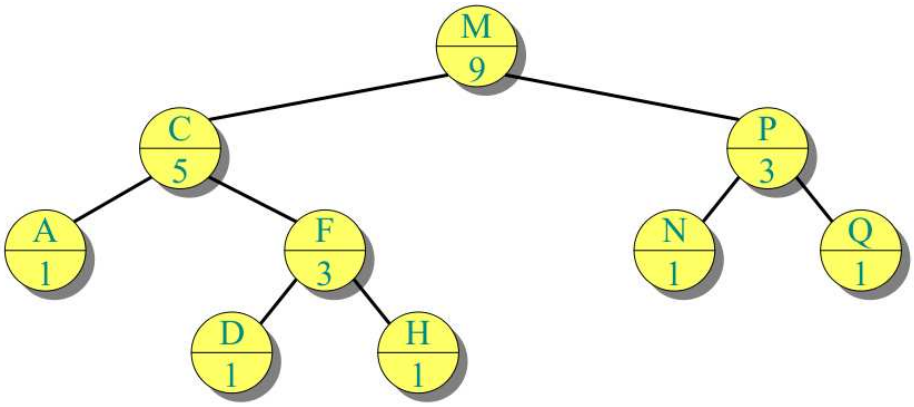
Мы уже знакомы с порядковыми статистиками, медианами и т.д. и находили их в массиве. Теперь предположим, что нам нужно найти порядковые статистики в динамическом множестве, в которое мы можем добавлять и удалять элементы.

Введём следующие дополнительные операции для нашего множества:

OS-Select(i, S) – возвращает i-й по порядку элементы (i-й наименьший элемент)

OS-Rank(x, S) – возвращает ранг $x \in S$ если множество S отсортировано

Идея: использовать для представления множества красно-чёрное дерево и хранить в его узлах размер поддеревьев этого узла.



$$size[x] = size[left[x]] + size[right[x]] + 1$$

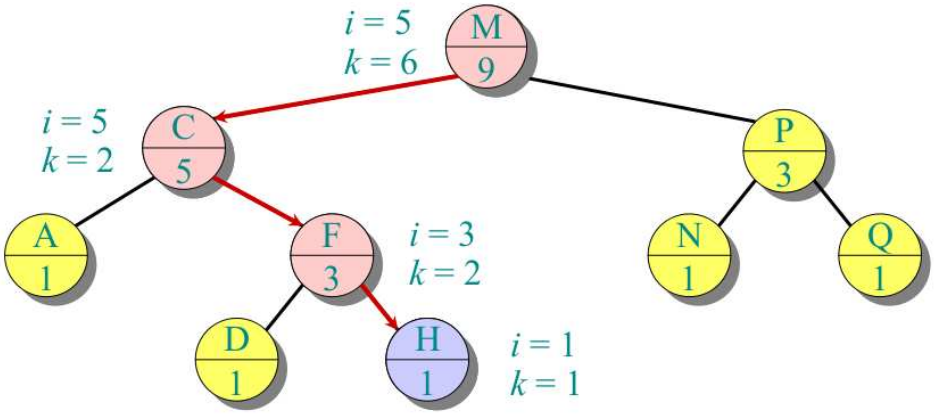
Храним в нодах два значения: ключ и размер поддеревьев.

(Предложить раскрасить дерево самостоятельно!)

Есть небольшая проблема с определением размера листьев дерева, т.к. у них левый и правый ребёнок – налл. Размер налла, очевидно, ноль, но это может вызвать ошибку. И для того чтобы не загромождать алгоритм проверками на налл, часто используют так называемый sentinel, фиктивная нода, такая что size[sentinel] = 0. И вместо того чтобы хранить налл, листья указывают на эту фиктивную ноду.

OS_Select(x, i)
1 $k \leftarrow size[left[x]] + 1$ $//k = rank[x]$
2 **if** $i = k$
3 **then return** x
4 **if** $i < k$
5 **then return** OS_Select$(left[x], i)$
6 **else return** OS_Select$(right[x], i - k)$

Пример: нахождение пятого по порядку элемента, т.е. OS_Select$(root, 5)$:



Анализ: OS_Select выполняется за $O(\lg n)$ операций, поскольку это красно-чёрное дерево.

Вопрос: почему бы не хранить сами ранги в дереве?

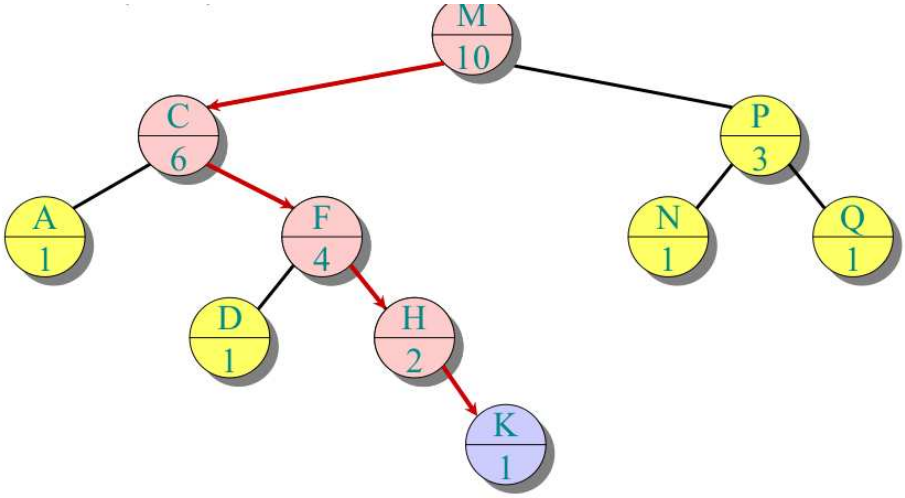
Ответ: неудобно поддерживать дерево. При вставке элемента в начало, придётся обновить все ранги. В случае же хранение размеров поддеревьев – только $\log n$ (вверх от элемента до корня дерева).

При расширении структуры данных важно понимать, что не только новые, добавленные операции должны работать быстро, но при этом и старые, уже существовавшие в базовой структуре операции не должны стать медленнее. Нужно сохранять их свойства, корректность и производительность.

Модифицирующие операции: вставка и удаление.

Стратегия сохранения свойств при вставке: пересчитывать размеры поддеревьев нескольких нод в процессе вставки.

Пример вставки элемента K:

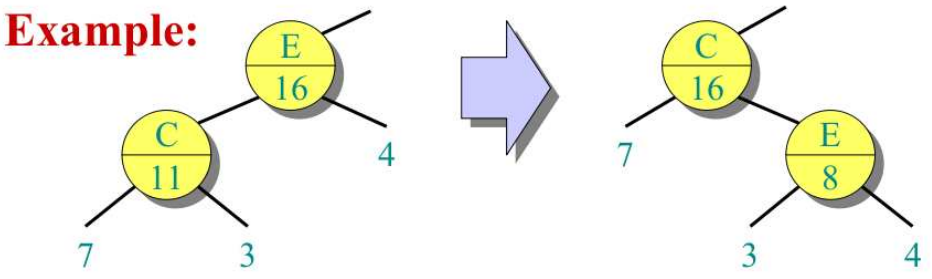


Кроме вставки элемента как в обычное бинарное дерево мы должны провести и ребалансировку дерева.

Ребалансирующие операции:

- изменение цветов узлов – не структурное изменение дерева, потому не требуется менять значения размера поддерева
- поворот – меняется структура, значит нужно пересчитывать и размер поддерева. Мы можем сделать это за $O(1)$ шагов, поскольку может быть только два поворота за один шаг и в процессе пересчёта мы пересчитываем размеры поддервьев только у одной ноды на основе её детей.

Повороты и пересчёты:



Таким образом, вставка продолжает выполняться за время $O(\lg n)$

То же можно сказать и об удалении.

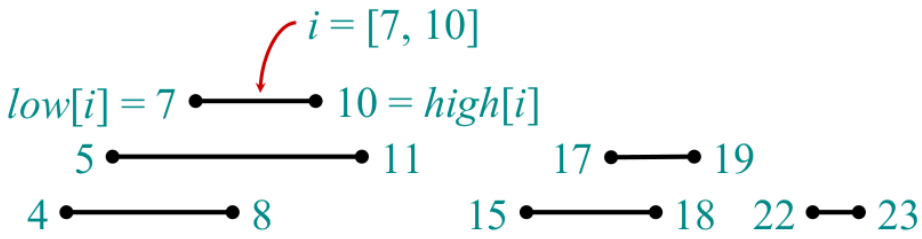
4 Методология расширения структур данных

1. Выбрать подходящую СД (красно-чёрные деревья)
2. Определить дополнительную информацию, которая будет храниться в СД (размеры поддеревьев)
3. Проверить, что эта информация может поддерживаться при модифицирующих операциях (удаление, вставка в КЧ-деревьях, учитывая повороты)
4. Задать новые операции, которые решают изначально поставленные задачи на основе дополнительной информации (OS-Select, OS-Rank)

Все эти шаги – это набор рекомендаций, а не строгое пошаговое руководство к действию. Обычно они выполняются в любом удобном порядке.

5 Деревья отрезков

Цель: поддерживать динамическое множество отрезков (например временных промежутков).



Задача: Для любого заданного отрезка i, найти отрезок, который перекрывается с заданным и находится в нашем множестве (любой, если их несколько).

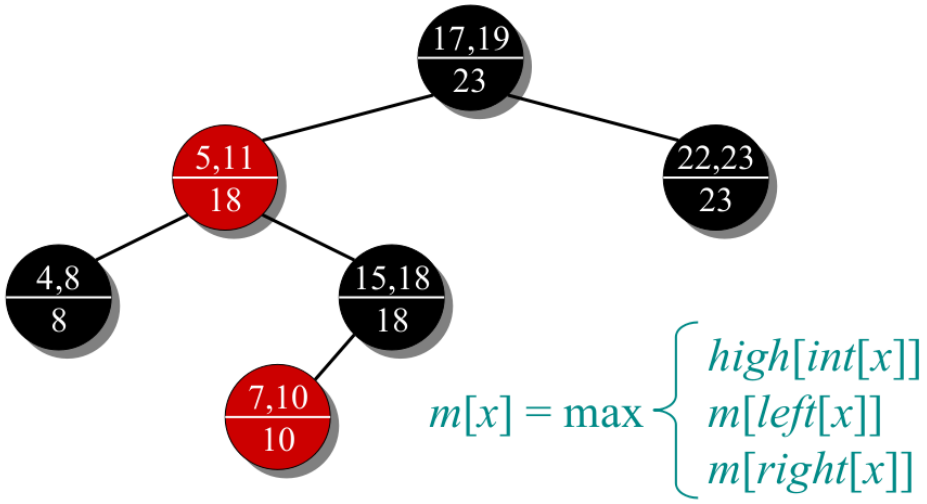
Следуем методологии:

1. Выбираем структуру данных: КЧ-деревья, в качестве ключа – нижняя граница отрезка. Таким образом, центрированный обход дерева приводит к перечислению отрезков в порядке сортировки по их левым концам.

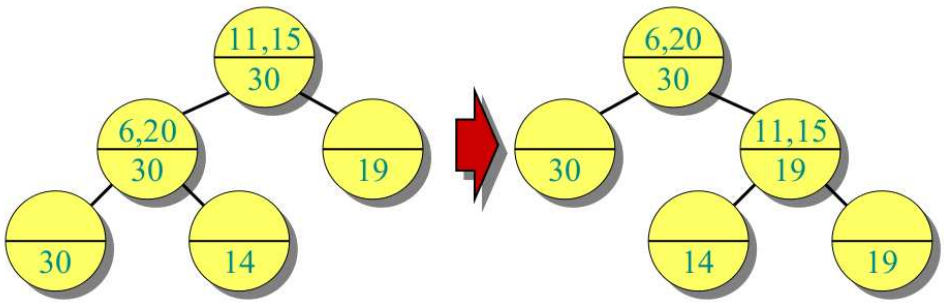
2. Дополнительная информация: будем хранить в каждом узле x значение $max[x]$, которое представляет собой максимальное значение всех конечных точек отрезков, хранящихся в поддереве, корнем которого является x. Проще говоря: самую правую точку поддерева отрезков.

3. Поддержка информации: после вставки мы должны пересчитать значение max, это можно сделать с помощью формулы:

$$max[x] = \max(high[int[x]], max[left[x]], max[right[x]])$$



Выполняем пересчёт max по мере спуска от корня, в процессе поиска места в дереве, куда будет производится вставка. Затем нужно выполнить балансировку, в частности вращения. Во время вращений вычисление max выполняется за константное время. Таким образом, операции вставки (аналогично и удаления) продолжают выполнятся за $O(\log n)$

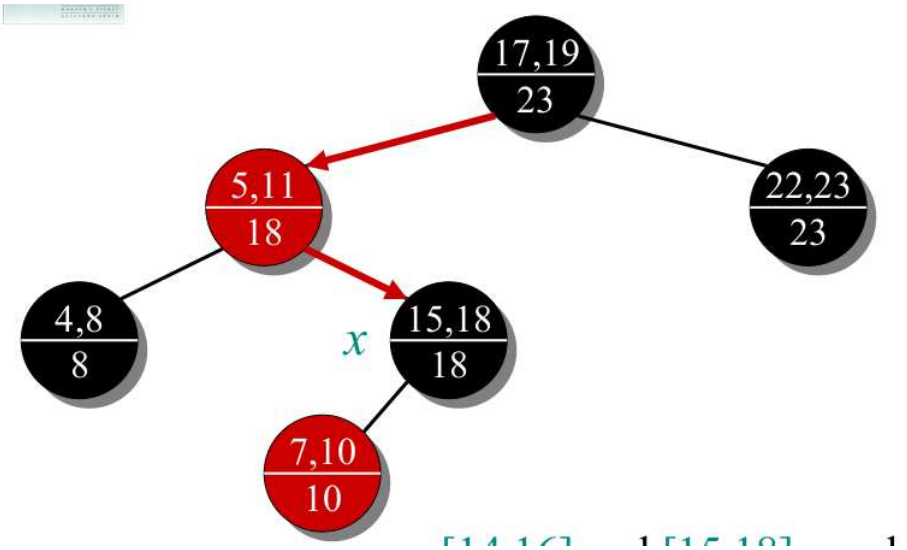


4. Новые операции.

INTERVAL_SEARCH(T, i)

```
1  x ← root[T]
2  while x ≠ NIL и i не перекрывается с int[x]
3        do if left[x] ≠ NIL и low[i] ≤ max[left[x]]
4              then x ← left[x]
5              else  x ← right[x]
6  return x
```

Пример поиска отрезка $[14, 16]$:



- начинаем с корня дерева ($[17, 19]$)
- проверяем не перекрывается ли текущий узел дерева (корень) с искомым отрезком (сравниваем $[14, 16]$ и $[17, 19]$ – не перекрываются)
- переходим к условию, т.е. сравниваем $low[i] = 14$ с $max[left[x]] = 18$, меньше – значит переходим к левому потомку ($[5, 11]$)
- снова проверяем не перекрывается ли текущий узел дерева с искомым отрезком (сравниваем $[14, 16]$ и $[5, 11]$ – не перекрываются)
- снова переходим к условию, т.е. сравниваем $low[i] = 14$ с $max[left[x]] = 8$, больше – значит переходим к правому потомку ($[15, 18]$)
- снова проверяем не перекрывается ли текущий узел дерева с искомым отрезком (сравниваем $[14, 16]$ и $[15, 18]$ – перекрываются)

- возвращаем $[15, 18]$

На примере $[12, 14]$ можно показать как работает алгоритм, когда дерево не содержит перекрывающихся с заданным отрезков.

Анализ: $O(\lg n)$, т.к. каждый шаг мы делаем элементарные операции и спускаемся вниз на один уровень, т.е. время выполнения пропорционально высоте дерева.

Для поиска всех перекрывающихся отрезков можно применить такую технику: найти, вывести, удалить, снова найти и т.д. После завершения поиска все удалённые отрезки добавить вновь. Время выполнения такого алгоритма $O(k \log n$, где k – количество выведенных отрезков. Такой алгоритм зависит от вывода, т.е. от того сколько он выводит.

Лучший известный результат на сегодня: $O(k + \log n)$

5.1 Доказательство корректности

Идея: на каждом шаге мы выбираем куда идти: влево или вправо. Если нам удастся показать, что мы всегда выбираем правильный путь, то нам удастся доказать корректность. "Правильный путь" означает, что мы всегда выдадим перекрывающийся интервал, если он есть.

Теорема: Пусть L – множество отрезков из левого поддерева узла x, т.е.

$$L = \{i' \in left[x]\}$$

R – множество отрезков из правого поддерева узла x, т.е.

$$R = \{i' \in right[x]\}$$

- если поиск идёт вправо, то

$$\{i' \in L \colon i' \text{ перекрывается с } i\} = \emptyset$$

- если поиск идёт влево, то

$$\begin{aligned} &\{i' \in L \colon i' \text{ перекрывается с } i\} = \emptyset \\ \Rightarrow &\{i' \in R \colon i' \text{ перекрывается с } i\} = \emptyset \end{aligned}$$

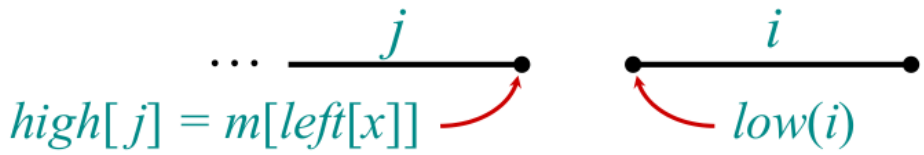
Другими словами: всегда безопасно выбирать только одного из детей узла дерева: мы либо найдём то, что нужно, а если не найдём – то значит ничего и не было.

Доказательство. Рассмотрим два случая:

1. Пусть поиск пошёл вправо.

   Такое возможно (см. псевдокод алгоритма) если $left[x] = NIL$, тогда поскольку левого поддерева нет вообще, выполняется $L = \emptyset$

   Иначе, чтобы поиск пошёл влево необходимо чтобы $low[i] > max[left[x]]$. Значение $max[left[x]]$ соответствует самой правой точке некого отрезка j из множества L. Поскольку эта точка *самая правая*, то никакой другой отрезок из L не может иметь свою правую точку ещё правее чем $high[j]$.



   Таким образом,

$$\{i' \in L : i' \text{ перекрывается с } i\} = \emptyset$$

2. Пусть поиск пошёл влево.

   Также по условию теоремы мы предполагаем, что

$$\{i' \in L : i' \text{ перекрывается с } i\} = \emptyset$$

   Согласно псевдокоду алгоритма:

$$low[i] \leq m[left[x]] = high[j]$$

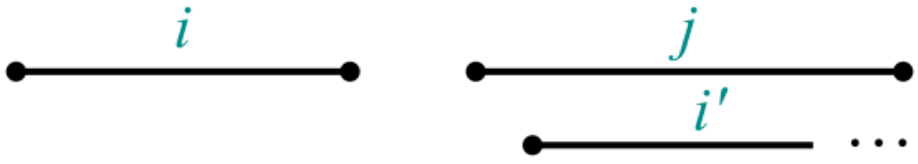
   для некоего отрезка j из L.

   Поскольку $j \in L$, то согласно предпосылке теоремы j не пересекается с i и как следствие получаем, что $high[i] < low[j]$ (т.к. существуют всего три варианта взаимного расположения отрезков, они либо перескаются, либо первый слева от второго, либо первый справа от второго).

   Теперь, используя свойство двоичных деревьев поиска и учитывая, что в качестве ключа мы используем левую точку отреза, отметим, что для всех $i' \in R$ выполняется $low[j] \leq low[i']$

   В таком случае:

$$\{i' \in R : i' \text{ перекрывается с } i\} = \emptyset$$



Что и требовалось доказать.