Министерство образования и науки Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего профессионального образования
«Московский физико-технический институт
(национальный исследовательский университет)»

Физтех-школа Прикладной Математики и Информатики

Кафедра банковских информационных технологий

Самоподстраивающиеся структуры данных с параллельным применением набора операций

(бакалаврская работа)

Направление: 03.03.01 «Прикладные математика и физика»

Выполнил студент гр. 791 _____ В.В. Краснов

Научный руководитель,
к.т.н. _____ В.Е. Аксенов

Москва – 2021

Аннотация

Объектом исследований данной работы являются самоподстраивающиеся структуры данных. Так как запросы к базе данных обычно имеют неравномерное распределение, предоставление более быстрого досутпа к наиболее частозапрашиваемым элементам является логичным этапом оптимизации структуры данных. Целью данной работы является разработка такой самоподстраивающейся структуры данных с параллельным применением набора операций.

При выполнении работы были поставлены следующие задачи: разработать и реализовать структуру данных, обладающую свойством самоподстраивания и способную выполнять набор операций вставки, поиска и удаления параллельно; проанализировать эффективность разработанного алгоритма; сравнить полученную структуру данных с аналогами.

Результатом работы является разработанный алгоритм самоподстраивающегося дерева поиска с параллельным применением набора операций. Полученное дерево является статически оптимальным и может выполнять набор из m запросов за $\mathcal{O}(\log(m)\log(n))$. Реализованная структура показала лучшую эффективность в сравнении со стандартными аналогами.

Оглавление

Введение

Хотя большинство алгоритмов балансировки деревьев стараются поддержтваать логарифмическую глубину дерева, они никак не учитывают частоту использоваяния тех или иных элементов, в то время как использование различных структур данных зачастую носит неравномерный характер. В таких случаях доступ к определенным элементам вызывает больший интерес, в то время как другие элементы так и остаются нетронутыми после добавления. Таким образом одним из способов ускорения средней работы алгоритма является предоставление быстрого доступа к частым элементам за счет замедления доступа к остальным. Тут то нам и приходят на помощь самоподстраивающиеся структуры данных. Такие структуры данных имеют неоднородную асимптотику доступа к различным элементам за счет постоянного изменения своей структуры на основе собранной статистики предыдущих обращений к API, в чем и заключается породившее название таких структур самоподстраивание. В этой работе была поставлена задача разработать и реализовать самоподстраивающуюся структуру данных с параллельным применением набора операций.

Глава 1

Обзор предметной области

Концепция самоподстраивающихся структур была придумано достаточно давно. Наиболее известными представителями такого класса объектов являются splay-tree [5] и tango-tree[4] основанные на двоичном дереве поиска и splay-list, предоставляющий более быстрый доступ к некоторым элементам, за счет пропуска большинства элементов в связанном списке. Следующим шагом в ускорении алгоритма является его многопоточная реализация. Примеры реализации конкурентных версий таких алгоритмов были описаны в статьях "CBTree: A Practical Concurrent Self-adjusting Search Tree-[2] и "The Splay-List: A Distribution-Adaptive Concurrent Skip List-[3]. Однако не всегда запросы приходят по одному, ровно как и не во всех случаях их выгодно выполнять по одному.

В этой работе мы рассмотрим алгоритм структуры данных, основанной на двоичном дереве поиска и отвечающей свойствам самоподстраивающейся структуры данных. Основной отличительной особенностью предоставленного алгоритма является возможность параллельного набора операций. Также предложенный алгоритм является статически-оптимальным, что означает он амортизированно не уступает по эффективности самому оптимальному бинарному дереву, которое можно предоставить, зная распределение запросов.

Глава 2

Параллельное самоподстраивающееся дерево

В данной главе будут рассмотрены основные концепции и необходимый инструмент для описания алгоритма работы параллельного самоподстраивающегося дерева поиска.

2.1. Определение

Для начала определим рассматриваемый объект и требуемый функционал. В основе структуры данных лежит двоичное дерево поиска, т.е. элементы в левом поддереве меньше чем в корне, а в правом больше, чем в корне. Таким образом, алгоритм поиска в нем наследуется от классической структуры данных. Рассматриваемые операции также аналогичны стандартному аналогу: поиск наличия элемента в дереве, а также вставка и удаление элементов. Первое отличие заключается в том, что выполнять мы будем не одну операцию, а сразу некоторый набор из m операций. Каждая операция представляет из себя пару: (тип операции, ключ). Здесь тип операции это значение из списка: поиск, вставка, удаление. Ключом может являться любой объект, для которого определено отношение порядка (оператор сравнения). Для ускорения работы запросы из одного набора выполняются одновременно и параллельно. Для простоты и определенности дальнейших рассуждений будем считать, что выполняемый набор операций отсортирован по возрастанию по ключам (если нет, то перед исполнением проведем сортировку).

2.2. Параллельное выполнение набора запросов

Итак, мы дошли до применения набора операций. Как говорилось выше, применять их мы будем параллельно. В этом нам поможет главное свойство бинарного дерева. Мы знаем, что все элементы в левом поддереве меньше чем все элементы в правом. Тогда, разделив запросы по ключу в корне (например используя partition – функцию, которая принимает на вход массив и ключ и разделяет исходные массив на два подмассива, в одном из которых находятся все элементы меньшие чем ключ, а в другом большие либо равные), мы получим два подмножества исходного множества запросов: в одном будут запросы к элементам левого поддерева, а в другом – правого. Также, рано или поздно окажется, что среди запросов есть запрос к текущему корню, в таком случае мы применим необходимую операцию.

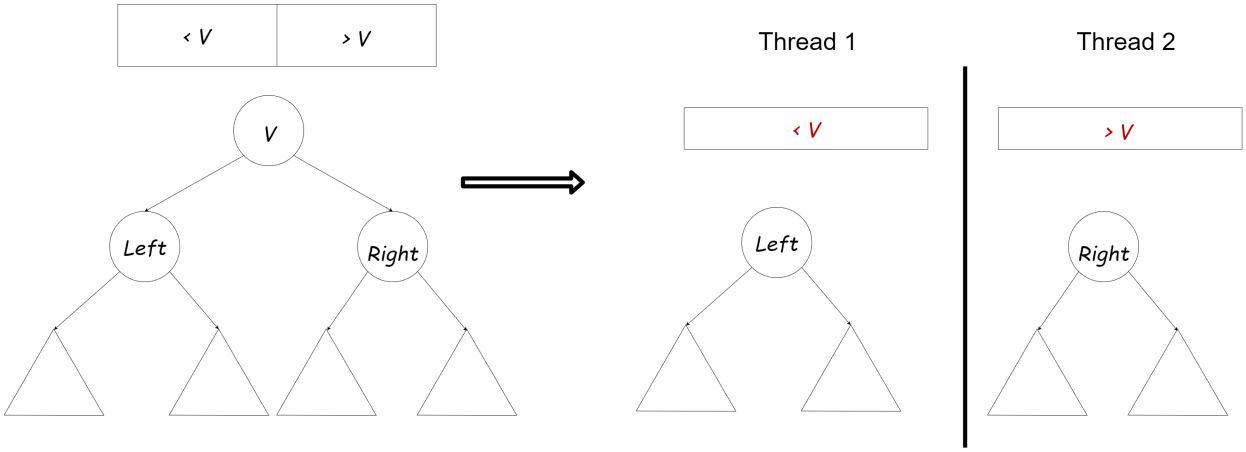

Рис. 2.1. Параллельное применение операций

Подробнее о применении операции будет рассказано дальше. Таким образом, у нас есть два независимых поддерева и два независимых набора операций к ним, следовательно, эти наборы операций можно применить к этим поддеревьям параллельно. Далее в каждом потоке рекурсивно повторим данный алгоритм.

2.3. Функция потенциалов

Введем ключевые понятия, которыми будем оперировать далее. На текущий момент у нас имеется структура данных с параллельным применением набора операций, у которой не хватает самоподстраивания. Балансировка по глубине поддерева нас не устраивает, ведь она никак не учитывает частоту обращения к элементам. Чтобы учесть это, будем хранить в каждом узле дерева суммарное количество обращений к элементам этого поддерева. Далее эта величина фигурирует как вес поддерева и обозначается W. Таким образом $W(v)$ — это вес дерева с корнем в вершине v. Заметим, что вес конкретной вершины подсчитать довольно быстро и просто, нужно всего лишь вычести из веса всего дерева веса левого и правого поддеревьев: $w(v) = W(v) - W(v.left) - W(v.right)$. Для того, чтобы обеспечить геометрическое убывание весов при спуске по дереву будем оперировать не просто значением веса, а его логарифмом. Назовем такую величину рангом дерева $r(v)$. И, наконец, потенциалом дерева Φ назовем сумму рангов всех его вершин.

Основные обозначения	
$w(v)$	вес вершины — число запросов к конкретной вершине v
$W(v)$	вес поддерева — суммарное число запросов ко всему элементам в поддереве с корнем в вершине v
$r(v) = \log_2 W(v)$	ранг поддерева
$\Phi(v) = \sum r(v)$	потенциал поддерева – сумма рангов всех вершин поддерева

Таблица 2.1. Основные обозначения

2.4. Балансировка

Наконец мы можем поговорить о балансировке. Производится она при помощи обыкновенных одинарных поворотов. Для примера рассмотрим левый поворот, при котором текущий корень становится левым ребенком своего правого ребенка, а тот, в свою очередь, становится новым корнем. Далее всегда все рассуждения будут на основе левого поворота, для правого поворота все симметрично.

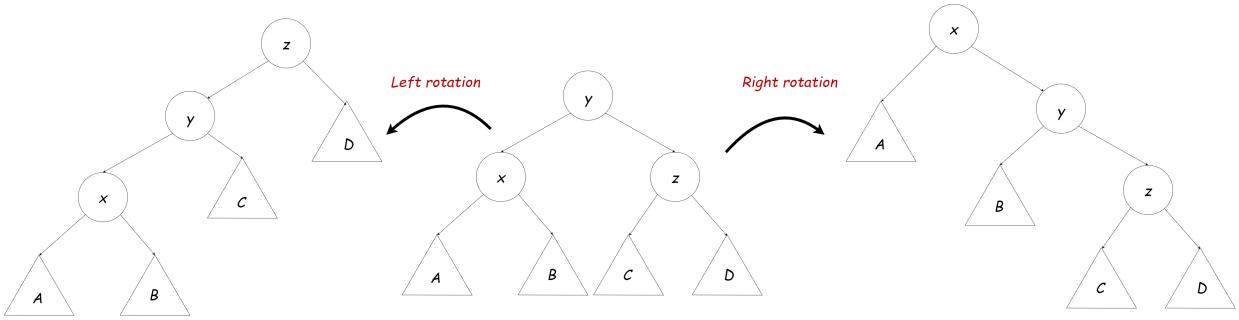

Рис. 2.2. Балансировка при помощи поворотов

Обозначим за $r'(v)$ ранг вершины v после выполнения поворота. Аналогично $\Phi'(v)$ – потенциал поддерева после поворота. Заметим, что при таком повороте изменяются лишь ранги вершин y и z. Тогда разность потенциалов будет зависеть лишь от этих вершин: $\Delta\Phi = \Phi' - \Phi = r'(y) + r'(z) - r(y) - r(z)$ Причем новый ранг $r'(z)$ вершины z будет равен рангу $r(y)$ вершины y до поворота:

$$r(y) = \log_2(W(x) + W(C) + W(D) + w(z) + w(y))$$

$$r'(z) = \log_2(W(x) + W(C) + w(y) + W(D) + w(z))$$

$$r'(z) = r(y)$$

Таким образом разность потенциалов между этими двумя состояниями:

$$\Delta\Phi = \Phi' - \Phi = r'(y) + r'(z) - r(y) - r(z) = r'(y) - r(z)$$

Отсюда можно сделать вывод, что подсчет разности потениалов не является дорогой процедурой и выполняется за константное время. Также, анализируя получившуюся формулу для разности потенциалов, заметим, что если при выполнении поворота новым корнем становится вершина, с большим рангом, разность потенциалов уменьшается. А ведь это именно то, чего мы и хотим добиться: вершины, с наибольшей частотой обращения находятся ближе к корню, чтобы минимизировать время доступа к ним. Таким образом, для достижения необходимого свойства нашей структуры данных, будем производить балансировку, стараясь минимизировать потенциал дерева на каждом шаге. Для этого, на каждой итерации достаточно производить поворот, результатом которого будет дерево с меньшим потенциалом или не производить поворота вовсе, если оба поворота увеличивают потенциал дерева.

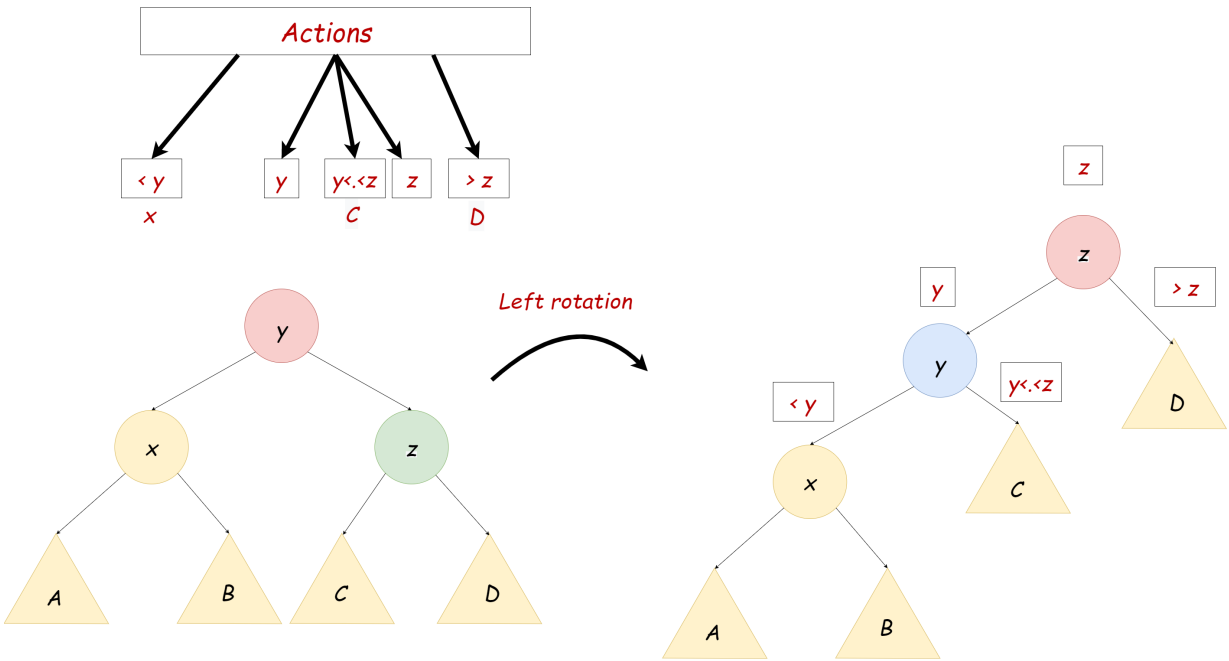

Рис. 2.3. Балансировка при помощи поворотов

Мы рассмотрели условие для балансировки, осталось выяснить, когда нам ее производить. Для начала, рассмотрим следующие свойства:

1. Ранги корня до поворота и нового корня после

поворота равны: $r'(z) = r(y)$.

2. На введенное выше условие поворота не влияют манипуляции, производимые в соседнем поддереве или выше рассматриваемого корня.

3. При проверке условия мы подразумеваем ранги, которые будут иметь вершины уже после применения к ним текущих операций и обновления весов в нужных узлах дерева.

4. Ранг вершины зависит лишь от общего числа операций производимых в рассматриваемом поддереве.

Из этих свойств следуют следующие утверждения:

1. При проверке условия поворота можно опираться лишь на текущую структуру дерева, не учитывая балансировку выше текущего корня и балансировку внутри поддеревьев.

2. Для оценки конкретных значений рангов вершин достаточно знать их текущий ранг и количество операций, производимых над поддеревом с корнем в данной вершине.

Таким образом мы можем производить балансировку при спуске.

Пусть при спуске мы попадаем в вершину y и имеем массив запросов actions для выполнения над этим поддеревом(см. рис. 2.3). Для начала разделим массив на 3 подмассива (например с помощью того же partition): запросы к поддереву с корнем в вершине x, запросы к поддеревьям C и D. На этом же этапе мы учитываем наличие запросов непосредственно к вершинам y и z. Тогда, обозначив за len(C) размер полученного подмассива, итоговая разность потенциалов будет равна:

$$\Delta\Phi = \log_2(W(x) + len(x) + W(C) + len(C) + w(y) + \{y\ is\ in\ actions\}) -$$

$$- \log_2(W(z) + len(z))$$

На основе рассчитанных потенциалов, мы можем решить выполнить правый, левый или не делать поворот и сразу же его применить, например, когда $\Delta\Phi < -\varepsilon$. Затем применить алгоритм рекурсивно к дочерним поддеревьям.

В четвертой главе мы покажем, что при выполнении таких поворотв время выполнения одного запроса происходит за $\mathcal{O}(\log \frac{W}{C(v)})$, где
$W = W(root)$ – вес текущего корня, а $C(v)$ – количество обращений к элементу v.

Глава 3

Алгоритм

3.1. Базовые структуры данных

Для начала определим используемые базовые структуры данных.

3.1.1. TreeNode

Структура `TreeNode` представляет из себя узел дерева. В ней хранится ключ, текущий вес и флаг, показывающий, что узел удален. Также в ней хранятся два указателя на дочерние узлы.

```
template<typename T>
struct TreeNode {
    T value;

    int weight;
    bool isDeleted;
    TreeNode *left;
    TreeNode *right;
}
```

3.1.2. Action

Вторая используемая структура это `Action`. Она представляет из себя пару: (ключ, тип операции), где ключ определяет цель операции, а тип может принимать одно из трех значений: вставка, удаление или поиск.

```
template <typename T>
struct Action {
    T value;
    ActionType actionType;
};
```

3.2. Алгоритм

Итак, наконец, мы можем собрать весь алгоритм во едино. Основной метод, выполняющий пришедшие запросы называется performActionsInParrallel. Он принимает на вход список запросов и массив, куда будет складывать ответы на эти запросы, а также текущий корень.

Algorithm 1: performActionsInParralel

Data: List<Action> actions, TreeNode* curRoot

Result: List<bool> *answers*

if *curRoot* == *null* **then**

  return create tree of actions values;

end

perform rotation if need;

perform action if need ;

root.weight ← root.weight + actions.size ;

doInParralel(

{performActionsInParralel($curRoot.Left$);},

{performActionsInParralel($curRoot.Right$);}

);

Выше приведен псевдокод основного рекурсивного алгоритма. В нем мы находим нужные нам элементы и управляем балансировкой. Также при спуске, мы можем оценить на сколько изменится вес вершины, после применения операций к ее поддеревьям и сразу обновить значение ее веса. Отдельно стоит выделить функцию doInParralel. Она запускает в двух потоках переданные ей в качестве аргументов инструкции и дожидается их выполнения. Роль этой функции в коде выполняет fork-join.

Далее рассмотрим некоторые части более подробно.

В первую очередь происходит проверка, что текущий корень не лист. Если он является листом, а у нас еще остались операции, не нашедшии свою цель, создадим для них узлы, организовав их в виде бинарного дерева

поиска и подвесим получившееся дерево к последнему листу.

Algorithm 2: create tree out of actions

Data: List<Action> actions

Result: TreeNode* newRoot

$m \leftarrow actions.size$;

$newRoot \leftarrow actions[m/2]$;

$newRoot.left \leftarrow$ create tree of actions values(actions[:m/2]);

$newRoot.right \leftarrow$ create tree of actions values(actions[m/2 + 1:]);

return newRoot;

Следующим шагом является проверка на необходимость выполнять тот или иной поворот и его выполнение.

Algorithm 3: perform rotation if need

Data: List<Action> actions, TreeNode* root

Result: TreeNode* newRoot

$deltaPhiLeft \leftarrow$ calculate $\Delta\Phi$ for left rotation;

$deltaPhiRight \leftarrow$ calculate $\Delta\Phi$ for right rotation;

if $deltaPhiLeft < EPS \ \& \ \& \ deltaPhiLeft < deltaPhiRight$ **then**

    newRoot ←make left rotation($curRoot$) ;

    return performActionsInParralel($newRoot$);

end

if $deltaPhiRight < EPS$ **then**

    newRoot ← make right rotation($curRoot$);

    return perfromActionsInParralel($newRoot$);

end

Для подсчета разности потенциалов при повороте и выполнения нужного поворота используются алгоритмы, описанные в главе про балансировку.

Затем мы проверяем, есть ли среди списка запросов запрос, с ключом равным текущему корню. Если такой присутствует, производятся действия,

зависящие от операции.

3.2.1. Вставка

Все операции (вставку, поиск и удаление) можно разделить на две части. Сначала мы ищем узел с нужным значением, если его нет, то создаем его. Эта часть оказывается общей для всех видов операций. Затем производим специфичные для каждой операции действия над полученным на прошлом шаге узлом. За эту часть отвечает функция performAction в приведенном выше псевдокоде.

Таким образом, чтобы вставить элемент, мы спускаемся по дереву, до тех пор пока либо не наткнемся на узел с нужным значением, либо не спустимся в лист. В случае, если мы нашли искомый элемент и он не помечен как удаленный, мы возвращаем $false$ в соответствующий элемент массива ответов. Если же элемент помечен как удаленный, то просто снимаем эту метку. Если же был достигнут лист, а нужный элемент так и не найдет, то просто создадим новый узел с искомым значением и подвесим его к последнему на пути листу.

3.2.2. Поиск

Как и во вставке мы сначала ищем нужный нам узел либо создаем его. Если элемент был найден в дереве и не помечен как удаленный, то в соответствующую ячейку массива ответов возвращается $true$. В противном случае – $false$. Заметим, что новый узел создается не только при вставке, но и при поиске и удалении. Это нужно, чтобы мы могли начать собирать статистику по элементу еще до того как он был вставлен, но к нему уже есть повышенный интерес.

3.2.3. Удаление

Аналогично предыдущим операциям, мы перемещаемся в нужный узел. Удалять узел физически нет смысла, ведь мы потеряем всю статистику по данному элементу, а также очень затратно, поэтому мы просто помечаем его как удаленный. Однако, если хочется сократить использование памяти, можно удалять узел, как только он оказывается листом, т.е. у него нет детей. В таком случае мы можем сделать вывод, что к элементу обращаются не слишком часто и мы не потеряем существенную часть статистики по нему, а главное сделать это можно очень быстро и дешево.

3.2.4. Создание новых узлов

В связи с тем, что мы работаем не с одним запросом, а с целым набором сразу, на практике иногда оказывается, что мы пришли в лист, а ненайденных узлов несколько. В таком случае, самым оптимальным способом создания и подвешивания сразу нескольких новых узлов является организация их в обычное сбалансированное дерево, корень которого мы и подвешиваем к последнему листу основного дерева.

Глава 4

Анализ

Существуют разные способы анализа производительности и эффективности мультипоточных программ [1]. В этой работе мы рассмотрим теоретический фреймворк — work и span. Для их определения, введем сначала такое понятие как directed acyclic graph (dag).

Dag — это одна из форм представления мультипоточных вычислений, представляющая собой направленный ацикличный граф. В нем каждая вершина представляет выполнение какой либо инструкции, например сложение, работа с памятью, рождение потока или синхронизирующую операцию. Вершина, являющаяся представлением операции рождения нового потока, имеет степень исхода два. Синхронизирующая операция же имеет степень входа два и дожидается пока все предшествующие ей операции в обоих потоках завершатся.

Теперь мы можем ввести work и span. Итак, work — это число вершин в dag представлении. Span — это размер самого длинного пути внутри графа.

Для оценки нам понадобятся несколько утверждений. Для начала рассмотрим случай, когда выполняется лишь один запрос одновременно.

Лемма 1. *Пусть* Φ *и* Φ' *потенциалы дерева до и после поворота соответственно. Вершина* z *является текущим корнем, а вершина* x *ее внуком (рис.4). Тогда справедлива следующая оценка:*

$$2 + \Delta\Phi \leq 2(r(z) - r(x))$$

Используя эту лемму, можно показать [2], что амортизированное время, затрачиваемое на повороты для узла v в дереве, с корнем $root$ оценивается как $\mathcal{O}(r(root) - r(v)) = \mathcal{O}(\log(W(root)/W(v)) + 1)$. Таким образом,

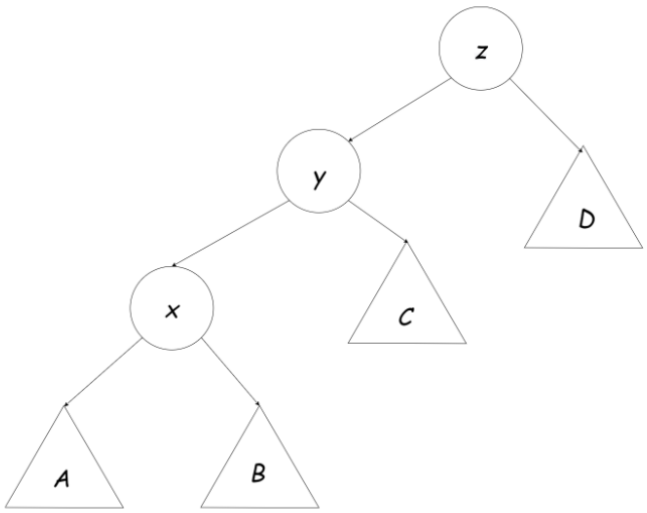

Рис. 4.1. Балансировка при помощи поворотов

для последовательности операций длиной m время выполнения запросов будет $\mathcal{O}(m + \sum_{i=1}^{n} c(v_i) \log \frac{m}{c(v_i)})$.

Теорема 1. *Серия из m поворотов в дереве, состоящем из n элементов $v_1, \cdots, v_n$, где каждый элемент элемент был запрошен хотя бы раз (имеет вес хотя бы 1) выполняется за время*

$$O\left(m + \sum_{i=1}^{n} c(v_i) \log \frac{m}{c(v_i)}\right),$$

где $c(v)$ — число запросов к элементу v.

Теперь, мы можем доказать [2], что длина пути, для достижения узла v в дереве, с узлами имеющими веса $W(u)$, составляет $\mathcal{O}(\log(W/C(v))$, где $W = W(root)$ — вес корня всего дерева.

Лемма 2. *Рассмотрим дерево, с весами $W(u)$ для каждого узла u. Длина пути, которую необходимо пройти для выполнения операции надо элементом v тогда $O(\log(\frac{W}{C(v)}) + 1)$, где $W = W(root)$, а $C(v)$ вес вершины v.*

Теперь вернемся к параллельному применению набора операций.

4.1. Span

Мы доказали, что глубина, на которой находится элемент после некоторой серии операций зависит от его веса. Осталось учесть затраты на управление набором операций. При спуске в каждый узел, мы разделяем массив операций на две части: на ту, ключи которой лежат в левом поддереве и ключи которой лежат в правом поддереве. На входе у нас есть отсортированный по ключу массив действий и мы можем воспользоваться бинарным поиском, чтобы найти позицию элемента, который делит этот массив на нужные нам части. Таким образом сделать это можно используя, например, std::partition за $\mathcal{O}(\log(m))$. Затем эти операции выполняются параллельно. Таким образом, итоговый span выполнения набора операций размера m в дереве размером m составляет $\mathcal{O}(\log(m)\log(\frac{W}{C(v)}))$.

Теорема 2. *Пусть на вход алгоритму подается набор из m операций с уникальными ключами. Итоговый span их применения составляет $\mathcal{O}\left(\log(m)\log(\frac{W}{C}\right)$, где $W = W(root)$ – вес текущего корня, а $C = \min\limits_{v\ in\ actions} C(v)$ – наименьший вес вершины среди вершин из набора.*

4.2. Work

Оценим work. Мы знаем, что каждая целевая вершина v находится на глубине $\mathcal{O}(\log(\frac{W}{C(v)}))$. При этом при каждом этапе происходит разделение пришедшего в эту вершину массива запросов на два подмассива: один, в котором элементы меньше, чем текущий корень, в другом – больше. Сделать это можно обычным проходом по массиву за $O(m)$, где m – длина текущего массива. При этом на каждом шаге массив делится пополам. Таким образом на глубине k будет массив длиной $\frac{m}{k}$. Для оценки числа вершин, будем считать, что граф представляет из себя максимально сбалансирован-

ное двоичное дерево глубины $p = \log(\frac{W}{C(v)})$. Таким образом

$$work = \mathcal{O}(\sum_{k=1}^{p} k\frac{m}{k}) = \mathcal{O}(pm)) = \mathcal{O}(m\log(\frac{W}{C(v)}))$$

.

Заметим, что, таким образом, наше дерево является статически оптимальным, то есть с точностью до константы оно работает не хуже чем самое оптимальное бинарное дерево, которое можно построить на этих элементах.

Также из асимптотики видно, что с увеличением частоты обращения к элементу $(C(v))$ объем необходимой работы для доступа к нему падает, чего мы и хотели добиться.

4.3. Корректность

Для начала рассмотрим случай, когда размер набора операций $m = 1$. В таком случае наше дерево ведет себя как обычное самоподстраивающееся бинарное дерево поиска. Основным отличием является лишь то, что балансировка происходит основываясь не на глубине дерева, а на весах узлов. Таким образом корректность в данном случае наследуется от обычного дерева поиска.

Теперь перейдем к случаю, когда $m \geq 1$. Основное отличие от предыдущего случая состоит в том, что мы теперь работаем с массивом операций, а не с одной.

На каждом шаге мы ищем, есть ли среди этого набора операция с ключом равным текущему корню. Если таковая имеется, то мы поиск для нее завершается. Затем мы разделяем массив операций на два подмассива, в первом из которых находятся только элементы меньше чем в текущем корне, а во втором больше чемв текущем корне и приминяем полученные наборы к левому и правому поддереву. В силу структуры бинарного дерева

поиска данные элементы могут находиться только в этих поддеревьях, а значит, если они там есть, мы их обязательно найдем. Если же их нет, то создаются новые вершины (или целое поддерево), удовлетворяющие свойствам дерева поиска, следовательно в будущих операциях мы их сможем найти.

Корректность поиска тем самым доказана.

Глава 5

Результаты

В этой главе будут описаны результаты проделанной работы и приведено сравнение разработанной структуры данных с ее аналогами.

5.1. Производительность

Тестирование проиводилось на сервере с 16-ядерным процессором Intel Xeon Processor 2.1 GHz и установленными 32 Gb оперативной памяти. На сервере установлена операционная система Ubuntu 20.04.1 LTS. Алгоритм реализован на языке `C++`. В реализации использовался алгоритм fork-join из библиотеки PCTL. Компиляция происходила при использовании OpenCilk 1.0, так как данный компилятор наиболее эффективен для параллельных вычислений.

Для оценки проихводительности было проведено несколько экспериментов. Заметим, что равномерная функция распределения нам не подходит, ведь преимущества самоподстраивающихся структур данных раскрываются когда есть элементы с повышенной частотой запросов к ним. Было установлено, что алгоритм показывает наибольшую эффективность на симметричных функциях распределения. Это не удивительно, ведь в таком случае на каждом шаге массив с запросами будет делиться примерно пополам, что помогает распределить нагрузку между процессорами. Таким образом для моделирования было выбрано нормальное распределение с нулевым матожиданием и различной дисперсией.

На графике 5.1 отображена зависимость времени применения набора операций от размера дерева. Поистроение происходило при фиксированном m, многократным применением сгенерированных из одного распределения наборов операций к одному дереву. Стоит отметить, что полученный гра-

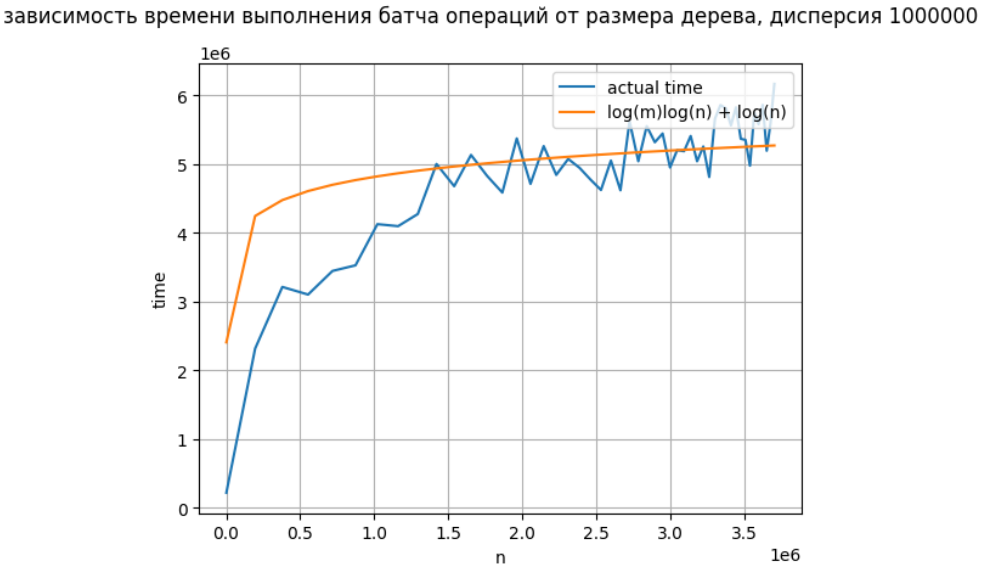

Рис. 5.1. Зависимость времени применения набора операций от размера дерева

фик довольно хорошо ложится на теоретически рассчитанную асимптотику

$\mathcal{O}(\log(m)\log(n) + \log(n))$.

Следующим этапом проверки производительности была проверка масштабируемости путем увеличения количества процессоров.

Рис. 5.2. Зависимость времени выполнения набора операций от размера дерева для разных наборов процессоров

На графике 5.2 предоставленна зависимость времени выполнения на-

бора операций от размера дерева для разного количества процессоров. На графике видно, что с ростом числа процессоров время выполнения запросов уменьшается. Более отчетливо это видно на графике 5.2, где показана зависимость времени выполнения фиксированного запроса от количества процессоров. Так например при переходе от 1 процессора к 12 наблюдается ускорение почти в 9 раз.

Рис. 5.3. Зависимость времени выполнения набора операций от числа потоков

Далее мы сравним производительность для разных распределений. В этом эксперименте данные генерировались используя нормальное распределение с различными значениями дисперсии.

На графике 5.4 показана зависимость времени выполнения набора операций для значений дисперсии $10^5, 10^6$, и 10^7.

Можно заметить, что чем меньше дисперсия, тем быстрее выполняются запросы, при одинаковом размере дерева. Это можно объяснить тем, что хоть и количество элементов в дереве одинаковое, при наполнении запросы к определенным элементам были намного чаще, а значит они находятся куда выше по дереву, что доказывает эффективность самоподстраивающейся структуры данных в таких случаях.

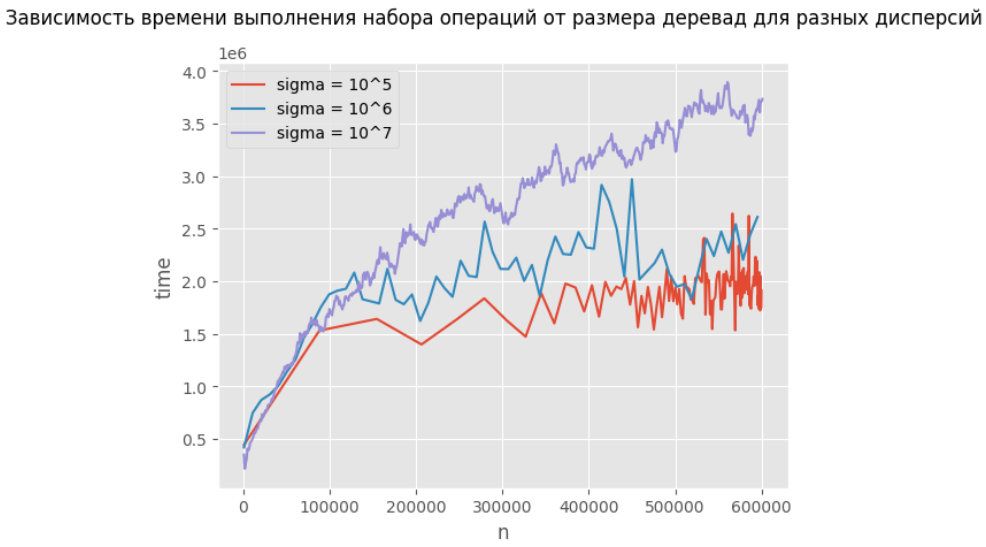

Рис. 5.4. Зависимость времени выполнения набора операций от характера входных данных.

5.2. Сравнение с аналогами

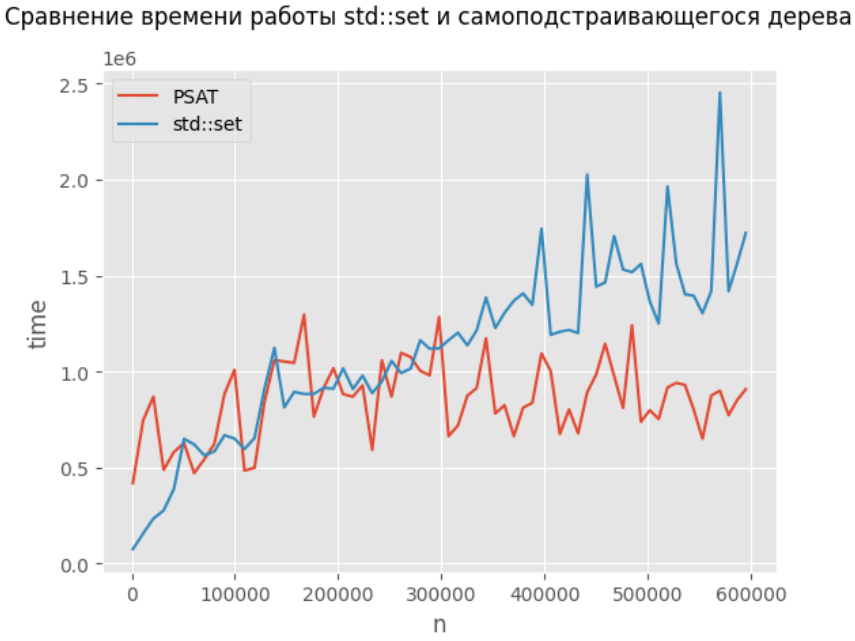

Рис. 5.5. Сравнение времени работы std::set и PSAT

Для сравнения был выбран std::set из стандартной библиотеки C++. На графике 5.5 представлена зависимость времени работы std::set и параллельного самоподстраивающегося дерева. Как можно заметить, при малых значениях n std::set выигрывает в производительности, однако после доста-

точного набора статистики PSATree выходит на лучшую асимптотику.

Заключение

В работе были описаны базовые принципы самоподстраивающихся структур данных, а также рассмотрены популярные представители этого класса алгоритмов.

Был разработан и представлен алгоритм, работающий сразу с набором операций параллельно, за счет чего он более оптимален для работы с данным типом задач чем однопоточные и конкурентные аналоги. Также алгоритм обладает свойством статической оптимальности. Было показано, что асимптотика выполнения набора операций $\mathcal{O}(\log(m)\log(\frac{W}{C(v)}))$, тем самым обеспечивается более быстрый доступ как наиболее частозапрашиваемым элементам, что подтверждается в экспериментах.

Исходя из результатов сравнения можно сделать вывод, что полученная реализация ведет ведет себя эффективнее стандартных аналогов, а также неплохо масштабируется на количества потоков с ускорением до 10 раз для 12 процессоров.

Список литературы

[1] Umut A. Acar. *Parallel Computing: Theory and Practice*. 2016. URL: http://www.cs.cmu.edu/afs/cs/academic/class/15210-f15/www/tapp.html.

[2] Yehuda Afek, Haim Kaplan, Boris Korenfeld, Adam Morrison, Robert E. Tarjan. «CBTree: A Practical Concurrent Self-Adjusting Search Tree». *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, с. 1—15. DOI: 10.1007/978-3-642-33651-5_1. URL: https://doi.org/10.1007/978-3-642-33651-5_1.

[3] Vitaly Aksenov, Dan Alistarh, Alexandra Drozdova, Amirkeivan Mohtashami. *The Splay-List: A Distribution-Adaptive Concurrent Skip-List*. 2020. eprint: arXiv:2008.01009.

[4] Dion Harmon. *New Bounds on Optimal Binary Search Trees*. 2000. URL: https://dspace.mit.edu/bitstream/handle/1721.1/34268/71014527-MIT.pdf?sequence=2.

[5] Daniel Dominic Sleator, Robert Endre Tarjan. «Self-adjusting binary search trees». *Journal of the ACM* **32** 3 (июль 1985), с. 652—686. DOI: 10.1145/3828.3835. URL: https://doi.org/10.1145/3828.3835.