Аннотация

Алгоритмы параллельной вставки и удаления для сбалансированных деревьев поиска с оптимальной асимптотикой уже известны. Но эти алгоритмы не применимы для персистентных двоичных деревьев поиск. Целью этой работы является разработка методов параллельной вставки и удаления, сохраняющих свойство персистентности. В этой выпускной квалификационной работе предлагаются два новых метода для персистентных сбалансированных деревьев поиска: insertAll и removeAll, позволяющих вставлять и удалять элементы параллельно. Метод insertAll принимает на вход массив элементов, которые нужно вставить, и возвращает массив персистентных деревьев. Этот массив содержит все деревья, которые пользователи могли получить, последовательно вызывая метод insert для всех элементов входного массива в отсортированном порядке. Аналогично для метода removeAll, который, наоборот, удаляет элементы. В этой работе демонстрируется псевдокод методов insertAll и removeAll, доказывается их корректность, считается их асимптотическая сложность и проводятся эксперименты над ними. В этой работе методы insertAll и removeAll описаны для АВЛ, красно-черных и декартовых деревьев. Для наших методов insertAll и removeAll получена асимптотика: $O(\log(n) \cdot \log(m))$ span, и $O(m \cdot \log(n))$ work, где n - число вершин в исходном дереве, а m - число операций над деревом. Также в работе показаны результаты эксперимента, по которым видно, что алгоритмы хорошо распараллеливается.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Персистентные структуры данных имеют множество различных прикладных применений. Например, персистентные структуры данных используются в функциональном программировании, также для оптимизации React приложений и в различных геометрических задачах. Одной из самых популярных структур данных является двоичное дерево поиска, так как его операции имеют логарифмическую от размера асимптотику.

Алгоритмы параллельной вставки и удаления для сбалансированных деревьев поиска с оптимальной асимптотикой work были уже предложены в “ Fast parallel operations on search trees ” [6]. Но эти алгоритмы напрямую не применимы для персистентных двоичных деревьев поиска. В этой работе мы предлагаем алгоритмы, которые позволяют эффективно вставлять или удалять множество объектов в персистентные сбалансированные деревья поиска. В частности, в работе рассматриваются красно-черные, АВЛ и декартовы деревья.

Целью работы является разработка параллельного метода insertAll для персистентных сбалансированных деревьев поиска, который принимает на вход массив элементов и возвращает массив деревьев, который должен совпадать с массивом версиями деревьев, если бы мы последовательно вставляли элементы массива в персистентное дерево. Аналогично работает метода deleteAll: он принимает на вход массив элементов для удаления. Также важными результатами работы является асимптотическая оценка эффективности предложенных алгоритмов, проведение экспериментов, позволяющих понять зависимость времени работы алгоритмов от числа процессов, и проведение сравнительного анализа с обычной последовательной вставкой и удалением элементов в персистентное дерево.

Хорошим примером применения может служить Point location problem. Point location problem - задача о местоположении точки. В самом общем виде

задача состоит в том, чтобы, учитывая разбиение пространства на непересекающиеся области, определить область, в которой находится точка запроса. Подробнее от этой задаче и её приложениях написано в “Point location” [5]. Задачи такого рода решаются в двух вариациях: offline и online. В offline-версиях все запросы даны заранее и можно обрабатывать их одновременно. В online-задачах следующий запрос можно узнать только после того, как найден ответ на предыдущий.

При решении offline-задачи предоставленный плоский многоугольник разбивается на полосы вертикальными прямыми, проходящими через вершины многоугольников, которые являются разбиением исходного многоугольника. Затем в этих полосах при помощи техники заметающей прямой ищется местоположение точки-запроса. При переходе из одной полосы в другую изменяется сбалансированное дерево поиска. Если использовать персистентную структуру данных, то для каждой полосы будет своя версия дерева и сохранится возможность делать к ней запросы.
Тогда Point location problem может быть решена в online. Алгоритм insertAll может быть в данной задаче очень полезен при online решении, так как с помощью него можно эффективно построить деревья для каждой полосы.

ГЛАВА 1. ФОРМУЛИРОВКИ И ОПРЕДЕЛЕНИЯ

Двоичное дерево — иерархическая структура данных, в которой каждый узел имеет не более двух потомков (детей). Как правило, первый называется родительским узлом, а дети называются левым и правым наследниками.

Двоичное дерево поиска — двоичное дерево, в каждой вершине которого содержится ключ и для которого выполняются следующие дополнительные условия, называемые *свойствами дерева поиска*):

1) оба поддерева — левое и правое — являются двоичными деревьями поиска;

2) у всех узлов *левого* поддерева произвольного узла X значения ключей данных *меньше*, чем значение ключа самого узла X;
3) у всех узлов *правого* поддерева произвольного узла X значения ключей данных *больше либо равны*, чем значение ключа самого узла X.

АВЛ-дерево — сбалансированное по высоте двоичное дерево поиска: для каждой его вершины высота её двух поддеревьев различается не более чем на единицу.

Красно-чёрное дерево — двоичное дерево поиска, в котором каждый узел имеет атрибут *цвета*. При этом

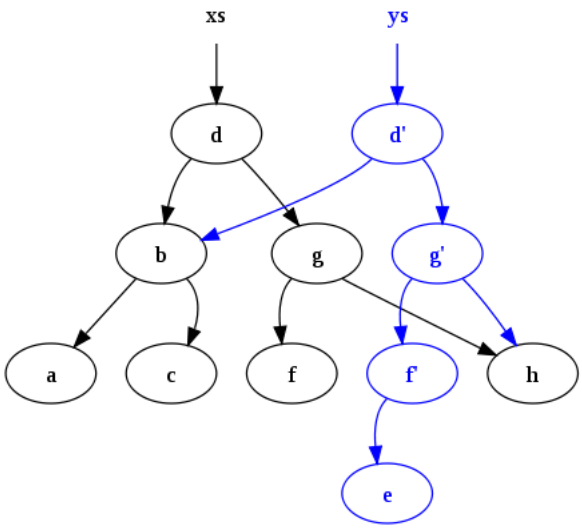
1) Узел может быть либо красным, либо черным и имеет двух потомков;
2) Корень — как правило чёрный. Это правило слабо влияет на сложность алгоритма, так как цвет корня всегда можно изменить с красного на чёрный. В этой работе для простоты не будем следовать этому ограничению.
3) Все листья, не содержащие данных — чёрные.
4) Оба потомка каждого красного узла — черные.
5) Любой простой путь от узла-предка до листового узла-потомка содержит одинаковое число черных узлов.

В дальнейшем под “деревом” будут подразумеваться двоичные деревья поиска.

Персистентные структуры данных — это структуры данных, которые при внесении в них каких-то изменений сохраняют все свои предыдущие состояния и доступ к этим состояниям.

Самое очевидный способ сделать структуру данных персистентной – это копировать исходную структуру каждый раз, когда необходимо выполнить операцию по ее изменению (insert/delete и т.д.). Но есть и более оптимальные решения. Рассмотрим вставку в двоичное дерево поиска. Пусть нам дано сбалансированное дерево поиска. Все операции в нем делаются за O(h), где h — высота дерева, а высота сбалансированного дерева поиска

асимптотически равно O(log(n)), где n — количество вершин. Пусть необходимо сделать какое-то обновление в этом сбалансированном дереве, например, добавить очередной элемент, но при этом нужно не потерять старое дерево. Возьмем узел, в который нужно добавить нового ребенка. Вместо того чтобы добавлять нового ребенка, скопируем этот узел, к копии добавим нового ребенка, также скопируем все узлы вплоть до корня, из которых достижим первый скопированный узел вместе со всеми указателями. Все вершины, из которых измененный узел недостижим, мы не трогаем. Число новых узлов всегда будет равно высоте дерева, то есть будет порядка логарифма. В результате мы имеем доступ к обеим версиям дерева: старой и новой. На изображении 1 приведен пример двух деревьев: черным цветом нарисована старая версия дерева и синим цветом нарисована новая версия дерева, ссылающаяся на узлы старого дерева, со вставленным элементом “e”.



(Рис. 1 - Пример добавления в персистентное дерево)

ГЛАВА 2. ВСПОМОГАТЕЛЬНЫЕ МЕТОДЫ И СТРУКТУРЫ

В этом разделе мы изучим вспомогательные методы и объекты, которые будут использованы в описании методов insertAll и deleteAll. В коде этих методов в качестве EMPTY_TREE будем полагать дерево, не содержащее

вершин. Также в коде будет использоваться метод fork2{//код1} and {//код2}, который запускает код 1 и код 2 в двух разных потоках.

2.1. Методы двоичных деревьев поиска

Метод join принимает на вход вершину item, бинарное дерево поиска tl, содержащее элементы, которые меньше или равны item, и дерево tr, содержащее элементы больше, чем item и возвращает дерево, содержащее все вершины tl, tr и вершину item. Сложность и псевдокод данного метода будет дан позже. Данный метод будет различаться для различных сбалансированных деревьев.

Метод splitLast принимает дерево tree и возвращает пару (newTree, x), где x - наибольшая вершина дерева tree, а newTree - дерево содержащее все вершины исходного, кроме наибольшей вершины

Псевдокод:

```
1: (Tree, Item) splitLast(Tree tree) {
2:   (L, k, R) = expose(tree)
3:   if  (R.isLeaf()) {
4:     return (L, k)
5:   } else {
6:     (T', k') = splitLast(R)
7:     return (join(L, k, T'), k')
8:   }
9: }
```

Метод join2 принимает на вход дерево tl, все вершины которого меньше или равны всех вершин tr и дерево tr, все вершины которого больше всех вершин tl и возвращает дерево, содержащее все вершины TL, TR. Сложность данного метода O(h(tl) + h(tr)). Доказательство сложности будет приведено в тексте позднее.

Псевдокод:

```
Tree join2(Tree tl, Tree tr){
  (t, x) = splitLast(tl)
  return join(t, x, tr)
}
```

Метод expose принимает на вход дерево t и возвращает кортеж (tl, x, tr), где tl – левое поддерево, tr – правое поддерево, а x - значение в корне дерева. Сложность данного метода O(1)

Псевдокод:

```
(Tree, Item, Tree) expose(Tree t) {
  return (t.tl, t.value, t.tr)
}
```

Метод split принимает на вход вершину x и дерево t, а возвращает кортеж `(tl, flag, tr)`, где tl дерево, которое содержит все вершины дерева t, которые меньше чем x, дерево tr, которое содержит все вершины дерева t, которые больше, чем x, а flag – это булевая переменная, которая равна true, если x содержался в исходном дереве, и false, если не содержался.

Псевдокод:

```
(Tree, Tree) split(Tree t, Item x){
  if (t = Leaf) {
    return (Leaf, false, Leaf)
  } else {
    (L, m, R) = expose(t);
    if (x = m) {
        return (L, true, R)
    } else {
        if (x < m) {
            (Ll, b, Lr) = split(L, x);
            return (Ll, b, join(Lr , m, R))
        } else {
            (Rl, b, Rr) = split(R, x);
            return (join(L, m, Rl), b, Rr)
        }
    }
  }
}
```

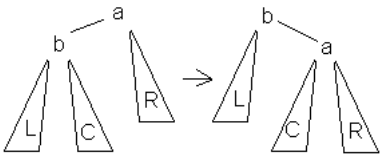
Метод h(Tree t) возвращает высоту дерева t. Его реализация тривиальна. А сложность равна O(n), где n – число вершин в дерева.

Метод k(Tree t) возвращает значение в корне дерева t. Его реализация тривиальна. А сложность равна O(1).

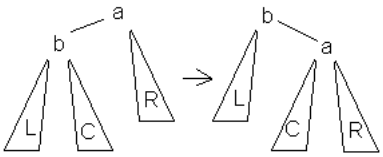
Методы R(Tree t) и L(Tree t) возвращают правое и левое поддерево соответственно. Их реализация тривиальна. А сложность равна O(1).

2.1.1 Методы для АВЛ-деревьев

Методы rotateLeft и rotateRight – это левый и правый поворот дерева соответственно в АВЛ-дереве. Их сложность O(1).



(Рис. 2 - Малое левое вращение.)



(Рис. 3 - Малое правое вращение.)

2.1.2 Методы для декартовых деревьев

Метод prior возвращает приоритет вершины для декартового дерева. Его реализация тривиальна. А сложность равна O(1).

2.1.3 Методы для красно-черных деревьев

Метод $\widehat{h}$(Tree t) возвращает черную высоту красно-черного дерева t. Его реализация тривиальна А сложность равна O(n), где n – число вершин в дерева.

Метод c(Tree t) возвращает цвет вершины для красно-черного дерева. Его реализация тривиальна. А сложность равна O(1).

2.2 Вспомогательные структуры данных

SplitNode – это следующая структура:

```
SplitNode<T> {
   T item;
   Tree newTl;
   Tree tr;
   Tree outerTl;
   Tree outerTr;
}
```

В дальнейшем эта структура будем необходима в алгоритмах insertAll и deleteAll для построения дерева сплитов.

ГЛАВА 3. МЕТОД JOIN

В предыдущей главе был описан метод Join. Он довольно сложен, но в то же время довольно важен для описываемых в этой работе методов insertAll и deleteAll. Поэтому он внесен в отдельную главу. В данной главе мы рассмотрим его поподробнее. Стоит заметить, что только этот метод различается для разных типов деревьев. Все остальные методы, предложенные в предыдущей главе, никак не поменяются.

3.1 Псевдокод метода join

Давайте рассмотрим псевдокод метода join для каждого из типов сбалансированного дерева, рассматриваемых в этой работе.

3.1.1 Метод join для АВЛ-деревьев

```
1: Tree joinRight(Tree tl, Item k, Tree tr) {
2:     (l ,k’,c) = expose(tl);
3:     if (h(c) <= h(tl) + 1) {
4:        T’ = new Node(c, k, tr);
5:        if (h(T’) <= h(l) +1) {
6:           return new Node(l, k’,T’);
7:        } else {
8:           return rotateLeft(
9:              new Node(l, k’, rotateRight(T’))
10:          );
11:       }
12:    } else {
13:       T’ = joinRight(c, k, tr);
14:       T” = new Node(l, k’, T’);
15:       if (h(T’) <h(l) +1) {
16:           return T”;
17:       } else {
18:           return rotateLeft(T”);
19:       }
20:    }
21: }
```

```
Tree join(Tree tl, Item k, Tree tr) {
    if (h(tr) > h(tr) + 1) {
        return joinRight(tl, k, tr);
    } else {
        if (h(tr) > h(tl) + n1) {
            return joinLeft(tl, k, tr);
        } else {
            return new Node(tl, k, tr);
        }
    }
}
```

3.1.2 Метод join для декартовых деревьев

```
Tree join(Tree tl, Item k, Tree tr) {
   (l1, k1, r1) = expose(tl);
   (l2, k2, r2) = expose(tr);
   if (prior(k, k1) and prior(k, k2)) {
      return Node(tl, k, tr);
   } else {
      if (prior(k1, k2)) {
         return new Node(l1, k1, join(r1, k, tr))
      } else {
         return new Node(join(T, k, l2), k2, r2)
      }
   }
}
```

3.1.3 Метод join для красно-черных деревьев

```
1:  Tree joinRightRB(Tree tl, Item k, Tree tr) {
2:    if (ĥ(tl) == ĥ(tr) and c(tl) == black) {
3:      return Node(tl, k, red, tr);
4:    } else {
5:      (L', k', c', R') = expose(tl);
6:      T' = new Node(
7:        L', k', c', joinRightRB(R', k, tr)
8:      );
9:      if (c' == black and c(R(T')) == c(R(R(T')))
10:         and c(R(T')) == red) {
11:       c(R(R(T'))) = black;
12:     }
13:       T" = rotateLeft(T');
14:       return T";
15:     }
16: }
```

```
1:  Tree join(Tree tl, Item k, Tree tr) {
2:    if (ĥ(tl) > ĥ (tr)) {
3:      T′ = joinRightRB(tl, k, tr);
4:      if (c(T′) = red and c(R(T′)) = red) {
5:        return new Node(L(T), k(T′), black, R(T′))
6:      } else {
7:        return T′
8:      }
9:    } else {
10:     if (ĥ(tr) > ĥ(tr)) {
11:       T′ =joinLeftRB(tl, k, tr);
12:       if (c(T′) == red and c(L(T′)) == red) {
13:          return new Node(
14:             L(T), k(T′), black, R(T′)
15:          );
16:       } else {
17:          return T′;
18:       }
19:     } else {
20:       if (c(tl) == black and c(tr) == black) {
21:         return new Node(tl, k, red, tr)
22:       } else {
23:         return new Node(tl, k, black, tr)
24:       }
25:     }
26:   }
27: }
```

3.2 Сложность и корректность метода join

В этой секции мы рассматриваем сложность и доказываем корректность метод**а** join для каждого из типов сбалансированного дерева, рассматриваемых в этой работе.

3.2.1 Сложность и корректность метода join для АВЛ-деревьев

Давайте рассмотрим код метода join для АВЛ-дерева, который был предоставлен ранее. Если высоты деревьев tl и tr различаются не больше чем на единицу, то необходимо только создать один новый узел Node(tl, k, tr). Если разница высот больше единицы, то, без ограничения общности, предположим, что h(tl) > h(tr) + 1. Случай h(tr) > h(tl) + 1 доказывается аналогично. Идея алгоритма заключается в том, что мы спускаемся по правым поддеревьям вниз, до тех пор пока h(c) > h(tr) + 1. Дальше мы создаем новый узел Node(c, k, tr). Заметим, что новый узел удовлетворяет инварианту АВЛ-дерева, так как h(c) = h(tr) или же h(c) = h(tr) + 1. Увеличение высоты полученного поддерева может сломать инвариант АВЛ-дерева для вершин родителей, поэтому мы делаем малые левые и правые повороты, восстанавливая высоту для любых дальнейших узлов предков. Таких поворотов совершается не больше двух. Поскольку алгоритм посещает на пути только узлы на пути от корня до c, то время работы пропорционально разнице высот tl и tr. В итоге мы получаем время работы O(|h(tl) – h(tr)|).

3.2.2 Сложность и корректность метода join для красно-черного дерева.

Рассуждения для красно-черного дерева похожи на рассуждения для АВЛ-дерева. В случае, когда черная высота tl равна tr, мы просто создаем новый узел Node(tl, k, tr) и красим его в соответствующий цвет, что делает за O(1). Рассмотрим случай, когда черная высота tl больше черной высоты tr. Обратный случай будет симметричен. Алгоритм идет по правым дочерним узлам tl до тех пор, пока черная высота поддерева не станет равно черной высоте tr и при этом корень поддерева будет черный. На место найденной

вершины вставляется новый узел красного цвета. Возможно, это нарушит инвариант красно-черного дерева. В худшем случае у нас может быть три красных узла подряд. Это фиксируется одним левым поворотом. Если черный узел v, а узлы R(v) и R(R(v)) оба красные, то мы перекрашиваем R(R(v)) в черный и выполняем один левый поворот по v. Вращение, однако, может снова нарушить правило красного, что может потребовать еще поворотов до тех пор, пока не дойдем до корня tl. Если исходный корень красного цвета, алгоритм может в конечном итоге получить красный корень с красным дочерним элементом, и в этом случае перекрасим корень в красный. Легко заметить, что время работы пропорционально разности высот деревьев. В итоге мы получаем время работы O(|h(tl) – h(tr)|).

3.2.3 Сложность и корректность метода join для декартового дерева.

Давайте рассмотрим код метода join, для декартового дерева, который был предоставлен ранее. Напомню, что в декартовом дереве для всех вершин известен их приоритет, который для произвольной вершины X обозначим как X.prior. Join для декартового дерева рекурсивно вызывается до тех пор пока tl.prior и tr.prior не меньше чем k.prior. Из этого следует, что время работы O(log(t)), где t – количество вершин, у которых приоритет больше k.

ГЛАВА 4. СЛОЖНОСТЬ ВСПОМОГАТЕЛЬНЫХ МЕТОДОВ

В прошлой главе была дана оценка метода join. Часть вспомогательных методов, упомянутых в главе 2, используют данный метод. В этой главе будет доказана заявленная ранее для них асимптотическая сложность.

4.1 Асимптотическая сложность метода split

Пусть глубина рекурсии выполнении метода split оказалось равной k. Обозначим левый результат split полученный на глубине i как t_i. Тогда у нас

k деревьев: t_k, t_{k-1}, … t_1. В процессе работы алгоритма мы по очереди выполняем join для этих деревьев. То есть

$$T_{k-1} = join(t_k, m_{k-1}, t_{k-1}),$$
$$T_{k-2} = join(t_{k-2}, m_{k-2}, T_{k-1})$$

и так далее k-1 раз. Для правых результатов split всё аналогично.

Для АВЛ и красно-черных деревьев не трудно убедиться, что

$$h(T_{i-1}) \leq h(t_{i-1}) + 1 \leq h(t_{i-2}).$$

Согласно доказанному ранее время работы одного join составляет $O(|h(t_i) - h(T_{i-1})|)$

Тогда суммарное время работы:

$$\sum_{i=o}^{k-1} |h(t_{k-i}) - h(T_{k-i-1})| \leq \sum_{i=0}^{k-1} h(t_{k-i}) - h(T_{k-i-1}) + 2 =$$
$$= O(h(T)) = O(log\ (T.size)),$$

где T – это исходное дерево.

Для декартового дерева join всегда в качества аргумента получает ключ с наивысшим приоритетом. Поэтому время работы каждого join составляет O(1), а работа split не более чем O(log(T.size))

4.2 Асимптотическая сложность метода join2

Давайте рассмотрим представленный ранее псевдокод метода join2. В процессе работы join2 мы спускаемся в левом поддереве вниз по правым поддеревьям. Спустившись вниз, мы поднимаемся на вверх выполняя операции join. Рассуждения на суммарную сложность работы всех join из метода splitLast аналогично рассуждению для метода split. После выполнения splitLast мы выполняем еще одну операцию join. В результате мы получаем сложность O(h(TL) + h(TR)).

ГЛАВА 5. МЕТОД insertAll

В этой главе мы полностью опишем метод insertAll, реализация которого является одной из целью этой работы. Также мы рассмотрим результат эксперимента по его запуску с разным числом процессов.

5.1 Описание метода insertAll

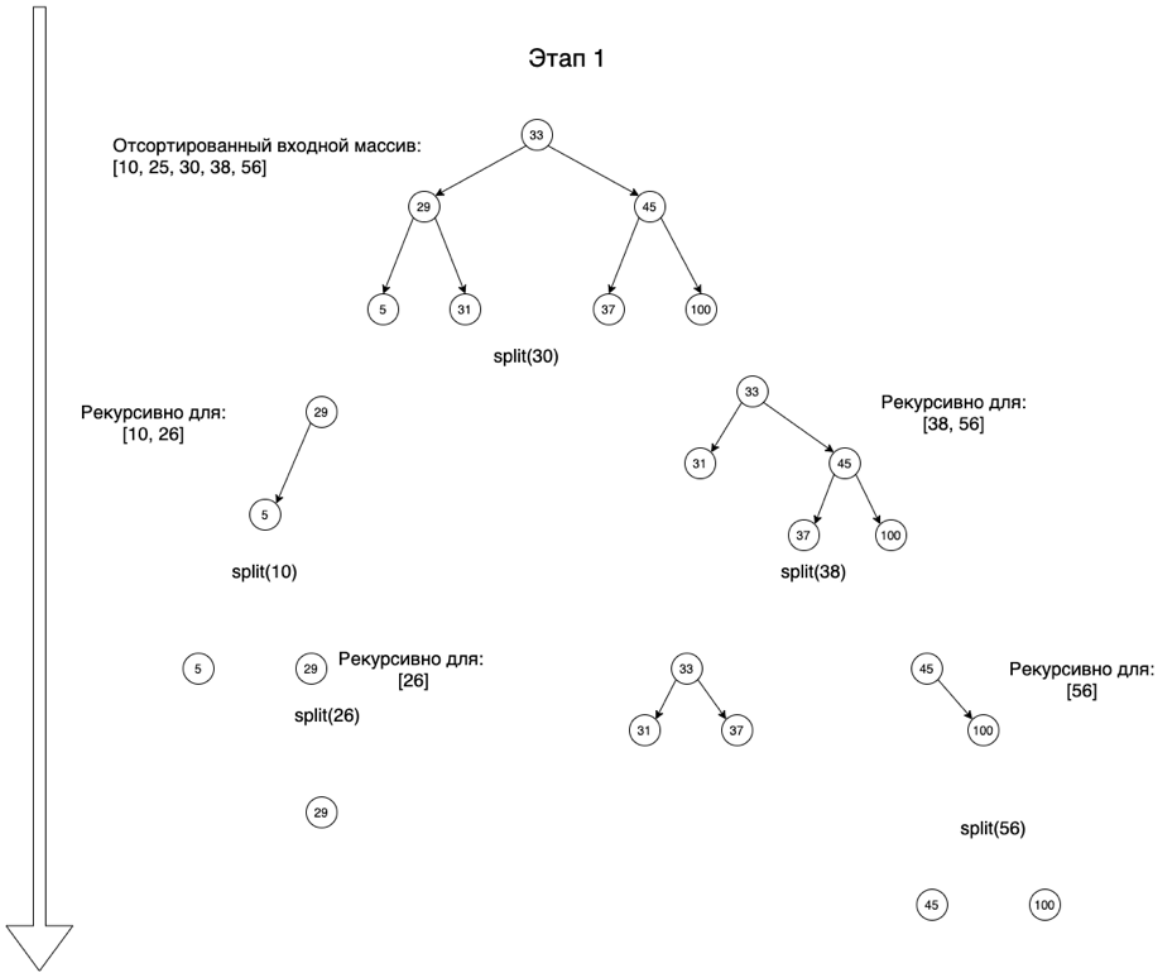
На вход методу insertAll подается массив элементов, которые нужно вставить. В качестве результата, операция должна вернуть массив деревьев, где на i-ой позиции содержится дерево, которое содержит все элементы исходного дерева и первые i элементов отсортированного входного массива. Полученный массив должен совпадать с массивом версиями деревьев, если бы мы последовательно вставляли элементы массива в персистентное дерево.

5.2 Описание алгоритма insertAll

Работу алгоритма можно разделить на несколько этапов.

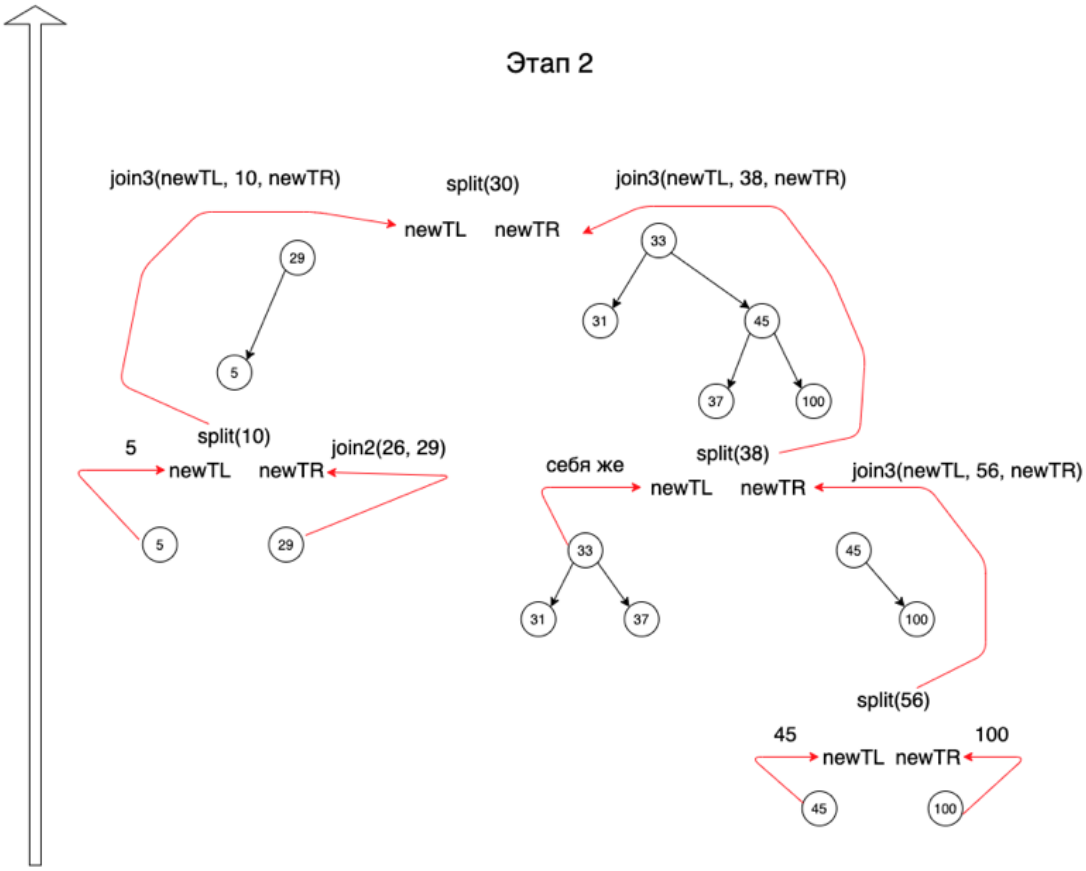
Этап 0. Предобработка. Во время предобработки отсортируем входной массив items. Создадим массив splitNodes, размер которого равен размеру массива items. Массив splitNodes будет хранить структуры SplitNode описанные ранее.

Этап 1. На этом этапе мы осуществляем спуск сверху вниз. Цель этого этапа заполнить поля item и tr для каждой SplitNode из массива splitNodes. Берем центральный элемент массива items (items[center], где center = items.size() / 2) и вызываем split исходного дерева по этому элементу. В результате выполнения метода split мы получаем два дерева tl и tr, где tl – это левый результат split, а tr правый результат split. В двух потоках повторяем рекурсивно первый этап для tl и tr, но в качестве items для tl передаем элементы меньше центрального, а для tr больше центрального. После каждого split для splitNodes[center] можем заполнить поле item, элемент по которому делали split, и поле tr, правый результат split.



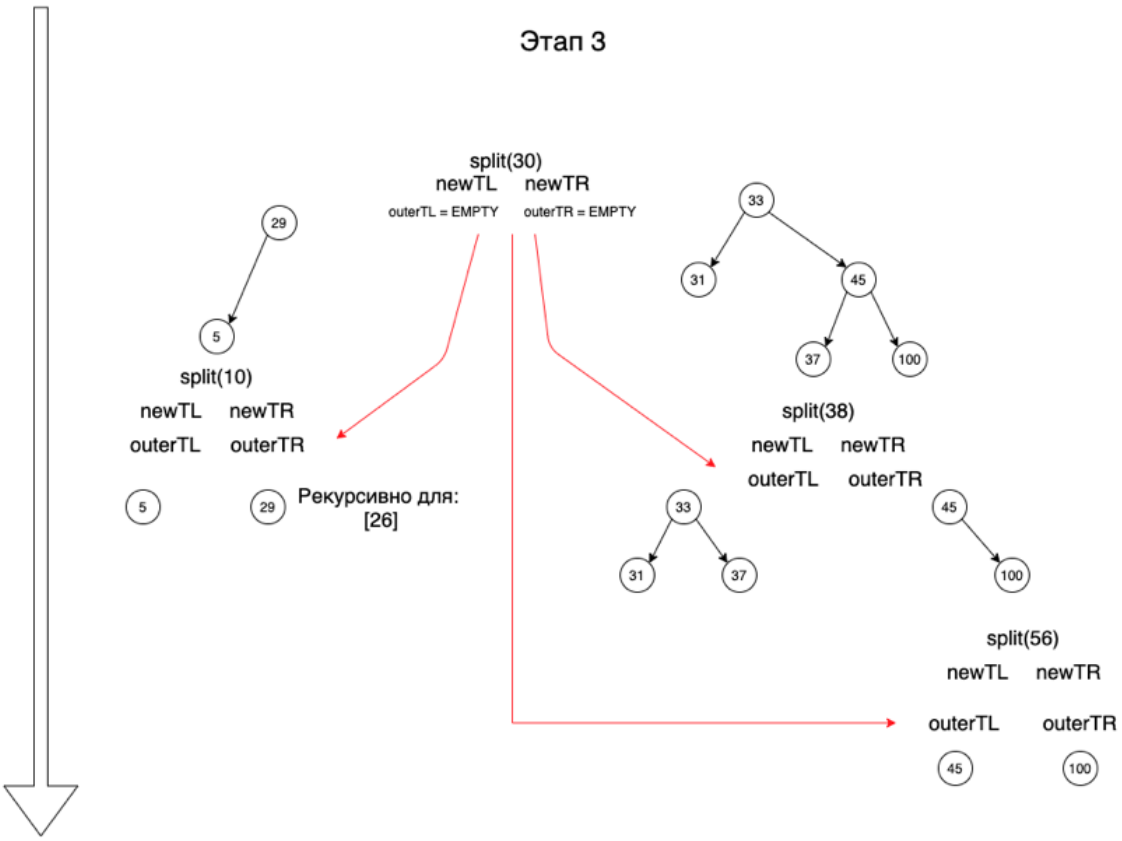
(Рис. 4 – Этап 1 insertAll)

Этап 2. На этом этапе мы осуществляем подъем снизу вверх. Цель этого этапа заполнить поле newTl для каждой SplitNode из массива splitNodes. newTl – это дерево, которое содержит все вершины tl и все элементы items, переданные на первом этапе при рекурсивном вызове. В реализации этот этап удобно объединять с первым этапом. Результатом вызова рекурсивной функции из первого этапа сделать деревья newTl и newTr. Для этого достаточно возвращать join(newTl, item, newTr).



(Рис. 5 – Этап 2 insertAll)

Этап 3. На этом этапе мы осуществляем спуск сверху вниз. Цель этого этапа заполнить поля outerTr и outerTl для каждой SplitNode из массива splitNodes. outerTl – это дерево содержащее все вершины левых результатов split, со вставленными новым элементами до выполнения split по текущему элементу. outerTr – это дерево содержащее все вершины правых результатов split до выполнения split по текущему элементу. Реализация похожа на первый этап. Рекурсивно для левой и правой части выполняем этот этап, тем самым распараллеливаясь. В качестве входных параметров два аргумента, которые будут outerTr и outerTl. Для левого ребенка передаем outerTl и join(tr, outerTr). Для правого ребенка join(outerTl, newTl).insert(item) и outerTr.



(Рис. 6 – Этап 3 insertAll)

Этап 4. Получение результата. Для получения результат необходимо просто пройтись по массиву splitNodes и для каждого элемента вызвать `join(join(outerTl, newTl), item, join(tr, outerTr))`. Так как outerTl – это дерево содержащее все вершины левых результатов split, со вставленными новым элементами до выполнения split по текущему элементу, а newTl – это дерево, которое содержит все вершины tl и все элементы items, переданные на первом этапе при рекурсивном вызове, то join(outerTl, newTl) – это дерево содержащее все элементы исходного дерева, которые меньше чем item, и содержащее все элементы вставляемого массива items, которые меньше чем item. Так как outerTr – это дерево содержащее все вершины правых результатов split до выполнения split по текущему элементу, а tr - это правый результат split по элементу item, то join(tr, outerTr) - это дерево содержащее все элементы исходного дерева, которые больше чем item. Тогда join(join(outerTl, newTl), item, join(tr, outerTr)) - это дерево

содержащее все элементы исходного дерева и содержащее все элементы массива items до элемента item включительно.

5.3 Псевдокод метода insertAll

```
1:  Array<Tree> insertAll(Tree tree, Array<T> items) {
2:     // 0 этап предобработка
3:     // отсортируем элементы
4:     items.sort();
5:     // создадим массив размера items.size());
6:     Array<SplitNode> splitNodes(items.size());
7:     // helper1 выполняет этапы 1 и 2 алгоритма
8:     helper1(tree, items, splitNodes);
9:     // helper2 выполняет этап 3 алгоритма
10:    helper2(splitNodes, EMPTY_TREE, EMPTY_TREE);
11:       // получение результата
12:    return splitNodes.map(node -> {
13:            join(
14:                join(node.outerTl, node.newTl),
15:                node.item,
16:                join(node.Tr, node.outerTr)
17:            );
18:    })
19: }
```

```
1:  // Рекурсивная функция соответствующая этапу 1 и 2
2:  Tree helper1(Tree tree, Array<T> items,
3:    Array<SplitNode> splitNodes) {
4:    Int center = items.size() / 2;
5:    (tl, flag, tr) = tree.split(items[center]);
6:    Tree newTl, newTr;
7:    fork2 {
8:      newTl = helper1(
9:          tl, items[0:center], splitNodes
10:     );
11:   } and {
12:     newTr = helper1(
13:         tl, items[center + 1 : items.size()],
14:         splitNodes
15:     );
16:   };
17:   splitNodes[center] = new SplitNode ();
18:   splitNodes[center].item = items[center];
19:   splitNodes[center].newTl = newTl;
20:   splitNodes[center].tr = tr;
21:   return join(newTl, items[center], newTr);
22: }
```

```
1:  // Рекурсивная функция соответствующая этапу 3
2:  Tree helper2(
3:      Array<SplitNode> splitNodes,
4:      Tree outerTl,
5:      Tree outerTr
6:  ) {
7:    Int center = splitNodes.size() / 2;
8:    splitNodes[center]. outerTl = outerTl;
9:    splitNodes[center]. outerTr = outerTr;
10:   Tree tr = splitNodes[center].tr
11:   Tree newTl = splitNodes[center].newTl;
12:   T item = splitNodes[center].item
13:   fork2 {
14:     helper2(
15:       splitNodes[0 : center], outerTl,
16:       join(tr, outerTr)
17:     );
18:   } and {
19:     helper2(
20:       splitNodes[center + 1 : splitNodes.size()],
21:      join(outerTl, newTl).insert(item),
22:       outerTr
23:     );
24:   };
25: }
```

5.4 Корректность алгоритма insertAll

Для доказательства корректности работы алгоритма покажем две вещи:

1) Все элемент массива splitNodes будут заполнены.

2) i-ым элементом массива, являющимся результатом выполнения insertAll, будет join(join(outerTl, newTl), item, join(Tr, outerTr)), где outerTl, newTl, item, Tr, outerTr это значения полей i-ого элемента массива splitNodes.

Первый пункт очевиден, так как каждая splitNodes[i] соответствует items[i]. На первом этапе мы для каждого элемента items делаем split, создаем SplitNode и заполняем tr и item. Остальные поля же заполняются, потому что мы ходим по одним и тем же вершинам, что и на первом этапе, на втором и третьем этапах.

Второй пункт более сложен для понимания. outerTl - это дерево, содержащее все вершины левых результатов split, со вставленными новым элементами, до выполнения split по текущему элементу. newTl – это дерево, которое содержит все вершины tl и все элементы items переданный на первом этапе при рекурсивном вызове. Тогда join(outerTl, newTl) – это дерево содержащее все элементы исходного дерева, которые меньше чем item, и все элементы items, которые меньше чем item. outerTr – это дерево содержащее все вершины правых результатов split до выполнения split по текущему элементу. Тогда join(Tr, outerTr) – это дерево, которое содержит все вершины исходного дерева, которые меньше чем item. В результате join(join(outerTl, newTl), item, join(Tr, outerTr)) – это дерево, которое содержит все вершины исходного дерева, все элементы массива items, которые меньше или равны item.

5.5 Оценка временной сложности метода insertAll

Дадим временную сложность работу алгоритма в худшем случае для АВЛ и красно-черного дерева и в среднем для декартового дерева. Пусть в исходном дереве n вершин. Хотим вставить m новых вершин. Для простоты возьмем оценку сверху на сложность метода join O(log n) и сложность join2 для красно-черного и АВЛ-дерева O(log n). Для декартового дерева возьмем для join2 среднее значение log n.

5.5.1 Work сложность метода insertAll

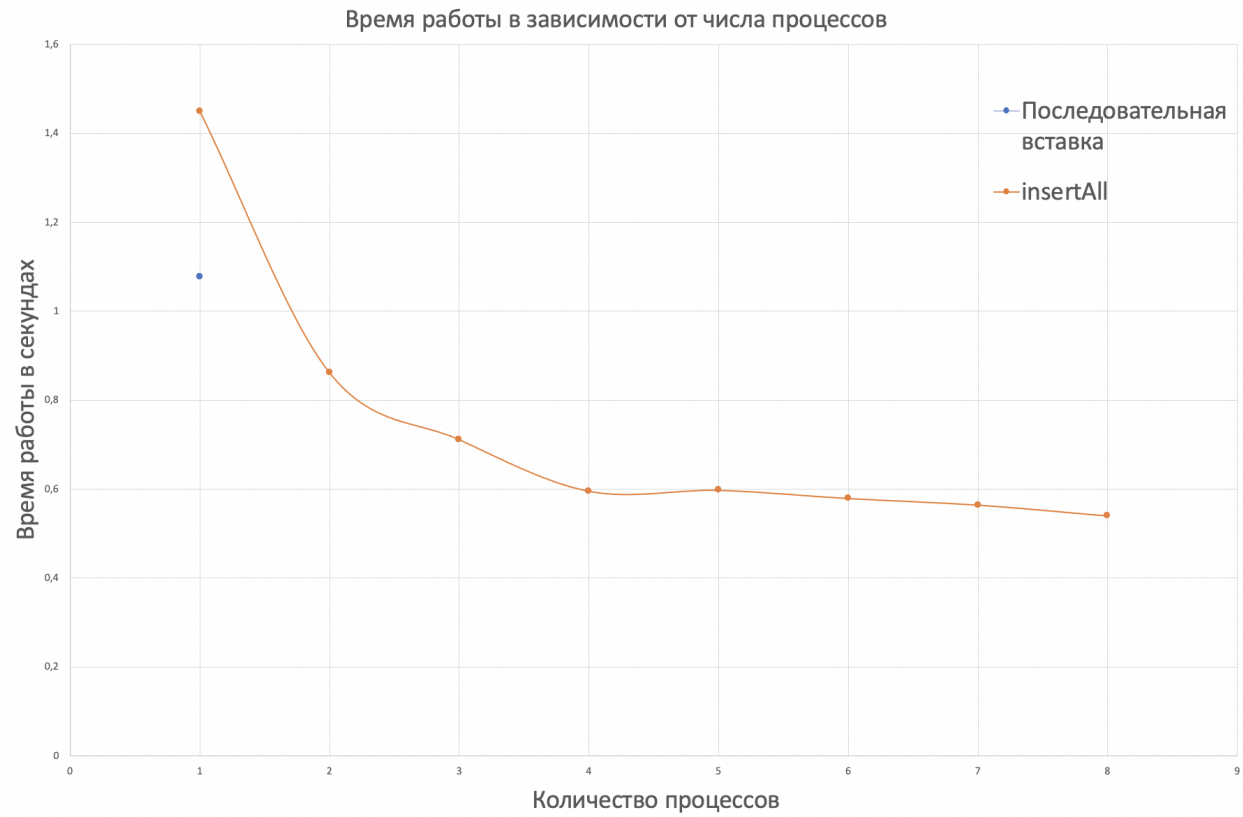
Посчитаем work сложность. На первом этапе m раз вызываем split сбалансированного дерева с n вершинами. То есть на первом этапе получаем O(m · log(n)) операций. На втором этапе вызываем функцию join m раз для деревьев, содержащих максимум n элементов. Получаем еще O(m · log(n)). На третьем m раз вызываем 2 раза join и 6 раз join2 и опять получаем O(m · log(n)). На финальном этапе проходим по массиву splitNodes размера m и вызываем join каждый раз, что также дает O(m · log(n)). Итого work сложность алгоритма составляет O(m · log(n)).

5.5.2 Span сложность метода insertAll

Посчитаем Span сложность алгоритма. Распараллеливание алгоритма происходит m раз. Тогда итоговая глубина равна O(log(m)). На каждом уровне глубины происходит несколько вызовов split и join, что дает O(log(n)). Итого Span сложность алгоритма составляет O(log(n) · log(m)).

5.6 Результаты эксперимента запуска insertAll

Для эксперимента использовался кластер, у которого 32 гигабайта оперативной памяти и 16 процессоров Intel Xeon Gold 6230 с тактовой частотой 2000 МГц. Эксперимент был следующий. Создавали дерево и вставляли в него 10^6 элементов. После этого засекали время и вставляли еще 10^5элементов. В результате получали время вставки 10^5 элементов в дерево размера 10^6. Ниже на изображении 4 можно увидеть график зависимости времени вставки в секундах от количество выделяемых процессов. Синяя точка на графике - это вставка 10^5 элементов последовательным вызовом insert 10^5 раз. Как видно из графика, время работы значительно уменьшилось даже при использовании только 2 процессов.



(Рис. 7 – График зависимости времени работы от числа процессов)

Заметим, что график согласуется с законом Амдала, который гласит, что ускорение параллельной программы зависит не от количества процессоров, а от величины последовательной части программы.

ГЛАВА 6. МЕТОД deleteAll

В этой главе мы полностью опишем метод deleteAll, реализация которого является одной из целью этой работы.

6.1 Описание метода deleteAll

На вход методу deleteAll подается массив элементов, которые нужно удалить. В качестве результата, операция должна вернуть массив деревьев, где на i-ой позиции содержится дерево, которое содержит все элементы исходного дерева кроме первых i элементов отсортированного входного массива. Полученный массив должен совпадать с массивом версиями

деревьев, если бы мы последовательно удаляли элементы массива в персистентном дереве.

6.2 Описание алгоритма deleteAll

Алгоритм метода deleteAll схож с методом insertAll. Работу алгоритма можно разделить на несколько этапов.

Этап 0. Предобработка. Во время предобработки отсортируем входной массив items. Создадим массив splitNodes, размер которого равен размеру массива items. Массив splitNodes будет хранить структуры SplitNode описанные ранее.

Этап 1. На этом этапе мы осуществляем спуск сверху вниз. Цель этого этапа заполнить поля tr для каждой SplitNode из массива splitNodes. Берем центральный элемент массива items (items[center], где center = items.size() / 2) и вызываем split исходного дерева по этому элементу. В результате выполнения метода split мы получаем два дерева tl и tr, где tl – это левый результат split, а tr правый результат split. В двух потоках повторяем рекурсивно первый этап для tl и tr, но в качестве items для tl передаем элементы меньше центрального, а для tr больше центрального. После каждого split для splitNodes[center] можем заполнить поле tr правым результатом split.

Этап 2. На этом этапе мы осуществляем подъем снизу вверх. Цель этого этапа заполнить поле newTl для каждой SplitNode из массива splitNodes. newTl – это дерево, которое содержит все вершины tl кроме элементов items переданный на первом этапе при рекурсивном вызове. В реализации данный этап удобно объединять с первым этапом. Результатом вызова рекурсивной функции из первого этапа сделать деревья newTl и newTr. Для этого достаточно возвращать join(newTl, newTr).

Этап 3. На данном этапе мы осуществляем спуск сверху вниз. Цель этого этапа заполнить поля outerTr и outerTl для каждой SplitNode из массива splitNodes. outerTl – это дерево содержащее все вершины левых результатов

split с удаленными элементами массива items до выполнения split по текущему элементу. outerTr – это дерево содержащее все вершины правых результатов split до выполнения split по текущему элементу. Реализация похожа на первый этап. Рекурсивно для левой и правой части выполняем этот этап, тем самым распараллеливаясь. В качестве входных параметров два аргумента, которые будут outerTr и outerTl. Для левого ребенка передаем outerTl и join(tr, outerTr). Для правого ребенка join(outerTl, newTl) и outerTr.

Этап 4. Получение результата. Для получения результат необходимо просто пройтись по массиву splitNodes и для каждого элемента вызвать

```
join(join(outerTl, newTl), join(Tr, outerTr))
```

Так как outerTl – это дерево содержащее все вершины левых результатов split, со удаленным элементами до выполнения split по текущему элементу, а newTl – это дерево, которое содержит все вершины tl и кроме элементов items, переданные на первом этапе при рекурсивном вызове, то join(outerTl, newTl) – это дерево содержащее все элементы исходного дерева, которые меньше чем item, кроме всех элементов удаляемого массива items, которые меньше чем item. Так как outerTr – это дерево содержащее все вершины правых результатов split до выполнения split по текущему элементу, а tr - это правый результат split по элементу item, то join(tr, outerTr) - это дерево содержащее все элементы исходного дерева, которые больше чем item. Тогда join(join(outerTl, newTl), join(Tr, outerTr)) - это дерево содержащее все элементы исходного дерева и кроме всех элементов массива items до элемента item включительно.

6.3 Псевдокод метода deleteAll

```
Array<Tree> deleteAll(Tree tree, Array<T> items) {
    items.sort();
    // создадим массив размера items.size());
    Array<SplitNode> splitNodes(items.size());
    // helper1 выполняет этапы 1 и 2 алгоритма
    helper1(tree, items, splitNodes);
    // helper2 выполняет этап 3 алгоритма
    helper2(splitNodes, EMPTY_TREE, EMPTY_TREE);
    // получение результата
    return splitNodes.map(node -> {
        join(
            join(node.outerTl, node.newTl),
            join(node.Tr, node.outerTr)
        );
    });
}
```

```
// Рекурсивная функция соответствующая этапу 1 и 2
Tree helper1(Tree tree, Array<T> items,
             Array<SplitNode> splitNodes) {
    Int center = items.size() / 2;
    (tl, contains, tr) = tree.split(items[center]);
    Tree newTl, newTr;
    fork2 {
        newTl = helper1(
           tl, items[0:center], splitNodes
        );
    } and {
        newTr = helper1(
           tl, items[center + 1 : items.size()],
           splitNodes
        );
    }
    splitNodes[center] = new SplitNode ();
    splitNodes[center].newTl = newTl;
    splitNodes[center].tr = tr;
    return join(newTl, newTr);
}
```

```
1:  // Рекурсивная функция соответствующая этапу 3
2:  Tree helper2(Array<SplitNode> splitNodes,
3:                Tree outerTl,
4:                Tree outerTr) {
5:      Int center = splitNodes.size() / 2;
6:      splitNodes[center]. outerTl = outerTl;
7:      splitNodes[center]. outerTr = outerTr;
8:      Tree tr = splitNodes[center].tr
9:      Tree newTl = splitNodes[center].newTl;
10:     fork2 {
11:        helper2(
12:          splitNodes[0 : center], outerTl,
13:          join(tr, outerTr)
14:        );
15:     } and {
16:        helper2(
17:          splitNodes[center + 1 : splitNodes.size()],
18:          join(outerTl, newTl), outerTr
19:        );
20:     };
21: }
```

6.4 Сложность и корректность метода deleteAll

Span сложность алгоритма составляет O(log(n) · log(m)). Work сложность алгоритма составляет O(m · log(n)). Из-за схожести с алгоритмом insertAll асимптотическая сложность доказывается аналогично.

Доказательство корректности также аналогично доказательству корректности для метода insertAll.

ЗАКЛЮЧЕНИЕ

В этой работе были разработаны алгоритмы параллельной вставки и удаления объектов для персистентных двоичных сбалансированных деревьев поиска, в частности для красно-черных, АВЛ и декартовых деревьев. В отличии от уже известных алгоритмов параллельной вставки и удаления они сохраняют свойства персистентности. Описанный в работе метод insertAll принимает массив вставляемых элементов и возвращает массив деревьев, совпадающий с массивом деревьев, которые можно получить последовательной вставкой элементов из отсортированного входного массива. Аналогично описанный в работе метод deleteAll, который, наоборот, удаляет элементы дерева. Для этих методов приложен их псевдокод. Также в работе оценена их асимптотическая сложность. Их span сложность составляет $O(\log(n) \cdot \log(m))$, а Work сложность составляет $O(m \cdot \log(n))$. Описываемые в этой работе эксперименты по запуску предлагаемых методов показали, что они хорошо распараллеливаются.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Бинарные деревья. Построение дерева. Обход дерева. Поиск по дереву. Удаление элементов. Сбалансированные деревья. АВЛдеревья. Красно-черные деревья. Оптимальные деревья поиска. [Электронный ресурс]: Мегаобучалка. Образовательный портал. Режим доступа: https://megaobuchalka.ru/2/253.html (дата обращения: 14.05.2021)

[2] Персистентные структуры данных - Базы данных, знаний и хранилища данных. Big data, СУБД и SQL и noSQL [Электронный ресурс]: Портал искусственного интеллекта. Образовательный портал. Режим доступа: https://intellect.icu/persistentnye-struktury-dannykh-7729 (дата обращения: 14.05.2021)

[3] Т.Кормен, Ч.Лейзерсон, Р.Ривест, К.Штайн - Алгоритмы. Построение и анализ. Издание 3-е.

[4] Guy E. Blelloch, Daniel Ferizovic, Yihan Sun - Just Join for Parallel Ordered Sets.

[5] Point location [Электронный ресурс]: Википедия. Свободная энциклопедия. – Режим доступа: https://en.wikipedia.org/wiki/Point_location (дата обращения: 14.05.2021).

[6] Y. Akhremtsev and P. Sanders. Fast parallel operations on search trees. arXiv preprint arXiv:1510.05433, 2015.