Тема 5. Быстрые деревья поиска

2-3 дерево

Еще один вариант построения быстрого дерева поиска — это так называемое 2-3 дерево. В этом случае узел дерева хранит либо одно значение, либо два и сооветственно такой узел имеет либо два, либо три потомка.

Подобное расширенияе понятия вершины дерева позволяет нам добиться в некотором смысле идеальной сбалансированности.

Определение. 2-3 дерево — это древовидный граф, который удовлетворяет следующим условиям:

1. Каждая вершина содержит либо одно, либо два ключа, и при этом любая внутренняя (не концевая) вершина имеет соответственно либо два либо три потомка, таким образом, мы вводим термины 2-вершина и 3-вершина.

2. Дерево упорядочено по ключам, именно, упорядоченность 2-вершины полностью аналогична упорядоченности обычного бинарного дерева поиска, а в 3-вершине ключи упорядочены по возрастанию и ключи поддерева, на которое показывает "средний" указатель, соответственно больше первого ключа и меньше второго ключа данной вершины.

3. Все листы (концевые вершины, не имющие потомков) данного дерева лежат на одном уровне.1111

Таким образом, мы получаем дерево, которое в минимальном варианте представляет собой идеально сбалансированное бинарное дерево, а в максимальном — идеальное тернарное дерево. Понятно, что для N элементов мы получаем глубину дерева от $log_3 N$ до $log_2 N$.

Поиск в данном дереве не представляет проблем с идейной точки зрения. Действительно, при анализе 3-вершины нужно просто сравнивать искомый ключ с двумя значениями и соответственно перемещаться в поддерево, которое удовлетворяет требуемому неравенству.

Но с точки зрения реализации такого дерева мы наталкиваемся на проблему что вершины имеют разную структуру, а нам надо как-то универсально задавать указатели на потомков.

Прямолинейный подход может заключаться в реализации только тернарных 3-вершин с указанием сколько конкретно значений или потомков — 2 или 3 — ей соответствует. Однако при таком решении значительная часть памяти может пропадать зря. Эффективному решению здесь может помочь концепция наследования.

Рассмотрим вот такую реализацию вершин (для краткости опустим пока template)

```
struct TreeNode {
    T value;
    TreeNode *first, *second;
};

struct TreeNode3 : public TreeNode {
    T value2;
    TreeNode *third;
};
```

Здесь 3-вершина (TreeNode3) есть наследник 2-вершины (TreeNode). В ней добавлены дополнительные поля для второго ключа и третьего потомка. Теперь мы можем использовать указатель `TreeNode *`, который будет в разных ситуаций представлять собой либо базовый класс, либо порожденный.

Рассмотрим теперь процедуры добавления и удаления элементов в такое дерево. При этом постараемся воспользоваться свойствами наследования для упрощения записи алгоритмов. Как всегда предполагаем, что все ключи в дереве уникальны.

Добавление

Идея. Новый элемент всегда добавляется в концевую вершину. Для этого сначала естественным поиском определяется эта вершина, а потом находится место для нового элемента в этой вершине. Если найденная вершина есть 2-вершина, то она просто преобразуеттся в 3-вершину, хранящую ее старый ключ и новый добавленный. Если она была 3-вершиной, то она расщепляется на 3 компоненты: левый ключ, средний "корень", правый ключ, которые как бы образуют 2-вершину с двумя листами-потомками, упорядоченно выстроенными из трех значений — двух старых корней данной 3-вершины и нового добавленного значения. Теперь вместо старого 3-листа мы должны включить эту конструкцию в родительскую вершину данного 3-листа. Таким образом, мы свели вставку дополнительного элемента в 3-вершину к вставке нового значения (вместе с его правым и левым поддеревьями) в родительскую вершину. В свою очередь, родительская вершина также может расщепиться из 3-вершины в два поддерева с общим корнем в 2-вершине, который должен быть включен в очередного родителя вверх по ветви дерева. Если этот процесс дойдет вплоть до корня всего дерева, то у дерева появится новый корень, и все дерево "подрастет" в длине.

Иллюстрация добавления в 2-3 дерево

С точки зрения реализации добавления можно предложить такую рекурсивную процедуру. Имеем на входе данное значение X и корень текущего поддерева. Сравнением с ключами данного корня определяем потомка, в который нужно добавлять это значение. Рекурсивно вызываем процедуру добавления от этого потомка. Этот вызов может завершиться следующим образом:

1. Корень поддерева не изменился — далее ничего делать не надо.
2. Корень поддерева сменил тип с 2-вершины на 3-вершину. Нужно только сменить указатель из текущей (родительской) вершины на этого потомка.
3. Поддерево расщепилось на 3 компоненты, и среднюю из этих компонент нужно добавить в текущею вершину вместе с указателями на крайние потомки-поддеревья.

Заметим, что п.3 по-разному будет реализован для текущей 2-вершины и 3-вершины. Это можно естественно сделать, используюя виртуальную функцию добавления значения в поддерево с корнем в конкретной вершине.

Возможная реализация приведена в примере Tree23.h, где реализовано добавление в 2-3 дерево с простейшим тестом (распечаткой состояния вершин). В этом примере рабочей функцией для добавления является

```
virtual TreeNode * Add(T &x, TreeNode *&q2);
```

которая возвращает указатель на корень поддерева после добавления. Если при этом не возникло расщепление, то указатель q2 будет равен nullptr. Если же расщепление возникло, но возвращаемый указатель есть левая компонента, q2 есть правая компонента, и x содержит значение, которое нужно добавить в текщую (родительскую вершину).

Удаление

Как и в бинарных деревьях, здесь применяется похожая техника. Значение всегда удаляется только из концевой вершины. Если требуется удалить значение из внутренней вершины, то ищется элемент из концевой вершины, который может его подменить, производится подмена, и потом удаляется этот подменный элемент.

При реальном удалении ситуация становится обратной. Вместо расщепления поддерева может потребоваться слияние поддеревьев.

Иллюстрация удаления

Таким образом, здесь тоже можно эффективно использовать рекурсивную технику. Находясь в текущей вершине, мы ищем потомка, из которого надо удалять значение, и рекурсивно вызываем процедуру удаления от этого потомка. Возможны 3 варианта завершения этой процедуры.

1. Корень поддерева не изменился — на этом все заканчивается.

2. Корень поддерева сменил тип, но поддерево сохранило прежнюю длину. Заменяем указатель на потомка и на этом конец.

3. В результате получилось поддерево меньшей длины. Проводится корректировка в текущей родительской вершине, которая может также привести к сокращению длины в этом поддереве.

По аналогии с добавлением можно предложить такую функцию

```
virtual TreeNode * Remove(T &x, bool &reduced);
```

Функция возвращяет указатель на корень поддерева после удаления, а параметр reduced показывает сократилась ли длина этого поддерева.

При реализации данной функции нам нужно проверять к какому типу относятся поддеревmя данной текущей вершины. Это можно сдедать с помощью инструкций

```
TreeNode *p;
p = ....
TreeNode3* q = dynamic_cast<TreeNode3*>(p);
```

Если p есть на самом деле TreeNode3*, то q получит правильное ненулевое значение соответствущего указателя. Если же p есть указатель на родительский класс TreeNode, то q будет равно nullptr.