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 соответственно такой узел имеет либо два, либо три потомка.

Подобное расширение 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.

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. Можно предложить несколько вариантов построения узле дерева.

Прямолинейный подход состоит реализации только тернарных 3-вершин.

```
struct TreeNode {
    T value[2];
    TreeNode *child[3];
};
```

Если узел является 2-вершиной, то часть данных в такой структуре не используется и память пропадает зря. Однако нам не надо заботиться о преобразованиях 2-вершину в 3-вершину, если этого потребуют алгоритмы работы с таким деревом (а они этого потребуют). Чтобы различать 2-вершину и 3-вершину можно принять дополнительные соглашения, например, указатель child[2] будет равняться 0 для любых 2-вершин, в листе child[0] = child[1] = 0, а указатель child[2] можно установить в адрес этого самого листа для случая 3-листа.

Можно в вершине поддерживать массивы разной длины, т.е.

```
struct TreeNode {
    T *value;
    TreeNode **child;
    int k; // тип вершины 2 или 3
};
```

и к этим указателям привязывать массивы разной длины.

Можно вообще воспользоваться стандартными контейнерами

```
struct TreeNode {
    vector<T> value;
    vector<TreeNode*> child;
};
```

Наконец, можно применить наследование, и ввести два типа вершин

```
struct TreeNode {
    T value;
    TreeNode *first, *second;
};
struct TreeNode3 : public TreeNode {
    T value2;
    TreeNode *third;
};
```

При этом указатель TreeNode* может представлять собой как указатель на базовый класс TreeNode, так и указатель на порожденный класс TreeNode3.

Рассмотрим теперь процедуры добавления и удаления элементов в такое дерево. 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 преобразуется 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 образуют 2-вершину с двумя листами-потомками, упорядоченно выстроенными из трех значений — двух старых значений данной 3-вершины и нового добавленного значения. Теперь вместо старого 3-листа мы должны включить эту конструкцию его в родительскую вершину. 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 рекурсивную процедуру

```
TreeNode * Add(const T& x, TreeNode *r);
```

которая будет возвращать указатель на поддерево, полученное после добавления от текущего корня r.

При рекурсивном подходе мы выясняем в какого потомка надо перенаправить добавление (в соответствии с тебуемой упорядоченностью), получаем в ответ корень измененного поддерева, и корректируем указатели от текущей корневой вершины

Добавление в поддерево может завершиться следующим образом:

1. Корень поддерева, куда мы добавляли, не изменился — далее ничего делать не надо (дерево перестроилось без влияния на его данный текущий корень).

2. Корень поддерева сменил тип с 2-вершины на 3-вершину. То есть подъем (расщепление) завершился в данной вершине так как в ней нашлось место для нового значения. Нужно только сменить указатель из текущей (родительской) вершины на этого потомка.

3. Поддерево в своем корне расщепилось на 3 компоненты, и среднюю из этих компонент нужно добавить в текущею вершину вместе с указателями на крайние потомки-поддеревья.

Заметим, что в случаях 2 и 3 нам уже не нужна вершина старого корня данного потомка, и кто-то ее должен уничтожить, так как она уже не участвует в перестроенном дереве.

Можно набросать примерный план реализации функции добавления. Основная сложность — проверки на тип вершины и выбор потомка, куда надо переадресовать добавления. Пусть эта работа выполняется отдельной процедурой

```
// r - указатель на текущую вершину
// x - добавляемое значение
// ch - потомок для добавления - 0, 1, 2, и -1, если x уже есть в этой вершине
// tp - тип вершины r - 2, 3
// return - указатель r->child[ch] или 0, если r лист

TreeNode * SelectChild (TreeNode *r, const T&x, int &ch, int &tp);
```

Также можно ввести процедуры преобразования вершин при добавлении

```
// добавляется дерево с 2-корнем v в текущую 2-вершину r
TreeNode * Transform2to3 (TreeNode *r, int ch, TreeNode *v);

// добавляется дерево с 2-корнем v в текущую 3-вершину r
// происходит расщепление
TreeNode * Transform3to2 (TreeNode *r, int ch, TreeNode *v);

// варианты функций, если r - это лист,
// тут уже нет поддеревьев, поэтому вставляем непосредственно x
TreeNode * Transform2to3 (TreeNode *r, int ch, const T&x);
TreeNode * Transform3to2 (TreeNode *r, int ch, const T&x);
```

Во всех случаях ch — это номер места, куда надо вставлять, а возвращаемое значение — преобразованная вершина (т.е. корень получившегося поддерева).

Теперь можно записать процедуру добавления с использованием этих функций. Пока для простоты считаем, что вершина дерева реализована по варианту 1, т.е. нам не надо тсоздавать вершины разных типов, а надо только правильно заполнять их содержимое.

```
TreeNode * Add(const T& x, TreeNode *r)
{
    TreeNode *q;
    // утв:  r ненулевой указатель
    int ch, tp;
    TreeNode *q;
    TreeNode *p = SelectChild(r, x, ch, tp);
    if (ch == -1) { // что делаем при наличии такого значения x?
        return nullptr;
    }
    // нужна рекурсия или нет?
    if (p == nullptr) { // мы в листе
       switch(tp) {  // проверка типа вершины
         case 2: q = Transform2to3 (r, ch, x);  break;
         case 3: q = Transform3to2 (r, ch, x);  break;
       }
       return q;
```

```
    } else {  // можно применить рекурсию
       q = Add(p, x);
       if (!q) return nullptr;  // значение уже есть
       // теперь надо разбираться с текущей вершиной
       if (r->child[ch] == q) { // вариант 1
          return r;
       } else if (q->child[2] == q) { // вариант 2, т.е. пришла 3 вершина
           r->child[ch] = q     // прикрепляем новую
       } else { // вариант 3, пришла 2 вершина от расщепления ниже
           switch(tp) {  // проверка типа текущей вершины
              case 2: p = Transform2to3 (r, ch, q);  break;
              case 3: p = Transform3to2 (r, ch, q);  break;
           }
           return p;
       }
    }
}
```

Здесь подразумевается, что функции, выполняющие преобразования вершин, сами удаляют старый вариант текущей вершины r.

Для случая наследования при реализации TreeNode нужно иметь в виду, что проверку того каким является указатель TreeNode* (базовым или порожденным) легко сделать с помощью динамического преобразования типа

```
TreeNode * TreeNode::Transform(TreeNode *v, int ch);  // 2 to 3
TreeNode * TreeNode::Transform(const T&x);  // то же для листа
TreeNode * TreeNode::SelectChild (const T&x, int &ch);

TreeNode * TreeNode3::Transform(TreeNode *v, int ch);  // 3 to 2
TreeNode * TreeNode3::Transform(const T&x);  // то же для листа
TreeNode * TreeNode3::SelectChild (const T&x, int &ch);

TreeNode * Add(const T& x, TreeNode *r)
{
    TreeNode *q;
    // утв:  r ненулевой указатель
    int ch;
    if (r->second == nullptr) { // проверка на лист
       q = r->Transform(x);
       delete r;   // удаляем старый
       return q;   // возвращаем новый
    }
    // теперь рекурсия
    TreeNode *p = r->SelectChild(x, ch);
    q = Add(x, p);
    if (p == q) {
        return r;
    } else if(dynamic_cast<TreeNode3*>(q)) {
        delete p;
        // обновить указатель от родителя не совсем просто
        switch (ch) {
          case 0:  r->first = q; break;
          case 1:  r->second = q; break;
          case 2:  dynamic_cast<TreeNode3*>(r)->third = q; break;
        }
```

```
        return r;
    } else {
        p = r->Transform(q, ch);
        delete r;
        return p;
    }
}
```

Здесь опущены проверки на существование значения, но они также легко встраиваются в функции SelectChild.

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 (наибольший слева или наименьший справа), 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 потомка не изменился — на этом все заканчивается.

2. Корень потомка 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.

Заметим, что факт изменения длины можно также обнаружить по изменению типа вершины в корне поддерева. Если тип изменился с 3 на 2, это означает, что “слияние вершин” произошло ранее, и из данной вершины было заимствование значения. Т.е. здесь длина не изменилась. Если тип изменился с 2 на 3, это означает, что слияние произодло именно здесь, и таким образом длина изменилась.

Процедура удаления строится аналогично процедуре добавления, т.е. отдельно обрабатывается случай листа, а далее каждый уровень рекурсивного вызова корректирует указатели родителя в зависимости от типа вершины, полученной от предыдущего рекурсивного вызова.