Тема 5. Быстрые деревья поиска

B-дерево

Теперь другой принцип построения быстрых деревьев.

Требуем идеальной сбалансированности по длине ветвей от корня, и добиваемся этого, допуская разное количество значений (ключей) в узлах дерева.

Определение. B-деревом порядка n называется древовидная структура, удовлетворяющая следующим условиям:

— вершиной дерева является массив, способный вместить $2n$ элементов данных;

— в каждой вершине элементы данных расположены в массиве в порядке возрастания их ключей.

— каждая вершина, кроме корневой, содержит не менее n и не более $2n$ элементов данных;

— вершина, содержащая k элементов данных, имеет ровно $(k+1)$ потомков, либо является концевой, т.е. не имеет потомков; при этом говорим, что j-й элемент имеет левое поддерево и правое поддерево, определяемое соответственно j-м и $(j+1)$-м потомками;

— для каждого элемента то ключи всех элементов левого поддерева меньше, а все элементы правого поддерева больше ключа этого элемента (т.е. ключи элементов и их потомков упорядочены естественным образом);

— все концевые вершины лежат на одном уровне дерева;

Иллюстраия: B-дерево порядка 2.

Легко видеть, что количество элементов B-дерева растет как n^k, где k — глубина дерева

Сложность поиска. Пусть N — количество элементов в дереве. Тогда глубина дерева есть $O(\log_n N)$. В каждой вершине бинарным поиском находим либо элемент, либо ссылку на потомка, т.е. за $O(\log_2 n)$. Общая сложность $O(\log_2 n \cdot \log_n N) = O(\log_2 N)$.

```
int n_order;  // порядок дерева

struct BTreeNode {
   T * values;         // массив элементов длины 2*n_order
   int size;          // количество потомков
   BTreeNode **child; // массив указателей на потомков (2*n_order + 1)
   BTreeNode *parent;

   BTreeNode (int n) {
      values = new T[2*n];
      child = new BTreeNode*[2*n+1];
      for (int i=0; i<2*n+1; i++) child[i] = nullptr;
      parent = nullptr;
      size = 0;
   }
};

// бинарный поиск в вершине, получает индекс элемента (если есть, return true)
// или индекс указателя на потомка (если нет, return false)
```

```
bool BinSearch (const T &x, T *values, int size, int &index);

BTreeNode *BTreeSearch (BTreeNode * root, const T &x, int &index)
{
    if (root == nullptr) {
        index = -1;
        return nullptr;
    }
    if (BinSearch (x, root->values, root->size, index)) {
        return root;
    }
    return BTreeSearch (x, root->child[index], index);
}
```

Добавление.

Ищем подходящее место и добавляем в концевую вершину. Если место есть, то все. Если места нет, то необходима балансировка. Два типа балансировки: перебрасывание от соседней вершины или разделение вершины.

Можно рассматривать рекурсивный вариант и нерекурсивный (с добавлением указателя на родительскую вершину).

Нерекурсивный вариант (набросок с массой неточностей):

```
BTreeNode * Add (BTreeNode * root, const T &x)
{
    BTreeNode *p = root;
    if (root == nullptr) {
        p = new BTreeNode(n_order);
        p->value[0] = x;
        p->size = 1;
        return p;
    }
    // поиск нужной концевой вершины
    bool r;
    int index;
    while(true) {
       r = BinSearch (x, root->value, root->size, index);
       if (r) {ошибка --- x должен быть уникальным}
       if (root->child[index] == nullptr) break;
       root = root->child[index];
    }
    // теперь root --- нужная концевая вершина
    // index --- позиция вставки
    BTreeNode *q = nullptr;
    while(true) {
       if (root->size < 2*n_order) {
          // места хватает
          InsertAt(x, root->value, root->size, index, q);
          ++(root->size);
```

```
            return p;
         } else {
            // а если места нет, то делим вершину
            q = new BTreeNode(n_order);
            // аккуратно распределяем элементы
            // с учетом позиции вставленного элемента x
            // значение y --- средний элемент
            root->size = n_order;
            q->size = n_order;
            x = y;
            if (root->parent) {
               // добавляем лишний элемент в родительскую вершину (продолжаем цикл)
               q->parent = root->parent;
               root = root->parent;
            } else {  // т.е. добавление было в корень всего дерева
               p = new BTreeNode(n_order);
               p->value[0] = x;
               p->size = 1;
               p->child[0] = root;
               p->child[1] = q;
               root->parent = p;
               q->parent = p;
               return p;
            }
         }
      }
   }

  InsertAt(x, root->value, root->size, index, q);
  вставляет элемент x по индексу index
  и указатель на потомка q по индексу index+1
```

Рекурсивный вариант. Добавляем в поддерево, а на выходе получаем два поддерева и еще один элемент. Потом все это собираем в полное дерево.

```
bool AddToSubtree (const T &x, BTreeNode *&first, BTreeNode *&second, T &y)
{
    bool r;
    int index;
    r = BinSearch (x, first->value, first->size, index);
    BTreeNode *p = first->child[index];
    if (p) {
        if (AddToSubtree(x, p, second, y)) return true;
    } else {
        second = 0;
        y = x;
    }
    // теперь должны вставить y и указатель second в текущую вершину
```

```
        // на место, определяемое позицией index
        if (first->size < 2*n_order) {
            InsertAt(y, first->value, first->size, index, second);
            ++(first->size);
            return true;
        } else {
            p = new BTreeNode(n_order);
            аккуратно вычисляем позиции частей по значениям и указателям
            определяем серединный элемент
            пересылаем половину значений и указателей в вершину p
            оставляем половину значений и указателей в вершине first
            y = серединный элемент;
            second = p;
            return false;
        }
    }

    BTreeNode * Add (BTreeNode *root, const T &x)
    {
        if (root == nullptr) {
            root = new BTreeNode(n_order);
            root->value[0] = x;
            root->size = 1;
            return root;
        }
        T y;
        BTreeNode * first = root, *second;
        if (AddToSubtree (x, first, second, y)) return first;

        // иначе добавляем в корень по аналогии
        // с возможным созданием нового корня с одним элементом
        if (root->size < 2*n_order) {
            InsertAt(y, first, first->size, index, second);
            return first;
        } else {
            root = new BTreeNode(n_order);
            root->value[0] = y;
            root->child[0] = first;
            root->child[1] = second;
            root->size = 1;
            return root;
        }
    }
```

Удаление.

Идейно — как обычно, т.е. физически удаляем только из концевой вершины, а если удаляемый элемент не в концевой врешине, то подменяем его на подходящий элемент из концевой, и потом удаляем подменный элемент.

Если при удалении количество элементов в вершине становится меньше n, то можно восстановить условя B-дерева либо переносом элементов из соседних вершин данного уровня, либо слиянием соседних вершин, по обратной аналогии с процедурой разделения вершин при добавлении.

B+ дерево

Еще одна модификация B дерева называется B+ деревом. Для описания такой структуры мы прежде всего заметим, что обычно элемент, хранящееся в дереве представляет собой пару "ключ — значение", т.е. дерево реализует отображене множества ключей на множество значений. Упорядоченность в данном случае происходит по ключам, а сооветствующие значения просто идут в пару к ключам.

В B+ дереве ключи и значения "отделяются" друг от друга, ключи хранятся в B дереве, а значения хранятся отдельно. Данную идею можно реализовать по-разному. Например, два возможных варианта выглядят так.

1. В дереве имеются узлы двух видов: внутренние узлы для ключей и концевые узлы для ключей и значений. При этом все значения и все ключи полностью присутствуют в концевых вершинах, а во внутренних вершинах хранятся только ключи, которые при поиске обеспечивают спуск до необходимой концевой вершины (по принципу B-дерева). При этом для облегчения выбора подходящего ключа во внутренних вершинах условие сравнения ключей может быть ослаблено включением равенства с одной стороны, например, Left <= key < Right. Концевые вершины также завязаны в двунаправленный список (т.е. не отдельные значения, а вершины целиком). Итератор по такому дереву реализуется совсем просто.

Иллюстрация.

2. Ключи собраны в обычное B-дерево, но вместе с каждым ключем также хранится указатель на значение, а все значения в свою очередь связаны в двунаправленный список. В этом варианте работа с B+ деревом ничем не отличается от работы с обычным деревом, и при этом итератор тоже реализуется намного проще (как итератор по списку). Фактически это получается просто B-дерево указателей на значения, завязанные в список.

В обих случаях при поиске элемента по ключу выполняется проход по узлем и веткам дерева и при обнаружении ключа мы имеем возможность спуститься далее к соответствующему потомку (в первом варианте) либо сразу добраться до значения по хранящемуся рядом указателю (во втором варианте).

Операции добавления и удаления модифицируются без проблем, поскольку добавляемый или удаляемый элемент имеет ближайших соседей в смысле упорядоченности ключей, доступ к которым легко определяется как по дереву (для ключей) так и по списку (для значений). Т.е. мы знаем место и можем вставить новый элемент в дерево и в список, либо удалить элемент из списка и дерева с сохранением требуемых связей.

Однако в первом варианте возникает проблема со ссылками на потомков из внутренних вершин дерева поскольку эти потомки в разных местах могут быть разными объектами (либо внутренними вершинами, либо концевыми).

Решить эту проблему может механизм наследования C++. О нем чуть позже, а сейчас еще один пример из этой же категории деревьев.

2-3-дерево

2-3-дерево — это B-дерево порядка 1. Таким образом, в вершинах 2-3-дерева хранятся либо 1, либо 2 ключа (и значения), и соответственно они имеют либо 2, либо 3 потомка. Все идеи работы с B-деревом переносятся и на этот случай, только при добавлении или удалении элементов всегда происходит разделение или слияние вершин, поскольку перенос вершин из соседей для освобождения/занятия места здесь уже не имеет смысла.

Можно реализовать 2-3-дерево на базе вершины с двумя ключами (значениями) и тремя потомками, и использовать из них столько, сколько требуется для конкретной вершины. Но такой подход может показаться нерациональным, так как часть памяти вершинах будет заведомо пропадать попусту. Поэтому мы можем попробовать ввести два типа вершин (2-вершины и 3-вершины по количеству потомков). В такой ситуации мы получаем ту же проблему, что и в B+ дереве — необходимость ссылаться на вершины разных типов из каждой конкретной родительской вершины.

Опять же эту проблему решает наследование.