Тема 5. Быстрые деревья поиска

B дерево

Теперь другой принцип построения быстрых деревьев.

Развитие тех же идей, что в 2-3 дереве, но на большее количество значений (ключей) в вершине дерева.

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

— вершина дерева содержит 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) { // случай пустого дерева
        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.

Хранение дерева во внешней памяти.

Небольшая глубина B деревьев оказалась очень удобна для использования их в базах данных при хранении на внешних устройствах (дисках). Чтение с диска является медленной операциейю и также за один аппаратный вызов читается некоторая порция данных (обычно такая порция кратна 1-4 килобайтам). Таким образом, мы можем разместить дерево целиком во внешней памяти, а в оперативной памяти держать только его малую часть, с которой в данный момент работаем. При необходимости недостающие части дерева считываются с диска или сохраняются на диск. Малая глубина минимиирует число обращений к внешней памяти при поиске и модификации дерева, что повышает быстродействие.

Если все вершины дерева записаны в файл, то позиция каждой отдельной вершиной может характеризоваться смещением записи от начала файла. Таким образом, мы можем модифицировать структуру узла дерева, добавив в нее это смещение. Тогда мы сможем записывать или читать эту вершину при необходимости ее сохранить на диске или загрузить в оперативную память.

```
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;
   size_t *offset;  // массив смещений файловых позиций для потомков
};                                    длина  (2*n_order + 1)
```

Если потомок размещен в памяти, то он определяется корректным указателем `child[i]`, в противном случае у нас есть смещение `offset[i]` и мы можем загрузить эту вершину с диска в оперативную память.

Итератор по B дереву.

Реализация итератора идейно повторяет предыдущие итераторы по деревьям, но есть одно существенное отличие. В случае бинарного дерева мы всегда проходили каждую конкретную вершину как бы два раза — один раз по пути “вниз” и один раз по пути “вверх”. Поэтому мы всегда могди выбрать на каком проходе нам эту вершины считать текущей и какая вершина будет считаться следующей. В случае B дерева мы попадаем в каждую конкретную вершину несколько раз — один раз при проходе “вниз” и $k+1$ раз при проходе вверх, где $k+1$ есть количество потомков данной вершины. При этом, возвращясь каждый раз в определенную вершину, мы не знаем к какому потомку мы отправились в прошлый раз и соответственно какое значение этой вершины или какой потомок долны быть следующими. У этой проблемы есть два решения.

1. Можно для каждой вершины запоминать позицию, с который мы ушли для обработки потомка. Вернувшись в эту вершину, мы прочитаем эту позицию и определим следующего потомка. Для этого храения естетсвенно использовать стек, который должен быть равен по максимальной глубине глубине данного дерева (по аналогии

с итератором по бинарному дереву без ссылки parent, когда запоминались указатель на пройденную в данный момент ветку дерева). Так как B дерево не очень глубокое, то это стек не будет для итератора слицком обременительным.

2. Можно определять предыдущую позицию непосредственной проверкой. Действительно, вернувшись в вершину от некоторого потомка, мы принесем с собой так же указатель на этого потомка. Потом можно последовательным поиском найти этот указатель в массиве child и тем самым определить предыдущую и далее следующую позиции отератора. Здесь выполняется относительно медленная процедура последовательного поиска, зачисящая от количества значений в данной вершине.

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 в некотором смысле отдельно.

Обычно предполагается, что 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, в левом потомке от некоторого внутреннего ключа key все значения $<$ key, а для правого потомка выполнено неравенство $>=$ key (т.е. key — минимальный ключ из правого поддерева). Концевые вершины также завязаны в двунаправленный список (т.е. не отдельные значения, а вершины целиком). Итератор по такому дереву реализуется совсем просто.

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.

При этом мы опять получаем вершины двух типов, которые должны однородным образом идентифицироваться указателями их внутренних вершин (либо на такие же внутренние, либо на качественно другие концевые). Как и раньше эта проблема легко решается наследованием класса концевых вершшин от класса внутренних вершин.