实验报告

B树

林逸泰

1. 需求分析
   1. B树构建：按照B树的设计要求进行插入
   2. B树的查找：对于给定数据，查找是否在B树中
2. 设计概要
   1. 设计思路：相比二叉树，B树复杂的点在于节点的动态变化，因此问题主要集中在如何设计好节点结构，从而为树提供更方便的接口
   2. 节点
      1. 节点结构：
         1. 基类：\_Node\_base\_：主要是为了测试，因为一开始设计较为复杂，实际上没能完成
         2. 节点类：BTree\_node：
            1. 数据：

static const int \_top\_nd\_num = \_Nm; //M阶B树

int num\_sub\_nds; //当前节点拥有的数据量

data\_list nd\_data; //数据列表

child\_list sub\_nds; //孩子指针列表

bool final\_node; //是否是叶子节点

* + - * 1. 主要方法：

self\* find\_next(const \_Tp\_data& d\_) const;

//非叶子节点，找到插入位置

data\_bag\* insert(const \_Tp\_data& d\_);

//插入数据，新数据，没有指针 之后进行检查是否是满足要求的节点

data\_bag\* insert(data\_bag\* bag);

//插入来自子节点的数据和指针 之后进行检查是否是满足要求的节点

self\* merge(const self& nd\_);

//合并点

bool find(const \_Tp\_data& d\_) const;

//查找

* 1. B树
     1. 数据：

node\_type\* root\_ptr; //根节点指针

* + 1. 主要方法：

bool Insert(const \_Tp\_data& d\_);//如果存在则返回false

bool Search(const \_Tp\_data& d\_);

1. 设计细节
   1. 数据插入实现：
      1. 思路：需要分裂的节点在分裂完成后会传回指针(bag结构体)，通过指针递归修改B树
      2. BTree中的函数：

BTree<\_Tp\_data\_, \_Nm>::insert(node\_type \* nd\_ptr, const \_Tp\_data & d\_)

{

data\_bag\* bag\_ptr;

if (nd\_ptr->is\_final\_node()) { //查看是否是尾部节点

bag\_ptr = nd\_ptr->insert(d\_);

return bag\_ptr;

}

else {//不是尾部节点，递归调用

bag\_ptr = this->insert(nd\_ptr->find\_next(d\_), d\_);

}//查看插入的返回值，如果需要调整，会返回已经调整后的子节点指针

if (bag\_ptr != NULL) { //

return nd\_ptr->insert(bag\_ptr);//合并到当前节点

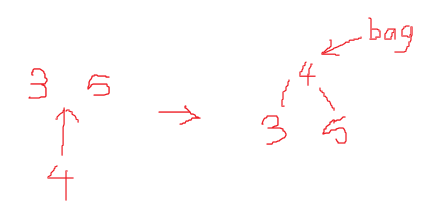
}

return NULL;

}

* + - 1. BTree类中插入函数有一个，如上所示
      2. 节点类中插入函数有两个重载版本，其中一个是插入数据，另一个是一个节点的指针
    1. 节点中的插入函数
       1. 实现上都分为：插入数据，检查是否是合法节点，对非法节点分裂 三部分
       2. 数据插入的比较简单就不展开，对于传入一个节点指针的插入函数：

比如如下的插入：



//插入数据

……

sub\_nds.erase(sub\_nds.begin() + pad);

//删除变化的节点指针，插入新的指针

sub\_nds.insert(sub\_nds.begin() + pad, \*(bag->\_get\_ptr(1)));

sub\_nds.insert(sub\_nds.begin() + pad, \*(bag->\_get\_ptr(0)));

//判断是否需要分裂

//如果需要

pad = (\_top\_nd\_num + 1) / 2;

self\* l\_child = new self(//分裂出左孩子

data\_list(nd\_data.begin(), (nd\_data.begin() + pad - 1)),

child\_list(sub\_nds.begin(), (sub\_nds.begin() + pad)),

false

);

self\* r\_child = new self(//分裂出右孩子

data\_list((nd\_data.begin() + pad), nd\_data.end()),

child\_list((sub\_nds.begin() + pad), sub\_nds.end()),

false

);

auto res = new self(//构造bag

data\_list(1, \*\_get\_data(pad - 1)),

child\_list{ l\_child, r\_child }

);

res->final\_node = false;

return res;

* 1. 查找实现：
     1. 在BTree\_node中：
        1. 遍历节点中的数据，由于节点中的数据为有序的，因而直接使用二分查找
     2. 在BTree中：
        1. 树结构的遍历，使用前序遍历算法进行遍历

if (nd\_ptr == NULL) {

return false;

}

if (!nd\_ptr->find(d\_)) {//当前节点找不到，遍历左右子树继续寻找

for (int i = 0; i != nd\_ptr->num\_sub\_nds; ++i) {

if (this->search((nd\_ptr->sub\_nds)[i], d\_)) {

return true;

}

}

return false;

}

return true;

1. 调试分析
   1. 遇到访问NULL位置数据：在删除子节点的时候使用了pop\_back，导致删除了最后一个指针而不是中间的指针，释放空间，最后访问即出错了
      1. 通过调试定位到出错位置，修正
   2. 新节点构造出错：在建立新节点的时候用错构造函数，定位所有新节点构造位置，解决
   3. 使用了Bag类作为节点分裂时候的数据包的类，但由于\_Nm参数不同，导致不同类之间不能访问私有数据成员，之后只能使用BTree\_node类构造bag，并删除bag类
   4. 查找过程一开始使用顺序查找，之后修改为顺序查找
2. 测试结果
   1. 插入阶段测试：插入从2-8，B树节点变化：

[2,]

[2,3,]

[3,][2,][4,]

[3,][2,][4,5,]

[3,5,][2,][4,][6,]

[3,5,][2,][4,][6,7,]

[5,][3,][2,][4,][7,][6,][8,]

* 1. 查找测试：

cout << BT.Search(6) << endl;

输出：1//查找成功

cout << BT.Search(10) << endl;

输出：0//查找失败

1. 总结
   1. 一开始稍微有考虑到整个数据结构的设计，但是后来的不断修改使得结构愈来愈混乱，没有时间写删除操作
   2. 由于一开始之想到要随机访问的问题，在保存节点数据以及子节点指针的时候使用了数组类型的结构保存，但是需要经常使用到插入，删除，应该使用链表结构
   3. 搜索操作应该独立设计类或者使用类模板，并写好仿函数，因为实际的查找操作应当针对key，而不是整个数据结构的查找
   4. 在BTree\_node中的两个insert函数，内容几乎一致，实际上应该能合成一个函数，或者将重复部分分离出来，比如节点分裂时候形成bag的操作