B树学习

- [x] B树的实现，纯内存方式

  基本实现：

  1. 插入操作：节点没满就直接加入值到节点中，满了就分裂成两个节点，把分裂两个节点的中间值和分裂出来的新节点传递给父节点进行操作
  2. 删除操作：
     - 先找到需要删除的key所在的节点位置。
       如果不是叶子节点，就和这个key对应的右子树的最小key交换位置（这样能保持树的定义的同时，把需要删除的key交换到了叶子节点）。
       如果这个节点是叶子节点，直接下一步
     - 然后就是删除叶子节点的一个key。
       直接删除这个key。
       如果删除之后这个叶子节点依旧满足定义，那么删除操作结束。
       如果删除之后不满足定义，那么就先向左右两个兄弟节点借一个key过来以维持定义（实际上更形象的是旋转，当前叶子节点向父亲节点拿一个key，兄弟节点给父节点还一个key）
       如果两个兄弟都没有多出来的key，那么这个叶子节点只能和兄弟节点合并。（两个叶子合并,同时把两个叶子对应在父节点的key拉下来加入到新的节点上）
  3. 查找操作

     dfs，根据定义沿着树走就可以了
  4. 中序遍历：

     中序遍历的结果 恰好是key按大小顺序排列
  5. 逐层打印：

     bfs，依次把后续节点加入到队列中
  6. 代码实现

```
#include<iostream>
#include<queue>
#include<sstream>
#include<string>
#include<algorithm>
using namespace std;
const int maxn = 3;
struct TTNode {
    int key[maxn];//key，多出一个空位，在实现的时候，先添加进去，再进行分裂
    TTNode * next[maxn + 1];
    int cntKey;
    TTNode() :cntKey(0) {
        for (int i = 0; i <= maxn; ++i) {
            next[i] = NULL;
        }
    }

    /*
    * 给节点增加一个key到适当的位置，并更新key的数目,并返回插入的位置
    */
    int addKey(int hyKey) {
        int i;
        for (i = cntKey - 1; i >= 0 && key[i] > hyKey; --i) {
            key[i + 1] = key[i];
        }
        key[i+1] = hyKey;
        ++cntKey;
        return i + 1;
    }

    /*
    * 给节点增加一个子节点，这个子节点的位置下标为pos，pos后面的值后移一个单位（区别于直接赋值）
    */
    void addNext(int pos, TTNode * hyNext) {
        for (int i = min(cntKey + 1 , maxn);i > pos ; --i) {
            next[i] = next[i - 1];
        }
        next[pos] = hyNext;
    }

    /*
    * 判断一个节点是否是叶子节点
    */
    bool isLeaf() {
        return next[0] == NULL;
    }


    /*
    * 删除第一个key值，只在叶子节点进行这个操作
    */
    void removeFirstKey() {
        for (int i = 1; i < cntKey; ++i) {
```

```
            key[i - 1] = key[i];
        }
        --cntKey;
    }

    /*
    * 删除下标位置为j的key
    */
    void removeKey(int j) {
        for (int i = j + 1; i < cntKey; ++i) {
            key[i - 1] = key[i];

        }
        --cntKey;
    }
};

struct TTTree {
    TTNode * root;
    TTTree() :root(NULL) {}

    /*
    *   给 23Tree 增加一个key
    */
    void add(int k) {

        // 23Tree 为空，那么就直接创建一个新的节点
        if (root == NULL) {
            root = new TTNode();
            root->addKey(k);
        }
        else {
            //不为空，就寻找合适的位置，并插入新的值
            int hyVal = 0;
            TTNode * hyPtr = NULL;
            addHelp(root, k,hyPtr, hyVal);
            //树根分裂了
            if (hyPtr != NULL) {
                TTNode * newRoot = new TTNode();
                newRoot->addKey(hyVal);
                newRoot->next[0] = root;
                newRoot->next[1] = hyPtr;
                root = newRoot;
            }
        }
    }

    /*
    * 增加新的key的辅助函数，用来实现递归操作
    */
    void addHelp(TTNode * hyRoot , int &k,TTNode *& retPtr, int & retVa
l) {
        //到达叶子节点
        if (hyRoot->isLeaf()) {
            hyRoot->addKey(k);
```

```
            //如果已经满了，那么就需要分裂节点
            if (hyRoot->cntKey == maxn) {
                splitNode(hyRoot, retPtr, retVal);
            }
            return;
        }
        //不是叶子，就一直走下去
        int i;
        for (i = 0; i < hyRoot->cntKey && hyRoot->key[i] < k; ++i);

        int hyVal;
        TTNode * hyPtr = NULL;
        addHelp(hyRoot->next[i], k, hyPtr, hyVal);
        //如果下一层有分裂节点
        if (hyPtr != NULL) {
            int pos = hyRoot->addKey(hyVal);
            hyRoot->addNext(pos+1, hyPtr);
            if (hyRoot->cntKey == maxn) {
                splitNode(hyRoot, retPtr, retVal);
            }
        }
    }

    /*
    * 分裂节点hyRoot,新的节点保存右边的节点（保存较大的一半值）
    */
    void splitNode(TTNode *hyRoot , TTNode *&retPtr , int &retVal) {
        retPtr = new TTNode();
        //平均分配key和子节点
        int i;
        for (i = maxn - 1; i * 2 > maxn; --i) {
            retPtr->addKey(hyRoot->key[i]);
            retPtr->next[i - maxn / 2] = hyRoot->next[i + 1];
        }
        retPtr->next[0] = hyRoot->next[i+1];
        retVal = hyRoot->key[i];
        hyRoot->cntKey = maxn / 2;
    }

    /*
    * 输出中序遍历的结果，23Tree中序遍历的结果刚好是满足大小顺序输出
    */
    void showInnerTraversal() {
        InnerTraversalHelp(root);
        cout << endl;
    }

    /*
    * 中序遍历辅助函数，用来实现递归操作
    */
    void InnerTraversalHelp(TTNode * hyRoot) {
        if (hyRoot == NULL) return;
        for (int i = 0; i < hyRoot->cntKey; ++i) {
            InnerTraversalHelp(hyRoot->next[i]);
            cout << hyRoot->key[i] << " ";
```

```
        }
        InnerTraversalHelp(hyRoot->next[hyRoot->cntKey]);
    }

    /*
    * 逐层打印出树的节点
    */
    void showByLevel() {
        queue<TTNode *> q;
        q.push(root);
        TTNode * tmp;
        while (!q.empty()){
            tmp = q.front(); q.pop();
            for (int i = 0; i < tmp->cntKey; ++i) {
                cout << tmp->key[i] << " ";
            }
            if (!tmp->isLeaf()) {
                for (int i = 0; i <= tmp->cntKey; ++i) {
                    q.push(tmp->next[i]);
                }
            }
        }
        cout << endl;
    }

    ~TTTree() {
        clear();
    }

    /*
    * 清空树
    */
    void clear() {
        clearHelp(root);
        root = NULL;
    }

    /*
    * 实现递归清除的辅助函数
    */
    void clearHelp(TTNode * hyRoot) {
        if (!hyRoot->isLeaf()) {
            for (int i = 0; i <= hyRoot->cntKey; ++i) {
                clearHelp(hyRoot->next[i]);
            }
        }
        delete hyRoot;
    }

    void remove(int k) {
        int hyMergeId = -1;
        removeHelp(root, k,NULL , hyMergeId);
    }
```

```
    /*
    * 合并 hyRoot 下标为i 和 i + 1的两个节点（还需要把第i个key加入到新的节
点中）
    */
    void MergeNode(TTNode * hyRoot, int mergeId) {
        TTNode *newNode = hyRoot->next[mergeId];
        TTNode *brother = hyRoot->next[mergeId + 1];
        //把父节点的第mergeId个key加入到新的节点中，
        newNode->addKey(hyRoot->key[mergeId]);
        //把兄弟节点的所有key加入到新的节点中
        for (int i = 0; i < brother->cntKey; ++i) {
            newNode->addKey(brother->key[i]);
            newNode->next[newNode->cntKey] = brother->next[i];
        }
        newNode->next[newNode->cntKey] = brother->next[brother-
>cntKey];


        for (int i = mergeId + 1; i < hyRoot->cntKey; ++i) {
            hyRoot->next[i] = hyRoot->next[i + 1];
        }
        hyRoot->removeKey(mergeId);
    }

    /*
    * 当删除key之后，出现key的数目不符合定义的情况，就需要对树进行修复
    */
    void FixTree(TTNode *hyRoot , TTNode *parent ,int & retMergeId) {
        //首先向兄弟结点借
        //同时这里确保，hyRoot一定有父节点,即parent 一定不为 NULL
        //看兄弟节点是否足够，
        int id;
        for (int i = 0; i <= parent->cntKey; ++i) {
            if (parent->next[i] == hyRoot) {
                id = i; break;
            }
        }

        TTNode * leftBrother = (id > 0 ? parent->next[id - 1]:NULL);
        TTNode * rightBrother = (id < maxn ? parent->next[id + 1] : NUL
L);
        if (leftBrother && leftBrother->cntKey > maxn / 2) {
            //向左兄弟借
            hyRoot->addKey(parent->key[id - 1]);
            hyRoot->addNext(0, leftBrother->next[leftBrother->cntKey]);

            parent->key[id] = leftBrother->key[leftBrother-
>cntKey - 1];
            leftBrother->removeKey(leftBrother->cntKey - 1);

        }else if (rightBrother && rightBrother->cntKey > maxn / 2) {
            //向右兄弟借
            hyRoot->addKey(parent->key[id]);
            hyRoot->addNext(hyRoot->cntKey, rightBrother->next[0]);
```

```
            parent->key[id] = rightBrother->key[0];

            rightBrother->removeFirstKey();
            for (int i = 1; i <= rightBrother->cntKey; ++i) {
                rightBrother->next[i - 1] = rightBrother->next[i];
            }
        }
        else {
            //兄弟节点也不够，就合并两个节点（这个操作通过mergeId来提示父节点来进行）
            //这里首先选择和左兄弟合并
            if(id > 0)  retMergeId = id - 1;
            else retMergeId = id;
        }
    }
    /*
    * 寻找hyRoot的最小叶子节点,找到之后交换key，并修复树(通过返回值来判断叶子节点是否需要合并),k是需要交换的key
    * mergeId 是 子节点中，需要合并的两个节点的位置较小的那个，值为-1的时候表示不需要合并。
    */
    void findMinLeafAndFixTree(int & k,TTNode * hyRoot,TTNode * parent,int &retMergeId) {
        int hyMergeId = -1;
        if (hyRoot->isLeaf()) {
            swap(k, hyRoot->key[0]);
            //交换之后，就是考虑删除的叶子key值
            hyRoot->removeFirstKey();

            if (hyRoot->cntKey < maxn / 2) {
                FixTree(hyRoot, parent, retMergeId);
            }
        }
        else {
            findMinLeafAndFixTree(k, hyRoot->next[0], hyRoot, hyMergeId);
        }
        if (hyMergeId != -1) {
            MergeNode(hyRoot, hyMergeId);
            if (hyRoot->cntKey < maxn / 2) {
                FixTree(hyRoot, parent, retMergeId);
            }
        }
    }

    /*
    * 删除值为k的辅助函数，flg表示递归的方式，为0表示寻找k为值，为1表示寻找最小叶子
    */
    void removeHelp(TTNode * hyRoot, int & k,TTNode * parent,int &retMergeId) {
        if (hyRoot == NULL) return;
        int i;
        for (i = 0; i < hyRoot->cntKey && k > hyRoot->key[i]; ++i);
```

```
        int hyMergeId = -1;
        if (i != hyRoot->cntKey && k == hyRoot->key[i]) {
            //找到了k所在的位置，为当前节点 下表为i的key
            //removeHelp(hyRoot->next[i], k,hyRoot, hyMergeId);
            if (!hyRoot->isLeaf()) {
                //当前节点不是叶子，那么就需要和右子树的最小key进行交换,这里
取key的引用来实现交换的目的
                findMinLeafAndFixTree(hyRoot->key[i], hyRoot->next[i + 1]
hyRoot, hyMergeId);
            }
            else {
                //当前节点直接就是叶子，那么直接删除它就是了
                hyRoot->removeKey(i);
                if (hyRoot->cntKey < maxn / 2) {
                    if (hyRoot->cntKey == 0 && parent == NULL) {
                        //树根为空的情况，那么就只需要删除树根即可
                        root = hyRoot->next[0];
                        delete hyRoot;
                    }
                    else FixTree(hyRoot, parent, retMergeId);
                }
            }

        }
        else {
            removeHelp(hyRoot->next[i], k, hyRoot,hyMergeId);
        }

        if (hyMergeId != -1) {
            MergeNode(hyRoot, hyMergeId);
            if (hyRoot->cntKey < maxn / 2) {
                if (hyRoot->cntKey == 0 && parent == NULL) {
                    //树根为空的情况，那么就只需要删除树根即可
                    root = hyRoot->next[0];
                    delete hyRoot;
                }
                else FixTree(hyRoot, parent, retMergeId);
            }
        }
    }

};

int main() {
    TTTree t;
    t.add(80);
    t.add(20);
    t.add(10);
    t.add(30);
    t.add(25);
    t.add(50);
    t.add(90);
    //t.add(91);
    t.remove(30);
    t.showByLevel();
```

```
    t.showInnerTraversal();
    return 0;
}
```

- [x] new delete 操作频繁，用freelist原理优化
  使用 TTNode的一个int类型变量(offset) 作为链表的next(具体见代码)

```
struct TTNode {
    ...
    void * operator new (size_t size) {
        if (freelist == NULL)   return ::new TTNode();
        TTNode * ret = freelist;
        freelist = (TTNode *)freelist->offset;
        return ret;
    }

    void operator delete(void * tmp) {
        if (tmp == NULL)    return;
        TTNode * cur = (TTNode *)tmp;
        cur->offset = (int)freelist;
        freelist = (TTNode *)tmp;
    }

    static TTNode * freelist;
};
TTNode * TTNode::freelist = NULL;
```

- [x] 使用单独的data类型
  在前面的实现，key和data都是一个东西，这里需要分离开来
  具体实现就是生成一个新的key类型，里面分别是key 和 data（合成一个新的类型，这样的目的是为了绑定对应key和data）
  然后把之前key类型修改一下

```cpp
#include<iostream>
#include<queue>
#include<sstream>
#include<string>
#include<algorithm>
using namespace std;
const int maxn = 3;

struct KeyType {
    int key;
    int data;
    bool operator < (const KeyType & a) const {
        return key < a.key;
    }
    bool operator > (const KeyType & a) const {
        return key > a.key;
    }
    bool operator == (const KeyType & a) const {
        return key == a.key;
    }
    friend ostream & operator << (ostream & out, const KeyType & obj) {
        out << "*key:" << obj.key << " data:" << obj.data<<"*";
        return out;
    }
    KeyType(int _key = 0 , int _data = 0):key(_key),data(_data) {}
};

struct TTNode {
    KeyType key[maxn];//key，多出一个空位，在实现的时候，先添加进去，再进行分裂
    TTNode * next[maxn + 1];
    int cntKey;
    TTNode() :cntKey(0) {
        for (int i = 0; i <= maxn; ++i) {
            next[i] = NULL;
        }
    }

    /*
    * 给节点增加一个key到适当的位置，并更新key的数目,并返回插入的位置
    */
    int addKey(KeyType hyKey) {
        int i;
        for (i = cntKey - 1; i >= 0 && key[i] > hyKey; --i) {
            key[i + 1] = key[i];
        }
        key[i + 1] = hyKey;
        ++cntKey;
        return i + 1;
    }

    /*
    * 给节点增加一个子节点，这个子节点的位置下标为pos，pos后面的值后移一个单位
（区别于直接赋值）
    */
```

```
    void addNext(int pos, TTNode * hyNext) {
        for (int i = min(cntKey + 1 , maxn); i > pos ; --i) {
            next[i] = next[i - 1];
        }
        next[pos] = hyNext;
    }

    /*
    * 判断一个节点是否是叶子节点
    */
    bool isLeaf() {
        return next[0] == NULL;
    }


    /*
    * 删除第一个key值，只在叶子节点进行这个操作
    */
    void removeFirstKey() {
        for (int i = 1; i < cntKey; ++i) {
            key[i - 1] = key[i];
        }
        --cntKey;
    }

    /*
    * 删除下标位置为j的key
    */
    void removeKey(int j) {
        for (int i = j + 1; i < cntKey; ++i) {
            key[i - 1] = key[i];

        }
        --cntKey;
    }

    void * operator new (size_t size) {
        if (freelist == NULL)   return ::new TTNode();
        TTNode * ret = freelist;
        freelist = (TTNode *)freelist->cntKey;
        return ret;
    }

    void operator delete(void * tmp) {
        if (tmp == NULL)    return;
        TTNode * cur = (TTNode *)tmp;
        cur->cntKey = (int)freelist;
        freelist = (TTNode *)tmp;
    }

    static TTNode * freelist;
};

TTNode * TTNode::freelist = NULL;
```

```
struct TTTree {
    TTNode * root;
    TTTree() :root(NULL) {}

    /*
    *   给 23Tree 增加一个key
    */
    void add(KeyType k) {

        // 23Tree 为空，那么就直接创建一个新的节点
        if (root == NULL) {
            root = new TTNode();
            root->addKey(k);
        }
        else {
            //不为空，就寻找合适的位置，并插入新的值
            KeyType hyVal;
            TTNode * hyPtr = NULL;
            addHelp(root, k, hyPtr, hyVal);
            //树根分裂了
            if (hyPtr != NULL) {
                TTNode * newRoot = new TTNode();
                newRoot->addKey(hyVal);
                newRoot->next[0] = root;
                newRoot->next[1] = hyPtr;
                root = newRoot;
            }
        }
    }

    /*
    * 增加新的key的辅助函数，用来实现递归操作
    */
    void addHelp(TTNode * hyRoot, KeyType &k, TTNode *& retPtr, KeyType & r
etVal) {
        //到达叶子节点
        if (hyRoot->isLeaf()) {
            hyRoot->addKey(k);
            //如果已经满了，那么就需要分裂节点
            if (hyRoot->cntKey == maxn) {
                splitNode(hyRoot, retPtr, retVal);
            }
            return;
        }
        //不是叶子，就一直走下去
        int i;
        for (i = 0; i < hyRoot->cntKey && hyRoot->key[i] < k; ++i);

        KeyType hyVal;
        TTNode * hyPtr = NULL;
        addHelp(hyRoot->next[i], k, hyPtr, hyVal);
        //如果下一层有分裂节点
        if (hyPtr != NULL) {
            int pos = hyRoot->addKey(hyVal);
```

```
            hyRoot->addNext(pos + 1, hyPtr);
            if (hyRoot->cntKey == maxn) {
                splitNode(hyRoot, retPtr, retVal);
            }
        }
    }

    /*
    * 分裂节点hyRoot,新的节点保存右边的节点（保存较大的一半值）
    */
    void splitNode(TTNode *hyRoot, TTNode *&retPtr, KeyType &retVal) {
        retPtr = new TTNode();
        //平均分配key和子节点
        int i;
        for (i = maxn - 1; i * 2 > maxn; --i) {
            retPtr->addKey(hyRoot->key[i]);
            retPtr->next[i - maxn / 2] = hyRoot->next[i + 1];
        }
        retPtr->next[0] = hyRoot->next[i + 1];
        retVal = hyRoot->key[i];
        hyRoot->cntKey = maxn / 2;
    }

    /*
    * 输出中序遍历的结果，23Tree中序遍历的结果刚好是满足大小顺序输出
    */
    void showInnerTraversal() {
        InnerTraversalHelp(root);
        cout << endl;
    }

    /*
    * 中序遍历辅助函数，用来实现递归操作
    */
    void InnerTraversalHelp(TTNode * hyRoot) {
        if (hyRoot == NULL) return;
        for (int i = 0; i < hyRoot->cntKey; ++i) {
            InnerTraversalHelp(hyRoot->next[i]);
            cout << hyRoot->key[i] << " ";
        }
        InnerTraversalHelp(hyRoot->next[hyRoot->cntKey]);
    }

    /*
    * 逐层打印出树的节点
    */
    void showByLevel() {
        queue<TTNode *> q;
        q.push(root);
        TTNode * tmp;
        while (!q.empty()) {
            tmp = q.front(); q.pop();
            for (int i = 0; i < tmp->cntKey; ++i) {
                cout << tmp->key[i] << " ";
            }
```

```
            if (!tmp->isLeaf()) {
                for (int i = 0; i <= tmp->cntKey; ++i) {
                    q.push(tmp->next[i]);
                }
            }
        }
        cout << endl;
    }

    ~TTTree() {
        clear();
    }

    /*
    * 清空树
    */
    void clear() {
        clearHelp(root);
        root = NULL;
    }

    /*
    * 实现递归清除的辅助函数
    */
    void clearHelp(TTNode * hyRoot) {
        if (!hyRoot->isLeaf()) {
            for (int i = 0; i <= hyRoot->cntKey; ++i) {
                clearHelp(hyRoot->next[i]);
            }
        }
        delete hyRoot;
    }

    void remove(KeyType k) {
        int hyMergeId = -1;
        removeHelp(root, k, NULL, hyMergeId);
    }


    /*
    * 合并 hyRoot 下标为i 和 i + 1的两个节点（还需要把第i个key加入到新的节点中）
    */
    void MergeNode(TTNode * hyRoot, int mergeId) {
        TTNode *newNode = hyRoot->next[mergeId];
        TTNode *brother = hyRoot->next[mergeId + 1];
        //把父节点的第mergeId个key加入到新的节点中，
        newNode->addKey(hyRoot->key[mergeId]);
        //把兄弟节点的所有key加入到新的节点中
        for (int i = 0; i < brother->cntKey; ++i) {
            newNode->addKey(brother->key[i]);
            newNode->next[newNode->cntKey] = brother->next[i];
        }
        newNode->next[newNode->cntKey] = brother->next[brother->cntKey];
```

```
        for (int i = mergeId + 1; i < hyRoot->cntKey; ++i) {
            hyRoot->next[i] = hyRoot->next[i + 1];
        }
        hyRoot->removeKey(mergeId);
    }

    /*
    * 当删除key之后，出现key的数目不符合定义的情况，就需要对树进行修复
    */
    void FixTree(TTNode *hyRoot, TTNode *parent, int & retMergeId) {
        //首先向兄弟结点借
        //同时这里确保，hyRoot一定有父节点,即parent 一定不为 NULL
        //看兄弟节点是否足够，
        int id;
        for (int i = 0; i <= parent->cntKey; ++i) {
            if (parent->next[i] == hyRoot) {
                id = i; break;
            }
        }

        TTNode * leftBrother = (id > 0 ? parent->next[id - 1] : NULL);
        TTNode * rightBrother = (id < maxn ? parent->next[id + 1] : NULL);
        if (leftBrother && leftBrother->cntKey > maxn / 2) {
            //向左兄弟借
            hyRoot->addKey(parent->key[id - 1]);
            hyRoot->addNext(0, leftBrother->next[leftBrother->cntKey]);

            parent->key[id] = leftBrother->key[leftBrother->cntKey - 1];
            leftBrother->removeKey(leftBrother->cntKey - 1);

        }
        else if (rightBrother && rightBrother->cntKey > maxn / 2) {
            //向右兄弟借
            hyRoot->addKey(parent->key[id]);
            hyRoot->addNext(hyRoot->cntKey, rightBrother->next[0]);

            parent->key[id] = rightBrother->key[0];

            rightBrother->removeFirstKey();
            for (int i = 1; i <= rightBrother->cntKey; ++i) {
                rightBrother->next[i - 1] = rightBrother->next[i];
            }
        }
        else {
            //兄弟节点也不够，就合并两个节点（这个操作通过mergeId来提示父节点来进行）
            //这里首先选择和左兄弟合并
            if (id > 0) retMergeId = id - 1;
            else retMergeId = id;
        }
    }
    /*
    * 寻找hyRoot的最小叶子节点,找到之后交换key，并修复树(通过返回值来判断叶子节点是否需要合并),k是需要交换的key
```

```
    * mergeId 是 子节点中，需要合并的两个节点的位置较小的那个，值为-1的时候表示不
需要合并。
    */
    void findMinLeafAndFixTree(KeyType & k, TTNode * hyRoot, TTNode * paren
t, int &retMergeId) {
        int hyMergeId = -1;
        if (hyRoot->isLeaf()) {
            swap(k, hyRoot->key[0]);
            //交换之后，就是考虑删除的叶子key值
            hyRoot->removeFirstKey();

            if (hyRoot->cntKey < maxn / 2) {
                FixTree(hyRoot, parent, retMergeId);
            }
        }
        else {
            findMinLeafAndFixTree(k, hyRoot->next[0], hyRoot, hyMergeId);
        }
        if (hyMergeId != -1) {
            MergeNode(hyRoot, hyMergeId);
            if (hyRoot->cntKey < maxn / 2) {
                FixTree(hyRoot, parent, retMergeId);
            }
        }
    }

    /*
    * 删除值为k的辅助函数，flg表示递归的方式，为0表示寻找k为值，为1表示寻找最小叶
子
    */
    void removeHelp(TTNode * hyRoot, KeyType & k, TTNode * parent, int &ret
MergeId) {
        if (hyRoot == NULL) return;
        int i;
        for (i = 0; i < hyRoot->cntKey && k > hyRoot->key[i]; ++i);

        int hyMergeId = -1;
        if (i != hyRoot->cntKey && k == hyRoot->key[i]) {
            //找到了k所在的位置，为当前节点 下表为i的key
            //removeHelp(hyRoot->next[i], k,hyRoot, hyMergeId);
            if (!hyRoot->isLeaf()) {
                //当前节点不是叶子，那么就需要和右子树的最小key进行交换,这里取ke
y的引用来实现交换的目的
                findMinLeafAndFixTree(hyRoot->key[i], hyRoot-
>next[i + 1], hyRoot, hyMergeId);
            }
            else {
                //当前节点直接就是叶子，那么直接删除它就是了
                hyRoot->removeKey(i);
                if (hyRoot->cntKey < maxn / 2) {
                    if (hyRoot->cntKey == 0 && parent == NULL) {
                        //树根为空的情况，那么就只需要删除树根即可
                        root = hyRoot->next[0];
                        delete hyRoot;
                    }
```

```
                    else FixTree(hyRoot, parent, retMergeId);
                }
            }

        }
        else {
            removeHelp(hyRoot->next[i], k, hyRoot, hyMergeId);
        }

        if (hyMergeId != -1) {
            MergeNode(hyRoot, hyMergeId);
            if (hyRoot->cntKey < maxn / 2) {
                if (hyRoot->cntKey == 0 && parent == NULL) {
                    //树根为空的情况，那么就只需要删除树根即可
                    root = hyRoot->next[0];
                    delete hyRoot;
                }
                else FixTree(hyRoot, parent, retMergeId);
            }
        }
    }
};
```

- [x] 把纯内存的B树改造成文件操作
  根节点常驻内存，每次访问子节点才在内存中创建一个节点存放文件读取的内容，每次访问完子节点就清除内存
  - [x] 单key，不能重复版本

```cpp
#include<iostream>
#include<queue>
#include<sstream>
#include<string>
#include<algorithm>
#include<fstream>
using namespace std;
const int maxn = 3;
struct TTNode {
    int offset; //当前节点所在的偏移
    int key[maxn];//key，多出一个空位，在实现的时候，先添加进去，再进行分裂
    int next[maxn + 1];//子节点的偏移
    int cntKey;
    TTNode() :cntKey(0) {
        for (int i = 0; i <= maxn; ++i) {
            next[i] = 0;
        }
    }

    /*
    * 给节点增加一个key到适当的位置，并更新key的数目,并返回插入的位置
    */
    int addKey(int hyKey) {
        int i;
        for (i = cntKey - 1; i >= 0 && key[i] > hyKey; --i) {
            key[i + 1] = key[i];
        }
        key[i + 1] = hyKey;
        ++cntKey;
        return i + 1;
    }

    /*
    * 给节点增加一个子节点，这个子节点的位置下标为pos，pos后面的值后移一个单位（区别于直接赋值）
    */
    void addNext(int pos, int hyNext) {
        for (int i = min(cntKey + 1 , maxn); i > pos ; --i) {
            next[i] = next[i - 1];
        }
        next[pos] = hyNext;
    }

    /*
    * 判断一个节点是否是叶子节点
    */
    bool isLeaf() {
        return next[0] == 0;
    }


    /*
    * 删除第一个key值，只在叶子节点进行这个操作
    */
```

```
    void removeFirstKey() {
        for (int i = 1; i < cntKey; ++i) {
            key[i - 1] = key[i];
        }
        --cntKey;
    }

    /*
    * 删除下标位置为j的key
    */
    void removeKey(int j) {
        for (int i = j + 1; i < cntKey; ++i) {
            key[i - 1] = key[i];

        }
        --cntKey;
    }
};

const char * dataFileName = "hyDataBase.dat";
struct TTTree {
    TTNode * root;
    fstream file;
    TTTree() :root(NULL) {}

    /*
    * 把节点信息写入到文件中
    */
    void writeNodeToFile(TTNode * hyRoot) {
        file.seekp(hyRoot->offset,ios::beg);
        file.write((char *)hyRoot, sizeof(TTNode));
    }

    /*
    * 分别在内存和文件中申请空间，并返回节点的内存地址和偏移
    */
    TTNode * newNode() {
        //在内存中申请节点
        TTNode * hyRoot = new TTNode();

        //在文件中申请存放节点的空间
        file.seekp(1, ios::end);
        hyRoot->offset = file.tellg();
        file.seekp(sizeof(TTNode), ios::cur);
        //只有写入东西，文件的大小才改变，所以在最后一个节点后面写入信息。
        file.write("1", 1);

        return hyRoot;
    }

    /*
    * 在文件中读取一个节点的信息，参数是节点的偏移，返回节点的地址
    */
    TTNode *readNode(int offset) {
        file.seekg(offset, ios::beg);
```

```
        TTNode * hyRoot = new TTNode();
        file.read((char *)hyRoot, sizeof(TTNode));
        return hyRoot;
    }

    /*
    *   给 23Tree 增加一个key
    */
    void add(int k) {

        // 23Tree 为空，那么就直接创建一个新的节点
        if (root == NULL) {
            //数据为空的时候就需要创建一个根节点

            //申请节点
            root = newNode();

            //保存节点偏移到数据文件首位
            file.seekp(0, ios::beg);
            file.write((const char *)&(root->offset), sizeof(int));

            //写入k到节点内存中
            root->addKey(k);
            ////////////注意 ，  到目前位置，根节点的内存信息都没有保存到数据文件中
        }
        else {
            //不为空，就寻找合适的位置，并插入新的值
            int hyVal = 0;
            TTNode * hyPtr = NULL;
            addHelp(root, k, hyPtr, hyVal);
            //树根分裂了
            if (hyPtr != NULL) {
                TTNode * newRoot;
                newRoot = newNode();

                newRoot->addKey(hyVal);

                //更新新节点的子节点偏移
                newRoot->next[0] = root->offset;
                newRoot->next[1] = hyPtr->offset;

                writeNodeToFile(root);
                writeNodeToFile(hyPtr);
                delete root;
                delete hyPtr;

                root = newRoot;

                //保存节点偏移到数据文件首位
                file.seekp(0, ios::beg);
                file.write((const char *)&(root->offset), sizeof(int));
            }
        }
    }
```

```
    /*
    * 增加新的key的辅助函数，用来实现递归操作
    */
    void addHelp(TTNode * hyRoot, int &k, TTNode *& retPtr, int & retVa
l) {
        //到达叶子节点
        if (hyRoot->isLeaf()) {
            hyRoot->addKey(k);
            //如果已经满了，那么就需要分裂节点
            if (hyRoot->cntKey == maxn) {
                splitNode(hyRoot, retPtr, retVal);
            }
            return;
        }
        //不是叶子，就一直走下去
        int i;
        for (i = 0; i < hyRoot->cntKey && hyRoot->key[i] < k; ++i);

        int hyVal;
        TTNode * hyPtr = NULL;
        TTNode * hyNext = readNode(hyRoot->next[i]);
        addHelp(hyNext, k, hyPtr, hyVal);
        writeNodeToFile(hyNext);
        delete hyNext;

        //如果下一层有分裂节点
        if (hyPtr != NULL) {
            int pos = hyRoot->addKey(hyVal);
            hyRoot->addNext(pos + 1, hyPtr->offset);
            if (hyRoot->cntKey == maxn) {
                splitNode(hyRoot, retPtr, retVal);
            }
            writeNodeToFile(hyPtr);
            delete hyPtr;
        }
    }

    /*
    * 分裂节点hyRoot,新的节点保存右边的节点（保存较大的一半值）
    */
    void splitNode(TTNode *hyRoot, TTNode *&retPtr, int &retVal) {
        retPtr = newNode();
        //平均分配key和子节点
        int i;
        for (i = maxn - 1; i * 2 > maxn; --i) {
            retPtr->addKey(hyRoot->key[i]);
            retPtr->next[i - maxn / 2] = hyRoot->next[i + 1];
        }
        retPtr->next[0] = hyRoot->next[i + 1];
        retVal = hyRoot->key[i];
        hyRoot->cntKey = maxn / 2;
    }

    /*
```

```
* 输出中序遍历的结果，23Tree中序遍历的结果刚好是满足大小顺序输出
*/
void showInnerTraversal() {
    if (root == NULL)   return;
    InnerTraversalHelp(root);
    cout << endl;
}

/*
* 中序遍历辅助函数，用来实现递归操作
*/
void InnerTraversalHelp(TTNode * hyRoot) {
    if (hyRoot == NULL) return;
    for (int i = 0; i < hyRoot->cntKey; ++i) {
        TTNode * tmpNode = readNode(hyRoot->next[i]);
        InnerTraversalHelp(tmpNode);
        cout << hyRoot->key[i] << " ";
        delete tmpNode;
    }
    TTNode * tmpNode = readNode(hyRoot->next[hyRoot->cntKey]);
    InnerTraversalHelp(tmpNode);
    delete tmpNode;
}

/*
* 逐层打印出树的节点
*/
void showByLevel() {
    if (root == NULL)   return;
    queue<TTNode *> q;
    q.push(root);
    TTNode * tmp;
    while (!q.empty()) {
        tmp = q.front(); q.pop();
        for (int i = 0; i < tmp->cntKey; ++i) {
            cout << tmp->key[i] << " ";
        }
        if (!tmp->isLeaf()) {
            for (int i = 0; i <= tmp->cntKey; ++i) {
                TTNode * tmpNode = readNode(tmp->next[i]);
                q.push(tmpNode);
            }
        }
        if (tmp != root)    delete tmp;
    }
    cout << endl;
}

~TTTree() {
    //程序退出之前保存根节点的信息
    if(root) writeNodeToFile(root);
    //清空内存
    delete root;
    file.close();
}
```

```
    void remove(int k) {
        int hyMergeId = -1;
        removeHelp(root, k, NULL, hyMergeId);
    }


    /*
    * 合并 hyRoot 下标为i 和 i + 1的两个节点（还需要把第i个key加入到新的节点中）
    */
    void MergeNode(TTNode * hyRoot, int mergeId) {
        TTNode *newNode = readNode(hyRoot->next[mergeId]);
        TTNode *brother = readNode(hyRoot->next[mergeId + 1]);
        //把父节点的第mergeId个key加入到新的节点中，
        newNode->addKey(hyRoot->key[mergeId]);
        //把兄弟节点的所有key加入到新的节点中
        for (int i = 0; i < brother->cntKey; ++i) {
            newNode->addKey(brother->key[i]);
            newNode->next[newNode->cntKey] = brother->next[i];
        }
        newNode->next[newNode->cntKey] = brother->next[brother->cntKey];

        for (int i = mergeId + 1; i < hyRoot->cntKey; ++i) {
            hyRoot->next[i] = hyRoot->next[i + 1];
        }
        hyRoot->removeKey(mergeId);

        writeNodeToFile(newNode);
        delete newNode;
        delete brother;
    }

    /*
    * 当删除key之后，出现key的数目不符合定义的情况，就需要对树进行修复
    */
    void FixTree(TTNode *hyRoot, TTNode *parent, int & retMergeId) {
        //首先向兄弟结点借
        //同时这里确保，hyRoot一定有父节点,即parent 一定不为 NULL
        //看兄弟节点是否足够，
        int id;
        for (int i = 0; i <= parent->cntKey; ++i) {
            if (parent->next[i] == hyRoot->offset) {
                id = i; break;
            }
        }

        TTNode * leftBrother = (id > 0 ? readNode(parent->next[id - 1]): NULL);
        TTNode * rightBrother = (id < maxn ? readNode(parent->next[id + 1]) : NULL);
        if (leftBrother && leftBrother->cntKey > maxn / 2) {
            //向左兄弟借
            hyRoot->addKey(parent->key[id - 1]);
```

```
            hyRoot->addNext(0, leftBrother->next[leftBrother->cntKey]);

            parent->key[id] = leftBrother->key[leftBrother-
>cntKey - 1];
            leftBrother->removeKey(leftBrother->cntKey - 1);

        }
        else if (rightBrother && rightBrother->cntKey > maxn / 2) {
            //向右兄弟借
            hyRoot->addKey(parent->key[id]);
            hyRoot->addNext(hyRoot->cntKey, rightBrother->next[0]);

            parent->key[id] = rightBrother->key[0];

            rightBrother->removeFirstKey();
            for (int i = 1; i <= rightBrother->cntKey; ++i) {
                rightBrother->next[i - 1] = rightBrother->next[i];
            }
        }
        else {
            //兄弟节点也不够，就合并两个节点（这个操作通过mergeId来提示父节
点来进行）
            //这里首先选择和左兄弟合并
            if (id > 0) retMergeId = id - 1;
            else retMergeId = id;
        }

        if (leftBrother)    writeNodeToFile(leftBrother);
        if (rightBrother) writeNodeToFile(rightBrother);
        delete leftBrother;
        delete rightBrother;
    }
    /*
    * 寻找hyRoot的最小叶子节点,找到之后交换key，并修复树(通过返回值来判断叶
子节点是否需要合并),k是需要交换的key
    * mergeId 是 子节点中，需要合并的两个节点的位置较小的那个，值为-1的时候
表示不需要合并。
    */
    void findMinLeafAndFixTree(int & k, TTNode * hyRoot, TTNode * paren
t, int &retMergeId) {
        int hyMergeId = -1;
        if (hyRoot->isLeaf()) {
            swap(k, hyRoot->key[0]);
            //交换之后，就是考虑删除的叶子key值
            hyRoot->removeFirstKey();

            if (hyRoot->cntKey < maxn / 2) {
                FixTree(hyRoot, parent, retMergeId);
            }
        }
        else {
            TTNode * hyNext = readNode(hyRoot->next[0]);
            findMinLeafAndFixTree(k, hyNext, hyRoot, hyMergeId);
            writeNodeToFile(hyNext);
            delete hyNext;
```

```
        }
        if (hyMergeId != -1) {
            MergeNode(hyRoot, hyMergeId);
            if (hyRoot->cntKey < maxn / 2) {
                FixTree(hyRoot, parent, retMergeId);
            }
        }
    }

    /*
    * 删除值为k的辅助函数，flg表示递归的方式，为0表示寻找k为值，为1表示寻找最小叶子
    */
    void removeHelp(TTNode * hyRoot, int & k, TTNode * parent, int &retMergeId) {
        if (hyRoot == NULL) return;
        int i;
        for (i = 0; i < hyRoot->cntKey && k > hyRoot->key[i]; ++i);
        TTNode * hyNext;
        int hyMergeId = -1;
        if (i != hyRoot->cntKey && k == hyRoot->key[i]) {
            //找到了k所在的位置，为当前节点 下表为i的key
            //hyNext = readNode(hyRoot->next[i]);
            //removeHelp(hyNext, k, hyRoot, hyMergeId);
            //writeNodeToFile(hyNext);
            //delete hyNext;

            if (!hyRoot->isLeaf()) {
                //当前节点不是叶子，那么就需要和右子树的最小key进行交换,这里取key的引用来实现交换的目的
                hyNext = readNode(hyRoot->next[i + 1]);
                findMinLeafAndFixTree(hyRoot->key[i], hyNext, hyRoot, hyMergeId);
                writeNodeToFile(hyNext);
                delete hyNext;
            }
            else {
                //当前节点直接就是叶子，那么直接删除它就是了
                hyRoot->removeKey(i);
                if (hyRoot->cntKey < maxn / 2) {
                    if (hyRoot->cntKey == 0 && parent == NULL) {
                        //树根为空的情况，那么就只需要删除树根即可
                        root = readNode(hyRoot->next[0]);
                        delete hyRoot;

                        //保存节点偏移到数据文件首位
                        file.seekp(0, ios::beg);
                        file.write((const char *)&(root->offset), sizeof(int));
                    }
                    else FixTree(hyRoot, parent, retMergeId);
                }
            }

        }
```

```
        else {
            if (hyRoot->isLeaf())   return;
            hyNext = readNode(hyRoot->next[i]);
            removeHelp(hyNext, k, hyRoot, hyMergeId);
            writeNodeToFile(hyNext);
            delete hyNext;
        }

        if (hyMergeId != -1) {
            MergeNode(hyRoot, hyMergeId);
            if (hyRoot->cntKey < maxn / 2) {
                if (hyRoot->cntKey == 0 && parent == NULL) {
                    //树根为空的情况，那么就只需要删除树根即可
                    root = readNode(hyRoot->next[0]);
                    delete hyRoot;
                    //保存节点偏移到数据文件首位
                    file.seekp(0, ios::beg);
                    file.write((const char *)&(root->offset), sizeof(in
t));
                }
                else FixTree(hyRoot, parent, retMergeId);
            }
        }
    }

    /*
    * 初始化数据库，数据库文件开头4个字节，表示根节点存放的偏移，后面就是每个
节点的信息，包括key值和next偏移等
    * 这里定义数据文件的最后一个节点后面有一个字节的数据作为标记。
    */
    void iniDataBase() {
        file.open(dataFileName, ios::in | ios::out | ios::binary | io
s::trunc);
        int rootOffset = 0;
        file.write((const char *)&rootOffset, sizeof(int));
        file.write("1", 1);
        file.close();
    }

    /*
    * 从数据库中读取数据进行必要的初始化
    */
    void iniData() {
        int rootOffset;
        file.open(dataFileName, ios::in | ios::out | ios::binary);
        file.seekg(0, ios::beg);
        file.read((char *)&rootOffset, sizeof(int));

        //读取根节点信息到内存中
        if (rootOffset == 0) {
            root = NULL;
        }else{
            root = readNode(rootOffset);
        }
    }
```

```
};

int main() {
    TTTree t;
    t.iniDataBase();
    t.iniData();
    t.add(60);
    t.add(80);
    t.showByLevel();
    t.remove(60);
    t.showByLevel();
    return 0;
}
```

- [ ] 解决B树插入重复key的问题

  待用方案：

  - [ ] 对于每个key使用溢出块，每次出现重复的key，就往溢出块后面叠加

    具体实现：

    把每个key的data改成指向dataBlock的一个指针

    dataBlock 实际上是一个链表，里面存放一个data数组和指向下一个dataBlock的指针

    每次插入,如果key已经存在，那么就直接增加data进去

    每次删除，先从dataBlock删除data，当数据为空的时候才删除节点。

    注：下面代码没有对 `DataBlock` 进行清理
  - [ ] 增加一个隐藏的属性，强行把重复的key，通过进一步转换，变成不重复的key
  - [ ] 重定义B树的节点，使得B树的定义能满足重复key的要求

    这里节点中的key重定义为 对应的子节点中，第一次出现的key，而不是原来的对应的子节点中，最小的key
  - [x] 直接不作修改，只是在查找的时候修改一下。

    这样节点的定义就变成的 $lkey \quad fa \quad rkey$,那么查找的时候左右两棵子树都要查找。

```
struct TTNode{
    ...
    /*
    * 给节点增加一个key到适当的位置，并更新key的数目,并返回插入的位置
    */
    int addKey(KeyType hyKey) {
        int i;
        for (i = cntKey - 1; i >= 0 && (key[i] > hyKey || key[i] == hyK
ey); --i) {
            key[i + 1] = key[i];
        }
        key[i + 1] = hyKey;
        ++cntKey;
        return i + 1;
    }

    /*
    * 给节点增加一个子节点，这个子节点的位置下标为pos，pos后面的值后移一个单
位（区别于直接赋值）
    */
    void addNext(int pos, int hyNext) {
        for (int i = min(cntKey + 1 , maxn); i > pos; --i) {
            next[i] = next[i - 1];
        }
        next[pos] = hyNext;
    }
};

struct TTTree{
    ...
    void search(int k) {
        searchHelp(root, k);
    }
    /*
    * 对于search查找到的节点进行一定的处理
    */
    void dataConsume(DataType &data) {
        cout << data.data_1 << endl;
    }
    //查找key对应的项目的data，并对满足这个key的所有data调用func这个函数。
    void searchHelp(TTNode * hyRoot, int &k) {
        /* 输出查找的路径
        cout << "  [  ";
        for (int i = 0; i < hyRoot->cntKey; ++i) {
            cout << hyRoot->key[i] << " ";
        }
        cout << "  ]  " << endl;
        */

        //到达叶子节点
        if (hyRoot->isLeaf()) {
            int i;
            for (i = 0; i < hyRoot->cntKey && hyRoot->key[i] < k; ++i);
            for (; i < hyRoot->cntKey && hyRoot->key[i] == k; ++i) {
```

```
                dataConsume(hyRoot->key[i].data);
            }
            //遍历所有key。
            return;
        }
        //不是叶子，就一直走下去
        int i;
        for (i = 0; i < hyRoot->cntKey && hyRoot->key[i] < k ; ++i);

        //枚举所有的key
        for (; i < hyRoot->cntKey && hyRoot->key[i] == k; ++i) {
            dataConsume(hyRoot->key[i].data);
            KeyType hyVal;
            TTNode * hyPtr = NULL;
            TTNode * hyNext = readNode(hyRoot->next[i]);
            searchHelp(hyNext, k);
            writeNodeToFile(hyNext);
            delete hyNext;
        }

        KeyType hyVal;
        TTNode * hyPtr = NULL;
        TTNode * hyNext = readNode(hyRoot->next[i]);
        searchHelp(hyNext, k);
        writeNodeToFile(hyNext);
        delete hyNext;
    }
};
```

- [ ] 查询资料，看内存映射文件 是否能用来优化
- [ ] 建立缓冲区，进一步优化IO速度
  也就是对最近访问过的节点，先不删除，当再次读取到的时候，直接调用。
  实际上只需要对freelist进一步修改，每次new之前，先看看freelist是否已经存在需要访问的节点，存在就直接返回，不存在再读取
  这里缓冲区定义为 保存最近访问过的若干的节点
  一句话：把freelist改成启发式链表
- [x] 逐层换行打印一棵树

```
void showByLevel2() {
    if (root == NULL)   return;
    queue<pair<TTNode *,int>> q;
    q.push(make_pair(root,1));
    int preLevel = 1,curLevel = 1;
    TTNode * tmp;
    while (!q.empty()) {
        tmp = q.front().first;
        curLevel = q.front().second;q.pop();
        if (curLevel != preLevel)   cout << endl;

        cout << "  [  ";
        for (int i = 0; i < tmp->cntKey; ++i) {
            cout << tmp->key[i] << " ";
        }
        cout << "  ]  ";

        if (!tmp->isLeaf()) {
            for (int i = 0; i <= tmp->cntKey; ++i) {
                TTNode * tmpNode = readNode(tmp->next[i]);
                q.push(make_pair(tmpNode, curLevel + 1));
            }
        }
        if (tmp != root)    delete tmp;
        preLevel = curLevel;
    }
    cout << endl;
}
```