B+树的模拟（删除）

报告者：数据科学与计算机学院 18340228 周思宇
合作者：数据科学与计算机学院 18340228 张德龙

一、实验概述

1. **B+树定义**：阶为 M，且定义了 L 的大小的的 B+树是一颗具有下列特性的 M 叉树：
    1. 数据项存储在树叶上
    2. 非叶节点存储知道 M-1 个关键字以指示搜索的方向：关键字 i 代表子树 i+1 中的最小的关键字。
    3. 树的根不是树叶，且其儿子数在 2 和 M 之间。
    4. 除根外，所有非叶节点的儿子数在 Ceiling(M/2)和 M 之间。
    5. 所有的树叶都在相同的深度上，并且每片树叶拥有的数据项其个数在 Ceiling((L-1)/2)和 L-1 之间。
2. **实验目的**：通过对 B+树的模拟掌握 B+树的逻辑结构和实现方法，并能够对 B+树的基本操作如插入、删除和查找的时间复杂度有大概的估计。
3. **实验限制**：一个外部存储块的大小为 40Bytes。规定关键字大小为 4Bytes，地址大小为 4Bytes，记录数据信息大小为 8Bytes。B+树的 M 值设置为 5，L 值设置为 4。
4. **B+树设计思想**：bottom up。

二、B+树设计（源代码各个主模块及其清单）

1. **B+树数据结构设计**：

叶子节点：

```
struct Leaf
{
    //此构造函数仅用于测试 无意义
    Leaf( data_type a, data_type b )
    {
        data[0] = a;
        data[1] = b;
        count = 2;
    }
    Leaf() {}
    char count;
    data_type data[L];
};
```

内部节点：

```
enum NodeTag:char { LEAF, NODE };   //用于分辨是这个结点是否连接树叶
struct Node
{
    char count;
    NodeTag tag;
    key_type key[order-1];
    union Branch
    {
        Node* node[order];
        Leaf* leaf[order];
    };
    Branch branch;  //结点所存储的分支
    Node( NodeTag t = NODE ): tag(t) {}
};

union ptr
{
    Node* node;
    Leaf* leaf;
}; //一个可以是叶子又可以是内点的union指针
```

2. **存储结构大小分析：**

叶子节点：叶子节点包括一个数据类型为 char 的计数变量 count，占用 1 个字节，存储的数据一个存储数据为 8 个字节，所以最多能放 4 个存储数据，再加上 1 个 count 计数变量，总大小小于 40 字节。

内部节点：对于一个阶数为 5 的 B+树而言，存储 5 个指针和 4 个关键词，共 36 个字节，由于 count 为 char 类型变量，标记为枚举类型，这两个剩下的数据相加小于 4 个字节，但是由于系统对齐的原因，这两个数据算作 4 个字节，所以一共 40 个字节。当然，阶数可以调整，如果阶数小于 5，那么 node 节点大小小于 40 字节。

3. **B+树功能设计：**

1. 用户能够调用的函数：

```
B_Tree(): root(nullptr), accessTime(0)
{
    init(); //默认初始化 固定的初始化值
}
B_Tree(data_type data[(L/2)*2]): root(nullptr), accessTime(0)
{
    init(data); //采用传入的参数初始化B+树
}
~B_Tree()
{
    clear();
}
```

```
/*
    功能：利用输入的(L/2)*2个数据初始化形成一颗最小的B+树
    前提：key_type重载了>或者是能够直接比较
    结果：形成由一个根和两片叶子组成的B+树
*/
void init(data_type data [(L/2)*2])
```

```
/*
    功能：查找B树中是否有含键值为x的数据
    结果：若包含则返回true 否则返回false
*/
bool contains( const key_type& x ) const;
/*
    功能：向B树中插入data x
    结果：若成功则返回true，重复返回false
*/
bool insert( const data_type& x );
/*
    功能：在B树中删除data x
    结果：若成功则返回true，若树中无x则返回false
*/
bool erase( const data_type& x );
```

```
/*
    功能：打印B树和总共对磁盘的访问次数
*/
void display() const
{
    display(root);
    displayAccessTime();
}
/*
    功能：释放B+树
*/
void clear()
{
    if( root != nullptr )
        clear(root);
    accessTime = 0;
}
```

2. 设计的内部函数：

```
//  int size;
/*
    功能：对内结点插入键值为key的数据data，若这个操作使得结点分裂，产生的新键放入newKey中，产生的新分支放入newBranch中
    结果：若插入成功则返回true 若重复则返回false
*/
State insert( Node*& n, const key_type& key, const data_type& data, key_type& newKey, Node*& newBranch);
/*
    功能：对叶子插入键值为key的数据data，若这个操作使得叶子分裂，产生的新键放入newKey中，产生的新分支放入newBranch中
    结果：若插入成功则返回true 若重复则返回false
*/
State insert( Leaf*& l, const key_type& key, const data_type& data, key_type& newKey, Leaf*& newBranch);
/*
    功能：向内结点n插入关键字key
    前提：n未满
*/
void insert_key( Node* n, const key_type& newkey, void* newBranch, size_t pos );
/*
    功能：向叶子l插入数据data
    前提：l未满
*/
void insert_data( Leaf* l, const data_type& newdata, size_t pos );
/*
    功能：在内结点n中的关键字中寻找关键字x应当出现的位置
    前提：x重载了<运算符
*/
size_t findPos( Node* n, const key_type& x ) const;
```

```
/*
    功能：在叶子l中的数据中寻找数据d应当出现的位置
    前提：x重载了<运算符
*/
size_t findPos( Leaf* l, const data_type& d ) const;
/*
    功能：通过插入cur_newKey和cur_newBranch使得结点n分裂 产生的新键置入newKey中 产生的新分支置入newBranch中
    前提：n已满
*/
void split( Node* n, const key_type& cur_newKey, Node* cur_newBranch, size_t pos, key_type& newKey, Node*& newBranch);
/*
    功能：通过插入data使得结点n分裂 产生的新键置入newKey中 产生的新分支置入newBranch中
    前提：l已满
*/
void split( Leaf* l, const data_type& data, size_t pos, key_type& newKey, Leaf*& newBranch );
/*
    功能：查找n和其包含的分支中是否含有键为x的data
    前提：x重载了==运算符
    结果：若包含则返回true 否则返回false
*/
bool contains( Node* n, const key_type& x ) const;
/*
    功能：查找叶子l是否含有键为x的data
    前提：x重载了==运算符
    结果：若包含则返回true 否则返回false
*/
bool contains( Leaf* l, const key_type& x ) const;
```

```
/*
    功能：打印结点n及其所包含分支
*/
void display( Node* n, int indent = 0 ) const;
/*
    功能：打印叶子l中的数据
*/
void display( Leaf* l, int indent = 0 ) const;

/*
    功能：从树中删除叶子中的数据y
    结果：若删除成功则返回true,删除失败返回false
*/
bool erase(Node*& n, const data_type& y, key_type& yplus);
/*
    功能：给定叶节点指针current,删除该叶节点内的数据x
    结果：若删除成功则返回true,删除失败返回false
*/
bool remove_inleaf(Leaf* current, const data_type& x);
/*
    功能：删除单个数据后在叶节点和内部节点之间依据B+树规则对树进行重新调整
    前提：有叶子的数据数目小于L/2
*/
void restore_inleaf(Node* current, const int& position);
```

```
/*
    功能：删除单个数据在内部节点和内部节点之间依据B+树规则对树进行重新调整
    前提：有内部节点关键词数目小于(order-1)/2
*/
void restore_innode(Node* current, const int& position);
/*
    功能：父亲的一个关键字下拉，右兄弟一个关键字上提，实现左旋转
    前提：右兄弟关键词数据大于(order-1)/2
*/
void movenode_left(Node* current, const int& position);
/*
    功能：父亲的一个关键字下拉，左兄弟一个关键字上提，实现右旋转
    前提：左兄弟关键词数据大于(order-1)/2
*/
void movenode_right(Node* current, const int& position);
/*
    功能：合并父亲节点和有current和keyposition确定的两个子节点
    前提：左右兄弟关键词数目都小于(order-1)/2
*/
void movenode_combine(Node* current, const int& keyposition);
/*
    功能：由current和position确定的叶节点从其右兄弟叶子借得一个数据，并调整关键字
    前提：右兄弟的数据数目大于L/2
*/
void moveleaf_left(Node* current, const int& position);
```

```
/*
    功能：由current和position确定的叶节点从其左兄弟叶子借得一个数据，并调整关键字
    前提：左兄弟的数据数目大于L/2
*/
void moveleaf_right(Node * current, const int& position);
/*
    功能：删除一个关键字，合并其原来的两个叶子节点
    前提：左右叶子数据数目都小于L/2
*/
void moveleaf_combine(Node* current, const int& keyposition);
/*
    功能：采用后序遍历释放结点n及其分支
*/
void clear( Node*& n );
/*
    功能：采用后序遍历释放叶子l
*/
inline void clear( Leaf*& l )
{
    delete l;
    l = nullptr;
}
```

虽然上面的可能有点冗长，但是仔细看看就可以了解我们对于整个类的设计和实现的思想。

三、B+树插入删除：

报告分两个人写，我实现的主要是删除的部分，插入操作可能比较简便。

1. B+树的插入：
    1. 从上向下插入，如果发现叶子节点有位置，那么直接插入。
    2. 如果叶子节点没有位置，那么分裂叶子节点。
    3. 如果存储叶子节点指针的内部节点的叶子个数超过了最大允许的节点数，那么分裂。
    4. 逐层向上传递信息，递归操作。
2. B+树的删除算法：
    1. **向下递归寻找到要删除的元素，如果找到要删除的元素那么就直接删除。**

    递归删除总函数：默认递归后需要重新访问当前节点。

```
template<typename data_type, typename key_type, typename getKey, int order, int L>
bool B_Tree<data_type, key_type, getKey, order, L>::erase(Node*& current, const data_type& y, key_type& yplus)
{
    getKey getkey;
    key_type x = getkey(y);
    increaseAccessTime(); //访问磁盘
    if(current -> tag == LEAF)
    {
        int position = findPos(current, x);
        increaseAccessTime(); //访问磁盘
        bool to_return = remove_inleaf(current -> branch.leaf[position],y);
        key_type temp = getkey(current -> branch.leaf[position] -> data[0]);
        yplus = temp;
        if(position)
            current -> key[position-1] = temp;
        if(current -> branch.leaf[position] -> count < L/2)
            restore_inleaf(current, position);
        return to_return;
    }
    int position = findPos(current, x);
    bool to_return = erase(current -> branch.node[position], y, yplus);
    // if(current == root)
    //     cout<< current -> branch.node[position] -> count;
    if(position&&x==current->key[position-1])
        current -> key[position-1] = yplus;
    increaseAccessTime(); //访问磁盘
    increaseAccessTime(); //访问磁盘
    if(current -> branch.node[position] -> count < (order-1)/2)
        restore_innode(current, position);
    return to_return;
}
```

对于 restore 函数，用于调用以下方法对删除后的 B+树进行调整。

2. 对于内部节点为包含叶子节点指针的内部节点：

1. 判断删除元素所在的叶子节点内元素个数是否小于允许的最小值，若小于，那么就执行以下操作，否则继续返回上一级。

2. 判断删除元素所在的叶子节点的左右兄弟叶子节点是否有多余的元素，即能够从兄弟叶子节点借一个元素过来同时剩下的元素个数又不小于 B+树允许的最小值。若是借不到，则进行步骤 3。若能够借到，则返回上一级。

使用的左旋转函数（右旋转函数同理）：左旋转函数需要额外访问左右子节点。

```
//该函数处理含有叶结点指针的节点的左旋转
template<typename data_type, typename key_type, typename getKey, int order, int L>
void B_Tree<data_type, key_type, getKey, order, L>::moveleaf_left(Node* current, const int& position)
{
    increaseAccessTime(); //访问磁盘
    getKey getkey;
    Leaf* right = current -> branch.leaf[position+1];
    Leaf* left = current -> branch.leaf[position];
    current -> key[position] = getkey(right -> data[1]);
    data_type temp = right -> data[0];
    right -> count--;
    for(int i = 0; i<right->count; i++)
        right -> data[i] = right -> data[i+1];
    char& tempcount = left -> count;
    left -> data[tempcount++] = temp;
}
```

3. 将删除元素所在的叶子节点剩下的所有节点放到其兄弟叶节点中，默认是

放到左兄弟中，若是没有左兄弟，则放到右兄弟中。同时删除当前内部节点的多余的关键字。返回上一级。

使用的 combine 函数：该函数需要额外访问左右子节点。

```
//合并，我选择右边合到左边，这样少1步;
template<typename data_type, typename key_type, typename getKey, int order, int L>
void B_Tree<data_type, key_type, getKey, order, L>::moveleaf_combine(Node* current, const int& keyposition)
{
    increaseAccessTime(); //访问磁盘
    increaseAccessTime(); //访问磁盘
    //cout<<"moveleaf_combine"<<endl;
    Leaf* left = current -> branch.leaf[keyposition];
    Leaf* right = current -> branch.leaf[keyposition+1];
    int temp = left -> count;
    left -> count += right -> count;
    //cout<<"hello"<<endl;
    for(int i = temp; i<left -> count; i++)
    {
        left -> data[i] = right -> data[i-temp];//将右边的叶子的数据都移动到左边的叶子;
    }
    delete right;
    for(int i = keyposition+1; i<current -> count; i++)//更新叶子指针数组;
        current -> branch.leaf[i] = current -> branch.leaf[i+1];
    current -> count--;//更新count;
    for(int i = keyposition; i<current -> count; i++)//更新键值数组;
        current -> key[i] = current -> key[i+1];
    //cout<<"hello"<<endl;
    //cout<<current -> count<<endl;
}
```

3. **对于包含内部节点指针的内部节点：**

1. 判断为了寻找被删除元素所使用的内部节点指针指向的内部节点中的关键词个数是否小于 B+树所允许的最小值，若小于，则执行以下步骤，否则返回上一级。

2. 判断关键词个数小于 B+树允许的最小值的节点的左右兄弟是否有多余的关键字，若有多余的关键字，则执行旋转操作，将父亲的一个关键字拉下来，左兄弟或者右兄弟的一个关键字上提到父亲节点。

左旋转函数（右旋转函数同理）：左旋转函数需要额外访问左右子节点。

```
//将右边元素放到左边
template<typename data_type, typename key_type, typename getKey, int order, int L>
void B_Tree<data_type, key_type, getKey, order, L>::movenode_left(Node* current, const int& position)
{
    increaseAccessTime(); //访问磁盘
    increaseAccessTime(); //访问磁盘
    Node* right = current -> branch.node[position+1];
    Node* left = current -> branch.node[position];
    key_type temp = current -> key[position];
    current -> key[position] = right -> key[0];
    Node* temp1 = right -> branch.node[0];
    right -> count--;
    for(int i=  0; i<right -> count; i++)
    {
        right -> key[i] = right -> key[i+1];
        right -> branch.node[i] = right -> branch.node[i+1];
    }
    right -> branch.node[right -> count] = right -> branch.node[right -> count+1];
    left -> key[left -> count++] = temp;
    left -> branch.node[left -> count] = temp1;
}
```

3. 若步骤 2 不能向左兄弟或者右兄弟借到节点，那么就执行合并操作。此时需要注意的是要判断是否为根节点。我使用的合并方法是将父亲节点中的一个关键字拉下到左子节点同时将右子节点的关键字都放入到左子节点中，在父亲节点中删除该关键字，删除有子节点。若父亲节点为根且只有 1 个关键字，那么需要降低高度，调整 root 的值。

合并函数：合并函数需要额外访问左右子节点。

```
//虽然用的for比较多，但是实际上循环次数不多；这里的参数keyposition代表关键词下标，需要注意；
template<typename data_type, typename key_type, typename getKey, int order, int L>
void B_Tree<data_type, key_type, getKey, order, L>::movenode_combine(Node* current, const int& keyposition)
{
    increaseAccessTime(); //访问磁盘
    increaseAccessTime(); //访问磁盘
    Node* left = current -> branch.node[keyposition];
    Node* right = current -> branch.node[keyposition+1];
    if(current -> count == 1 && current == root)
    {
        current -> tag = left -> tag;
        int temp_count = left -> count;
        for(int i = 0; i<temp_count+1; i++)
        current -> branch.node[keyposition + i] = left -> branch.node[i];
        for(int i = 0; i<right -> count+1; i++)
        current -> branch.node[keyposition + temp_count + i + 1] = right -> branch.node[i];
        current -> key[keyposition + temp_count] = current -> key[keyposition];
        for(int i = 0; i<temp_count; i++)
        current -> key[keyposition + i] = left -> key[i];
        for(int i = 0; i<right -> count; i++)
        current -> key[keyposition + temp_count +i +1] = right -> key[i];
        current -> count = temp_count + current -> count + right -> count;
        delete left;
        delete right;
    }
```

```
        else
        {
            int temp_count = left -> count+1;
            key_type temp_key = current -> key[keyposition];
            current -> count -= 1;
            for(int i = keyposition; i< current -> count; i++)
            {
                current -> key[i] = current -> key[i+1];
                current -> branch.node[i+1] = current -> branch.node[i+2];
            }
            left -> key[left -> count] = temp_key;
            left -> count += right -> count +1;
            for(int i = temp_count; i< left-> count; i++)
            {
                left -> key[i] = right -> key[i-temp_count];
                left -> branch.node[i] = right -> branch.node[i-temp_count];
            }
            left -> branch.node[left-> count] = right -> branch.node[right-> count];
            delete right;
        }
    }
```

四、初始化函数：

输入一个数据数组，数据数组的长度为 B+树允许的树叶所含数据数量的最小值的两倍。形成一颗我们定义的最小的 B+树。

```
/*
    功能：利用输入的(L/2)*2个数据初始化形成一颗最小的B+树
    前提：key_type重载了>或者是能够直接比较
    结果：形成由一个根和两片叶子组成的B+树
*/
void init(data_type data [(L/2)*2])
{
    clear();
    getKey getkey;
    int init_count = L/2;
    for(int i = 0; i< init_count*2; i++)
    {
        for(int j = 0; j< init_count*2-i-1; j++)
        {
            if(getkey(data[j])>getkey(data[j+1]))
                swap(data[j], data[j+1]);
        }
    }
    root = new Node (LEAF);
    root -> count = 1;
    root -> branch.leaf[0] = new Leaf();
    root -> branch.leaf[1] = new Leaf();
    root -> branch.leaf[0] -> count = init_count;
    root -> branch.leaf[1] -> count = init_count;
    for(int i = 0; i< init_count; i++)
    {
        root -> branch.leaf[0] -> data[i] = data[i];
         root -> branch.leaf[1] -> data[i] = data[i+init_count];
     }
     root -> key[0] = getkey(data[init_count]);
 }
```

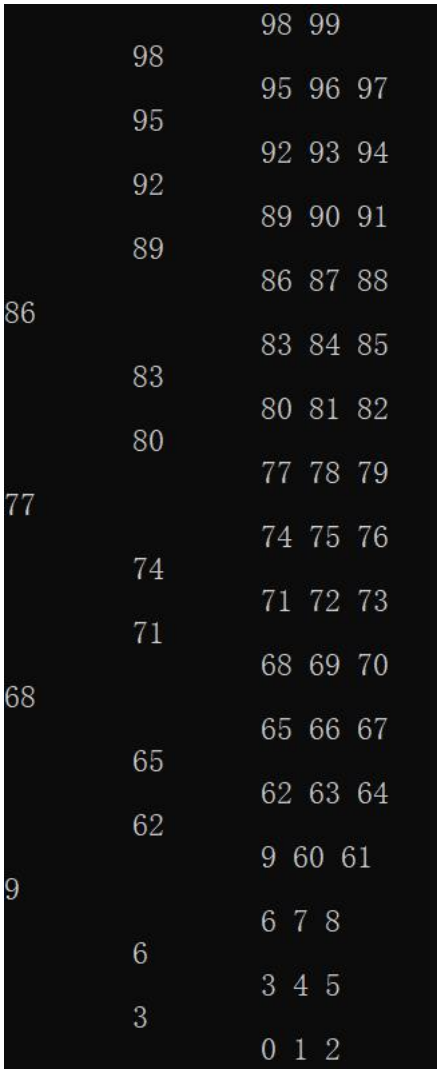
五、实验测试：

① **初始化至少 50 个测试样例：**

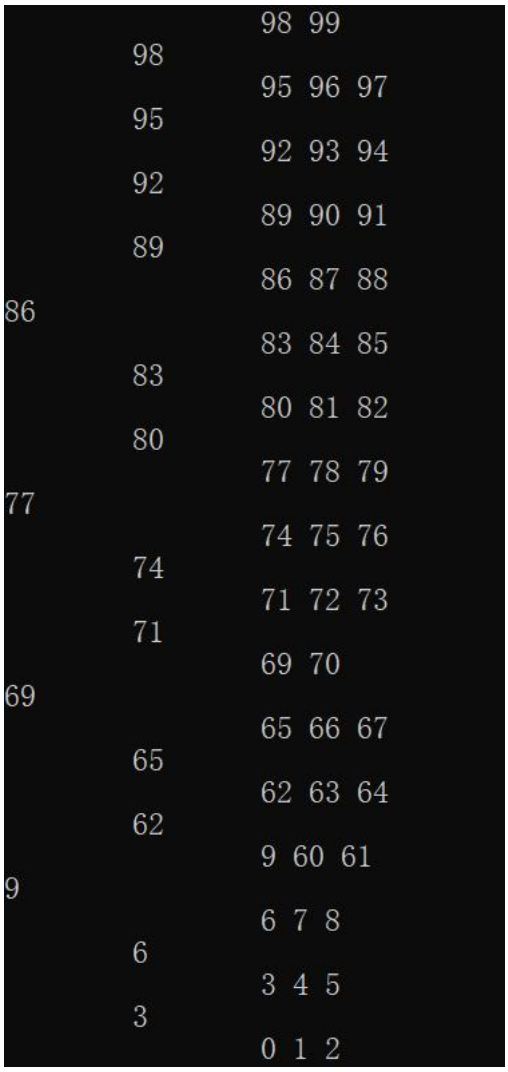
在 test.cpp 中我们初始化树为 15、23、89、50，形成一颗最简单的 B+树，然后我们使用 for 循环插入 0~99 再删除 10~59。途中每次插入删除会打印 B+树，可以观察 B+树的变化并使用查找函数检查是否发现插入或者删除错误。

② **test.cpp 中有不完善的地方，即它的插入删除太规律了，所以需要进行一些额外的测试，额外的测试需要证明所有内部节点关键字的值为其右儿子能寻找到的数据的最小值。**

原测试样例结果：



我修改的样例是再删除最小 68，检查是否会对上面节点的关键字有影响：



发现上面内部节点关键字更新为 69，符合我们 B+树的定义的要求。这是对测试样例的一些小小的补充。