# IndexManager：

邬凡

Index Manager负责B+树索引的实现，实现B+树的创建和删除（由索引的定义与删除引起）、等值查找、插入键值、删除键值等操作，并对外提供相应的接口。

B+树中节点大小应与缓冲区的块大小相同，B+树的叉数由节点大小与索引键大小计算得到。

为了实现B+树，IndexManager包含四个类：

> **BPTreeKey**
> 用来表示索引键

```
class BPTreeKey {
    ......
    int getKeyDataLength();
    void convertToRawData();
    void parseFromRawData();

    bool operator<  (const BPTreeKey &key);
    bool operator== (const BPTreeKey &key);
    bool operator>  (const BPTreeKey &key);
    bool operator>= (const BPTreeKey &key);

    BPTreeKeyType   type;
    char            charData[256];
    char            rawData[256];
    int             intData;
    float           floatData;
    int             keyLen;
};
```

> 整个key的实现与**Attribute**类相似，但是少了attributeName等属性，而为了使得API方便构造BPTreeKey，类里面提供了类型为**Attribute**的拷贝构造。通过重载与运算符方便了树的插入操作。

> **BPTreeEntry**
> 用于表示key，pointer对

```
class BPTreeEntry {
    ......
    int getEntryDataLength();

    BPTreeKey       key;
    PageIndexType   pagePointer;
};
```

**BPTreeEntry**的主要目的在于整合key，pointer对。代码也比较少。

**BPTreeNode**
用来表示一个B+树节点

```
enum class BPTreeNodeType {
    BPTreeUndefinedNode = 0,
    BPTreeInternalNode,
    BPTreeLeafNode
};

class BPTreeNode {
    ......
    void            readNode();
    void            writeNode();

    bool            isOverflow();
    bool            isUnderflow();

    bool            insertEntry(BPTreeEntry entry);
    bool            deleteEntry(BPTreeKey key);

    BPTreeEntry     *nodeEntries;
    int             entryNumber;
    int             keyDataLength;
    BPTreeKeyType   keyType;
    PageIndexType   parentNodePagePointer;
    PageIndexType   siblingNodePagePointer;
    Page            nodePage;
    BPTreeNodeType  nodeType;
};
```

实现中采取了使用4096字节即一页来存储一个node,通过成员nodePage来读取和存储整个node。一个node在硬盘中的大致数据结构如下:
**header | entry0 | entry1 | entry2 |...**
转换成硬盘数据的过程也很简单，将所有的数据按顺序读取二进制字节码写入Buffer即可。
代码如下:

```
void BPTreeNode::convertToRawData() {
    char *cursor = nodePage.pageData;
    BPTreeNodeHeader &nodeHeader = (BPTreeNodeHeader&)(*cursor);

    nodeHeader.entryNumber = entryNumber;
    nodeHeader.keyDataLength = keyDataLength;
    nodeHeader.keyType = keyType;
    nodeHeader.parentNodePagePointer = parentNodePagePointer;
    nodeHeader.siblingNodePagePointer = siblingNodePagePointer;
    nodeHeader.nodeType = nodeType;

    cursor += sizeof(BPTreeNodeHeader);

    for (int i = 0; i < entryNumber; ++i) {
        nodeEntries[i].key.convertToRawData();
        memcpy(cursor, nodeEntries[i].key.rawData,
nodeEntries[i].key.getKeyDataLength());
        cursor += nodeEntries[i].key.getKeyDataLength();
        memcpy(cursor, &nodeEntries[i].pagePointer,
sizeof(PageIndexType));
        cursor += sizeof(PageIndexType);
    }
}
```

**BPTreeNodeHeader** 结构体为方便读取和存储header而设计，代码如下:

```
struct BPTreeNodeHeader {
    int             entryNumber;
    int             keyDataLength;
    BPTreeKeyType   keyType;
    PageIndexType   parentNodePagePointer;
    PageIndexType   siblingNodePagePointer;
    BPTreeNodeType  nodeType;
};
```

> header目的在于存储当前节点的信息，信息包括节点中entry的数量，节点中索引键的长度，索引键的类型，父节点的页号，兄弟节点的页号，节点为内部节点或者页节点。通过这样的设计使得**API**只需要知道节点的页号就可以读取节点的所有完整信息。
> node中的所有entry作为数组保存在node的内存中，由于entry消耗的内存较大，在实现中采取了使用new和delete的方法动态分配内存。

下面依次讲解**node**中公开的接口作用。

```
void readNode();
void writeNode();
```

> 当成员nodePage设定好表名，属性名以及页属性之后，调用readNode()函数可以从硬盘中读取数据并解析后放入成员变量的内存中，类似的调用writeNode()函数可以将当前node类对象在内存中的数据转换成能在硬盘中存储的数据并写入硬盘。由于写入读取都经过了Buffer,在实际的情况中可能并没有写入到硬盘里面。

```
bool isOverflow();
bool isUnderflow();
```

> 以上的两个函数用来判断当前的node类对象在内存中的数据量的程度。如果转换成的存储在硬盘中的数据大于PAGESIZE常量(4096)，isOverflow()会返回true，否则返回false。类似的当数据小于PAGESIZE/2,isUnderflow()会返回true，否则返回false。

```
bool insertEntry(BPTreeEntry entry);
bool deleteEntry(BPTreeKey key);
```

> 以上的两个函数根据参数插入和删除在内存中的类对象数据。insert会根据参数中的entry的key选择合适的位置将参数entry插入到entry数组中，而delete会根据参数key在node中寻找出拥有与参数相同的entry并将其删除。

> **BPTree**
> BPTree类根据B+树实现了针对B+树的各种操作，类对外接口如下：

```
class BPTree {
    ......
    bool                insertKeyPointerPair(BPTreeKey key,
PageIndexType pagePointer);
    ......
    bool                deleteKey(BPTreeKey key);
    ......

    PageIndexType       searchKeyForPagePointer(BPTreeKey key);
    ......

    string              tableName;
    string              attributeName;
    BPTreeKeyType       keyType;
    int                 keyDataLength;
};
```

在使用**BPTree**类实例之前，需要使用构造函数对其成员变量tableName, attributeName, keyType, keyDataLength进行设定。传递给构造函数后构造函数回自动配好针对Buffer的设定读取硬盘文件。下面针对每一个公共接口详细介绍实现方法。

```
    bool insertKeyPointerPair(BPTreeKey key, PageIndexType
pagePointer);
```

此函数作用在于对当前B+树插入一个键值对，当插入成功返回true，失败返回false。

```
    BPTreeNode splitLeaveNode(BPTreeNode &node);
    BPTreeNode splitInternalNode(BPTreeNode &node);
    BPTreeNode createNode();
    bool insertEntryIntoNode(BPTreeEntry entry, BPTreeNode node);
    bool updateEntryIntoNode(BPTreeEntry entry, BPTreeNode node);
```

以上为为了实现插入功能所实现的辅助函数。
**splitLeaveNode(BPTreeNode&)**和**splitInternalNode(BPTreeNode&)**会将参数node分裂成两个node,由于传递的是实参，函数会将一个分裂出来的node保存在参数中并将另一个作为返回值返回。之所以实现了两个split是因为B+树的非叶子节点的最初的entry只有指针起作用，跟叶子节点的分裂有少许不同。
**createNode()**会根据当前树key信息设定好node的属性并返回一个新的node。
**insertEntryIntoNode()**会根据当前参数node的属性来进行操作，如果当前node是内部节点，函数会根据node中的key来寻找应当插入的字节点，如果当前node是叶子节点，函数会直接调用node的insert函数插入entry，然后判断当前节点是否overflow，如果overflow便开始分裂叶子节点，产生了新节点，这时候调用**updateEntryIntoNode()**插入分裂出来的节点产生的entry进入父节点，update函数如果插入后发现父节点也overflow的话便会调用分裂内部节点函数分裂父节点然后递归调用update函数更新父节点的父节点直到递归到Root，当Root也overflow得时候创建出两个新节点作为Root的字节点并将Root当前的所有数据分配给这两个节点。
**insertKeyPointerPair**在创建好后entry以后调用**insertEntryIntoNode()**到根节点即实现了插入。

```
bool deleteKey(BPTreeKey key);
```

此函数根据参数key来删除树中拥有相同key的entry。

```
bool deleteKeyInNode(BPTreeKey key, BPTreeNode node);
bool handelUnderflowInChildNodeOfNodePage(BPTreeNode node,
PageIndexType childPage);
```

以上为辅助函数。 **deleteKeyInNode()**与insert类似，首先递归找到需要删除的entry然后删除这个entry，如果当前删除的node underflow了，调用**handelUnderflowInChildNodeOfNodePage()**，参数分别为当前节点的父节点和当前节点的页号，**handelUnderflowInChildNodeOfNodePage()**会找到underflow的node所对应的entry，并尝试与其兄弟节点进行重新分配，如果不成功就将其合并，并删除当前父节点，如果当前父节点也underflow便递归调用直到根节点，如果根节点为空，删除更换根节点为其子节点并删除根节点，不为空则结束。

```
PageIndexType searchKeyForPagePointer(BPTreeKey key);
```

> 搜索值的接口实现相对来说比较简单。从根节点开始递归查找，如果不是叶子节点就寻找相应叶子进行查找，如果是叶子节点便找与参数key相匹配的entry，如果找不到返回-1。

测试方法:
代码如下:

```
void printAll(BPTreeNode node) {
    for (int i = 1; i < node.entryNumber; ++i) {
        cout << node.nodeEntries[i].key.intData << endl;
    }
}

void printTree(BPTree &tree, BPTreeNode node, int deepth) {
    srand(time(0));
    if (node.nodeType == BPTreeNodeType::BPTreeLeafNode) {
        for (int j = 0; j < deepth; j++)
            cout << "-";
        cout << "deepth " << deepth << " ";
        cout << "Leaf node is ";//<<
node.nodeEntries[1].key.intData << endl;
        for (int i = 1; i < node.entryNumber; ++i) {
            cout << node.nodeEntries[i].key.intData;
//                for (int j = 0; j < 20; j++)
//                    cout << node.nodeEntries[i].key.charData[j];
            cout << " ";
        }
        cout << " page " << node.nodePage.pageIndex;
        cout << " parentpage " << node.parentNodePagePointer;
        cout << endl;
//        for (int i = 1; i < node.entryNumber; ++i) {
//            for (int j = 0; j < deepth; j++)
//                cout << "-";
//            cout << node.nodeEntries[i].key.intData << endl;
//        }
//        cout << endl << endl << endl;
    } else {
        for (int i = 0; i < node.entryNumber; ++i) {
            for (int j = 0; j < deepth; j++)
                cout << "-";
            cout << "internal node pointer ";
            cout << node.nodeEntries[i].key.intData << endl;
//            for (int k = 0; k < 20; k++)
//                cout << node.nodeEntries[i].key.charData[k];
//            cout << endl;


            printTree(tree,
tree.getNodeAtPage(node.nodeEntries[i].pagePointer), deepth + 1);
        }
    }
}
int saver[1000024];

void testDelete() {
```

```
    srand(time(0));
    BPTree tree("test2", "test2", BPTreeKeyType::INT, 4);
    BPTreeKey key;
    key.keyLen = 4;
    key.type = BPTreeKeyType::INT;
    map<int, bool> used;
    for (int i = 0; i < 1000000; i++) {
        while (used[key.intData])
            key.intData = rand() % 100000000;
        used[key.intData] = true;
        saver[i] = key.intData;
    key.type = BPTreeKeyType::INT;
        tree.insertKeyPointerPair(key, i);
    }
//    printTree(tree, tree.getNodeAtPage(ROOTPAGE), 1);
    cout<<endl<<endl<<endl;
    for (int i = 0; i < 1000000; i++) {
        key.intData = saver[i];
    key.type = BPTreeKeyType::INT;
        tree.deleteKey(key);
//        cout << "deleting " << saver[i] << endl;
//    printTree(tree, tree.getNodeAtPage(ROOTPAGE), 1);
//        cout << endl << endl << endl;
    }
    printTree(tree, tree.getNodeAtPage(ROOTPAGE), 1);
}

void testBPTree() {
    BPTree tree("test", "test", BPTreeKeyType::CHAR, 20);
    BPTreeKey key;
    key.keyLen = 20;
    key.type = BPTreeKeyType::CHAR;
    BPTreeNode node = tree.getNodeAtPage(ROOTPAGE);
//    for (int i = 1; i < node.entryNumber; i++) {
//        cout << node.nodeEntries[i].key.floatData << endl;
//        cout << node.nodeEntries[i].pagePointer << endl;
//        cout <<
tree.searchKeyForPagePointer(node.nodeEntries[i].key) << endl;
//    }
    printTree(tree, tree.getNodeAtPage(ROOTPAGE),1);
//    cout << tree.getLeadingPage() << endl;
    PageIndexType leading = tree.getLeadingPage();

    for (;leading != UNDEFINEED_PAGE_NUM; leading =
tree.getNodeAtPage(leading).siblingNodePagePointer) {
        cout << leading << endl;
    }

    for (int i = 1; i <= 10000; i++) {
```

```
        for (int j = 0; j < 20; j++)
            key.charData[j] = rand() % 26 + 'a';
        key.charData[20] = '\0';
//        cout << i << endl;
//        cout << string(key.charData) << endl;

//        key.floatData = rand() % 100000000;
//        key.intData = 2001;
//        cout << "insert " << key.intData << endl;
//        tree.insertKeyPointerPair(key, i);
//    printTree(tree, tree.getNodeAtPage(ROOTPAGE),1);
//        cout << endl << endl << endl << endl;
//        cout << i << endl;
//        cout << key.floatData << endl;
//        cout << tree.searchKeyForPagePointer(key) << endl;
//        cout << "orz" << endl << endl << endl;
    }
//    node = tree.getNodeAtPage(leading);
//    for (;;) {
//        for (int i = 1; i < node.entryNumber; ++i)
//            cout << node.nodeEntries[i].key.intData << endl;
//        if (node.siblingNodePagePointer == UNDEFINEED_PAGE_NUM)
break;
//        node = tree.getNodeAtPage(node.siblingNodePagePointer);
//    }
//    printTree(tree, tree.getNodeAtPage(ROOTPAGE),1);
}

void testBpNode() {
    BPTreeNode node;
    node.nodeType = BPTreeNodeType::BPTreeLeafNode;
}

void testKey() {
    BPTreeKey key1;
    key1.type = BPTreeKeyType::CHAR;
    key1.keyLen = 20;
    BPTreeKey key2;
    key2 = key1;
    memcpy(key1.charData, "aaaaaaaaaaaaaaaaaaaa", 20);
    memcpy(key2.charData, "ffffffffffffffffffff", 20);
    cout << key1.compare(key2) << endl;
    cout << key2.compare(key1) << endl;
}
```

测试有两方面，一个是插入小数据打印整个树，此时需要改变PAGESIZE，另一个是插入100W个点然后随机10个点打印并删除除了这10个节点意外的节点的所有节点，打印树，进行观察。