Lesson01--- B-树

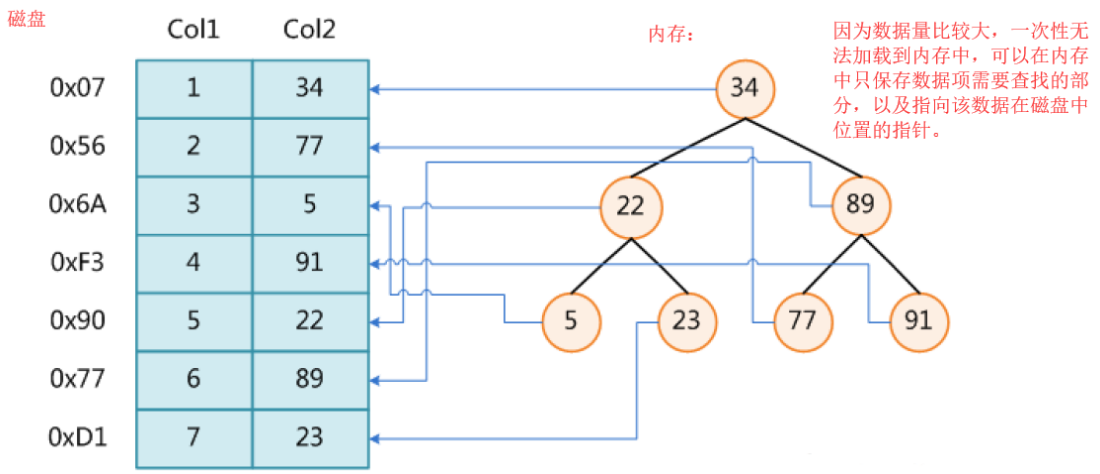
【本节目标】

- **1. 基本搜索结构**
- **2. B-树概念**
- **3. B-树的插入分析**
- **4. B-树的插入实现**
- **5. B+树和B*树**
- **6. B-树的应用**

1. 基本搜索结构

种类	数据格式	时间复杂度
顺序查找	无要求	O(N)
二分查找	有序	O($log_2 N$)
二叉搜索树	无要求	O(N)
二叉平衡树(AVL树和红黑树)	无要求-最后随机	O($log_2 N$)
哈希	无要求	O(1)
位图	无要求	O(1)
布隆过滤器	无要求	O(K)(K为哈希函数个数，一般比较小)

以上结构适合用于数据量不是很大的情况，如果数据量非常大，一次性无法加载到内存中，使用上述结构就不是很方便。比如：使用平衡树搜索一个大文件



上面方法其实只在内存中保存了每一项数据信息中需要查找的字段以及数据在磁盘中的位置，整体的数据实际也在磁盘中。

缺陷：

1. 树的高度比较高，查找时最差情况下要比较树的高度次
2. 数据量如果特别大时，树中的节点可能无法一次性加载到内存中，需要多次IO

那如何加速对数据的访问呢?

1. 提高IO的速度
2. 降低树的高度---多叉树平衡树

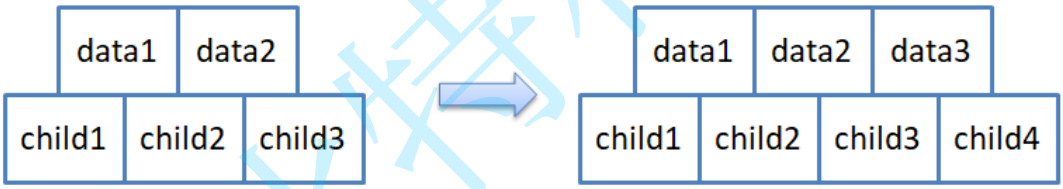
2. B-树概念

1970年，R.Bayer和E.mccreight提出了一种适合外查找的树，它是一种平衡的多叉树，称为B树(有些地方写的是B-树，注意不要误读成"B减树")。**一棵M阶(M>2)的B树，是一棵平衡的M路平衡搜索树，可以是空树**或者满足一下性质：

1. **根节点至少有两个孩子**
2. **每个非根节点至少有M/2(上取整)个孩子,至多有M个孩子**
3. **每个非根节点至少有M/2-1(上取整)个关键字,至多有M-1个关键字，并且以升序排列**
4. **key[i]和key[i+1]之间的孩子节点的值介于key[i]、key[i+1]之间**
5. **所有的叶子节点都在同一层**

3. B-树的插入分析

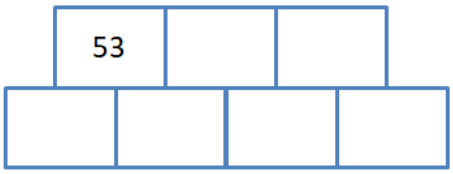
为了简单起见，假设M = 3. 即**三叉树，每个节点中存储两个数据，两个数据可以将区间分割成三个部分，因此节点应该有三个孩子**，为了后续实现简单期间，节点的结构如下：



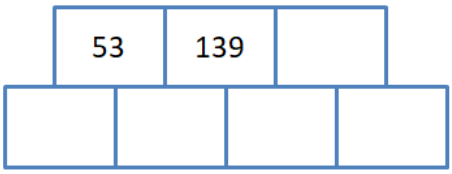
注意：**孩子永远比数据多一个。**

用序列{53, 139, 75, 49, 145, 36, 101}构建B树的过程如下：

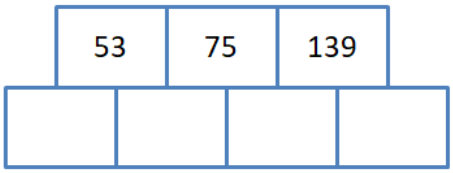
1. 插入53



2. 插入139

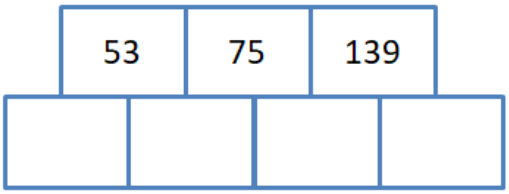


3. 插入75

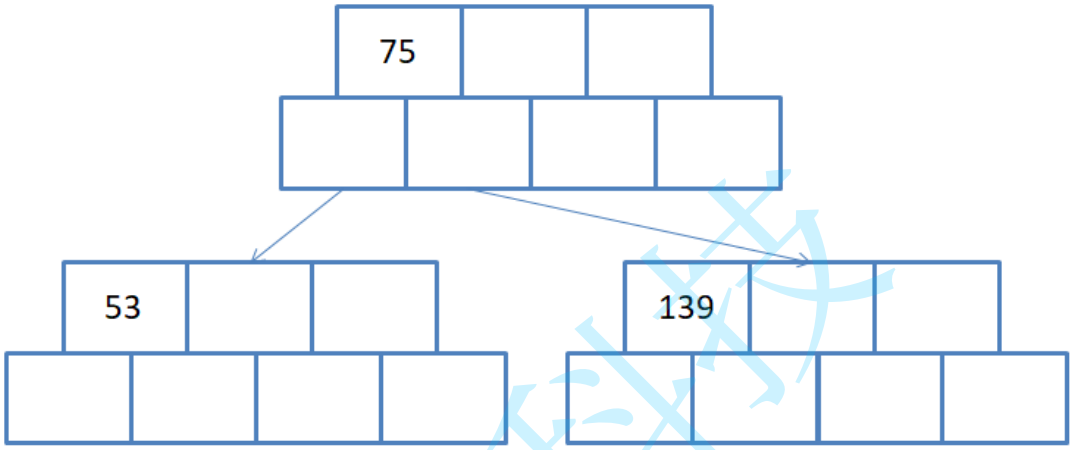


插入75的过程如下：
a. 按照插入排序思想将75插入到序列中
b. 插入后该节点不满足情况，需要对该节点进行分裂

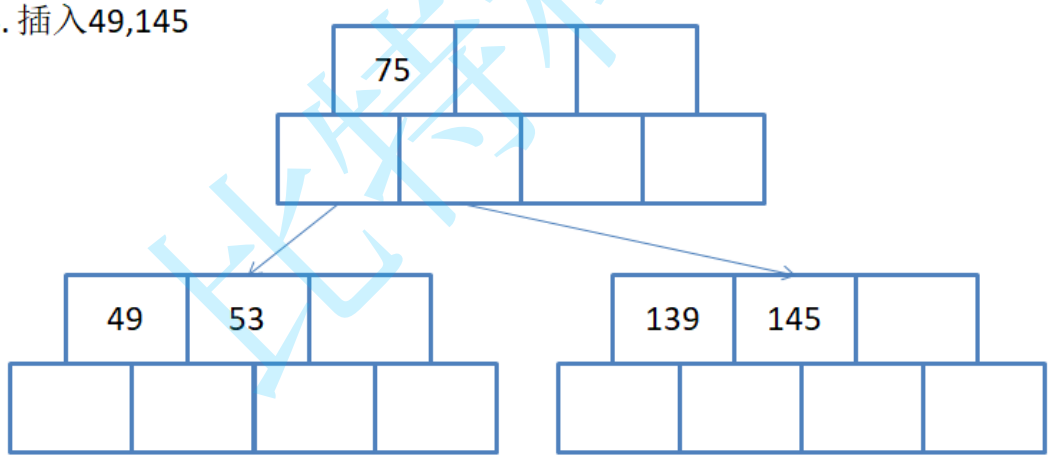
分裂节点：



1. 找到节点数据域的中间位置
2. 给一个新节点，将中间位置的数据搬移到新节点中
3. 将中间位置数据搬移到父节点中
4. 将节点连接好

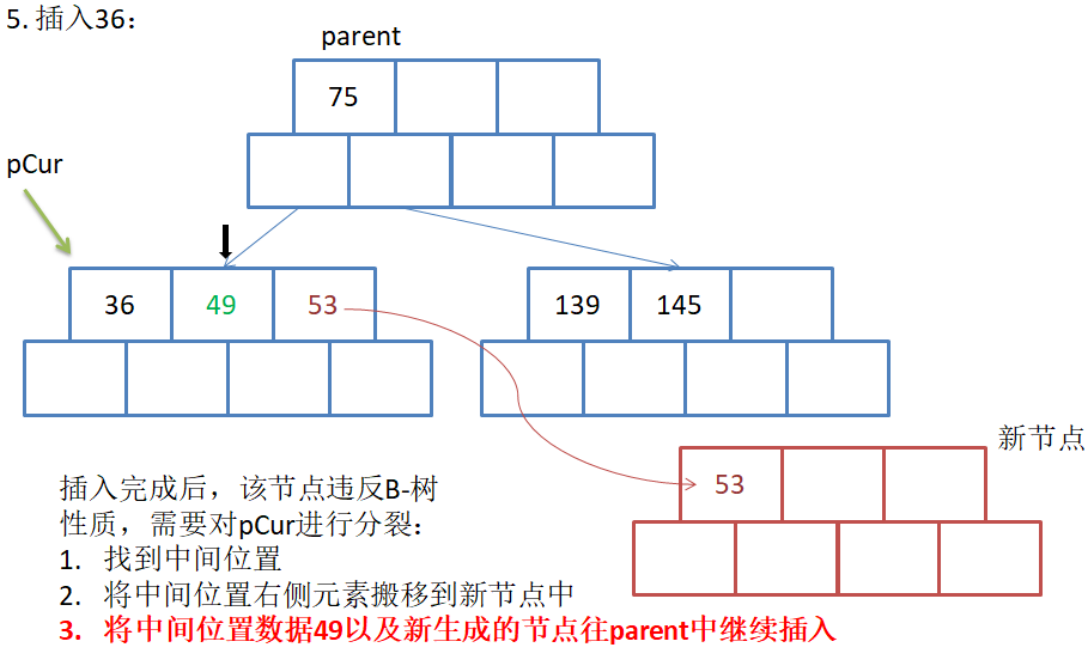


4. 插入49,145



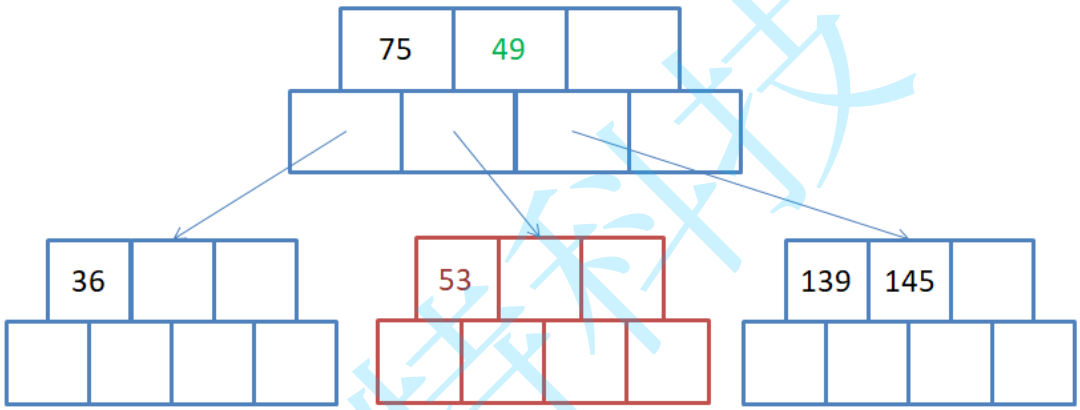
1. 找到该元素的插入位置(索要插入节点pCur)
2. 按照插入排序思想将节点插入到该节点(pCur)的合适位置
3. 检测该节点是否满足B树的性质
   满足：插入结束
   不满足：对节点进行分裂

5. 插入36：



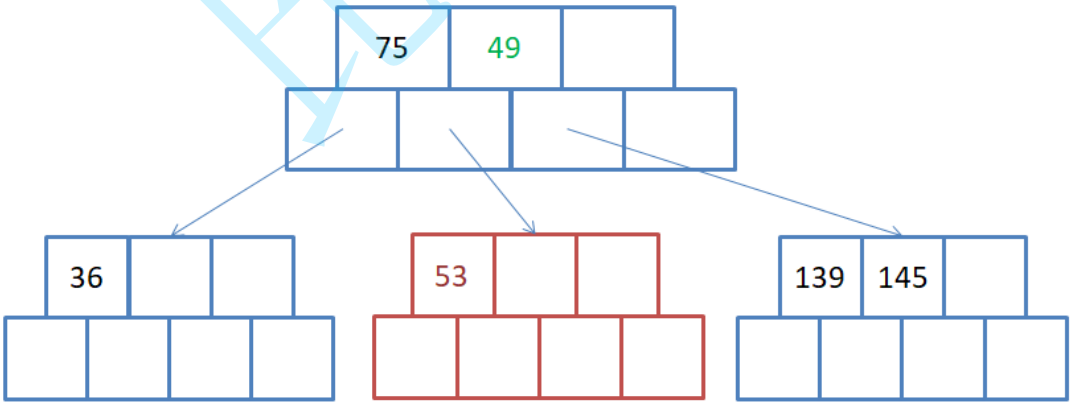
插入完成后，该节点违反B-树性质，需要对pCur进行分裂：

1. 找到中间位置
2. 将中间位置右侧元素搬移到新节点中
3. **将中间位置数据49以及新生成的节点往parent中继续插入**



6. 插入101

先在B-树中找到该节点的插入位置




```
// 参数：key为待查找的元素
// 返回值：PNode代表找到的节点，int为该元素在该节点中的位置
pair<PNode, int> Find(const K& key)
{
 // 从根节点的位置开始查找
    PNode pCur = _pRoot;
    PNode pParent = NULL;
    size_t i = 0;

 // 节点存在
```

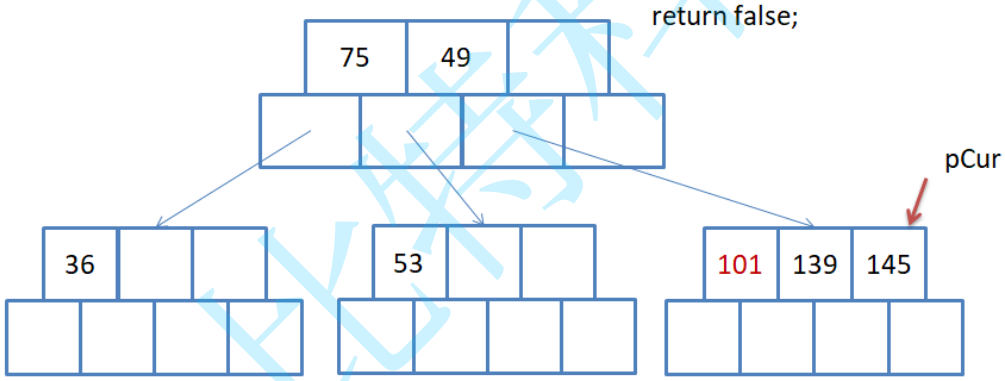
```
    while(pCur)
    {
        i = 0;
     // 在该节点的值域中查找
        while(i < pCur->_size)
        {
         // 找到返回
            if(key == pCur->_keys[i])
                return pair<PNode, int>(pCur, i);
            else if(key < pCur->_keys[i])  // 该元素可能在i的左边的孩子节点中
                break;
            else
                i++;  // 继续向右查找
        }

     // 在pCur中没有找到，到pCur节点的第i个孩子中查找
        pParent = pCur;
        pCur = pCur->_pSub[i];
    }

 // 没有找到
    return pair<PNode, int>(pParent, -1);
}
```

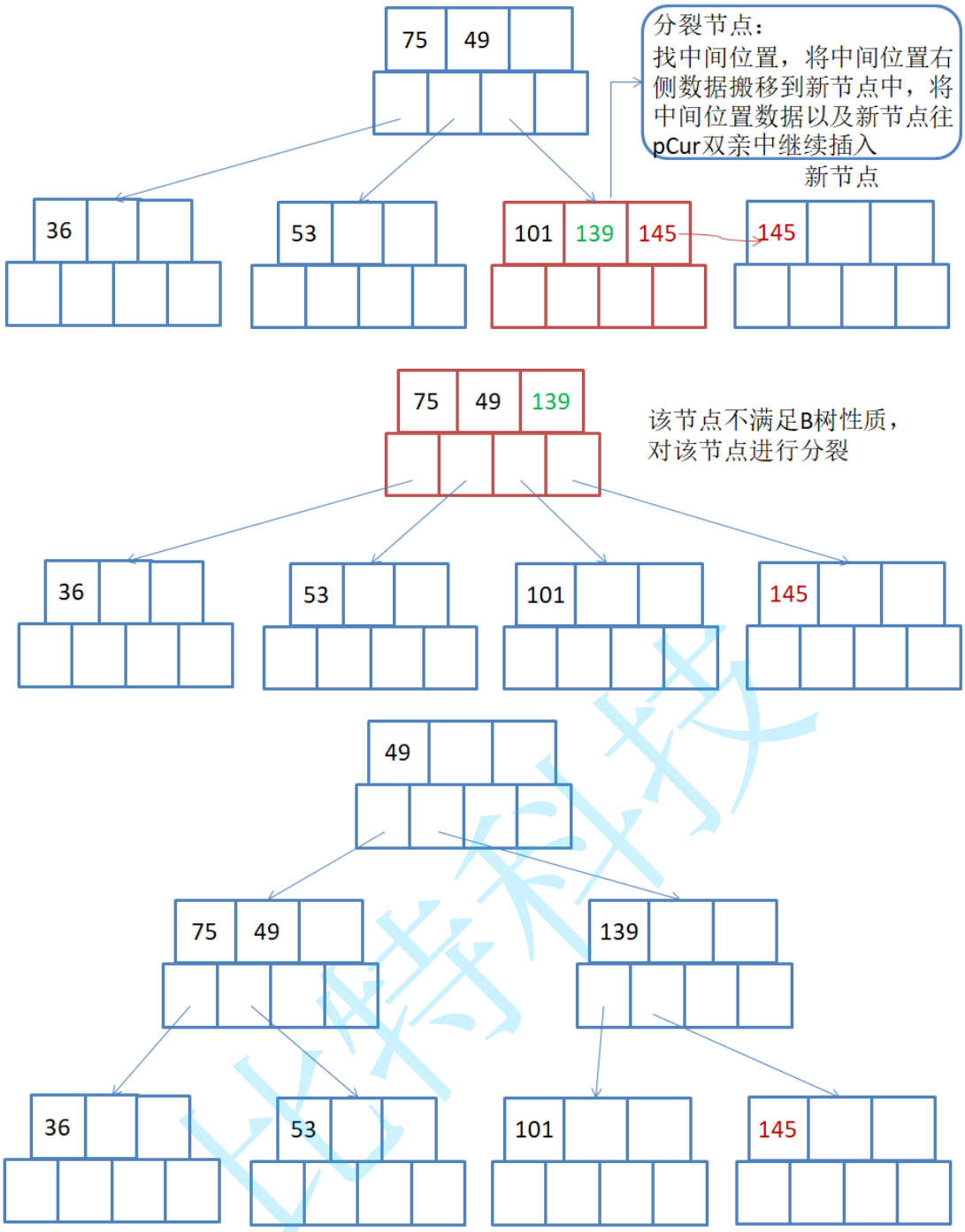
1. 先在B-树中找到该节点的插入位置

```
pair<Pnode, int> ret = t.Find(key);
if(-1 != ret.second) // 该元素已经存在
  return false;
```



2. 按照插入排序思想将该节点插入到pCur节点中

3. 检测该节点是否满足B树的性质，如果该节点中存放元素个数是否小于M
   小于M：插入完成
   等于M：需要对该节点进行分裂



插入过程总结：

1. 如果树为空，直接插入新节点中，该节点为树的根节点
2. 树非空，找待插入元素在树中的插入位置(注意：找到的插入节点位置一定在叶子节点中)
3. 检测是否找到插入位置(假设树中的key唯一，即该元素已经存在时则不插入)
4. 按照插入排序的思想将该元素插入到找到的节点中
5. 检测该节点是否满足B-树的性质：即该节点中的元素个数是否等于M，如果小于则满足
6. 如果插入后节点不满足B树的性质，需要对该节点进行分裂：
    - 申请新节点
    - 找到该节点的中间位置
    - 将该节点中间位置右侧的元素以及其孩子搬移到新节点中
    - 将中间位置元素以及新节点往该节点的双亲节点中插入，即继续4
7. 如果向上已经分裂到根节点的位置，插入结束

4. B-树的插入实现

4.1 B-树的节点设计

```
// M叉树：即一个节点最多有M个孩子，M-1个数据域
// 为实现简单期间，数据域与孩子与多增加一个(原因参见上文对插入过程的分析)
template<class K, int M = 3>
struct BTreeNode
{
    K _keys[M];  // 存放元素
    BTreeNode<K, M>* _pSub[M+1];  // 存放孩子节点，注意：孩子比数据多一个
    BTreeNode<K, M>* _pParent;    // 在分裂节点后可能需要继续向上插入，为实现简单增
加parent域
    size_t _size;  // 节点中有效元素的个数

    BTreeNode()
        : _pParent(NULL)
        , _size(0)
    {
        for(size_t i = 0; i <= M; ++i)
            _pSub[i] = NULL;
    }
};
```

4.2 插入key的过程

按照插入排序的思想插入key，注意：在插入key的同时，可能还要插入新分裂出来的节点。

```
void _InsertKey(PNode pCur, const K& key, PNode pSub)
{
 // 按照插入排序思想插入key
    int end = pCur->_size-1;
    while(end >= 0)
    {
        if(key < pCur->_keys[end])
        {
         // 将该位置元素以及其右侧孩子往右搬移一个位置
            pCur->_keys[end+1] = pCur->_keys[end];
            pCur->_pSub[end+2] = pCur->_pSub[end+1];
            end--;
        }
        else
            break;
    }

 // 插入key以及新分裂出的节点
    pCur->_keys[end+1] = key;
    pCur->_pSub[end+2] = pSub;

 // 更新节点的双亲
    if(pSub)
        pSub->_pParent = pCur;

    pCur->_size++;
}
```

4.3 B-树的插入实现

```
bool Insert(const K& key)
```

```
{
 // 如果树为空，直接插入
    if(NULL == _pRoot)
    {
        _pRoot = new Node();
        _pRoot->_keys[0] = key;
        _pRoot->_size = 1;
        return true;
    }

    // 找插入位置，如果该元素已经存在，则不插入
    pair<PNode, int> ret = Find(key);
    if(-1 != ret.second)
        return false;

    K k = key;
    PNode temp = NULL;
    PNode pCur = ret.first;
    while(true)
    {
     // 将key插入到pCur所指向的节点中
        _InsertKey(pCur, k, temp);

     // 检测该节点是否满足B-树的性质，如果满足则插入成功返回，否则，对pCur节点进行分裂
        if(pCur->_size < M)
            return true;

        // 申请新节点
        temp = new Node;

        // 找到pCur节点的中间位置
     // 将中间位置右侧的元素以及孩子搬移到新节点中
     int mid = (M >> 1);
        for(size_t i = mid+1; i < pCur->_size; ++i)
        {
            temp->_keys[temp->_size] = pCur->_keys[i];
            temp->_pSub[temp->_size++] = pCur->_pSub[i];

          // 跟新孩子节点的双亲
            if(pCur->_pSub[i])
                pCur->_pSub[i]->_pParent = temp;
        }

        // 注意：孩子比关键字多搬移一个
        temp->_pSub[temp->_size] = pCur->_pSub[pCur->_size];
        if(pCur->_pSub[pCur->_size])
            pCur->_pSub[pCur->_size]->_pParent = temp;

     // 更新pCur节点的剩余数据个数
        pCur->_size -= (temp->_size+1);

     // 如果分裂的节点为根节点，重新申请一个新的根节点，将中间位置数据以及分裂出的新节点
插入到新的根节点中，插入结束
        if(pCur == _pRoot)
        {
            _pRoot = new Node;
            _pRoot->_keys[0] = pCur->_keys[mid];
            _pRoot->_pSub[0] = pCur;
```

```
            _pRoot->_pSub[1] = temp;
            _pRoot->_size = 1;
            pCur->_pParent = temp->_pParent = _pRoot;
            return true;
        }
        else
        {
         // 如果分裂的节点不是根节点，将中间位置数据以及新分裂出的节点继续向pCur的双亲
中进行插入
            k = pCur->_keys[mid];
            pCur = pCur->_pParent;
        }
    }

    return true;
}
```

4.4 B-树的简单验证

对B树进行中序遍历，如果能得到一个有序的序列，说明插入正确。

```
void _InOrder(PNode pRoot)
{
    if(NULL == pRoot)
        return;

    for(size_t i = 0; i < pRoot->_size; ++i)
    {
        _InOrder(pRoot->_pSub[i]);
        cout<<pRoot->_keys[i]<<" ";
    }

    _InOrder(pRoot->_pSub[pRoot->_size]);
}
```

4.5 B-树的性能分析

对于一棵节点为N度为M的B-树，查找和插入需要$log_{M-1}N$~$log_{M/2}N$次比较，这个很好证明：**对于度为M的B-树，每一个节点的子节点个数为M/2 ~(M-1)之间，因此树的高度应该在要$log_{M-1}N$和$log_{M/2}N$之间，在定位到该节点后，再采用二分查找的方式可以很快的定位到该元素。**

B-树的效率是很高的，对于**N = 62*1000000000个节点，如果度M为1024，则$log_{M/2}N$ <= 4，即在620亿个元素中，如果这棵树的度为1024，则需要小于4次即可定位到该节点，然后利用二分查找可以快速定位到该元素，大大减少了读取磁盘的次数。**

4.5 B-树的删除

B树的删除请同学们参考《算法导论》和《数据结构-殷人昆》

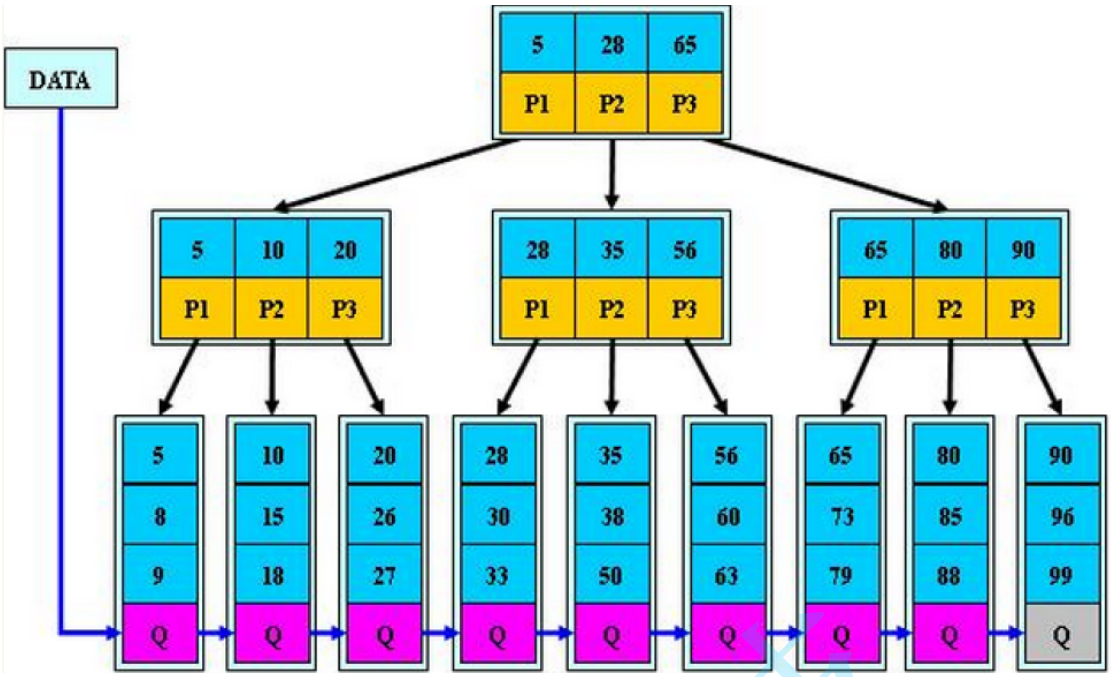
5. B+树和B*树

5.1 B+树

B+树是B-树的变形，也是一种多路搜索树：

1. 其定义基本与B-树相同，除了：
2. 非叶子节点的子树指针与关键字个数相同
3. 非叶子节点的子树指针p[i]，指向关键字值属于【k[i]，k[i+1])的子树

4. 为所有叶子节点增加一个链指针
5. 所有关键字都在叶子节点出现



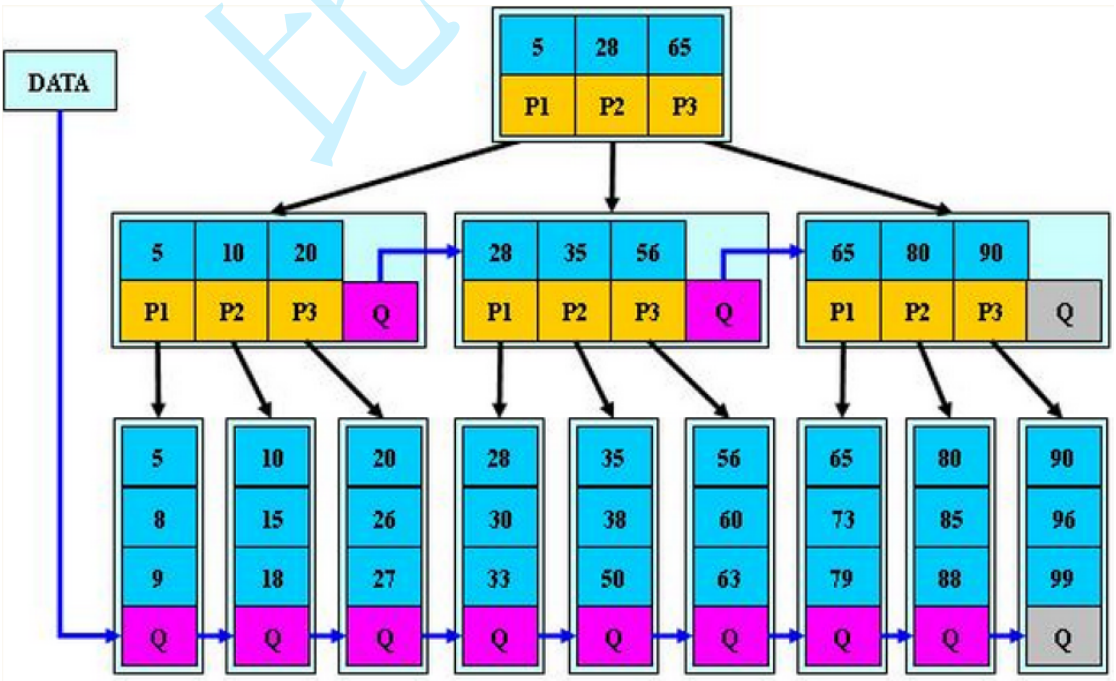
B+树的搜索与B-树基本相同，区别是B+树只有达到叶子节点才能命中(B-树可以在非叶子节点中命中)，其性能也等价与在关键字全集做一次二分查找。

B+树的特性：

1. 所有关键字都出现在叶子节点的链表中(稠密索引)，且链表中的节点都是有序的。
2. 不可能在非叶子节点中命中。
3. 非叶子节点相当于是叶子节点的索引(稀疏索引)，叶子节点相当于是存储数据的数据层。
4. 更适合文件索引系统

5.2 B*树

B*树是B+树的变形，在B+树的非根和非叶子节点再增加指向兄弟节点的指针。



B*树定义了非叶子结点关键字个数至少为(2/3)*M，即块的最低使用率为2/3（代替B+树的1/2）；

B+树的分裂：当一个结点满时，分配一个新的结点，并将原结点中1/2的数据复制到新结点，最后在父结点中增加新结点的指针；B+树的分裂只影响原结点和父结点，而不会影响兄弟结点，所以它不需要指向兄弟的指针；

B*树的分裂：当一个结点满时，如果它的下一个兄弟结点未满，那么将一部分数据移到兄弟结点中，再在原结点插入关键字，最后修改父结点中兄弟结点的关键字（因为兄弟结点的关键字范围改变了）；如果兄弟也满了，则在原结点与兄弟结点之间增加新结点，并各复制1/3的数据到新结点，最后在父结点增加新结点的指针；

所以，B*树分配新结点的概率比B+树要低，空间使用率更高；

5.3 总结

B-树：多路搜索树，每个结点存储M/2到M个关键字，非叶子结点存储指向关键字范围的子结点；

所有关键字在整颗树中出现，且只出现一次，非叶子结点可以命中；

B+树：在B-树基础上，为叶子结点增加链表指针，所有关键字都在叶子结点中出现，非叶子结点作为叶子结点的索引；B+树总是到叶子结点才命中；

B*树：在B+树基础上，为非叶子结点也增加链表指针，将结点的最低利用率从1/2提高到2/3；

####6. B-树的应用

#####6.1 索引

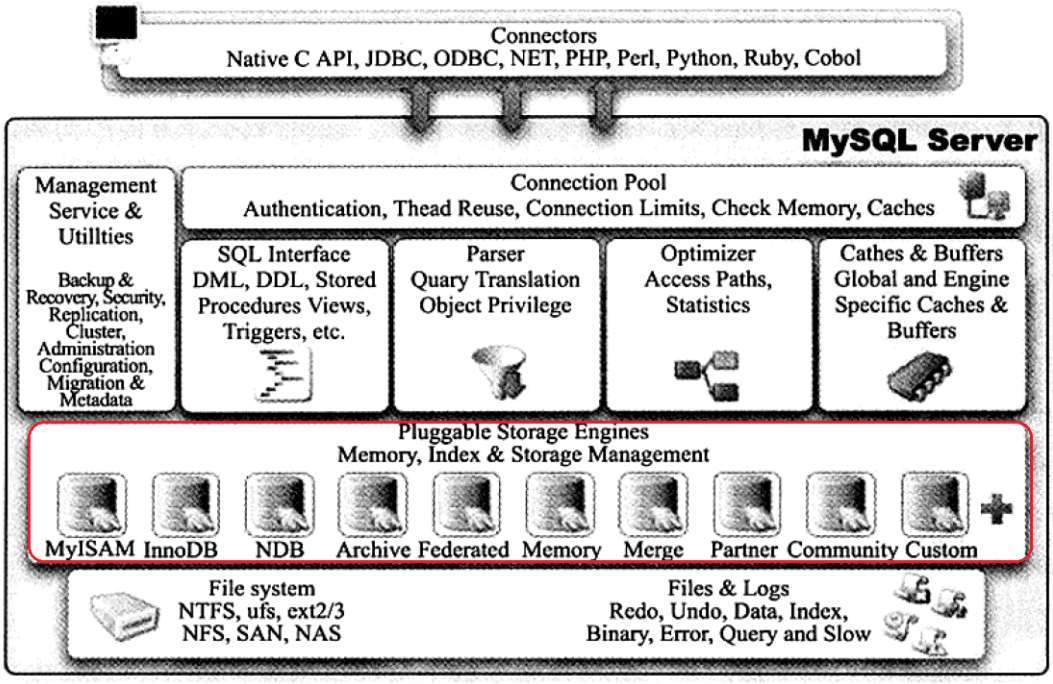
B-树最常见的应用就是用来做索引。索引通俗的说就是为了方便用户快速找到所寻之物，比如：书籍目录可以让读者快速找到相关信息，hao123网页导航网站，为了让用户能够快速的找到有价值的分类网站，本质上就是互联网页面中的索引结构。

MySQL官方对索引的定义为：**索引(index)是帮助MySQL高效获取数据的数据结构，简单来说：索引就是数据结构。**

当数据量很大时，为了能够方便管理数据，提高数据查询的效率，一般都会选择将数据保存到数据库，因此数据库不仅仅是帮助用户管理数据，而且数据库系统还维护着满足特定查找算法的数据结构，这些数据结构以某种方式引用数据，这样就可以在这些数据结构上实现高级查找算法，该数据结构就是索引。

6.2 MySQL索引简介

mysql是目前**非常流行的开源关系型数据库，不仅是免费的，可靠性高，速度也比较快，而且拥有灵活的插件式存储引擎**，如下：

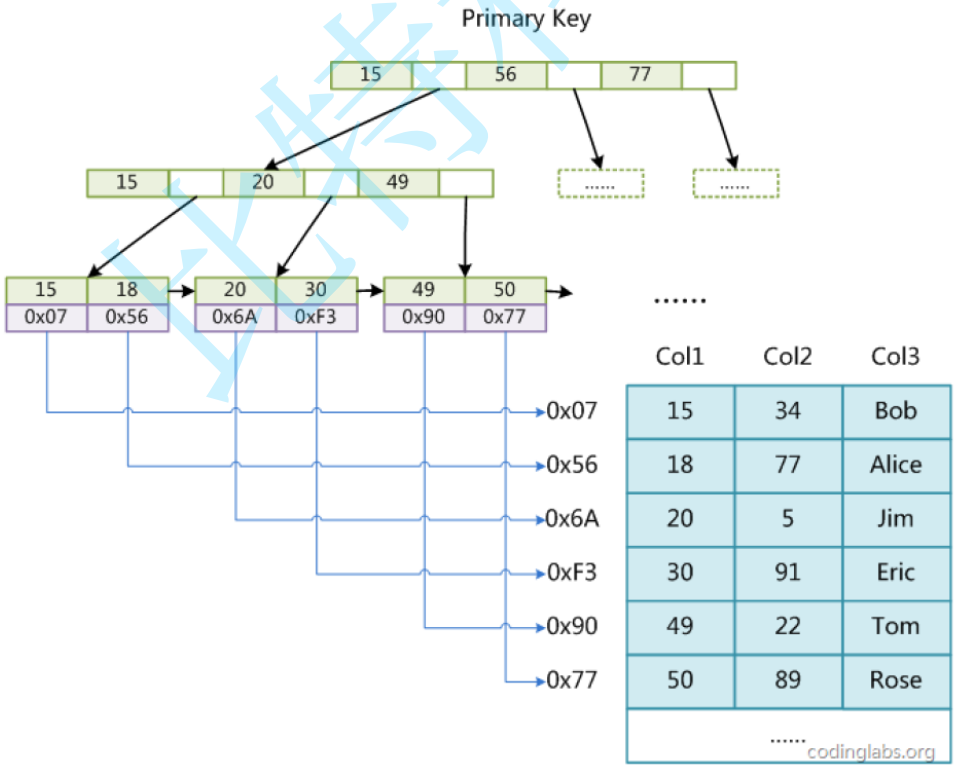


MySQL中索引属于存储引擎级别的概念，不同存储引擎对索引的实现方式是不同的。

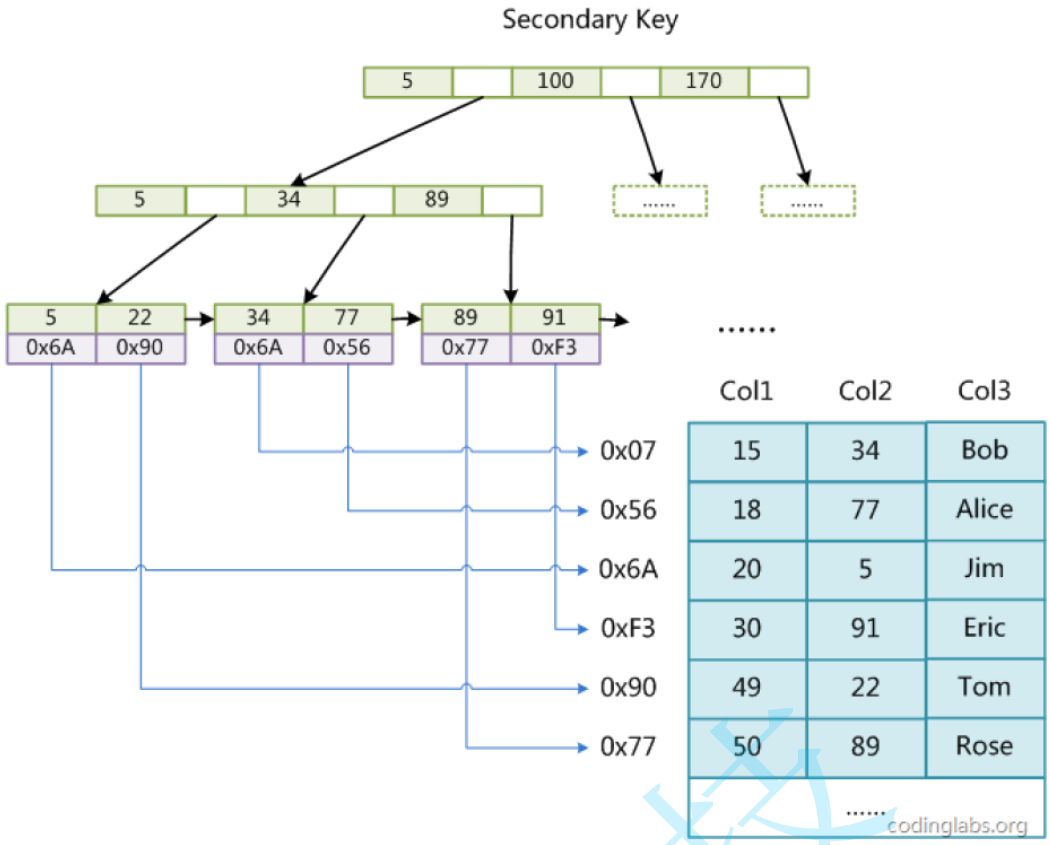
注意：索引是基于表的，而不是基于数据库的。

6.2.1 MyISAM

MyISAM引擎是MySQL5.5.8版本之前默认的存储引擎，不支持事物，支持全文检索，使用B+Tree作为索引结构，叶节点的data域存放的是数据记录的地址，其结构如下：



上图是以以Col1为主键，MyISAM的示意图，可以看出**MyISAM的索引文件仅仅保存数据记录的地址。在MyISAM中，主索引和辅助索引（Secondary key）在结构上没有任何区别，只是主索引要求key是唯一的，而辅助索引的key可以重复。**如果想在Col2上建立一个辅助索引，则此索引的结构如下图所示：

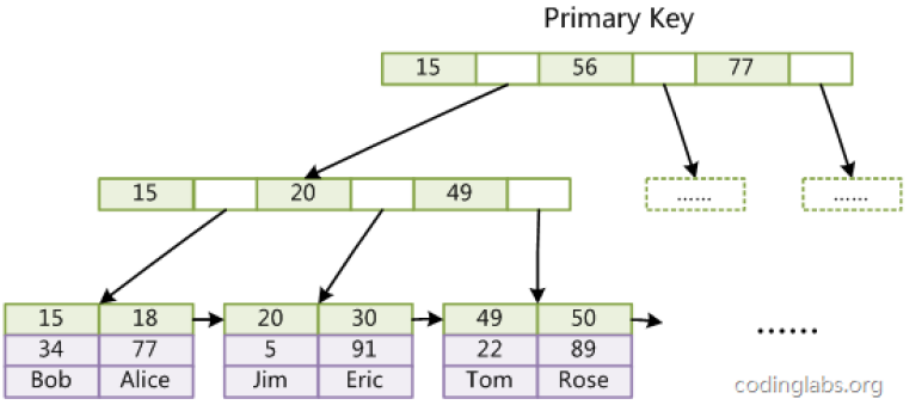


同样也是一棵B+Tree，data域保存数据记录的地址。因此，MyISAM中索引检索的算法为首先按照B+Tree搜索算法搜索索引，如果指定的Key存在，则取出其data域的值，然后以data域的值为地址，读取相应数据记录。 MyISAM的索引方式也叫做“非聚集索引”的。

6.2.2 InnoDB

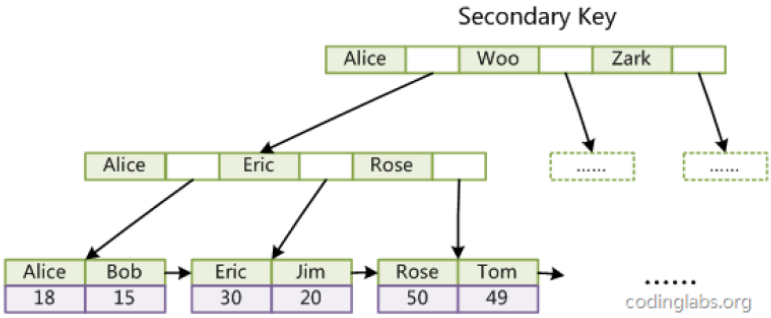
InnoDB存储引擎支持事务，其设计目标主要面向在线事务处理的应用，从**MySQL数据库5.5.8版本开始，InnoDB存储引擎是默认的存储引擎**。InnoDB支持B+树索引、全文索引、哈希索引。但InnoDB使用B+Tree作为索引结构时，具体实现方式却与MyISAM截然不同。

第一个区别是**InnoDB的数据文件本身就是索引文件。MyISAM索引文件和数据文件是分离的，索引文件仅保存数据记录的地址。**而**InnoDB索引，表数据文件本身就是按B+Tree组织的一个索引结构，这棵树的叶节点data域保存了完整的数据记录。**这个索引的key是数据表的主键，因此InnoDB表数据文件本身就是主索引。



上图是**InnoDB主索引**（同时也是数据文件）的示意图，可以看到**叶节点包含了完整的数据记录，这种索引叫做聚集索引。**因为**InnoDB的数据文件本身要按主键聚集，所以InnoDB要求表必须有主键**（MyISAM可以没有），**如果没有显式指定，则MySQL系统会自动选择一个可以唯一标识数据记录的列作为主键，如果不存在这种列，则MySQL自动为InnoDB表生成一个隐含字段作为主键，这个字段长度为6个字节，类型为长整形。**

第二个区别是**InnoDB的辅助索引data域存储相应记录主键的值而不是地址**,所有辅助索引都引用主键作为data域。



聚集索引这种实现方式使得按主键的搜索十分高效，但是**辅助索引搜索需要检索两遍索引：首先检索辅助索引获得主键，然后用主键到主索引中检索获得记录。**

问题：既然在表格上建立索引可以提高搜索效率，那是否可以在一个表格上任意建立索引?

http://www.cnblogs.com/yangecnu/p/Introduce-B-Tree-and-B-Plus-Tree.html http://www.cnblogs.com/oldhorse/archive/2009/11/16/1604009.html http://blog.codinglabs.org/articles/theory-of-mysql-index.html

数据库的底层原理