# B+树

## 主要数据结构

class bplus\_node {  
public:  
 static int m;//表示阶数  
 static int treeNodeNum;//树总结点数  
 static int saveKeyNum; //存储的元素的个数  
 int key\_num; //该节点中的关键字的个数，可以看作keys[]的逻辑边界  
 key\_type \*keys; //存放的关键字 内部节点存放的是每个孩子节点的最大值 叶子节点存放的就是关键字

bplus\_node \*\*child; //子节点的指针数组，同时用于判断是否是叶子节点，为空肯定就是叶子节点  
 bplus\_node \*parent; // 父节点  
 bplus\_node \*next;//兄弟节点，仅在叶子节点内有意义  
public:  
 bplus\_node(bplus\_node \*parent= nullptr, bool is\_leaf\_node=false);  
 void insert\_node(key\_type key); //向B+树中插入节点，只有head指针可以调用，head是头指针，key则是该节点所插入的内容的hash值 头指针会指向一个root，root才是真正的根节点  
 void insert\_key(bplus\_node \*now,key\_type key);//向node节点的keys数组中插入一个key，使其仍旧保持有序,同时令now->key\_num++,只用于叶子节点  
 void division\_node(bplus\_node \*now);//节点向上递归分裂,对节点now进行分裂，同时更新因分裂导致需要更新的所有内容  
 int get\_the\_next\_child(bplus\_node \*now,key\_type key); //根据key的值找到now中下属节点的id,也就是应该是now->child[id]  
 void insert\_child(bplus\_node \*now,bplus\_node \*new\_child,int k);//将new\_child插入到now中的合适位置，用于内部节点，同时更新keys和key\_num,新节点需要放到now->child[k]上  
 void init\_head(int m);//初始化头节点 传递进来阶数  
 int query(key\_type key);//只有head节点可以调用，头节点，不是根节点, 查找成功时返回查找的次数，查找不成功的时候返回-1  
 void printf(bplus\_node \*now);//递归输出now及其子节点的内容  
 void close(bplus\_node \*sqt);//释放空间，只能由head节点调用,传递进来sqt指针  
 void recurFree(bplus\_node \*now);//递归关闭now及其子节点  
};

创建了一个bplus\_node类，这个类表示的是b+树中的每个节点，以及对节点的对应操作，见表1-1和表1-2是该类属性及其方法的含义。

表1-1 bplus\_node属性及其含义

|  |  |  |
| --- | --- | --- |
| 属性 | 类型 | 含义 |
| m | static int | 表示当前树的阶数 |
| treeNodeNum | static int | 表示当前b+树中所有节点的个数 |
| saveKeyNum | static int | 表示当前b+树中存储的元素的个数，也是叶子节点的个数 |
| key\_num | int | 表示当前节点中关键字的个数，也就是存储了几个关键字 |
| keys | string \* | 内部节点存放的是每个孩子节点的最大值 叶子节点存放的就是关键字 |
| child | bplus\_node\*\* | 可以区别叶子节点和内部节点，叶子节点该值为nnullptr；内部节点该值不为nullptr，其表示孩子指针的数组 |
| parent | bplus\_node\* | 该节点的父节点，只有haed头节点的父节点为nullptr，这也是区分是否是head节点的唯一方法 |
| next | bplus\_node | 兄弟节点，只有在叶子节点内才有意义 |

表1-2 bplus\_node方法及其含义

|  |  |
| --- | --- |
| 方法 | 含义 |
| bplus\_node(bplus\_node \*parent = nullptr, bool is\_leaf\_node=false) | 类的构造函数，里面会更新treeNodeNum的值，同时更新各个属性默认为初始值 |
| void insert\_node(key\_type key) | 只有head节点调用这个方法才是正确的，向b+树中插入一个值为key的节点 |
| void insert\_key(bplus\_node \*now, key\_type key) | 将一个key插入到节点now中，做的是有序的插入 |
| void division\_node(blpus\_node \*now) | 节点的递归分裂，将now进行分裂，并更新其因为分裂带来的所有属性的更新 |
| void insert\_child(bplus\_node \*now, bplus\_node \*new, int k) | 在节点now的位置k插入节点new，其后面的child依次后移 |
| void init\_head(int m) | 初始化头节点，每次重新构建树的时候需要调用此函数，m表示初始该树的阶数 |
| void printf(bplus\_node \*now) | 递归输出节点now及其子节点的内容 |
| void close(bplu\_node \*sqt) | 释放空间，只能由head节点调用，传递进来sqt指针，关闭b+树 |
| void recurFree(bplus\_node \*now) | 递归释放now及其子节点，该函数由close()函数调用 |

## 主要流程

首先创建head指针和sqt指针，然后使用head指针调用函数init\_head(m)初始化b+树。然后使用insert\_node(value)函数向b+树中插入节点，多次调用该函数，以此来构建一颗完整的b+树。最后使用head\_query(value)查询该内容是否存在于b+树中。

inser\_node(value)的流程是根据value的值遍历b+树，当value是b+树中最大的值的时候更新b+树的遍历的节点的最大值。当遍历到叶子节点的时候，调用insert\_key()将value插入到叶子节点中，如果此时叶子节点关键字的个数大于m，则调用division\_node()函数递归分裂该节点。

division\_node(now)，首先判断带分裂的节点是否是root节点，如果是root节点，则需要新增一个root节点，然后将老root节点进行分裂，同时更新其child，key，key\_num,next，parent等信息，分裂的两个节点分别作为新root节点的子节点。如果不是root节点，则直接将该节点进行分裂，然后更新相应的属性，最后调用insert\_child()将分裂的节点插入到其原节点的父亲上。

insert\_child(now,new\_child,k)，将new\_child插入到now的child[k]上，其原先及其后面的child节点依次后移，同时判断now节点的关键字的个数是否大于m，如果大于m调用，division\_node方法

query(key)，从root节点开始进行遍历，找到其关键字大于等于该节点的，然后遍历其child子节点，如此循环，一直到叶子节点，查看叶子节点是否有这个key，有的话返回查询比较的次数，没有返回-1

## 实验过程

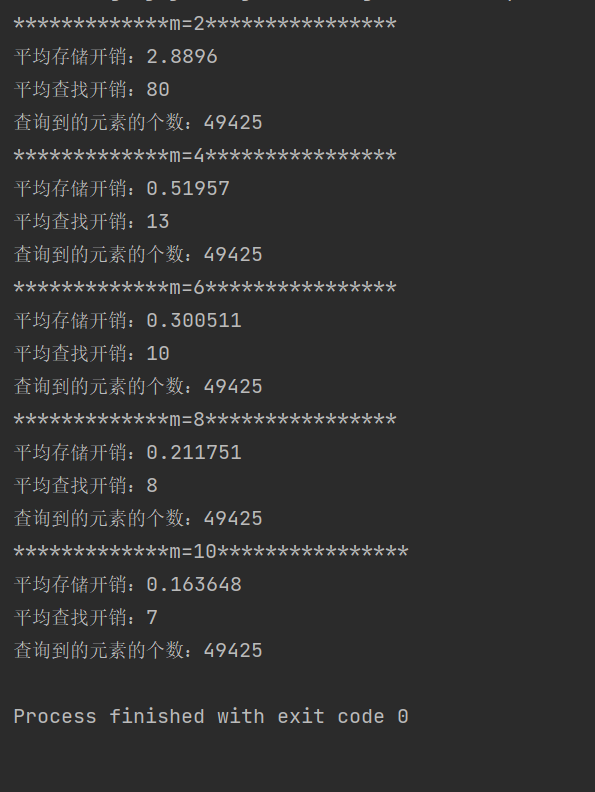


图1 运行结果

如图1所示，是程序运行5中不同阶数的b+树的运行结果

## 遇到的问题

节点分裂的时候，只是将父节点的child孩子节点指向分裂出来的新节点，没有将分裂出来的新节点的父亲指针指向原节点的父亲节点，这样就导致运行程序的时候会产生一些节点重复，一些节点消失的情况。

叶子节点的next指针需要每次在叶子节点分裂的时候进行更新，否则Sqt遍历的时候也会出问题。

## 技术指标

表1-3 b+树技术指标

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| b+树阶数 |  | 平均查找开销 | 平均存储开销 | 内存 |
| 2 |  | 80 | 2.8896 | 594.3MB |
| 4 |  | 13 | 0.51957 | 153.8MB |
| 6 |  | 10 | 0.300511 | 119MB |
| 8 |  | 8 | 0211751 | 106.1MB |
| 10 |  | 7 | 0.163648 | 99.9MB |

## 结论和总结

B+树的查找效率比较稳定，因为每一次的查询都是从root节点查到叶子节点，而B+树的每个叶子节点的深度都是一样的。但是B+树浪费的空间也是比较大，如果插入的数据是按照有序的顺序，那么叶子节点中的关键字只有一般的使用率，另一半分裂的时候空余了，会导致浪费的空间较大。

总的来说，B+树的查询效率是比较高且稳定的，只是B+树在书写的时候逻辑处理稍微复杂一点，因为每一个节点都有多个指针信息需要维护，这样就导致程序的编写复杂度较高。

# Trie树

## 主要数据结构

class TrieNode {  
public:  
 static int m; //分支的个数，是2的幂次采访，ptr开辟的空间应该为m+1,第一个表示的是$也就是指向一个具体的叶子节点，表示当前已经是一个字符串了，后m个表示多少种字符  
 static int treeNodeNum;//开辟的总的节点  
 static int saveKeyNum;//所存储元素的个数  
 int nodeKind; //0分支节点， 1叶子节点  
 string value;//叶子节点用到的value值  
 TrieNode \*\*ptr;//分支节点用到的内容  
 TrieNode(int nodeKind);  
 void insert\_node(string value);//将value插入到trie树中，只有root根节点可以调用该函数  
 int query(string value);//查询value，只能root根节点调用，查不到返回-1，查到返回比较的节点的次数  
 void init();//只能是head节点进行调用  
};

创建了一个TrieNode类，这个类表示的是Trie树中的每个节点，以及节点对应的操作，见表2-1和表2-2是该类属性及其方法的含义。

表2-1 TrieNode属性及其含义

|  |  |  |
| --- | --- | --- |
| 属性 | 类型 | 含义 |
| m | static int | 表示Trie树的阶数 |
| treeNodeNum | static int | 表示Trie树的总的节点数 |
| saveKeyNum | static int | 表示Trie数存储的元素的个数 |
| nodeKind | int | 表示当前节点是叶子节点还是分支节点，0表示分支节点，1表示叶子节点 |
| value | string | 叶子节点存储的值 |
| ptr | TrieNode\*\* | 分支节点的孩子指针 |

表2-2 TrieNode的方法及其含义

|  |  |
| --- | --- |
| 方法 | 含义 |
| TrieNode(int nodekind) | 创建一个nodekind类型的字典树节点 |
| void insert\_node(string value) | 将value插入到trie树中，只有root根节点可以调用该函数 |
| int query(string value) | 在trie树中查询value值是否存在，存在返回比较的次数，不存在返回-1，只有root节点可以调用该函数 |
| void init() | 初始化节点静态变量，只有head节点可以调用 |

## 主要流程

首先创建head节点，然后调用head=>init()初始化Trie树的静态变量，然后多次调用insert\_node()向trie树中插入内容，最后调用query()函数查询树中的内容。

insert\_node(value),首先根据静态变量m获得每层存储的二进制数位的个数。遍历vlaue,根据当前其二进制位num，查找到输入哪一个子节点。如果该子节点为空，则开辟分支节点，如果不为空，则遍历到下一层，如果是叶子节点，则开辟分支节点，并将新的分支节点插入到该叶子节点之上，然后遍历到该分支节点。当遍历完value，最后一次遍历，如果子节点位空，则直接创建叶子节点，如果不为空，则创建叶子节点放到下一层的ptr[0]上。

query(value)，根据静态变量m获得每层的二进制位数，遍历value,计算出当前的num,如果对应的子节点为空，则返回-1说明没有找到该内容，如果不为空则继续遍历下一层。当value遍历到最后一位的时候，如果是叶子节点则判断其内容是否是value，是返回比较的次数，否返回-1，如果是分支节点的话，则判断其对应下一层的ptr[0]的内容是否是value，是返回比较的次数，不是返回-1表明没有找到。

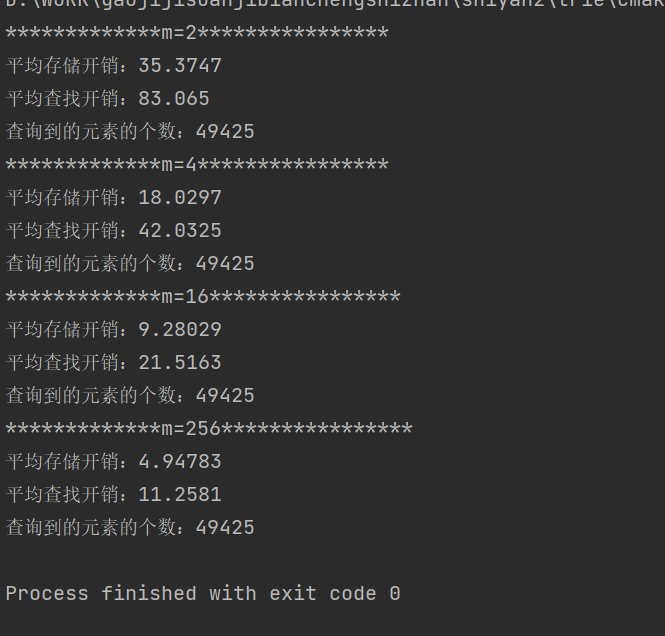


图2 实验结果

如图2所示，是Trie树不同阶数的实验结果。

## 技术指标

表2-3 trie树技术指标

|  |  |  |  |
| --- | --- | --- | --- |
| 阶数 | 平均查找开销 | 平均存储开销 | 内存 |
| 2 | 83.065 | 35.3747 | 3371.9MB |
| 4 | 42.0325 | 18.0297 | 1991.6MB |
| 16 | 21.5163 | 9.28029 | 1536.2MB |
| 256 | 11.2581 | 4.94783 | 8456.3MB |

## 结论和总结

通过实验指标可以看出来Trie树的缺点是内存消耗大，但是其实现相比较B+树来说要容易的多。Trie树和b+树的另一个不同是，Trie树每次查找的次数不固定，但是B+树每次都是查询到叶子节点，并且叶子节点都在同一层，因此B+树的查询次数固定。总的来说，Trie树也属于log(n)级别的查找复杂度，也是非常优秀的一种算法。

# 压缩的Trie树

## 主要数据结构

class TwoCTrieNode {  
public:  
 static int treeNodeNum;//节点总数  
 static int saveKeyNum; //存放的节点总数  
 unsigned char \* value; //表明当前节点存放的内容  
 int length;//表明当前节点存放的内容的长度  
 TwoCTrieNode \*left;//左节点  
 TwoCTrieNode \*right;//右节点  
 bool isEnd;//是否是一个终点11  
 TwoCTrieNode(bool isEnd = false);//初始化函数，length=0,value=null,left=null,right=null,isEnd=false;  
 void insert\_node(string value);//只有root节点可以调用该函数，root节点中也会存放value，  
 int query(string value);//查询，如果查询到了返回查询比较的次数，如果没有查询到，返回-1 只有root节点可以调用  
};

创建了一个TwoCTrieNode类，表示压缩的二叉Trie树中的每个节点，其属性和方法的含义见表3-1和表3-2。

表3-1 TwoCTrieNode属性及其含义

|  |  |  |
| --- | --- | --- |
| 属性 | 类型 | 含义 |
| treeNodeNum | static int | 压缩Trie树的总结点数 |
| saveKeyNum | static int | 压缩Trie数保存的关键字数 |
| value | unsigned char\* | 当前节点存放的内容 |
| length | int | 当前节点存放的内容的长度 |
| left | TwoCTrieNode \* | 左孩子 |
| right | TwoCTrieNode | 右孩子 |
| isEnd | bool | true表示是一个终点，否则不是 |

表3-2 TwoCTrieNode方法及其含义

|  |  |
| --- | --- |
| 方法 | 含义 |
| TwoCTrieNode(bool isEnd=false) | 节点的构造函数 |
| void insert\_node(string value) | 在树中插入节点，只有root节点可以调用 |
| int query(string value) | 查询到给定内容返回查询的比较次数，否则返回-1 |

## 主要流程

创建root节点，多次调用root->insert\_node()函数向树中插入节点，然后使用root->query()函数查询给定内容是否在树中出现。

insert\_node()，将给定的字符串转换为01数组，查询数组的首位来判断是对当前节点的左孩子还是右孩子进行遍历。比较和当前节点的匹配的长度，如果长度等于当前节点的内容长度，则判断是否匹配到了数组结尾，如果是，则将该节点设置为终端节点，如果不是则继续匹配下一层。如果长度不等于当前节点的内容长度，则将当前节点进行拆分，后面不等的内容组成一个新节点，此时判断是否匹配到了数组结尾，如果没有则将数组后面没有匹配的内容设置为一个新节点。

query()，查询给定内容是否在树中。将给定的字符串转换为01数组。判断当前节点长度和已经匹配的数组的长度的和是否小于等于数组的长度，如果不是，则返回-1表示没找到。如果是，则匹配当前节点的内容和数组未匹配的内容进行比较，如果不相等，则返回-1，表示没找到。如果匹配成功，判断当前数组是否全部匹配，如果是，并且当前是终端节点，则返回比较的次数，表示找到了，如果不是则返回-1表示没找到。

## 实验过程

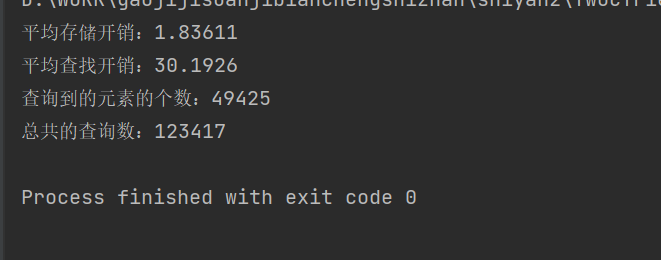


图3 实验过程

## 技术指标

表3-3 技术指标

|  |  |  |
| --- | --- | --- |
| 平均查找开销 | 平均存储开销 | 内存 |
| 30.1926 | 1.83611 | 368.1MB |

## 结论和总结

压缩的Trie相比较未压缩的Trie树，不论是平均查找开销、平均存储开销还是内存都有了非常大的优化，但相对而言其逻辑也是更加的复杂，相比较插入而言或许开销也会更大。

而且压缩Trie树有更频繁的动态申请和销毁空间，因为一个节点存储的一个内容，该节点会更频繁的进行分裂，那么就需要对原先的内容进行销毁并重新申请，因为分裂后的内容和之前的长度不同，如果不进行销毁并重新申请会耗费更大的内存。