**B+ Tree**

* **设计思路**
* **查找**

对B＋树可以进行两种查找运算：

1.从最小关键字起顺序查找；

2.从根结点开始，进行随机查找。

在查找时，若非终端结点上的剧组机等于给定值，并不终止，而是继续向下直到叶子结点。因此，在B＋树中，不管查找成功与否，每次查找都是走了一条从根到叶子结点的路径。

* **插入**

B+树的插入仅在叶子节点上进行，当节点中的关键字个数大于m时要分裂成两个结点，他们所包含关键字的个数分别为(m+) / 2和m / 2.并且，它们的双亲结点中应同时包含这两个结点中的最大关键字。如果插入的元素是当前结点的最大值，则需要递归向上更新父节点。

* **实现过程**
* **B+tree数据结构：**

**typedef struct**{  
 **unsigned int** order;  
 **unsigned int** size;  
 **unsigned int** layers;  
 BplusNode \* root;  
 BplusNode \* sqt;  
 **int** (\*cmp)(**const DATATYPE**, **const DATATYPE**);  
}BplusTree;

* **树结点数据结构：**

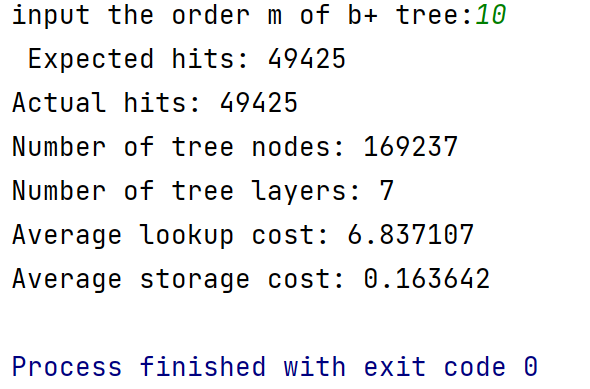
**typedef struct** BplusNode  
{  
 **int** keynum;  
 **KEY** \*keys;  
 **bool** is\_leaf;  
*// DATATYPE \* data;* **struct** BplusNode \*\* child;  
 **struct** BplusNode\* parent;  
 **struct** BplusNode \* next;  
} BplusNode;

* **接口：**

*/\* initialize b+ tree \*/*BplusTree \* init\_bplus\_tree(**unsigned int**, **int** (\*)(**const DATATYPE**, **const DATATYPE**));  
*/\* search the specified string in b+tree, and save the search times \*/***bool** search(BplusTree \*, **DATATYPE** \*, **unsigned int** \*);  
*/\* insert the specified string in b+ tree \*/***void** insert(BplusTree \*, **DATATYPE** \*);  
*/\* destroy the b+tree \*/***void** destroy(BplusTree \*);

* **实验结果**

输入b+树的阶数



更多更具体的测试结果见excel表

**Trie**

* **设计思路**

实现两个数据结构：TrieTree和TrieNode，分别表示前缀树和树节点，前缀树结构需要包括一个指向根节点的指针，以及与树本身相关的数据成员，如树的节点总数，树的阶数等。由于压缩树结构本身与是否使用了压缩技术无关，因此再实现一个数据结构CTrieNode，表示使用了压缩技术的节点，TrieTree中可以使用一个变量标识是否使用了压缩技术，而用一个union结构保存两种不同类型的根节点，C实现代码如下：

*// trie tree***typedef struct** TrieTree  
{  
 **union** {  
 TrieNode \* troot;  
 CTrieNode \* ctroot;  
 } root;  
 **unsigned int** mask\_len; *// 2-tree is 1 4-tree is 2 16-tree is 4...* **unsigned int** size; *// the total number of node* **bool** is\_compress;  
} TrieTree;

其中mask\_len指明，需要根据 多少位来判断下一个子节点的索引，与阶数相关，若阶数为2，则只需要一个bit即可判断。

对于非压缩树节点，需要有一个数组用来保存指向孩子节点的指针；以及一个标识是否该节点可以完全匹配一个已插入的字符串（当搜索到此节点且字符串已经完全匹配完时返回true），节点根据当前字符串的指定开始位判断接下来需要继续匹配的是哪一个孩子节点，因此不需要存储字符（匹配一个字符串时不断根据字符串中的位判断下一个需要处理的树节点是哪一个孩子即可），C实现代码如下：

*// the struct of Trie Tree (non-compress)***typedef struct** TrieNode  
{  
 **struct** TrieNode \*\* children;  
 **bool** is\_key;  
}TrieNode;

对于使用了压缩技术的树节点，字符串是按位段存放在从根（不包括）到叶子节点上的。匹配时，通过当前的位索引开始的指定几位，判断下一个需要进行匹配的子节点，然后继续匹配该子节点，子节点有一段长为n字符位，若完全匹配则继续匹配当前位索引后n位开始的位，并判断从哪个下一子节点继续匹配，因此相比非压缩的前缀树节点，压缩的树节点需要一段内存保存相关的位，以及一个数据成员指明位的长度是多少，C实现代码如下：

**typedef struct** CTrieNode  
{  
 **char** \* data;  
 **unsigned int** len; *// length of data`s bits* **struct** CTrieNode \*\* children;  
 **bool** is\_key;  
}CTrieNode;

* 非压缩前缀树插入过程：
* 初始时设置开始匹配的位索引start为0，根据start开始的且长度为mask\_len的位段来判断下一个匹配的当前节点（初始为根节点）的子节点的索引；
* 若不存在该子节点则创建之，当前节点再移动到该子节点，start前移masl\_len长度，直到start为字符串总位长（8倍字符串长度），相当于每一次匹配一个mask\_len长度的位段。
* 非压缩前缀树查找过程：
* 初始时设置开始匹配的位索引start为0，根据start开始的且长度为mask\_len的位段来判断下一个匹配的当前节点（初始为根节点）的子节点的索引；
* 若不存在该子节点则返回false，反之，当前节点再移动到该子节点，start前移masl\_len长度，直到start为字符串总位长（8倍字符串长度），相当于每一次匹配一个mask\_len长度的位段。
* 匹配完整个字符串后返回true
* 压缩前缀树插入过程：
* 初始时设置开始匹配的位索引start为0，根据start开始的且长度为mask\_len的位段来判断下一个匹配的当前节点（初始为根节点）的子节点的索引。若不存在该子节点，则将剩余的未匹配的位串存入一个新的节点，并设置节点的终结标识为true；
* 反之，当前节点再移动到该子节点，从start开始，匹配当前节点的data字段，该字段存着该节点的一段位串。若完全匹配，则前移该位串的长度，反之，需要分类该节点的位串，将未匹配的部分存入一个新的节点，该节点继承了当前节点的子节点，并设置新的节点为当前节点的一个子节点，将剩余的未匹配的目标字符串存入另一个新节点，设置为当前节点的另一个子节点，并设置终结标识为true。
* 压缩前缀树查找过程：
* 初始时设置开始匹配的位索引start为0，根据start开始的且长度为mask\_len的位段来判断下一个匹配的当前节点（初始为根节点）的子节点的索引。若不存在该子节点，则将剩余的未匹配的位串存入一个新的节点，并设置节点的终结标识为true；
* 反之，当前节点再移动到该子节点，从start开始，匹配当前节点的data字段，该字段存着该节点的一段位串。若完全匹配，则前移该位串的长度，反之，返回false。若待匹配字符串匹配完，则返回当前节点的终结标识
* **实现过程**
* 提供的对外接口：

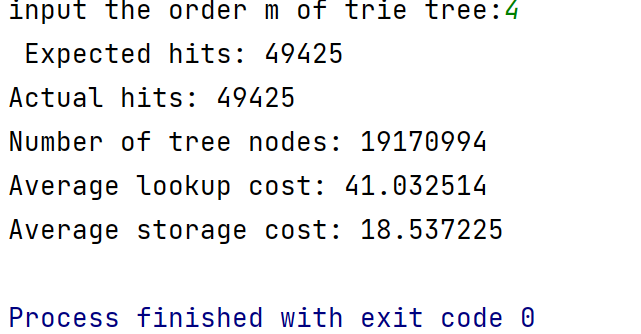
*/\* initialize the trie tree  
 \* and indicate whether compression technology is required \*/*TrieTree \* init\_trie(**unsigned int**, **bool**);  
*/\* search a specified string in the trie tree, and save the compared times \*/***bool** search(TrieTree \*, **const char** \*, **unsigned int** \*);  
*/\* insert a specified string to a trie tree \*/***void** insert(TrieTree \*, **char** \*);  
*/\* destroy the trie tree \*/***void** destroy(TrieTree \*);

* 内部辅助函数：

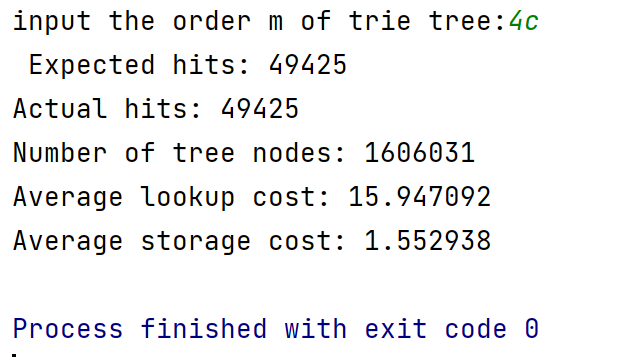
*/\* 局部函数声明 \*/  
/\* allocate an uncompressed node \*/***static** TrieNode \* allocate\_node(TrieTree \*, **unsigned int**);  
*/\* destroy an uncompressed node \*/***static void** destroy\_node(TrieNode\*, **unsigned int**);  
  
*/\* allocate a compressed node \*/***static** CTrieNode \* allocate\_cnode(TrieTree \*, **unsigned int**);  
*/\* destroy a compressed node \*/***static void** destroy\_cnode(CTrieNode\*, **unsigned int**);  
*/\* returns the number of bits matching the bit segment of the specified string and the specified node \*/***static unsigned int** match\_bits(CTrieNode \*, **const char** \*, **unsigned int**, **unsigned int**);  
*/\* gets the index of the specified string from the specified length of the specified positioning index \*/***static unsigned int** get\_index(**const char** \*, **unsigned int**, **unsigned int**, **unsigned int**);  
*/\* Split String Bitwise \*/***static void** split\_string(**char** \*\*, **char** \*\*, **unsigned int**, **unsigned int**, **bool**);  
  
*/\* uncompressed trie tree search \*/***static bool** search\_nonc(TrieTree \*, **const char** \*, **unsigned int** \*);  
*/\* uncompressed trie tree insert \*/***static void** insert\_nonc(TrieTree \*, **char** \*);  
  
*/\* compressed trie tree search \*/***static bool** search\_c(TrieTree \*, **const char** \*, **unsigned int** \*);  
*/\* compressed trie tree insert \*/***static void** insert\_c(TrieTree \*, **char** \*);

* **实验结果**

输入4，表示阶数为4且未使用压缩技术：



输入4c，表示阶数为4且使用压缩技术：



具体更多的测试结果见excel统计表