

计算机应用编程实验

树结构字符串检索

熊永平@网络技术研究院

ypxiong@bupt.edu.cn

教三楼132

2020.9

实验任务

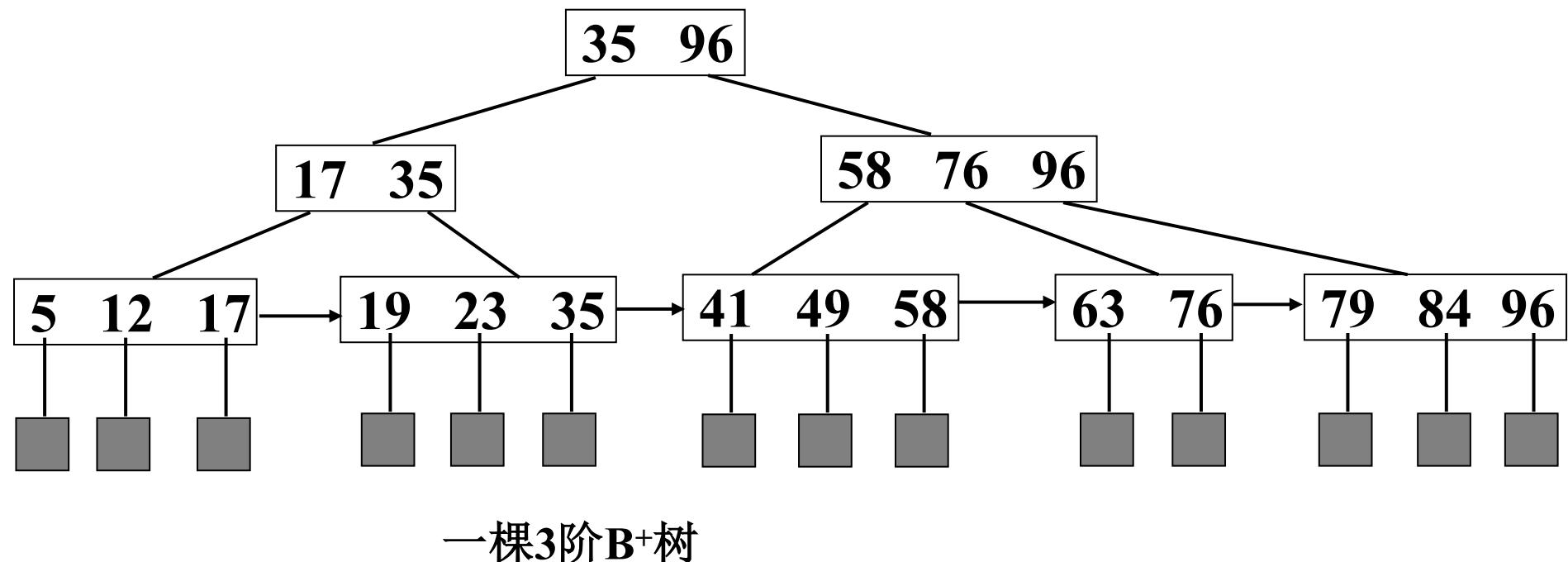
- 利用树形结构实现大规模字符串查找
 - B+树
 - Trie树
 - 多叉搜索树
 - Radix树

B⁺树

- 数据库系统中广泛使用的索引结构为m阶B⁺树。
- 主要特点是叶子结点中存储记录。
- 在B⁺树中，所有的非叶子结点可以看成是索引，而其中的关键字是作为分界关键字界定某一关键字的记录所在的子树。
- 平衡的m路搜索树
- m阶B⁺树
 - (1) 若一个结点有n棵子树，则必含有n个关键字；
 - (2) 所有叶子结点中包含了全部记录的关键字信息以及这些关键字记录的指针，且叶子结点按关键字的大小从小到大顺序链接；
 - (3) 所有的非叶子结点可以看成是索引的部分，结点中只含有其子树的根结点中的最大(或最小)关键字。

B+树示例

由于B⁺树的叶子结点和非叶子结点结构上的显著区别，因此需要一个标志域加以区分，结点结构定义如下：



B+树构造

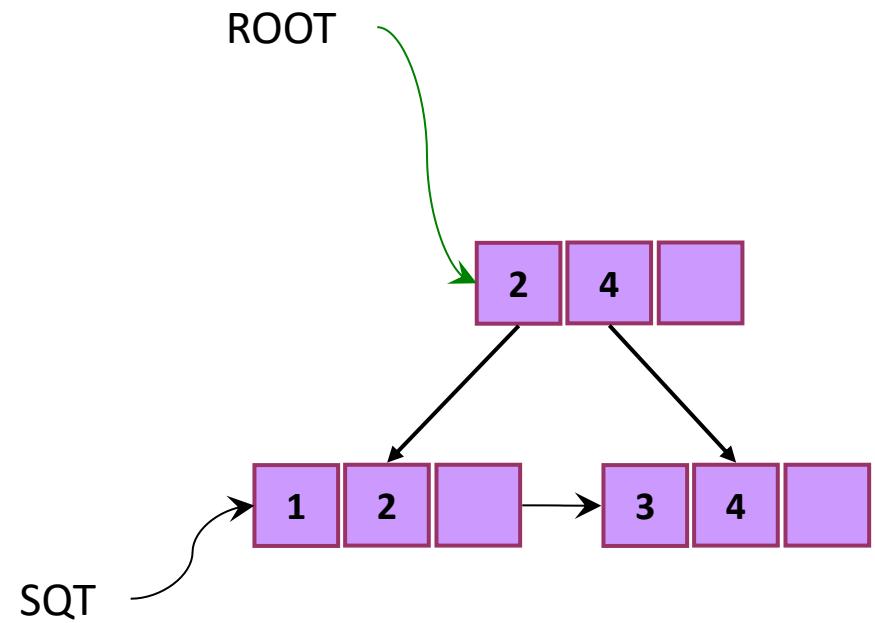
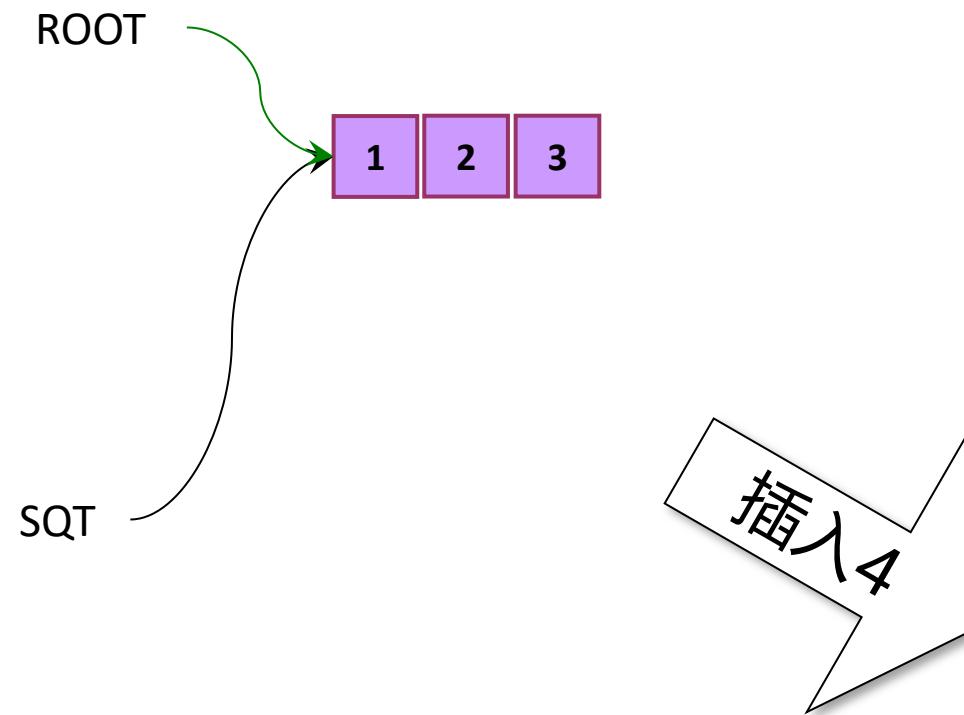
- **B⁺树的插入**

- B⁺树的插入仅在叶子结点上进行，当结点中的关键字个数大于m时要分裂成两个结点，它们所含关键字的个数分别为 $\left\lceil \frac{m+1}{2} \right\rceil$ 和 $\left\lceil \frac{m}{2} \right\rceil$ 。
- 他们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节点的最小值或最大值，需要递归向上更新父节点。
- 生成新节点“左顾右盼”，尽最大可能节省内存空间，不生成额外的新结点，除非所有的节点都满了。

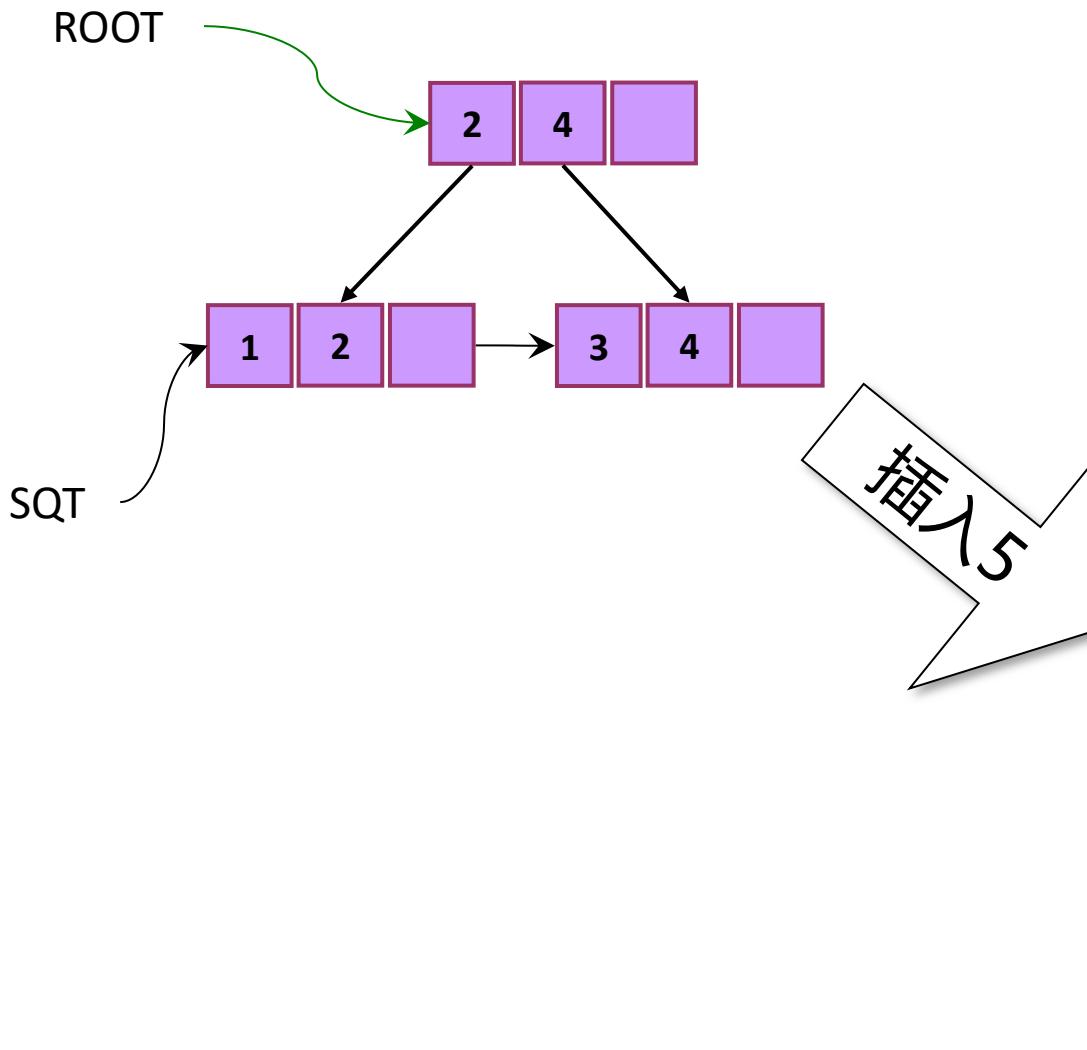
- **B⁺树的查找**

- 两种查找运算：从最小关键字起顺序查找；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。

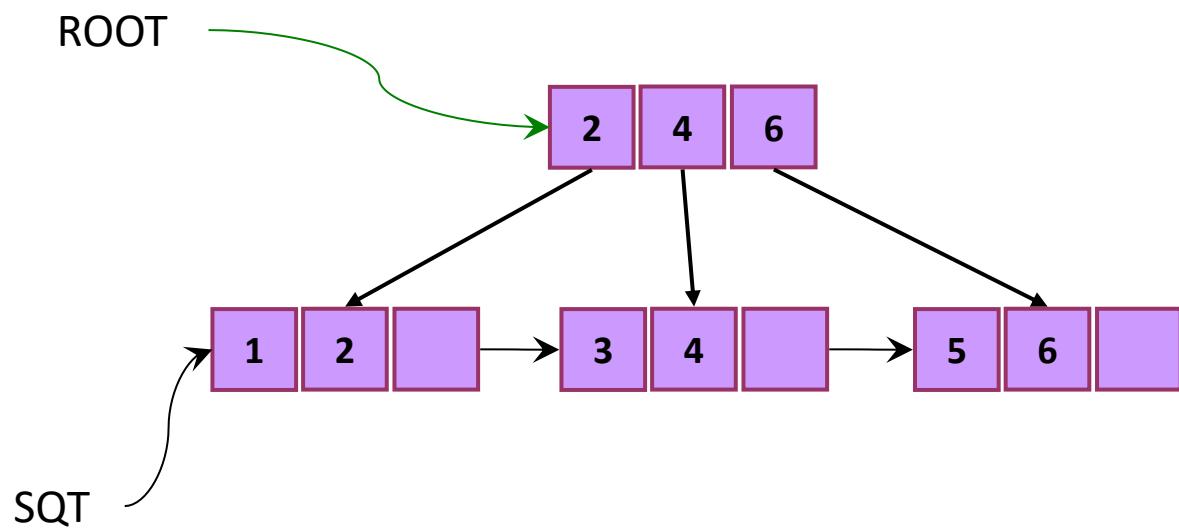
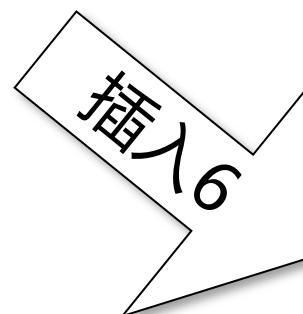
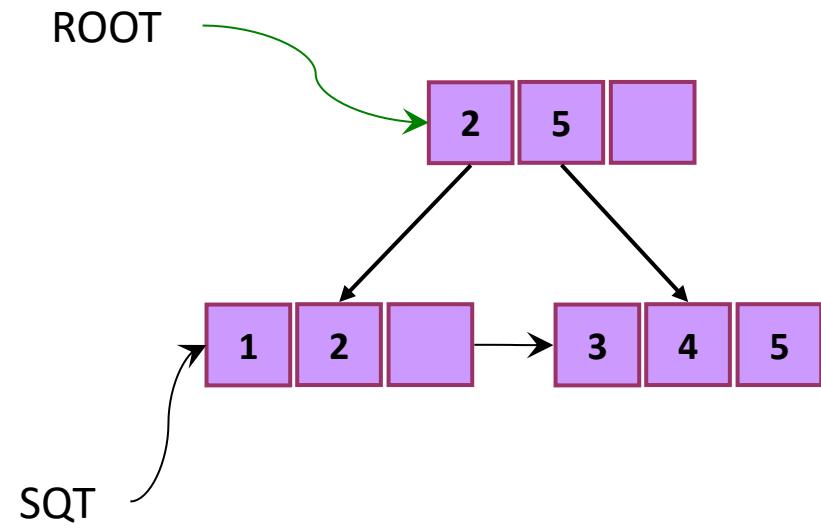
B+树插入



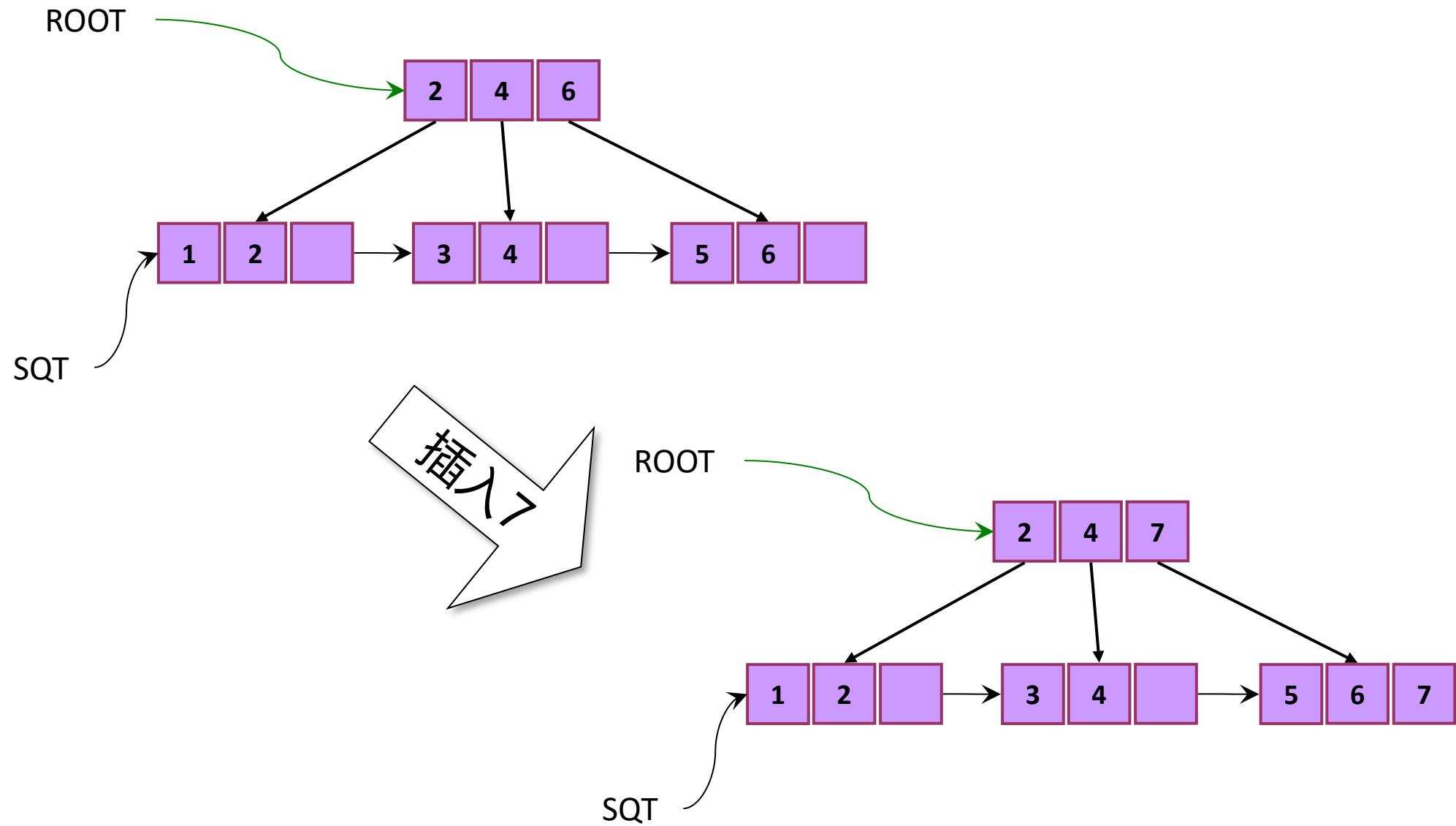
B+树插入



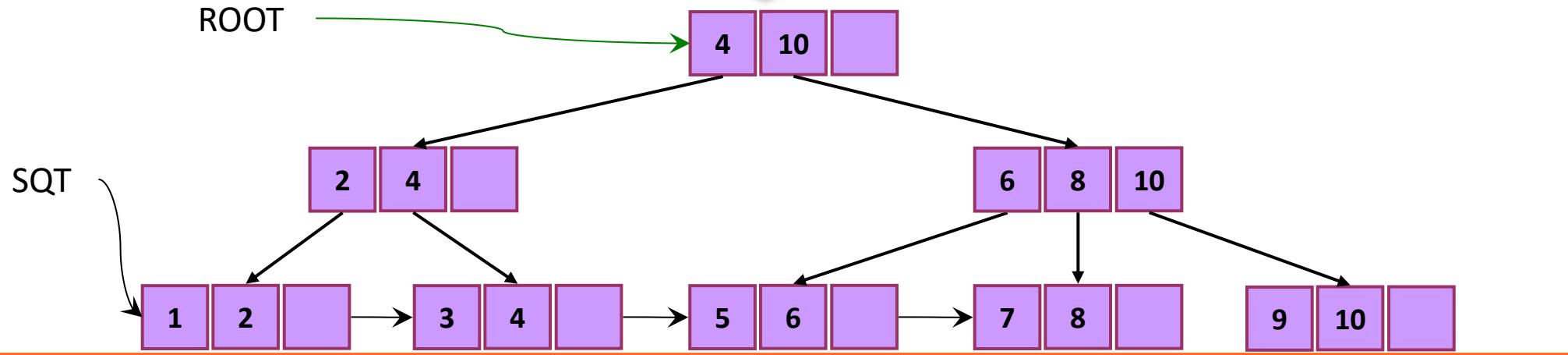
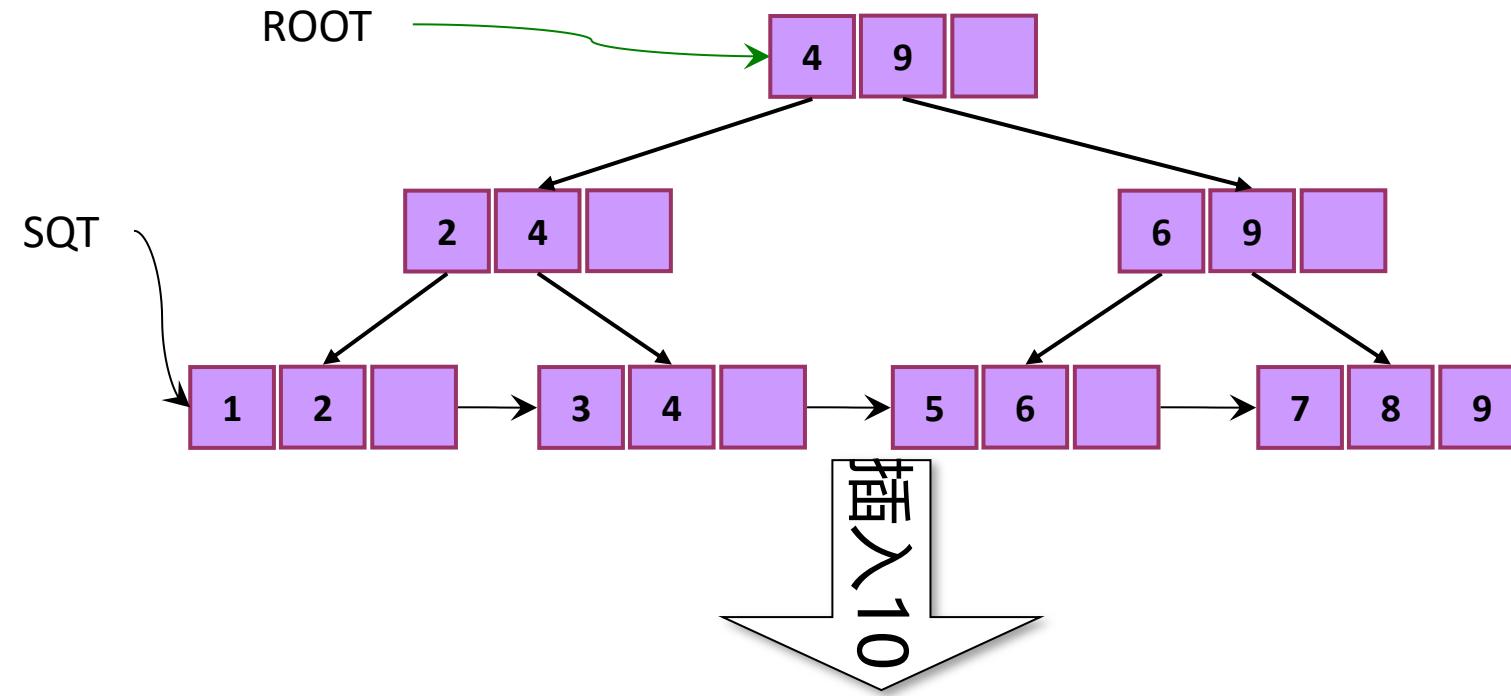
B+树插入



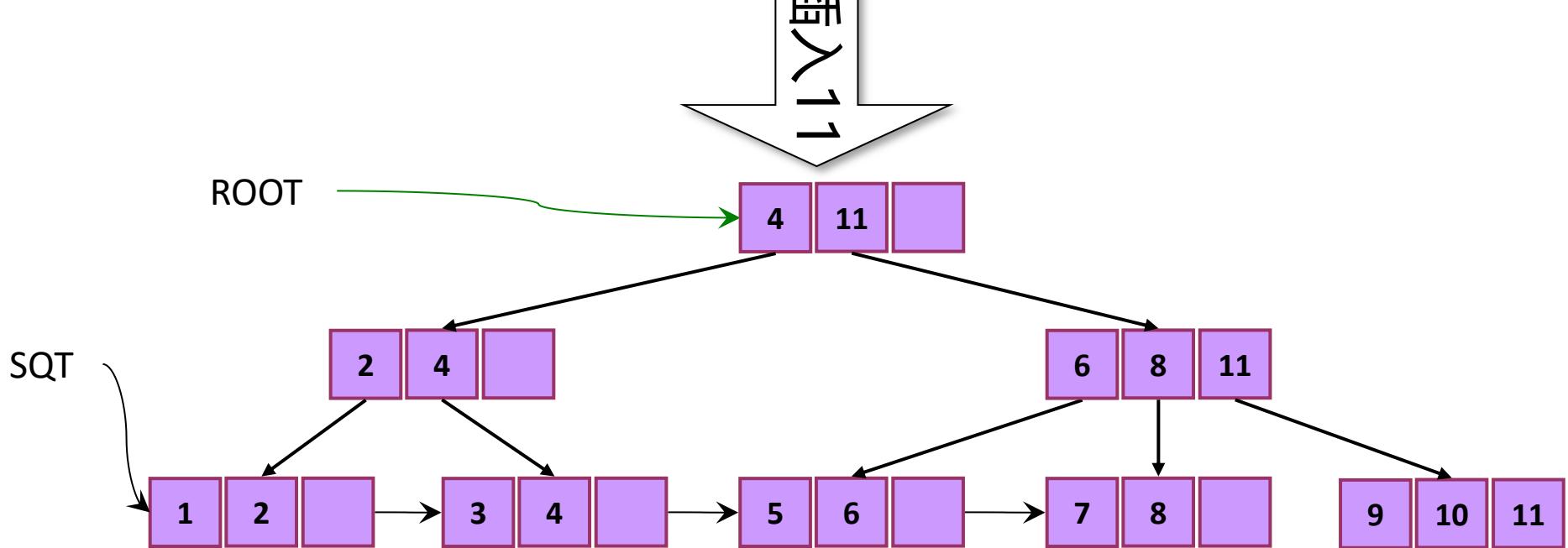
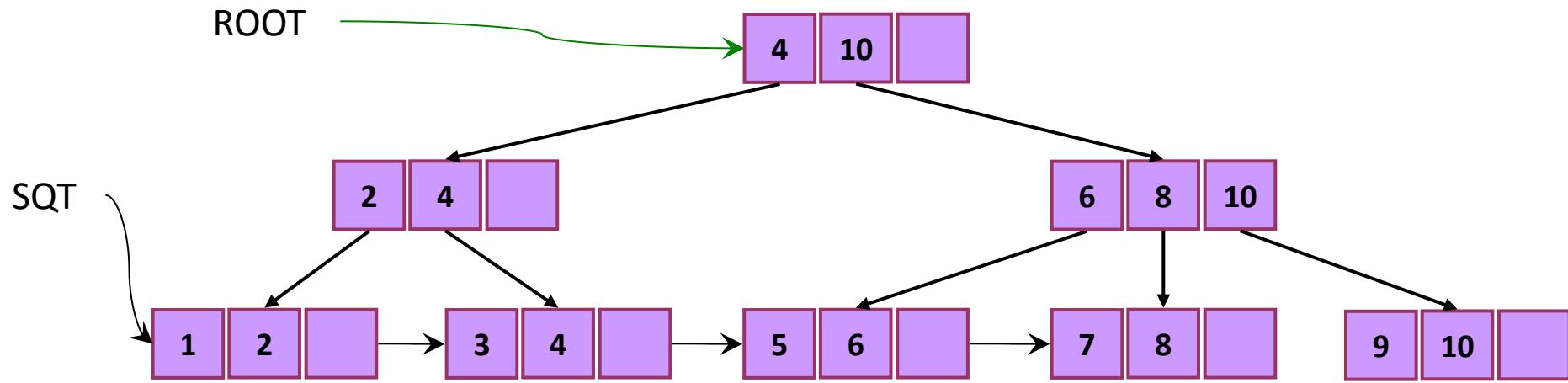
B+树插入



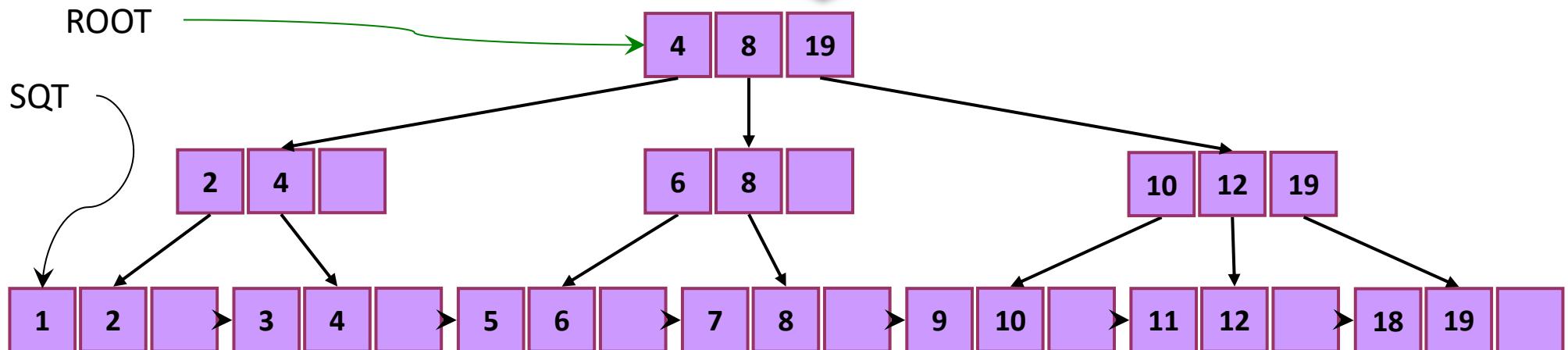
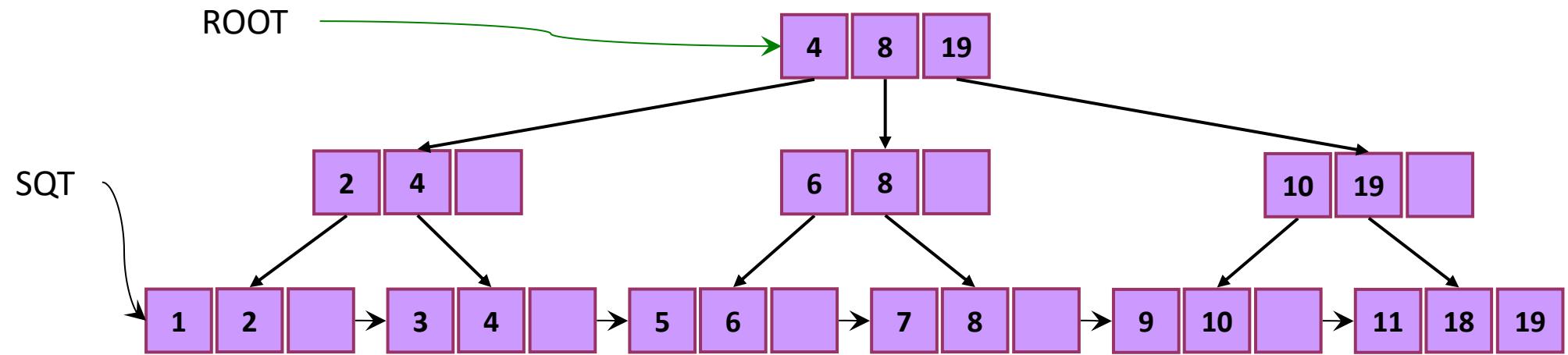
B+树插入



B+树插入



B+树插入

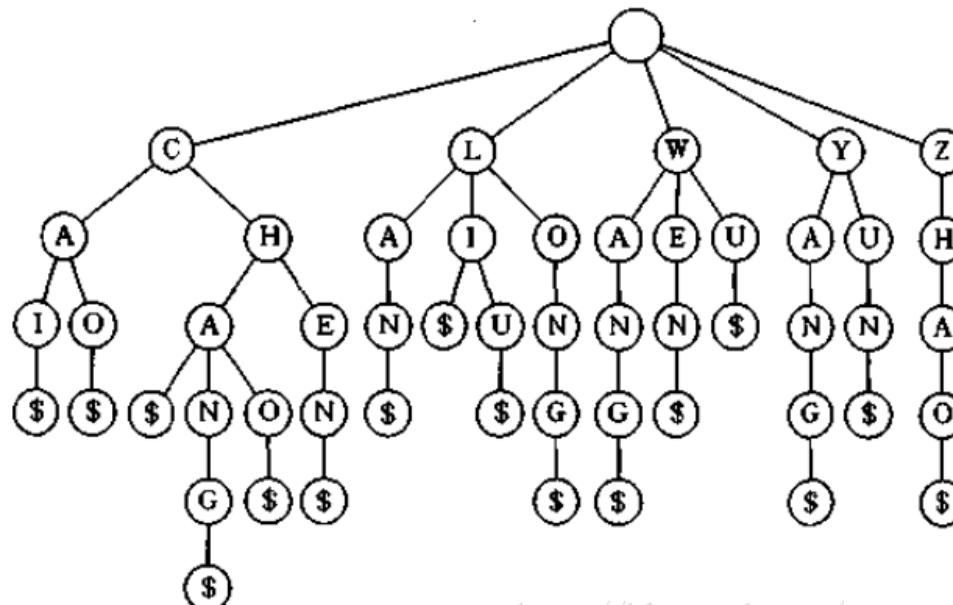


参考实现

```
typedef struct bplus_node{
    int keynum;
    int* keys; //用来存储数据
    //如果是叶子结点是存储数据
    //如果是中间结点存储的是key值
    struct bplus_node** child;
    //需要注意的是,这里的child是一个指针数组
    //类型struct bplus_node* *child是数组
    //其实就是个二维数组,但其中一维只存了一个元素
    struct bplus_node* parent;
    struct bplus_node* next;
}Node,Tree;
```

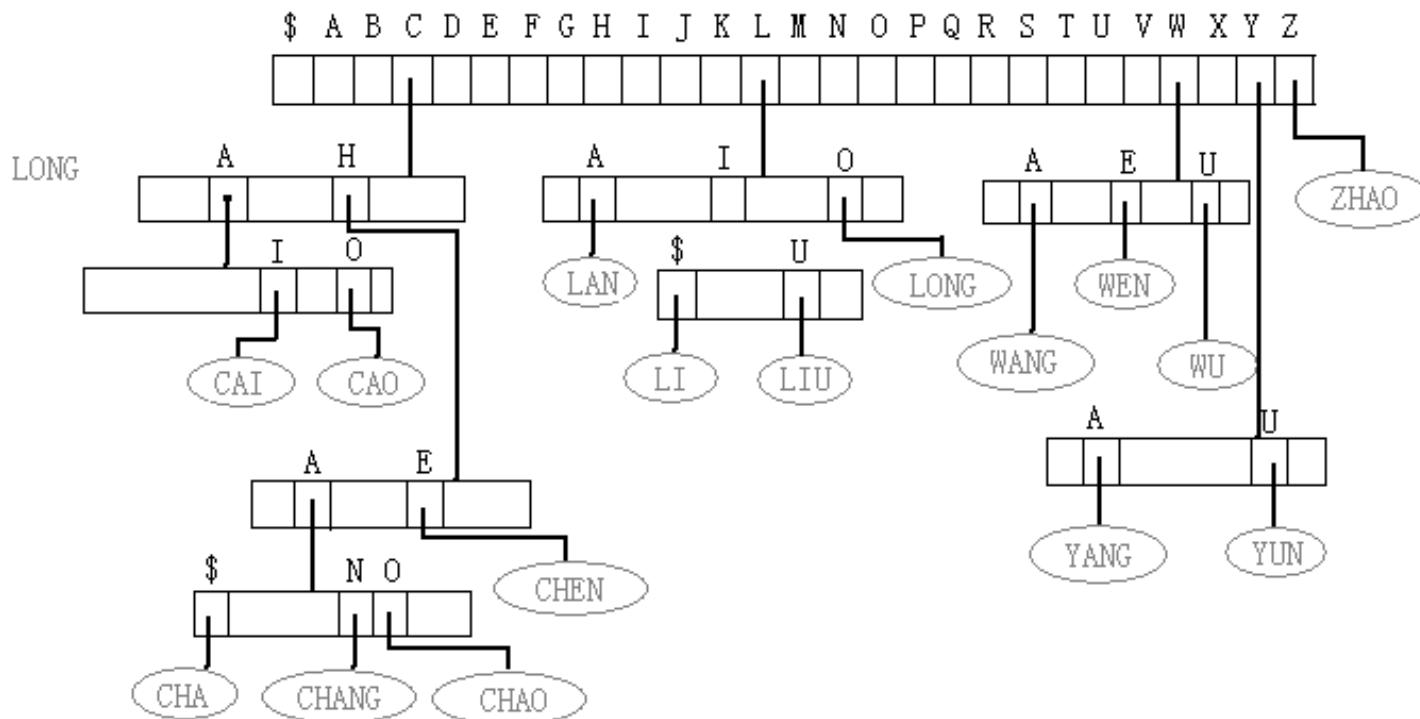
Trie树

- 定义
 - 又称单词字典树，是一种树形结构，是一种用于快速检索的多叉树结构
- 存储词典
 - { CAI、CAO、LI、LAN、CHA、CHANG、WEN、CHAO、YUN、YANG、LONG、WANG、ZHAO、LIU、WU、CHEN }
 - 树的高度为最长字符串长度



Trie数据结构示例

- 数据结构实现
 - 分支结点：含有d个指针域和一个指示该结点中非空指针域的个数的整数域。
 - 分支结点所表示的字符是由其指向子树指针的索引位置决定的叶子结点：含有关键字域和指向记录的指针域。



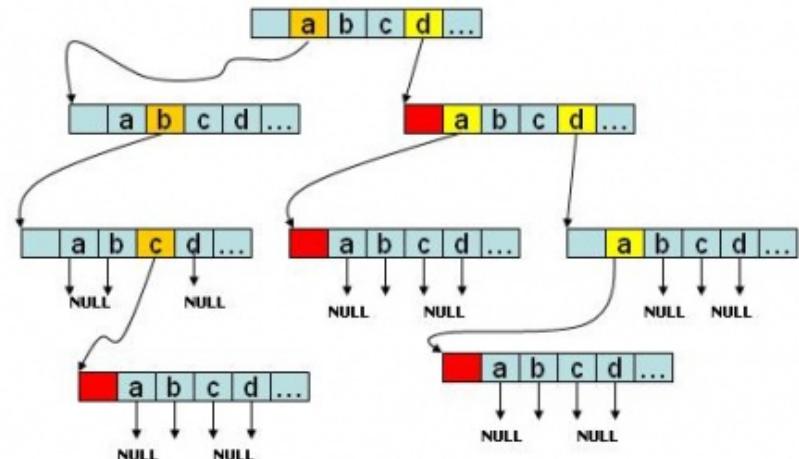
```
Typedef Struct TrieNode{  
    NodeKind kind ;  
    union {  
        struct {KeyType K; Record *infoptr}  
        If ; //叶子结点  
        struct {TrieNode *ptr[27]; int num}  
        bh ; //分支结点};  
    } TrieNode,*TrieTree ;
```

Trie树的插入

- 插入
 - ▣ 首先根据插入纪录的关键码找到需要插入的结点位置
 - ▣ 如果该结点是叶结点，那么就将为其分裂出两个子结点，分别存储这个纪录和以前的那个纪录
 - ▣ 如果是内部结点，则在那个分支上应该是空的，所以直接为该分支建立一个新的叶结点即可

Trie查找

- 查找
 - 在Trie树上进行检索总是始于根结点。
 - 取得要查找关键词的第一个字母，并根据该字母选择对应的子树并转到该子树继续进行检索。
 - 在某个结点处相应的子树上，取得要查找关键词的第二个字母，并进一步选择对应的子树进行检索。
 - 关键词的所有字母已被取出，则读取附在该结点上的信息，即完成查找
- 示例
 - trie树中保存了abc、d、da、dda四个单词



Trie查找效率分析

- 在trie树中查找一个关键字的时间和树中包含的结点数无关，而取决于组成关键字的字符数。
- 对比：二叉查找树的查找时间和树中的结点数有关 $O(\log_2 n)$ 。
- 如果要查找的关键字可以分解成字符序列且不是很长，利用trie树查找速度优于二叉查找树。
 - 若关键字长度最大是5，则利用trie树，利用5次比较可以从 $26^5 = 11881376$ 个可能的关键字中检索出指定的关键字。而利用二叉查找树至少要进行 $\log_2 26^5 = 23.5$ 次比较。

Trie结构总结

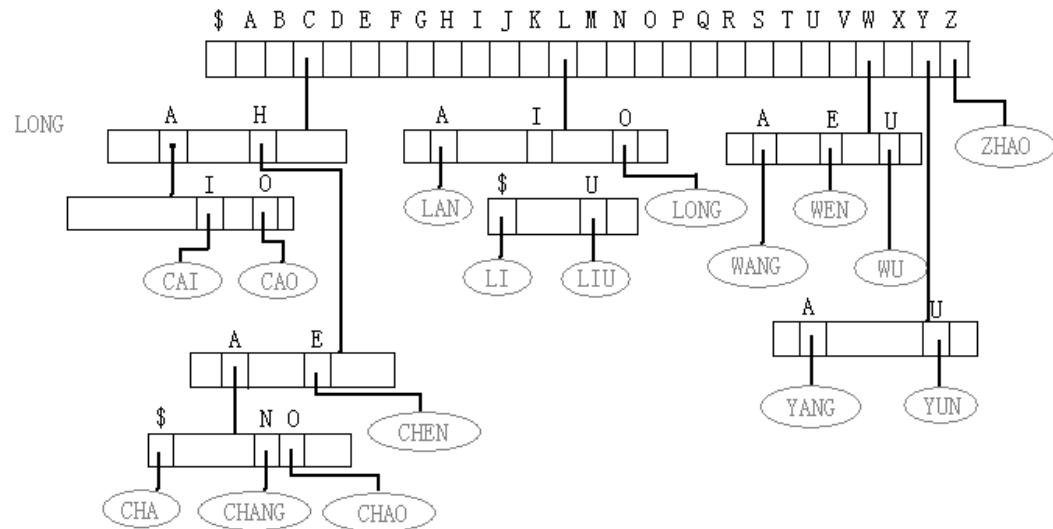
- 核心思想
 - 空间换时间
 - 利用字符串的公共前缀来降低查询时间的开销以达到提高效率的目的
- 优点
 - 查找效率高，与词表长度无关
 - Trie树的查找效率只与关键词长度有关
 - 索引的插入，合并速度快
- 缺点
 - 内存空间消耗大
 - 如果是完全m叉树，节点数指数级增长
 - 不可达上限：词数 \times 字符序列长度 \times 字符集大小 \times 指针长度
 - 例如： $20000 \times 6 \times 256 \times 4 = 120M$
 - 实现较复杂

用Trie来做中文字符串查找？

- ## ● 来一个256叉的Trie树

```
struct trienode_t  
{  
    char num;  
    struct trienode_t *child[256];  
};
```

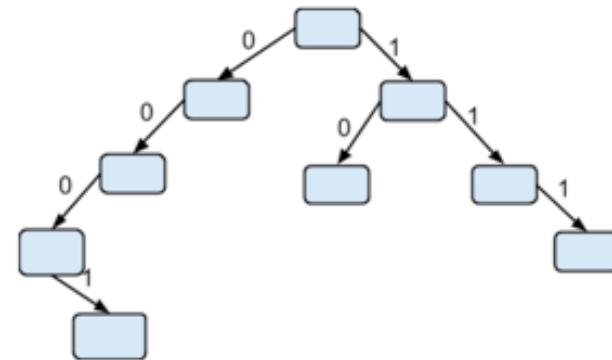
- 127万个字符串内存占用约4G
 - 分支太多？



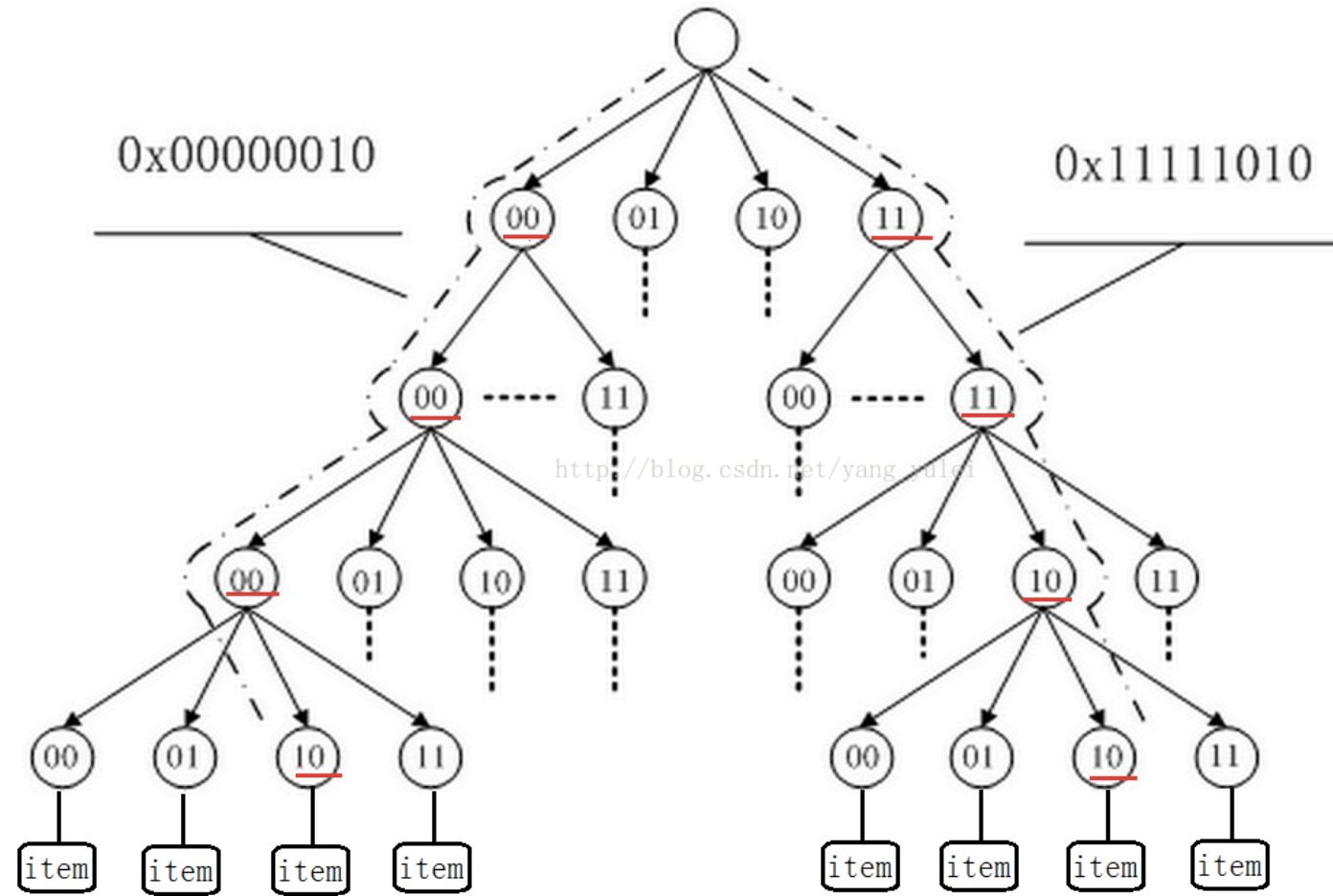
Trie优化1：m叉查找树

- Trie结构缺点
 - Trie结构显然不是平衡的
 - 存取英文单词时，显然t子树下的分支比z子树下的分支多很多
 - 26个分支因子使得树的结构过于庞大，检索不便
- M叉Trie (“Practical Algorithm To Retrieve Information Coded In Alphanumeric”)
 - 关键码二进制形式存储
 - 根据关键码每个二进制位的编码来划分
 - 是对整个关键码大小范围的划分

每个内部结点都代表一个位的比较，必然产生两个子结点，所以它是一个满二叉树，进行一次检索，最多只需要关键码位数次的比较即可。

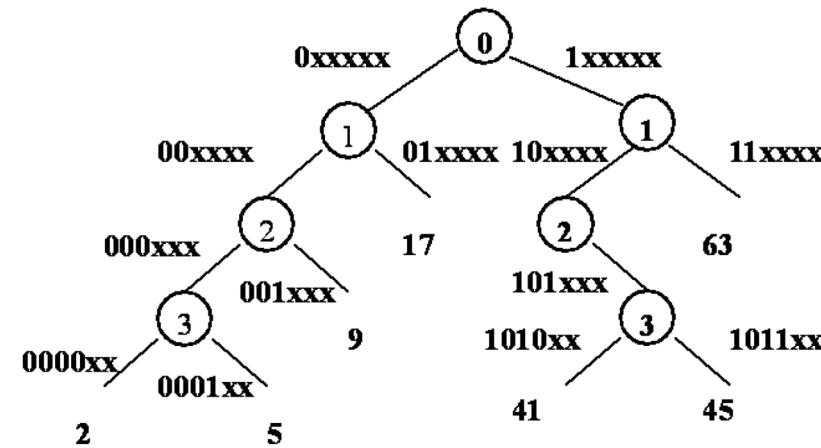


4叉查找树



2叉查找树

- 举例（2、5、9、17、41、45、63）
 - 因为最大的数是63，用6位二进制表示
 - 每个结点都有一个标号，表示它是比较第几位，然后根据那一位是0还是1来划分左右两个子树
 - 标号为2的结点的右子树一定是编码形式为 $xx1xxx$ ，（x表示该位或0或1，标号为2说明比较第2位）
 - 在图中检索5的话，5的编码为000101
 - 首先我们比较第0位，从而进入左子树，然后在第1位仍然是0，还是进入左子树，在第2位还是0，仍进入左子树，第3位变成了1，从而进入右子树，就找到了位于叶结点的数

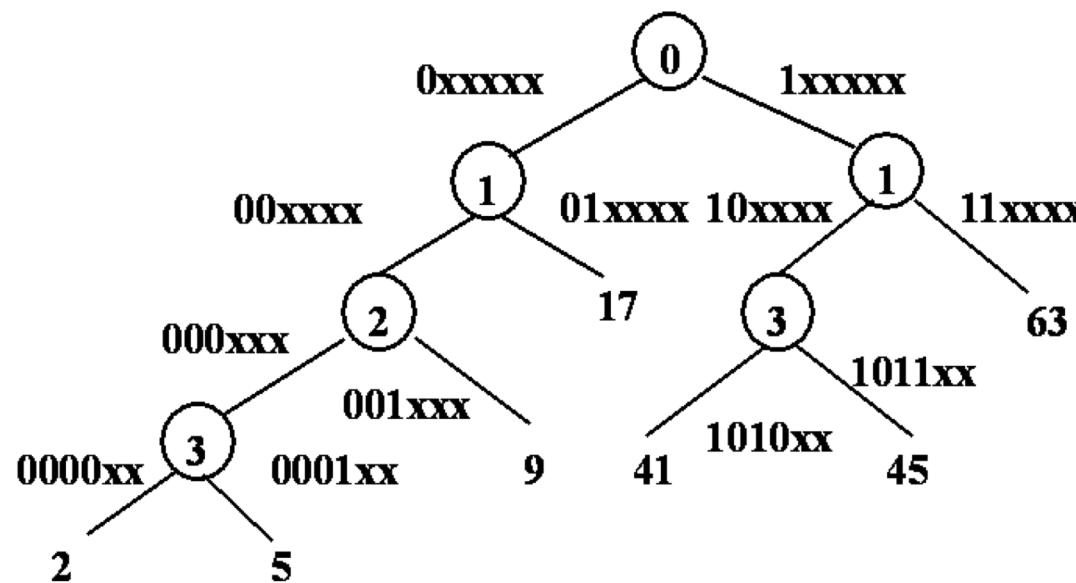


编码：
2: 000010 5: 000101 9: 001001
17: 010001 41: 101001 45: 101101 63: 111111

2叉查找树路径优化

- 优化

- 在区分2和5、41和45时，第3个二进制位的比较不能区别它们，可以将它省略，得到一棵更为简洁的树。

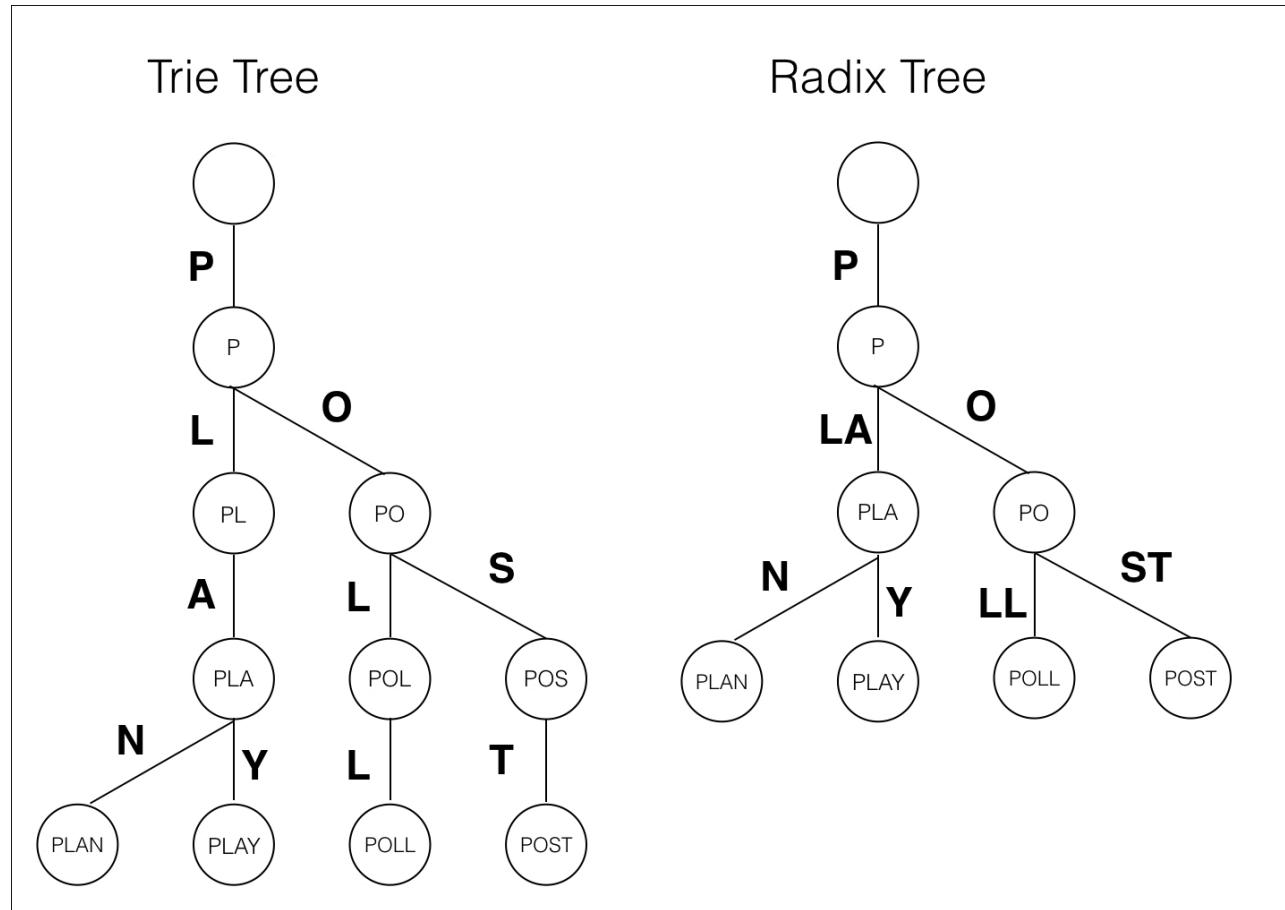


编码: 2: 000010 5: 000101 9: 001001

17: 010001 41: 101001 45: 101101 63: 111111

Trie树演进2： Radix Tree

- 定义



- 核心思想

- 压缩路径

Radix树

- Radix树是基于二进制的
- 先挑几个字母将其转换为二进制作为例子

字符	a	b	c	A
二进制	0b1100001	0b1100010	0b1100011	0b1000001
从低到高	10000110	01000110	11000110	10000010

字符串	二进制流
abc	100001100100011011000110
aac	100001101000011011000110
Aac	100000101000011011000110

Radix树

- 首先插入第一个二进制串

1000011001000110110001

10

Radix树

- 插入第二个二进制串

1000011010000110110001

10

1000011001000110110001

10

Radix树

- 得到他们的公共前缀

1000011010000110110001

10

1000011001000110110001

10

Radix树

- 将公共前缀独立为新节点，并重置两个结点的内容

10000110110001

10

01000110110001

10

10000110

Radix树

- 按照节点的最前面的k个比特内容，作为节点的序号

10000110

01000110110001
10

10000110110001
10



Radix树

- 插入第三个节点

1000001010000110110001
10

10000110

01000110110001
10

10000110110001
10

Radix树

- 得到公共前缀

1000001010000110110001
10

10000110

01000110110001
10

10000110110001
10

Radix树

- 将公共前缀独立为单独的节点

001010000110110001
10

0110

1000

01000110110001
10

10000110110001
10

Radix树

- 同样将其插入到新节点对应的位置即可

1000

001010000110110001
10

0110

0100011
0110001
10

1000011
0110001
10

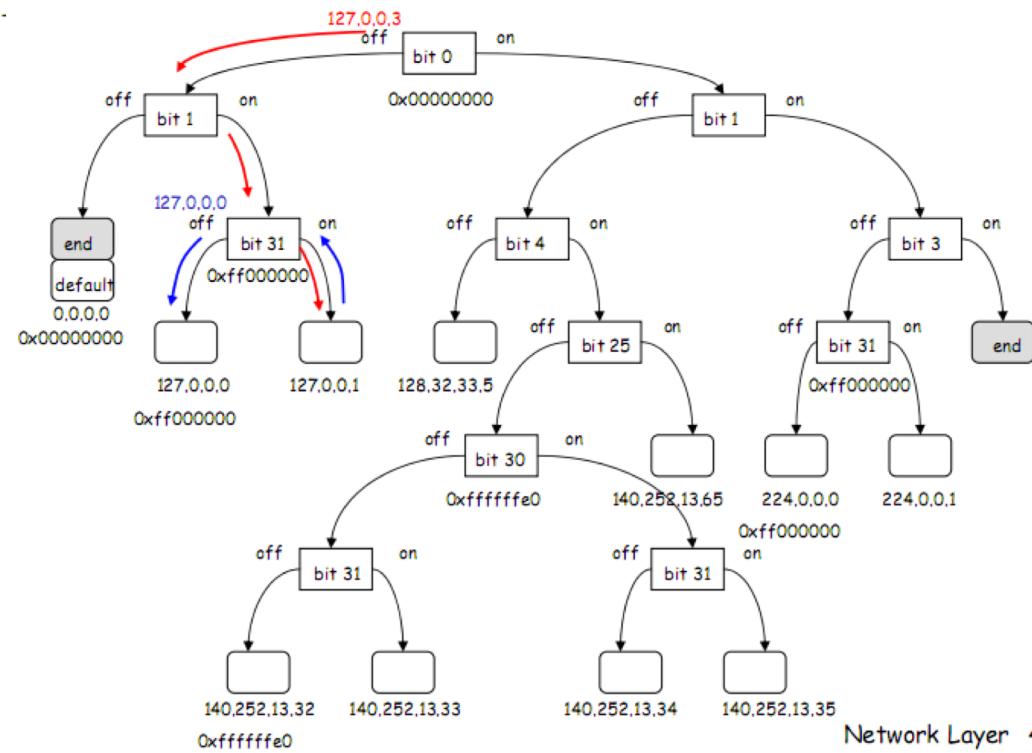
Radix应用：路由表查找

➤ Routing Table

Destination	Gateway	Flags	Ref	Use	Interface
default	140.252.13.33	UGS	0	3	le0
127	127.0.0.1	UGSR	0	2	lo0
127.0.0.1	127.0.0.1	UH	1	55	lo0
128.32.33.5	140.252.13.33	UGHS	2	16	le0
140.252.13.32	link#1	UC	0	0	le0
140.252.13.33	8:0:20:3:f6:42	UHL	11	55146	le0
140.252.13.34	0:0:c0:c2:9b:26	UHL	0	3	le0
140.252.13.35	0:0:c0:6f:2d:40	UHL	1	12	lo0
140.252.13.65	140.252.13.66	UH	0	41	s10
224	link#1	UC	0	0	le0
224.0.0.1	link#1	UHL	0	5	le0

32-bit IP address (bits 32-63)	
Bit:	0123 4567 8911 1111 1111 2222 2222 2233 01 2345 6789 0123 4 567 8901
	0000 0000 0000 0000 0000 0000 0000 0000 0.0.0.0 0111 1111 0000 0000 0000 0000 0000 0000 127.0.0.0 0111 1111 0000 0000 0000 0000 0000 0001 127.0.0.1 1000 0000 0010 0000 0010 0001 0000 0101 128.32.33.5 1000 1100 1111 1100 0000 1101 0010 0000 140.252.13.32 1000 1100 1111 1100 0000 1101 0010 0001 140.252.13.33 1000 1100 1111 1100 0000 1101 0010 0010 140.252.13.34 1000 1100 1111 1100 0000 1101 0010 0011 140.252.13.35 1000 1100 1111 1100 0000 1101 0100 0001 140.252.13.65 1110 0000 0000 0000 0000 0000 0000 0000 224.0.0.0 1110 0000 0000 0000 0000 0000 0000 0001 224.0.0.1

查找127.0.0.3



Linux内核的Radix Tree

- Linux内核的应用

- 管理内存分配
- 缓存区映射
- R=4

```
1 struct radix_tree_node {  
2     unsigned int    path;  
3     unsigned int    count;  
4     union {  
5         struct {  
6             struct radix_tree_node *parent;  
7             void *private_data;  
8         };  
9         struct rcu_head rCU_head;  
10    };  
11    /* For tree user */  
12    struct list_head private_list;  
13    void __rcu    *slots[RADIX_TREE_MAP_SIZE];  
14    unsigned long  tags[RADIX_TREE_MAX_TAGS][RADIX_TREE_TAG_LONGS];  
15};
```

程序要求

- 分别实现四个版本，程序运行参数：
 - bplus_search m dict.txt string.txt
 - m阶B+树
 - rawtrie dict.txt string.txt
 - 原始256叉树
 - mtrie m dict.txt string.txt
 - m是节点分支数
 - radix_search dict.txt string.txt
 - 2叉压缩树
 - 输入输出文件格式与之前相同
 - 实验报告给出创建检索结构后的内存占用量

报告要求

- 实验报告
 - 主要数据结构和流程
 - 实验过程
 - 遇到的问题
 - 结果指标: cpu 内存 准确率等
 - 结论和总结

