

高级计算机编程实战

树结构字符串检索

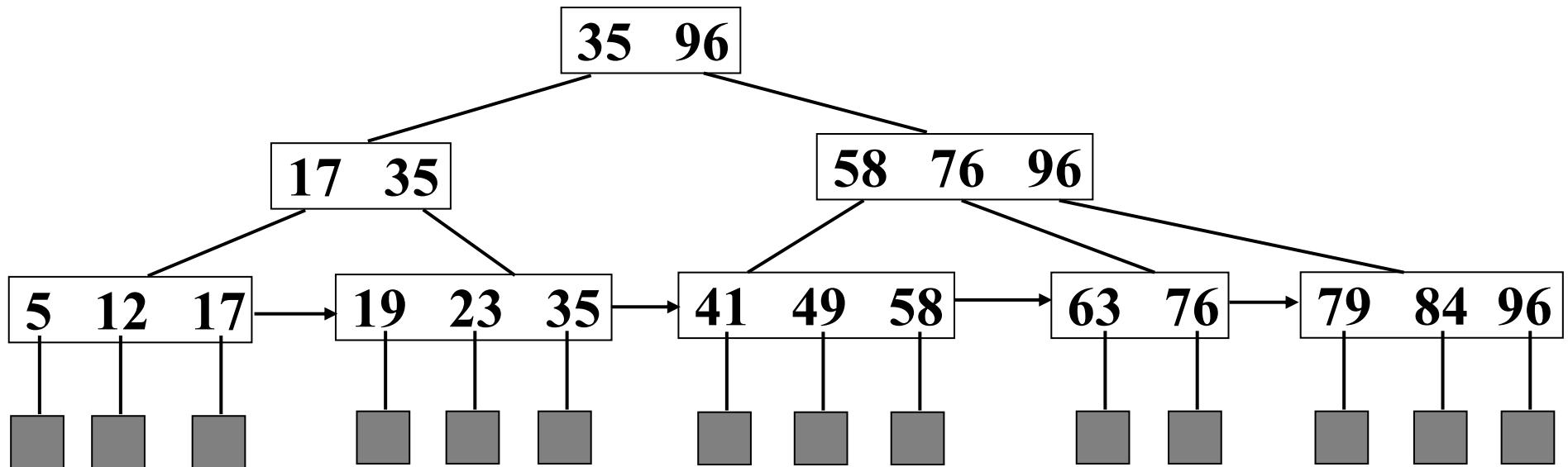
熊永平@计算机学院

实验任务

- 利用树形结构实现大规模字符串查找
 - B+树
 - 多叉Trie树

B+树

- B+树通常用于数据库和操作系统的文件系统中。NTFS、ReiserFS、NSS、XFS、JFS、ReFS和BFS等文件系统都在使用B+树作为元数目索引。
- 数据库的索引结构如MySQL的InnoDB存储引擎使用B+树结构。
- B+树的特点是能够保持数据稳定有序，其插入与修改拥有较稳定的对数时间复杂度。
- B+树元素自底向上插入。



一棵3阶B+树

B+树数据结构

B+树的节点可分为叶子与非叶子节点（内部节点），叶子节点主要是用于存储数据，而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”，用来界定某一关键字的记录所在的子树。

- B+树中的节点通常被表示为一组有序的元素和子指针。
- 如果此B+树的阶数是m，则除了根之外的每个节点都包含**最少** $[m/2]$ 个元素**最多** [m]个元素，对于任意的结点有最多m个子指针。
- 对于所有内部节点，子指针的数目总是与元素的数目相同。
- 所有叶子都在相同的高度上，叶结点本身按关键字大小从小到大链接

```
//定义B+树的节点数据结构
typedef struct bplus_node {
    int keynum; //该节点的关键字的个数，可以看作keys[]的逻辑边界
    KEY* keys; //主键数组，最大max=m个,最小min=m/2个,空间
    max+1, 以备分裂操作时的中间步骤使用
    /*int* data; //存放数据的数组，最大max个,最小min个,空间max+1,
内部节点时，data是NULL
本例中使用hash函数处理数据，keys就代表了要存放的数据，故不需要
data字段*/
    struct bplus_node** child; //子节点的指针数组，同时用于判断是否
    是叶子节点
    struct bplus_node* parent;
    struct bplus_node* next; //兄弟节点，仅在叶子节点内有意义
}bplus_node, * bplus_tree;
```

B+树构建

- **B⁺树的查找**

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。

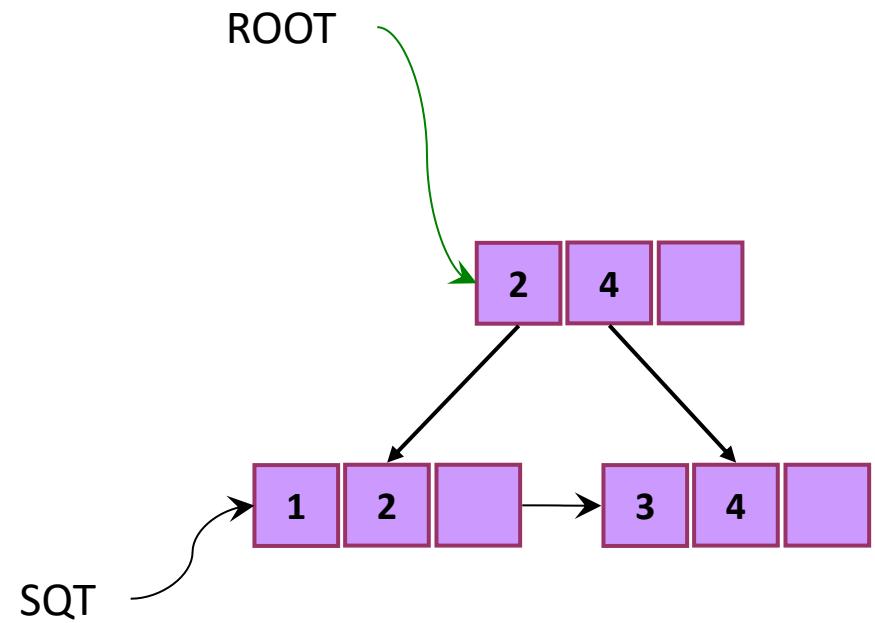
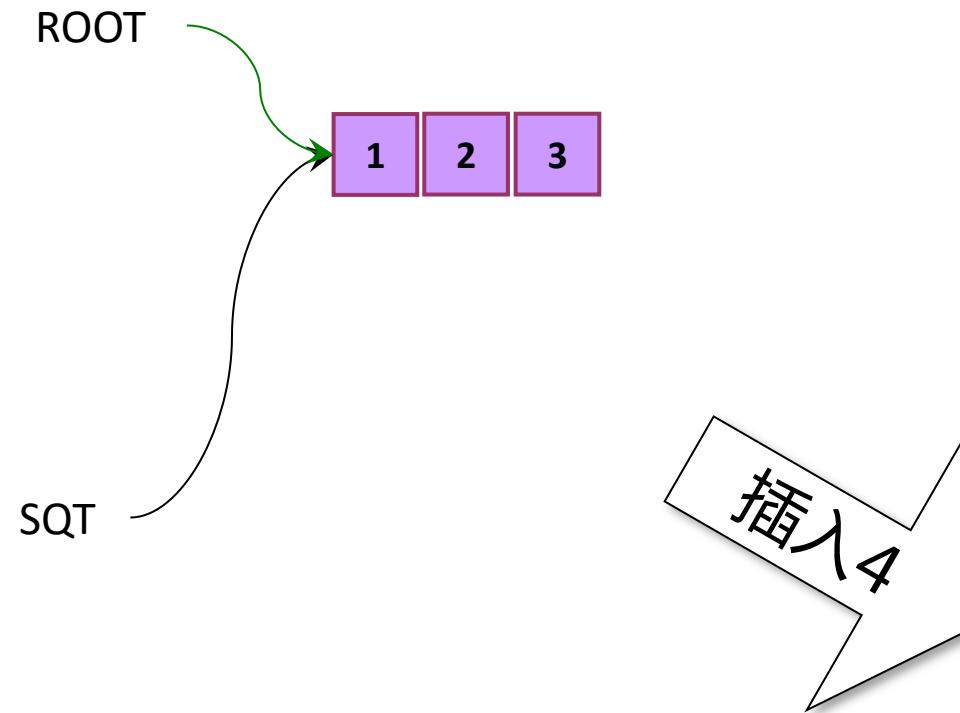
- **B⁺树的插入**

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 $\left\lceil \frac{m}{2} \right\rceil$ 。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节点的**最小值或最大值**，需要**递归向上更新父节点**。

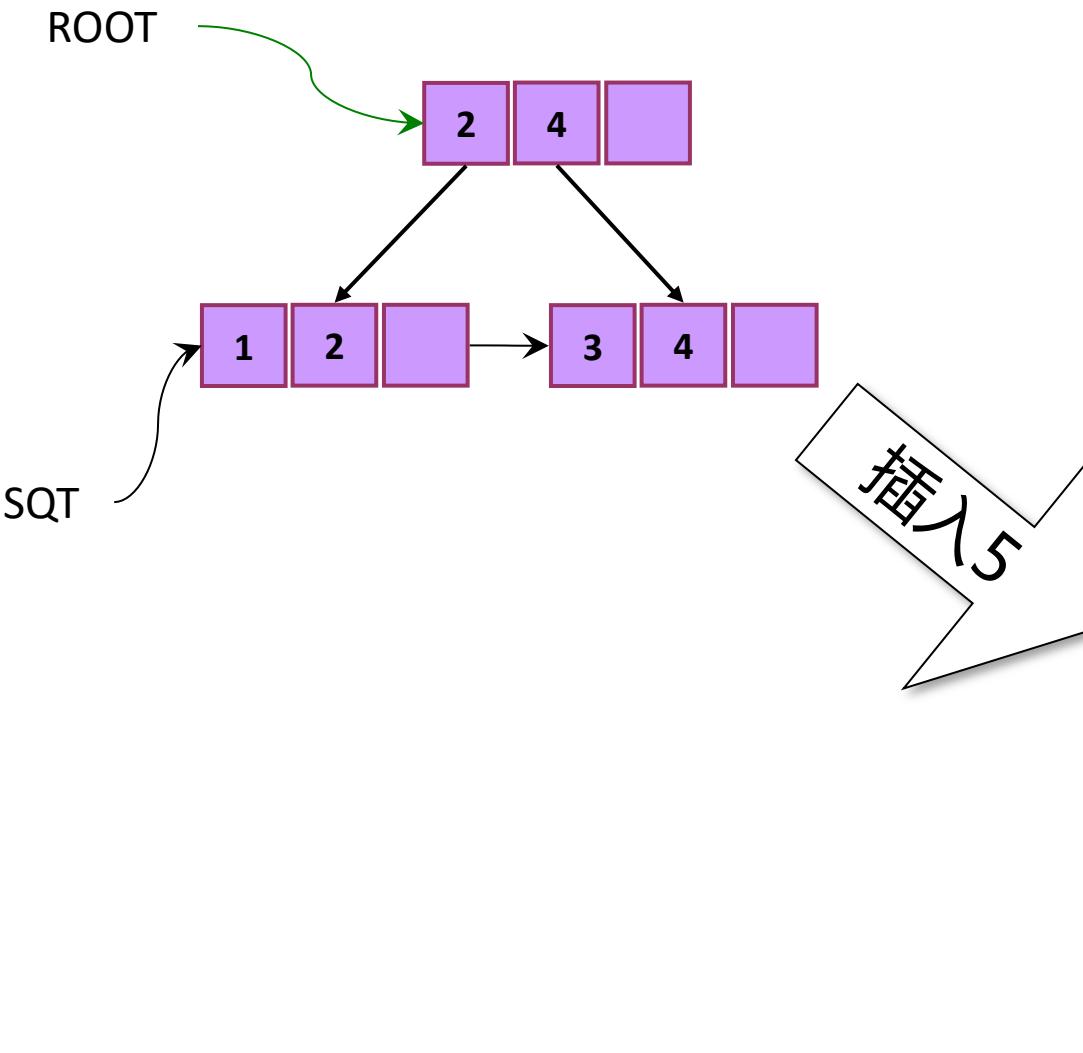
- **B⁺树的删除**

B⁺树的删除也仅在叶子结点进行，当叶子结点中的最大关键字被删除时，其在非终端结点中的值可以作为一个“分界关键字”存在。若因删除而使结点中关键字的个数少于 $\left\lceil \frac{m}{2} \right\rceil$ 时，和兄弟结点**合并**。

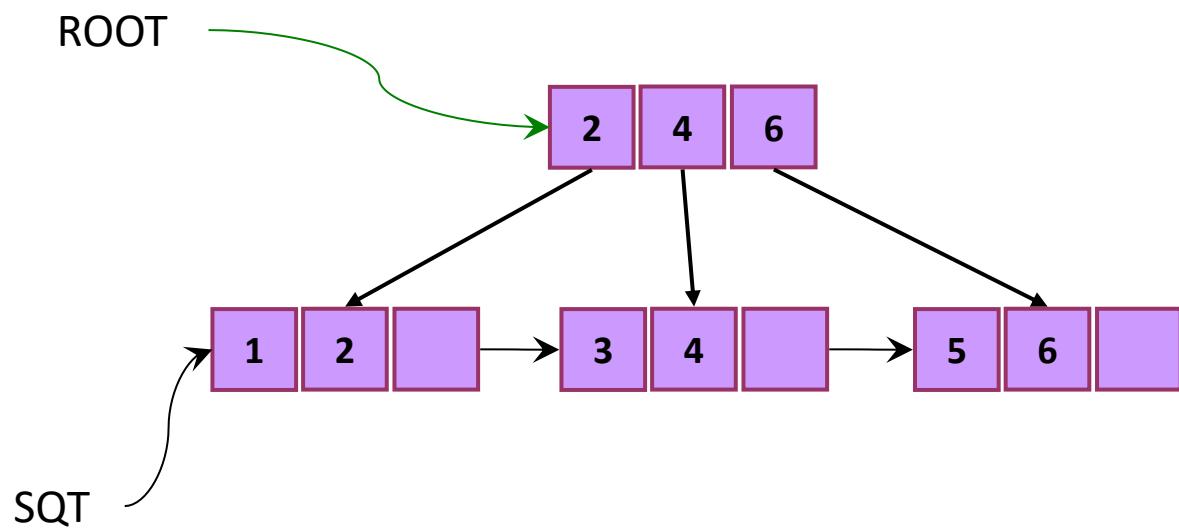
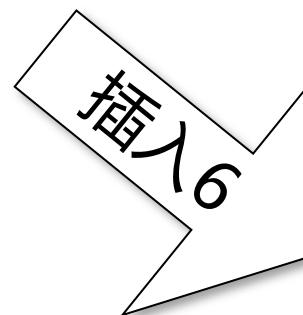
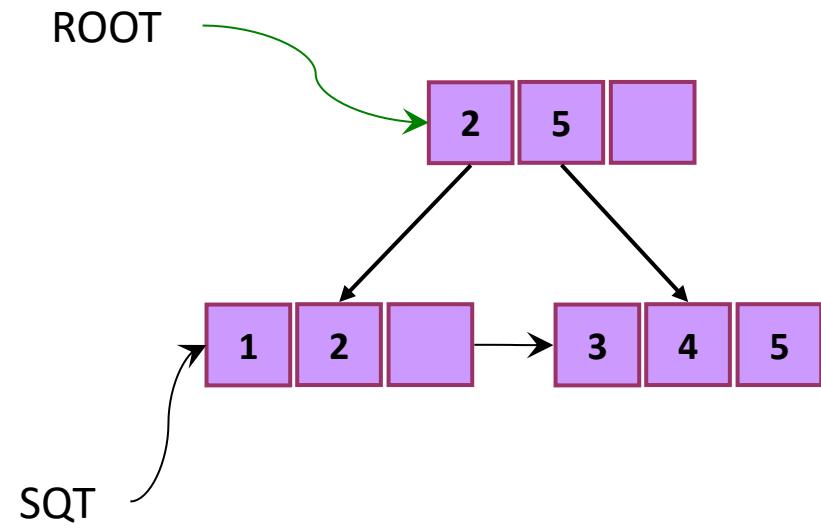
B+树插入



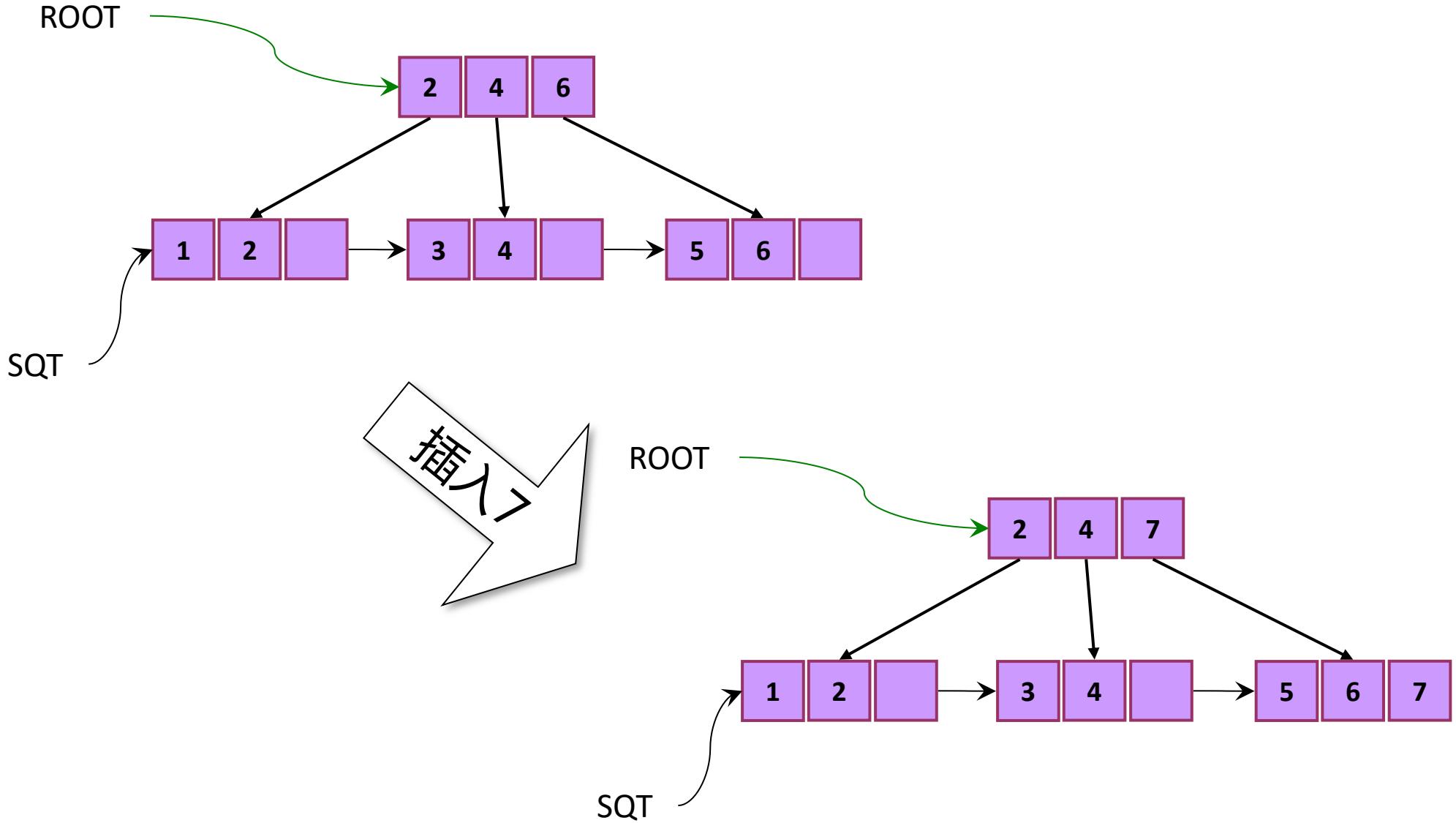
B+树插入



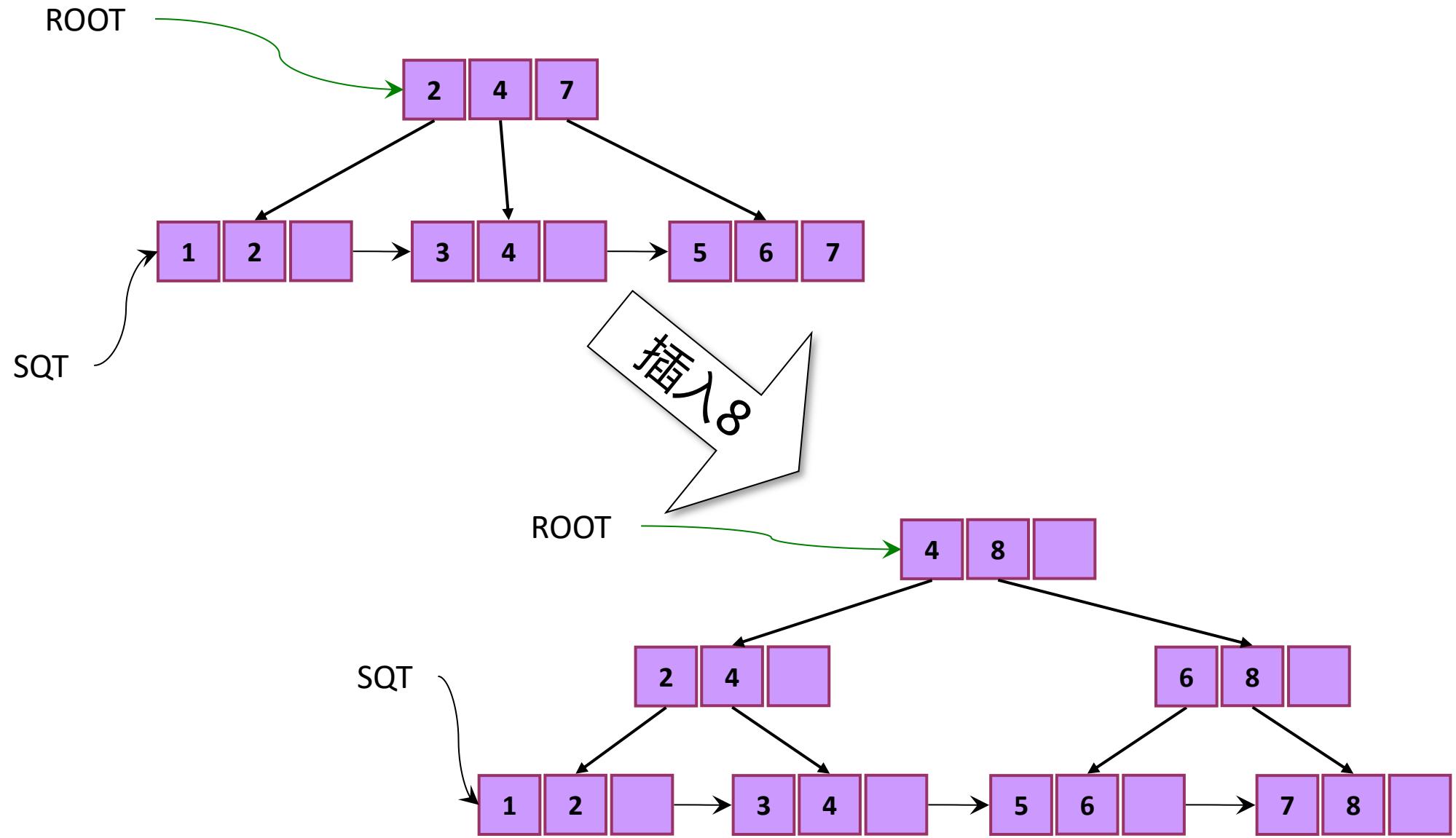
B+树插入



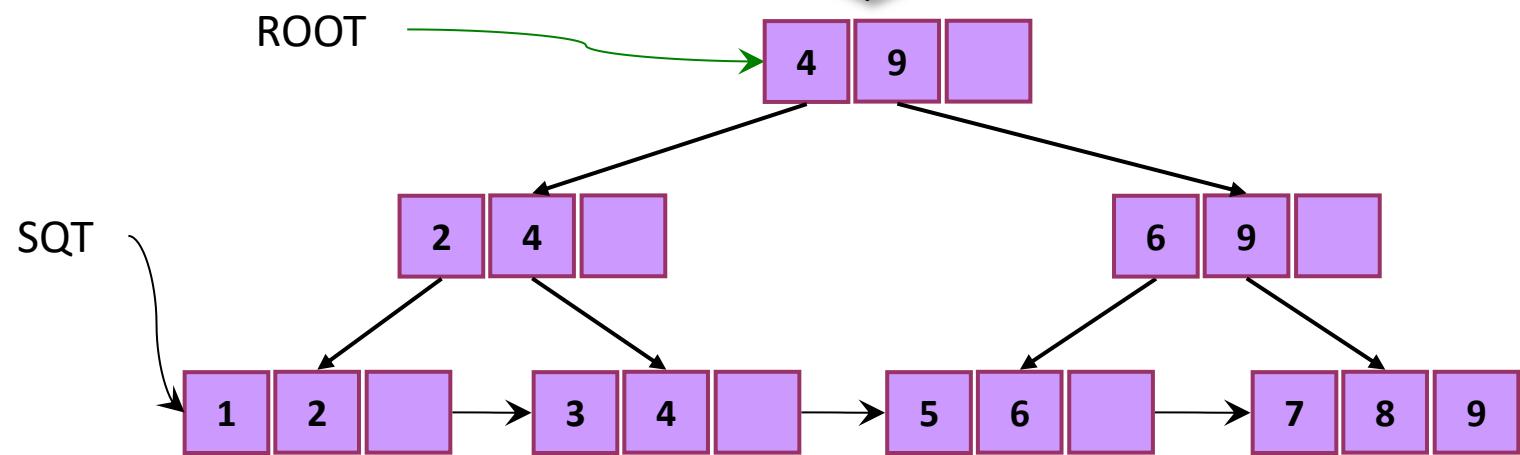
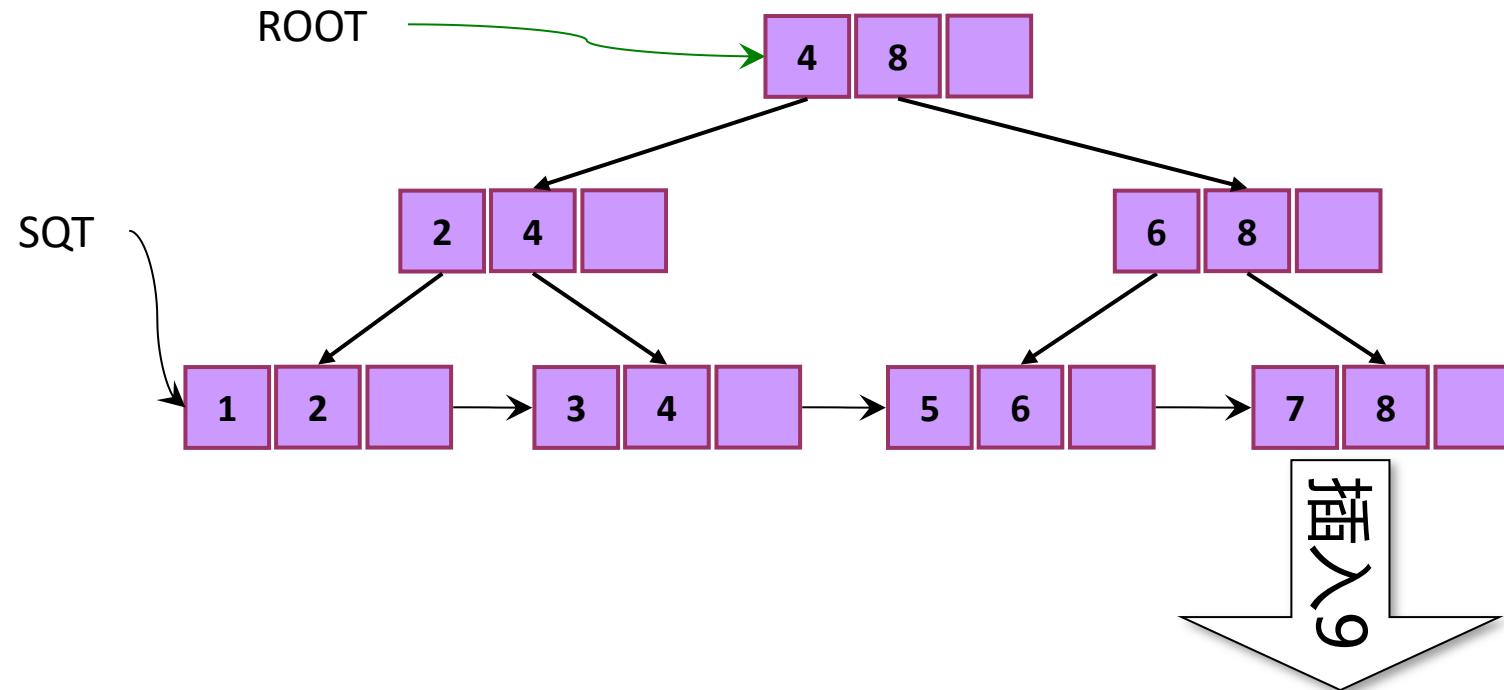
B+树插入



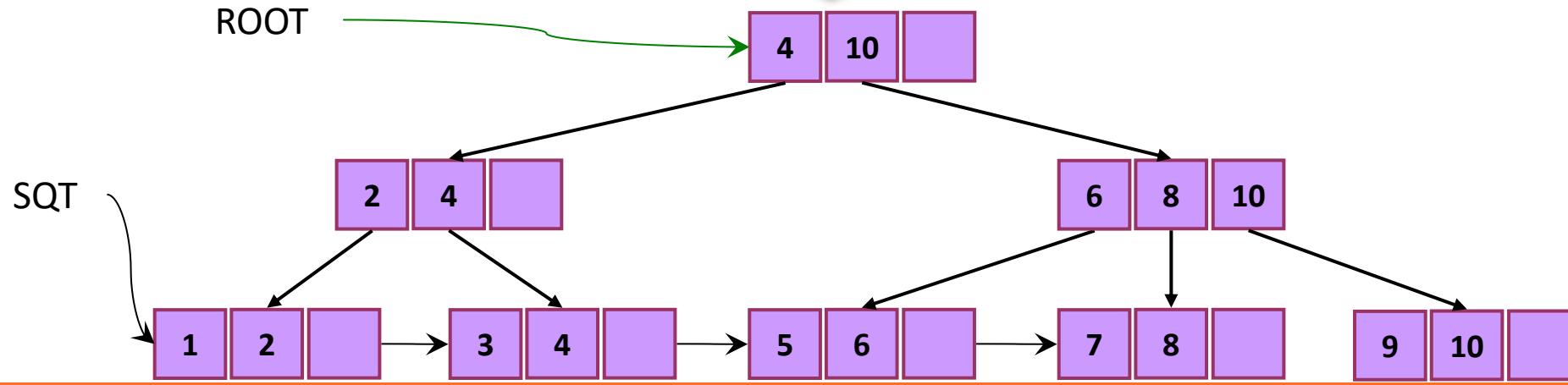
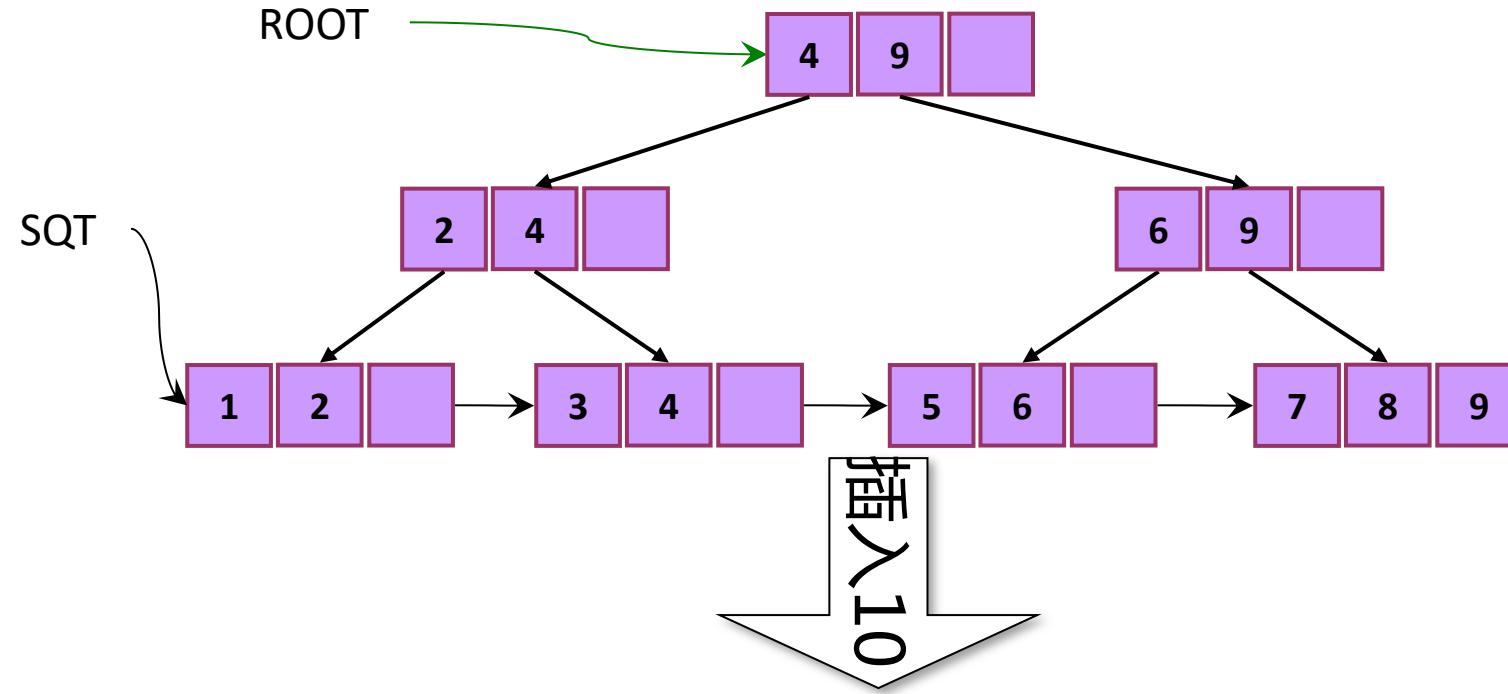
B+树插入



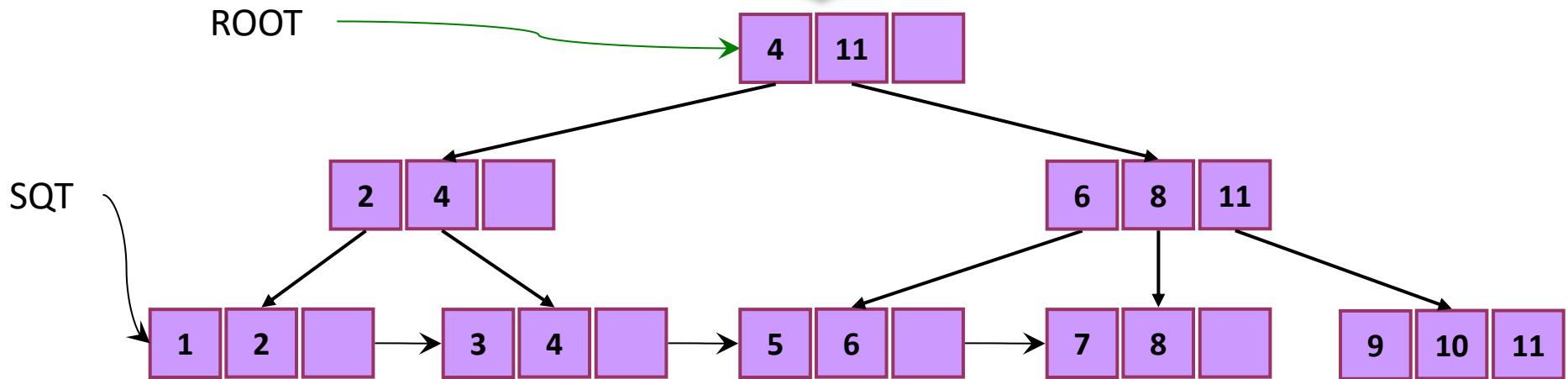
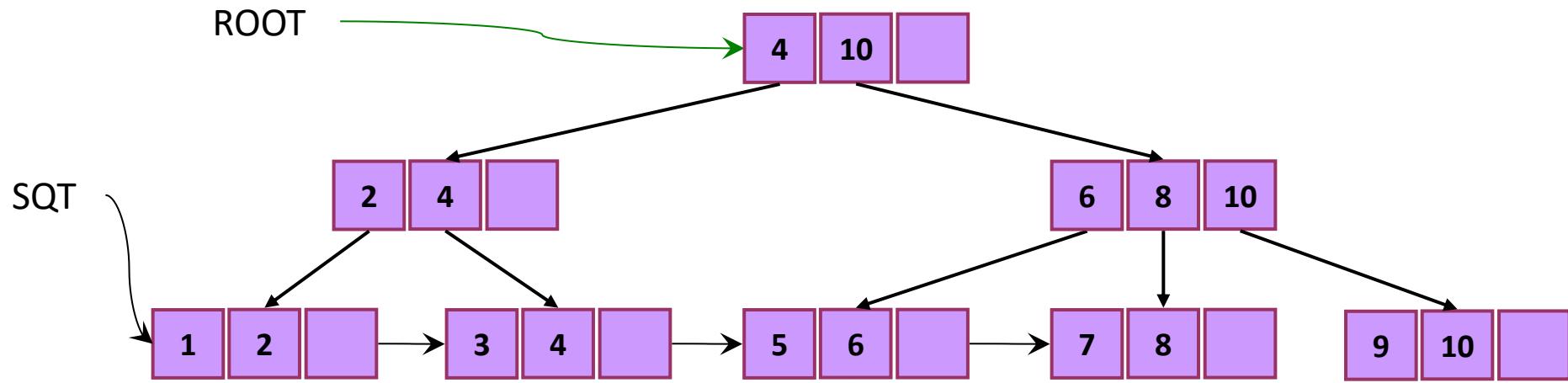
B+树插入



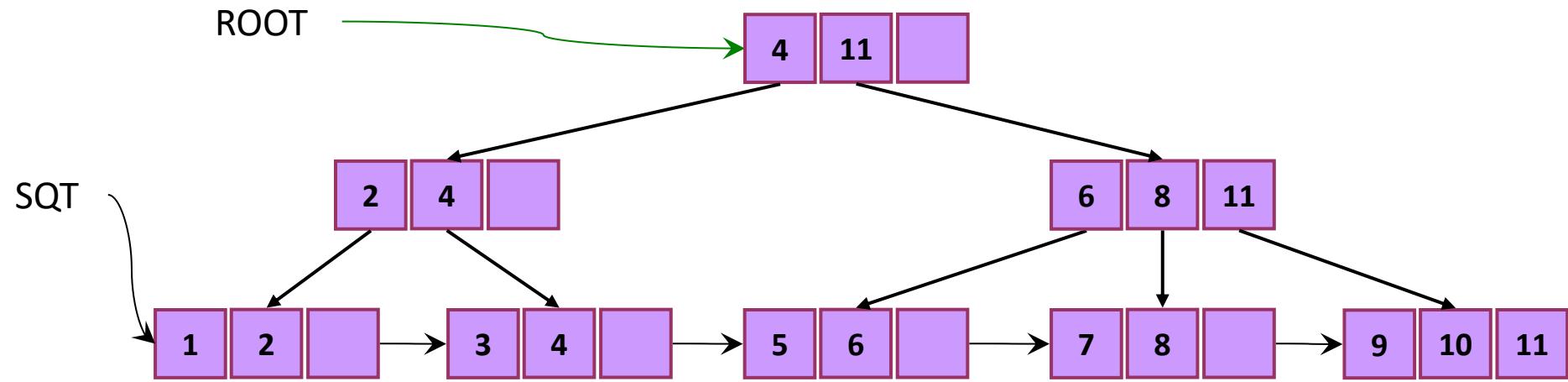
B+树插入



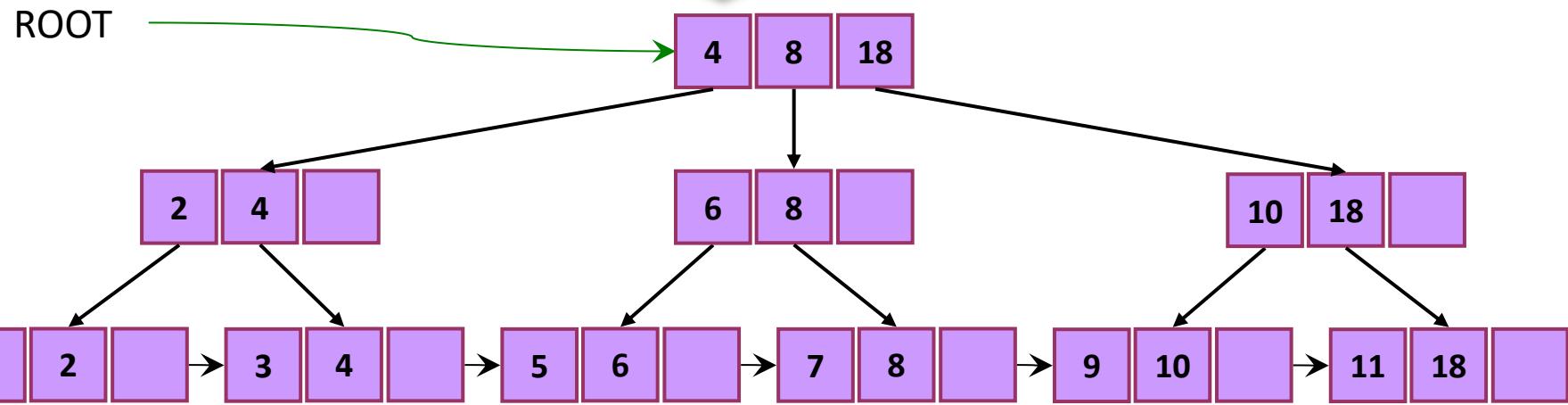
B+树插入



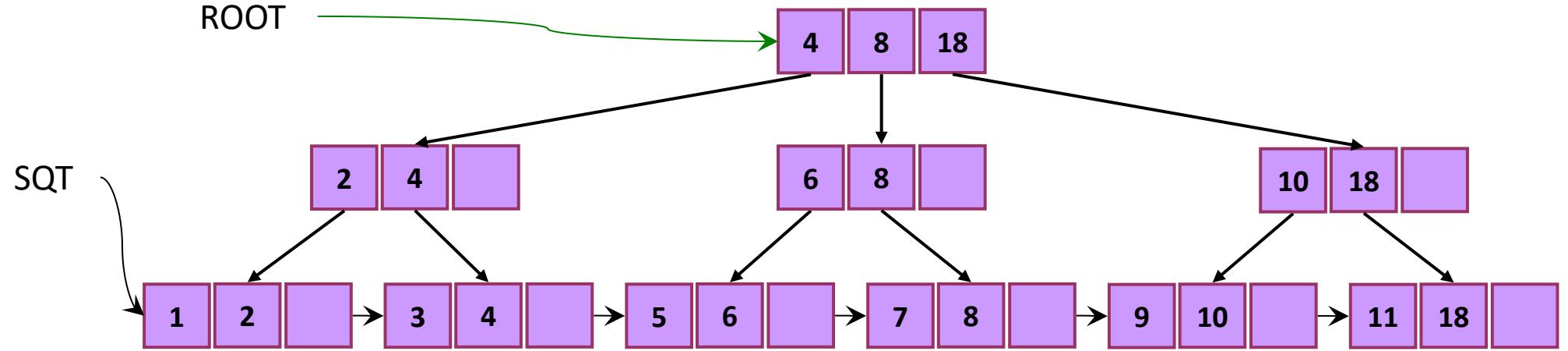
B+树插入



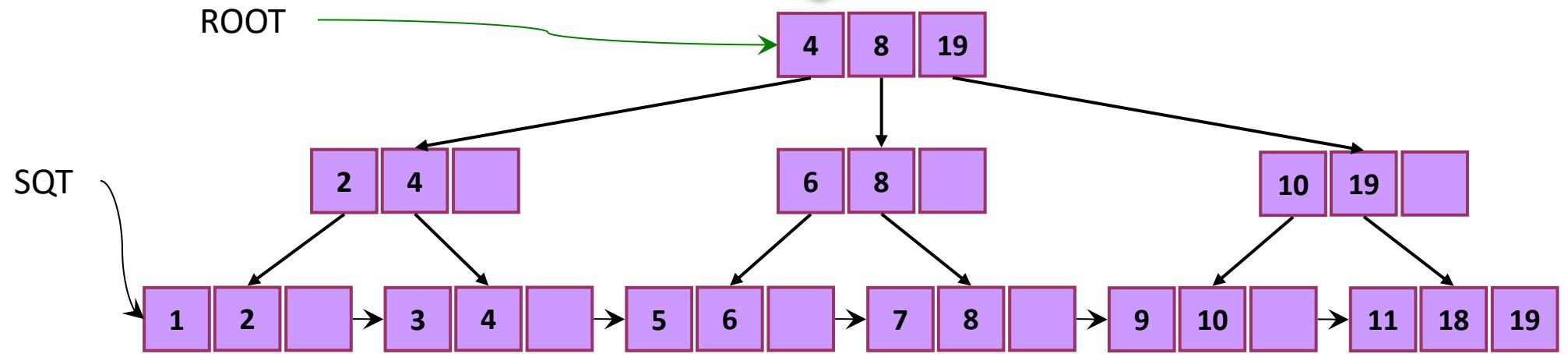
插入 18



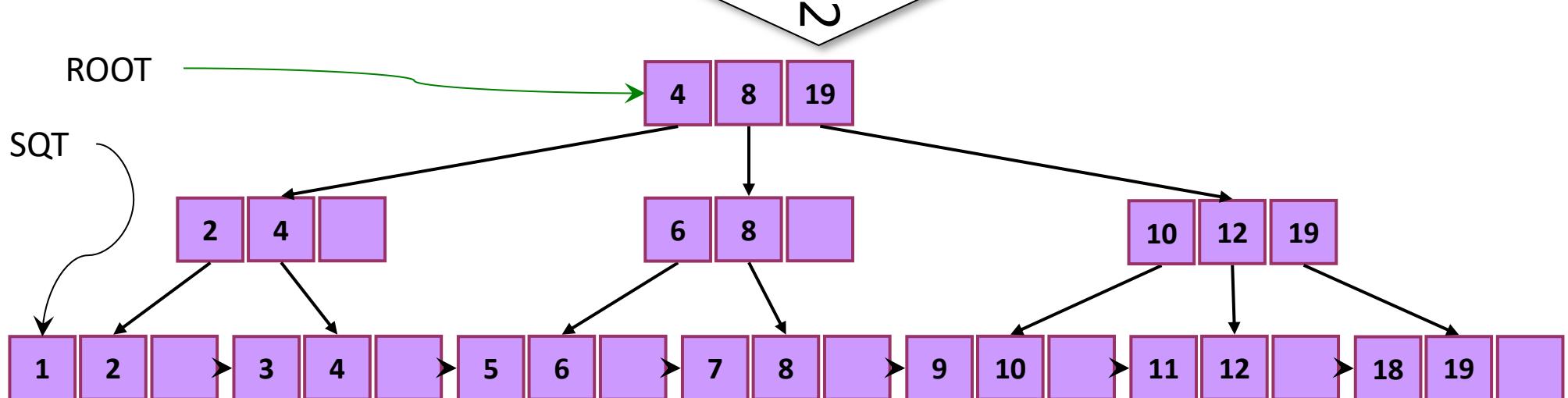
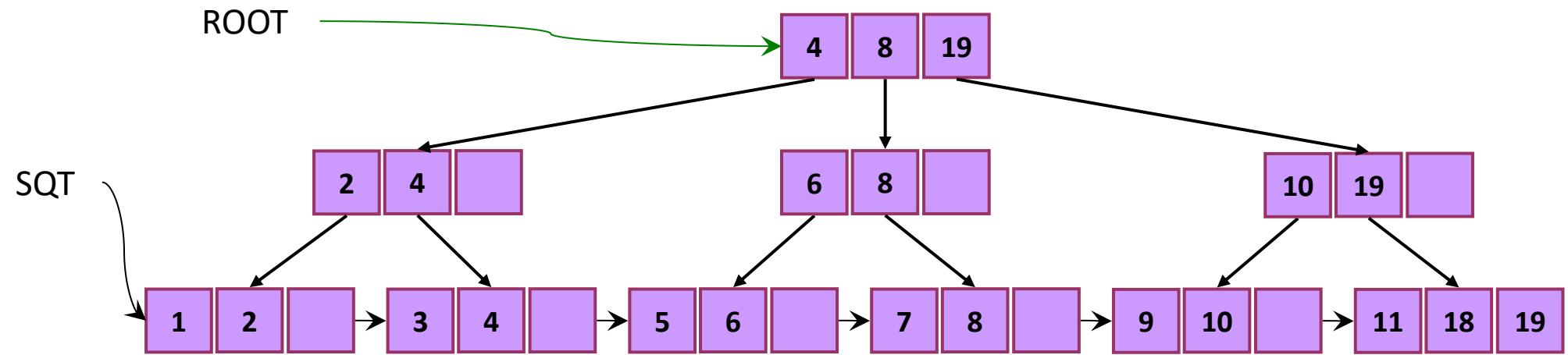
B+树插入



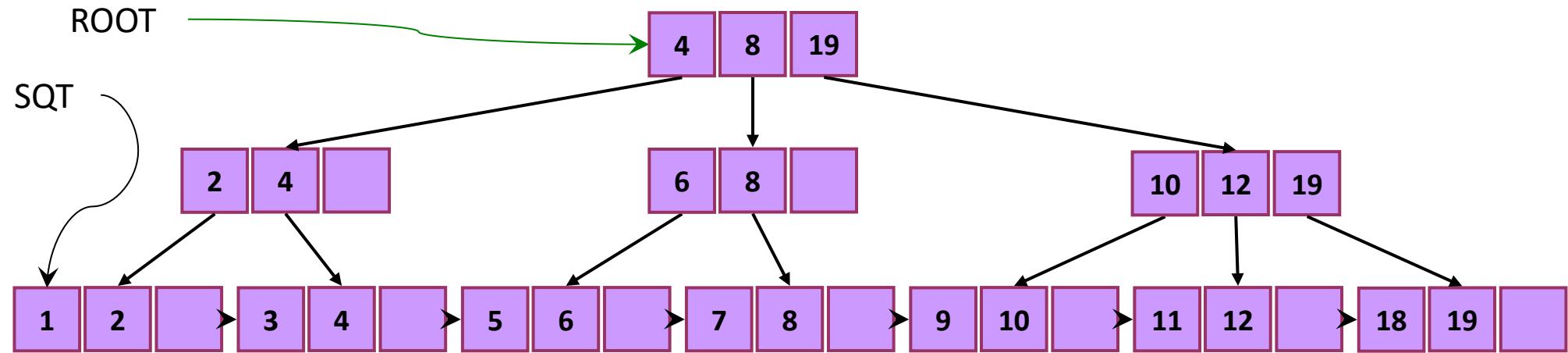
插入 19



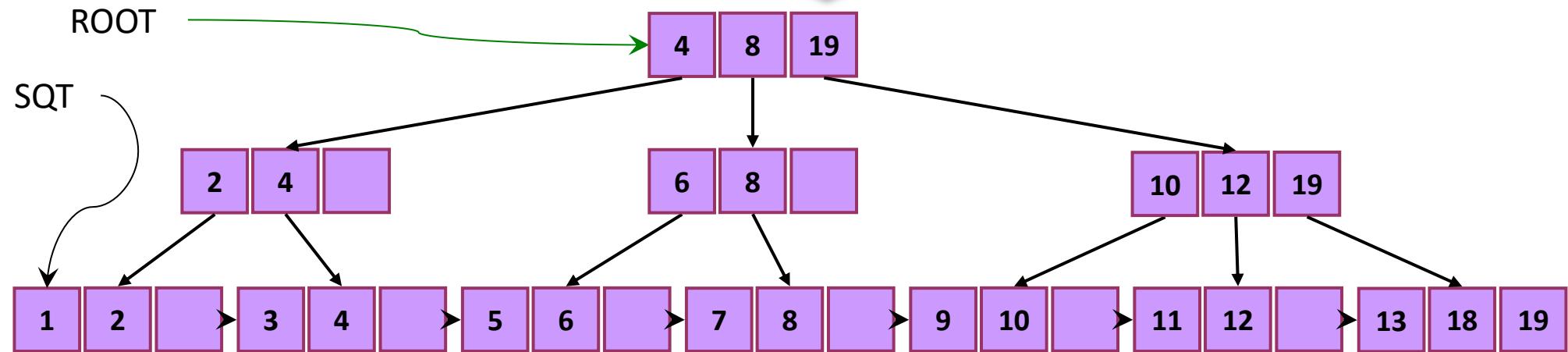
B+树插入



B+树插入

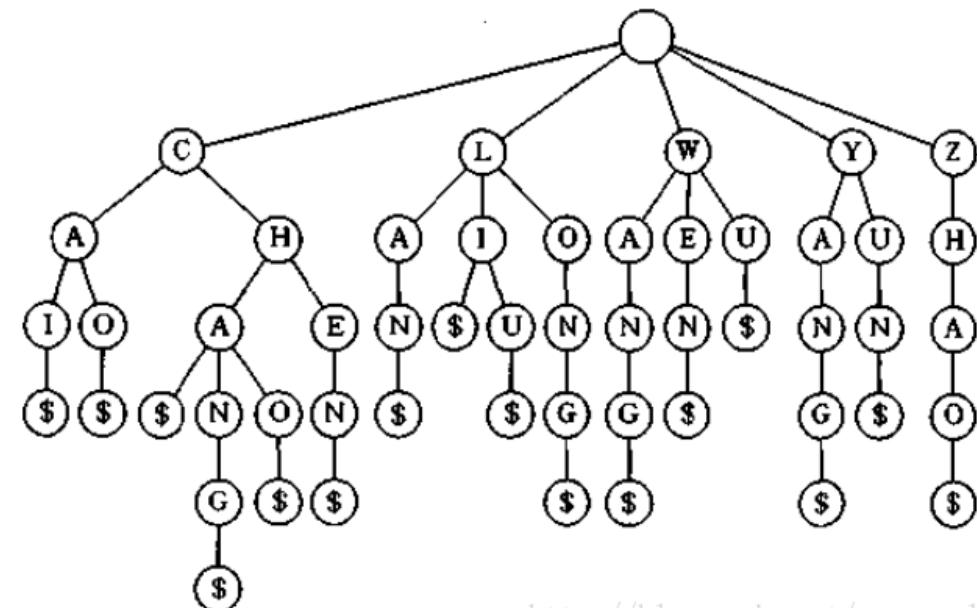


插入 13



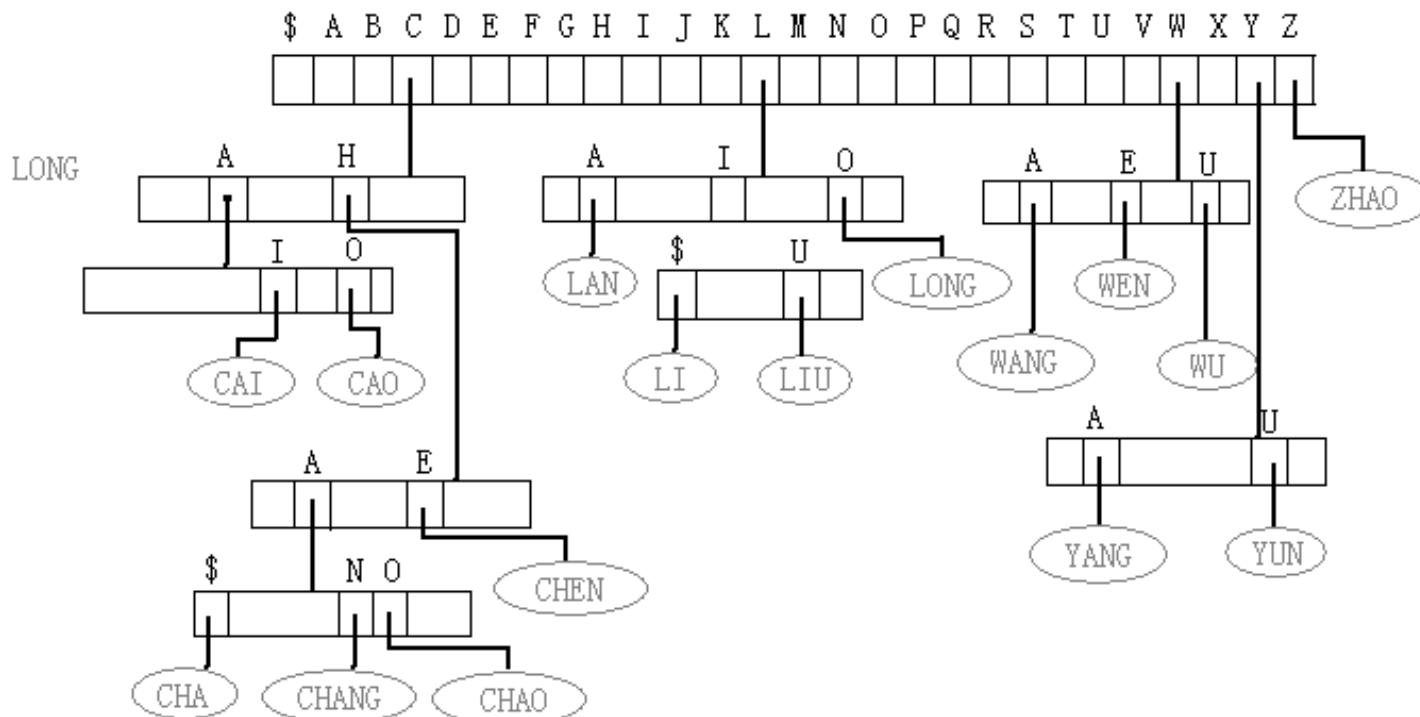
Trie树

- 定义
 - 又称字典树，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 ;
```

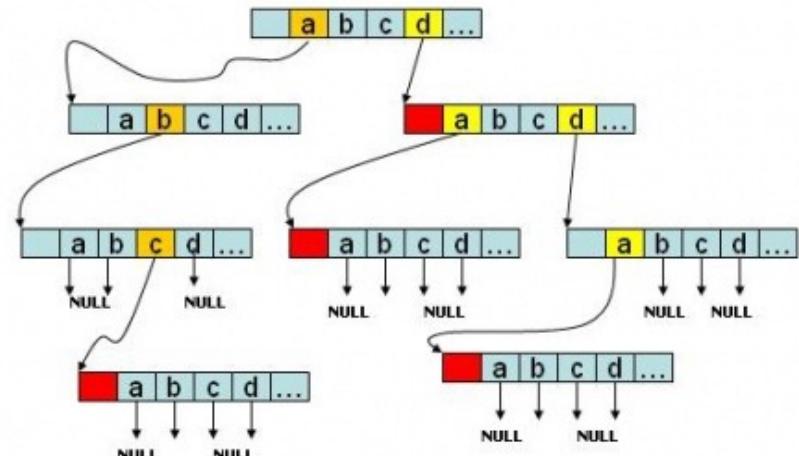
Trie树的插入

- 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？

- 256叉的Trie树

```
struct trienode_t
```

```
{
```

```
    char num;
```

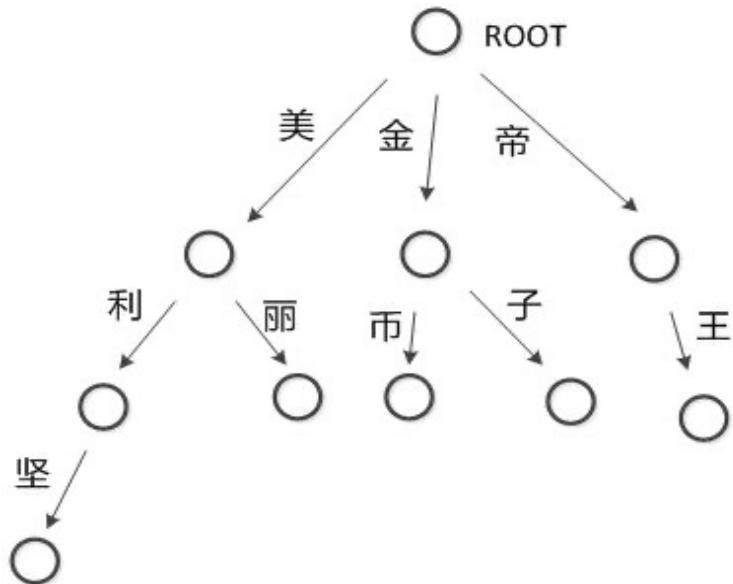
```
    struct trienode_t *child[256];
```

```
};
```

- 127万个字符串内存占用约4G

- 叉太多了

字符串“美利坚”、“美丽”、“金币”、“金子”、“帝王”所构成的字典树。



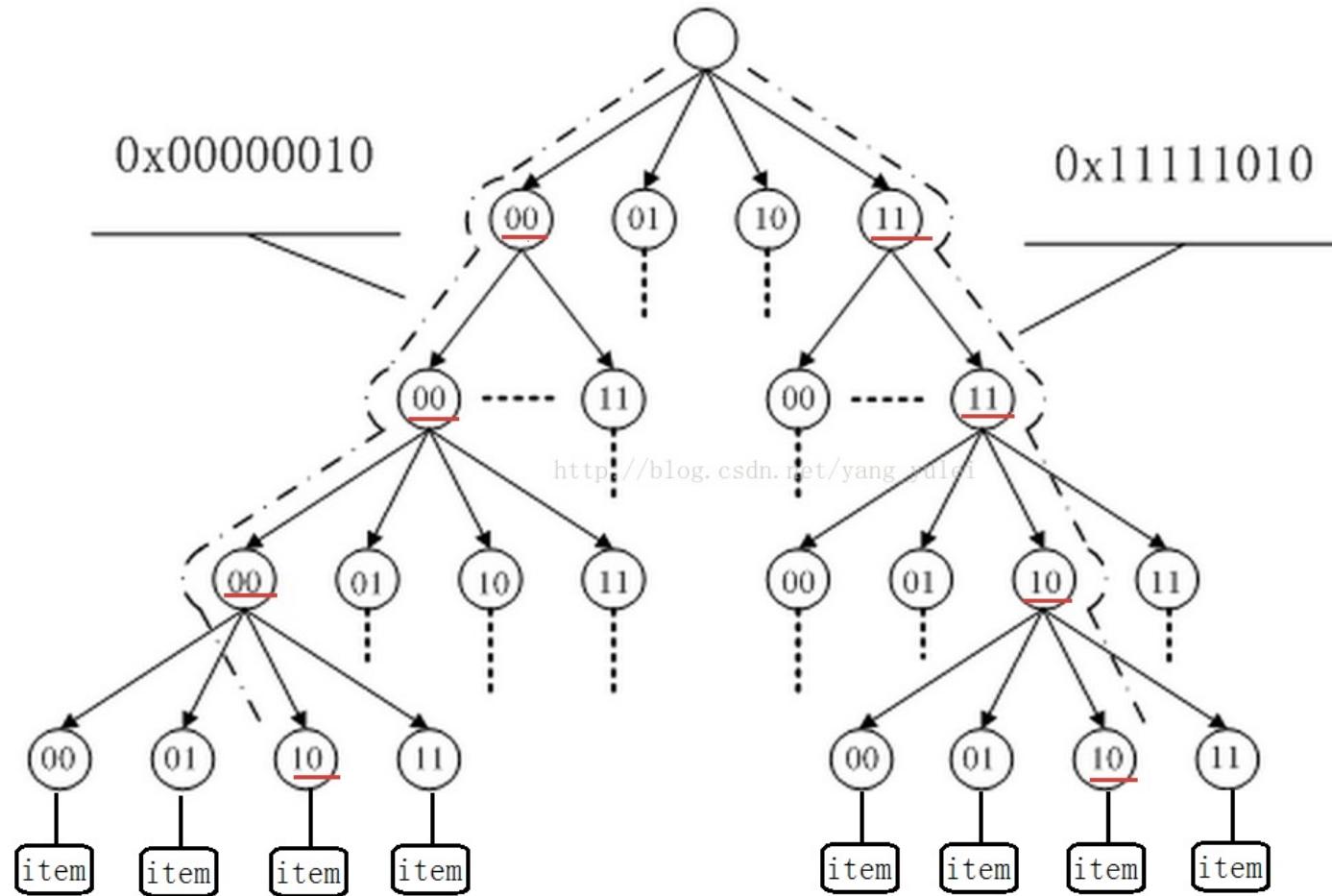
如何缩小分叉

- 回到二进制表示

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

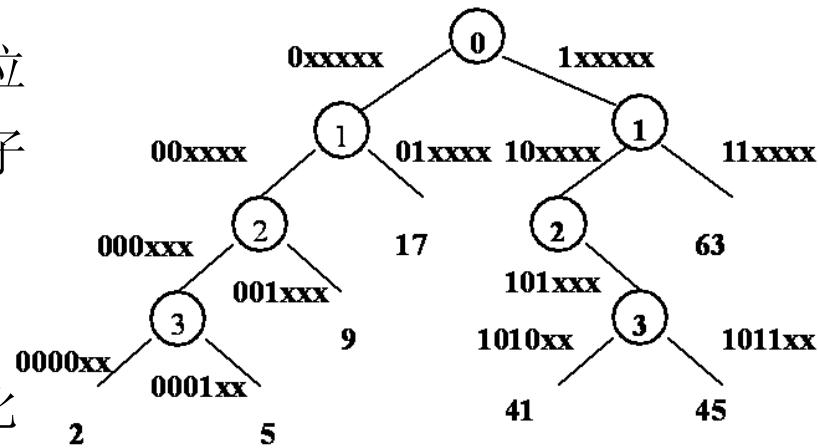
4叉Trie?



- Linux内核应用
- 管理内存分配
- 缓存区映射
- R=4

2叉Trie

- 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字5



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

多少叉最好？

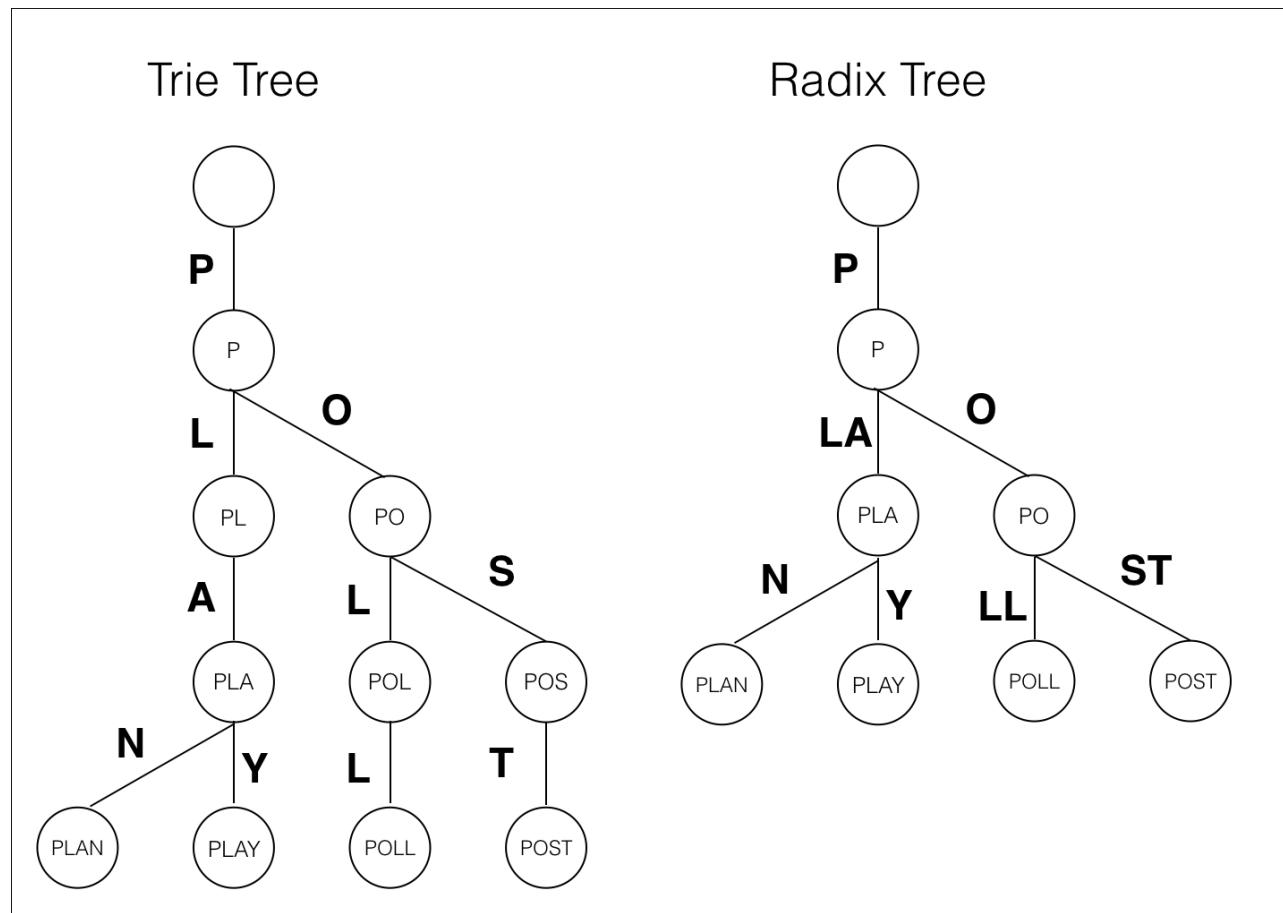
据计算，根据pattern.txt中数据要建立的树的总字节数为12257012字节，总行数为1282551，则平均每行9.56字节，由此计算树的大致规模如下

节点数据 容量	节点最大 child数	树平均深 度	实测节点 数量	实验二中占用 内存	实验二中运行 时间
8bit	256	9.56		5GB以上	10s以上
4bit	16	19.12	8300000	1171MB	1.25s
2bit	4	38.24	15900000	760.9MB	1.35s

根据实验二结论，若节点采用8bit容量则树的内存占用可达5GB左右，且树深较浅，不能充分发挥KMB算法优势，排除。其中当采用2bit容量时运行时间虽然略长，但占用内存小，且树深大，推测采用KPM算法时能更好发挥跳转的优势，因此节点数据容量优先使用2bit大小。

压缩路径Trie

- 定义



- 核心思想

- 压缩路径

压缩2叉Trie (2C-Trie)

- 首先插入第一个二进制串

字符串	二进制
abc	100001100100011011000110
aac	100001101000011011000110
Aac	100000101000011011000110

100001100100011011000110

压缩2叉Trie (2C-Trie)

- 插入第二个二进制串

100001101000011011000110

100001100100011011000110

压缩2叉Trie (2C-Trie)

- 得到他们的公共前缀

100001101000011011000110

100001100100011011000110

压缩2叉Trie (2C-Trie)

- 将公共前缀独立为新节点，并重置两个结点的内容

1000011011000110

0100011011000110

10000110

压缩2叉Trie (2C-Trie)

- 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

压缩2叉Trie (2C-Trie)

- 插入第三个节点

100000101000011011000110

10000110

01000110110001
10

10000110110001
10

压缩2叉Trie (2C-Trie)

- 得到公共前缀

100000101000011011000110

10000110

01000110110001
10

10000110110001
10

压缩2叉Trie (2C-Trie)

- 将公共前缀独立为单独的节点

0101000011011000110

110

10000

01000110110001

10

10000110110001

10

压缩2叉Trie (2C-Trie)

- 同样将其插入到新节点对应的位置即可

10000

0101000011011000110

110

0100011011
000110

100001101
1000110

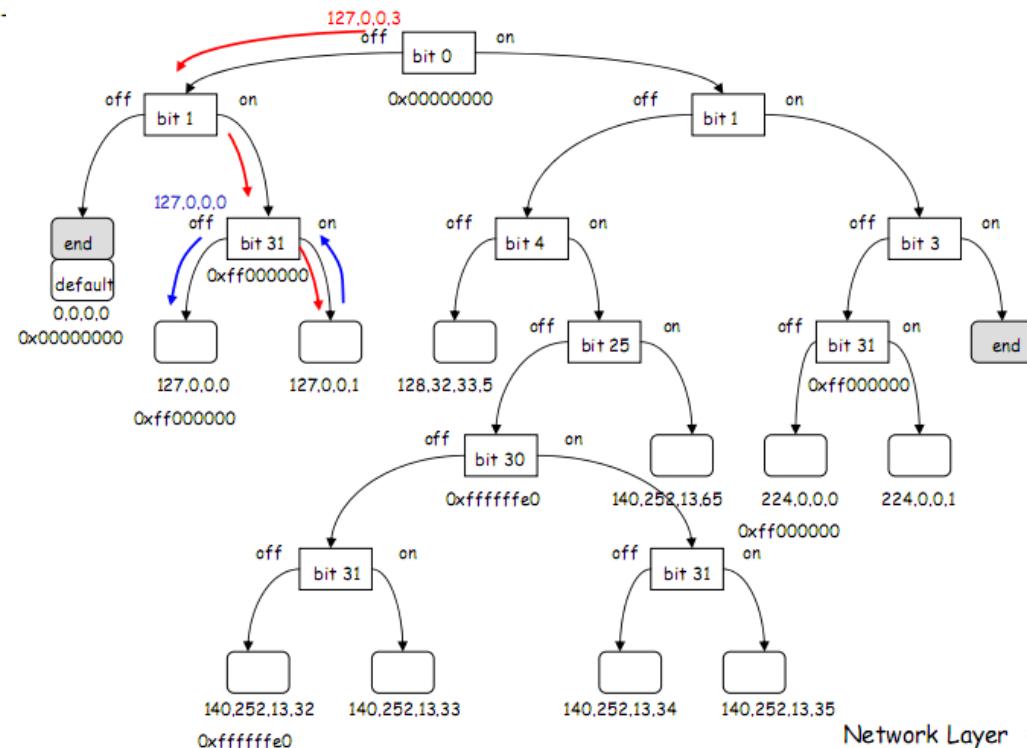
压缩2C-Trie应用：路由表查找

➤ Routing Table

Destination	Gateway	Flags	Ref	Use	Interface
<hr/>					
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



程序要求

- 分别实现四个版本，程序运行参数：
 - bplussearch m dict.txt string.txt
 - m 分别是2、4、6、8、10阶
 - mtrie m dict.txt string.txt
 - m分别是2、2C、4、16、256

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
 - 技术指标
 - 平均存储开销：树总结点数/所存储元素个数
 - 平均查找开销：成功查找到的元素所比较过的树节点个数
 - 内存占用：程序运行时占用的峰值内存
 - 结论和总结

THE END