Linux内核中常用的数据结构和算法

胡　隽

Tel：13867498758

Email：huj@zucc.edu.cn

- 链表
  - 双向链表
  - 哈希链表
- 树
  - 树—二叉树--二叉搜索树--红黑树

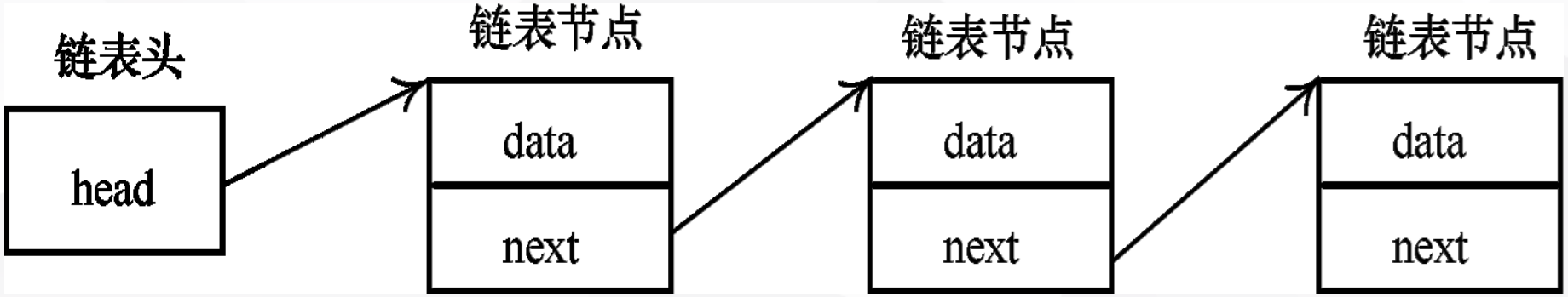
链表

- Linux内核代码大量使用了链表这种数据结构。
- 链表是在解决数组不能动态扩展这个缺陷而产生的一种数据结构。链表所包含的元素可以动态创建并插入和删除。链表的每个元素都是离散存放的，因此不需要占用连续的内存。
- 链表通常由若干结点组成，每个结点的结构都是一样的，由有效数据区和指针区两部分组成。有效数据区用来存储有效数据信息，而指针区用来指向链表的前继结点或者后继结点。
- 因此，链表就是利用指针将各个结点（节点）串联起来的一种存储结构。

单向链表

- 单向链表的指针区只包含一个指向下一个结点的指针，因此会形成一个单一方向的链表

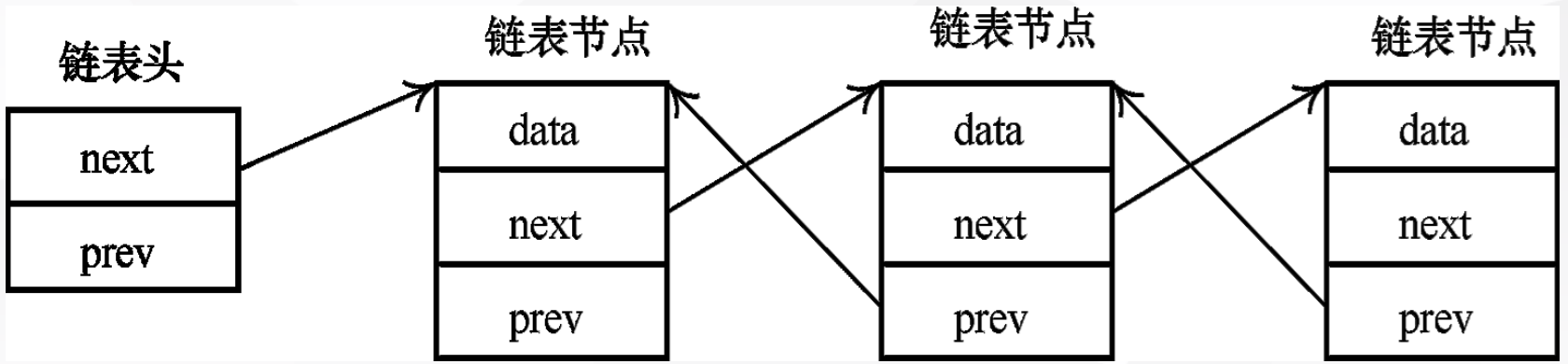
```
struct list {
int data;                /*有效数据*/
struct list *next;       /*指向下一个元素的指针*/
};
```



双向链表

- 指针区包含了两个指针，一个指向前继结点，另一个指向后继结点

```
struct list {
int data;                /*有效数据*/
struct list *next;       /*指向下一个元素的指针*/
struct list *prev;       /*指向上一个元素的指针*/
};
```



Linux内核链表的实现1

- Linux内核实现了一套纯链表的封装，链表结点数据结构只有指针区而没有数据区，另外还封装了各种操作函数，如创建结点函数、插入结点函数、删除结点函数、遍历结点函数等。
  - 链表类型定义在include/linux/types.h文件中

```
186 struct list_head {
187         struct list_head *next, *prev;
188 };
```

  - 链表操作定义在include/linux/list.h文件中
  - 链表在Linux中的应用有17464处
  - https://code.woboq.org/data/symbol.html?root=../linux/&ref=list_head

Linux内核链表的实现2

- 内核中大量的利用双链表结构，配合哈希表或者树结构，来迅速的定位内核中数以万计的各种数据结构
- 在Linux内核链表中，不是在链表结构中包含数据，而是在数据结构中包含链表结点。
- struct list_head数据结构不包含链表结点的数据区，通常是嵌入其他数据结构，在遍历链表的时候，根据双链表结点的指针获取"它所在结构体的指针"，从而再获取数据。
- 如struct page数据结构中嵌入了一个lru链表结点，通常是把page数据结构挂入LRU链表。

```
struct page {
……
struct list_head lru;
……
}
```

Linux内核链表的操作1

- INIT_LIST_HEAD
- list_add
- list_add_tail
- list_del
- list_replace
- list_replace_init
- list_del_init
- list_move
- list_move_tail
- list_bulk_move_tail
- list_is_first
- list_is_last
- list_empty
- list_empty_careful
- list_rotate_left
- list_is_singular
- list_cut_position
- list_cut_before
- list_splice
- list_splice_tail
- list_splice_init
- list_splice_tail_init

Linux内核链表的操作2

- 在linux中，以"__"开头的函数意味着是内核的内部接口，外部不应该调用该接口。

```
50  /*
51   * Insert a new entry between two known consecutive entries.
52   *
53   * This is only for internal list manipulation where we know
54   * the prev/next entries already!
55   */
56  static inline void __list_add(struct list_head *new,
57                                struct list_head *prev,
58                                struct list_head *next)
59  {
60          if (!__list_add_valid(new, prev, next))
61                  return;
62
63          next->prev = new;
64          new->next = next;
65          new->prev = prev;
66          WRITE_ONCE(prev->next, new);
67  }
```

```
278 #define WRITE_ONCE(x, val) \
279 ({                                                      \
280         union { typeof(x) __val; char __c[1]; } __u =   \
281                 { .__val = (__force typeof(x)) (val) }; \
282         __write_once_size(&(x), __u.__c, sizeof(x));    \
283         __u.__val;                                      \
284 })
285
286 #endif /* __KERNEL__ */
```

Linux内核链表的操作3

```
69  /**
70   * list_add - add a new entry
71   * @new: new entry to be added
72   * @head: list head to add it after
73   *
74   * Insert a new entry after the specified head.
75   * This is good for implementing stacks.
76   */
77  static inline void list_add(struct list_head *new, struct list_head *head)
78  {
79          __list_add(new, head, head->next);
80  }
81
82
83  /**
84   * list_add_tail - add a new entry
85   * @new: new entry to be added
86   * @head: list head to add it before
87   *
88   * Insert a new entry before the specified head.
89   * This is useful for implementing queues.
90   */
91  static inline void list_add_tail(struct list_head *new, struct list_head *head)
92  {
93          __list_add(new, head->prev, head);
94  }
```

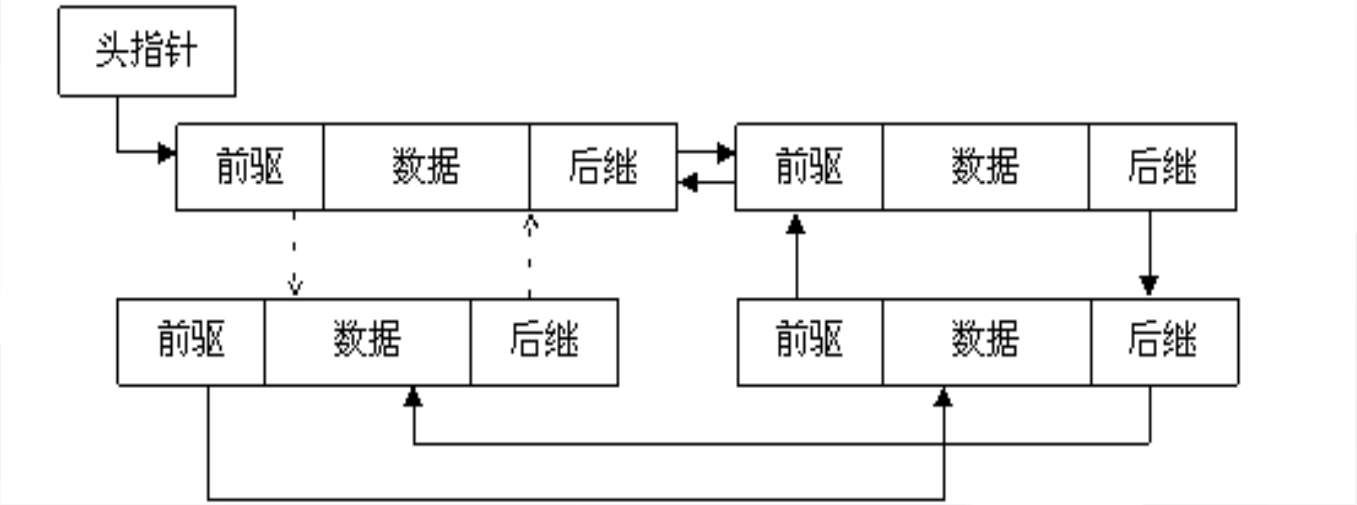
linux内核的哈希链表

- 双向循环链表vs哈希链表
  - 双向循环链表是循环的，哈希链表不是循环的
  - 双向循环链表不区分头结点和数据结点，都用list_head表示，而哈希链表区分头结点(hlist_head)和数据结点(hlist_node)
- 哈希链表有关的两个数据结构

```
186 struct list_head {
187         struct list_head *next, *prev;
188 };
189
190 struct hlist_head {
191         struct hlist_node *first;
192 };
193
194 struct hlist_node {
195         struct hlist_node *next, **pprev;
196 };
```

从链表到树

- 通过设计前驱和后继两个指针域，双链表可以从两个方向遍历，这是它区别于单链表的地方。如果打乱前驱、后继的依赖关系，就可以构成"二叉树"；如果再让首结点的前驱指向链表尾结点、尾结点的后继指向首结点，就构成了循环链表；如果设计更多的指针域，就可以构成各种复杂的树状数据结构。



树的基本概念

- 树（Tree）是n（n≥0）个结点的有限集合T，它满足两个条件 ：
  - 有且仅有一个特定的称为根（Root）的结点
  - 其余的结点可以分为m（m≥0）个互不相交的有限集合T1、T2、……、Tm，其中每一个集合又是一棵树，并称为其根的子树（Subtree）。

树的基本概念

- 一个结点的子树的个数称为该结点的**度数**，一棵树的度数是指该树中**结点**的最大度数。
- 度数为零的结点称为**树叶**或**终端结点**，度数不为零的结点称为**分支结点**，除根结点外的分支结点称为内部**结点**。
- 一个结点的子树之根结点称为该结点的**子结点**，该结点称为它们的**父结点**，同一结点的各个子结点之间称为**兄弟结点**。一棵树的根结点没有父**结**点，叶结点没有子**结**点。

树的基本概念

- 一个结点系列k1,k2, ……,ki,ki+1, ……,kj,并满足ki是ki+1的父结点，就称为一条从k1到kj的路径，路径的长度为j-1,即路径中的边数。路径中前面的结点是后面结点的祖先，后面结点是前面结点的子孙。
- 结点的层数等于父结点的层数加一，根结点的层数定义为一。树中结点层数的最大值称为该树的高度或深度。
- 若树中每个结点的各个子树的排列为从左到右，不能交换，即兄弟之间是有序的，则该树称为有序树。一般的树是有序树。
- m（m≥0）棵互不相交的树的集合称为森林。树去掉根结点就成为森林，森林加上一个新的根结点就成为树。

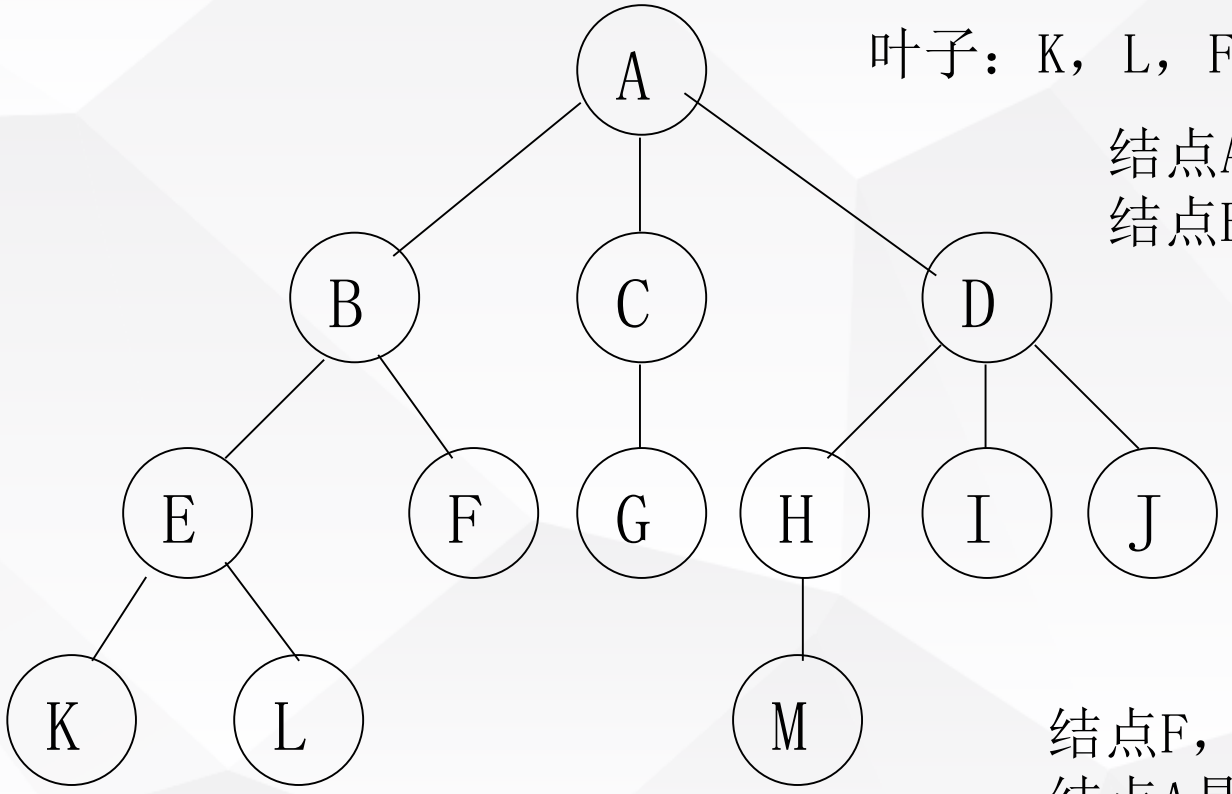
树的基本概念

结点A的度：3
结点B的度：2
结点M的度：0

树的度：3

结点A的层次：1
结点M的层次：4

树的深度：4



叶子：K，L，F，G，M，I，J

结点A的孩子：B，C，D
结点B的孩子：E，F

结点I的双亲：D
结点L的双亲：E

结点B，C，D为兄弟
结点K，L为兄弟

结点F，G为堂兄弟
结点A是结点F，G的祖先

树的逻辑结构 ：树中任何结点都可以有零个或多个直接后继结点（子结点），但至多只有一个直接前趋结点（父结点），根结点没有前趋结点，叶结点没有后继结点。

二叉树的概念

- 二叉树（Binary Tree）是n（n≥0）个结点的有限集合，它或者是空集（n=0），或者是由一个根结点以及两棵互不相交的、分别称为左子树和右子树的二叉树组成。二叉树与普通有序树不同，二叉树严格区分左孩子和右孩子，即使只有一个子结点也要区分左右。

二叉树的性质

- 二叉树第i（i≥1）层上的结点最多为2^{i-1}个。
- 深度为k（k≥1）的二叉树最多有2^k-1个结点。
- 在任意一棵二叉树中，树叶的数目比度数为2的结点的数目多一。

  总结点数为各类结点之和：n = n0 + n1 + n2

  总结点数为所有子结点数加一：n = n1 + 2*n2 + 1

  故得：n0 = n2 + 1 ;

完全二叉树和满二叉树

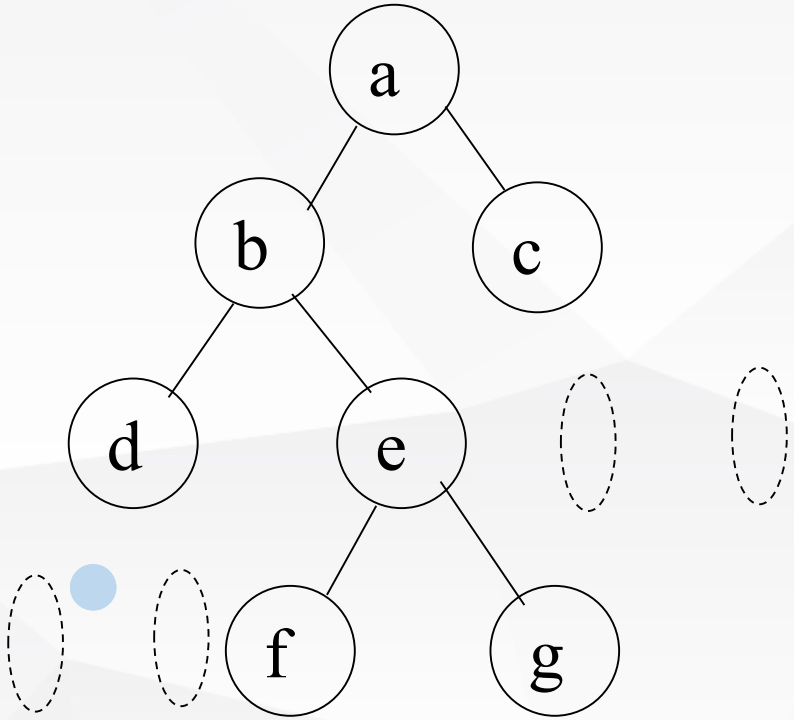
- **满二叉树：** 深度为k（k≥1）时有2^k-1个结点的二叉树。
- **完全二叉树：** 只有最下面两层有度数小于2的结点，且最下面一层的叶结点集中在最左边的若干位置上。

  具有n个结点的完全二叉树的深度为：

  （log2n）＋1或『log2(n+1)。

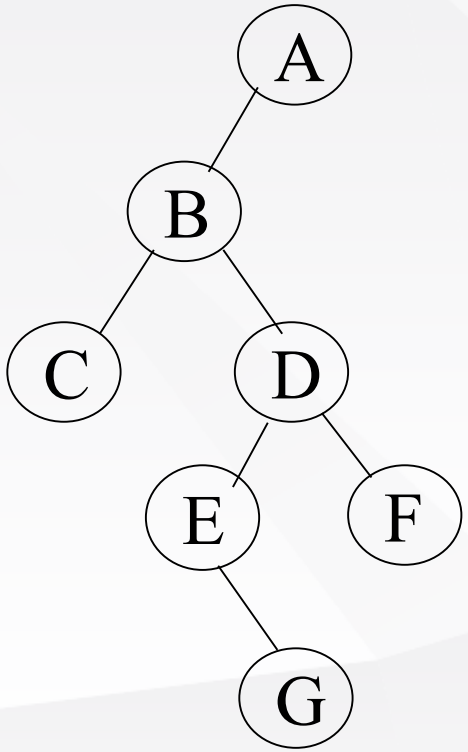
二叉树的顺序存储结构

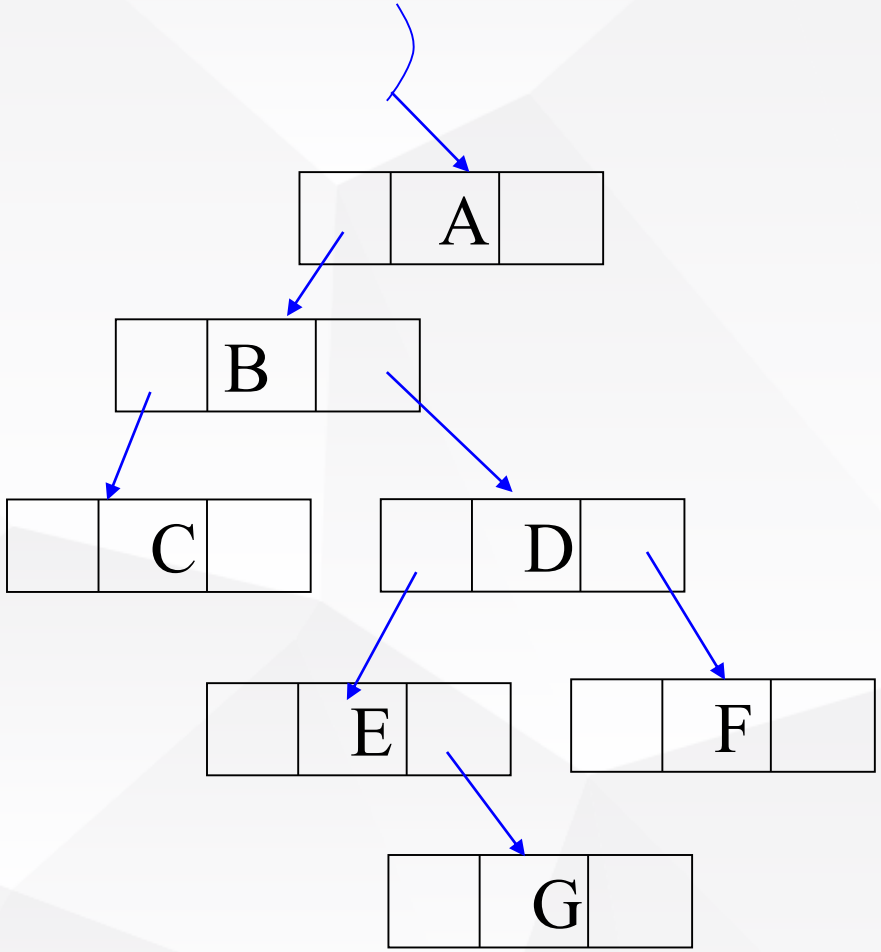
- 若二叉树不是完全二叉树，通过补虚结点，将其补成完全二叉树。
- 按从上到下，从左到右的顺序编号，根结点为1
- 按编号一次存储在连续空间中，虚结点用特殊符号标识即可。



1	2	3	4	5	6	7	8	9	10	11
a	b	c	d	e	0	0	0	0	f	g

二叉树的链式存储结构



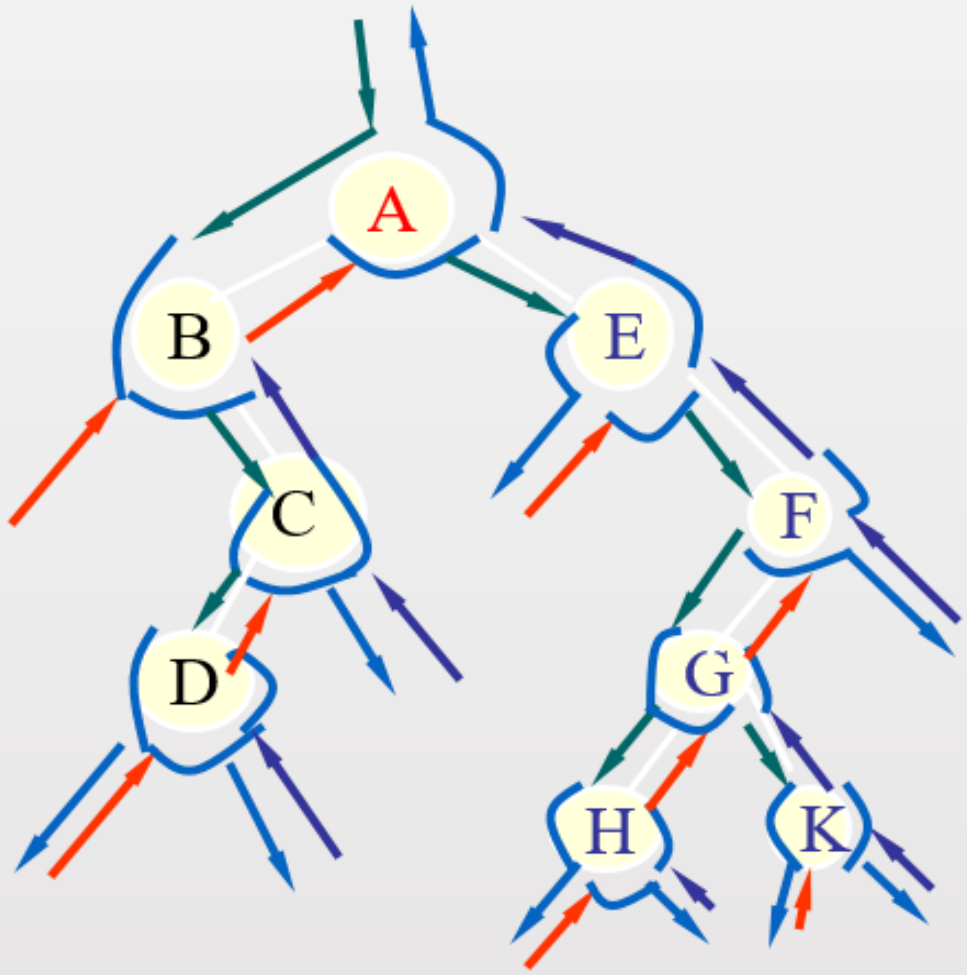


二叉树的遍历

- 遍历 ：沿某条搜索路径周游二叉树，对树中的每一个结点访问一次且仅访问一次。
- “遍历”是任何类型均有的操作，对线性结构而言，只有一条搜索路径(因为每个结点均只有一个后继)，故不需要另加讨论。而二叉树是非线性结构，每个结点有两个后继，则存在如何遍历即按什么样的搜索路径进行遍历的问题。

二叉树的遍历

- 由于二叉树的递归性质，遍历算法也是递归的。



先序序列：

A B C D E F G H K

中序序列：

B D C A E H G K F

后序序列：

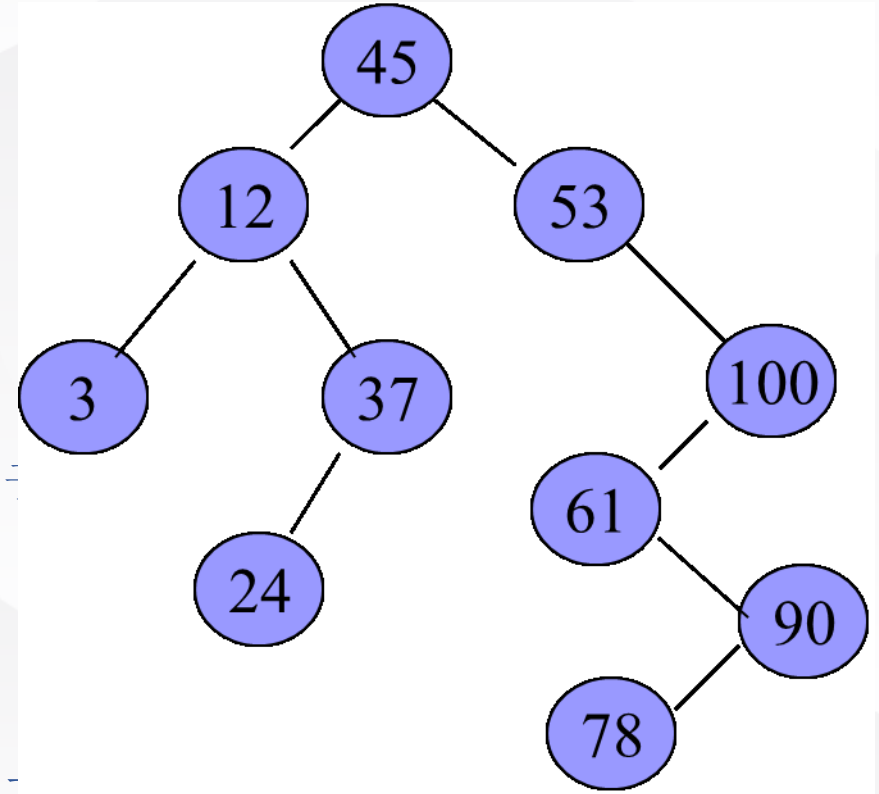
D C B H K G F E A

二叉树遍历算法实现

- 先序遍历算法:void pre_order(btee_pnode *t);
- 中序遍历算法: void mid_order(btee_pnode *t);
- 后序遍历算法: void post_order(btee_pnode *t);
- 层次遍历算法: void level_order(btee_pnode *t);

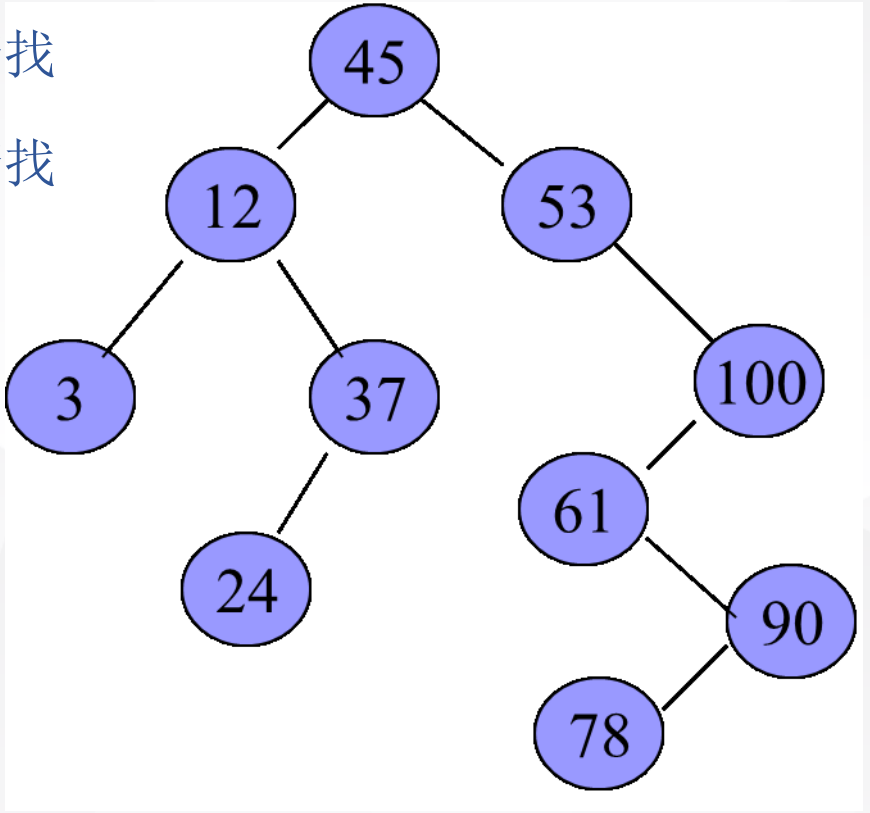
二叉搜索树Binary Search Tree（BST）

- 定义
  - 或者是一棵空树；
  - 是具有下列性质的二叉树：
    - 对于任何一个结点，设其值为K
    - 则该结点的 左子树(若不空)的任意一个结点的值都 小于
    - 该结点的 右子树(若不空)的任意一个结点的值都 大于
    - • 而且它的左右子树也分别为BST
- 性质：中序遍历是正序的（由小到大的排列）



二叉搜索树Binary Search Tree（BST）

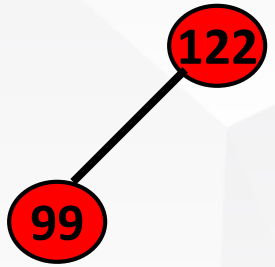
- 若二叉排序树为空，则查找不成功；否则，
  - 若给定值等于根结点的关键字，则查找成功
  - 若给定值小于根结点的关键字，则继续在左子树上进行查找
  - 若给定值大于根结点的关键字，则继续在右子树上进行查找
- 平衡二叉树： AVL、红黑树

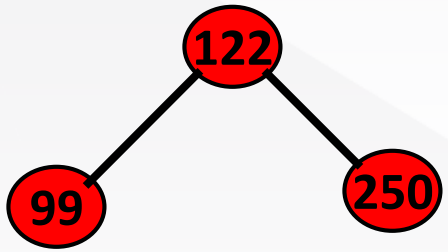


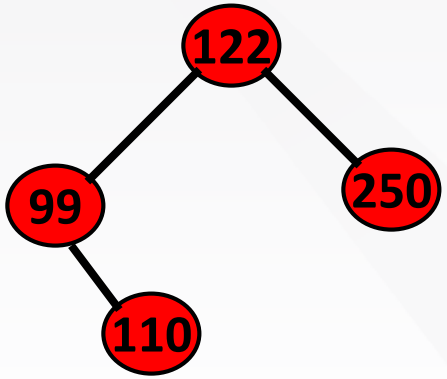
二叉搜索树 – 插入

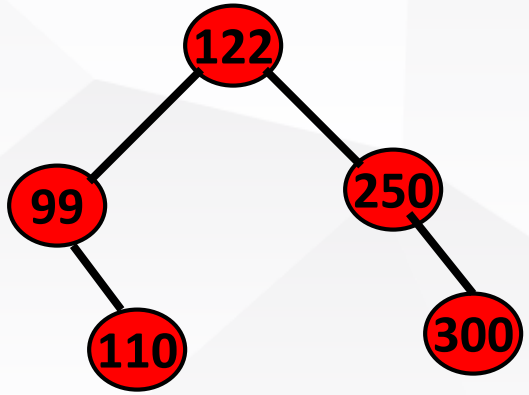
- 首先执行查找算法，找出被插结点的父亲结点。
  - 判断被插结点是其父亲结点的左、右儿子
  - 将被插结点作为叶子结点插入
  - 若二叉树为空。则首先单独生成根结点

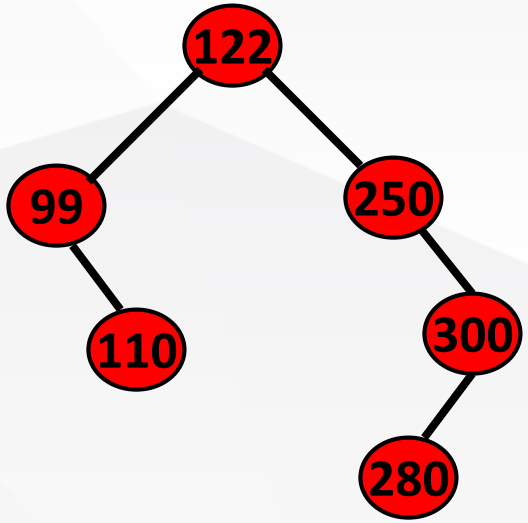
将数的序列：122、99、250、110、300、280 作为二叉排序树的结点的关键字值，生成二叉排序树。











若将数的序列改成：99， 110， 122、 250 、280、300

二叉搜索树 – 删除

- 和插入相反，删除在查找成功之后进行，并且要求在删除二叉排序树上某个结点之后，仍然保持二叉排序树的特性。可分为3种情况：
  - 被删除的结点是叶子；
  - 被删除的结点只有左子树或者只有右子树；
  - 被删除的结点既有左子树，也有右子树；

平衡二叉树Balanced Binary Tree

- 平衡二叉树又称AVL树。
- 它或者是一棵空树，或者是具有下列性质的二叉树：
  - 它的左子树和右子树都是平衡二叉树，且左子树和右子树的深度之差的绝对值不超过1。
  - 若将二叉树上结点的平衡因子BF（BalanceFactor）定义为该结点的左子树的深度减去它的右子树的深度，则平衡二叉树上所有结点的平衡因子只可能是-1、0和1。

红黑树

- 红黑树
  - ✓ 红黑树（Red Black Tree）被广泛应用在内核的内存管理和进程调度中，用于将排序的元素组织到树中。
  - ✓ 红黑树被广泛应用在计算机科学的各个领域中，它在速度和实现复杂度之间提供一个很好的平衡。

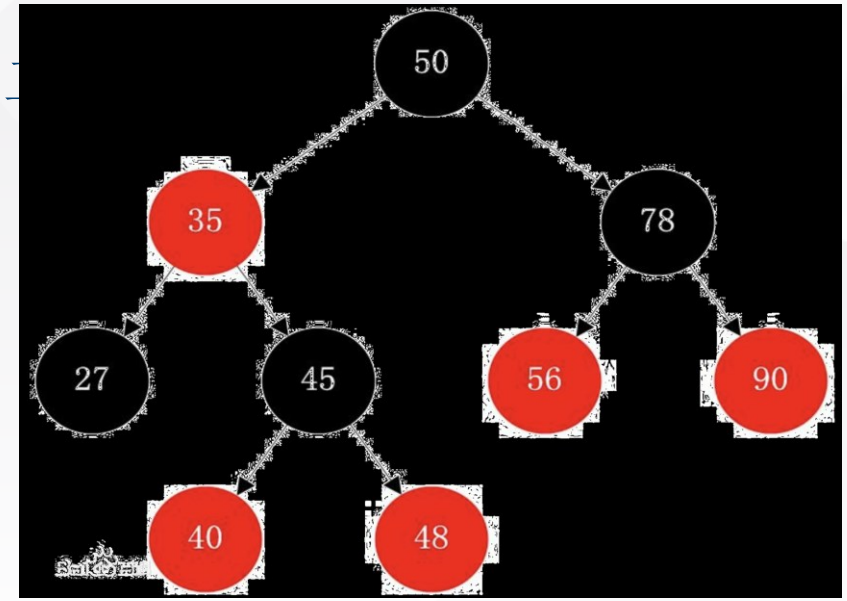
- 红黑树在内核的例子
  - ✓ documentation/Rbtree.txt
  - ✓ VMA的管理

红黑树定义

- rbtree with the following properties.
  1. Every node has a value.
  2. The value of any node is greater than the value of its left child and less than the value of its right child.
  3. Every node is colored either red or black.
  4. Every red node that is not a leaf has only black children.
  5. Every path from the root to a leaf contains the same number of black nodes.
  6. The root node is black.

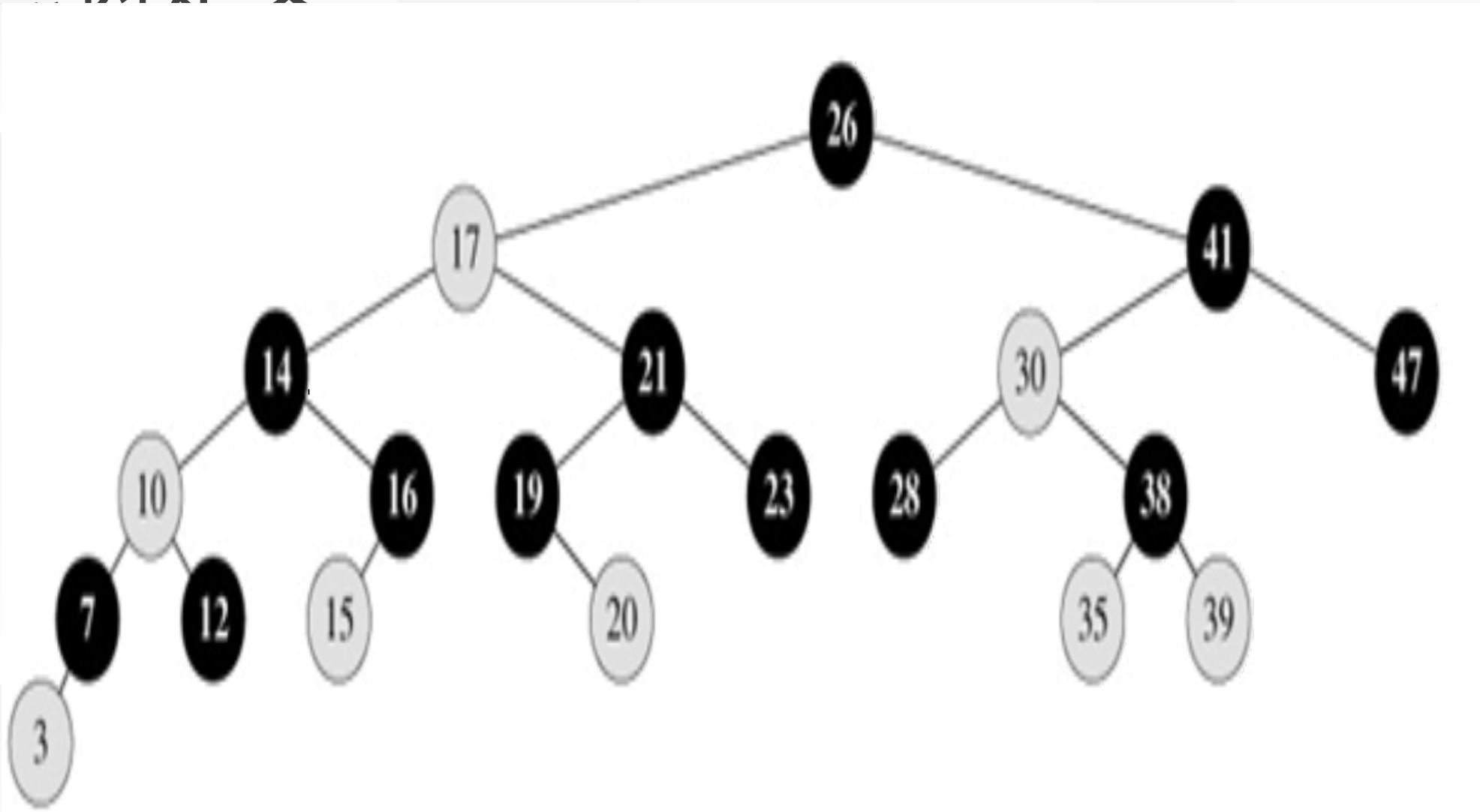
红黑树定义

- 红黑树是一种在插入或删除结点时都需要维持平衡的二
- 红黑树可以始终保持平衡。
- 每个结点都具有颜色属性：



1. 一个结点要么是红色的，要么是黑色的。
2. 根结点是黑色的。
3. 如果一个结点是红色的，那么它的子结点必须是黑色的，也就是说在沿着从根结点出发的任何路径上都不会出现两个连续的红色结点。
4. 从根结点到叶结点的每条路径上必须包含相同数目的黑色结点。

红黑树定义



Linux内核中红黑树定义1

- /include/linux/rbtree.h
- 红黑树节点必须保存四个值：
  1. 自身的颜色(color，红黑树的必须)
  2. 父节点指针（使得红黑树向上走查找父节点的时间复杂度是0(1)）
  3. 左孩子节点指针（使得红黑树向左走查找左孩子的时间复杂度是0(1)）
  4. 右孩子节点指针（使得红黑树向右走查找右孩子的时间复杂度是0(1)）

Linux内核中红黑树定义2

- __attribute__((aligned(sizeof(long))))属性保证了红黑树中的每个结点的首地址都是32位对齐的（在32位机上），也就是说每个结点首地址的bit[1]和bit[0]都是0，因此就可以使用bit[0]来存储结点的颜色属性而不干扰到其双亲结点首地址的存储。

```
36 struct rb_node {
37         unsigned long  __rb_parent_color;
38         struct rb_node *rb_right;
39         struct rb_node *rb_left;
40 } __attribute__((aligned(sizeof(long))));
41     /* The alignment might seem pointless, but allegedly CRIS needs it */
42
43 struct rb_root {
44         struct rb_node *rb_node;
45 };
```

Linux内核中红黑树的操作1

- /include/linux/rbtree.c
- /include/linux/rbtree_augmented.h

#define rb_parent(r)   ((struct rb_node *)((r)->rb_parent_color & ~3))

//获得其双亲结点的首地址

32bit

3=0000 0000 0000 0000 0000 0000 0000 0011

~3 =1111 1111 1111 1111 1111 1111 1111 1100

__rb_parent_color & ~3=过滤掉低2位（非0变成0），保留第3-32位。由于__rb_parent_color是无符号整数，强制转换成struct rb_node *类型。

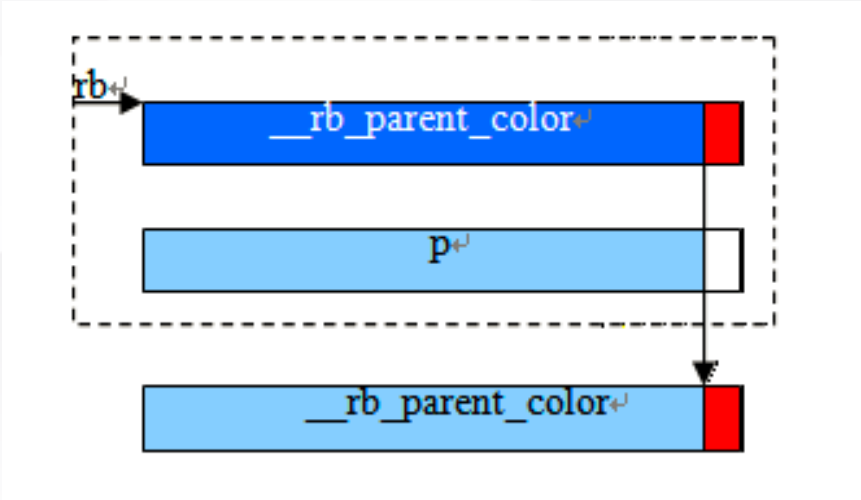
Linux内核中红黑树的操作2

```
#define rb_color(r)   ((r)->rb_parent_color & 1) //获得颜色属性

#define rb_is_red(r)   (!rb_color(r))   //判断颜色属性是否为红

#define rb_is_black(r) rb_color(r) //判断颜色属性是否为黑

#define rb_set_red(r)  do { (r)->rb_parent_color &= ~1; } while (0)  //设置红色属性

#define rb_set_black(r)  do { (r)->rb_parent_color |= 1; } while (0) //设置黑色属性

#define RB_EMPTY_ROOT(root)      ((root)->rb_node == NULL) //判断树是否为空
```

设置父结点指针

更新节点的父结点指针使得其指向新的父结点

```
static inline void rb_set_parent(struct rb_node *rb, struct rb_node *p)
//设置其双亲结点首地址的函数
{
        rb->rb_parent_color = (rb->rb_parent_color & 3) | (unsigned long)p;
}
```



设置父结点指针和节点颜色

```
static inline void rb_set_color(struct rb_node *rb, int color)
//设置结点颜色属性的函数
{
        rb->rb_parent_color = (rb->rb_parent_color & ~1) | color;
}
```

初始化新结点

```
static inline void rb_link_node(struct rb_node * node, struct rb_node * parent,
                                struct rb_node ** rb_link)
{
        node->rb_parent_color = (unsigned long )parent;
        //设置其双亲结点的首地址(根结点的双亲结点为NULL),且颜色属性设为黑色
        node->rb_left = node->rb_right = NULL;   //初始化新结点的左右子树
        *rb_link = node;  //指向新结点
}
```

指向红黑树根结点的指针

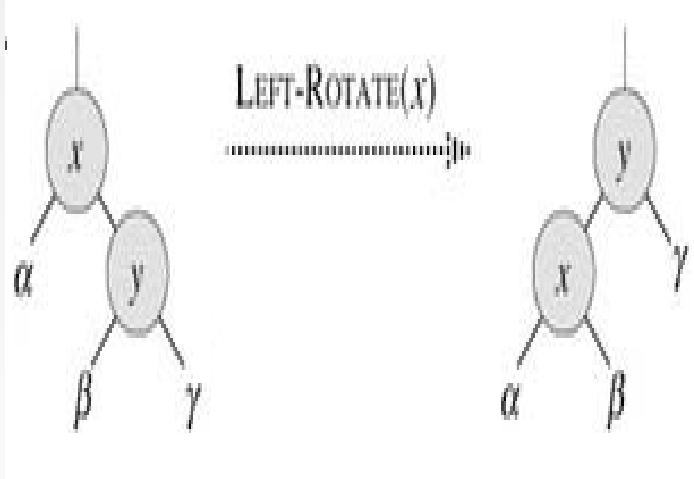
```
struct rb_root
{
        struct rb_node *rb_node;
};
#define RB_ROOT          (struct rb_root) { NULL, }  //初始化指向红黑树根结点的指针
#define  rb_entry(ptr, type, member)       container_of(ptr, type, member)
                                          //用来获得包含struct rb_node的结构体的首地址
#define RB_EMPTY_ROOT(root)       ((root)->rb_node == NULL) //判断树是否为空
#define RB_EMPTY_NODE(node)       (rb_parent(node) == node)  //判断node的双亲结点是否为自身
#define RB_CLEAR_NODE(node)       (rb_set_parent(node, node)) //设置双亲结点为自身
```

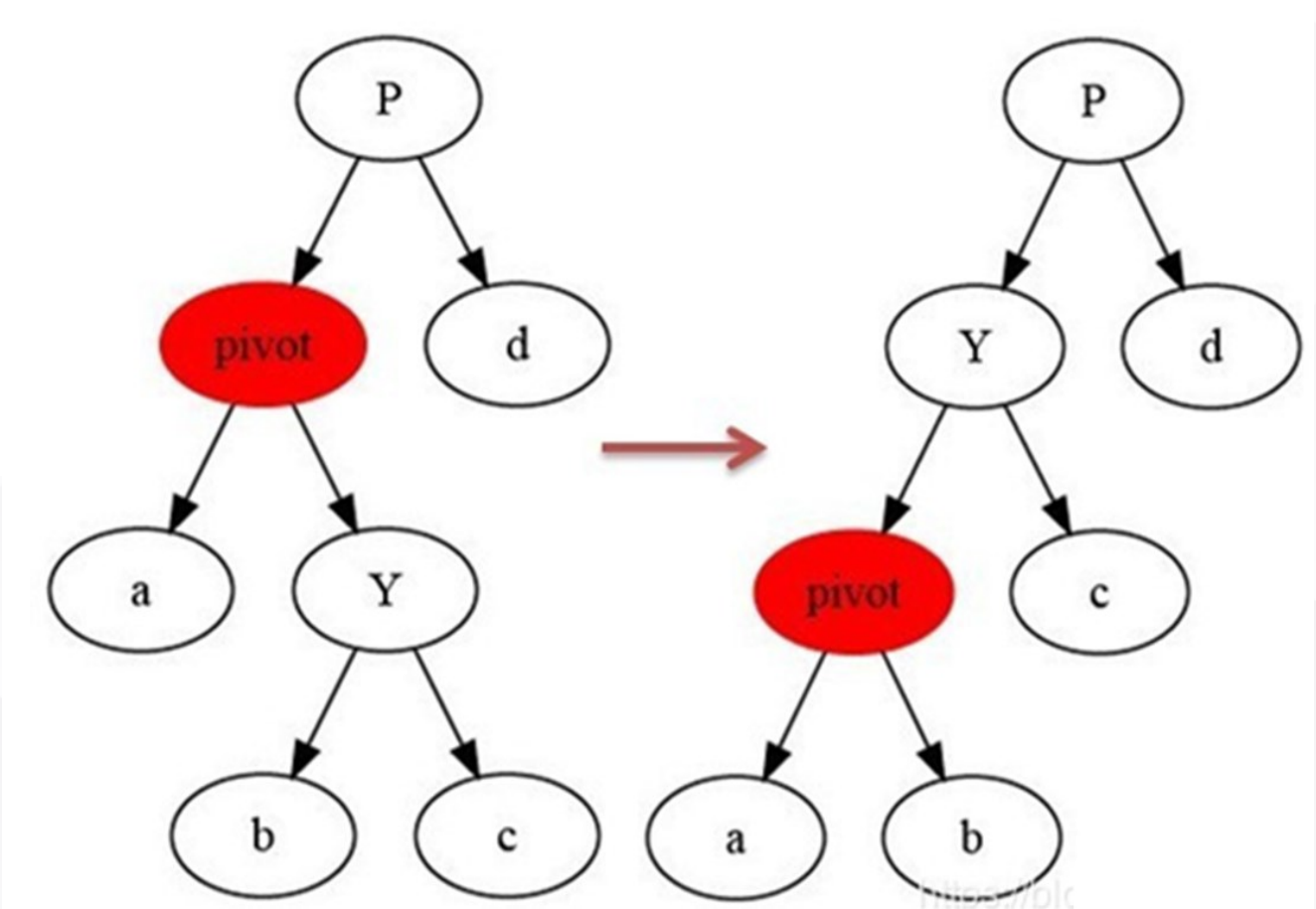
红黑树的插入1

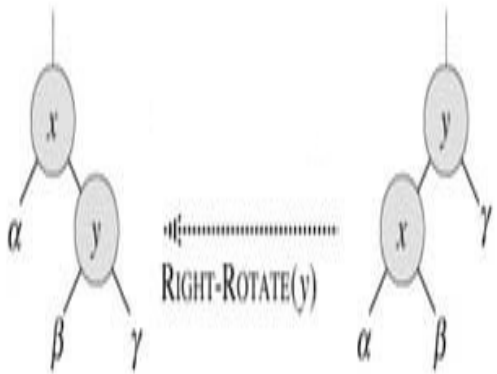
- 首先像二叉查找树一样插入一个新结点，然后根据情况作出相应的调整，以使其满足红黑树的颜色属性（其实质是维持红黑树的平衡）。
- 针对扩展红黑树的操作，必须提供特定操作的回调函数。核心定义一个扩展红黑树的回调函数数据结构。包括三部分，分别是：**繁殖，拷贝，旋转**
- linux/rbtree_augmented.h

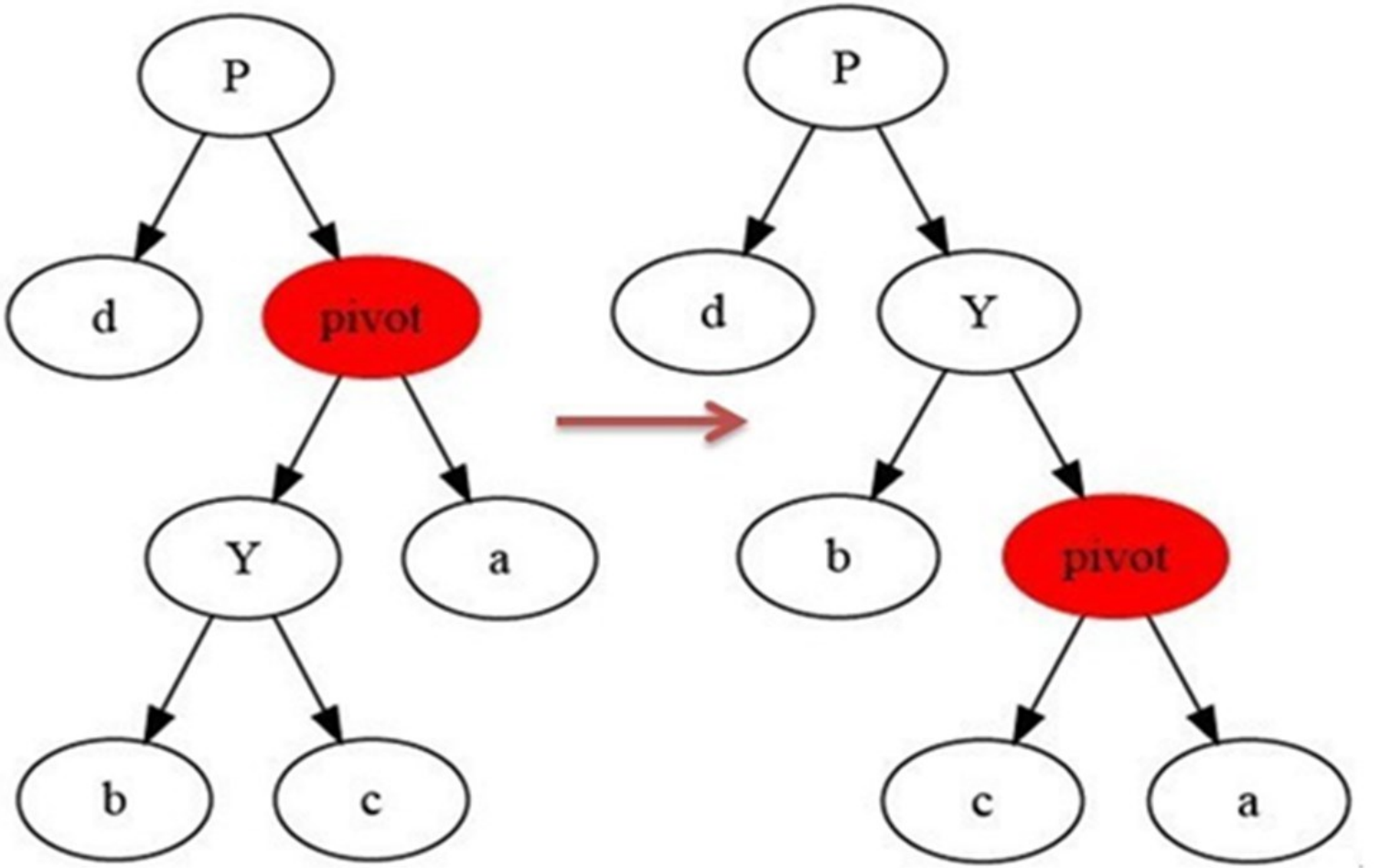
```
39 struct rb_augment_callbacks {
40         void (*propagate)(struct rb_node *node, struct rb_node *stop);
41         void (*copy)(struct rb_node *old, struct rb_node *new);
42         void (*rotate)(struct rb_node *old, struct rb_node *new);
43 };
```

- 函数rb_insert_color使用while循环不断地判断双亲结点是否存在，且颜色属性为红色。
- 若判断条件为真，则分成两部分执行后续的操作


LEFT-ROTATE(x)
x
α
y
β
γ
y
x
γ
α
β


P
pivot
d
a
Y
b
c
P
Y
d
pivot
c
a
b


x
y
α
β
γ
RIGHT-ROTATE(y)
y
x
α
β
γ


P
d
pivot
Y
a
b
c
P
d
Y
b
pivot
c
a

红黑树的操作

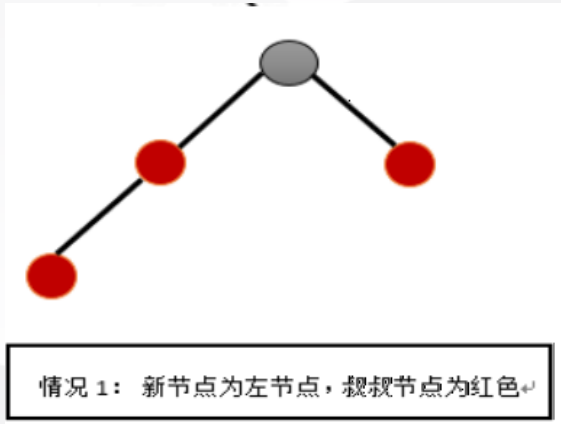
- rbtree的插入、删除操作除去根节点，其他的节点操作无外乎下面几种情况

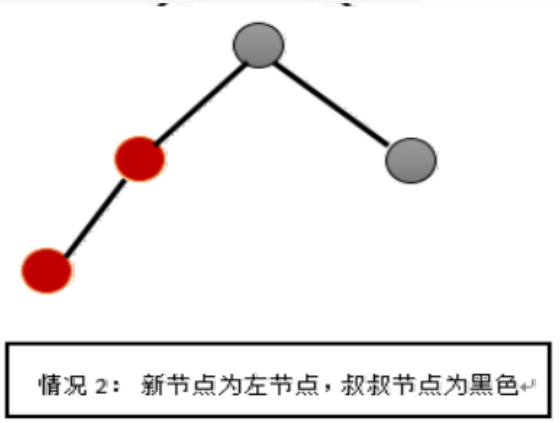
C1：新节点为左节点，叔叔节点为红色。

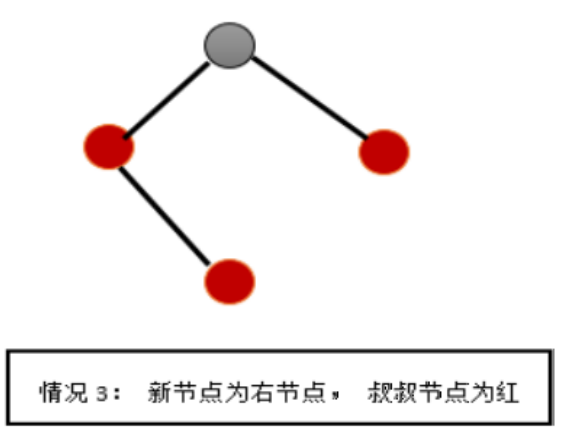
C2：新节点为左节点，叔叔节点为黑色。

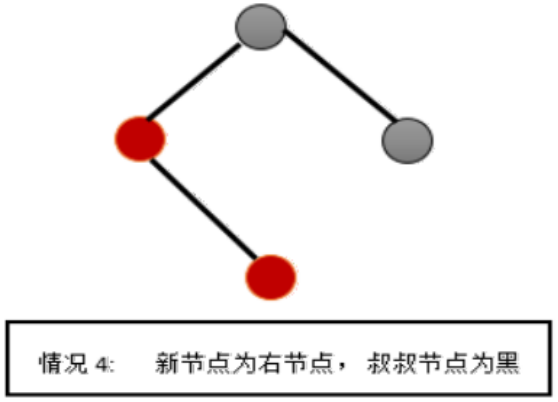
C3：新节点为右节点，叔叔节点为红色。

C4：新节点为右节点，叔叔节点为黑色。





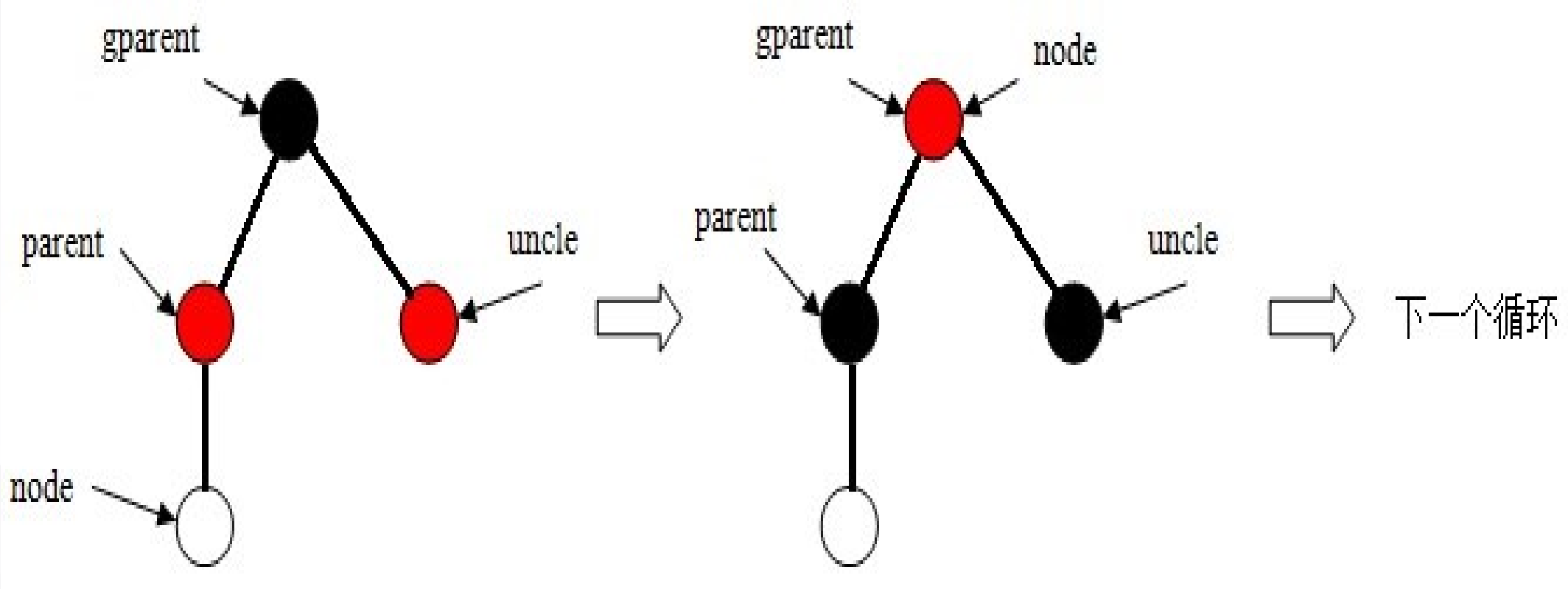




红黑树的插入2

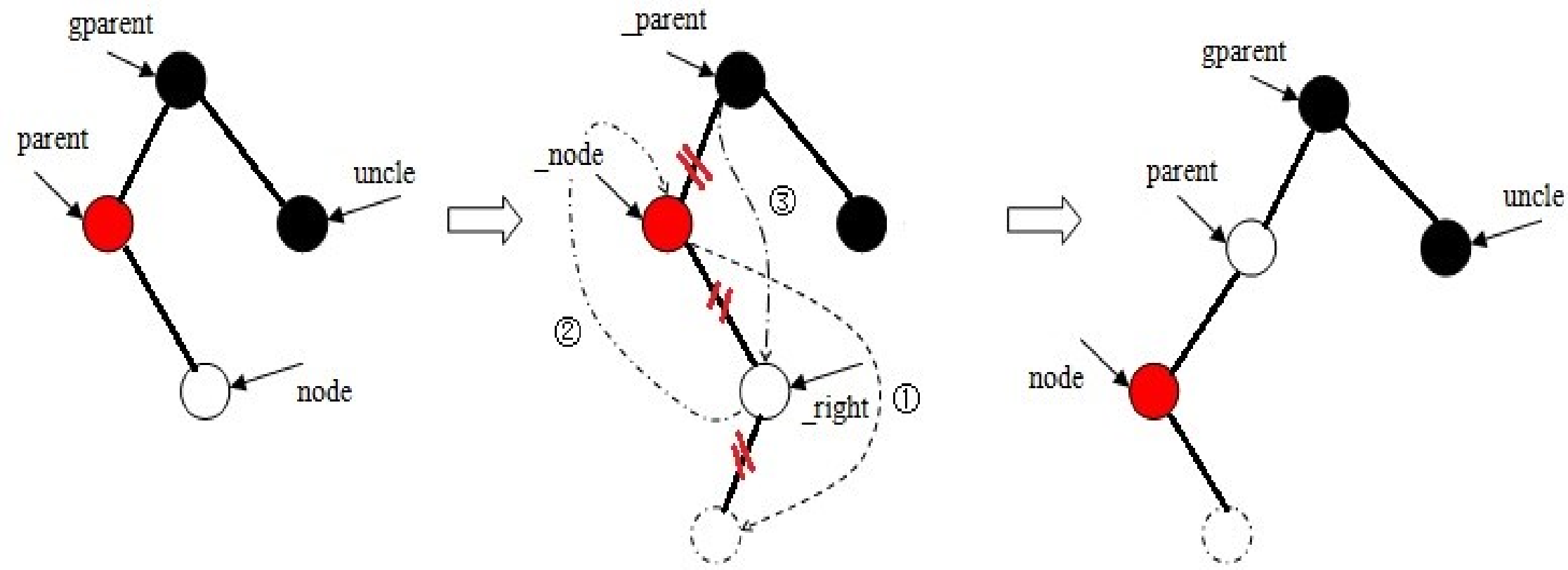
（1）、当双亲结点是祖父结点左子树的根时，则：

a、存在叔父结点，且颜色属性为红色。



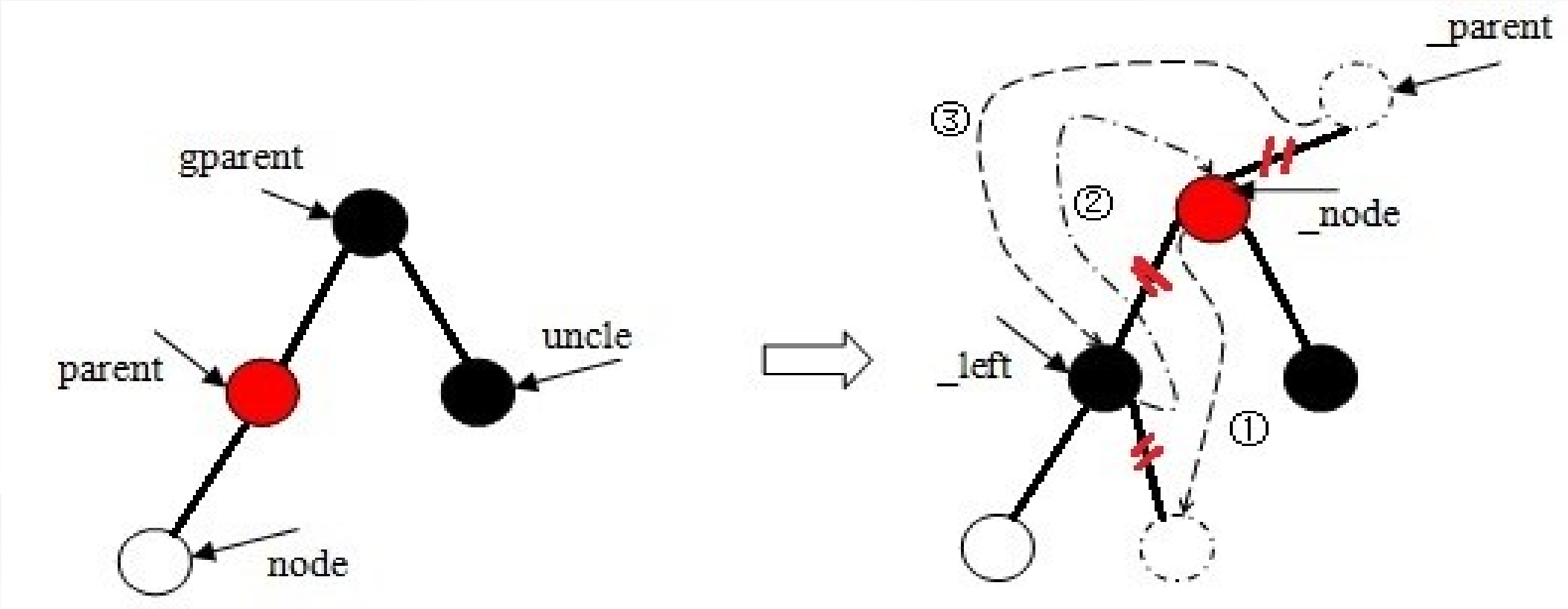
红黑树的插入3

b、当node是其双亲结点右子树的根时，则左旋，然后执行第c步。



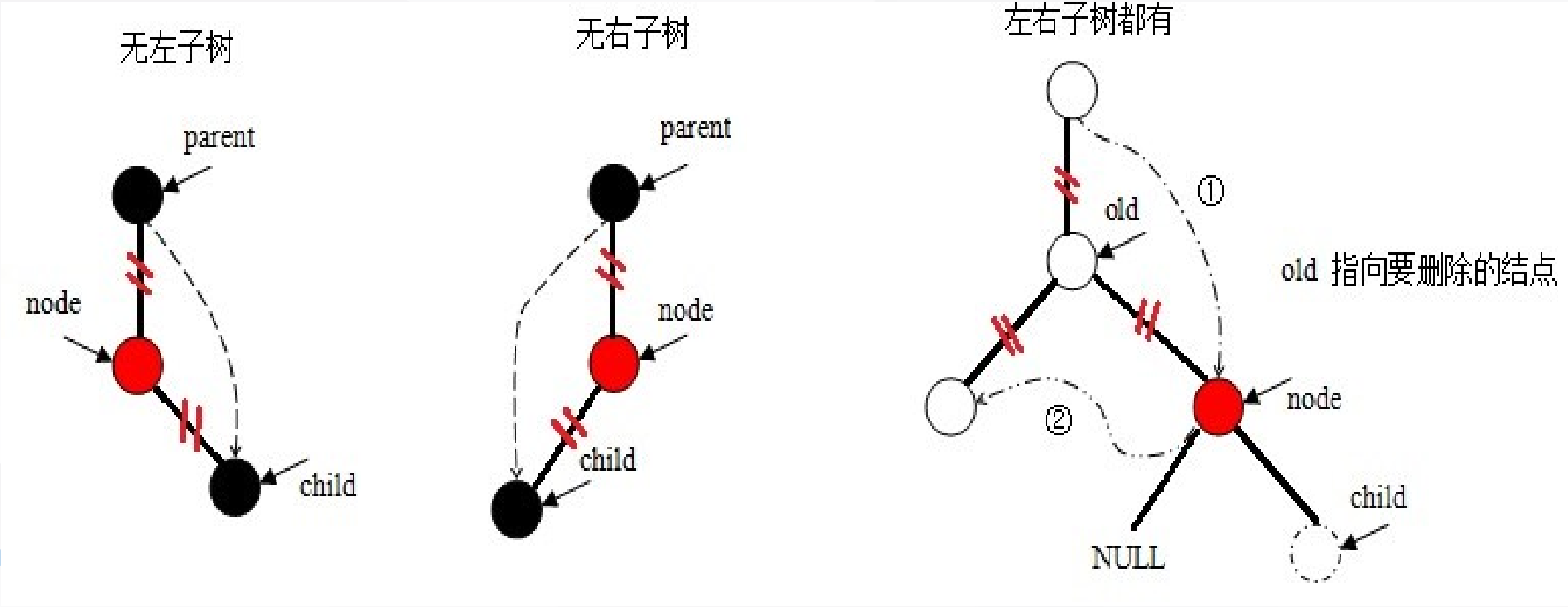
红黑树的插入4

c、当node是其双亲结点左子树的根时。



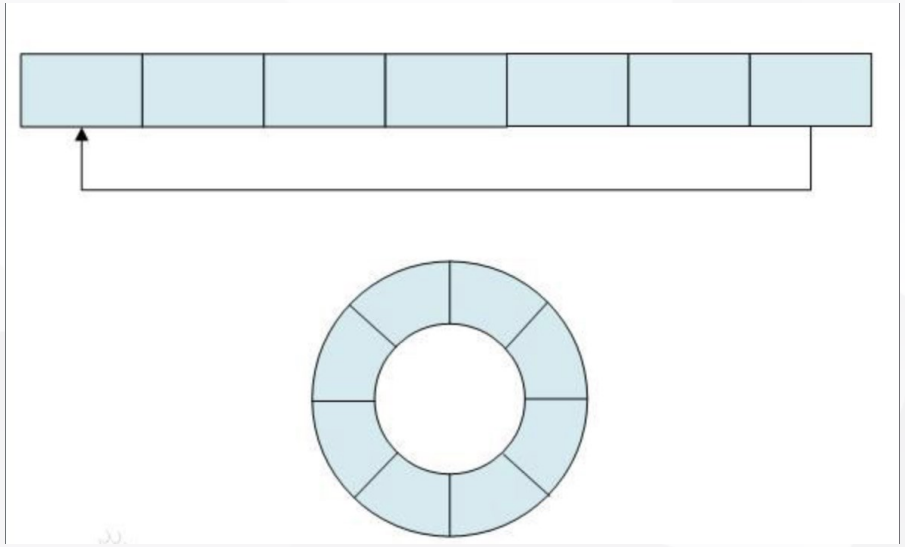
红黑树的删除

- 像二叉查找树的删除操作一样，首先需要找到所需删除的结点，然后根据该结点左右子树的有无分为三种情形：



无锁环形缓冲区

- 环形缓冲区
  - ✓ 生产者和消费者模型是计算机编程中最常见的一种模型。
  - ✓ 生产者产生数据，而消费者消耗数据，如一个网络设备，硬件设备接收网络包，然后应用程序读取网络包。
  - ✓ 环形缓冲区是实现生产者和消费者模型的经典算法。环形缓冲区通常有一个读指针和一个写指针。读指针指向环形缓冲区中可读的数据，写指针指向环形缓冲区可写的数据。通过移动读指针和写指针实现缓冲区数据的读取和写入。



KFIFO无锁环形缓冲区

- KFIFO
  - ✓ 在Linux内核中，KFIFO是采用无锁环形缓冲区的实现。FIFO的全称是“First In First Out”，即先进先出的数据结构，它采用环形缓冲区的方法来实现，并提供一个无边界的字节流服务。采用环形缓冲区的好处是，当一个数据元素被消耗之后，其余数据元素不需要移动其存储位置，从而减少复制，提高效率。

- KFIFO的API
  - ✓ 创建KFIFO：INIT_KFIFO(fifo)
  - ✓ 入列：kfifo_in(fifo, buf, n)
  - ✓ 出列：kfifo_out(fifo, buf, n)
  - ✓ 获取大小：kfifo_size(fifo)
  - ✓ 判断空或满：kfifo_is_empty(fifo)　　kfifo_is_full(fifo)