# 紅黑樹

Turn off your  please



## 内容纲要

- 红黑树
  - 查找表
    - 主要操作
    - 查找效率
    - 静态查找表
    - 动态查找表
  - 二叉查找树
    - 二叉树
    - BST 树
    - AVL 树
    - 红黑树
  - 红黑树应用
    - Linux 存储管理
    - Linux 调度算法 CFS

## 基本概念

- 查找表是数据项的集合，核心操作是插入、删除、查找。
- 查找表的主要操作：

| | |
|---|---|
| 插入 | 插入新数据项 |
| 查找 | 查找给定关键字的数据项 |
| 删除 | 删除给定关键字的数据项 |
| 合并 | 合并两个查找表 |
| 选择 | 选择第 $k$ 小元 |

- 查找表分为静态查找表和动态查找表，静态查找表构成之后数据项不再增删，而动态查找表中的数据项可能会频繁增删。
- 查找表的实现称为查找算法。





## 平均查找长度

- 比较关键字是查找的典则操作，评价标准为平均查找长度ASL，更细致的评价标准要考虑查找成功（表中存在查找的数据项）与查找不成功（表中不存在查找的数据项）两种情形，分别记为$\mathrm{ASL}_{\mathrm{succ}}$、$\mathrm{ASL}_{\mathrm{unsucc}}$，有时还要区分等概率与不等概率两种情形。
- 设查找表中数据项个数为 $n$，则

$$\mathrm{ASL} = P_1 C_1 + P_2 C_2 + \cdots + P_i C_i + \cdots + P_n C_n$$

其中，$C_i$ 表示查找到第 $i$ 个数据项所需的比较次数，$P_i$ 表示查找表中第 $i$ 个数据项的概率。

- 等概率情形

$$\mathrm{ASL} = \frac{C_1 + C_2 + \cdots + C_i + \cdots + C_n}{n}$$

# 顺序查找与折半查找

- 顺序查找的平均效率：$O(n)$

```
int seq_search(int a[], int l, int r, int key)
{
  for (int i=l; i<=r; i++) if (a[i]==key) return 1;
  return 0;
}
```

- 折半查找的平均效率：$O(\log n)$

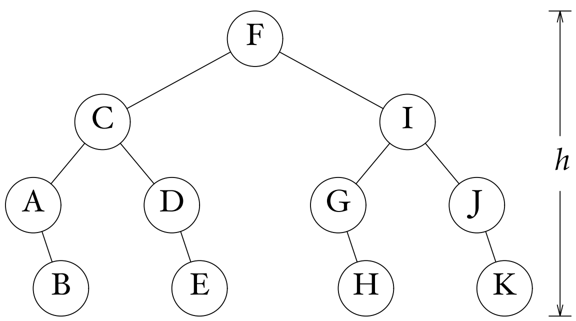
```
int bin_search(int a[], int l, int r, int key)
{
  if (l>r) return 0;
  int m = (l+r)/2;
  if      (key < a[m]) return bin_search(a, l, m-1, key);
  else if (key > a[m]) return bin_search(a, m+1, r, key);
  else return 1;
}
```





# 树与查找

- 二叉查找树：BST 树，Splay 树。



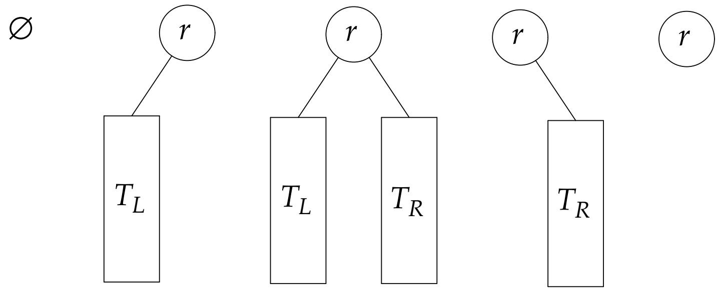

- 平衡二叉查找树：AVL 树，Red-Black 树。
- 平衡外部查找树：$B$-树，$B^+$-树。

# 二叉树定义

- 二叉树 $T$ 定义为

$$T = \begin{cases} \varnothing, & |T| = 0; \\ \{ r, T_L, T_R \}, & |T| > 0. \end{cases}$$

- 二叉树的五种形态



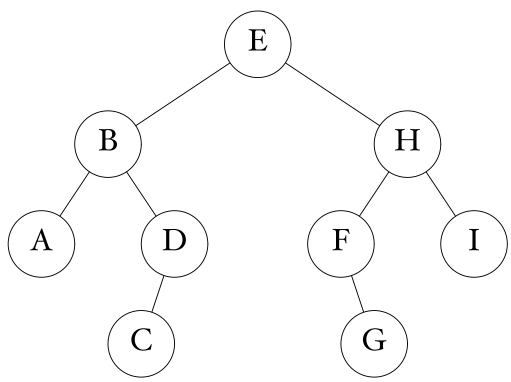

(1) (2) (3) (4) (5)





# 二叉树例

- 二叉树的图示



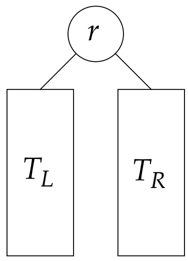

- 二叉树的广义表表示

T = ( E ( B ( A () () ) ( D ( C () () ) () ) ) ( H ( F () ( G () () ) ) ( I () () ) ) )

## BST 定义

- BST(Binary Search Tree) 是一棵二叉树，树中结点的关键字满足下式：

$$\text{key}(\,T_L\,) \le \text{key}(\,r\,) \le \text{key}(\,T_R\,)$$

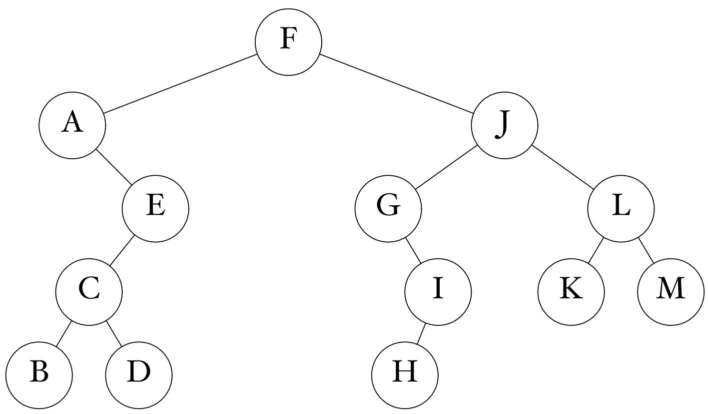



- 显然，BST 的中序遍历序列有序。
- 显然，有序表的折半查找判定树是 BST。
- 反之，BST一定是折半查找判定树吗？





## BST 举例




- F J G I H 是一次 BST 查找吗？ F J G I？ J G I H？
- A B C D 是一次 BST 查找吗？ F A E C B D？

# BST 结点定义及其初始化

```
typedef struct node *link;
struct node {
  int item;
  link l, r;
};
link NODE(int item, link l, link r)
{
  link t = malloc(sizeof *t);
  t->item = item;
  t->l = l; t->r = r;
  return t;
}
```





# BST 查找、插入操作

```
link bst_search(link t, int key)
{
  if (t==NULL) return NULL;
  if (key < t->item) return bst_search(t->l, key);
  if (key > t->item) return bst_search(t->r, key);
  return t;
}
link bst_insert(link t, int key)
{
  if (t==NULL) return NODE(key, NULL, NULL);
  if      (key < t->item) t->l = bst_insert(t->l, key);
  else if (key > t->item) t->r = bst_insert(t->r, key);
  return t;
}
```

# BST 删除操作

```
link bst_remove(link t, int key)
{
  if (t==NULL) return NULL;
  if      (key < t->item) t->l = bst_remove(t->l, key);
  else if (key > t->item) t->r = bst_remove(t->r, key);
  else {
    if      (t->l != NULL)
      t->l = bst_remove(t->l, t->item = pred(t)->item);
    else if (t->r != NULL)
      t->r = bst_remove(t->r, t->item = succ(t)->item);
    else { free(t); t = NULL; }
  }
  return t;
}
```





# BST 查找前驱、后继

- 查找前驱

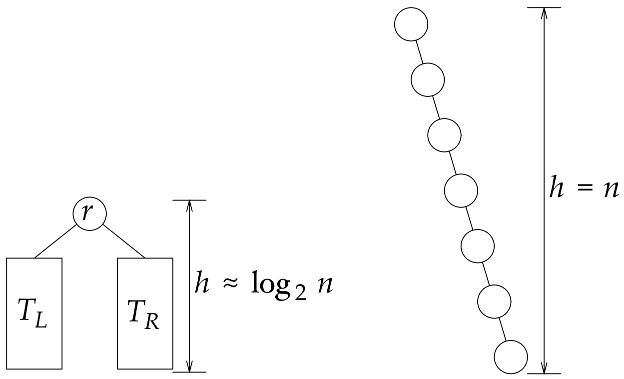
```
link pred(link t)
{
  if (t->l == NULL) return NULL;
  for (t=t->l; t->r!=NULL; t=t->r) ;
  return t;
}
```

- 查找后继

```
link succ(link t)
{
  if (t->r == NULL) return NULL;
  for (t=t->r; t->l!=NULL; t=t->l) ;
  return t;
}
```

# BST 最优、最差查找效率

- 如果 $n$ 个结点的 BST，其形态与长度为 $n$ 的有序表的折半查找判定树相同，这棵 BST 的高度最低，查找效率最优，为 $O(\log n)$。
- 如果 $n$ 个关键字按升序或降序插入 BST，这棵 BST 的高度最高，达到 $n$，这时查找效率蜕化为最差的 $O(n)$，与顺序查找相同。








# BST 平均查找效率

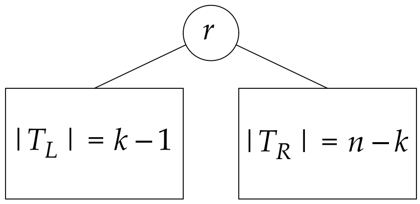
- 令 BST 树 $T$ 的结点个数为 $n = |T|$，左子树 $T_L$ 的结点个数为 $k-1$，则右子树 $T_R$ 的结点个数为 $n-k$。令查找成功的平均比较次数为 $C_n$，我们得到如下递推公式：

$$C_n = n-1+\frac{1}{n}\sum_{1\le k\le n}(C_{k-1}+C_{n-k}) = n-1+\frac{2}{n}\sum_{1\le k\le n}C_{k-1} \qquad (1)$$

求解(1)，得到

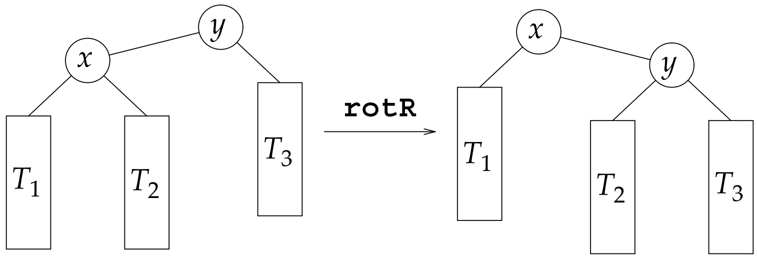
$$C_n = 2\ln n \approx 1.39\log_2 n$$

所以，BST 的平均查找效率为 $O(\log n)$。



## BST 的旋转

- BST 的基本操作除插入、查找、删除之外，还有旋转。




- 旋转之后，$T_1$，$x$，$T_2$，$y$，$T_3$ 在树中的位置仍然满足以下限定：

$$\mathrm{key}(T_1) \le \mathrm{key}(x) \le \mathrm{key}(T_2) \le \mathrm{key}(y) \le \mathrm{key}(T_3)$$

- 旋转的时间复杂度是 $O(1)$。





## 新结点插入 BST 成为根

- 经典 BST 插入的新结点都是叶子，如果在递归回程做相应旋转，那么每次插入的新结点就成为根。

```
link rotR(link t)
{ link x = t->l; t->l = x->r; x->r = t; return x; }
link rotL(link t)
{ link x = t->r; t->r = x->l; x->l = t; return x; }
link bst_insert_root(link t, int key)
{
  if (t==NULL) return NODE(key, NULL, NULL);
  if      (key < t->item)
  { t->l = bst_insert_root(t->l, key);  t = rotR(t); }
  else if (key > t->item)
  { t->r = bst_insert_root(t->r, key);  t = rotL(t); }
  return t;
}
```

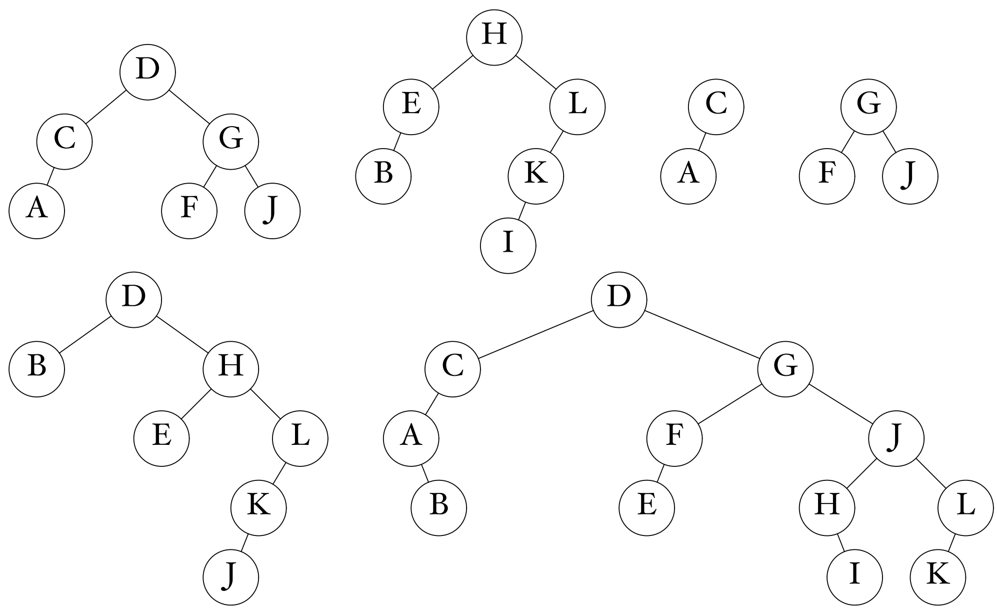
# 合并两棵 BST

```
link bst_join(link a, link b)
{
  if (b==NULL) return a;
  if (a==NULL) return b;
  b = bst_insert_root(a->item);
  b->l = bst_join(a->l, b->l);
  b->r = bst_join(a->r, b->r);
  free(a);
  return b;
}
```





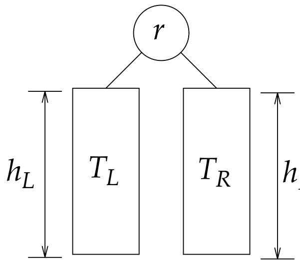
# 合并两棵 BST 举例



# AVL 树定义

- AVL 树是一棵平衡BST，即每个结点的左、右子树高度差不超过1。

$$\text{abs}(h_L - h_R) \le 1$$

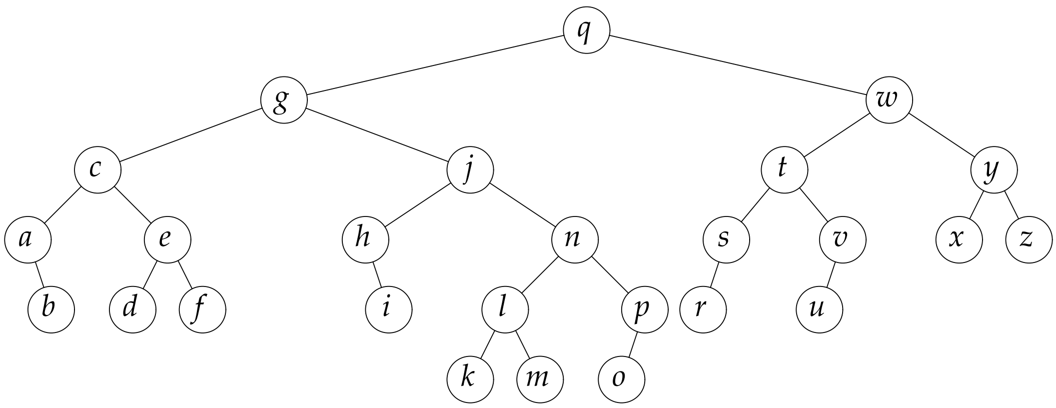



- AVL 树的查找效率在最差情形也是 $O(\log n)$。别忘了，BST 的查找效率在最差情形是 $O(n)$。



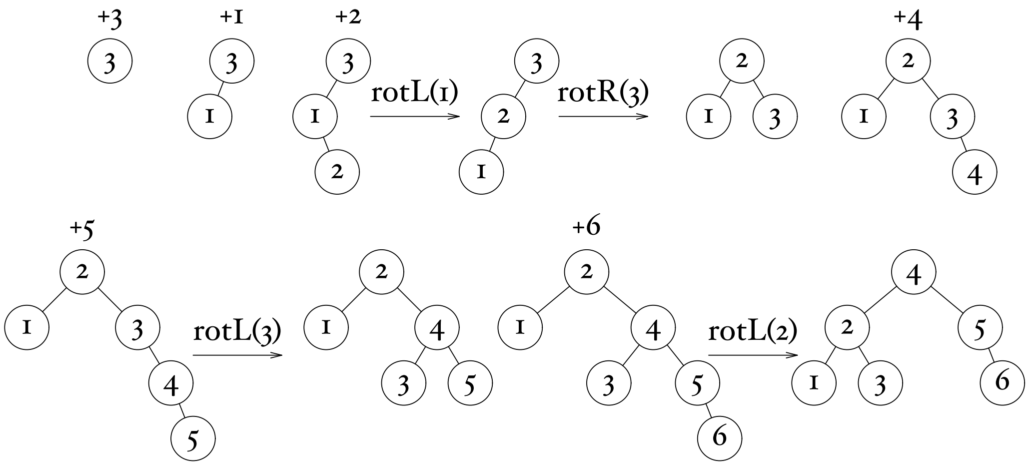


# AVL 树举例



# AVL 树插入举例

↳ 向空 AVL 树插入 3, 1, 2, 4, 5, 6。

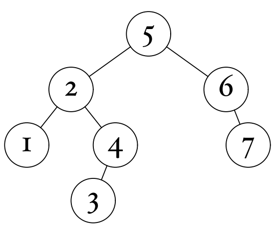


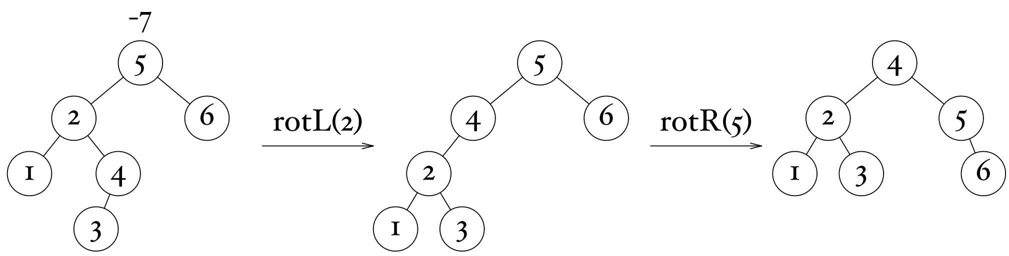





# AVL 树删除举例

↳ 从如下 AVL 树中删除 7。





## AVL 树结点定义及其初始化

```
#define tl (t->l)
#define tll (t->l->l)
#define tlr (t->l->r)
#define tr (t->r)
#define trr (t->r->r)
#define trl (t->r->l)
typedef struct node *link;
struct node { int item; char bf; link l, r; };
link NODE(int item, char bf, link l, link r)
{
  link t = malloc(sizeof *t);
  t->item = item; t->bf = bf;
  t->l= l; t->r= r;
  return t;
}
```





## AVL 树插入新结点

```
link insert_node(link t, int item, int *grow)
{
  if (t==NULL) return NODE(item, '  ', NULL, NULL);
  if (item < t->item) {
    tl = insert_node(tl, item, grow);
    if (*grow) t = growL(t, grow);
  } else {
    tr = insert_node(tr, item, grow);
    if (*grow) t = growR(t, grow);
  }
  return t;
}
link AVLinsert(link t, int item)
{
  int grow = 1;
  return insert_node(t, item, &grow);
}
```

# AVL 树左子树长高

```
link growL(link t, int *grow)
{
  switch (t->bf) {
    case ' ': *grow = 1; t->bf = 'L'; break;
    case 'R': *grow = 0; t->bf = ' '; break;
    case 'L': *grow = 0;
          char tlbf = tl->bf;
          t->bf = tl->bf = ' ';
          switch (tlbf) {
            case 'R':
              if (tlr->bf=='L') t->bf = 'R';
              if (tlr->bf=='R') tl->bf = 'L';
              tl = rotL(tl);
            case 'L': t = rotR(t); t->bf = ' ';
          }
  }
  return t;
}
```

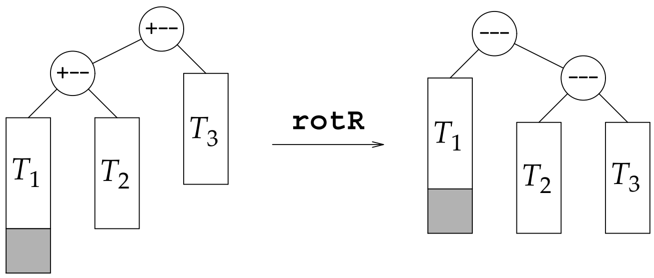


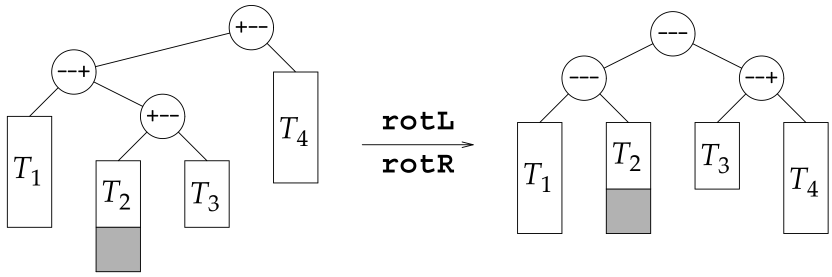


# AVL 树右子树长高

```
link growR(link t, int *grow)
{
  switch (t->bf) {
    case ' ': *grow = 1; t->bf = 'R'; break;
    case 'L': *grow = 0; t->bf = ' '; break;
    case 'R': *grow = 0;
          char trbf = tr->bf;
          t->bf = tr->bf = ' ';
          switch (trbf) {
            case 'L':
              if (trl->bf=='R') t->bf = 'L';
              if (trl->bf=='L') tr->bf = 'R';
              tr = rotR(tr);
            case 'R': t = rotL(t); t->bf = ' ';
          }
  }
  return t;
}
```

# AVL 树左子树长高

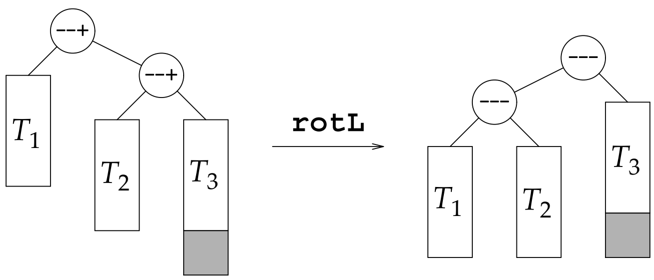


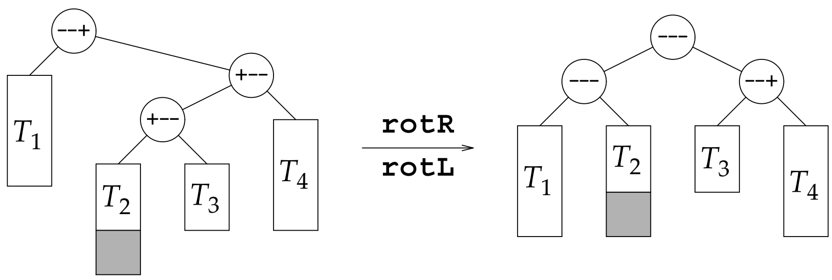





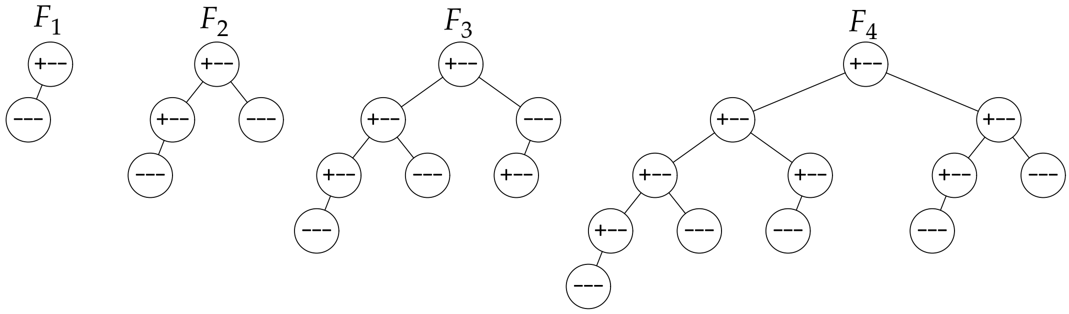



# AVL 树右子树长高



## AVL 树的最大高度等于 Fibonacci 树的高度




| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 树高 | $h$ | | | 0 | 1 | 2 | 3 | 4 | 5 | $\cdots$ |
| 树中结点个数 | $n_h$ | | | 0 | 1 | 2 | 4 | 7 | 12 | $\cdots$ |
| Fibonacci 序列 | $f_{h+2}$ | 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | $\cdots$ |

表中给出明显的关系式 $n_h = f_{h+2} - 1\ (h \ge 0)$





## AVL 树在最差情形的查找效率

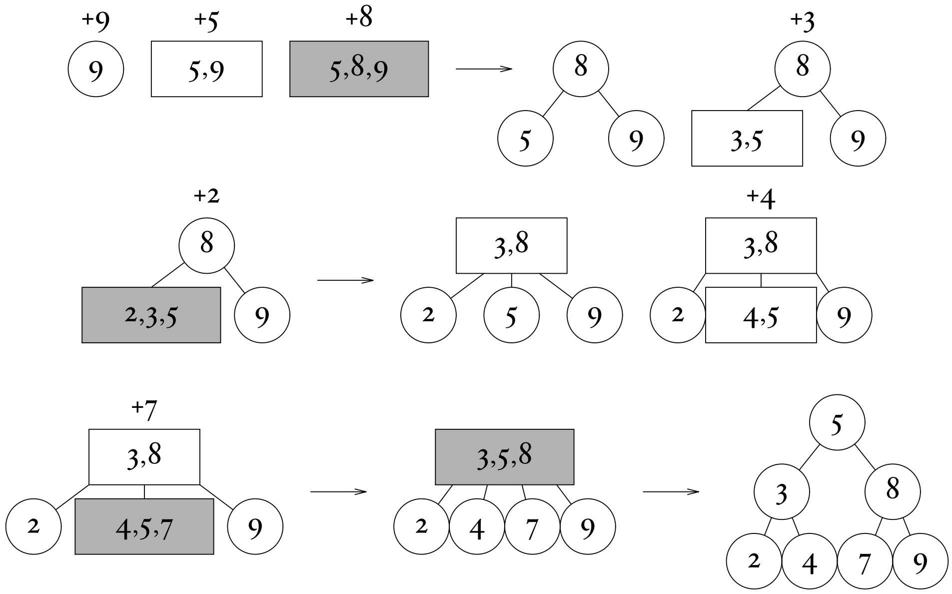
- Fibonacci 序列的通项公式为 $f_k = \frac{1}{\sqrt{5}}(\phi^k - \hat{\phi}^k)$，$k \ge 0$
- $n_h + 1 = f_{h+2} \approx \frac{1}{\sqrt{5}}\phi^{h+2}$，故

$$h \approx \log_{\phi}\sqrt{5}(n_h + 1) - 2 \approx 1.44\log_2 n_h$$

- 由于深度为 $h$ 的 AVL 树的最少结点个数是 $n_h$，所以，反过来说，结点个数为 $n$ 的 AVL 树，其最大深度是 $O(\log n)$。
- 因此，AVL 树查找的复杂度在最差情形为 $O(\log n)$。
- $\phi = \frac{1+\sqrt{5}}{2}$，$\hat{\phi} = 1 - \phi = \frac{1-\sqrt{5}}{2}$，$\hat{\phi}^k \to 0 (k \to \infty)$

# 2-3 树

➯ 2-3 树是由 2-结点和 3-结点构成的平衡查找树。插入 9, 5, 8, 3, 2, 4, 7 序列：

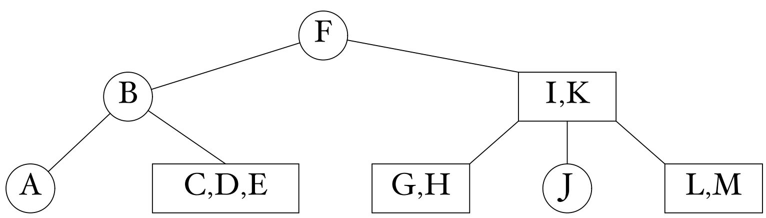


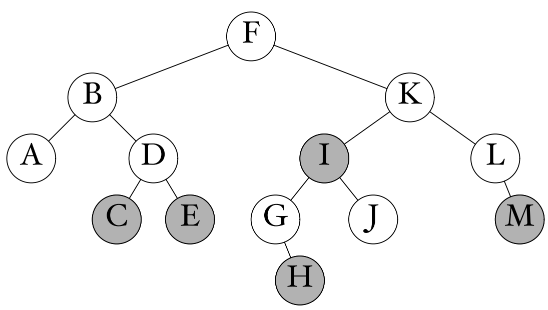





# 红黑树定义

➯ 红黑树是由 2-结点构成的 2-3-4 平衡查找树



# 红黑树结点定义及其初始化

```
typedef struct node *link;
struct node { int item; int red; link l, r; };
link NODE(int item, link l, link r, int red)
{
  link t = malloc(sizeof *t);
  t->item = item; t->red = red; t->l= l; t->r= r;
  return t;
}
link null;
void init_null()
{
  null = NODE(0, NULL, NULL, 0); null->l = null->r = null;
}
link RBinsert(link t, int item)
{
  t = insert_node(t, item, 0); t->red = 0;
  return t;
}
```



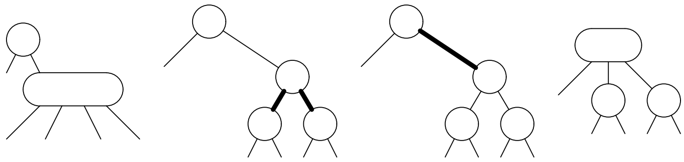


# 红黑树插入

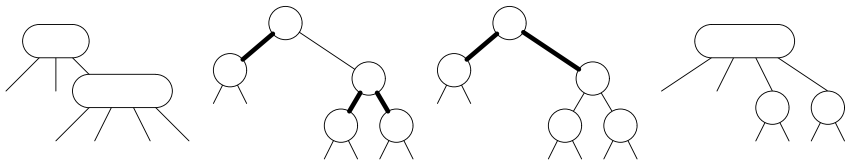
```
link insert_node(link t, int item, int sw)
{
  if (t==null) return NODE(item, null, null, 1);
  if (t->l->red && t->r->red)
  { t->red = 1; t->l->red = 0; t->r->red = 0; }
  if (item < t->item) {
    t->l = insert_node(t->l, item, 0);
    if (t->red && t->l->red && sw) t = rotR(t);
    if (t->l->red && t->l->l->red)
    { t = rotR(t); t->red = 0; t->r->red = 1; }
  } else {
    t->r = insert_node(t->r, item, 1);
    if (t->red && t->r->red && !sw) t = rotL(t);
    if (t->r->red && t->r->r->red)
    { t = rotL(t); t->red = 0; t->l->red = 1; }
  }
  return t;
}
```

# 红黑树调整结点颜色

- 从 2-结点走向 4-结点，无需旋转。



- 从 3-结点走向 4-结点，无需旋转。

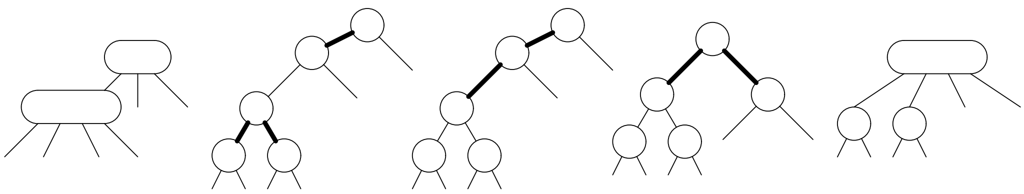




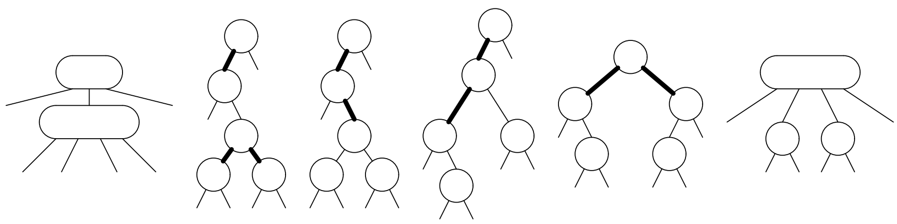


# 红黑树调整结点颜色

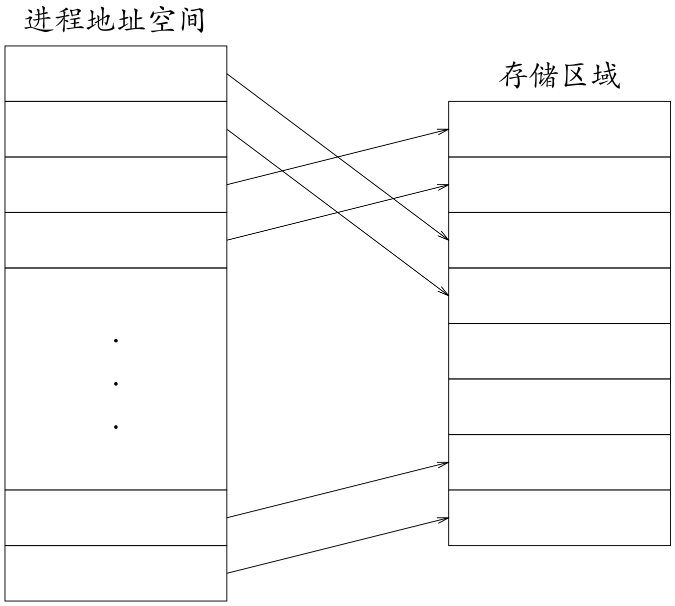
- 从 3-结点走向 4-结点，需单旋。



- 从 3-结点走向 4-结点，需双旋。



# 进程地址空间与存储区域








# 存储管理的主要数据结构

```
 struct mm_struct {
   struct vm_area_struct *mmap; /* list of memory areas    */
   struct rb_root;              /* red-black tree of VMAs */
    ...
 };
struct vm_area_struct {
  struct mm_struct      *vm_mm;    /* associated mm_struct   */
  unsigned long         vm_start;  /* VMA start, inclusive   */
  unsigned long         vm_end;    /* VMA end,   exclusive   */
  struct vm_area_struct *vm_next;  /* list of VMA's          */
  pgprot_t          vm_page_prot;  /* access permissions     */
  unsigned long         vm_flags;  /* flags */
  struct rb_node        vm_rb      /* VMA's node in the tree */
   ...
};
```

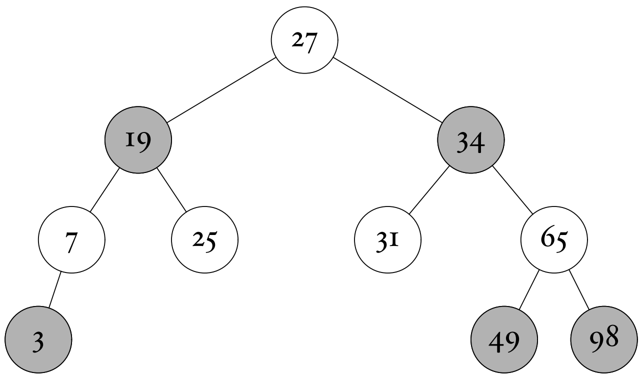
## Linux 调度算法简介

- Linux-1.2：数据结构是循环链表，插入、删除简单。
- Linux-2.2：引入调度类，包括实时、非实时、抢占、非抢占调度，开始支持 SMP。
- Linux-2.4：$O(n)$。
- Linux-2.6：由 Ingo Molnar 实现，数据结构为多级队列。虽然复杂度为 $O(1)$，但采用启发式规则，代码庞大无规律。
- Con Kolivas's RSDL(Rotating Staircase Deadline Scheduler) 支持公正调度，实现简明，启发了 CFS。
- Linxu-2.6.23：Ingo Molnar's CFS (Completely Fair Scheduler)，时间复杂度为 $O(\log n)$，数据结构采用红黑树，调度策略完全基于数学函数，计算具有一致性。





## CFS 调度算法的红黑树




每个结点表示一个任务，结点中数据表示任务的 virtual runtime。

# 参考文献


[1] 严蔚敏、吴伟民.《数据结构（C语言版）》. 清华大学出版社.

[2] R. Sedgewick. *Algorithms in C*, Part 1—4. 3rd Ed., Addison-Wesley, 1998.

[3] R. Love. *Linux Kernel Development*. 2nd Ed., Novell Press, 2005.

[4] <http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/index.html>






# Trees

I think that I shall never see
A poem lovely as a tree.

A tree whose hungry mouth is prest
Against the sweet earth's flowing breast;

A tree that looks at God all day,
And lifts her leafy arms to pray;

A tree that may in summer wear
A nest of robins in her hair;

Upon whose bosom snow has lain;
Who intimately lives with rain.

Poems are made by fools like me,
But only God can make a tree.