Searching: Introduction and Notation

Information retrieval is one of the most important applications of computers.

1. Search Table and Keys

查找操作的对象(集合), 但对象中数据元素之间的关系并未指定, 因此"松散".

查找

For a given key(target), to find a record from the set that contains other information associated with the key

```
typedef … KeyType
typedef struct list{
  …
  KeyType key;
  … } List;
```

- 顺序/二分/分块查找(查找对象是线性表)
- BST/AVL(查找对象是树)及Hash表查找

Static search table
Dynamic search table

⇩ Two search possibilities

① More than one records with the same key ② No record at all with the given key

✓ Primary key
✗ Secondary key

Locate the record(s)

2. Assume keys are any generic type and comparable

```
Several Macros:
#define EQ(a, b) ((a)==(b))
#define LT(a, b) ((a)<(b))
#define EQ(a, b) (!strcmp((a), (b)))
#define LT(a, b) (strcmp((a), (b))<0)
```

↔

```
Several new functions:
Boolean EQ(KeyType k1, KeyType k2);
Boolean LT(KeyType k1, KeyType k2);
Boolean GT(KeyType k1, KeyType k2);
```

3. External and Internal Searching

①many records, or
②each one quite large

⇨ Be external to the computer memory ⇨ B-trees(B树)

4. Time Complexity Analysis ⇨ ASL(Average Search Length)

Sequential Search / 顺序查找⇨查找对象是线性表(以顺序表为例)

```
typedef struct list {
    ElemType elem[Maxsize+1];
    int length; } List;
```
⇦
```
typedef struct {
    KeyType key;
    … } ElemType;
```
⇦
```
typedef  … KeyType;
```

```
//SequentialSearch: contiguous version
//Pre: The contiguous list has been created. Post: If an element in list has key
//equal to target, then the function returns the location of the lement(Success).
//Otherwise the function returns 0(Failure).
int SequentialSearch(List list, KeyType target)
{ int location;
  for (location=1; location<=list.length; location++ )
     if (EQ(list.elem[location].key, target)) return location;
  return 0; }
```

和P216~217/算法9.1相比, why “sentinel”?

```
int SequentialSearch(List list, KeyType target)  //P216-217/A9.1
{ int location;
  list.elem[0].key = target;  //list.elem[0].key is a sentinel
  for (location=list.length;!EQ(list.elem[location].key,target);location-- );
  return location; }
```

```
//list.elem[list.length] = target;
//for (location=0; !EQ(list.elem[location].key, target); location++ );
```

Sequential Search / 顺序查找

```
int SequentialSearch(List list, KeyType target)  //P216-217/A9.1
{ int location;
  list.elem[0].key = target;  //list.elem[0].key is a sentinel
  for (location=list.length; !EQ(list.elem[location].key, target); location-- );
  return location; }
```

算法分析(假设查找表中有n个数据元素/记录, 即表长list.length=n)

$MSL_{ss.succ}$ (最大查找长度) = n (即在最坏情况下, 查找成功的比较个数)

$MSL_{ss.unsucc}$ (= $ASL_{ss.unsucc}$) = n+1 (对表中任意数据元数的查找均不成功)

ASL_{ss} (平均查找长度) = $\sum_{i=1}^{n} P_i \times C_i$

其中, P_i为查找表中第i个数据元素的可能性/概率, $\Sigma P_i = 1$

C_i为查找第i个元素时需和target比较的关键字个数

$$ASL_{ss.succ} = \sum_{i=n}^{1} \frac{1}{n} \times (n-i+1) = \frac{n+1}{2} = O(n)$$

结论: 其查找效率依赖于查找表长度; 若元素被查找概率不同, 则概率高的排在后/前.

例如: 对n=10个记录做100次查找: 一个被查找37次, 两个各被查找25次, 三个各被查找3次, 其余各被查找1次. 若按查找次数递增顺序排列, 按A9.1, $ASL = \Sigma P_i \times C_i = 0.37 \times 1 + 0.25 \times (2+3) + 0.03 \times (4+5+6) + 0.01 \times (7+8+9+10) = 2.41 < (n+1)/2 = 5.5$.

Binary Search / 二分查找 / 折半查找

二分查找的操作对象是有序顺序表

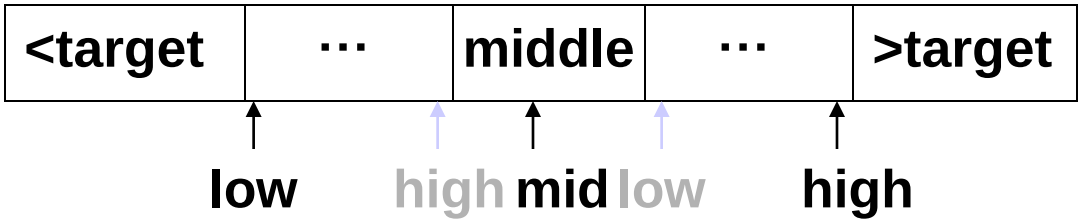
Definition: An ordered list is a list in which each entry contains a key, such that the keys are in order. That is, if entry i comes before entry j in the list, then the key of entry i is less than or equal to the key of entry j. ⇨ if i < j, then list.elem[i].key ≤ list.elem[j].key

二分查找的基本思路

1. 二分查找的操作需要设置两个辅助的指示变量(use two indices):
   low/bottom: 指示待查元素所在范围的下界, 初值 = 1(0)
   high/top: 指示待查元素所在范围的上界, 初值 = list.length(list.len-1)
2. 计算middle值并compare(target, list.elem[mid].key)

   $$mid = \lfloor \frac{low + high}{2} \rfloor$$

   if target == list.elem[mid].key, then SUCCESS
   if target < list.elem[mid].key, then high = mid-1即继续查找前半部分
   if target > list.elem[mid].key, then low = mid+1 即继续查找后半部分
3. 循环操作第2步，直至SUCCESS 或 low > high(UNSUCCESS)



Binary Search / 二分查找 / 折半查找

二分查找算法(P220/A.9.2: 非递归版本)

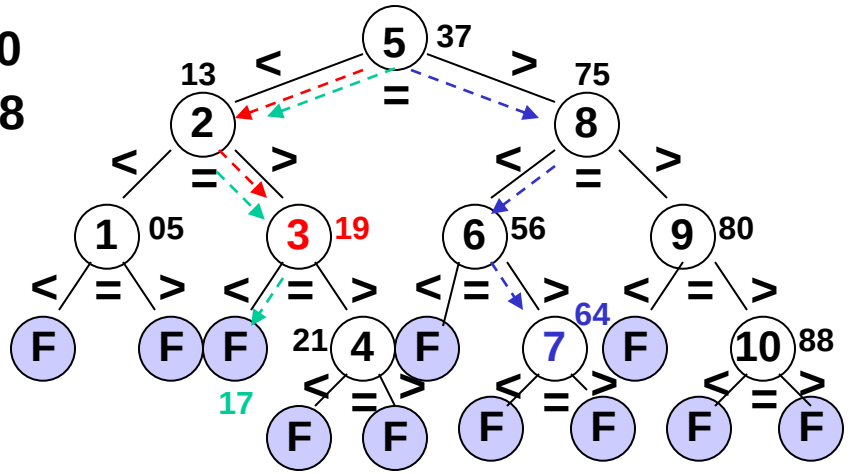
```
int BinarySearch(List list, Keytype target)
{ low = 1; high = list.length;
  while (low <= high) {
    mid = (low + high)/2;
    if (EQ(target, list.elem[mid].key))  return mid;  //SUCCESS
    else if (LT(target, list.elem[mid].key))
            high = mid – 1;  //reduce to the low half
         else low = mid + 1;  }  //reduce to the high half
  return 0;   }
```

二分查找算法分析 – 二叉判定树/比较树(decision trees/comparison trees)

二叉判定树本不存在, 用它来形象地刻画二分法的查找过程. / A decision tree is obtained by tracing through the action of the algorithm.

Order: 1 2 3 4 5 6 7 8 9 10

keys: 05, 13, 19, 21, 37, 56, 64, 75, 80, 88



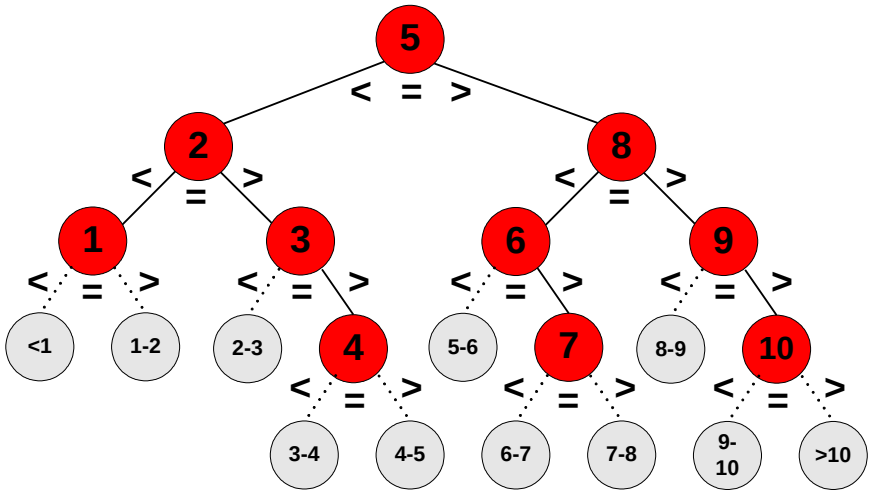
注: ①含有n个关键字的有序表, 其判定树的树形唯一, 且与有序表中elem[1..n].key的实际取值无关.

②关键字比较的个数 与 次数 不同

Binary Search / 二分查找 / 折半查找

二分查找算法分析

可见, 借助于二叉判定树很容易求得二分查找的平均查找长度ASL:
设有序表的长度为n, 如图中n=10, 则红色结点即为表中的n个元素, 并称之为二叉判定树中的内部结点, 灰色结点为n+1个叶结点(why?), 并称之为二叉判定树中的外部结点(即叶子结点), 二叉判定树是一棵扩展的二叉树, 也是一棵正则二叉树. 因此说成功的二分查找恰好走了一条从判定树的根结点到某个内部结点的路径, 其关键字比较个数为该内部结点在树中的层次数; 二分查找失败则走了一条从判定树的根结点到某个外部结点的路径, 其关键字比较次数是该路径上内部结点的总数. ⇨二分查找算法的ASL与二叉判定树的高度有直接关系.



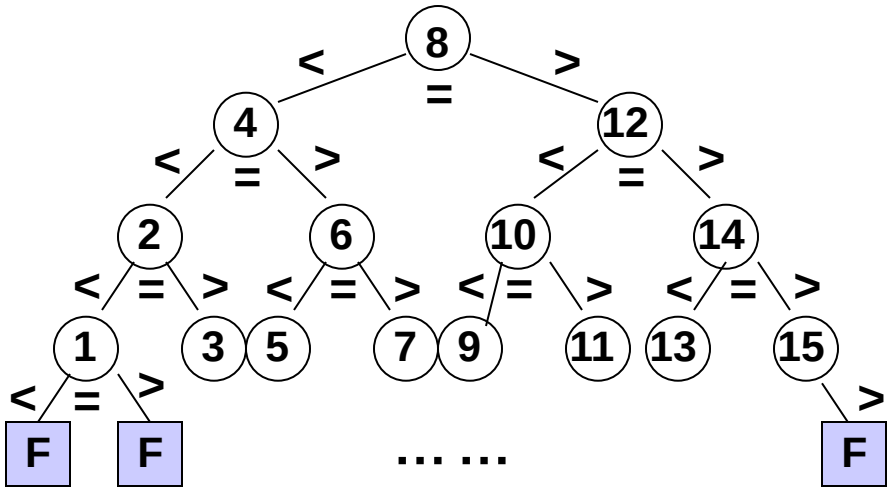
结点中的数表示数据元素在有序表中的位置序号

因为二叉判定树中度<2的结点只可能在最下面两层(不计外部结点), 所以n个内部结点的判定树和n个结点的完全二叉树的高度相同, 即为$\lceil \log_2 n \rceil \approx \log_2 n$.
为便于讨论,设 $n = 2^h - 1$, 则判定树是高度为 $h = \log_2(n+1)$的满二叉树.

$$ASL_{bs} = \sum_{i=1}^{n} P_i \times C_i = \frac{1}{n} \sum_{i=1}^{n} C_i$$

$$= \frac{1}{n} \sum_{j=1}^{h} j \times 2^{j-1} = O(\log_2 n) \text{ 个}$$

例: 设有一个有序文件, 其中各记录的key值为{1, 2, 3, … , 15}, 当用二分查找算法查找关键字2, 4, 8, 11, 17时, 其关键字比较的个数和次数各是多少?
其查找成功和查找失败的平均比较个数和次数分别是多少?(设记录数n=15)



共16个叶结点

个数分别为3, 2, 1, 4, 4
次数分别为5, 3, 1, 7, 8

先求“个数”:

$ASL_{bs.suss} = (1\times1 + 2\times2 + 4\times3 + 8\times4) / 15 = 3.26$

“次数” = 2 × “个数” - 1 = 2 × $ASL_{bs.suss}$ - 1 = 5.52

$ASL_{bs.unsuss} = (4 + 4 + \cdots + 4) / 16 = 4$

“次数” = 2 × “个数” = 2 × $ASL_{bs.unsuss}$ = 8

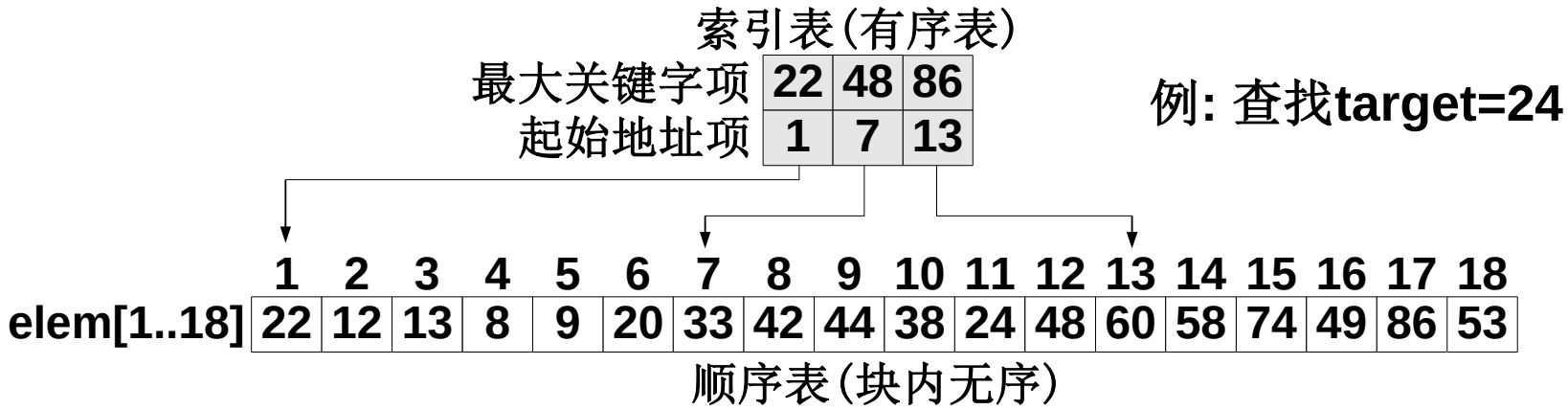
索引顺序表的查找 / Indexed Table Search P225
(又称 分块查找 / Blocking Search)

顺序查找的又一改进:

将顺序表按关键字特征“分成”若干块, 并按块建立“索引表”.

索引表(access table): an access table is an auxiliary array used to find data stored elsewhere. An access table is also sometimes called access vector. 即除顺序表本身之外, 尚需建立一个有序顺序表---“索引表”.

索引表通常包括两项内容 {
- 关键字项: 其值为该子表内最大关键字值
- 指示项: 指示子表的第一个数据元素的位置



算法思路 {
- 先在索引表中确定待查记录所在块/子表 -- 算法9.2 or 算法9.1
- 再在块/子表中进行顺序查找 -- 算法9.1

索引顺序表的查找 / 分块查找

算法分析: 分块查找的平均查找长度 $ASL_{bs} = ASL_b + ASL_s$

其中, ASL_b为查找索引表的ASL, ASL_s为查找块的ASL.

设将长度为n的顺序表平均分成b块, 每块含s个元素, 则 $b = \lceil n/s \rceil$

且设每个元素查找的概率相同.

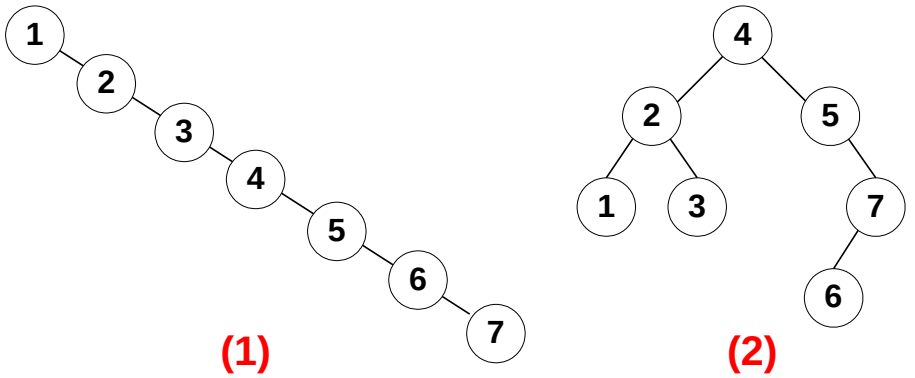
若查找索引表为二分查找, 则 $ASL_{bs} = \log_2(n/s+1) + s/2$

若查找索引表为顺序查找, 则 $ASL_{bs} = \frac{b+1}{2} + \frac{s+1}{2} = \frac{1}{2}\left(\frac{n}{s} + s\right) + 1$

平衡二叉树(Balanced Binary Trees) 又称AVL Trees

引出背景(Why AVL): 二叉查找树&查找算法分析.

对同样一组关键字序列, 若关键字排列次序不同, 则所生成的BST形态和高度可能不同. 如设有(n=7)序列 {1, 2, 3, 4, 5, 6, 7} 和 {4, 5, 7, 2, 1, 3, 6}:



$ASL_{BST\text{-}1.succ} = (1+2+3+4+5+6+7)/7 = 4 = (n+1)/2 \Rightarrow O(n)$ worst-case(顺序查找)

$ASL_{BST\text{-}2.succ} = \sum_{i=1}^{7} P_i \times C_i = (1+2\times2+3\times3+1\times4)/7 \approx 2.6 = \log_2 n \le \lceil \log_2 n \rceil \Rightarrow O(\log_2 n)$ average-case

可见, 在BST上进行查找的平均查找长度和二叉树的形态有关.

n个关键字则有n!种可能的排列序列, 数学方法已经证明(Sterling公式)由这n!个关键字序列所产生的n!棵BST(其中有些形态相同)的平均高度是$O(\log_2 n)$.

BST是在空树中通过不断插入结点而动态生成的, 那么如何在生成过程中动态地保持其"平衡", 以确保BST的高度不超过$\lceil \log_2 n \rceil$? ⇨ 平衡二叉树

平衡二叉树(Balanced Binary Trees) 又称AVL Trees

AVL树定义: 形态匀称的二叉树称为~. 具体地: 若T是一棵非空二叉树(空二叉树是AVL树), 当且仅当①| hl – hr | ≤ 1 (hl、hr是T的左、右子树的高度) ②TL、TR(指T的左、右子树)是AVL树.

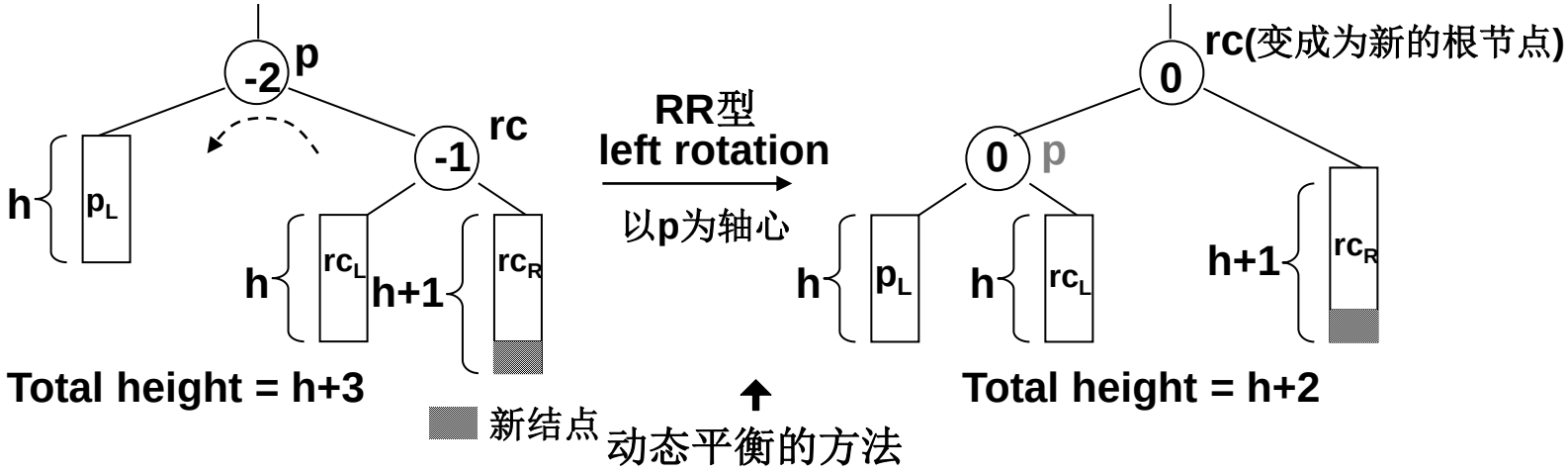
约定: 结点的平衡因子BF(Balanced Factor) = hl – hr

平衡二叉树上所有结点的BF∈{-1, 0 , +1}, 反之亦然.

平衡二叉树(Balanced Binary Trees) 又称AVL Trees

根据AVL树的定义, 如何构造出平衡二叉排序树?

①在构造二叉排序树过程中, 检查新插入的结点是否破坏了其平衡性. 若不平衡, 则找出最小不平衡子树⇨即以离新结点最近且其 |BF|>1的结点作根结点的子树(如下例, 记该子树的根结点为p).



②To re-balance a BST without violating the BST property:
keys(left) < key(root) < keys(right)

动态平衡的过程: 共2种方法、分4种情况 P234－238

平衡二叉树(Balanced Binary Trees) 又称AVL Trees

动态平衡的过程: 共2种方法、分4种情况 P234－238

2种方法 即 Left Rotation & Right Rotation

```
BTNode *leftRotate(BTNode *x) {
    BTNode *y = x->rchild;
    BTNode *T2 = y->lchild;
    //perform rotation
    x->rchild = T2; y->lchild = x;
    return y; }
```

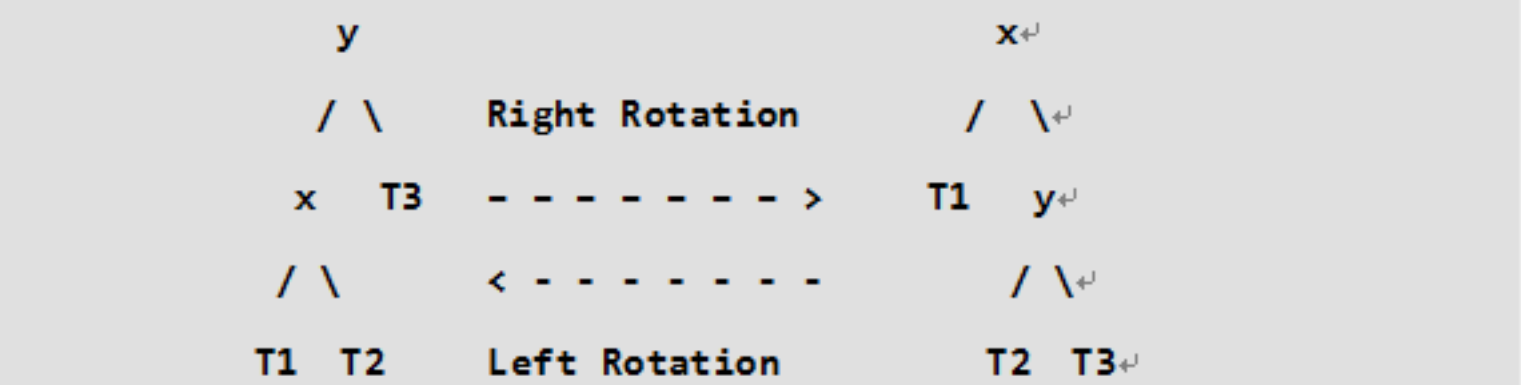
↖

↗

```
BTNode *rightRotate(BTNode *y) {
    BTNode *x = y->lchild;
    BTNode *T2 = x->rchild;
    //perform rotation
    y->lchild = T2; x->rchild = y;
    return x; }
```

T1, T2 and T3 are subtrees of the tree rooted with y (on left side)

or x (on right side)



Keys in both of the above trees follow the following order

keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)

平衡二叉树(Balanced Binary Trees) 又称AVL Trees

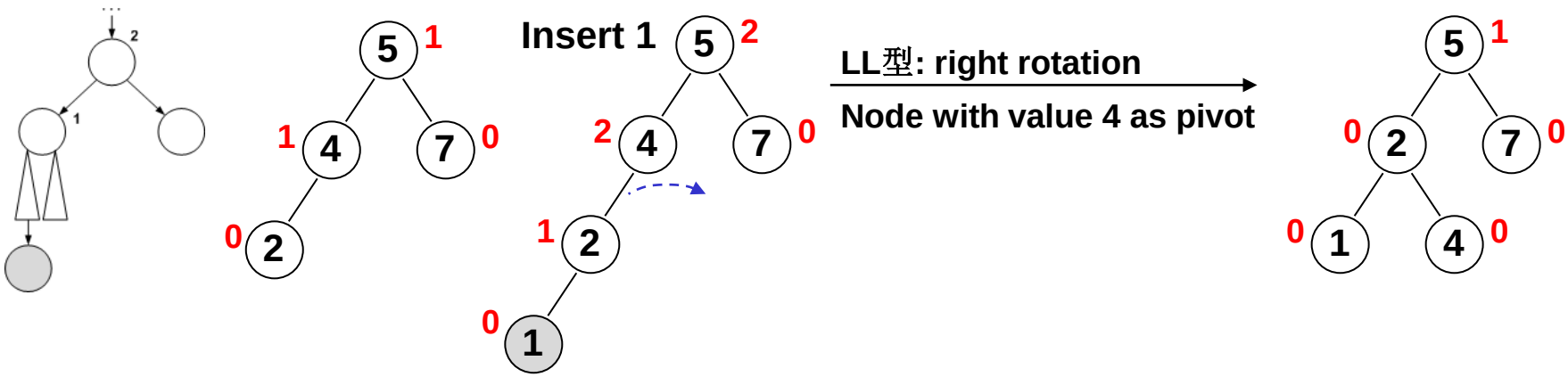
动态平衡的过程: 共2种方法、分4种情况 P234－238

①LL(left higher):新结点在z的左孩子y的左子树上➔right rotation
②RR(right higher):新结点在z的右孩子y的右子树上➔left rotation
③LR(equal height):新结点在z的左孩子y的右子树上➔double rotation
④RL(equal height):新结点在z的右孩子y的左子树上➔double rotation
其中, z是最小不平衡子树的根结点, y是z的孩子结点.

①Left Left Case

T1, T2, T3 and T4 are subtrees.

```
        z                                      y
       / \                                   /   \
      y   T4      Right Rotate (z)          x      z
     / \          - - - - - - - - ->       /  \    /  \
    x   T3                                T1  T2  T3  T4
   / \
  T1   T2
```



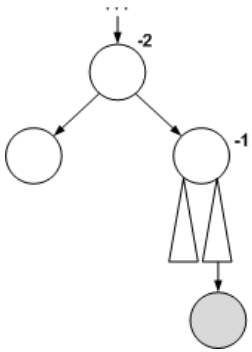
平衡二叉树(Balanced Binary Trees) 又称AVL Trees

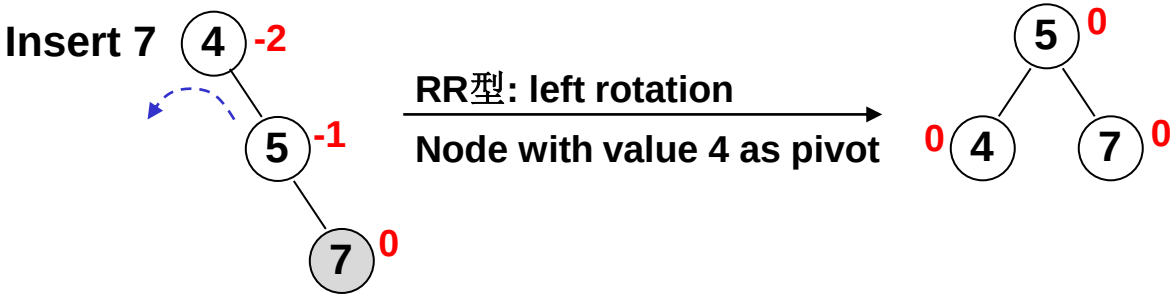
动态平衡的过程: 共2种方法、分4种情况 P234－238

①LL(left higher):新结点在z的左孩子y的左子树上➔right rotation
②RR(right higher):新结点在z的右孩子y的右子树上➔left rotation
③LR(equal height):新结点在z的左孩子y的右子树上➔double rotation
④RL(equal height):新结点在z的右孩子y的左子树上➔double rotation
其中, z是最小不平衡子树的根结点, y是z的孩子结点.

② Right Right Case

```
  z                                y
 / \                             /   \
T1   y     Left Rotate(z)       z      x
   /  \   - - - - - - - - ->   / \    / \
  T2   x                      T1  T2 T3  T4
      / \
     T3  T4
```



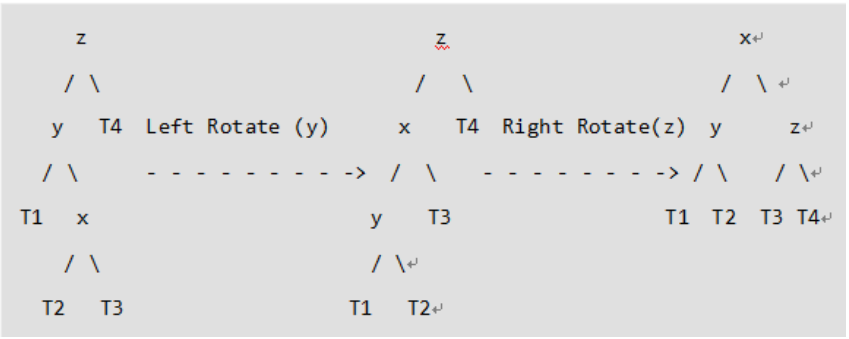


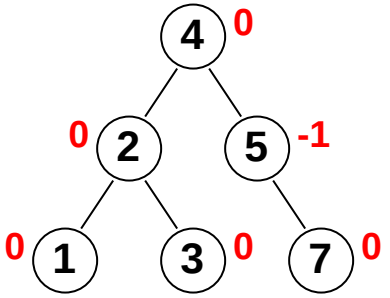
平衡二叉树(Balanced Binary Trees) 又称AVL Trees

动态平衡的过程: 共2种方法、分4种情况 P234－238

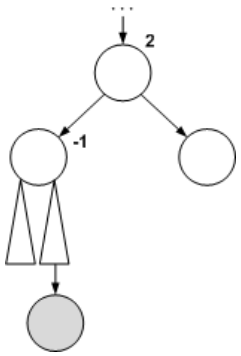
①LL(left higher):新结点在z的左孩子y的左子树上➔right rotation
②RR(right higher):新结点在z的右孩子y的右子树上➔left rotation
③LR(equal height):新结点在z的左孩子y的右子树上➔double rotation
④RL(equal height):新结点在z的右孩子y的左子树上➔double rotation
其中, z是最小不平衡子树的根结点, y是z的孩子结点.

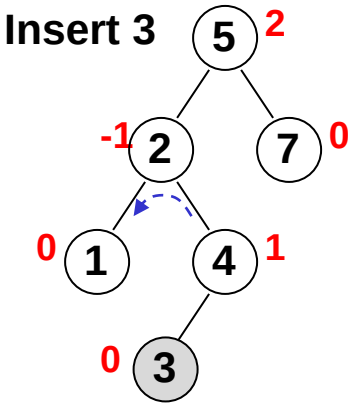
③ Left Right Case



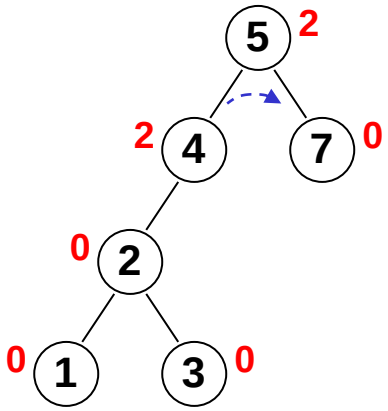


LL型: right rotation
5 as pivot





LR型: left rotation
2 as pivot



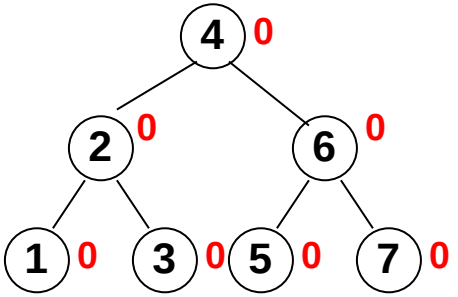
平衡二叉树(Balanced Binary Trees) 又称AVL Trees

动态平衡的过程: 共2种方法、分4种情况 P234－238

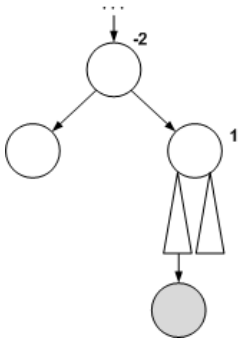
①LL(left higher):新结点在z的左孩子y的左子树上➔right rotation
②RR(right higher):新结点在z的右孩子y的右子树上➔left rotation
③LR(equal height):新结点在z的左孩子y的右子树上➔double rotation
④RL(equal height):新结点在z的右孩子y的左子树上➔double rotation
其中, z是最小不平衡子树的根结点, y是z的孩子结点.

④ Right Left Case

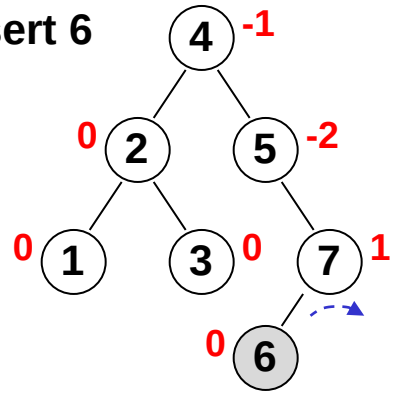
```
   z                                z                                x
  / \                              / \                              / \
T1   y   Right Rotate (y)       T1   x    Left Rotate(z)         z     y
   / \  - - - - - - - - ->       / \    - - - - - ->           / \   / \
  x   T4                        T2   y                       T1  T2 T3  T4
 / \                                / \
T2  T3                             T3  T4
```



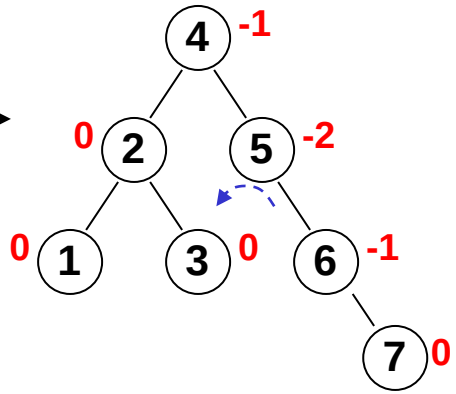
RR型: left rotation
5 as pivot



Insert 6



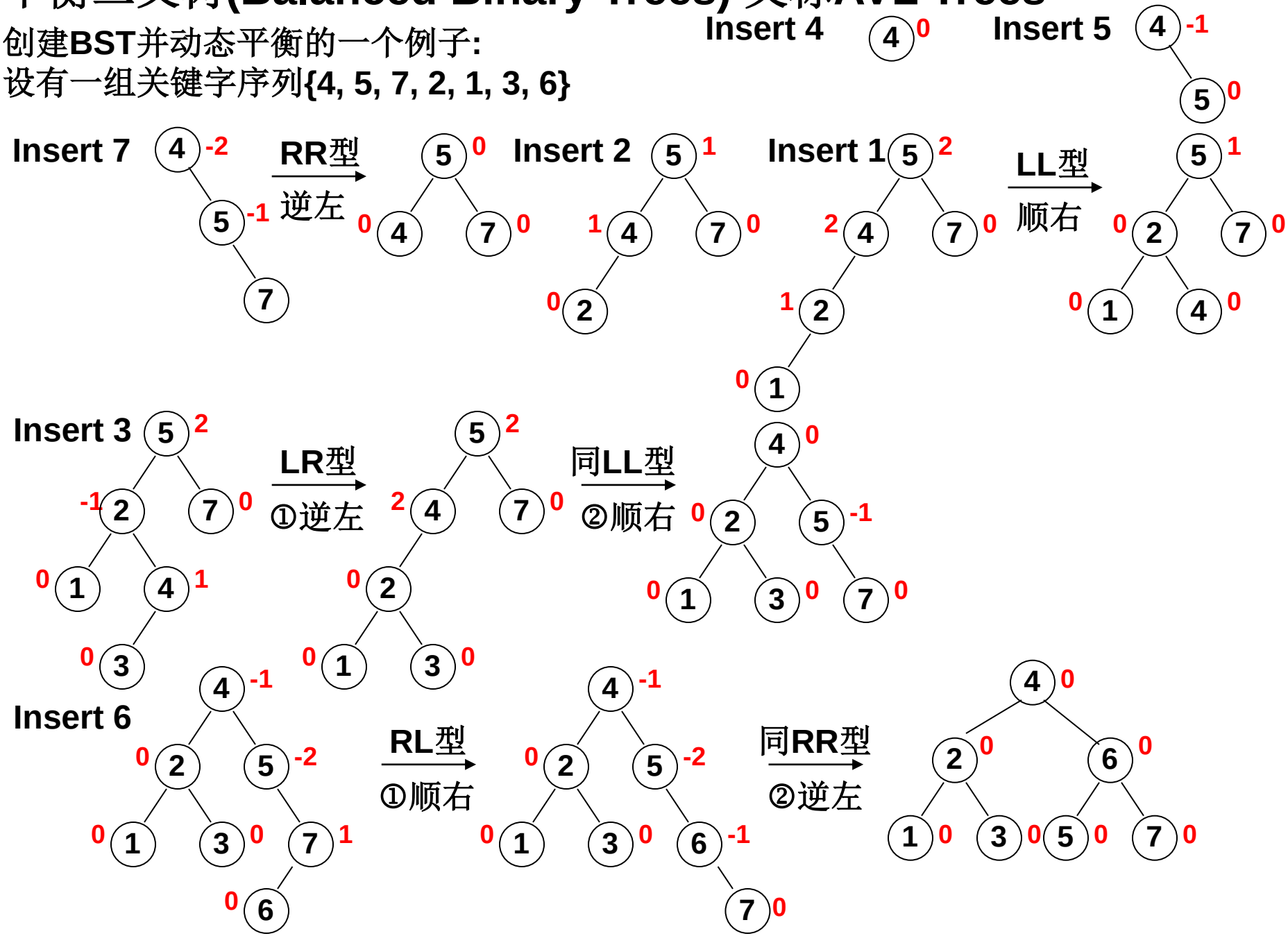
RL型: right rotation
7 as pivot



平衡二叉树(Balanced Binary Trees) 又称AVL Trees

创建BST并动态平衡的一个例子:

设有一组关键字序列{4, 5, 7, 2, 1, 3, 6}



平衡二叉树(Balanced Binary Trees) 又称AVL Trees

```
//Recursive function to insert key in subtree rooted with node and returns new root of subtree.
struct Node* insert(struct Node* node, int key) { //AVL二叉查找树动态平衡的算法
    //1.Perform the normal BST insertion
    if (node == NULL) return (newNode(key));
    if (key < node->key) node->left  = insert(node->left, key);
    else if (key > node->key) node->right = insert(node->right, key);
            else return node; // Equal keys are not allowed in BST
    //2.Update height of this ancestor node
    node->height = 1 + max(height(node->left), height(node->right));
    //3.Get the balance factor of this ancestor node to check whether this node became unbalanced
    int balance = getBalance(node);
    if (balance > 1 && key < node->left->key) return rightRotate(node); //Left Left Case
    if (balance < -1 && key > node->right->key) return leftRotate(node); //Right Right Case
    if (balance > 1 && key > node->left->key) {
        node->left =  leftRotate(node->left); return rightRotate(node); } //Left Right Case
    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right); return leftRotate(node); } //Right Left Case
    return node; //return the (unchanged) node pointer } //时间复杂度O(log2n)
```

```
struct Node {
    int key;
    struct Node *left, *right;
    int height; } //AVL tree node
```

```
int height(struct Node *N) {
    if (N == Null)  return 0;
    return N->height; }
```

```
struct Node* newNode(int key) {
    node = (struct Node*)malloc(sizeof(struct Node));
    node->key = key; node->left = node->right = Null;
    node->height = 1;  // new node is initially added at leaf
    return node; }
```

```
int getBalance(struct Node *N) {
    if (N == NULL) return 0;
    return height(N->left) - height(N->right); }
```

External Searching:B-Trees(B树)

To access { High-speed memory: microseconds
A Disk: milliseconds
A Floppy: seconds

Goal in external searching: To minimize the number of disk accesses

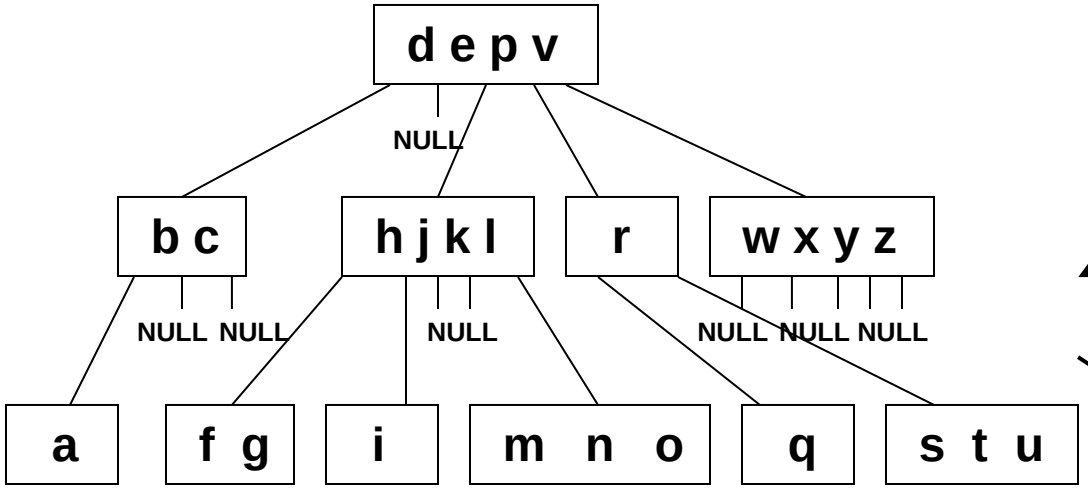
How?

To make a multiway (多路) decision concerning which block to access next

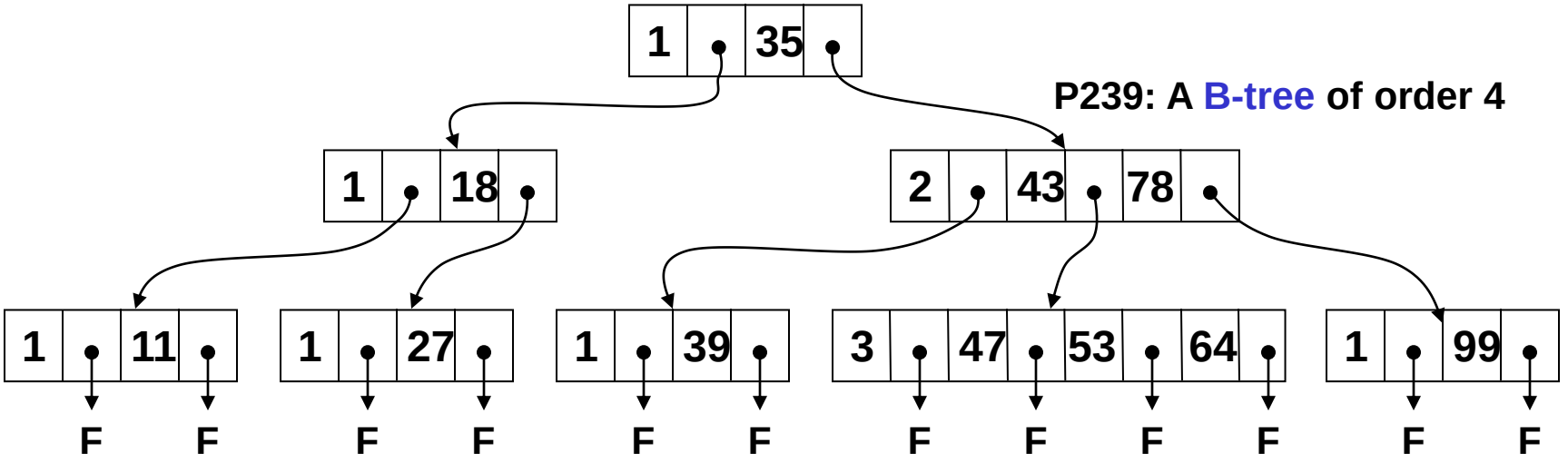
To make the height of the tree as small as possible

Balanced Multiway Trees

B-Trees(B树)⇨B+树



A 5-way search tree (26 letters)



P239: A B-tree of order 4

B-Trees(B树) P238

一棵m阶的B树, 或为空树, 或为满足下列条件的m叉树: 1) 树中每个结点至多有m棵子树; 2) 若根结点不是叶结点(即只包含一个结点的B树), 则根结点至少有两棵子树; 3) 除根结点、叶结点之外, 其它所有结点至少有$\lceil m/2 \rceil$棵子树; 4) 所有叶结点都在同一层, 且不包含任何关键字信息; 5) 有K个孩子的非叶结点恰好包含K-1个关键字, 且按关键字有序, 即

n	P_0	K_1	P_1	K_2	P_2	…	P_{n-1}	K_n	P_n

其中, K_i为关键字: $i \in [1, n]$ 且 $K_i < K_{i+1}$

P_i为指向包括在关键字(K_i , K_{i+1})之间的子树

- P_0指向$<K_1$的子树
- P_n指向$>K_n$的子树

n为关键字个数: $n \in [\lceil m/2 \rceil -1, m-1]$ 如m=3阶, 则$n \in [1, 2]$;

如m=4阶,则$n \in [1, 3]$; 如m=5阶,则$n \in [2, 4]$

B树的存储结构:

```
#define MAX  4  //maximum number of keys in node: MAX=m-1
#define MIN 2  //minimum number of keys in node: MIN=⌈m/2⌉-1
#define TreeNode struct treenode;
struct treenode { int count;  //除根结点外，the lower limit is MIN
TreeEntry entry[MAX+1];  //每个entry包含Key等数据域, 0#单元未用
TreeNode  *branch[MAX+1]; };
```

B-Tree Retrieval P240/A.9.13

基本思路 <
- 在B树中找结点--- SearchTree(相当于对AVL树的查找运算)
- 在结点中找关键字---SearchNode(相当于对顺序表的查找运算)

```
TreeNode *SearchTree(Key target, TreeNode *root, int *targetpos)
//pre: The B-tree pointed by root has been created.
//post: If the key target is present in the B-tree, then return value
//points to the node containing target in position targetpos.
//Otherwise the return value is NULL and targetpos is undefined.
{if (!root) return NULL;
 else if (SearchNode(target, root, targetpos)) return root;
       else return SearchTree(target,root->branch[*targetpos],targetpos); }

Boolean *SearchNode(Key target, TreeNode *current, int *pos)
//pre: Target is a key and current points to a node of the B-tree.
//post: Searches keys in a node for target, returns location pos of target
//or branch on which to continue search.
{if (LT(target, current->entry[1].key)) { //take the leftmost branch
    *pos = 0; return FALSE; }
 else {  //start a sequential search through the keys
    for (*pos = current->count;
          LT(target, current->entry[*pos].key) && pos>1; (*pos)-- ) ;
    return EQ(target, current->entry[*pos].key);  } }
```

B-Tree Retrieval

B树查找算法分析:

B树查找 ＜ 在B树中找结点---外部查找(∵需从存在外存的B树中读结点至RAM)
在结点中找关键字---内部查找(在RAM中顺序查找)

外部查找读磁盘次数≤B树的高度h, 每个结点内关键字比较个数<B树的阶数m.
∴ B树查找运算时间为O(h×m). ➔ 如何求得B树的高度h?

定理: 对任意一棵含有n个关键字的m阶B树, 其树高h至多为 $\log_{\lceil m/2 \rceil}((n+1)/2)+1$.
其中$\lceil m/2 \rceil$是除根结点外的每个非终端结点的最小度数　P240~241

证明: 根据B树定义知, 位于第1层的根结点至少含有2棵子树, 则第2层上至少有2个结点; 又知除根结点外的每个非终端结点至少有$\lceil m/2 \rceil$棵子树, 则第3层至少有$2\times\lceil m/2 \rceil$个结点; 依此类推, 第L+1层上至少有$2\times(\lceil m/2 \rceil)^{L-1}$个结点, 而第L+1层的结点为叶结点. 若m阶B树中含有n个关键字, 则叶结点数即为查找不成功的结点数为n+1, 由此有: $n+1 \geq 2\times(\lceil m/2 \rceil)^{L-1}$, 解得: $L \leq \log_{\lceil m/2 \rceil}((n+1)/2)+1$

这就是说, 在含有n个关键字的m阶B树中进行查找结点时, 从根结点到关键字所在结点的路径上涉及到的结点数不超过 $\log_{\lceil m/2 \rceil}((n+1)/2)+1 = O(\log_{\lceil m/2 \rceil} n)$.

⇨ n ≥ Σ(每个结点包含的最少关键字数×每层结点数总和)

$$= 1 + (\lceil m/2 \rceil - 1) \sum_{i=2}^{L} 2\times(\lceil m/2 \rceil)^{i-2} = 2\times(\lceil m/2 \rceil)^{L-1} - 1$$

B-Tree Retrieval

B树查找算法分析结果的讨论:

在含有n个关键字的m阶B树中进行查找结点时, 从根结点到目标关键字所在结点的路径上涉及到的结点数不超过 $\log_{\lceil m/2 \rceil}((n+1)/2)+1 = O(\log_{\lceil m/2 \rceil}n)$.

因为 $\log_{\lceil m/2 \rceil}n = \log_2 n / \log_2(\lceil m/2 \rceil)$ [即对数换底公式], 而$\log_2 n$是含n个结点的平衡二叉树(AVL)的高度, 由该式知它是m阶B树的高度的$\log_2(\lceil m/2 \rceil)$倍.
例如, 设m=1024, 则$\log_2(\lceil m/2 \rceil) = \log_2 512 = 9$, 若B树高为4, 则平衡二叉树高为36. 显然: n一定时, 若m越大, 则B树高度越小.

结论: ①当需对含大量记录的文件(通常存储在磁盘上)进行查找时, B树高度越小越好, 这时所需的m值也往往较大.

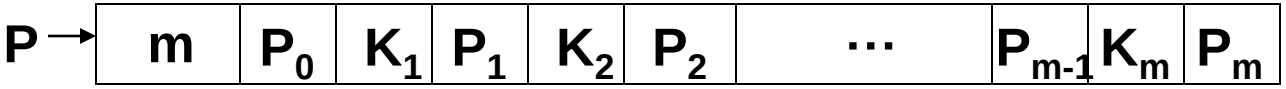
②当仅需在内存中对B树进行查找时, B树必须取较小的值.(B树查找时间= $O(h \times m) = O(m \times \log_{\lceil m/2 \rceil}n) = O(\log_2 n \times (m/\log_2(\lceil m/2 \rceil)))$, 因为 $m/\log_2(\lceil m/2 \rceil) > 1$, 故m较大时, B树查找时间$O(m \times \log_{\lceil m/2 \rceil}n)$比平衡二叉树上相应的查找时间$O(\log_2 n)$大得多. 通常取m=3, 此时B树中每个非终端结点可以有2或3个子树, 这种3阶B树便又称为2-3树)

Insertion Into a B-tree P241

思路: 对于叶结点处于第L+1层的B树, 则插入的关键字总在第L层上的某个结点中. 若该结点关键字数n满足$n\in[\lceil m/2\rceil-1, m-1)$, 则可直接完成插入操作; 否则若该结点处于n=m-1的临界状态, 则需作分裂(split)处理, 甚至会导致B树的高度增加一层.

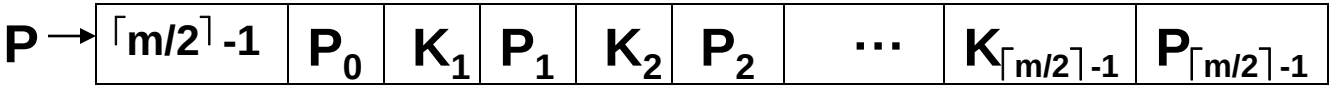
关键问题: 对n>m-1时, 结点如何实现“分裂”? P244

设*P结点中已有m-1个keys, 当插入一个新key后, 该结点中的信息为:

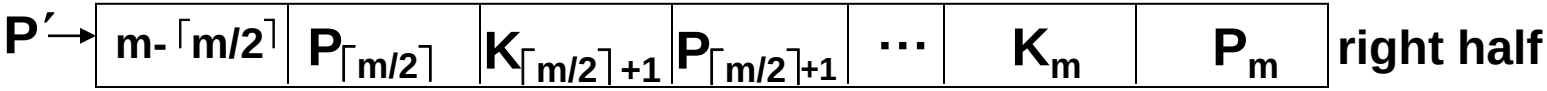


这时先将*P结点split为*P和*P′ (new node)两个结点, 其中

*P结点中的信息为:



*P′ 结点中的信息为:



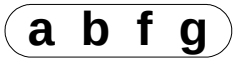
同时, 将关键字$K_{\lceil m/2\rceil}$(median key)和指针P′ 一起插入到*P结点的双亲结点中.

实例: ① P242-243 ② ➔

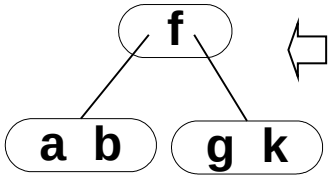
算法描述: P244/A.9.14, 该算法时间复杂性同B树的查找.

Insertion Into a B-tree

实例: 将如下有20个关键字的序列 { a, g, f, b, k, d, h, m, j, e, s, i, r, x, c, l, n, t, u, p } 插入至一棵初始状态为空的5阶B树中 -- B树的生长过程: 创建一棵B树就是一个不断插入新记录的过程.

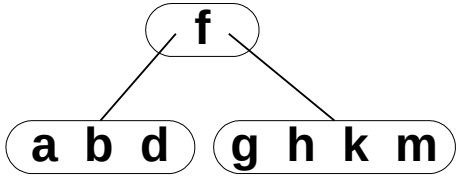


1. 插入a, g, f, b后

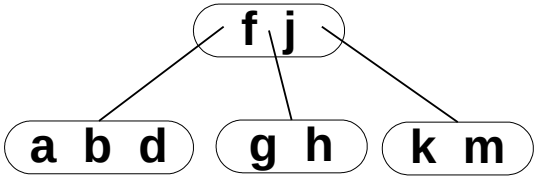


- Node splitting
- Upward propagation

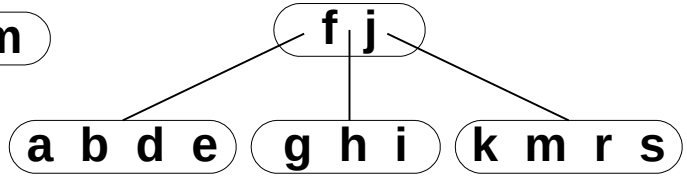
2. 插入k后



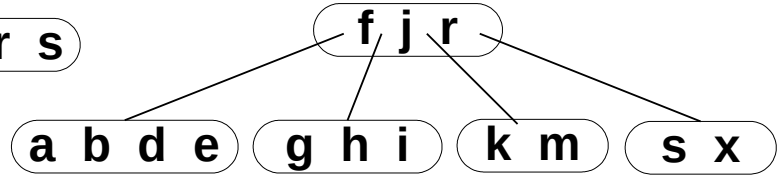
3. 插入d,h,m后



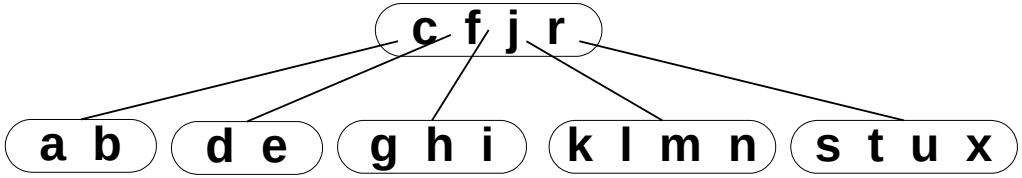
4. 插入j后



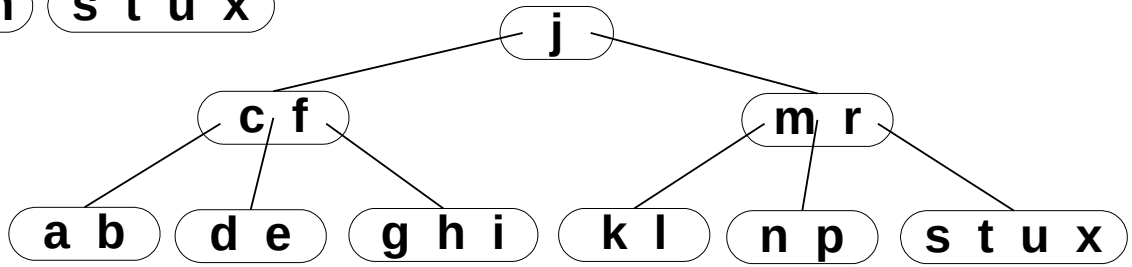
5. 插入e,s,i,r后



6. 插入x后



7. 插入c,l,n,t,u后



8. 插入p后

Deletion From a B-tree P244

思路: 若删除的关键字K_i不在第L层, 则先需将与它在B树中的前趋/后继关键字(它一定是K_i的左/右子树中最右/左下的结点中最后/前的一个关键字)的位置对换, 然后再删除该关键字. 同样, 若n满足$n \in (\lceil m/2 \rceil - 1, m-1]$, 则可直接删除$K_i$及相应指针$P_i$; 否则若该结点处于$n = \lceil m/2 \rceil - 1$的临界状态, 则需作合并(combine)处理, 甚至会导致B树的高度减少一层.

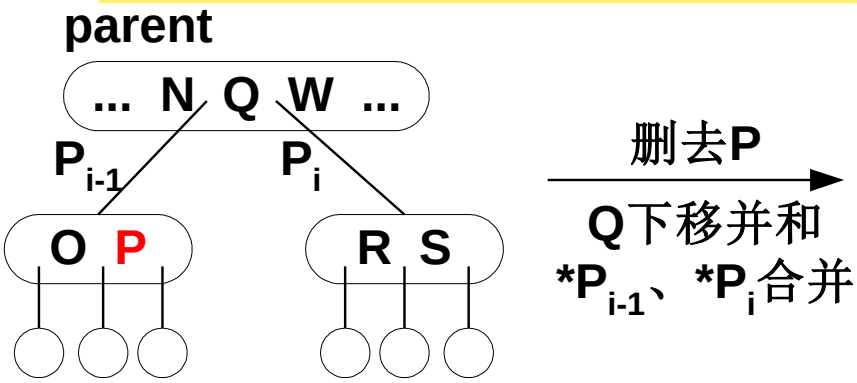
关键问题: 对$n < \lceil m/2 \rceil - 1$时, 结点如何实现"合并"? P244-245分两种情况.

①若被删关键字所在结点中的关键字数 $n = \lceil m/2 \rceil - 1$ 且与该结点相邻的左或右兄弟结点中的关键字数 $n > \lceil m/2 \rceil - 1$, 则需将其左/右兄弟结点中的最大/小的关键字上移至双亲结点中且将双亲结点中大于/小于并紧靠该上移关键字的关键字下移至被删关键字所在的结点中.

设为m=5的B-树

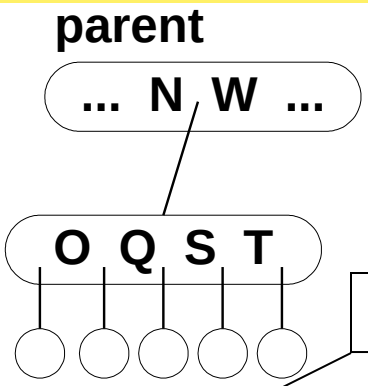
Deletion From a B-tree

②若被删关键字所在结点中的关键字数与其相邻的兄弟结点中的关键字数均为 $n = \lceil m/2 \rceil - 1$, 假设该结点有右兄弟且其右兄弟结点地址由双亲结点中的指针P_i所指向, 则在删去关键字后, 它所在结点中剩余的关键字和指针加上双亲结点中的关键字K_i一起合并到P_i所指向的兄弟结点中.

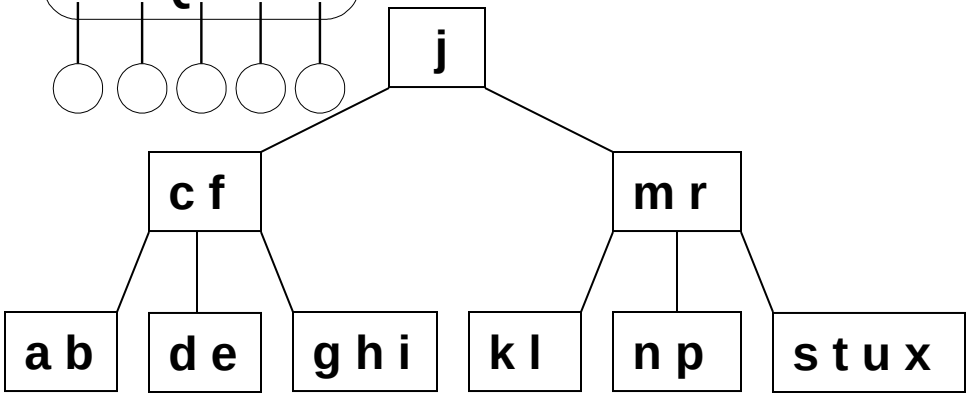


删去P

Q下移并和 $*P_{i-1}$、$*P_i$合并

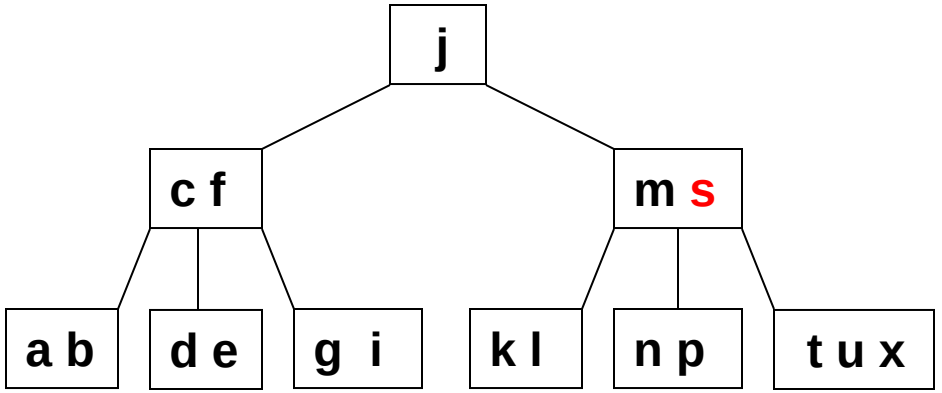


例: 设有一棵如右所示的5阶B树:

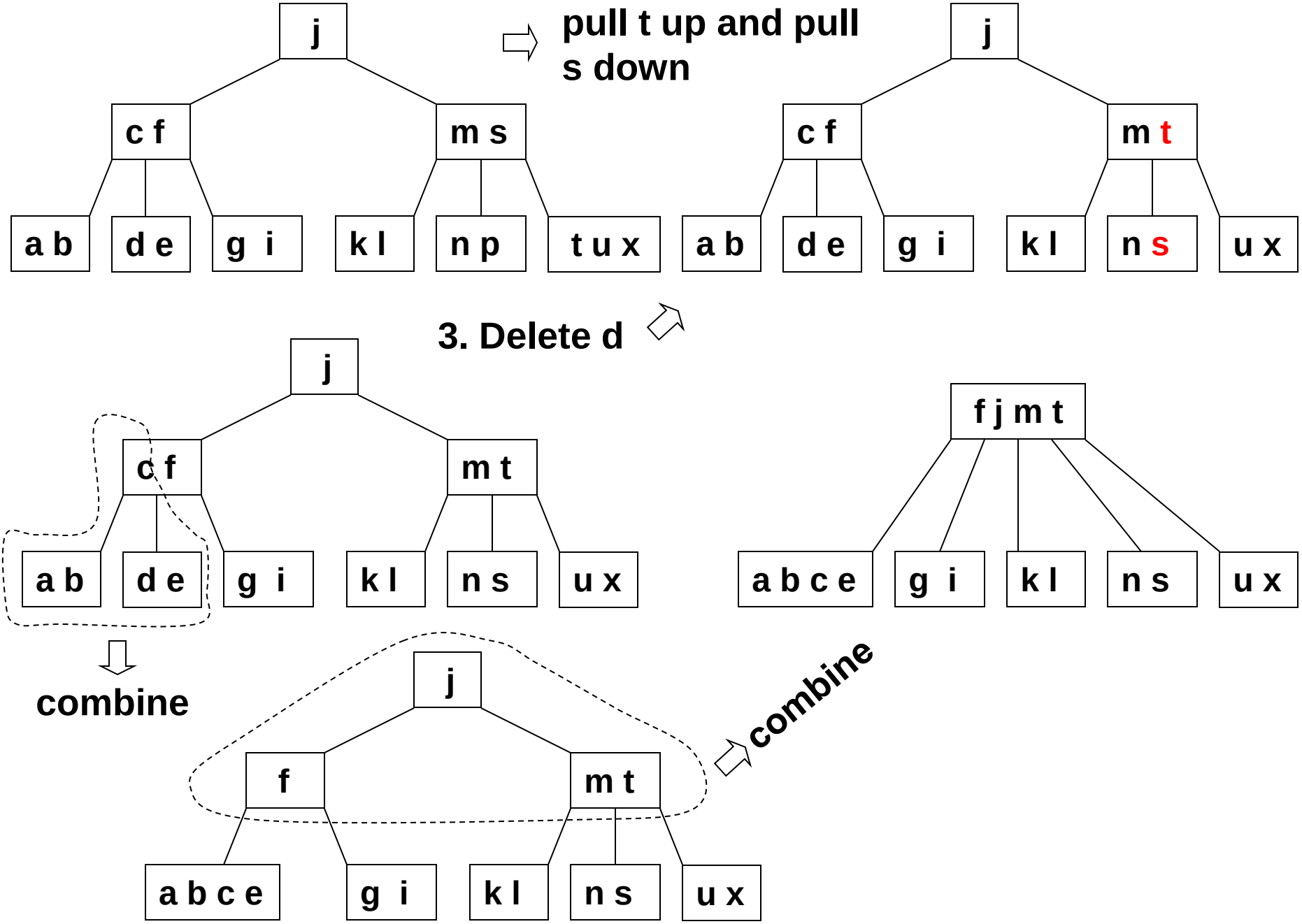


1. Delete h, r

⇨ promote s and delete from leaf



2. Delete p


j
c f
m s
a b
d e
g i
k l
n p
t u x
pull t up and pull s down
j
c f
m t
a b
d e
g i
k l
n s
u x
3. Delete d
j
c f
m t
a b
d e
g i
k l
n s
u x
f j m t
a b c e
g i
k l
n s
u x
combine
j
f
m t
a b c e
g i
k l
n s
u x
combine

Hash Table and Its Search P251

1. 背景

- 对线性表的顺序、二分查找以及对非线性表(二叉排序树、AVL树、B树)的查找, 其查找的时间开销依赖于查找过程中和Key值比较的次数;
- 不经比较能直接存取所需记录: MAP(关键字, 存储位置) -- Hash函数

2. 几个术语

哈希函数H/散列函数H为以记录的关键字值为自变量, 计算出对应的函数H(key), 该函数值称为哈希地址/散列地址. 设表长=n, 记HT[0..HashSize-1]为存储记录的地址空间, 且HashSize≥n, 称HT为长度是HashSize的哈希表.

⇨ ①将关键字值为key的记录存储到HT[H(key)], H(key)∈[0, HashSize-1];
②哈希函数作为映射并非一一对应的关系. 即不同的关键字值, 可能具有相同的哈希地址, 称之为"冲突"/"碰撞"(collision); 把这样的关键字称为synonym.

3. 哈希函数的构造

- 目标是地址均匀分布(uniform distribution)以免"冲突"
- 方法P253-256 ·truncation ·folding ·modular arithmetic •••
- 选择时需考虑的因素P256

4. 解决"冲突"和"溢出"的方法 ▪open addressing ▪chaining

5. Hash查找效率分析 Hash表中已装"满"了记录

- 哈希函数的构造

• **Truncation(截断法)**

如: 设**HashSize=1000**, 且**key**为**8**位整型值, 截其中的第**1**、**2**、**5**位, 那么对**key=62538194**则散列至**394**

特点: **a fast method, but it often fails to distribute the keys evenly through the Hash table**

• **Folding(折叠法)**

如: 设**HashSize=1000**, 且**key**为**8**位整型值, 按**3**、**3**、**2**位分组折叠(相加)后取模, 那么对**key=62538194**则散列至**(625+381+94)%1000=100**

特点: 利用了一个**key**中的所有信息, 相对截断法具有较好的分布性

• **modular arithmetic(模运算法)**

如: 设**HashSize=1000**, 且**key**为**8**位整型值, 取适当的模数(**modulus**: 一般用**HashSize**左右的一个质数以减少冲突, 如**modulus=997**或**1009**)进行模运算, 那么对**key=62538194**则散列至**62538194%997=372**

特点: 数学经验表明, 相对折叠法来说具有更好的分布性

```
//一个Hash function例子: determine the Hash value of key s
//Pre: s is a valid key type.
//Post: s has been hashed, returning a value between 0 and HashSize-1
int Hash(Key s)
{ unsigned h = 0;
  while (*s)   h += *S++;
  return h%HashSize; }
```

- **open addressing(开放地址法) for collision handling**

基本思想: 若在位置t(t=Hash(key))上发生冲突, 则从位置$t+d_i$开始“探测”(寻找)“下一个”空位置ProbePos. 该过程中将HT看作一个循环表:

ProbePos = (Hash(key) + d_i) % HashSize

- 若d_i=1, 2, 3, …, HashSize-1, 即d_i=i, 则称为线性探测(linear probing)再散列. 但该方法最易于产生“聚集”现象 [见下页例]
- 若$d_i=1^2, -1^2, 2^2, -2^2, 3^2, \ldots, k^2 (k \le HashSize/2)$, 即$d_i=\pm i^2$, 则称为二次探测(quadratic probing)再散列
- 若d_i=srand((unsigned int)(time(NULL)%HashSize)),则称为随机探测(random probing)再散列

The function time returns the number of seconds elapsed since 00:00:00 GMT, Jan. 1, 1970.

- 若d_i=Hash1(key),2×Hash1(key),…,(HashSize-1)×Hash1(key), 即d_i=i×Hash1(key), 因该方法采用了两个散列函数则称之为双重散列函数探测(double hashing probing)再散列

开发地址法解决冲突的一个例子 --- 以线性探测再散列法为例:

开放地址法的一些基本运算:

- **open addressing(开放地址法) for collision handling**

线性探测再散列法的一个例子:

已知一组(n=10)关键字序列{26, 36, 41, 38, 44, 15, 68, 12, 06, 51},
设HashSize=13, H(key)=key%HashSize, 并用线性探测法解决冲突.
构造这组关键字的哈希表/散列表HT[0..HashSize-1].

Key = { 26, 36, 41, 38, 44, 15, 68, 12, 06, 51 }
H(Key) = (0, 10, 2, 12, 5, 2, 3, 12, 6, 12)

根据散列函数 H(key)=key%13 分别计算出上述关键字序列的哈希地址:
(0 , 10 , 2 , 12 , 5 , 2 , 3 , 12 , 6 , 12). 据此建立哈希表:

	0	1	2	3	4	5	6	7	8	9	10	11	12
HT[0..12]	26	12	41	15	68	44	6	51			36		38
探测次数	1	3	1	2	2	1	1	9			1		1

P257 “聚集(clustering)”现象或称“堆积”现象: 在“冲突”处理过程中发生的多个哈希地址不同的元素争夺同一个后继哈希地址. 即若用线性探测法解决冲突, 当哈希表中 i, i+1, … , i+k的位置上已有元素时, 哈希地址为 i, i+1, … , i+k, i+k+1的元素都将插入在位置 i+k+1 上. 如上述的H(15)=2, H(68)=3, 15和68本不是同义词, 却出现了“聚集”.
一些其它的探测法可减少“聚集”现象.

- **open addressing(开放地址法) for collision handling**

开放地址法的一些基本运算(以线性探测为例, 且设所有哈希函数为Hash):

数据结构:

```
//declarations for a Hash table with open addressing
#define HashSize ???
typedef char *Key;
typedef struct item { Key key; } Entry;
typedef Entry HashTable[HashSize];
```

在HT中查找一个数据元素的操作:

```
int HashSearch(HashTable HT, Key target)
{ int probe;  //position currently probed in HT
  int increment = 1;  // increment used for linear probing
  probe = Hash(target);
  while (HT[probe].key != NULL &&  //Is the location empty?
          !EQ(target, HT[probe].key))  //Duplicate key present?
     {  probe = (probe + increment) % HashSize; }
  if (EQ(target, HT[probe].key)) return probe;
  else  return  –1;  }
```

- **open addressing(开放地址法) for collision handling**

  开放地址法的一些基本运算(以线性探测为例, 且设所有哈希函数为Hash) .
  在HT中插入一个数据元素的操作:

```
//insert an item using open addressing and linear probing P259A.9.17-18
//Pre: The Hash Table HT  has been created and is not full.
//Post: The item newitem has been inserted into HT.
void HashInsert(HashTable HT, Entry newitem)
{ int pc = 0;  //probe count to be sure that table is not full
  int probe;  //position currently probed in HT
  int increment = 1;  // increment used for linear probing
  probe = Hash(newitem.key);
  while (HT[probe].key != NULL &&  //Is the location empty?
          strcmp(newitem.key, HT[probe].key) &&  //Duplicate key present?
          pc <= HashSize/2) {  //Has overflow(clustering) occurred?
     pc++;  probe = (probe + increment) % HashSize;
  }  //Prepare increment for next iteration
  if (HT[probe].key == NULL) HT[probe] = newitem; //Insert the new entry.
  else if (strcmp(newitem.key, HT[probe].key) == 0)
           Error("The same key cannot appear twice in the Hash Table.");
        else Error("The HT is full.") or ReHash;  }
```

- **open addressing(开放地址法) for collision handling**

开放地址法的特点:

优: 思路清晰, 算法简单

缺: 无法处理“溢出”;

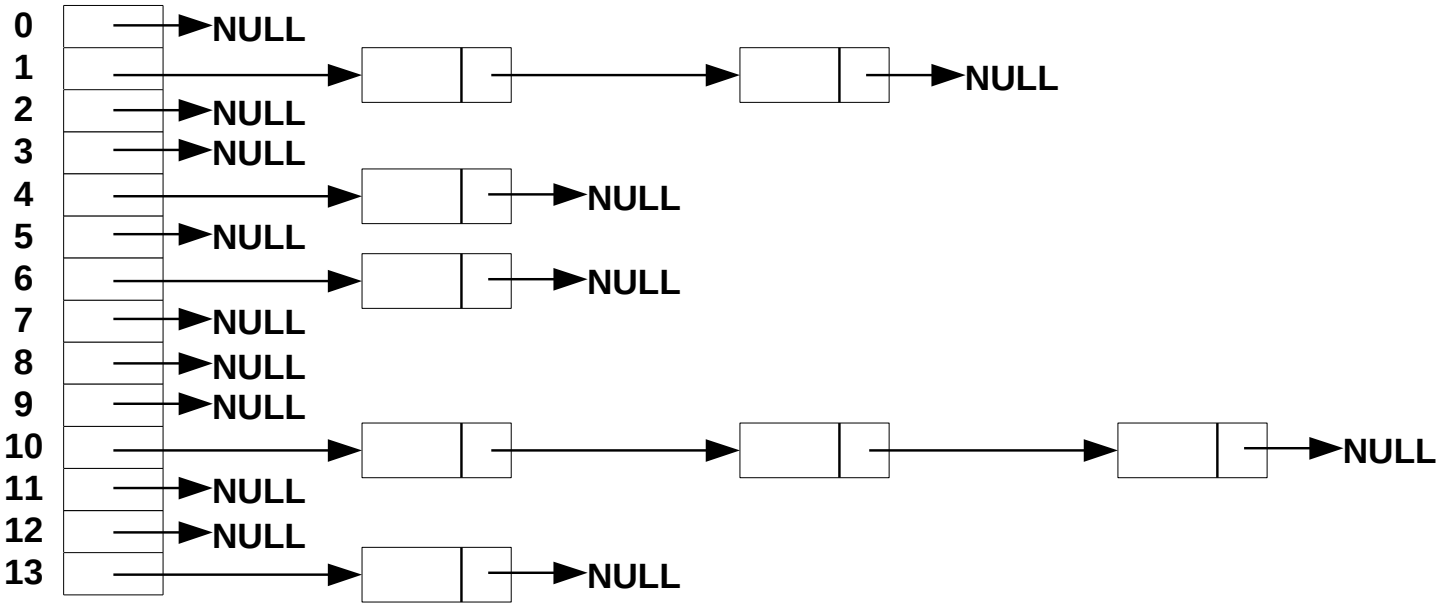
HT中的元素不易删除;

易于产生“聚集”现象.

- **separate chaining(链地址法) for collision handling**

基本思路: 将所有关键字为同义词的数据元素存储在同一线性链表中.

Figure: A chained hash table



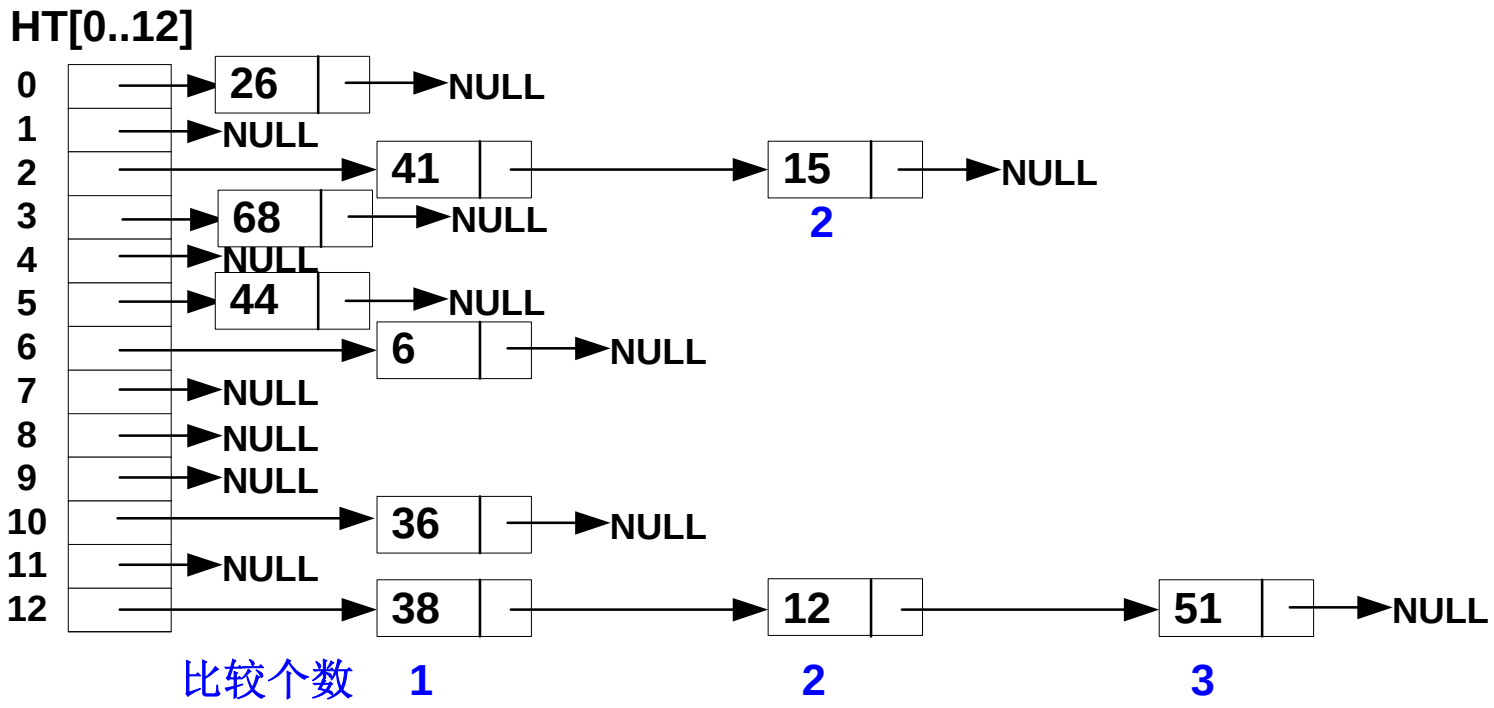

```
//declarations for a Hash table with chaining
#define HashSize ???
typedef char *Key;
typedef struct item { Key key; } Entry;
typedef struct node { Entry entry; struct node *next; } Node;
typedef Node *HashTable[HashSize];  //HashTable is an array of pointers
```

- **separate chaining(链地址法) for collision handling**

链地址法的一个例子:

已知一组(n=10)关键字序列{26, 36, 41, 38, 44, 15, 68, 12, 06, 51},
设HashSize=13, H(key)=key%HashSize, 并用链地址法解决冲突.
构造这组关键字的哈希表/散列表HT[0..HashSize-1].

根据散列函数 H(key)=key%13 分别计算出上述关键字序列的哈希地址:
(0, 10, 2, 12, 5, 2, 3, 12, 6, 12). 据此建立哈希链表:



- **separate chaining(链地址法) for collision handling**

链地址法的运算: (参见Ex, 其它建立、查找、插入运算)

链地址法的特点:

优: save space (for large records);
collision resolution;
no overflow;
easy deletion;

Disadvantage: extra use of space.

- 哈希表操作算法效率分析

Hash表的插入和删除运算的时间均取决于查找时间, 故只分析查找算法的时间复杂性. 从对哈希表的查找过程可见: **P260**

①虽然哈希表在关键字与元素的存储位置之间建立了直接映象, 理想情况下无须关键字的比较就能找到待查元素, 但由于“冲突”的产生, 使得哈希表的查找过程仍然是一个给定值和关键字比较的过程. 即仍需以**ASL**来计量哈希表的查找效率.

假设查找每个元素的概率相同, 则线性探测法和链地址法查找成功的平均查找长度分别为:

$$ASL_{succ} = (1\times6 + 2\times2 + 3\times1 + 9\times1) / 10 = 2.2$$

$$ASL_{succ} = (1\times7 + 2\times2 + 3\times1) / 10 = 1.4$$

其中, **i×j** 中的**i**表示比较个数, **j**表示比较**i**个关键字的元素的个数

假设查找每个元素的概率相同, 则顺序查找和二分查找查找成功的平均查找长度分别为:

$$ASL_{succ} = (10 + 1) / 2 = 5.5$$

$$ASL_{succ} = (1\times1 + 2\times2 + 3\times4 + 4\times3) / 10 = 2.9$$

结论: 哈希表的查找效率优于顺序查找和二分查找.

- 哈希表操作算法效率分析

Hash表的插入和删除运算的时间均取决于查找时间, 故只分析查找算法的时间复杂性. 从对哈希表的查找过程可见:

②查找过程需和给定值进行比较的关键字个数取决于三个因素: **Hash**函数、处理“冲突”的方法、哈希表的装填因子**(Load Factor).**

哈希函数的“好坏”直接影响出现“冲突”的频度, 但一般说来哈希函数都能达到“均匀”的目标.

结论: 一般不考虑哈希函数对**ASL**的影响.

同一哈希函数、不同的处理“冲突”的方法, 其**ASL**不同. 如上页所示.

对处理“冲突”方法相同的哈希表, 其**ASL**依赖于装填因子.

装填因子$\alpha = \dfrac{\text{已装入记录数}}{\text{HT长度HashSize}}$ ⇨ α越小, 发生冲突的可能性越小; α越大, 发生冲突的可能性越大

装填因子α过小, 空间浪费就越多.

开放地址法(线性探测): $ASL_{succ} = \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$ $ASL_{unsucc} = \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$

链地址法: $ASL_{succ} = 1 + \frac{\alpha}{2}$ $ASL_{unsucc} = \alpha + e^{-\alpha}$

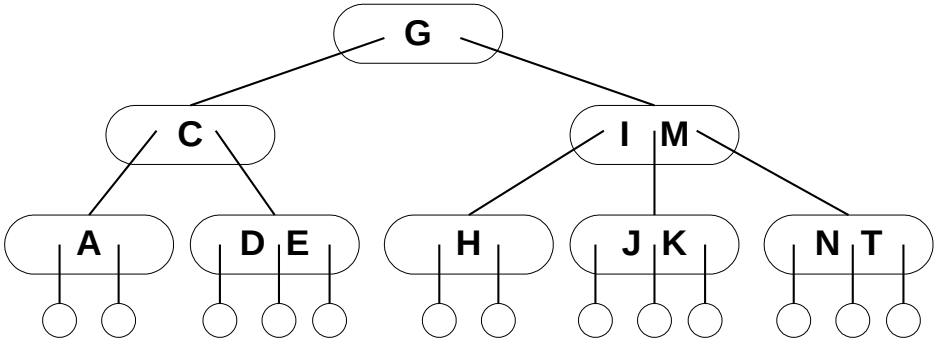
- **Exercise (After class)**

1. 将算法**9.2**改成递归描述, 设头函数定义为:
int BinarySearch(List list, KeyType target, int low, int high) { … }

2.设有一棵**5**阶**B**树, 若该树共**3**层即树的高度为**3(**假设叶结点所在层不计在其内**),** 则该**B**树中全部结点所包含关键字个数的最小值是多少**?**

3. Define *preorder* traversal of a B-tree recursively to mean visiting all the entries in the root node first, then traversing all the subtrees, from left to right, in preorder. Write a function that will traverse a B-tree in preorder.

4. 下图是一棵**3**阶**B**树, 依次画出插入关键字**B, L, P, Q, R**后的**3**阶**B**树**.**



5. 含有**8**个关键字的**2-3**树中最多、最少各有几个结点**?** 分别画出其**B**树的树形**.**

6. 设**HashSize=13,** 将下列**keys**映射到哈希表中**:10,100,32,45,58,128,3,29,200, 400,0.** 假设①**H(k)=key%HashSize** ②解决**collision**的方法为 **linear probing.**

7. 写出从哈希表中删除一个记录的算法, 设所用哈希函数为**H,** 解决**collision**的方法为**chaining: Entry *DeleteTable(HashTable HT, Key target) { … }**