数据结构篇

Hash方法与Java中的几种数据结构

哈希

Hash，一般翻译做“散列”，也有直接音译为“哈希”的，就是把任意长度的输入，通过散列算法，变换成固定长度的输出，该输出就是散列值。这种转换是一种压缩映射，也就是，散列值的空间通常远小于输入的空间，不同的输入可能会散列成相同的输出，所以不可能从散列值来唯一的确定输入值。简单的说就是一种将任意长度的消息压缩到某一固定长度的消息摘要的函数。

所有散列函数都有如下一个基本特性：**根据同一散列函数计算出的散列值如果不同，那么输入值肯定也不同。但是，根据同一散列函数计算出的散列值如果相同，输入值不一定相同。**

两个不同的输入值，根据同一散列函数计算出的散列值相同的现象叫做碰撞。

常用的Hash方法:

> 直接定址法：直接以关键字k或者k加上某个常数（k+c）作为哈希地址。
>
> 数字分析法：提取关键字中取值比较均匀的数字作为哈希地址。
>
> 除留余数法：用关键字k除以某个不大于哈希表长度m的数p，将所得余数作为哈希表地址。
>
> 分段叠加法：按照哈希表地址位数将关键字分成位数相等的几部分，其中最后一部分可以比较短。然后将这几部分相加，舍弃最高进位后的结果就是该关键字的哈希地址。
>
> 平方取中法：如果关键字各个部分分布都不均匀的话，可以先求出它的平方值，然后按照需求取中间的几位作为哈希地址。
>
> 伪随机数法：采用一个伪随机数当作哈希函数。

衡量一个哈希函数的好坏的重要指标就是发生碰撞的概率以及发生碰撞的解决方案。

- 开放定址法：
  - 开放定址法就是一旦发生了冲突，就去寻找下一个空的散列地址，只要散列表足够大，空的散列地址总能找到，并将记录存入。
- 链地址法
  - 将哈希表的每个单元作为链表的头结点，所有哈希地址为i的元素构成一个同义词链表。即发生冲突时就把该关键字链在以该单元为头结点的链表的尾部。
- 再哈希法
  - 当哈希地址发生冲突用其他的函数计算另一个哈希函数地址，直到冲突不再产生为止。
- 建立公共溢出区
  - 将哈希表分为基本表和溢出表两部分，发生冲突的元素都放入溢出表中。

HashMap的数据结构

JDK1.7

由两个方法`int hash(Object k)`和`int indexFor(int h, int length)`来实现。

```
final int hash(Object k) {
    int h = hashSeed;
    if (0 != h && k instanceof String) {
        return sun.misc.Hashing.stringHash32((String) k);
    }

    h ^= k.hashCode();
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}

static int indexFor(int h, int length) {
    return h & (length-1);
}
```

Java之所有使用位运算(&)来代替取模运算(%)，最主要的考虑就是效率。**位运算(&)效率要比代替取模运算(%)高很多，主要原因是位运算直接对内存数据进行操作，不需要转成十进制，因此处理速度非常快。**

key的hashCode进行扰动计算，防止不同hashCode的高位不同但低位相同导致的hash冲突。简单点说，就是为了把高位的特征和低位的特征组合起来，降低哈希冲突的概率，也就是说，尽量做到任何一位的变化都能对最终得到的结果产生影响。

HashTable

JDK1.7

Java 7中HashTable的hash方法的实现。

```
private int hash(Object k) {
    // hashSeed will be zero if alternative hashing is disabled.
    return hashSeed ^ k.hashCode();
}
```

HashTable默认的初始大小为11，之后每次扩充为原来的2n+1。

也就是说，HashTable的链表数组的默认大小是一个素数、奇数。之后的每次扩充结果也都是奇数。

由于HashTable会尽量使用素数、奇数作为容量的大小。当哈希表的大小为素数时，简单的取模哈希的结果会更加均匀。

ConcurrentHashMap

JDK1.7

```
private int hash(Object k) {
    int h = hashSeed;

    if ((0 != h) && (k instanceof String)) {
        return sun.misc.Hashing.stringHash32((String) k);
    }

    h ^= k.hashCode();

    // Spread bits to regularize both segment and index locations,
    // using variant of single-word Wang/Jenkins hash.
    h += (h <<  15) ^ 0xffffcd7d;
    h ^= (h >>> 10);
    h += (h <<   3);
    h ^= (h >>>  6);
    h += (h <<   2) + (h << 14);
    return h ^ (h >>> 16);
}

int j = (hash >>> segmentShift) & segmentMask;
```

上面这段关于ConcurrentHashMap的hash实现其实和HashMap如出一辙。都是通过位运算代替取模，然后再对hashcode进行扰动。区别在于，ConcurrentHashMap 使用了一种变种的 Wang/Jenkins 哈希算法，其主要母的也是为了把高位和低位组合在一起，避免发生冲突。至于为啥不和HashMap采用同样的算法进行扰动，我猜这只是程序员自由意志的选择吧。至少我目前没有办法证明哪个更优。

排序算法

插入排序

实现代码如下：

```
void insertSort(vector<int> &nums)
{
    if(nums.size()==0) return;
    int tmp;
    for(int i=0;i<nums.size();i++)
    {
        int j=i;
        while(j>0&&nums[j]<nums[j-1])
        {
            tmps=nums[j-1];
            nums[j-1]=nums[j];
            nums[j]=tmp;
            j--;
        }
    }
}
```

插入排序的最坏时间复杂度是O(n^2),最好是O(n),最好情况是排好序的，这是个稳定的算法。 空间复杂度是O(1)

选择排序

实现代码如下：

```
void selectSort(vector<int> &nums)
{
    if(nums.size()==0) return;
    int min;
    int tmp;
    for(int i=0;i<nums.size();i++)
    {
        min=i;
        for(int j=i+1;j<nums.size();j++)
        {
            if(nums[j]<nums[min])
                min=j;
        }
        tmp=nums[min];
        nums[min]=nums[i];
        nums[i]=tmp;
    }
}
```

最坏最好的时间复杂度都是O(n^2),空间复杂读是O(1)，选择排序算法是不稳定的。

冒泡排序

实现代码：

```
void dubbleSort(vector<int> &nums)
{
    if(nums.size()==0) return;
    int tmp=0;
    for(int i=0;i<nums.size();i++)
    {
        for(int j=i;j>0;j--)
        {
            if(nums[j]<nums[j-1])
            {
                tmp=nums[j-1];
                nums[j-1]=nums[j];
                nums[j]=tmp;
            }
        }
    }
```

```
}
```

最坏是O(n^2),最好也是O(n^2),这个排序算法是稳定的。

快速排序

实现代码：

```
void quickSort(vector<int> &nums,int start,int end)
{
    if(nums.size()==0) return;
    if(start>=end) return;
    int low=start;
    int high=end;
    int t=nums[low];
    while(low<high)
    {
        while(low<high&&nums[high]>=t)
        {
            high--;
        }
        nums[low]=nums[high];
        while(low<high&&nums[low]<=t)
        {
            low++;
        }
        nums[high]=nums[low];
    }
    nums[low]=t;
    quickSort(nums,start,low-1);
    quickSort(nums,low+1,end);
}
```

最好时间复杂度是O(NlogN),最坏算法是O(n^2),是一种不稳定的排序算法。

归并排序

```
void mergeSort(vector<int> &nums,int tmp[],int left,int right)
{
    int mid=(left+right)/2;
    //如果数组只有一个元素，那么就结束分割
    if(left==right) return;
    //一步一步划分数组，直到只有一个元素
    mergeSort(nums,tmp,left,mid);
    mergeSort(nums,tmp,mid+1,right);
    //将子数组拷贝到临时数组
```

```
    for(int i=left;i<=right;i++)
    {
        tmp[i]=nums[i];
    }
    int i1=left,i2=mid+1;
    for(int i=left;i<=right;i++)
    {
        //左边的数组耗尽
        if(i1==mid+1)
        {
            nums[i]=tmp[i2++];
        }else if(i2>right) //右边的数组耗尽
        {
            nums[i]=tmp[i1++];
        }
        else if(tmp[i1]>tmp[i2])
        {
            nums[i]=tmp[i2++];
        }
        else
        {
            nums[i]=tmp[i1++];
        }
    }
}
```

归并排序的时间复杂度，无论是最好，最差，还是平均都是O(nLogN),空间复杂度是O(n),是一种不稳定的排序算法。

希尔排序

基于插入排序，但并不是稳定的

实现代码：

```
void shellInsertSort(int a[],int length,int incr)
{
    for(int i=incr;i<length;i+=incr)
    {
        int j=i;
        while(j-incr>=0&&a[j-incr]>a[j])
        {
            int tmp=a[j-incr];
            a[j-incr]=a[j];
            a[j]=tmp;
            j-=incr;
        }
    }
}
```

```
void shellSort(int a[],length)
{
    for(int incr=length/2;incr>0;incr/=2)
    {
        shellinsert(a,length,incr);
    }
}
```

堆排序

堆，是一颗完全二叉树。

实现代码如下

```
void shiftdown(int a[],int pos,length)
{
    //如果是非叶子节点，那就遍历向下
    while(!isLeaf(a[pos]))
    {
        int left=2*pos;
        int right=2*pos+1;
        //右孩子是否在这个树上
        if(right<length)
        {
            left=a[left]>a[right]?left:right;
        }
        //是否需要调整堆
        if(a[pos]>a[left]) return;
        //调整堆
        swap(a,pos,left);
        pos=left;
    }
}
void  buildHeap(int a[],int length)
{
    for(int i=length/2-1;i>=0;i--)
    {
        shiftdown(a,i,length);
    }
}
```

堆排序的最佳、最差、平均的时间复杂度都是 O(NlogN)

堆的删除：过程是将最高的元素和最后的元素交换，然后调整堆。

总结

排序方法	平均时间	最好时间	最坏时间
桶排序(不稳定)	O(n)	O(n)	O(n)
基数排序(稳定)	O(n)	O(n)	O(n)
归并排序(稳定)	O(nlogn)	O(nlogn)	O(nlogn)
快速排序(不稳定)	O(nlogn)	O(nlogn)	O(n^2)
堆排序(不稳定)	O(nlogn)	O(nlogn)	O(nlogn)
希尔排序(不稳定)	O(n^1.25)		
冒泡排序(稳定)	O(n^2)	O(n^2)	O(n^2)
选择排序(不稳定)	O(n^2)	O(n^2)	O(n^2)
直接插入排序(稳定)	O(n^2)	O(n)	O(n^2)

树

树的周游

前序遍历

```
void preOrder(TreeNode* root,vector<int> &order)
{
    if(root==NULL)
    {
        return;
    }
    order.push_back(root->val);
    preOrder(root->left,order);
    preOrder(root->right,order);
}
```

中序遍历

```
void inOrder(TreeNOde* root,vector<int> &order)
{
    if(root==NULL) return;
    inOrder(root->left,order);
    order.push_back(root->val);
    inOrder(root->right,order);
}
```

后序遍历

```
void afterOrder(TreeNode* root,vector<int> &order)
{
    if(root==NULL) return;
    afterOrder(root->left,order);
    afterOrder(root->right,order);
    order.push_back(root->val);
}
```

层次遍历

```
vector<vector<int>> levelOrder(TreeNode * root) {
        // write your code here
        vector<vector<int>> result;
        if(root==NULL) return result;
        queue<TreeNode*> nodes;
        nodes.push(root);
        int i=0;
        vector<int> tmp;
        while(nodes.size())
        {
            i=nodes.size();
            int j=0;
            while(j<i)
            {
                TreeNode* t=nodes.front();
                tmp.push_back(t->val);
                j++;
                nodes.pop();
                if(t->left)
                    nodes.push(t->left);
                if(t->right)
                    nodes.push(t->right);
            }
            result.push_back(tmp);
            tmp.clear();
        }
        return result;
    }
```

图

图可以用G=(V,E)来表示，每个图都包括一个顶点集合V和一个边集合E，其中E中的每变都是V中某一对丁点的链接。顶点总数记为|V| ,边的总数记为|E|,|E|的取值范围是0到ø（|E|^2）。边数比较少的图成为稀疏图，边数较多的图成为密集图，包含所有边的图称为完全图。

图的周游

基于图的拓扑结构，以特定顺序依次访问图中各个顶点是很有用的。这称为图的周游，从概念上讲与树的周游类似。

深度优先搜索

第一种系统的图周游方式称为深度优先搜索。在搜索的过程中，每当访问某个定点v后，DFS就会递归地访问它所有未被访问的相邻顶点。或者，先访问顶点v，把所有与v相关联的边存入栈中，弹出栈顶元素，栈顶元素代表的边所关联的另一个顶点就是要访问的下一个元素，对该元素重复对v的操作；依次类推直到栈中所有元素都被处理完毕。这个结果是沿着图的某一分支搜索直到末端，然后回溯，沿着另一分支搜索，依次类推。

```
void DFS(Graph* G, int v) { // Depth first search
  PreVisit(G, v);           // Take appropriate action
  G->setMark(v, VISITED);
  for (int w=G->first(v);w<G->n();w = G->next(v,w))
    if (G->getMark(w) == UNVISITED)
      DFS(G, w);
  PostVisit(G, v);          // Take appropriate action
}
```

深度优先遍历的过程图

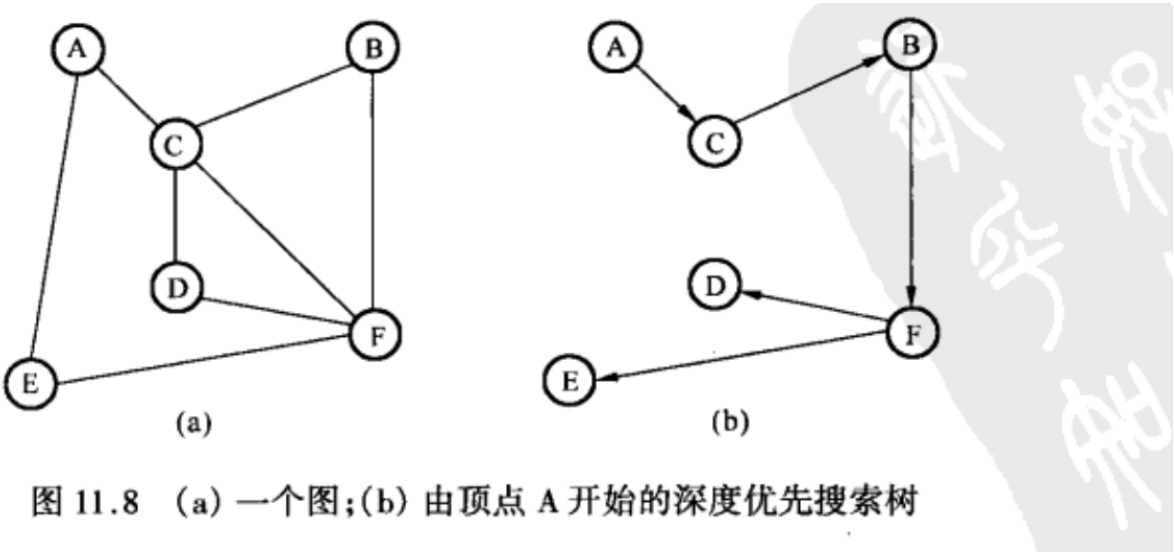
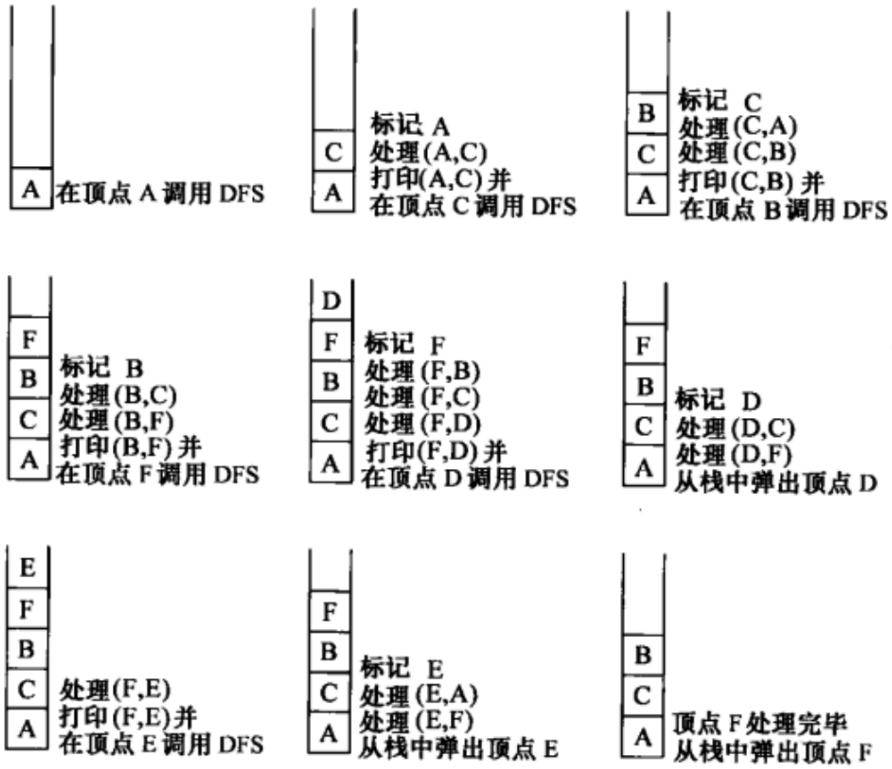


图 11.8 （a）一个图；(b) 由顶点 A 开始的深度优先搜索树

在一个有向图中，DFS对某条边处理一次。在无向图，DFS从两个方向处理每条边。每个顶点都必须都被访问，而且只能访问一次，因此总代价是ø(|V|+|E)



广度优先遍历

广度优先遍历也称为BFS，BFS在进一步深入访问其他顶点之前，检查起点的所有领节点，除了用队列替代递归栈以外，BFS的实现与DFS相似。如果图是一个树结构，而且起点为书的根节点，BFS将由顶至底逐层对各个节点进行访问(树的层次周游)。

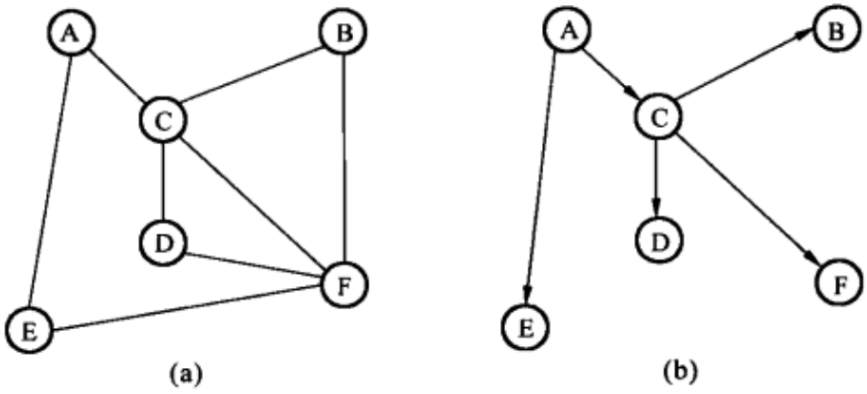
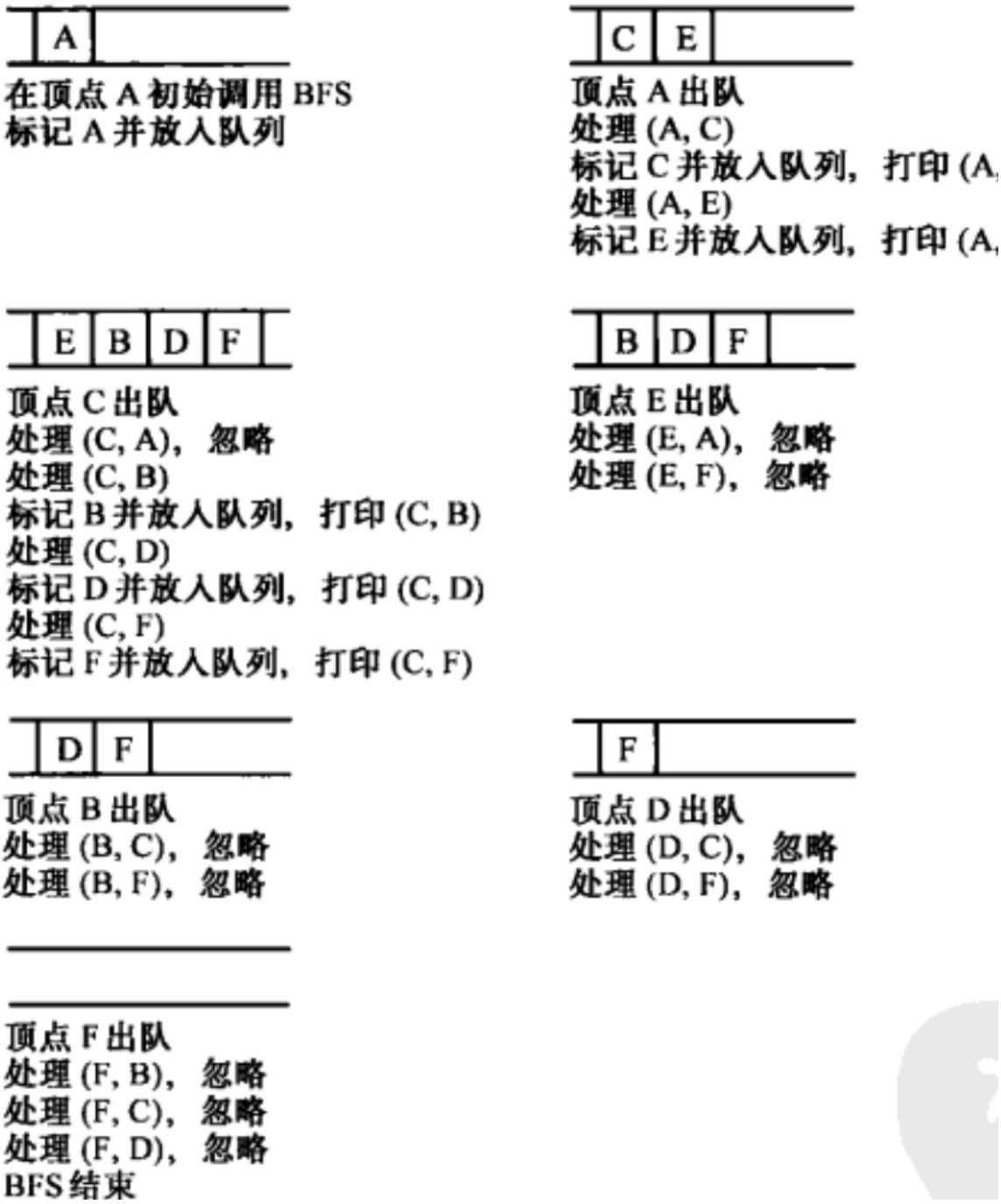


图 11.11 (a) 一个图;(b) 从顶点 A 开始的广度优先搜索树

BFS过程如下:



拓扑排序

如果我需要为一组人物安排进度，只有一项任务的先觉条件任务完成后才可以开始这项任务。我们希望以某种线性顺序组织这些任务，以便在能够满足先决条件的情况逐个完成各项任务。可以用一个有向无环图来为该问题建模。

将一个DAG中所有顶点在不违反先决条件规定的基础排成线性序列的过程称为拓扑排序。可以通过图进行上深度优先搜索来寻找拓扑序列。当访问某个顶点时，不对这个顶点进行任何的处理。当递归返回这个顶点时，函数PostVisit打印这个顶点。这将产生一个逆序拓扑序列。序列从那个顶点开始并不重要，只要所有顶点最终都被访问到。下面是基于DFS算法的实现:

```
void topsort(Graph* G) {       // Topological sort:recursive
  int i;
  for (i=0; i<G->n(); i++)  // Initialize Mark array
    G->setMark(i, UNVISITED);
  for (i=0; i<G->n(); i++)  // Process all vertices
    if (G->getMark(i) == UNVISITED)
      tophelp(G, i);              // Call recursive helper function
}

void tophelp(Graph* G, int v)   { // Process vertex v
  G->setMark(v, VISITED);
  for (int w=G->first(v); w<G->n(); w = G->next(v,w))
    if (G->getMark(w) == UNVISITED)
      tophelp(G, w);
  printout(v);                    // PostVisit for Vertex v
}
```

也可以使用队列代替递归来实现拓扑排序。做法如下:首先访问所有的边，计算指向每个顶点的边数，也就是计算每个顶点的先决条件。将所有没有先决条件的顶点放入队列，然后开始处理队列。当从队列中删除一个顶点，就把它打印出来，同时将其所有相领顶点的先决条件减1.当某个相领顶点的计数为0，就将其放入队列。如果还有顶点未被打印，而队列是空。

```
// Topological sort: Queue
void topsort(Graph* G, Queue<int>* Q) {
  int Count[G->n()];
  int v, w;
  for (v=0; v<G->n(); v++) Count[v] = 0; // Initialize
  for (v=0; v<G->n(); v++)    // Process every edge
    for (w=G->first(v); w<G->n(); w = G->next(v,w))
      Count[w]++;   // Add to v2's prereq count
  for (v=0; v<G->n(); v++)    // Initialize Queue
    if (Count[v] == 0)        // Vertex has no prerequisites
      Q->enqueue(v);
  while (Q->length() != 0) { // Process the vertices
    Q->dequeue(v);
    printout(v);              // PreVisit for Vertex V
    for (w=G->first(v); w<G->n(); w = G->next(v,w)) {
      Count[w]--;     // One less prerequisite
      if (Count[w] == 0)      // This vertex is now free
        Q->enqueue(w);
    }
  }
}
```

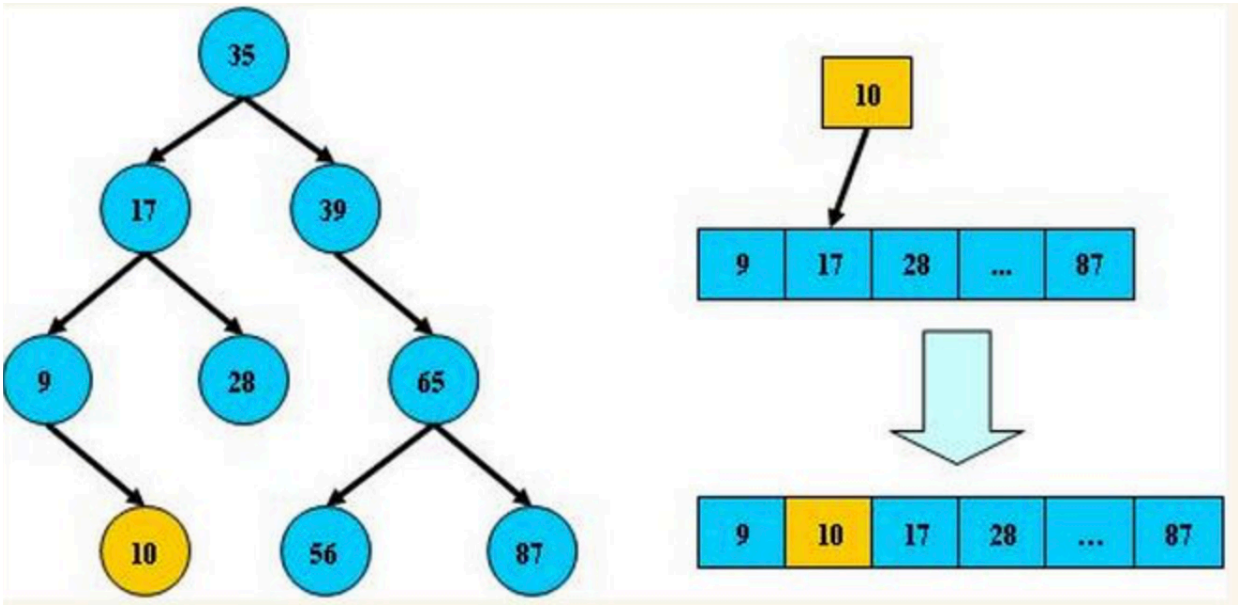
高级数据结构

AVL树

AVL树是基于二分法的策略提高数据的查找速度的二叉树的数据结构；

平衡二叉树是采用二分法思维把数据按规则组装成一个树形结构的数据，用这个树形结构的数据减少无关数据的检索，大大的提升了数据检索的速度；平衡二叉树的数据结构组装过程有以下规则：

非叶子节点只能允许最多两个子节点存在，每一个非叶子节点数据分布规则为左边的子节点小当前节点的值，右边的子节点大于当前节点的值。



总结平AVL树特点：

（1）非叶子节点最多拥有两个子节点；

（2）非叶子节值大于左边子节点、小于右边子节点；

（3）树的左右两边的层级数相差不会大于1;

（4）没有值相等重复的节点;

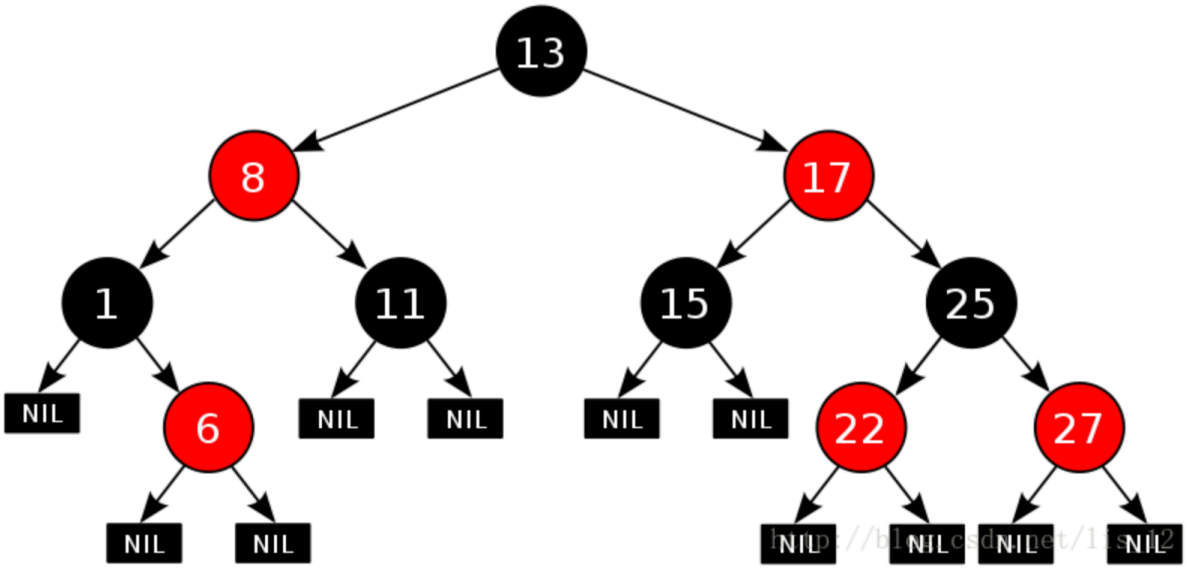
红黑树

红黑树是一棵二叉查找树，但它在二叉查找树的基础上增加了着色和相关的性质使得红黑树相对平衡，从而保证了红黑树的查找、插入、删除的时间复杂度最坏为O(log n)。

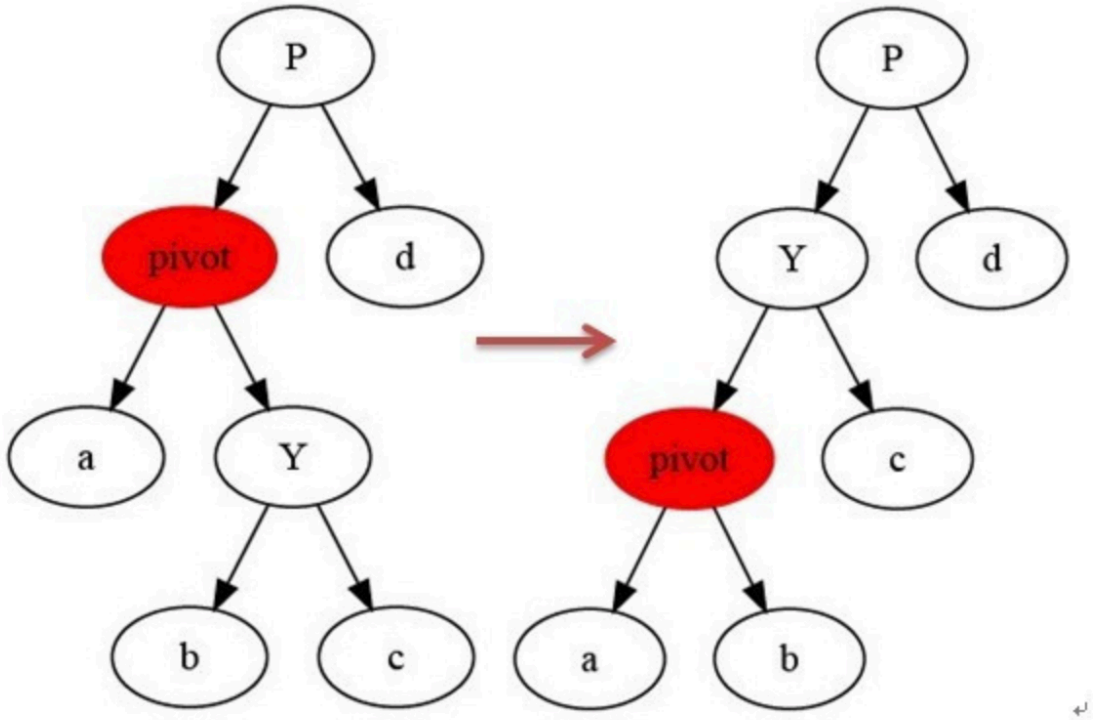
红黑树的五个性质：

1. 每个结点要么是红的要么是黑的。
2. **根结点是黑的。**
3. 每个叶子结点（叶结点即指树尾端NIL指针或NULL结点）都是黑的。
4. 如果一个结点是红的，那么它的两个儿子都是黑的。
5. 对于任意结点而言，其到叶结点树尾端NIL指针的每条路径都包含相同数目的黑结点。

红黑树结构如下：



红黑树的左旋：

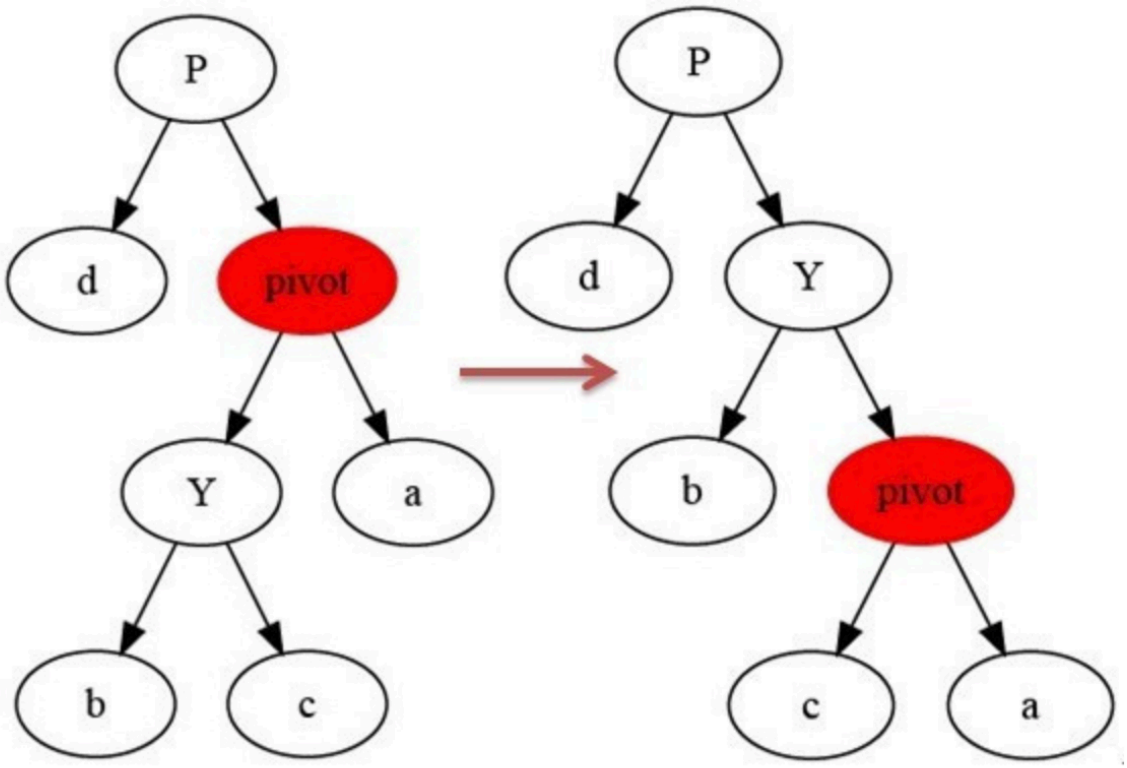


左旋的过程描述:如果原来的节点是a,左旋的时候将a的右孩子取代a的位置，然后将a节点作为a的右孩子的左孩子将a原来的右孩子的左孩子作为a的右孩子

左旋的代码：

```
LeftRoate(T, x)
y ← x.right                    //定义y：y是x的右孩子
x.right ← y.left                //y的左孩子成为x的右孩子
if y.left ≠ T.nil
    y.left.p ← x
y.p ← x.p                      //x的父结点成为y的父结点
if x.p = T.nil
    then T.root ← y
else if x = x.p.left
    then x.p.left ← y
else x.p.right ← y
y.left ← x                      //x作为y的左孩子
x.p ← y
```

红黑树的右旋：



右旋的描述:如果一个节点a需要右旋,将a的左孩子作为新的节点,然后a作为左孩子的右节点,最后将a节点的左孩子的右节点作为a的左孩子

树在经过左旋右旋之后，树的搜索性质保持不变，但树的红黑性质则被破坏了，所以，红黑树插入和删除数据后，需要利用旋转与颜色重涂来重新恢复树的红黑性质。

B树

B树和平衡二叉树稍有不同的是B树属于多叉树又名平衡多路查找树（查找路径不只两个），数据库索引技术里大量使用者B树和B+树的数据结构。

B树的规则：

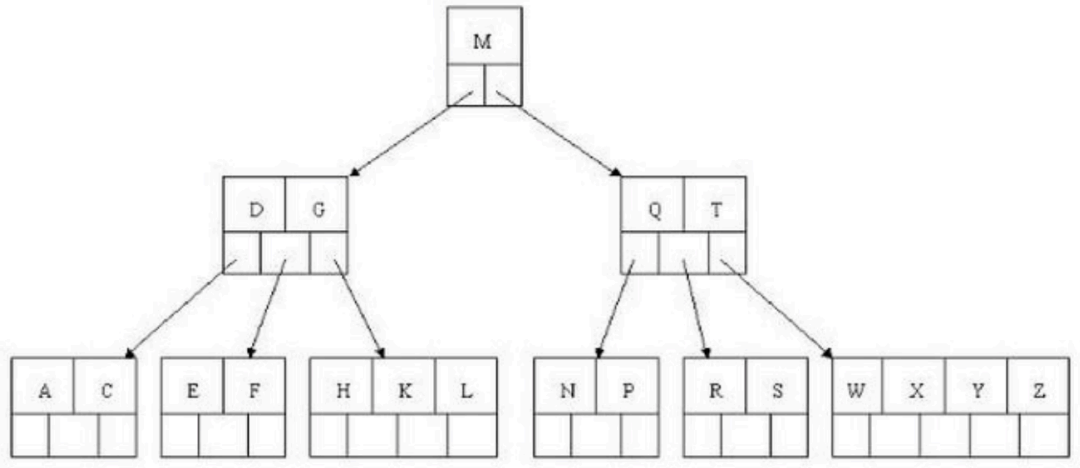
（1）树种的每个节点最多拥有m个子节点且m>=2,空树除外（注：m阶代表一个树节点最多有多少个查找路径，m阶=m路,当m=2则是2叉树,m=3则是3叉）；

（2）除根节点外每个节点的关键字数量大于等于ceil(m/2)-1个小于等于m-1个，非根节点关键字数必须>=2;（注：ceil()是个朝正无穷方向取整的函数 如ceil(1.1)结果为2)

（3）所有叶子节点均在同一层、叶子节点除了包含了关键字和关键字记录的指针外也有指向其子节点的指针只不过其指针地址都为null对应下图最后一层节点的空格子

（4）如果一个非叶节点有N个子节点，则该节点的关键字数等于N-1;

（5）所有节点关键字是按递增次序排列，并遵循左小右大原则；



B树的查询流程： 如上图我要从上图中找到E字母，查找流程如下

（1）获取根节点的关键字进行比较，当前根节点关键字为M，E要小于M（26个字母顺序），所以往找到指向左边的子节点（二分法规则，左小右大，左边放小于当前节点值的子节点、右边放大于当前节点值的子节点）；

（2）拿到关键字D和G，D<E<G 所以直接找到D和G中间的节点；

（3）拿到E和F，因为E=E 所以直接返回关键字和指针信息（如果树结构里面没有包含所要查找的节点则返回null）；

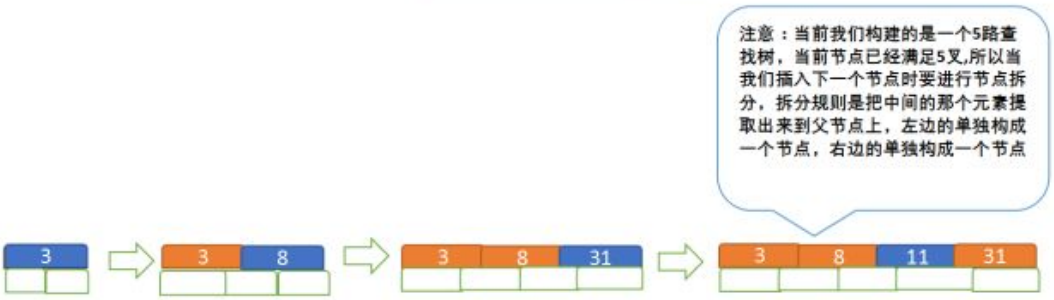
B树的插入节点流程

定义一个5阶树（平衡5路查找树;），现在我们要把3、8、31、11、23、29、50、28 这些数字构建出一个5阶树出来;

遵循规则：

（1）当前是要组成一个5路查找树，那么此时m=5,关键字数必须大于等于cei(5/2)小于等于5-1（关键字数小于cei(5/2) -1就要进行节点合并，大于5-1就要进行节点拆分,非根节点关键字数>=2）；

（2）满足节点本身比左边节点大，比右边节点小的排序规则;



B树节点的删除

规则：

（1）当前是要组成一个5路查找树，那么此时m=5,关键字数必须大于等于cei(5/2)-1，小于等于5-1，非根节点关键字数大于2；

（2）满足节点本身比左边节点大，比右边节点小的排序规则;

（3）关键字数小于二时先从子节点取，子节点没有符合条件时就向向父节点取，取中间值往父节点放；

B树相对于平衡二叉树的不同是，每个节点包含的关键字增多了，特别是在B树应用到数据库中的时候，数据库充分利用了磁盘块的原理（磁盘数据存储是采用块的形式存储的，每个块的大小为4K，每次IO进行数据读取时，同一个磁盘块的数据可以一次性读取出来）把节点大小限制和充分使用在磁盘快大小范围；把树的节点关键字增多后树的层级比原来的二叉树少了，减少数据查找的次数和复杂度;

B+树

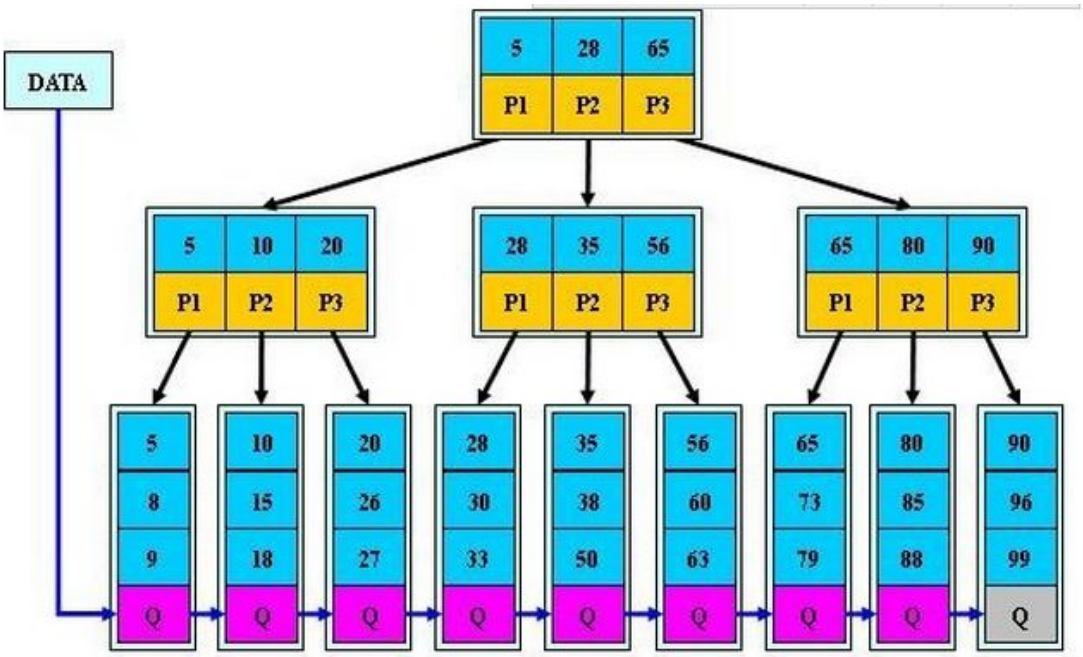
B+树是B树的一个升级版，相对于B树来说B+树更充分的利用了节点的空间，让查询速度更加稳定，其速度完全接近于二分法查找。为什么说B+树查找的效率要比B树更高、更稳定；我们先看看两者的区别

（1）B+跟B树不同B+树的非叶子节点不保存关键字记录的指针，这样使得B+树每个节点所能保存的关键字大大增加；

（2）B+树叶子节点保存了父节点的所有关键字和关键字记录的指针，每个叶子节点的关键字从小到大链接；

（3）B+树的根节点关键字数量和其子节点个数相等;

（4）B+的非叶子节点只进行数据索引，不会存实际的关键字记录的指针，所有数据地址必须要到叶子节点才能获取到，所以每次数据查询的次数都一样；



特点：

在B树的基础上每个节点存储的关键字数更多，树的层级更少所以查询数据更快，所有指关键字指针都存在叶子节点，所以每次查找的次数都相同所以查询速度更稳定;