# 线性表

线性表的定义：一个线性表是n个数据元素的有限序列。（注意：线性表中元素的个数是有限的）。

线性表中元素的个数是n(n>=0)称作线性表的长度。线性表中的元素可以是各种各样的，如数字，符号，一页书等。但同一个线性表中的元素个数必须是相同的。n=0时称线性表为空表。

在稍复杂的线性表中，一个数据元素可以由若干个数据项（item）组成，此时称数据元素为记录，含有大量记录的线性表称作文件。

## 1.1线性表的顺序表示和实现

顺序表指的是用一组地址连续的存储单元依次存储线性表的数据元素，按照这种存储结构存储的机制称作线性表的顺序存储结构或顺序映像。

顺序表的特点、优点和缺点如下表1所示：

表1：顺序表的特点、优点和缺点表

|  |  |
| --- | --- |
| 特点 | ⑴ 线性表中相邻的元素在物理上存储位置相邻。即以元素在计算机内物理位置相邻来表示线性表中数据元素之间的逻辑关系。 |
| ⑵ 随机存取 |
| ⑶ 通常对应高级程序语言中的一维数组 |
| 优点 | ⑴ 不需要额外的存储空间来表示元素间的逻辑关系 |
| ⑵ 可随机存取 |
| 缺点 | ⑴ 插入和删除元素时需要移动大量的元素 |
| ⑵ 必须先进行空间分配，而且还需要考虑空间的扩充 |

## 1.2线性表的链式表示和实现

线性表的链式表示主要有三种：单链表、循环链表和双向链表。

### 1.2.1单链表

单链表指的是用一组任意的存储单元存储线性表的数据元素。使用单链表的时候，关心的只是它所表示的线性表中数据元素之间的逻辑顺序，而不是它所表示的线性表中数据元素之间的物理顺序。

单链表的特点、优点和缺点表如下表2所示：

表2：单链表的特点、优点和缺点表

|  |  |
| --- | --- |
| 特点 | ⑴ 存储单链表的存储单元地址可以连续也可以不连续。 |
| ⑵ 建立链表的过程就是一个动态生成链表的过程。 |
| 优点 | ⑴ 在单链表中插入或删除一个节点时，仅需修改指针而不需要移动元素。 |
| ⑵ 根据需要申请空间，且不要求连续的存储空间。 |
| 缺点 | ⑴ 用指针指示元素之间的逻辑关系，存储空间利用率低。 |
| ⑵ 对表中元素只能按照顺序存取，不可随机访问。 |

单链表中关键字解释表如表3所示：

表3：单链表中名词解释表

|  |  |
| --- | --- |
| 名词 | 解释 |
| 结点(node) | 单链表中存储的每个元素ai。每个节点都包括两个域，数据域和指针域。 |
| 数据域 | 存储的内容是数据信息 |
| 指针域 | 存储的是直接后继存储位置 |
| 头指针 | 单链表中的第一个元素 |
| 头结点 | 在单链表的第一个结点之前附设的结点 |

### 1.2.2循环链表

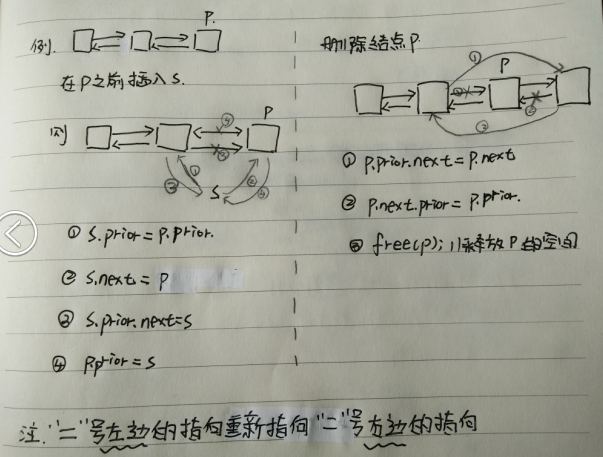
单链表中最后一个结点的指针域指向头结点。整个链表形成一个环，从表中任一点出发均可找到表中其他节点。

### 1.2.3双向链表

双向链表中的节点都有两个指针域，其一指向该结点的直接前趋，另一个指针域指向该结点的直接后继结点。

与单链表的循环链表类似，双向链表也可以有循环双向链表。

在双向链表中插入和删除节点时，需要同时修改两个方向上的指针。



1. 栈和队列

从数据结构角度看，栈和队列是操作受限的线性表。

从数据类型角度看，栈和队列与线性表又有所不同。

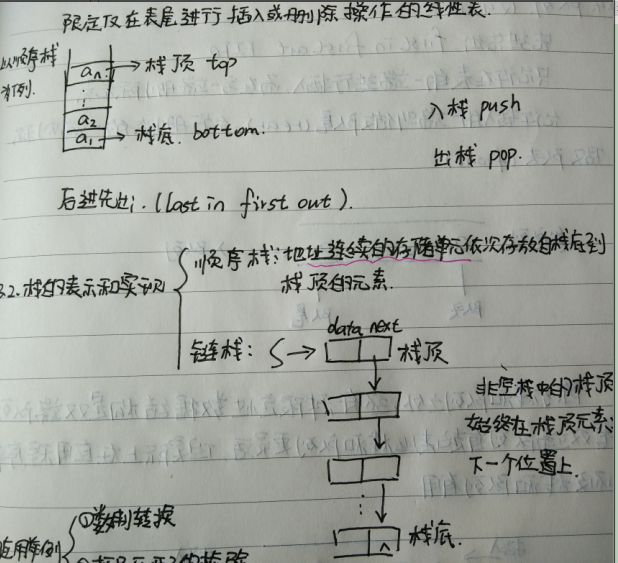
## 2.1栈

栈是限定仅在表尾进行插入和删除操作的线性表。所以栈的最大特点是先进后出（last in first out）。

栈的表示形式有两种：顺序栈和链栈。

栈的引用举例：数制转换、括号匹配的检验、行编辑程序、迷宫求解和表达式求值。

一个直接调用自己或通过一系列的调用语句间接地调用自己的函数，称作是递归函数。



## 2.2队列

队列是一种先进先出的线性表，只允许在表的一端进行插入，而在另外一端删除元素。

允许插入的一端叫做队尾（rear），允许删除的一端则叫做队头(front)。

## 2.3双端队列

除了栈和队列之外，还有一种限定性数据结构是双端队列，尽管双端队列看起来比栈和队列更灵活，但实际上在应用程序中远不及栈和队列有用。

## 2.4链队列

用链表表示的队列。

## 2.5循环队列

循环队列是用顺序表表示的队列，在非空队列中，头指针指向队列元素，而尾指针始终指向队列尾元素的下一个位置。如果用户程序中设有循环队列，则必须为它设定一个最大队列长度，若用户无法预估所用队列的最大长度，则宜采用链队列。

1. 串
2. 串(string)(或字符串)是由零个或多个字符组成的有限序列。
3. 字符串有3种机内表示方法：①定长顺序存储表示；②堆存储表示；③串的块链存储表示。
4. 串的模式匹配算法

子串的定位操作通常称做串的模式匹配，是各种串处理系统中最重要的操作之一。

1. 字符串模式匹配

给定字符串S(匹配串)和T(模式串)，求字符串T在S中首次出现的下标。

注意：对于字符串S来说，其S[0]存放的是字符串的长度。如S=”abcd”,则S[0]=4.

KMP算法：

|  |
| --- |
| int index\_KMP(String S, String T){  int i = 1;  int j = 1;  while(i<=S[0] && j <=T[0]){  /\*如果j == 0 || S[i] == T[j]，则向后移动继续比较\*/  if( j == 0 || S[i] == T[j]){  i++;  j++;  }else{//否则j回溯  j = next[j];  }  }  } |

如何得到模式串T的next数组。此时T即是匹配串，也是模式串。

|  |
| --- |
| int get\_next(SString T){  int i = 1;  int j = 0;//注意这个地方j初始化为0  next[1] = 0;  while(i<T[0]){  if( j == 0 || S[i] == T[j]){//由S[i] == T[j]是否相等，来得出next[i+1]  i++;  j++;  next[i] = j;  }else{//否则j回溯  j = next[j];  }  }  } |

如何得到模式串T的nextval数组。

|  |
| --- |
| int get\_nextval(SString T){  int i = 1;  int j = 0;//注意这个地方j初始化为0  nextval[1] = 0;  while(i<T[0]){  if( j == 0 || S[i] == T[j]){//由S[i] == T[j]是否相等，来得出next[i+1]  i++;  j++;  if(T[i] == T[j]){  nextval[i] = nextval[j];  }else{  nextval[i] = j;  }  }else{//否则j回溯  j = next[j];  }  }  } |

1. 树和二叉树
2. 树(Tree)是n(n>=0)个结点的有限集。
3. 森林(Forest)是m(m>=0)棵互不相交的树的集合。对树中每个结点而言，其子树的集合即为森林。
4. 结点拥有的子树数称为该结点的度(Degree).
5. 二叉树(Binary Tree)是另一种树型结构，它的特点是每个结点至多只有两颗子树(即二叉树中不存在度大于2的结点)，并且，二叉树的子树有左右之分，其次序不能任意颠倒。

## 4.1二叉树的性质

① 在二叉树的第i层上至多有2i-1个结点(i>=1)；

② 深度为k的二叉树至多有2k-1个结点；

证明：深度为k，说明是有k层，结点个数最多为2k-1个

③ 对任何一棵二叉树T，如果终端结点数为n0，度为2的结点数为n2，则n0 = n2+1.

④ 具有N个结点的完全二叉树的深度为log2N+1。(log2N是向下取整)

## 4.2二叉树的存储结构

① 顺序存储结构

用一组连续的存储单元依次自上而下、自左至右存储完全二叉树上的结点元素，即将完全二叉树上编号为i的结点元素存储在如上定义的一维数组中下标为i-1的分量中。

② 链式存储结构

二叉树的结点由一个数据元素和分别指向其左右子树的两个分支构成，则表示二叉树的链表的节点至少包含3个域：数据域和左右指针域。

在有n个结点的二叉链表中必定存在n+1个空链域。(原因是，对于n个结点共有2n个链域，root结点不需要有指针指向，其他n-1个指针需要n-1个链域，所以还剩下2n-(n-1)=n+1)

有时为了便于找到结点的双亲，则还可在结点结构中增加一个指向其双亲结点的指针域。

利用这两种结点结构所得二叉树的存储结构分别称之为二叉链表和三叉链表。

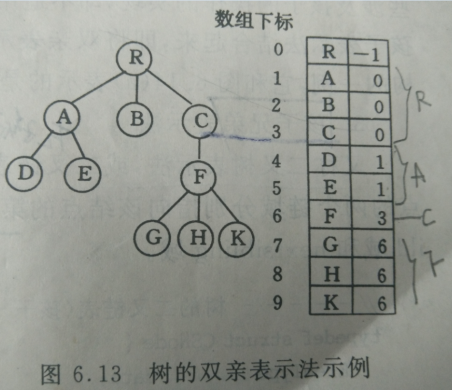
在具体应用中采用什么存储结构，除根据二叉树的形态之外还应考虑需要进行何种操作。

## 4.3树的存储结构

1. 双亲表示法

以一组连续空间存储树的结点，同时在每个结点中附设一个指示器指示其双亲结点在链表中的位置。

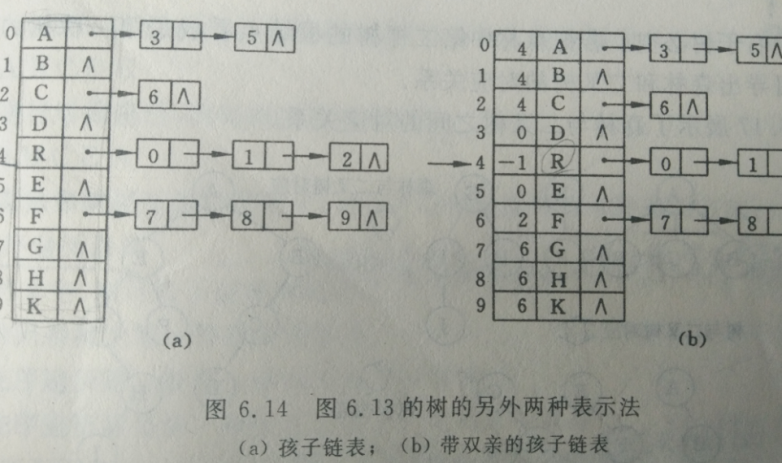
这种存储结构利用了每个结点(根结点除外)只有唯一的双亲的性质。可以利用O(1)的时间复杂度找到每个结点的父亲结点，但是如果要找结点的孩子结点则需要遍历整个结构。



②孩子表示法

把每个结点的孩子排列起来，看成一个线性表，且以单链表作为存储结构，则n个结点有n个孩子链表。n个头指针又组成一个线性表，为了便于查找，可以采用顺序存储结构。

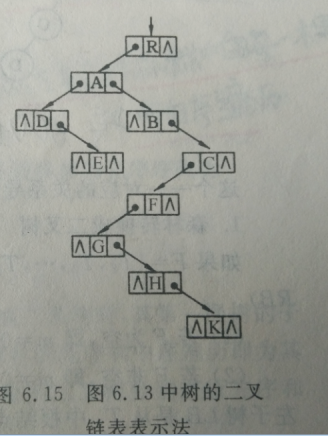
与双亲表示法相反，孩子表示法方便找到孩子结点，但是不方便找到双亲结点。不过可以把双亲表示法和孩子表示法结合起来。



③ 孩子兄弟表示法

又称二叉树表示法，或二叉链表表示法，即以二叉链表作树的存储结构。链表中结点的两个链域分别指向该结点的第一个孩子结点和下一个兄弟结点(左孩子右兄弟)，分别命名为firstchild和nextsibling域。

从firstchild域找到第一个孩子结点，然后沿着孩子结点的nextsibling域连续走i-1步，便可找到x的第i个孩子。此外，也可以为每一个结点增设一个PARENT域，则同样能方便地实现找某个结点的双亲结点。



## 4.4森林与二叉树的转换

给定一棵树，可以找到唯一的一棵二叉树与之对应。

可以把森林中第二棵树的根结点看成是第一棵树的根结点的兄弟，则同样可以导出森林与二叉树的对应关系。

先序遍历森林，相当于其对应二叉树的先序遍历，相当于依次对森林中的每棵树的先序遍历。

中序遍历森林，相当于其对应二叉树的中序遍历，相当于依次对森林中的每棵树做后续遍历。

1. 哈希表

## 1.定义

根据设定的哈希函数H(key)和处理冲突的方法将一组关键字映射到一个有限的连续的地址集（空间）上，并以关键字在地址集中的“像”作为记录在表中存储位置。这种表则称为hash表。这个映射过程称为哈希造表或散列，所得存储位置称为哈希地址或散列地址。

一般情况下，冲突只可能尽量少，但是不能避免。所以，在建造hash表的时候，选择好的hash构造函数和解决冲突的办法至关重要。

## 2.哈希函数的构造函数。

若对于关键字集合中的任意一个关键字，经哈希函数映像到地址集合中的任意一个位置的概率是相等的，则称此类哈希函数是均匀的。也就是说可以使任意一个关键字可被随机地映射到地址中，以便使一组关键字均匀地散列到地址集合中，以便减少冲突。

散列函数有一个共同性质，即函数值应按同等概率取其值域的每一个值。

常见的构造hash函数有：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 编号 | 名称 | 定义 | 冲突 | 备注 | 适用场景 |
| 1 | 直接定义法 | 取关键字或关键字的某个线性函数值为hash地址。H(key)=key或者H(key) = a\*key+b | 不会 | 实际中使用很少。该方法关键字集合和地址集合大小相等 ，不会发生冲突。 | 适合查找表较小且连续的情况。 |
| 2 | 数字分析法 | 假设关键字是以r为基的数（如以10为基的十进制数），并且hash表中可能出现的关键字都是事先知道的，则可以取关键字的若干数位组成哈希地址 | 会 | 不常用 | 关键字位数比较大，就需要事先知道关键字的分布且关键字的若干位分布较均匀 |
| 3 | 平方取中法 | 取关键字平方之后的中间几位为哈希地址。取的位数由表长决定。 | 会 | 较常用 | 不知道关键字的分布，而位数又不是很大的情况 |
| 4 | 折叠法 | 将关键字分割成位数相同的几部分（最后一部分的位数可不同），然后取这几部分的叠加和（舍去最高进位）作为hash地址。  如04 4220 5864 可做04+4220+5864=0088，舍弃最高进位 | 会 | 一般常用， | 事先不需要知道关键字的分布，适合关键字位数较多的情况。 |
| 5 | 除留余数法 | 取关键字被某个不大于hash表表长m的数p除后所得余数为哈希地址。H(key)=key mod p;p<=m  一般取p为小于m的质数或不包含小于20的质因数的合数。 | 会 | 最简单，最常用的构造hash函数的方法。不仅可以对关键字直接取模，也可在折叠、平方取中等运算之后取模 | 关键在于p的选择，选择较好的p则有效减少冲突。一般取p为小于m的质数或不包含小于20的质因数的合数。 |
| 6 | 随机数法 | 选择一个随机函数，取关键字的随机函数值作为它的哈希地址 ，H(key)=random(key) | 会 | 通常当关键字长度不等时采用此法，构造哈希函数较恰当 | 当关键字的长度不等时 |
| 7 | 相乘取整法 | 首先用关键字key乘上某个常数A(0<A<1)，并抽取出key.A的小数部分；然后用m乘以该小数后取整 |  |  |  |

实际中根据不同情况采用不同哈希方法，考虑的因素主要有：

1. 计算hash方法所需时间；2）关键字长度；3）哈希表的大小；4）关键字的分布情况；5）记录的查找频率。

总之，设计哈希构造函数的时候主要要考虑计算简单和散列地址分布均匀这两个特点。

## 3.处理冲突的方法

既然在构造hash表的过程中出现冲突不可避免，那么就需要有处理冲突的方法。下面介绍几种常用的处理冲突的方法。

|  |  |
| --- | --- |
| 开放地址法 | 线性探测再散列 |
| 线性补偿探测法 |
| 二次探测再散列 |
| 伪随机探测再散列 |
| 再哈希法 |  |
| 链地址法（拉链法） |  |
| 建立公共溢出区 |  |

### 3.1 开放地址法

开放地址法主要包括线性探测再散列、线性补偿探测法、二次探测再散列和伪随机探测再散列。

Hi = (H(key)+di)%m

其中H(key)为哈希函数，m为哈希表表长，di为增量序列。

（1）di = 1,2,3..m-1称线性探测再散列；

（2）将线性探测的步长从 1 改为 Q ，即将上述算法中的 j ＝ (j ＋ 1) % m 改为： j ＝ (j ＋ Q) % m ，而且要求 Q 与 m 是互质的，以便能探测到哈希表中的所有单元。（线性补偿探测法）

（3）di = 12,-12,22,-22….k2,- k2称作二次探测再散列。

（4）di = 伪随机序列，称作伪随机探测再散列。

在处理冲突的过程中，两个hash地址不同的关键字争夺同一个后继哈希地址的现象称作是“二次聚集”，即在处理同义词的冲突过程中又添加了非同义词的冲突。

对于线性探测再查找，在哈希表未填满的状态下总能找到一个不发生冲突的地址。对于二次探测再散列，只有哈希表的长度m为形如4\*j+3(j是整数)的素数时，才可以保证哈希表未满情况下总能找到一个不发生冲突的地址。随机探测再散列，则取决于伪随机数列。

### 3.2 再哈希法

Hi = RHi (key) i=1,2,3…k

RHi均是不同的hash函数，即在同义词产生冲突时计算另一个哈希函数地址，直到不产生冲突为止。该方法虽然不容易产生聚集，但是会增加计算的时间。

### 3.3 链地址法（拉链法）

将关键字为同义词的记录存储在同一线性表中，在链表的插入为可以在表头或表尾，也可以在中间，以保持同义词在同一线性链表中按关键字有序。

### 3.4 建立公共溢出区

所有关键字和哈希表中已有的记录发生冲突的，就把这个关键字填入到溢出表中。（溢出表是另建立的一张新表，专门放那些在确定其hash地址的时候产生冲突的记录）

### 3.5 拉链法和开放地址法比较

**与开放定址法相比，拉链法有如下几个优点：**

①拉链法处理冲突简单，且无堆积现象，即非同义词决不会发生冲突，因此平均查找长度较短；

②由于拉链法中各链表上的结点空间是动态申请的，故它更适合于造表前无法确定表长的情况；

③开放定址法为减少冲突，要求装填因子α较小，故当结点规模较大时会浪费很多空间。而拉链法中可取α≥1，且结点较大时，拉链法中增加的指针域可忽略不计，因此节省空间；

④在用拉链法构造的散列表中，删除结点的操作易于实现。只要简单地删去链表上相应的结点即可。而对开放地址法构造的散列表，删除结点不能简单地将被删结点的空间置为空，否则将截断在它之后填人散列表的同义词结点的查找路径。这是因为各种开放地址法中，空地址单元(即开放地址)都是查找失败的条件。因此在用开放地址法处理冲突的散列表上执行删除操作，只能在被删结点上做删除标记，而不能真正删除结点。

**拉链法的缺点**  
    　拉链法的缺点是：指针需要额外的空间。故当结点规模较小时，开放定址法较为节省空间，而若将节省的指针空间用来扩大散列表的规模，可使装填因子变小，这又减少了开放定址法中的冲突，从而提高平均查找速度。

## 4.哈希表的查找及分析

哈希表的查找过程与哈希表的构造过程一致，给定k值，根据哈希函数计算得到哈希地址，若此位置上为空，则查找不成功；否则与关键字进行比较，若相等，则查找成功，否则根据处理冲突的方法查找下一个槽位上的记录。直到找到查找成功或者哈希表中某个位置为空。

哈希表的平均查找长度与哈希表的装载因子有关，与关键字集合包含元素个数无关 。因此，不管n多大，我们总可以选择一个合适的装载因子，以便将平均查找长度限定在一个范围内。

## 5.哈希表的适用

散列技术既是一种存储方法，也是一种查找方法。散列技术最适合的求解问题是查找与定值相等的记录。

散列表不适合范围查找；散列表也不能获得表中记录的顺序，如获得表中最大值和最小值是不可行的。

1. 图

## 6.1图的深度遍历DFS

可采用递归和循环两种方式实现。

方法一：采用递归的方式。

定义一个标志数组表示某个结点是否已经被访问过。以邻接矩阵的形式表示图。依次对深度遍历图中的每个未被遍历过的结点，然后针对该结点未被遍历过的邻接点再进行深度遍历，每个结点被遍历后加入到结果中，并且设定该结点已经被访问过。

方法二：不采用递归的方式而是采用栈保存已经遍历过并且还有其孩子结点可能还没有遍历过的结点。与递归方式中一样，也是定义一个标志数组表示某个结点是否已经被访问过。始化栈内存放第一个结点。判断栈顶元素的第一个还未被访问邻接点，把它压入到栈中，每个结点入栈的时候，把其对应的标志数组元素设为true同时把该结点放到结果中。如果栈顶元素没有了未被访问的邻接点，则将该元素弹栈。循环直到栈空。

|  |
| --- |
| /\*\*  \* 方法一：采用递归的方式。  \* 定义一个标志数组表示某个结点是否已经被访问过。  \* 以邻接矩阵的形式表示图。  \* 依次对深度遍历图中的每个未被遍历过的结点，针对该结点未被遍历过的邻接点再进行深度遍历，每个结点被遍历后加入到结果中，并且设定该结点已经被访问过。  \*/  **public** **static** **boolean** *visited*[];  **public** **static** ArrayList<Integer> *list* = **new** ArrayList<Integer>();  **public** **static** ArrayList<Integer> DFS\_Graph1(**int** graph[][]){  **int** size = graph.length;//求得结点的个数  *visited* = **new** **boolean**[size];  /\*Step1：依次遍历树中还未被遍历的结点\*/  **for**(**int** i=0;i<size;i++){  **if**(*visited*[i] == **false**){  *visited*[i] = **true**;  *list*.add(i);  /\*Step2:深度遍历以结点i为开始的路径上的结点\*/ *DFS*(graph,i);  }  }  **return** *list*;  }  /\*\*  \* 深度递归遍历以图中node结点为起点的结点。  \* **@param** graph  \* **@param** node  \*/  **public** **static** **void** DFS(**int** graph[][],**int** i){  **int** size = graph.length;  **for**(**int** j=0;j<size;j++){  /\*这个当前结点可达且未被访问的结点j才可以\*/  **if**(graph[i][j] == 1 && *visited*[j] == **false**){  *visited*[j] = **true**;  *list*.add(j);  *DFS*(graph,j);  }  }  }    /\*\*  \* 方法二：不采用递归的方式而是采用栈保存已经遍历过并且还有其孩子结点可能还没有遍历过的结点。  \* 与递归方式中一样，也是定义一个标志数组表示某个结点是否已经被访问过。  \* 初始化栈内存放第一个结点。  \* 判断栈顶元素的第一个还未被访问邻接点，把它压入到栈中，每个结点入栈的时候，把其对应的标志数组元素设为true同时把该结点放到结果中。  \* 如果栈顶元素没有了未被访问的邻接点，则将该元素弹栈。  \* 一直循环直到栈空。  \*  \* **@param** graph  \* **@return**  \*/  **public** **static** ArrayList<Integer> DFS\_Graph2(**int** graph[][]){  Stack<Integer> stack = **new** Stack<Integer>();  **int** size = graph.length;  *visited* = **new** **boolean**[size];  /\*Step1:把初始第一个结点放到栈内\*/  stack.push(0);  *visited*[0] = **true**;  *list*.add(0);  /\*  \* Step2:找栈顶元素的第一个还未被访问邻接点，把它压入到栈中，  \* 每个结点入栈的时候，把其对应的标志数组元素设为true同时把该结点放到结果中。  \* 如果栈顶元素没有了未被访问的邻接点，则将该元素弹栈。  \* 循环结束的条件是栈为空。  \*/  **while**(!stack.isEmpty()){  **int** i = stack.peek();//读栈顶元素  **int** j = 0;  /\*找栈顶元素的第一个未被访问的邻接点\*/  **for**(;j<size;j++){  **if**(graph[i][j] == 1 && *visited*[j] == **false**){  **break**;  }  }  **if**(j<size){//说明找到了  *visited*[j] = **true**;  *list*.add(j);  stack.push(j);  }**else**{//只有当前栈顶元素不存在未被访问的邻接点时才弹出栈顶元素  stack.pop();  }  }  **return** *list*;  } |

## 6.2图的广度优先遍历（BFS）

借助队列实现，同时设置一个包含结点个数的visited数组表示某个结点是否已经被遍历过。

初始化时把第一个结点加入到结果list中，并且把visited[0]设置为true。

循环开始时队列中存储的是图中的第一个结点，循环结束的条件是队列变成空。

循环体内取出队列的第一个元素，并且把这个元素的所有未被访问过的邻接点都加入到队列中、加入到结果list中、设置其对应的状态为已访问。

类似于树的层次遍历.

|  |
| --- |
| /\*\*  \* 图的广度优先遍历。  \* 借助队列实现，同时设置一个包含结点个数的visited数组表示某个结点是否已经被遍历过。  \* 初始化时把第一个结点加入到结果list中，并且把visited[0]设置为true。  \* 循环开始时队列中存储的是图中的第一个结点，循环结束的条件是队列变成空。  \* 循环体内取出队列的第一个元素，并且把这个元素的所有未被访问过的邻接点都加入到队列中、加入到结果list中、设置其对应的状态为已访问。  \*/  **public** **static** **boolean** *visited*[];  **public** **static** ArrayList<Integer> *list* = **new** ArrayList<Integer>();  **public** **static** ArrayList<Integer> BFS\_Graph(**int** graph[][]){  Queue<Integer> queue = **new** ArrayBlockingQueue<Integer>(10);  **int** size = graph.length;  *visited* = **new** **boolean**[size];  /\*Step1:把初始第一个结点放到栈内。\*/  queue.add(0);  *visited*[0] = **true**;  *list*.add(0);  **int** i ;  /\*Step2:循环队列中的元素直到队列为空。\*/  **while**(!queue.isEmpty()){  i = queue.poll();  **for**(**int** j =0;j<size;j++){//找刚刚被弹出队列的队头元素可达的点  **if**(graph[i][j] ==1 && *visited*[j] == **false**){  *visited*[j] = **true**;  *list*.add(j);  queue.add(j);  }  }  }  **return** *list*;  } |

1. 海量数据处理

通过对海量数据的挖掘能有效地揭示用户的行为模式，加深对用户需求的理解，提取用户的集体智慧，从而为研发人员决策提供依据，提升产品用户体验，进而占领市场。

海量处理的基本方法：

## 7.1Hash法

构造hash表和处理冲突的方法和上面介绍的一样。

在处理海量数据的过程中，使用Hash方法一般可以快速存取、统计某些数据，将大量数据进行分类，例如提取某日访问网站次数最多的ip。

## 7.2BitMap法

BitMap(位图)法的基本原理是使用数组来表示某些元素是否存在。

BitMap(位图)法的时间复杂度为O(n)，比一般的排序都快，但它是以空间换时间，而且又一些限制，即数据状态不是很多，例如排序前集合大小最好已知，而且集合中元素的最大重复次数必须已知。

## 7.3.Bloom Filter 法（适用于对低错误率可以容忍的场合）

Bloom Filter是一种空间效率和时间效率都很高的随机数据结构，可用来检测一个元素是否属于一个集合。其基本原理是位数组与Hash函数的联合使用。首先，Bloom Filter是一个包含m位的位数组，数组的每一位都初始化为0；其次定义k个不同的hash函数，每个函数都可以将集合中的元素映射到位数组中的某一位，这样k个hash函数就将某个元素映射到了位数组中的k个位。查询的时候，根据k个hash函数可以得到数组中的k个位，判断这个k个位是否全为1，若是则说明该元素可不存在。插入的时候，把k个hash函数就将某个元素映射到了位数组中的k个位全部置为1。

Bloom Filter的难点是如何根据输入元素个数n确定m和hash函数。

特点：确定不存在的元素一定不存在，确定存在的数据不一定存在。Bloom Filter只能插入不能删除元素。

优点：具有很好的空间效率和时间效率，它的插入和查询时间都是常数，另外它不保存元素本身，具有良好的安全性。

缺点：牺牲了正确率，当它判断一个元素已存在于集合中时可能不正确。因为可能该元素对应的k个1位可能是被其他多个元素导致的。

## 7.4 数据库优化法

互联网上的数据一般都是被存储到数据库中，如何从数据库中存储的海量信息中提取对自己有用的信息就涉及到了数据的查询技术等相关内容。

常见的数据库优化方法有数据分区、索引、缓存机制、分批处理、优化查询语句、使用采样数据进行数据挖掘等。

## 7.5 倒排索引法

倒排索引是目前搜索引擎公司对搜索引擎最常用的存储方式，也是搜索引擎的核心内容。按照关键字建立索引，关键字指向了包含它的文档。优点是在处理复杂的多关键字时，可在倒排表中完成查询的并、交等逻辑运算，得到结果后再对结果进行存取，这样把记录的查询转换为地址集合的运算，不必对每个记录随机存取，从而提高查找速度。

## 7.6 外排序法

外排序是相对于内部排序而言的，它是大文件的排序，由于内存限制，不能一下子把所有的待排内容都读到内存中进行排序，需要在内存和外部存储器之间进行多次数据交换才能达到对整个文件进行排序的目的。

一般采用归并排序等方式进行外部排序，首先生成若干个子文件，分别对这些子文件进行排序，然后对这些子文件进行多次归并，使得有序的归并段主键扩大，最后在外存上形成整个文件的单一归并段。

外排序适用于大数据的排序以及去重，但外排序的缺陷是会消耗大量的IO，效率不高。

## 7.7 Trie树

Trie树是一种用于快速字符串检索的多叉树结果 ，原理是利用字符串的公共前缀来减少空间开销。经常被搜索引擎系统用于文件词频统计。优点是：最大限度地减少无谓的字符串比较，查询效率比散列表高。适合用于数据量大，重复多，但是数据种类小可以放入内存的情况。

但是当系统中存在大量字符串且字符串基本没有公共前缀，则相应的Trie树会非常消耗内存。

## 7.8 堆

|  |  |
| --- | --- |
| 最大堆 | 求前k小 |
| 最小堆 | 求前k大 |
| 双堆 | 中位数 |

比如用最小堆求前k个最大的元素：

先从待找的n个数字中找出前k个建成小顶堆，然后依次读后面的元素并与小顶堆的堆顶进行比较，如果当前元素小或相等（这个相等是自己加的，需要进一步验证，应该是正确的，因为此时的堆顶元素是前k个里面最小的。如果遇到相等的即使该元素在最后前k个里面，也可以不要那个元素因为之前的已经够k个了，不需要再进行替换了），则继续读后面的元素；如果当前的元素大，则用当前元素替换堆顶元素，然后调整小顶堆。

## 7.9 双层桶法

双层桶不是一种数据结构 ，而是一种算法思想，类似于分治思想。因为元素范围很大，不能利用直接寻址表，所以通过多次划分，逐步确定范围，然后最后在一个可以接受的范围内进行。

桶排序一般适用于寻找第k个大的数，寻找中位数、寻找不重复或重复的数字。

如：1）在一个文件中有10G个整数，乱序排列，要求找出中位数，内存限制是2GB；

2）现在有一个0~30000的随机数生成器。请根据这个随机数生成器，设计一个抽奖范围是0~350000彩票中奖号码列表，其中要包含20000个中奖号码。

## 7.10 MapReduce法

MapReduce是云计算的核心技术之一，是一种简化并行计算的分布式编程模型，主要目的是为了大型集群的系统能在大数据集上进行并行工作，并用于大规模数据的并行运算。

海量数据处理的最大难题在于数据规模巨大，使得传统处理方式面临计算能力不和存储能力不足的瓶颈问题，而基于Hadoop可以非常轻松和方便完成处理海量数据的分布式并行程序，并运行与大规模集群上。

1. 红黑树

## 8.1性质

红黑树是一棵二叉搜索树，它在每个结点上增加了一个存储位来表示结点的颜色，是黑色还是红色。

通过任何一条从根到叶子的简单路径上各个结点的颜色进行约束，红黑树确保没有一条路径会比其他路径长出两倍，因此也是近似于平衡的。

一棵红黑树确保满足下面红黑性质的二叉搜索树。

① 每个节点要么是红色的要么是黑色的；

② 根节点是黑色的；

③ 每个叶节点(NIL)是黑色的；

④ 若一个结点是红色的，则它的两个孩子结点都是黑色的；

⑤ 对每个结点，从该结点到其所有后代叶结点的简单路径上，均包含相同数目的黑色结点。

## 8.2 黑高

从某个结点x(不包含该结点)出发到达一个叶结点的任意一条简单路径的黑色结点个数称为该结点的黑高(black-height)，记为bh(x)。

定义红黑树的黑高为其根结点的黑高。

对于给定的黑色高度为n的红黑树，从根到叶结点的简单路径的最短长度为(n-1)，从根到叶结点的简单路径的最大长度为2(n-1).

## 8.3 定理

一个有n个内部结点的红黑树的高度至多为2lg(n+1).

## 8.4 操作

默认插入的节点颜色都是红色，因为插入黑色节点会破坏根路径上的黑色节点总数，但即使如此，也会出现连续红色节点的情况。因此在一般的插入操作之后，出现红黑树约束条件不满足的情况（称为失去平衡）时，就必须要根据当前的红黑树的情况做相应的调整其他。

## 8.5 其他

红黑树并不追求“完全平衡”，它只要求部分地达到平衡要求。降低了对旋转的要求，从而提高了性能。

红黑树能够以O(lgn)的时间复杂度进行搜索，插入和删除操作，此外，由于红黑树的设计，任何不平衡都会要求在三次旋转之间解决。

红黑树并不应用所有应用树的领域。如果数据基本上是静态的，那么让数据待在能够插入，并且不影响平衡的地方具有更好的性能。如果数据是完全静态的，例如，做一个hash表，性能可能会更好一点。

在实际的系统中，需要使用动态规划的防火墙系统，使用红黑树而不是散列表，被实践证明具有更好的伸缩性。