# Data Structure

线性表，栈和队列

线性表的两种表示:  顺序表(array)和链表(list)

可利用空间表(Freelist)，将删除的节点放入可利用表中，当申请新节点的时候，从可利用空间表中找节点，如果找不到再分配空间。

XOR链表，见算法一文。

栈的表示方式: 顺序栈和链式栈

队列的表示方式: 顺序队列和链式队列

# 二叉树

满二叉树：除了叶节点， 每个节点都有两个子节点

完全二叉树：除了最后一层，其他每层都是满的；底层叶节点集中在左边若干位置上

前，中，后续周游。

用**数组可以实现完全二叉树**，parent a[n], left child, a[2n + 1], a[2n+2]

**Huffman编码**：贪心法的典型代表。最频繁的字母占用最短的空间。路径就是其编码

首先按照频率大小顺序将字母排序，然后取出最小的两个作为子节点，其父节点的频率为两者之和。继续这一过程，直到所有字母在树中。整个过程中始终保持左子节点比右子节点小。

**二叉检索树(BST) :** 按照中序周游打印各个节点，就可将节点排序

删除BST节点时，如果该节点只有一个子节点，则将被删除节点的父节点指向子节点即可； 如果有两个子节点，则需要找出右边子树的最左节点（最小节点），将该节点放到被删除的节点的位置。

**堆与优先队列**:

堆是完全二叉树且局部有序(父节点比两个子节点都大或者小）.

void buildheap()

{

    for (int i = n/2-1; i>=0; i--) shiftdown(i);

}

void shiftdown(i)

{

while (!isLeaf(i))

{

    int j = leftChild(i);

    if ((j < n - 1) && heap[j] < heap[j +1])

       j++; //Go to right child;

   if (heap[i] >= heap[j]) return;

   swap(heap[i], heap[j]);

   i = j;

}

}

# 树

树的先根周游和后根周游与二叉树类似，中根周游五明确定义

树的父指针表示法：每个子保持一个到父的指针。该表示法可精确回答：给定两个节点，它们是否在同一颗树中。

用Union/Find算法实现集合，每一个树是一个集合，可以通过root确定两个元素是否在同一个集合中。

bool differ (int a, int b)

{

    return root(array[a]) != root(array[b]);

}

bool union(int a, int b)

{

   if (root(array[a] != root(array[b]))

     root(array[a]).parent = root(array[b]);

}

该方法也可以用于判断和合并等价类。 （等价元素组成树，如果两个等价元素在不同的树中，合并之）。该方法也可用于求最大连通图。

可以通过**重量权衡合并规则**（小树指向大树的根节点）降低树的深度。**路径压缩**可以产生更浅的树，在查找某个节点X时，顺便把该节点到根的所有节点的父节点指向根节点。

树的表示法：

子节点表 （索引，值，父节点， 兄弟节点索引表）

左子节点/右兄弟节点 （左子节点，值，父节点，右兄弟节点,其中左子节点通常是单步，右兄弟有多步）

动态节点表示法，每个节点存储值以及指向所有子的表（链表或数组）

树的顺序表示法，按先序周游输出，如果是子节点，后面跟)，如果是父节点的最后一个子节点，后面跟多跟一个)。对于一个n层树的最右子节点，有n个).

# 图

图的存储方式：相邻举证（密集图）或邻接表（稀疏图）

图的深度周游和广度周游, **DFS采用递归，BFS采用队列**，对于每个节点，需要记录visited的状态，防止重入。

拓扑排序可以通过：**DFS**并倒序输出，无法判断回路。通过**队列**，对没有先决条件的定点入队，并输出，然后出队，并且对以该顶点为先决条件的其他定点先决条件－1，如果先决条件为0，则继续入队，直到所有都输出。如果输出个数小于节点个数，**证明有环路存在**

单点最短路径:**Djistra算法**

d(s,x) = min(d(s, u) + d(u,x))

* 从起始点A出发,存储起始点到其他所有定点的距离入d数组, 不可达的设为无限大;
* 找出最小距离的定点B,并且通过B到其他定点的距离更新A到其他定点的距离;
* 如此继续,直到所有顶点被处理, d数组就是A到所有顶点的最短路径.

 使用顺序查找(密集图)或者最小堆(稀疏图)可帮助提高性能

每对顶点的最短路径

1. 所有顶点采用Djistra算法

2. **floyd算法**,三重循环, d[i,j] = Min(d[i, k] + d[k,j]),采用动态规划法(i,j,k in [0..n))

最小支撑树(MST)

**Prim算法**:贪心法. 取一个顶点A,找出与它边最小的相邻节点B组成集合, 然后找到A, B边中最小的且目标C不在AB集合中的边,加入C到该集合, 并加该边入MST.

**Kruskal算法**: 将每个顶点作为一个等价类, 然后按照边的权重大小顺序采用Find/Union方法合并等价类. 如果某条边连接了两个等价类,则加该边入MST.可用最小堆提高性能.

**每次需要找最大值和最小值时,都可以考虑用堆提高性能.**

# 排序和检索

插入排序，冒泡排序，选择排序(O(n2))，其中插入排序有最佳的最小代价

Shell 排序：按照不同的步长进行排序，最后按照步长为1进行排序，每个子排序采用插入或其他传统排序。增量的选择很重要(~O(n1.5)

**快速排序**：最差性能(O(n2)), 平均性能(O(nlogn)). **分治法 (vs 贪心法 vs 动态规划法)**

qsort的轴值选择很重要，通常选择array[(i+j)/2]，也可以采用从多点中进行动态选择提高效率。

当qsort分割到一定细度（如元素个数小于9时），采用传统方法（如插入）可提高性能, 也可以用递归替代栈.

qsort (Ele \* array, int i, int j)

{

    int pivot = FIndPivot(i,j);

    swap(array[pivot], array[j]);

    int k = Partition(array, i, j, array[j]);

    swap(array[pivot], array[j]); //swap back;

    if (k - i > 1) qsort(array, i, k -1);

    if (j - k > 1) qsort(array, k+1, j);

}

int Partition(Ele \* array, int i, int j, int v)

{

    do {

     while (array[++i] < v) ; move index right till the number bigger than v is found.

     while (r && array(--r) > v); move index left till the number small than v is found.

     swap(array[i], array[j]);

    } while (i < j);

    swap(array[i], array[j]) //Reverse the last swap. even if i==j, it doesn't matter.

    return i;

}

**归并排序**：(O(nlogn)).

list mergesort(list li)

{

    if (li.length == 1) return list;

    list lil = mergesort(left half of li);

    list lir = mergesort(right half of li);

    return merge (lil, lir);

}

list merge(lil, lir)

{

    reverse the list lir;

    int indexl = first index of lil;

    int indexr = last index of lir;

    for (int i = 0; i < list.length; i++)

      if (lil[indexl] < lir[indexr]

                list[i] = lil[indexl++];

      else list[i] = lil[indexr--);

}

**堆排序**: (O(nlogn));

1. 用右下的值替代最小值并重新建堆（shiftdown(0));

2. 返回最小值

依次输出返回值就已经排序

**分配排序：**O(n)，费空间。将节点放入与关键字对应的位置。比如对关键字在0-n之间的排序。

**桶式排序：**每个桶和多个关键字对应（比如末尾为1），然后对桶中的值采用其他方法排序。

**基数排序:** O(nlogn) 对关键字的某个或者某些位来排序（比如比较int时，按照byte从低到高排序），基数越大，需要的趟数越少，(比如比较int，如果采用bit要32趟，采用byte需要4趟）。32/4%8.

为了节省空间提高效率，对数组A排序时，可以对每个桶的将有的数据个数采用数组count来基数。使用数组B记录每次结果。count的长度是基数的范围（如byte就是8)，count的值为对应该基数值的数据在B中的最大index.

**quicksort, heapsort, mergesort, radixsort** are O(nlogn).

# 文件管理和外排序

数据缓冲方法：FIFO, LFU(最不频繁使用）, LRU(最近最少使用）

所有好的外排序基于如下两步：

1. 把文件分成大的**初始顺串**

2. 把所有的顺串**归并**到一起，组成一个大的顺串

外排序的关键是建立顺串. 加入有内存M, 可以把Ｍ大小的数据载入内存,排序,并输出为顺串

采用置换排序,建立最小堆大小为M, 将root输出,然后对堆外的数据与输出的值进行比较, 如果大,则将其加入堆中, 如果小, 将堆中最后一个元素作为root, 重建堆, 并将堆外的数据载入到内存(但不在堆内),直到堆的大小为0. 从平均概率来说,能建立大小为2M的堆.

如果内存够大,可以存放多个块, 采用多路归并可以减少归并次数.

外排序提高效率:

1. 尽可能建立长的初始顺串

2. 在所有阶段尽可能把输入,处理和输出重叠

3. 使用尽可能多的工作主存

# 检索

**二分检索和字典检索(插值检索, 类似于字典里找字)**

**自组织线性表**,根据访问的行为建立线性表来优化访问:　计数法, 移到最前端, 与前一元素转置等.

**比特向量**:

对于某个检索特性,对内容建立比特响亮,比如对10以内的素数, 0110101000.如果需要查询包含多个向量的内容,可以通过&, |, -(&~), XOR等方式来进行组合查询.

对于单个内容(比如文件),也可以针对检索特性建立比特响亮, 1为包含此特性, 0为没有. 此方法称为签名文件.

**散列方法:**

散列方法一般用于把关键码范围很大的记录存储在槽相对较少的表中.

散列不适于范围检索, 也不适于多个记录有同样的key(冲突),而适于检查那个记录有关键码值K.

大型数据库通常使用散列和B树（索引的一种）.

散列检索: 1. 计算表的位置H(key), 从槽H(key)开始,使用冲突解决策略定位包含key的记录.

**散列函数**需要尽量让记录均匀分布, 做法是尽量让key的每以为都参与.

解决冲突的一个方法是**开散列方法**, 但是在磁盘中有效的存储一个开散列表比较困难,它们通常在不同的磁盘页上.

**闭散列方法**: 当冲突产生时,由冲突解决策略决定将记录放在哪个位置.

**桶式散列**比较适合磁盘存储.每个Slot可以放多个记录,如果Slot里放不下了,统统放入溢出桶中. 查找时,如果在Slot里头中查不到,就到溢出桶中查找. 溢出桶是所有Slot共享的.

**线性探查**: 如果Slot被占据,就通过线性探查函数继续查找下一个空Slot,直到找到一个. 检索时,采用同样的方式,直到找到同一个Key. 通过**伪随机探查**(随机数是预先确定的) 和 **双散列方法**(第一个散列用key确定基位置,第二个散列用key确定探查序列)来防止探查序列的聚集.

删除散列时,可以才用在散列位置放入**墓碑**的方法,避免后续散列无法搜索. 插入时,移除墓碑.

# 索引技术

**索引文件**的记录由关键码值一起与其相关的指向主数据库中的完整记录的指针构成。索引文件允许访问特定记录，避免重新组织原来数据的高昂代价。

一个数据库可能有多个索引文件。除了主关键码索引文件（针对数据表key的索引文件）， 还可能有次关键码索引文件。通常次关键码索引与具有这个次关键码的每一个记录的主关键码相关联。

**线性索引**是按关键码/指针对顺序组织的文件。适合于二分法搜索。线性索引的缺点是频繁重建和空间浪费（当多个记录有相同key时）。

**倒排表（倒排文件）**按照次关键词排序，每个次关键词指向一组主关键词（主关键词指向记录指针）。存储倒排表的一个好方法是将主关键码按照（主关键码，下一个主关键码index)的数组的方式存储；次关键码指向第一个主关键码的index. 倒排表适用于次关键码较少并且大量重复。

**B树:** 一个m阶B树是书中每个内部节点都包含多达m个分治的树。内部节点有m个子女，存储m－1个关键码。

一个m阶B树有如下特征

1. 根是一个叶节点或者至少有两个子女

2. 除了根和叶节点外，每个节点有『m/2』 到 m 个子女

3. 所有叶节点都在同一层，树是高度平衡的。

插入节点的时候，从下往上split来保持B树的特性。

**B+树**：在实际中，几乎都使用B+树来实现。

B+树中，中间节点值存储关键码值（两颗子树的分界值），只在叶节点同时存储关键码和记录（或者记录的指针）。B+树的叶节点一般链接起来， 形成链表。

低层次树（当阶够大时），连续索引集中存储（减少磁盘访问），所有索引构成链表（快速范围检索），空间浪费少（磁盘页至少保持半满）。

B+树插入类似B树插入；B+树删除节点时，可以从右边子节点借节点过来保持子节点个数>=m/2，如果无法借，则合并节点。删除节点时，从下到上更新中间节点的索引值。

B/B+树在阶够大时，在树层次很浅的情况下就可以索引很多记录（4层100阶的B+树可以最多索引1亿个记录）.

**跳跃表:** 多级链表,级数越高越稀疏. 搜索时先从高级数开始定位,然后开始搜索低级数,直到搜索完成. 均匀分布的级数会导致插入和删除困难.一般做法是当插入某个记录时,随机生成它的级数(n级的概率是n-1的1/2),并且将其与其他同级的链表相连. 数据够大时,搜索性能接近logn.

稀疏矩阵的表示可以用**正交表**.

存储分配可以用**伙伴方法**或者**顺序分配**

**顺序分配**: first fit; best fit, worst fit

伙伴方法: 把已有内存切成2^n大小的块(每个块的n值不同).分配时,找到最小的n满足2^n>m,如果有,分配整个块;否则分割2^(n+1)为两块,并且分配其中一块.

**k-d树**: BST(二叉搜索树)对多关键字记录的一种变种. 对于每一层, 一次采用不同的关键字进行BST排序.搜索时,沿着同样的方式检索.

4分树适合二维空间分割;8分树适合三维空间分割.