# Part 1 Concept

## Base

* 逻辑结构：数据之间的逻辑关系
* 储存结构：数据元素及其关系在储存器中的表示
* 顺序储存结构：逻辑上相邻的节点在物理位置上相邻的储存单元里

时间复杂度

Time complexity。它定性描述该算法的运行时间

渐进时间复杂度

若存在函数 f（n），使得当n趋近于无穷大时，T(n）/f（n）的极限值为不等于零的常数，则称 f（n）是T（n）的同数量级函数。记作 T（n）= O（f（n）），称O（f（n）） 为算法的渐进时间复杂度，简称时间复杂度。

复杂度 O（1）< O（logn）< O（n）< O（n^2）

## 文件

逻辑结构和储存结构：

数据结构的存储结构是和相应的数据在内存中的物理地址之间的关系有关。而逻辑结构只是描述数据之间的关系（三大逻辑结构的一种）。

举例说，线性表（元素之间的逻辑关系是线性的）可以是顺序存储的方式，即所有元素相邻存放，在物理地址上是连续的（存储结构）；而对于链式存储的线性表，他的所有元素之间不一定是线性相连的，可能是第一个结点（元素）的地址为0x123,而第二个元素又出现在物理地址0x100上。也就是说逻辑结构是线性的但是存储结构不一定就是线性的了。

**文件的组织方式（存储结构）：**

在外存的组织结构：顺序组织，索引组织。。。

**顺序文件：**

逻辑结构和物理结构顺序一致

**索引文件：**

**索引顺序文件：**

ISAM（Indexed Sequential Access Method）：索引顺序存取

静态索引结构，

VSAM(Virtual Storage Access Method):虚拟储存存取方法

也是顺序索引结构，采用B+树的动态索引结构

**散列文件：**

**多关键字文件：**

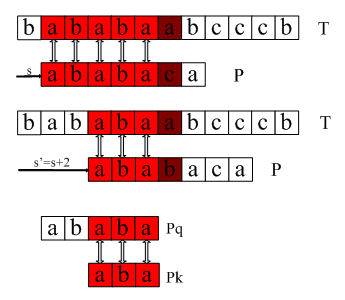
第二章 线性表

# Part 2 数据结构

## 串

子串定位又模式匹配Pattern Matching。子串称模式串，主串称目标串；

朴素的串匹配：



## 顺序表

Sequential list

1. #define ListSize 100
2. Typedef **int** DataType;
3. Typedef **struct**{
4. DataType data[ListSize];
5. Int length;
6. }Seqlist

插入：

指在表第i（1<=i<=n+1）位置，插入x，使n长度变成n+1

void insertList(Seqlist \* L,DataType x,int i)

{

if(i<1||i>L->length+1) //可以在length+1上插

Error(“position error”);

If(L->length>=ListSize)

Error(“overflow”);

For(j=L->length-1;j>=i-1;j--)

{

L->data[j+1]=L->data[j];

}

L->data[i-1]=x;

L->length++;

表中位置i比向量下标大1，

## Heap

是计算机科学中的一种特别的树状数据结构。满足下面特性

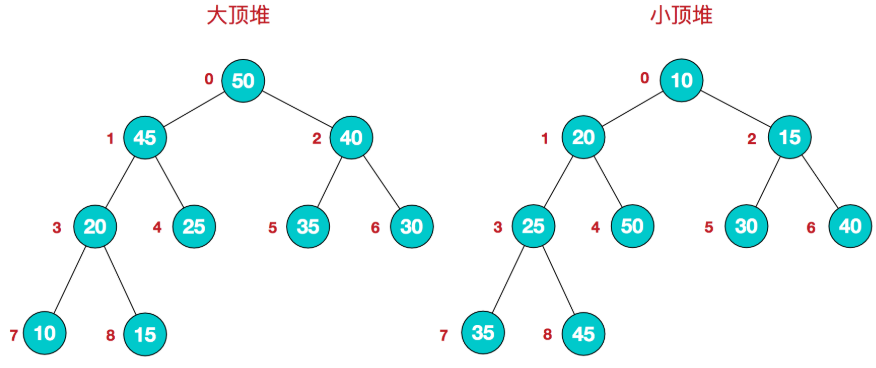
大顶堆：每个结点的值都大于或等于其左右孩子结点的值。

小顶堆：每个结点的值都小于或等于其左右孩子结点的值。

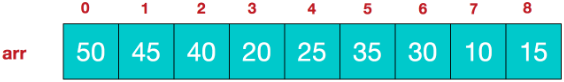
二叉堆

堆并不一定是完全二叉树，平时使用完全二叉树的原因是易于存储，并且便于索引。

* 二叉堆是一颗完全二叉树。
* 每一次堆调整的复杂度为O(logn)，数的高度?
* 平均时间复杂度均为O(nlogn)，它也是不稳定排序。



同时，我们对堆中的结点按层进行编号，将这种逻辑结构映射到数组中就是下面这个样子



该数组从逻辑上讲就是一个堆结构，我们用简单的公式来描述一下堆的定义就是：

大顶堆：arr[i] >= arr[2i+1] && arr[i] >= arr[2i+2]

小顶堆：arr[i] <= arr[2i+1] && arr[i] <= arr[2i+2]

## Stack

**顺序栈：**

#define StackSize 100

typedef struct{

DataType data[StackSize];

int top;

} SeqStack;

Int StackEmpty(SeqStack \* s)

{

Return s->top==-1;

}

void push(SeqStack \* s,DataType x)

{

s->data[++s->top]=x;

}

**链栈**

struct StackNode{

DataType data;

struct StackNode \* next;

}

Typedef struct{

StackNode \* top;

}LinkStack;

指向指针的指针。LinkStack \*s;

initStack :s->top=NULL;

int StackEmpty

{

return s->top==NULL;

};

入栈

void push(LinkStack \* s,x)

{

StackNode \* p=(StackNode \*)malloc(sizeof(StackNode));

p->data=x;

p->next=s->top;

s->top=p;

}

## Queue

区别于线性表list

双链表

Typedef struct dlistnode{

DataType data;

Struct dlistnode \* prior,\* next;

}DlistNode;

**队列：**

struct Queue

{

char data[QueueSize];

int front,rear;

};

**循环队列：**

顺时针旋转，rear逻辑上在front之前,m[0]上有元素，rear提前有个空位置时front=(rear-len+m)%m。

rear没有时（题目中会说**rear指向尾元素的位置**），（rear-len+m+1）%m

Rear，front的长度都<QuenueSize，当转一圈后，从num为1；

typedef struct

{

char data[QueueSize];

int count;

int rear,front;

}CirQueue;

void initCirQueue(CirQueue \*cirQueue)

{

cirQueue->count=0;

cirQueue->front=0;

cirQueue->rear=0;

}

int isCirQueueEmputy(CirQueue \*cirQueue)

{

return cirQueue->count==0;

}

int isCirQueueFull(CirQueue \*cirQueue)

{

return cirQueue->count==QueueSize;

}

void EnQueue(CirQueue \*cirQueue,char x)

{

if(isCirQueueFull(cirQueue))

{

printf("cirQueue is Full");

exit(1);

}

cirQueue->data[cirQueue->rear]=x; //队尾进入

int num= (++cirQueue->rear)%QueueSize;//如果队列长10，

cirQueue->rear=num;

cirQueue->count++;

}

char DeQueue(CirQueue \*cirQueue)

{

if(isCirQueueEmputy(cirQueue))

{

printf("cirQueue is Emputy");

exit(1);

}

char str=cirQueue->data[cirQueue->front];

cirQueue->front=(++cirQueue->front)%QueueSize;都是++，顺序循环

cirQueue->count--;

return str;

}

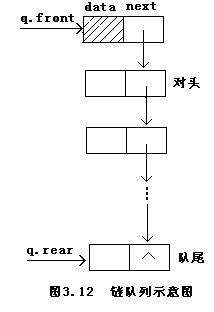
**循环队列**

Typedef struct quenuenode{

DataType data;

Struct quenuenode \* next;

}QuenuNode;



Typedef struct{

QueueNode \* front;

QueueNode \* rear;

}LinkQueue;

循环单链表：

rear->next->next来表示front

双链表：

Typedef struct dlistnode{

DataType data;

Struct dlistnode \* prior,\*next;

}DlistNode;

## Graph

 connected component

 connected component of an [undirected graph](http://en.wikipedia.org/wiki/Undirected_graph) is a [subgraph](http://en.wikipedia.org/wiki/Subgraph) in which any two vertices are [connected](http://en.wikipedia.org/wiki/Connected_graph) to each other by [paths](http://en.wikipedia.org/wiki/Path_(graph_theory)), and to which no more vertices or edges (from the larger graph) can be added while preserving its connectivity. That is, it is a maximal connected subgraph.

1.undirected graph

2.in graph,any two vertices are connected

3.connected component is maximal connect subgraph(to which no more vertices or edges)

**strongly connected component**

A [directed graph](http://en.wikipedia.org/wiki/Directed_graph) is called *strongly connected* if there is a [path](http://en.wikipedia.org/wiki/Path_(graph_theory)) from each [vertex](http://en.wikipedia.org/wiki/Vertex_(graph_theory)) in the graph to every other vertex. In particular, this means paths in each direction; a path from *a* to *b* and also a path from *b* to *a*.(两个点双向都能达到就是强连通的)

The strongly connected components of a directed graph *G* are its maximal strongly connected [subgraphs](http://en.wikipedia.org/wiki/Subgraph). If each strongly connected component is contracted to a single vertex, the resulting graph is a directed acyclic graph, the condensation of *G*. A directed graph is acyclic if and only if it has no (nontrivial) strongly connected subgraphs (because a cycle is strongly connected, and every strongly connected graph contains at least one cycle).

1.directd graph

2.maximal strongly connected,

3.a single vertex or cycle

源(source):当一个强连通分量没有一条从其他连通分量指向这个连通分量中的任何一个顶点时(就是没有入度)，我们就可以称之为一个源；比如说上图的A就是一个源。

汇(sink):当一个强连通分量没有一条从这个连通分量指向其他连通分量中的顶点时(就是没有出度)，我们就可以称之为一个汇；比如说上图的{G,H,I,J,K}就是一个汇。

## 图

**图**

G=(V,E),v是Vertex，E是Edge。

<Vj,Vi>。有向边也称弧（Arc），始点是弧尾（Tail），终点弧头（Head）；

**有向图（Digraph）顶点集和边集的表示：**

V（G1）={v1,v2,v3}

E(G1)={<v1,v2>,<v2,v1>,<v2,v3>}; //无向图不用写重复的

无向图（unDigraph）的度（Degree）：

**D(v)关联该顶点边的数目**

有向图是D(v)=ID(v)+OD(v);

**路径**：只要能从一个顶点到另个。如果除了开始和结尾有重复，则称是**简单路径，**若开始结尾相同就称**简单回路或简单环**

**无向图：**

Vj到Vi有路径称Vj和Vi连通，若V（G）任意连通即有路径称G为**连通图**（Connecte Graph）

极大连通子图是**连通分量**（Connected Component）。极大指走过节点多。即全部走未走过的节点

**有向图：**

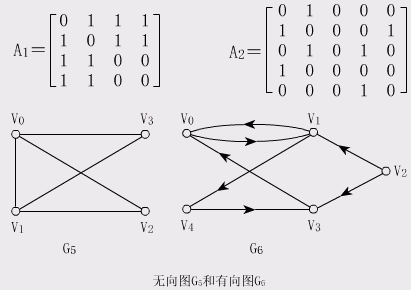
有根图：存在顶点v能到所有顶点，称图为有根图，v称为图的**根**

连通图有向时是**强连通图**

**图的储存结构**

1. **邻接矩阵表示法**

 　设G=(V，E)是具有n个顶点的图，则G的邻接矩阵是具有如下性质的n阶方阵：

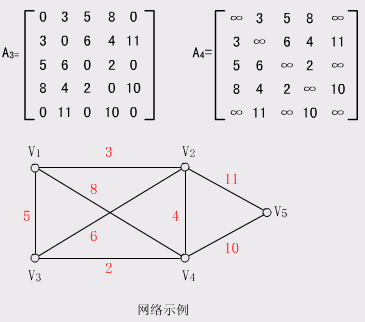


**总结：**

1、有几个顶点就有几行几列。做深度遍历时同邻接表在顶点(Vertex)的跳动，它是在不同行之间的跳动。

3．网络的邻接矩阵

若G是网络，则邻接矩阵可定义为：  
        
    
  
  其中：  
      wij表示边上的权值；  
      ∞表示一个计算机允许的、大于所有边上权值的数。  
【例】下面带权图的两种邻接矩阵分别为A3和A4。

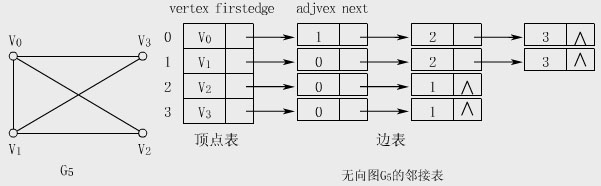


1. **邻接表表示**

把所有邻接于vi的顶点链接称链，称vi的**邻接表**。

无向图的邻接表为**边表**。

有向图是**出边表**。有向图只链接从某顶点的出边，逆邻接表是**入边表**



**邻接矩阵（Adjacency Matrix）的逻辑结构**

#define MaxVertexNum 100;

typedef char VertexType; //顶点类型

typedef int EdgeType; //权值类型

typedef struct{

VertexType vexs[MaxVertexNum];

EdeType edges[MaxVertexNum][ MaxVertexNum];

int n,e;

}

**邻接表的逻辑结构：**

typedef struct node{

int adijvex;

struct node \* next;

}EdgeNode; //边表节点

typedef struct vnode{

VertexType vertex;

EdgeNode \* firstedge;

}VertexNode; //顶点节点

typedef VertexNode AdjList[MaxVerexNum];

typedef struct{

AdjList adjlist;

int n,e; //n当前顶点数，e 边数

}ALGraph;

**图的遍历**

Typedef enum{TRUE,FALSE} Boolean;

Boolean visited[MaxVertexNum];

void DFSTtraverse(ALGraph \* G)

{

int i;

for(i=0;i<G->n;i++)

Visited[i]=FALSE;

for(i=0;i<G->n;i++)

If(!visited[i])

DFS(G,i);

}

void DFS(ALGraph \* G,int i)

{

EdgeNode \* p;

print(“vist vertex:%c”,G->adjlist[i].vertex);

visted[i]=TRUE;

p=G->adjlist[i].firstedge;

while(p)

{

if(!visted[p->adjvex])

DFS(G,p->adjvex);

p=p->next;

}

}

void DFSM(Mgraph \* G, int i)

{

int j;

printf(“visit vertex:%c”,G->vexs[i]);

visited[i]=TRUE;

for(j=0;j<G->n;j++)

if(G->edges[i][j]==1&&!visited[j])

DFSM(G,j)

}

**广度遍历(Breadth-First Traversal)**

void BFS(ALGraph \* G,int k)

{

int i;

CirQuenu Q;

EdgeNode \* p;

initQueue($=&Q);

printf(“visit vertex:%c”,G->adjlist[k].vertex);

visted[k]=TRUE;

EnQueue(&Q,k);

while(!Queue(&Q)){

i=DeQueue(&Q);

p=G->adjlist[i].firstedge;

while(p)

{

if(!visited[p->adjvex]){

printf(“visit vertex:%c”,G->adjlist[k].vertex);

visited[p->adjvex]=true;

EnQueue(&Q,p->adjvex);

}

p=p->next;

}

}

}

**生成树和最小生成树**

无回路的连通图，没确定根，也称**自由树**。

连通图的子图如果是包含G的所有顶点的树，称**生成树**(Spanning Tree)

实际生成树是有N个顶点和有n-1条边的极小连通子图。极小指边数最小。

深度遍历（DFS）：从v开始，搜索**第一个**邻接点w，从w开始重新DFS。若有未访问过的，**随便**选一个开始DFS。

深度优先生成树：

广度优先生成树：

**最小生成树**

Prim算法：

1. 从0开始，用虚线相连确定权最小用实线相连，依次到实线那头继续。。

Kruska算法：

## 多维数组和广义表

计算机内存是一维的。一般不对数组插入，采用顺序存储。

1. 行优先顺序：a11,a12…,a

LOC(aij)=LOC(a11)+[(i-1)\*n+j-1]\*d

1. 列优先:a11,a21..an1,a12,a22…an2….

**矩阵的压缩存储**

**对称矩阵（**对角线上有数据的**）**

N阶方阵 Aij=Aji;一般只存一半；压缩存储时

对角线上是aii.

**三角矩阵（**对角线上有数据的**）**

上三角矩阵：下三角（不包括主角线）均为常数；

**对角矩阵：**

**稀疏矩阵：**

1. 三元组表
2. 带行表的三元组表

**广义表（Lists）**

对线性和树的推广。

LS=（a1,a2,….an）; a1是表头，其余元素组成的表LS称表尾

A=(x,L)=(x,(a,b)) 或(x,L(a,b))

**空表 E=（）。不能运算。tail（（））=( );**

**tail( (( )) )=head( (( )) )=( )。(( ),( ))不对？**

1. 子表大写
2. 深度指单方向括号的层数。A是2
3. 线性表都是原子项。与树对应纯表。
4. head(LS) 原子和字表都行

广义表的图形表示：

**A( ) B(e) C(a,(b,c,d)) D(A(),B(e),C(a,(b,c,d)))**

**E((a,(a,b),((a,b),c)))**



纯表: 与树对应

再入表：允许节点再入

递归表：允许节点递归

广义表是线性表和树的推广

# Part 3 树和二叉树

常见的数组、链表、栈和队列都是线性结构，在存储大量数据时访问速度比较慢，而树(Tree)则是一种非线性结构，使得访问时间复杂度降低到O(logn)。

术语

* 度：节点的度指节点孩子数量。一棵树的度指最大孩子树。
* 叶子节点：没有子节点（即度为0），简称叶子。
* 阶：m阶B-tree指每个节点最多有m个子节点。
* 深度(h)=高度
* 森林(Forest)：由多棵不相交的树的集合。

哈希适合等于性的查找，树结构适合”范围查找“，lucene适合字符串的查找，，数据组织要选择合适的的算法。

## 3.1 二叉树

Binary Tree。树中节点的度不大于2的有序树。常见的性质：

* 第i层上至多有2i-1个节点
* 深度为h的二叉树中至多含有2h-1个节点
* 叶节点个数为n0，度数为2的节点个数为n2，则有: n0 = n2 + 1

证明：(1)n= n0+n1+ n2 (n指数量)(2)n= n1+2\* n2+1 算孩子的总数

### 3.1.1 Base

二叉树，森林，树的转换：

* 都去掉双亲到右孩子的线
* 二叉树—>右孩子到双亲
* 森林-->连接兄弟。双亲到右孩子
* 顺时针旋转45·

#### 存储结构

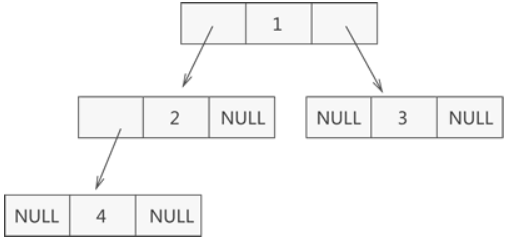
有两种，分别为顺序存储和链式存储

顺序存储

指的是使用顺序表（数组）存储二叉树。注意，顺序存储只适用于完全二叉树。如果我们想顺序存储普通二叉树，需要提前将普通二叉树转化为完全二叉树。

链式存储结构

采用链式存储，只需从树的根节点开始，将各个节点及其左右孩子使用链表存储即可。

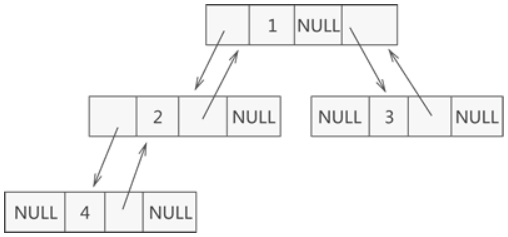


其节点结构由 3 部分构成：



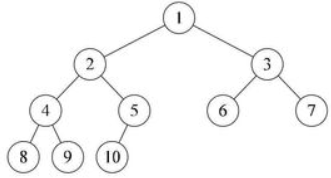
1. **typedef** **struct** BiTNode{
2. TElemType data;//数据域
3. **struct** BiTNode \*lchild,\*rchild;//左右孩子指针
4. **struct** BiTNode \*parent;
5. }BiTNode,\*BiTree;

某些场景需查找某节点的父节点"，可在节点结构中再添加一个指针域，指向其父亲节点。这样的链表结构，通常称为三叉链表。可轻松找到各节点的父节点。



### 3.1.1 完全二叉树

* 只有最下面的两层结点度数小于2，其余各层的结点度数都等于2，
* 所有叶子节点都在最下一层且集中在最左边。



性质：

* 树的高度h=log2n
* 非根节点的父节点的下标为(i-1)/2;
* 2i+1 <= n-1,则节点i的左子结点的下标为2i+1;否则没有左子结点.
* 2i+2 <= n-1,则节点i的右子节点的下标为2i+2;否则没有右子节点

O(logN)

### 3.1.2 满二叉树

完全二叉树的特例

特性

* 节点数n=2^k-1
* 节点i，左子节点编号是2i，而右子节点是2i+1

## 3.2 二叉树的运算

### 3.2.1 二叉树的生成

### 3.2.2 二叉树的遍历

遍历算法

二叉树的遍历

* 先序遍历：先访问根节点，然后访问左节点，最后访问右节点(根->左->右)
* 中序遍历：先访问左节点，然后访问根节点，最后访问右节点(左->根->右)
* 后序遍历：先访问左节点，然后访问右节点，最后访问根节点(左->右->根)

## 3.3 线索二叉树

### 二叉树的线索化

### 二叉线索链表上的运算

## 3.4 树和森林

### 树的存储结构

### 树、森林与二叉树的转换

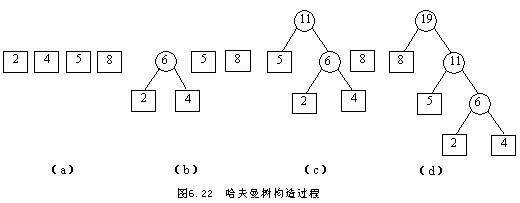
### 树和森林的遍历

## 3.5 哈夫曼树及其应用

### 3.5.1 最优二叉树(哈夫曼树)

在权为w1，w2…..的n个**叶子**所构成的二叉树带权路径长度最小的二叉树

1. n个节点，每个都是孤立的节点。经过n-1次合并产生n-1个节点共2n-1个



遍历：

### 3.5.2 哈夫曼编码

# Part 4 排序

基本概念

内部排序：排序过程都放在内存不涉及内外存交换。大多数算法的开销是记录的比较和记录的移动，算法所需辅助空间不依赖于问题的规模n，辅助空间是O（1）。又称就地排序。

外部排序：指记录数量很大，内存一次不能容纳全部记录，需辅助空间。

算法评价标准

时间开销及需要的附加空间。另外算法本身的复杂度也是考虑的重要因素。

时间开销：关键字比较次数和记录移动次数。

本章讨论中，都假定顺序表作为存储结构

1. #define MAXSIZE 100
2. **typedef** **int** KeyType;
3. **typedef** **struct**{
4. KeyType key;
5. InfoType otherinfo;
6. } RecType;
7. **typedef** RecType SeqList[MAXSIZE+1];//0元素空着或用作哨兵单元
8. SeqList R;  //R为待排序的记录文件

## 插入排序

基本思想：每次将待排序的记录插入前面已排序文件的适当位置。

### 4.1.1 直接插入排序

1. public **void** insertSort(**int**[] arr) {
2. **int** j; // 已排序列表下标
3. **for** (**int** i = 1; i < arr.length; i++) {
4. **if** (arr[i] < arr[i - 1]) {
5. **int** temp = arr[i];
6. **for** (j = i - 1; j >= 0 && temp < arr[j]; j--) {
7. arr[j + 1] = arr[j]; //后移
8. }
9. arr[j + 1] = temp; // 插入待排序元素
10. }
11. }
12. }

### 4.1.2 希尔排序

分组的的插入排序

void shellSort(Seqlist R)

{

int increment=1;

do{

increment=increment/3+1;

shellPass(R,increment);

}while(increment>1)

}

void shellPass(Seqlist R,int d)

{

int i,j;

for(i=d+1;i<=n;i++)

{

a[0]=a[i]; //搬走后留下空位置

j=i-d;

while(a[0]<a[j]&&j>0)

{

a[j+d]=a[j]; //搬走后留下空位置

j=j-d;

}

a[j+d]=a[0];

}

}

## 4.2 交换排序

两两比较待排序记录的关键字，若次序相反即进行交换，直到所有记录都没有反序为止。

### 冒泡排序

1. **public** **void** bubbleSort(**int**[] arr) {
2. **for** (**int** i = 1; i < arr.length; i++) {
3. **boolean** flag = **true**;//若为true，则表示此次循环没有进行交换
4. **for** (**int** j = 0; j < arr.length - i; j++) {
5. **if** (arr[j] > arr[j + 1]) {
6. **int** tmp = arr[j];
7. arr[j] = arr[j + 1];
8. arr[j + 1] = tmp;
9. flag = **false**;
10. }
11. }
12. **if** (flag) **break**;
13. }
14. }

最好情况：比较次数：n-1； 移动：0

### 快速排序

改进冒泡排序

## 4.3 选择排序

### 直接选择排序

每次查找从所有无序区中找到最小的。第i趟排序需n-i次比较，总比较次数：

n(n-1)/2=O(n`2)

### 堆排序

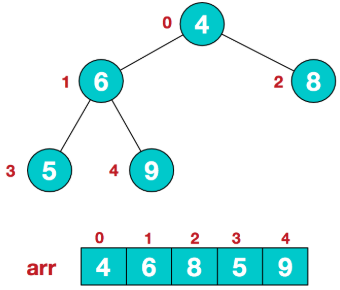
[图解排序算法(三)之堆排序](https://www.cnblogs.com/chengxiao/p/6129630.html)

堆排序是利用堆这种数据结构而设计的一种排序算法

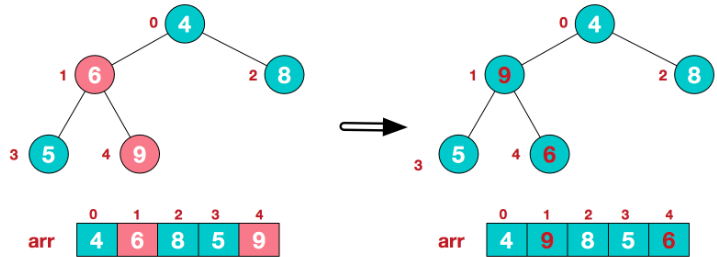
堆排序基本思想及步骤

步骤一 构造大顶堆。（升序用大顶堆，降序用小顶堆)。

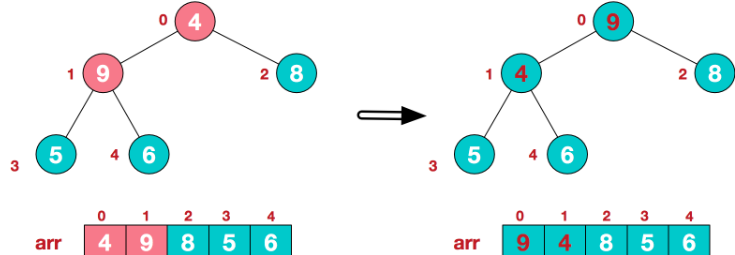
a.假设给定无序序列结构如下



此时我们从最后一个非叶子结点开始（第一个非叶子结点 arr.length/2-1=5/2-1=1），从左至右，从下至上进行调整



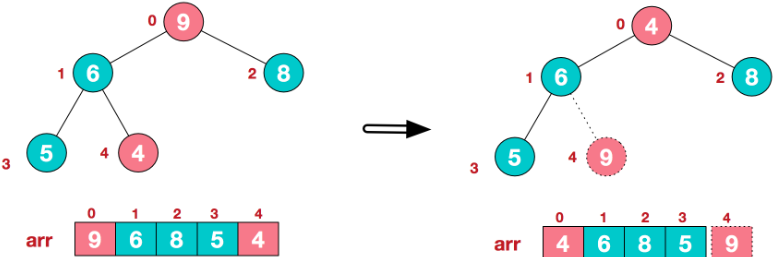
找到第二个非叶节点4，由于[4,9,8]中9元素最大，4和9交换



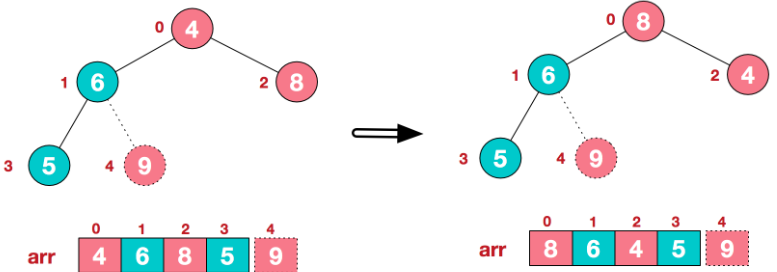
此时，我们就将一个无需序列构造成了一个大顶堆。

步骤二 将堆顶元素与末尾元素进行交换，使末尾元素最大。然后继续调整堆，再将堆顶元素与末尾元素交换，得到第二大元素。如此反复进行交换、重建、交换。

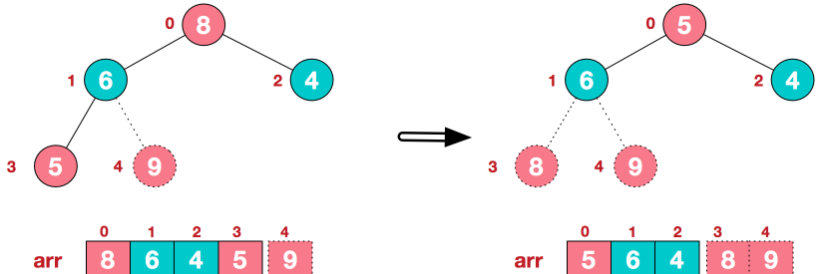
a.将堆顶元素9和末尾元素4进行交换



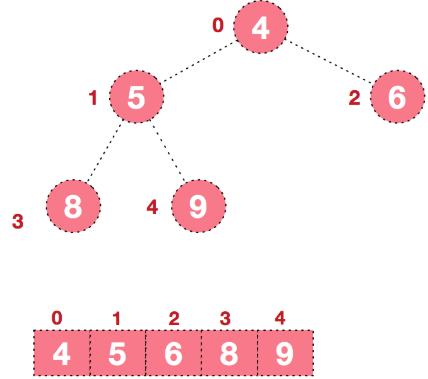
b.重新调整结构，使其继续满足堆定义



c.再将堆顶元素8与末尾元素5进行交换，得到第二大元素8.

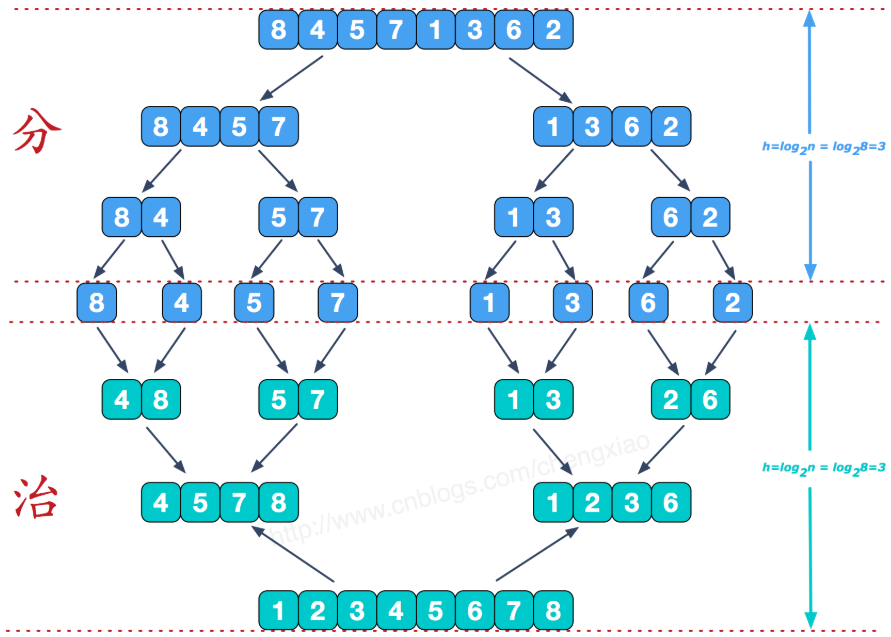


后续过程，继续进行调整，交换，如此反复进行，最终使得整个序列有序



## 4.4 归并排序

[图解排序算法(四)之归并排序](https://www.cnblogs.com/chengxiao/p/6194356.html)



最简单的方式两个有序子文件放在R[low…m],R[m+1…high]合并到R1，复制回R[low,high].如

Void merge(SeqList R,int low,int m,int high)

{

Int i=low,j=m+1;p=0;

RecType \* R1;

while(i<=m&&j<=high>

R1[p++]=(R[i].key<=R[j].key)?R[i++]:R[j++]//R指向R1。

}

注意：申请的空间较大，加判断。

归并排序的两种实现：

自底向上和自顶向下。即为**二路归并排序**。把R[1..n]看成长度是1的顺序子文件顺序两两相邻归并。在某趟归并，子文件长度length(最后一个字文件可能小于)，归并前有[n/length]个子文件，R[1..length],R[length+1…..2length],…(R[n/length]-1)\*length+1…n]

## 4.5 分配排序

### 桶排序

Bucket Sort，也称箱排序（Bin Sort）

箱个数取决关键字取值范围，R[0…n-1],分配时间O（n），收集时间O（m）或O（m+n）

1箱子容量不可预知，2保持稳定故箱子采用**链队列**

注意：适用于关键字取值范围较小。

箱排序的变种：桶排序，假设输入n个关键字是分布在[0,1）上，找某个合适的数映射到此区间。区间是一个桶。指针B[0…n-1]表示n个桶，记录0<=R[i]<=n-1通过n\*(R[i].key)

### 基数排序

## 4.6 内部排序方法的分析比较

# Part 5 查找

查找算法

顺序查找：复杂度O(n)，缺点：的算法在数据量很大时显然是糟糕的

二分查找（binary search）：要求被检索数据有序。

二叉树查找（binary tree search）等。，只能应用于二叉查找树上

## 5.1 顺序表的查找

### 顺序查找

二分法查找（Binary Search）折半查找：

1. //R[1……..n]
2. **int** BinarySearch(SeqList R,Key K){
3. **int** low=1,high=n,mid;
4. **while**(low<=high){
5. mid=(low+high)/2;
6. **if**(R[mid].key==K)
7. **return** mid;
8. **if**(R[mid].key>K)
9. high=mid-1;
10. **else**
11. low=mid+1;
12. }
13. **return** 0;
14. }

### 二分查找

### 索引顺序查找

### 三种查找方法的比较

分块查找

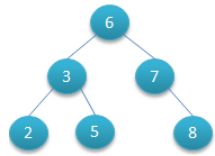
每块不一定有序，但分块有序。将[1..n]分为n块，前b-1快都有[n/b];抽取每块最大关键字组成ID[1..b];由于块内无序，只能用顺序查找。

## 5.2 树的查找

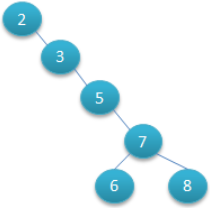
### 5.2.1 二叉查找树(BST)

Binary Search Tree,又叫二叉排序树（Binary Sort Tree）、二叉搜索树。

左子树<根<右子树，中序遍历得到有序递增序列。通常采用二叉链表作为存储结构。



二叉查找树可以任意地构造，同样是这几个数字，也可以按照下图的方式来构造：



但是这棵二叉树的查询效率就低了。因此若想二叉树的查询效率尽可能高，需要这棵二叉树是平衡的，从而引出新的定义——平衡二叉树，或称AVL树

存储结构

顺序存储结构

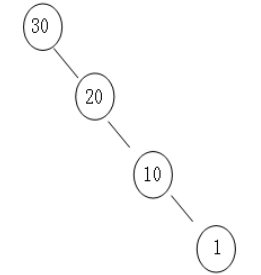
链式存储结构：

1. **typedef** **struct** BinTNode{
2. **char** data;
3. BinTNode \* lchild,\* rchild;
4. }BinTNode;
5. **typedef** BinTNode \* BinTree;

### 5.2.2 平衡二叉树（AVL）

Balanced Binary Tree，又称AVL（Adelson-Velskii and Landis）树，是带有平衡条件（balance condition）的二叉查找树。任一节点的左右子树最大高度差为1，因此也称高度平衡树

二叉查找树在最坏的情况下会出现“链表”的形式，复杂度退化到O(N)，比如下图。



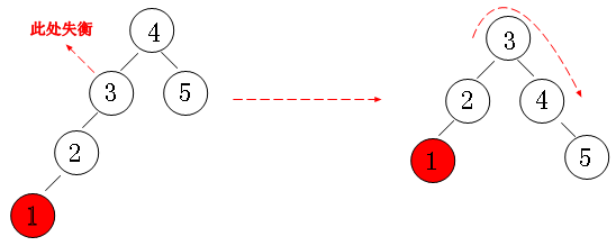
树节点旋转成完全二叉树，做到查找是严格的O(logN)。

旋转

节点再怎么失衡都逃不过4种情况

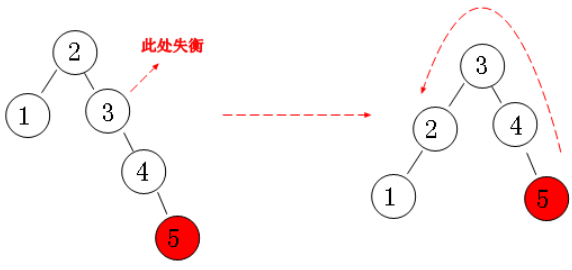
1）左左型(左子树的左边)

左子树的左边节点处添加节点。如图追加节点1时，导致了“节点3"失衡。把这棵树比作齿轮，我们在“节点5”处把齿轮往下拉一个位置，也就变成了后面这样“平衡”的形式。

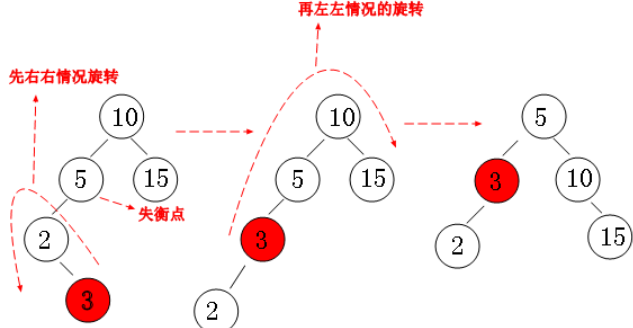


2）右右型(右子树的右边节点)

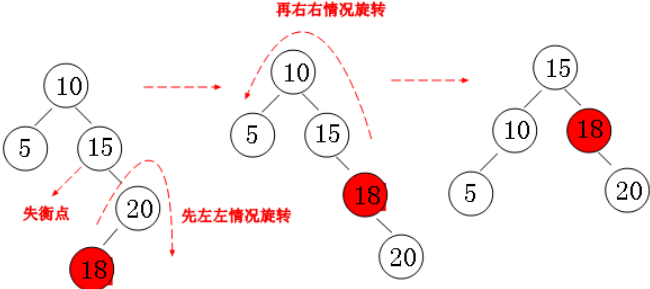
和左左型类似，节点1的地方将树往下拉一位，形成平衡效果。



3）左右型(左子树的右边节点)



4）右左型(右子树的左边节点)



[6天通吃树结构—— 第二天 平衡二叉树](https://www.cnblogs.com/huangxincheng/archive/2012/07/22/2603956.html)

### 5.2.3 B-Tree

即Balanced Tree，B-Tree是一般化的平衡二叉树，一个节点可以拥有2个以上的子节点(度的大小取决于磁盘大小)，适用于读写相对大的数据块的存储系统，例如磁盘。B树减少定位记录时所经历的中间过程，从而加快存取速度。这种数据结构常被应用在数据库和文件系统，比如MongoDB，innodb。B-Tree有一些变体，如B+树和B\*树。

一棵m阶的B-Tree有如下特性：

1. 每个节点最多有m个孩子。

2. 除了根节点和叶子节点外，其它每个节点至少有Ceil(m/2)个孩子。

3. 若根节点不是叶子节点，则至少有2个孩子

4. 所有叶子节点都在同一层，且不包含其它关键字信息

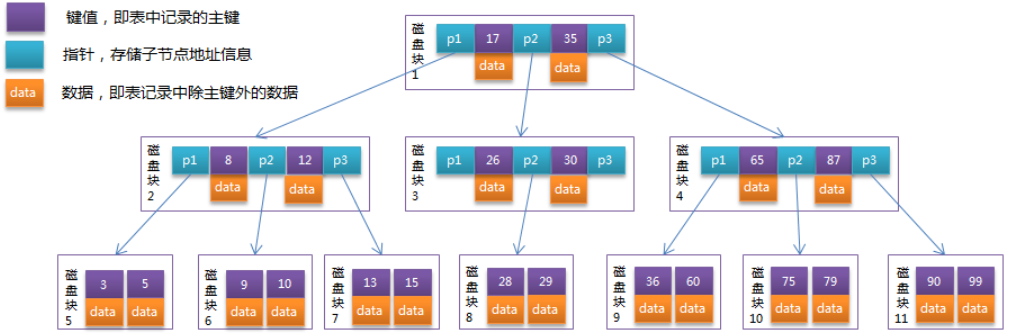
5. 每个非终端节点包含n个关键字信息（P0,P1,…Pn, k1,…kn）

6. 关键字的个数n满足：ceil(m/2)-1 <= n <= m-1

7. ki(i=1,…n)为关键字，且关键字升序排序。

8. Pi(i=1,…n)为指向子树根节点的指针。P(i-1)指向的子树的所有节点关键字均小于ki，但都大于k(i-1)

B-Tree中的每个节点根据实际情况可以包含大量的关键字信息和分支，如图一个3阶的B-Tree：



每个节点占用一个盘块的磁盘空间，一个节点上有两个升序排序的关键字和三个指向子树根节点的指针，指针存储的是子节点所在磁盘块的地址。两个关键词划分成的三个范围域对应三个指针指向的子树的数据的范围域。

模拟查找关键字29的过程：

1. 根据根节点找到磁盘块1，读入内存。【磁盘I/O操作第1次】
2. 比较关键字29在区间（17,35），找到磁盘块1的指针P2。
3. 根据P2指针找到磁盘块3，读入内存。【磁盘I/O操作第2次】
4. 比较关键字29在区间（26,30），找到磁盘块3的指针P2。
5. 根据P2指针找到磁盘块8，读入内存。【磁盘I/O操作第3次】
6. 在磁盘块8中的关键字列表中找到关键字29。

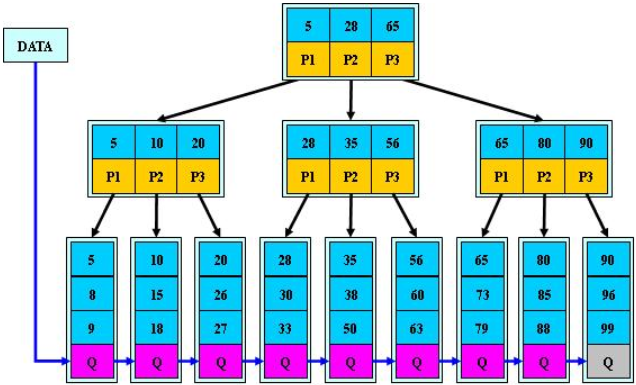
分析上面过程，发现需要3次磁盘I/O操作，和3次内存查找操作。由于内存中的关键字是一个有序表结构，可以利用二分法查找提高效率。而3次磁盘I/O操作是影响整个B-Tree查找效率的决定因素。B-Tree相对于AVLTree**缩减了节点个数**，使每次磁盘I/O取到内存的数据都发挥了作用，从而提高了查询效率。

### 5.2.4 B+Tree

B+Tree是B树基础上为了更充分的利用结点空间，让遍历查询速度更稳定而扩展的结构，它规定只在叶子节点存数据，非叶子结点只存索引，使内部节点的度数尽可能的大。且叶子结点用一个链表连接起来。特性：

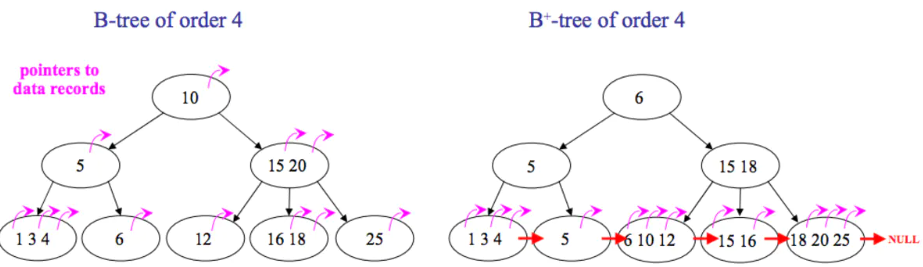
* B+树的非叶子节点不保存关键字记录的指针，这样使得B+树每个节点所能保存的关键字大大增加；
* B+叶子节点保存了父节点的所有关键字和关键字记录的指针，每个叶子节点的关键字从小到大链接；
* B+树的根节点关键字数量和其子节点个数相等;
* B+的非叶子节点只进行数据索引，不存实际的数据，所有数据地址必须要到叶子节点才能获取到，所以每次数据查询的次数都一样；

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;



区别

Btree非叶子节点可存数据记录



为什么说B+tree比B树更适合实际应用中操作系统的文件索引和数据库索引？

* B+tree的磁盘读写代价更低

B+Tree的内部结点没有具体信息，一个盘块所能容纳的更多关键字。一次性读入内存中的需要查找的关键字也就越多。相对来说IO读写次数也就降低了。

举个例子，假设磁盘中的一个盘块容纳16bytes，而一个关键字2bytes，一个关键字具体信息指针2bytes。一棵9阶B-tree(一个结点最多8个关键字)的内部结点需要2个盘快。而B+ 树内部结点只需要1个盘快。当需要把内部结点读入内存中的时候，B 树就比B+ 树多一次盘块查找时间(在磁盘中就是盘片旋转的时间)。

* B+tree的查询效率更加稳定

由于非叶子结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。

* B树在提高了磁盘IO性能的同时并没有解决元素遍历的效率低下的问题。正是为了解决这个问题，B+树应运而生。B+树只要遍历叶子节点就可以实现整棵树的遍历。而且在数据库中基于范围的查询是非常频繁的，而B树不支持这样的操作（或者说效率太低）。

### 5.2.5 红黑树

红黑树是一种自平衡的二叉查找树。除了符合BST基本特性，还有下列特性：

* 节点是红色或黑色。
* 根节点是黑色。
* 每个叶子节点都是黑色的空节点（NIL节点）。
* 每个红色节点的两个子节点都是黑色。(路径上不能有两个连续红色节点)
* 从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。

TreeMap

* 对key进行自然排序，
* 添加的对象需要实现Comparable接口（jdk自带的类型已实现）
* TreeMap内部基于红黑树排序

## 5.3 散列表1的查找

### 5.3.1 散列表的概念

菜单递归查找

Find all root menu

1. **public** **void** buildMenu() {
2. List<Menu> allMenus = **new** ArrayList<Menu>();//dao.getAllMenus();
3. List<Menu> parents = **new** ArrayList<Menu>();
4. //find all 1 level menu
5. **for** (Menu menu : allMenus) {
6. **if** (menu.getParentId() == **null**) {
7. parents.add(menu);
8. }
9. }
10. **for** (Menu menu : parents) {
11. menu.setChildren(getChild(menu.getMenuId(),allMenus));
12. }
13. }

getChild

1. **public** List<Menu> getChild(**long** menuId,List<Menu> menuList) {
2. List<Menu> childList = **new** ArrayList<Menu>();
3. **for** (Menu menu : menuList) {
4. **if** (menu.getParentId().equals(menuId)) {
5. childList.add(menu);
6. }
7. }
8. //递归终止条件
9. **if** (childList.size()==0) {
10. **return** **null**;
11. }
12. //开始递归
13. **for** (Menu menu : childList) {
14. menu.setChildren(menu.getChild(menu.getMenuId(),menuList));
15. }
16. **return** childList;
17. }