Data Structure

Vine

2022 年 8 月 8 日

目录

1 绪论 **4**

2 线性表 **5**

3 栈和队列 **6**

4 串 **7**

5 数组和广义表 **8**

6 树和二叉树 **9**

7 图 **10**

7.1 定义 10
7.2 图的存储结构 10
7.2.1 数组表示法 11
7.2.2 邻接表 (Adjacency List) 表示法 12
7.2.3 十字链表 (Orthogonal List) 12
7.2.4 邻接多重表 (Adjacency Multlist) 13
7.3 图的遍历 13
7.3.1 深度优先搜索 13
7.3.2 广度优先搜索 14
7.4 图的连通性问题 14
7.4.1 无向图的连通分量和生成树 14
7.4.2 有向图的强连通分量 15
7.4.3 最小生成树 15
7.4.4 关节点和重连通分量 16
7.5 有向无环图及其应用 17
7.5.1 拓扑排序 17
7.5.2 关键路径 18
7.6 最短路径 18
7.6.1 从某个源点到其余各顶点的最短路径 18
7.6.2 每一对顶点之间的最短路径 18
7.6.3 关键路径 18

8 动态存储管理 **19**

9 查找 **20**

9.1 静态查找表 20
9.1.1 顺序表的查找 20
9.1.2 有序表的查找 20
9.1.3 静态树表的查找 21
9.1.4 索引顺序表的查找 22
9.2 动态查找表 22
9.2.1 二叉排序树和平衡二叉树 22
9.2.2 B-树和 B+ 树 27

9.2.3 键树 28
9.3 哈希表 29
9.3.1 哈希表 29
9.3.2 哈希函數構造方法 29
9.3.3 衝突處理方法 29
9.3.4 哈希表的查找及其分析 30

10 内部排序 32
10.1 概述 32
10.2 插入排序 32
10.2.1 直接插入 32
10.2.2 其他插入 32
10.3 快速 33
10.3.1 起泡排序 33
10.3.2 快速排序 33
10.4 选择排序 34
10.4.1 简单选择排序 34
10.4.2 树形排序 34
10.4.3 堆排序 34
10.5 归并排序 34
10.6 基数排序 35
10.6.1 多关键字的排序 35
10.6.2 链式基数排序 35
10.7 各内部排序方法的比较讨论 36

11 外部排序 37

12 文件 38

1 绪论

2 线性表

3 栈和队列

4 串

5 数组和广义表

6 树和二叉树

7 图

7.1 定义

顶点 (Vertex)

弧 (Arc),$< v, w >$ $v \longrightarrow w$

有向图 (Digraph),$G_1 = (\{v_1, v_2, v_3\}, \{< v_1, v_2 >, < v_1, v_3 >, < v_2, v_3 >\})$

边 (Edge),(v, w) $v - w$

无向图 (Undigraph),$G_1 = (\{v_1, v_2, v_3\}, \{(v_1, v_2), (v_1, v_3), (v_2, v_3)\})$

通路 $P(v, w)$

顶点数目 n

弧或边数目 e, 无向完全图 e=$\frac{1}{2}n(n-1)$, 有向完全图 e=$n(n-1)$,

稀疏图，稠密图

$G = (V, \{E\}), G^{'} = (V^{'}, \{E^{'}\}), if \quad V^{'} \subseteq V, W^{'} \subseteq W, G^{'}为G子图$

无向图 $(v, v^{'}) \in E, \quad v, v^{'}$ 互为邻接点。$(v, v^{'})$ 依附于顶点。$(v, v^{'})和v, v^{'}$ 相关联

无向图顶点的度 (和顶点关联的边的数目)

无向图路径，顶点序列 $(v = v_{i,0}, v_{i,1}, \ldots, v_{i,m} = v^{'}), (v_{i,j-1}, v_{i,j}) \in E, 1 \leqslant j \leqslant m$

路径，连通，连通图

连通分量，极大连通子图

连通图的生成树，极小连通子图，全顶点，n-1 边.(n 点，n-1 边不一定是生成树)

非连通图 $n, e < n-1$

存在环 $n, e > n-1$

有向图 $< v, v^{'} > \in A, \quad v$ 邻接到 $v^{'}$ 。$< v, v^{'} >$ 和$v, v^{'}$ 相关联。

顶点的入度 (以顶点为头的弧的数目，InDegree),ID。顶点的出度 (以顶点为尾的弧的数目，OutDegree),OD。

有向图顶点的度 $TD = ID + OD$

有向图路径，顶点序列 $(v = v_{i,0}, v_{i,1}, \ldots, v_{i,m} = v^{'}), < v_{i,j-1}, v_{i,j} > \in A, 1 \leqslant j \leqslant m$

强连通图

强连通分量

有向树，有向图恰好有一个顶点入度为 0，其余顶点入度为 1

有向图的生产森林，若干棵有向树，含有全部顶点，只有足以构成若干不相交的有向树的弧

边或者弧 $e = \frac{1}{2}\sum_{i=1}^{n} TD(v_i)$

路径的长度，路径上边或者弧的数目

回路或环，路径上首尾点相同

简单路径，路径上顶点不重复

简单回路或环，除首尾点，路径上顶点不重复

顶点在图中位置，人为的随意排列中的位置

7.2 图的存储结构

树的最大和最小度差别很大

7.2.1 数组表示法

```
//-----图的数组(邻接矩阵)存储表示------
#define INFINITY  INT_MAX
#define MAX_VERTEX_NUM 20
typedef enum{DG,DN,UDG,UDN} GraphKind;
typedef struct ArcCell{
    VRType adj;         //VRType 顶点关系类型, 对无权图用1或0表示是否相邻, 带权图则为权值
    InfoType *info;     //弧相关信息的指针
}ArcCell, AdjMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM];

typedef struct{
    VertexType ves[MAX_VERTEX_NUM]; //顶点向量
    AdjMatrix arcs;                 //邻接矩阵
    int     vexnum,arcnum;          //图的当前顶点数和弧数
    GraphKind kind;                 //图的种类标志
}MGraph;
```

无向图，$TD(v_i)=\sum_{j=0}^{n-1}A[i][j],(n=MAX\_VERTEX\_NUM)$

有向图，$TD(v_i)=OD(v_i)+ID(v_i)=\sum_{j=0}^{n-1}A[i][j]+\sum_{i=0}^{n-1}A[i][j],,(n=MAX\_VERTEX\_NUM)$

网，$A[i][j]=\begin{cases} w_{i,j}, & \text{若} < v_i,v_j > or(v_i,v_j)\in VR \\ \infty, & \text{反之} \end{cases}$

构造无向网复杂度 $o(n^2+e\cdot n)$, 初始化耗费 $o(n^2)$

```
Status CreateGraph(MGraph &G){
    scanf(&G.kind);
    switch(G.kind){
        case DG: return CreateDG(G);
        case DN: return CreateDN(G);
        case UDG: return CreateUDG(G);
        case UDN: return CreateUDN(G);
        default :return ERROR;
    }
}//GreateGraph

Status CreateUDN(MGraph &G){
    //采用数组(邻接矩阵)表示法, 构造无向网G
    scanf(&G.vexnum,&G.arcnum,&IncInfo);        //IncInfo 为0则各弧不含其他信息
    for(i=0;i<G.vexnum;++i) scanf(&G.vexs[i]);  //构造顶点向量
    for(i=0;i<G.vexnum;++i)                     //初始化邻接矩阵
        for(j=0;j<G.vexnum;++j) G.arcs[j][j]={INFTY,NULL};      //{adj,info}
    for(k=0;k<G.arcnum;++k){                    //构造邻接矩阵
        scanf(&v1,&v2,&w);                      //输入一条边依附的顶点及权值
        i=LocateVex(G,v1); j=LocateVex(G,v2);   //确定v1和v2在G中位置,在数组
        G.arcs[i][j].adj=w;
        if(IncInfo) Input(* G.arcs[i][j].info)  //若弧含有相关信息, 则输入
        G.arcs[j][i]=G.arcs[i][j];              //置<v1,v2>的对称弧<v2,v1>
    }
    return OK;
}//CreateUDN
```

7.2.2 邻接表 (Adjacency List) 表示法

```
//------图的邻接表存储表示------
#define MAX_VERTEX_NUM 20
typedef struct ArcNode{
    int adjvex;                 //该弧所指向的顶点的位置
    struct ArcNode *nextarc;    //指向下一条弧的指针
    InfoType *info;             //该弧相关信息的指针
}ArcNode;

typedef struct VNode{
    VertexType data;            //顶点信息
    ArcNode * firstarc;         //指向第一条依附该顶点的指针
}VNode,AdjList[MAX_VERTEX_NUM];

typedef struct{
    AdjList vetices;
    int vexnum,arcnum;           //图的当前顶点和弧数
    int kind;                    //图的种类标志
}ALGraph;
```

n 边 2e 顶点 $e \ll \frac{n(n+1)}{2}$ 时比邻接表矩阵节省
出度，第 i 各链表顶点数
入度，遍历链表
逆邻接表
建立邻接表时间复杂度 $o(n+e)$ 或者 $o(n \cdot e)$

7.2.3 十字链表 (Orthogonal List)

有向图的存储结构

```
//--------有向图的十字链表表示-------
#define MAX_VERTEX_NUM 20
typedef struct ArcBox{
    int tailvex,headvex;            //该弧的尾和头顶点的位置
    struct ArcBox *hlink,*tlink;    //分别为弧头相同和弧尾相同的弧和链域
    InfoType    *info;              //该弧相关信息的指针
}ArcBox;

typedef struct VexNode{
    VertexType data;
    ArcBox     *firstin,*firstout; //分别指向该顶点第一条入弧和出弧
}VexNode;

typedef struct{
    VexNode xlist[MAX_VERTEX_NUM];  //表头向量
    int vexnum,arcnum;              //有向图的当前顶点数和弧数
}OLGraph;

Status CreateDG(OLGraph &G){
    //采用十字链表存储表示，构造有向图G
    scanf(&G.vexnum,&G.arcnum,&G.IncInfo);                   //IncInfo为0则各弧不含其他信息
    for(i=0;i<G.vexnum;++i){                                 //构造表头向量
        scanf(&G.xlist[i].data);                             //输入顶点值
        G.xlist[i].firstin=NULL;G.xlist[i].firstout=NULL;    //初始化指针
    }
    for(i=0;i<G.arcnum;++k){                                 //输入各弧并构造十字链表
        scanf(&v1,&v2);                                      //输入一条弧的始点和终点
        i=LocateVex(G,v1);j=LocateVex(G,v2);                 //确定v1,v2位置
        p=(ArcBox *) malloc (sizeof(ArcBox));                //假定有足够空间
        *p={i,j,G.xlist[j].firstin,i,G.xlist[i].firstout,NULL }; //对弧顶点赋值
        G.xlist[j].firstin=G.xlist[i].firstout=p;             //完成在入弧和出弧链头的插入
        if(IncInfo) Input(*p->info);                          //若弧含有相关信息，则输入
    }
}//CreateDG
```

7.2.4 邻接多重表 (Adjacency Multlist)

无向图的存储结构

```
//----无向图的邻接多重表存储表示------
#define MAX_VERTEX_NUM 20;
typedef enum{unvisited,visited} VisitIf;
typedef struct EBox{
    VisitIf  mark;              //访问标记
    int      ivex,jvex;         //该边依附的两个顶点的位置
    struct EBox  *link,*jlink   //分别指向依附这两个顶点的下一条边
}EBox;

typedef struct VexBox{
    VertexType data;
    EBox *firstedge;            //指向第一条依附该顶点的边
}VexBox;

typedef struct{
    VexBox  adjmulist[MAX_VERTEX_NUM];
    int  vexnum,edgenum;                //无向图的当前定点数
}AMLGraph;
```

7.3 图的遍历

从图的某一顶点出发，访遍图的每一顶点，每个顶点仅被访问一次
辅助数组 $visited[0 \ldots n-1]$

7.3.1 深度优先搜索

树的先根遍历

```
Boolean visited[MAX];           //访问标志数组
Status (* VisitFunc)(int v);    //函数变量

void DFSTraverse(Graph G,Status (* Visit)(int v)){
    //对图作深度优先遍历
    VisitFunc=Visit;                            //使用全局变量VisitFunc,使DFS不必使用函数指针参数
    for(v=0;v<G.vexnum;++v) visited[v]=FALSE;   //访问标志数组初始化
    for(v=0;v<G.vexnum;++v)
        if(!visited[v]) DFS(G,v);               //对尚未访问的顶点调用DFS
}

void DFS(Graph G,int v){
    //从第v个顶点出发递归地深度优先遍历图G
    visited[v]=TRUE;VisitFunc(v);                       //访问第v个顶点
    for(w=FirstAdjVex(G,v);w>=0; w=NextAdjVex(G,v,w))
        if(!visited[w]) DFS(G,w);                       //对v的尚未访问的邻接点w递归调用DFS
}
```

二维数组表示，查找邻接点时间 $o(n^2)$
邻接表存储，查找邻接表时间 $o(e)$, 深度优先搜索遍历时间 $o(n+e)$

7.3.2 广度优先搜索

树的层次遍历

```
void BFSTraverse(Graph G,Status (* Visit)(int v)){
    //按广度优先非递归遍历图G。使用辅助队列Q和访问标志数组visite。
    for(v=0;v<G.vexnum;++v) visited[v]=FALSE;
    InitQueue(Q);                               //置空的辅助队列Q
    for(v=0;v<G.vexnum;++v)
        if(!visited[v]){                        //v尚未访问
            visited[v]=TRUE;Visit(v);
            EnQueue(Q,v);                       //v入队列
            while(!QueueEmpty(Q)){
                DeQueue(Q,u);                   //队头元素出队并置为u
                for(w=FirstAdjVex(G,u);w>=0;w=NextAdjVex(G,u,w))
                    if(!visited[w]){            //w为u的尚未访问的邻接点
                        visited[w]=TRUE; Visit(w);
                        EnQueue(Q,w);
                    }
            }
        }
}//BFSTraverse
```

7.4 图的连通性问题

7.4.1 无向图的连通分量和生成树

连通图 G，所有边的集合 E(G), 遍历所得边集合 T(G), 剩余边集合 B(G)
连通图的生成树包含 T(G),G 中所有顶点 (极小连通子图)
深度优先生成树

非连通图，非连通图的生成森林包含 $T(G)_i, G_i \quad \sum T(G)_i = T(G), \sum G_i = G$
深度优先生成森林
孩子兄弟链表

```
void DFSForest(Graph G,CSTree &T){
    //建立无向图G的深度优先生成森林的孩子兄弟链表T
    T=NULL;
    for(v=0;v<G.vexnum;++v) visited[v]=FALSE;
    for(v=0;v<G.vexnum;++v){
        if(!visited[v]){                        //第v顶点为新的生成树的根结点
            p=(CSTree) malloc(sizeof(CSTree));  //分配根结点
            *p={GetVex(G,v),NULL,NULL};         //结点赋值
            if(!T) T=p;                         //是第一课生成树的根
            else q->nextsibling=p;              //是其他生成树的根
            q=p;                                //q指向当前生成树的根
            DFSTree(G,v,p);                     //建立以p为根的生成树
        }
    }
}

void DFSTree(Graph G,int v,CSTree &T){
    //从第v个顶点出发深度优先遍历图G，建立以T为根的生成树
    visited[v]=TRUE;first=TRUE;
    for(w=FirstAdjVex(G,b);w>=0;w=NextAdjVex(G,v,w))
        if(!visited[w]){
            p=(CSTree) malloc(sizeof(CSTree));  //分配孩子结点
            *p={GetVex(G,w),NULL,NULL};

            if(first) T->lchild=p;first=false;  //w是v的第一个未被访问的邻接点，是根的左孩子节点
            else q->nextsibling=p;              //w是v的其他未被访问的邻接点，是上一邻接点的右兄弟节点

            q=p;
            DFSTree(G,w,q)                      //从第w个顶点出发深度优先遍历图G，建立生成子树q
        }
}
```

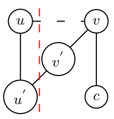
7.4.2 有向图的强连通分量

```
//--------step1 get finished数组------------
void DFSTraverse1(Graph G,Status (* Visit)(int v)){
    //对图作深度优先遍历
    count=0;
    VisitFunc=Visit;                            //使用全局变量VisitFunc,使DFS不必使用函数指针参数
    for(v=0;v<G.vexnum;++v) visited[v]=FALSE;   //访问标志数组初始化
    for(v=0;v<G.vexnum;++v)
        if(!visited[v]) DFS1(G,v);               //对尚未访问的顶点调用DFS
}

void DFS1(Graph G,int v){
    //从第v个顶点出发递归地深度优先遍历图G
    visited[v]=TRUE;VisitFunc(v);                       //访问第v个顶点
    for(w=FirstAdjVex1(G,v);w>=0; w=NextAdjVex1(G,v,w))  //以v为弧尾深度优先
        if(!visited[w]) DFS1(G,w);                        //对v的未访问的邻接点w递归调用DFS
        finished[count++]=v;
}

//--------step1 use finished数组---------

void DFSTraverse2(Graph G,Status (* Visit)(int v)){
    //用finished数组对图作逆向深度优先遍历
    flag=0;                                     //记录第flag个分量
    VisitFunc=Visit;                            //使用全局变量VisitFunc,使DFS不必使用函数指针参数
    for(v=0;v<G.vexnum;++v) visited[v]=FALSE;   //访问标志数组初始化
    for(v=finished[num],num=G.vexnum-1;num>=0;v=finished[--num])
        if(!visited[v]){flag++; DFS2(G,v);  }            //对尚未访问的顶点调用DFS
}

void DFS2(Graph G,int v){
    //从第v个顶点出发递归地深度优先遍历图G
    visited[v]=TRUE;VisitFunc(v);                       //访问第v个顶点
    for(w=FirstAdjVex2(G,v);w>=0; w=NextAdjVex2(G,v,w))   //以v为弧头逆向深度优先
        if(!visited[w]) DFS2(G,w);                        //对v的未访问的邻接点w递归调用DFS
}
```

7.4.3 最小生成树

构造网的最小代价生成树

MST 性质:$N=(V,\{E\}),U\subseteq V,u\in U,v\in V-U,(u,v)$ 是具有最小权值的边，则存在最小生成树包含 (u,v)



普里姆 (Prim)

连通网 $N=(V,\{E\})$

初始状态 $U=\{u_0\},TE=\{\}$，

选择 $u\in U,v_1\in V-U,min\{(u,v_1)\}$，

更新 $U=\{u_0,v_1\},TE=\{(u,v_1)\}$，

重复选择至 U=V

$closedge[i-1].lowcost=Min\{cost(u,v_i)|u\in U\}$

时间复杂度 $o(n^2)$

克鲁斯卡尔 (Kruskal)

连通网 $N=(V,\{E\})$

初始状态 $T=(V,\{\})$, 每个顶点自成连通分量

选择 $e_i=MIN(E),e_i$ 若属于不同连通分量更新，否则舍弃

更新 $E=e-e_i,T=(V,\{e_i\})$

舍弃 $E=e-e_i$

重复选择至 T 中所有顶点在一个连通分量上

时间复杂度 $o(e\log e)$

```
typedef struct{
    VertexType adjvex;
    VRType lowcost;
}closedge[MAX_VERTEX_NUM]       //记录U到V-U最小代价边的辅助数组定义


void MiniSpanTree_PRIM(MGraph G,VertexType u){
    //用普里姆算法从第u个顶点构造网G的最小生成树T, 输出T的个边
    k=LocateVex(G,u);

    for(j=0;j<G.vexnum;++j)         //辅助数组初始化
        if(j!=k) closedge[j]={u,G.arcnum[k][j].adj};    //{adjvex,lowcost}
    closedge[k].lowcost=0;                              //初始, U={u}
    for(i=1;i<G.vexnum;++i){                            //选择其余G.vexnum-1个顶点
        k=minimum(closedge);                            //求出k的下一个节点
        //此时 closedge[k].lowcost=MIN{closedge[v_i].lowcost | closedge[v_i].lowcost>0,v_i \in V-U}
        printf(closedge[k].adjvex,G,vexs[k]);           //输出生成树的边
        closedge[k].lowcost=0;                          //第k顶点并入U集
        for(j=0;j<G.vexnum;++j){
            if(G.arcs[k][j].adj<closedge[j].lowcost)
                closedge[j]={G.vexs[k],G.arcs[k][j].adj };  //新顶点并入U后重新选择最小边
        }
    }
}//MiniSpanTree
```

7.4.4 关节点和重连通分量

删除点和依附点的边生成两个或者以上的连通分量，**关节点**

没有关节点的图，任意一对顶点间存在不止一条路径，**重连通图**

删除 k 个顶点才破坏连通性，连通度 k

关节点特性

生成树的根右两个及以上子树，根节点为关节点

非叶子节点 v，及其子树，没有指向双亲回边,v 为关节点

$$low(v) = Min \left\{ visited[v], visited[k], low[w] \;\middle|\; \begin{array}{l} w\text{是}v\text{在生成树上孩子节点} \\ k\text{是}v\text{在生成树上回边连接组选节点} \\ (v,w),(v,k) \in Edge \end{array} \right\}$$

顶点 v，孩子节点 w，若 $low[w] \geqslant visited[v]$, 则 w 及其子孙均无指向 v 的回边

```
void FindArticul(ALGraph G){
    //连通图G以邻接表存储, 查找输出关节点, 全局变量count对方问记数
    count=1;visited[0]=1;                   //设定邻接表上0号顶点为生成树的根
    for(i=1;i<G.vexnum;++i) visited[i]=0;   //初始化, 其余节点未访问
    p=G.vertices[0],firstarc; v=p->adjvex;
    DFSArticul(G,v);                        //从第v点出发深度优先查找关节点
    if(count<G.vertices[0].data){           //生成树至少有两棵子树
        printf(0,G,vertices[0].data);       //根节点是关节点
        while(p->nextarc){
            p=p->next;v=p->adjvex;
            if(visited[v]==0) DFSArticul(G,v);
        }
    }
}

void DFSArticul(ALGraph G,int v0){
    //从第v0个顶点出发深度优先遍历图G, 查找并输出关节点
    visited[v0]=min=++count;            //v0是第count个访问的顶点
    for(p=G.vertices[v0].firstarc;p;p=p->nextarc){     //对v0的每个邻接顶点检查
        w=p->adjvex;                                    //w为v0的邻接点
        if(visited[w]==0){                              //w未曾访问, 是v0的孩子
            DFSArticul(G,w);                            //返回前求得low[w]
            if(low[w]<min) min=low[w];
            if(low[]>=visited[v0]) printf(v0,G.vertices[v0].data);  //关节点
        }else if{visited[w]<min} min=visited[w];        //w已访问, w是v0在生成树上的祖先
    }
    low[v0]=min;
}
```

时间复杂度 $o(n+e)$

7.5 有向无环图及其应用

有向无环图 (directed acycline graph /DAG)

7.5.1 拓扑排序

偏序 (部分关系)，全序 (全部关系)
偏序 + 人为 = 全序，拓扑有序
顶点活动，弧表优先关系的有向图 (Activity On Vertex Network /AOV)
拓扑排序

(1) 选择无前驱节点的顶点，输出
(2) 删除顶点，和以他为尾的弧
(3) 重复 (1) (2)

```
Status TopologicalSort(ALGraph G){
    //有向圖G採用鄰接表存儲結構
    //若G無迴路， 則輸出G的頂點的一個拓撲序列並返回OK， 否則返回ERROR
    FindInDegree(G,indegree);        //對個點求入度 indegree[0~vernum-1]
    for(i=0;i<G.vexnum;++i)          //零入度頂點棧
        if(!indegree[i]) Push(S,i);  //入度為0者入棧
    count=0;                         //對輸出點計數
    while(!StackEmpty(S)){
        Pop(S,i); printf(i,G.vertices[i].data); ++count;  //輸出i號頂點並計數
        for(p=G.vertices[i].firstarc ;p;p=p->nextarc){
            k=p->adjvex;
            if(!(--indegree[k])) Push(S,k);      //若入度減為0,則入棧
        }
    }
    if(count<G.vexnum) return ERROR;             //該有向圖有迴路
    else return OK;
}
```

n 頂點 e 邊圖, 時間複雜度 $o(n+e)$

7.5.2 关键路径

邊表活动的有向图，帶權有向無環圖 (Activity On Edge Network /AOE)
完成的最短時間，路徑長度最長的路徑的長度
路徑長度最長的路徑 **關鍵路徑**

```
Status TopologicalOrder(ALGraph G,Stack &T){
    //有向網G採用鄰接表存儲結構，求各頂點事件的最早發生時間 ve全局變量
    //T為拓撲序列頂點棧，S為零入度頂點棧
    //若G無迴路，則用棧T返回G的一個拓撲序列，且函數值為OK，否則為ERROR
    FindInDegree(G,indegree);       //對個頂點求入度indegree[0~vernum-1]
    //建零入度頂點棧
    InitStack(T);count=0;ve[0..G.vexnum-1]=0;   //初始化
    while(!StackEmpty(S)){
        Pop(S,j);Push(T,j); ++count;    //j號頂點入T棧並計數
        for(p=G.vertices[j].firstarc;p;p=p->nextarc){
            k=p->adjvex;                        //對j號頂點的各個鄰接點的入度減1
            if(--indegree[k]==0) Push(S,k);     //若入度減為0，則入棧
            if(ve[j]+ *(p->info) >ve[k])  ve[k]=ve[j] + *(p->info);
        }
    }
    if(count<G.vexnum) return ERROR;    //該有向圖有迴路
    else return OK;
}

Status CriticalPath(ALGraph G){
    //G為有向網，輸出G的各項關鍵活動
    if(!TopologicalOrder(G,T)) return ERROR;
    vl[0..G.vexnum-1]=ve[0..G.vexnum-1];        //初始化頂點事件的最遲發生時間
    while(!StackEmpty(T))                       //按拓撲逆序求各頂點的vl值
        for(Pop(T,j),p=G.vertices[j].firstarc;p;p=p->nextarc){
            k=p->adjvex; dut=*(p->info);        //dut<j,k>
            if(vl[k]-dut < vl[j])  vl[j]=vl[k]-dut;
        }
    for(j=0;j<G.vexnum;++j)             //求ee,el和關鍵活動
        for(p=G.vertices.firstarc;p;p=p->nextarc){
            k=p->adjvex;dut=*(p->info);
            ee=ve[j];el=ve[k]-dut;
            tag=(ee==el)? '*' : '' ;
            printf(j,k,dut,ee,el,tag);  //輸出關鍵活動
        }
}
```

n 頂點 e 邊圖, 時間複雜度 $o(n+e)$

7.6 最短路径

7.6.1 从某个源点到其余各顶点的最短路径

7.6.2 每一对顶点之间的最短路径

7.6.3 关键路径

8 动态存储管理

9 查找

```
//可能关键字类型
typedef float KeyType;
typedef int KeyType;
typedef char *KeyType;
//可能数据元素类型
typedef struct{
    KeyType key;
    ...
}SElemType;
//数值比较
#define EQ(a,b) ((a)==(b))
#define LT(a,b) ((a)<(b))
#define LQ(a,b) ((a)<=(b))
//字符串比较
#define EQ(a,b) (strcmp(!(a),(b)) )
#define LT(a,b) (strcmp((a),(b))<0)
#define LQ(a,b) (strcmp((a),(b))<=0)
```

9.1 静态查找表

9.1.1 顺序表的查找

顺序查找

```
typedef struct{
    ElemType *elem;
    int length;
}SSTable;

int Search_Seq(SSTable ST,KeyType key){
    //在顺序表ST中查找关键字等于key的数据元素
    //若找到，函数值为该元素在表中位置，否则为0
    ST.elem[0].key=key;                      //哨兵
    for(i=ST.length;!EQ(ST.ele[i].key,key);--i){  //从后往前找
        return i                             //找不到时i=0
    }
}
```

平均查找长度

$$\begin{aligned} ASL &= \sum_{i=1}^{n} P_iC_i \xlongequal{C_i=n-i+1} nP_1+(n-1)P_2+\cdots+2P_{n-1}+P_n \\ ASL_{SS} &= \sum_{i=1}^{n} P_iC_i \xlongequal[C_i=n-i+1]{\sum_{i=1}^{n} P_i=1, P_i=\frac{1}{n}} \frac{1}{n}\sum_{i=1}^{n}(n-i+1)=\frac{n+1}{2} \\ ASL_{SS}' &= \sum_{i=1}^{n} P_iC_i+Q_iD_i \xlongequal[C_i=n-i+1, D_i=n+1]{\sum_{i=1}^{n} P_i=Q_i=\frac{1}{2}} \frac{1}{2n}\sum_{i=1}^{n}(n-i+1)+\frac{1}{2}(n+1)=\frac{3(n+1)}{4} \end{aligned}$$

9.1.2 有序表的查找

折半查找

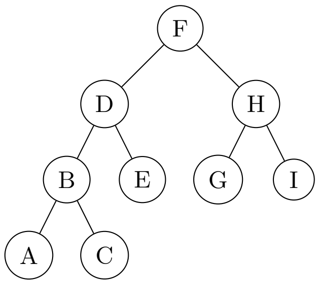
```
int Search_Bin(SSTable ST,KeyType key){
    //在有序表ST中折半查找关键字等于key的数据元素
    //若找到，函数值为该元素在表中位置，否则为0
    low=1;high=ST.length;              //置区间初值
    while(low<=high){
        mid=(low+high)/2;
        if(EQ(key,ST.elem[mid].key))return mid;         //找到待查元素
        else if (LT(key,ST.elem[mid].key))high=mid-1;   //继续前半区间查找
        else low=mid+1;                                  //继续后半区间查找
    }
    return 0;                                            //表中不存在待查元素
}
```

$$
\begin{aligned}
\sum_{j=1}^{h} j * x^{j-1} &= \left(\sum_{j=1}^{h} x^j\right)' \\
&= \left(\frac{x-x^{h+1}}{1-x}\right)' \\
&= \frac{[1-(h+1)x^h](1-x)+x-x^{h+1}}{(1-x)^2} \\
\sum_{j=1}^{h} j * 2^{j-1} &\xlongequal{x=2} [1-(h+1)2^h](1-2)+2-2^{h+1} \\
&= (h+1)2^h - 1 + 2 - 2 \cdot 2^h \\
&= (h-1)2^h + 1 \\
2^h - 1 = n \Rightarrow h &= \log_2(n+1) \\
ASL_{bs} &= \sum_{i=1}^{n} P_i C_i \\
&= \frac{1}{n} \sum_{j=1}^{h} j * 2^{j-1} (\text{层高} * \text{节点数}) \\
&= \frac{1}{n}[(h-1)2^h + 1] \\
&= \frac{1}{n}[(\log_2(n+1)-1)2^{\log_2(n+1)} + 1] \\
&= \frac{1}{n}[(\log_2(n+1)-1)(n+1) + 1] \\
&= \frac{1}{n}[(\log_2(n+1))(n+1) - n - 1 + 1] \\
&= \frac{n+1}{n} \log_2(n+1) - 1 \\
&\approx \log_2(n+1) - 1, (n > 50)
\end{aligned}
$$

9.1.3 静态树表的查找

$$
\begin{aligned}
sw_i &= \sum_{j=l}^{i} w_j \\
\Delta P_i &= \left|\sum_{j=i+1}^{h} w_j - \sum_{j=l}^{i-1} w_j\right| \\
&= |(sw_h - sw_i) - (sw_{i-1} - sw_{l-1})| \\
&\xlongequal{sw_{l-1}=0, w_{l-1}=0} |sw_h + sw_{l-1} - sw_i - sw_{i-1}|
\end{aligned}
$$

j	0	1	2	3	4	5	6	7	8	9
key		A	B	C	D	E	F	G	H	I
w_i	0	1	1	2	5	3	4	4	3	5
sw_i	0	1	2	4	9	12	16	20	23	28
$l=1, h=9, \Delta P_i$		27	25	22	15	7	0	8	15	23
$l_1=1, h_1=5, l_1=6, h_1=9, \Delta P_i$		11	9	6	1	19		8	1	7




```c
typedef BiTree SOSTree;            //次优查找树采用二叉链表存储结构
int SecondOptimal(BiTree &T,ElemType R[],float sw[],int low ,int high){
    //由有序表R[low...high]及其累计权值表sw(sw[0]==1)递归构造次优查找数T
    i=low;min=abs(sw[high]-sw[low]);dw=sw[high]+sw[low-1];
    for(j=low+1;j<=high;++j){                //选择最小ΔPi值
        if(abs(dw-sw[j]-sw[j-1])<min){
            i=j;min=abs(dw-sw[j]-sw[j-1]);
        }
    }
    T=(BiTree)malloc(sizeof(BiTNode));
    T->data=R[i];                                   //生成节点
    if(i==low) T->lchild=NULL;                      //左子树空
    else SecondOptimal(T->lchild,R,sw,low,i-1);    //构造左子树
    if(i==high) T->rchild=NULL;                     //右子树空
    else SecondOptimal(T->rchild,R,sw,i+1,high);   //构造右子树
}

Status CreateSOSTree(SOSTree &T,SSTable ST){
    //有序表ST构造一棵次优查找树T，ST的数据元素含有权域weight
    if(ST.length==0) T=NULL;
    else{
        FindSW(sw,ST);      //按照有序表ST中各元素的weight域求累计权值表sw
        SecondOptimal(T,ST.elem,sw,1,ST.length);
    }
    return OK;
}
```

$$
\begin{aligned}
&F_0=0,F_1=1\\
&F_n &&=F_{n-1}+F_{n-2}\quad(n\geqslant 2)\\
&F_n-sF_{n-1} &&=(1-s)(F_{n-1}+\tfrac{1}{1-s}F_{n-2})\quad(n\geqslant 2)\\
& &&\xlongequal{-s=\frac{1}{1-s}}(1-s)(F_{n-1}+sF_{n-2})\quad(n\geqslant 2)\\
& &&=(1-s)^{n-1}(F_1+sF_0)\\
& &&=(1-s)^{n-1}\\
&F_n+k(1-s)^{n-1} &&=sF_{n-1}+(1+k)(1-s)^{n-1}\\
& &&=s[F_{n-1}+\tfrac{(1+k)}{s}(1-s)^{n-1}]\\
& &&=s[F_{n-1}+\tfrac{(1+k)(1-s)}{s}(1-s)^{n-2}]\\
& &&\xlongequal{k=\frac{(1+k)(1-s)}{s}}s[F_{n-1}+k(1-s)^{n-2}]\\
& &&=s^{n-1}[F_1+k(1-s)^0]\\
& &&=s^{n-1}(1+k)\\
&F_n &&=(1+k)s^{n-1}-k(1-s)^{n-1}\\
& &&=\tfrac{1\pm\sqrt{5}}{\pm2\sqrt{5}}(\tfrac{1\pm\sqrt{5}}{2})^{n-1}-\tfrac{1\mp\sqrt{5}}{\pm2\sqrt{5}}(\tfrac{1\mp\sqrt{5}}{2})^{n-1}\\
& &&=\tfrac{1}{\sqrt{5}}\left[\tfrac{1\pm\sqrt{5}}{\pm2}(\tfrac{1\pm\sqrt{5}}{2})^{n-1}-\tfrac{1\mp\sqrt{5}}{\pm2}(\tfrac{1\mp\sqrt{5}}{2})^{n-1}\right]\\
& &&=\tfrac{1}{\sqrt{5}}\left[\tfrac{1\pm\sqrt{5}}{\pm2}(\tfrac{1\pm\sqrt{5}}{2})^{n-1}\mp\tfrac{1\mp\sqrt{5}}{2}(\tfrac{1\mp\sqrt{5}}{2})^{n-1}\right]\\
& &&=\begin{cases}\tfrac{1}{\sqrt{5}}\left[\tfrac{1+\sqrt{5}}{+2}(\tfrac{1+\sqrt{5}}{2})^{n-1}-\tfrac{1-\sqrt{5}}{2}(\tfrac{1-\sqrt{5}}{2})^{n-1}\right]\\ \tfrac{1}{\sqrt{5}}\left[\tfrac{1-\sqrt{5}}{-2}(\tfrac{1-\sqrt{5}}{2})^{n-1}+\tfrac{1+\sqrt{5}}{2}(\tfrac{1+\sqrt{5}}{2})^{n-1}\right]\end{cases}\\
& &&=\tfrac{1}{\sqrt{5}}\left[(\tfrac{1+\sqrt{5}}{2})^{n}-(\tfrac{1-\sqrt{5}}{2})^{n}\right]\\
& &&\approx\tfrac{1}{\sqrt{5}}(\tfrac{1+\sqrt{5}}{2})^{n}
\end{aligned}
$$

$$
\begin{cases}-s=\frac{1}{1-s}\\ k=\frac{(1+k)(1-s)}{s}\end{cases}\Rightarrow\begin{cases}-1-s+s^2=0\\ ks=1-ks+k-s\end{cases}\Rightarrow\begin{cases}s=\frac{1\pm\sqrt{5}}{2}\\ (2s-1)k=1-s\end{cases}\Rightarrow\begin{cases}s=\frac{1\pm\sqrt{5}}{2}\\ 1-s=\frac{1\mp\sqrt{5}}{2}\\ k=\frac{1-s}{(2s-1)}=\frac{1\mp\sqrt{5}}{\pm2\sqrt{5}}\\ 1+k=\frac{1\pm\sqrt{5}}{\pm2\sqrt{5}}\end{cases}
$$

9.1.4 索引顺序表的查找

$$
\begin{aligned}
ASL_{bs}&=L_b+L_w\\
&=\tfrac{1}{b}\textstyle\sum_{j=1}^{b}j+\tfrac{1}{s}\sum_{j=1}^{s}j\\
&=\tfrac{1+b}{2}+\tfrac{1+s}{2}\\
&=\tfrac{1}{2}(\tfrac{n}{s}+s)+1\\
ASL_{bs}^{'}&\approxeq\log_2(\tfrac{n}{s}+1)-1+\tfrac{1+s}{2}\\
&\approxeq\log_2(\tfrac{n}{s}+1)+\tfrac{s}{2}-\tfrac{1}{2}\\
&\approxeq\log_2(\tfrac{n}{s}+1)+\tfrac{s}{2}
\end{aligned}
$$

9.2 动态查找表

9.2.1 二叉排序树和平衡二叉树

二叉排序树及其查找过程

二叉排序树是空树或者具有性质
(1) 非空左子树上所有节点小于根节点
(2) 非空右子树上所有节点大于根节点
(3) 左右子树分别为二叉排序树

二叉排序树的插入和删除

```
Status SearchBST(BiTree T,KeyType key,BiTree f,BiTree &p){
    //二叉排序树T中查找key
    //成功p指向节点，返回TRUE，失败p指向访问节点，返回FALSE
    //f指向双亲节点，初始值为NULL
    if(!T){p=f;return FALSE;}  //查找失败
    else if EQ(key,T->data.key){p=T;return TRUE;} //查找成功
    else if LT(key,T->data.key) return SearchBST(T->lchild,key,T,p);
    else  return SearchBST(T->rchild,key,T,p);
}

Status InsertBST(BiTree &T,ElemType e){
    //二叉排序树T中不存在key，插入e返回TRUE
    //否则返回FALSE
    if(!SearchBST(T,e.key,null,p)){
        s=(BiTree)malloc(sizeof (BiTNode));
        s->data=e;s->lchild=s->rchild=NULL;
        if(!p)T=s;
        else if(LT(e.key,P->data.key))p->lchild=s;
        else p->rchild=s;
        return TRUE;
    }
    else return FALSE;
}
```

$$
\text{双亲节点 } *f \text{ 删除节点 } *p
\begin{cases}
p_L, p_R\text{均为空树, 改双 *f 亲指针} \\
p_L \; or \; p_R\text{为空树，子树为双亲 *f 子树} \\
p_L, p_R\text{均不为空树，}
\begin{cases}
(1)p_L\text{为双亲 *f 左子树,}p_r\text{为}p_L\text{最右} \\
(2)p_L\text{最右 *s 替代}*p\text{ 删除}*s\text{重复操作}
\end{cases}
\end{cases}
$$

```
Status DeleteBST(BiTree &T,KeyType key){
    //若二叉树T中存在key，删除该节点
    //并返回TRUE，否则返回FALSE
    if(!T) return FALSE;             //不存在key
    else{
        if(EQ(key,T->data.key)) return Delete(T);  //找到key
        else if(LT(k,T->data.key)) return DeleteBST(T->lchild,key);
        else return DeleteBST(T->rchild,key);
    }
}

Status Delete(BiTree &p){
    //从二叉树删除节点p，重接左子树或右子树
    if(!p->rchild){q=p;p=p->lchild;free(q);}
    else if(!p->lchild){q=p;p=p->rchild;free(q);}
    else{
        q=p;s=p->lchild;                    //左转
        while(s->rchild){q=s;s=s->rchild;}  //右转到尽头，
        p->data=s->data;                    //s指向p前驱，q指向s双亲
        if(q!=p)q->rchild=s->lchild;  //重接q右子树
        else q->lchild=s->lchild;     //重接q右子树(左单支)
        free(s);
    }
    return TRUE;
}
```

二叉排序树的查找分析

$(\frac{a_1+a_2+\cdots+a_n}{n})^n \geqslant a_1 a_2 \dots a_n$

$e = \lim_{n\to\infty}(1+\frac{1}{n})^n \approx 2.76$(自然常數)

$s_n = (1+\frac{1}{n})^n = (\frac{1+n}{n})^n \cdot 1 \leqslant \left(\frac{\frac{n+1}{n}+\cdots+\frac{n+1}{n}+1}{n+1}\right)^{n+1} = (\frac{n+2}{n+1})^{n+1} = s_{n+1}$, 單增 $\quad n \geqslant 1$

$t_n = \left(1+\frac{1}{n}\right)^{n+1} = \left(\frac{n+1}{n}\right)^{n+1}$, 單減 $\quad n \geqslant 1$

$\frac{1}{t_n} = \left(\frac{n}{n+1}\right)^{n+1} = \left(\frac{n}{n+1}\right)^{n+1} \cdot 1 \leqslant \left(\frac{\frac{n}{n+1}+\cdots+\frac{n}{n+1}+1}{n+2}\right)^{n+2} = \left(\frac{n+1}{n+2}\right)^{n+2} = \frac{1}{t_{n+1}}, t_n \geqslant t_{n+1}$

$2 = s_1 < s_n < t_n < t_1 = 4$

$(1+\frac{1}{n})^n < s_{max} = e < t_{min} < (1+\frac{1}{n})^{n+1}, n\ln(\frac{n+1}{n}) < 1 < (n+1)\ln(\frac{n+1}{n})$

$\frac{1}{n+1} < \ln(\frac{n+1}{n}) < \frac{1}{n}$

$\gamma_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} - \ln n$
$> \ln(\frac{2}{1}) + \ln(\frac{3}{2}) + \ln(\frac{4}{3}) + \cdots + \ln(\frac{n+1}{n}) - \ln n$
$> \ln(n+1) - \ln n > 0,$ 正项
$\gamma_{n+1} - \gamma_n = \frac{1}{n+1} - \ln(n+1) + \ln n = \frac{1}{n+1} - \ln(\frac{n+1}{n}) < 0, \gamma_{n+1} < \gamma_n$ 单减
$0 < \gamma_n < 1$
$k = \sum_{j=1}^{n} \frac{1}{j}$
$\int_1^n \frac{1}{x} dx = \ln n < 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n-1} = k + \frac{1}{n}$
$\int_1^n \frac{1}{x} dx = \ln n > \frac{1}{2} + \frac{1}{3} + \frac{1}{3} + \cdots + \frac{1}{n} = k - 1$
$\int_1^n \frac{1}{x} dx = \ln n < \frac{1}{2}(1 + \frac{1}{2}) + \frac{1}{2}(\frac{1}{2} + \frac{1}{3}) + \frac{1}{2}(\frac{1}{3} + \frac{1}{4}) + \cdots + \frac{1}{2}(\frac{1}{n-1} + \frac{1}{n}) = k - \frac{1}{2n} - \frac{1}{2}$
$\frac{1}{2} < \frac{1}{2} + \frac{1}{2n} < (k - \ln n) = \gamma_n < 1$
$k - \ln n = \gamma \approx 0.577$(欧拉常数)
$\sum_{j=2}^{n} \frac{1}{j} = k - 1 = \ln n + \gamma - 1 < \ln n$

$P(n,i) = \frac{1}{n}[1 + i \cdot (P(i)+1) + (n-i-1) \cdot (P(n-i-1)+1)]$
$P(n) = \frac{1}{n} \sum_{i=0}^{n-1} P(n,i)$
$= \frac{1}{n} \sum_{i=0}^{n-1} \frac{1}{n}[1 + i \cdot (P(i)+1) + (n-i-1) \cdot (P(n-i-1)+1)]$
$= \frac{1}{n^2} \sum_{i=0}^{n-1} [i \cdot P(i) + i + 1 + (n-i-1) \cdot P(n-i-1) + n - i - 1]$
$= \frac{1}{n^2} \sum_{i=0}^{n-1} [i \cdot P(i) + (n-i-1) \cdot P(n-i-1) + n]$
$= 1 + \frac{1}{n^2} \sum_{i=0}^{n-1} [i \cdot P(i) + (n-i-1) \cdot P(n-i-1)]$
$= 1 + \frac{1}{n^2}[0 \cdot P(0) + (n-1) \cdot P(n-1) + 1 \cdot P(1) + (n-2) \cdot P(n-2) + \cdots + (n-1) \cdot P(n-1) + 0 \cdot P(0)]$
$= 1 + \frac{2}{n^2} \sum_{i=0}^{n-1} i \cdot P(i)$
$k_n = \sum_{i=0}^{n-1} i \cdot P(i), k_{n-1} = \sum_{i=0}^{n-2} i \cdot P(i)$
$k_n = k_{n-1} + (n-1)P(n-1)$
$P(n) = 1 + \frac{2}{n^2} k_n$
$P(n-1) = 1 + \frac{2}{(n-1)^2} k_{n-1}$
$\frac{n^2}{2}(P(n) - 1) = \frac{(n-1)^2}{2}(P(n-1) - 1) + (n-1)P(n-1)$
$P(n) = \frac{n^2-1}{n^2} P(n-1) + \frac{2n-1}{n^2}, P(1) = 1, P(0) = 0$
$\frac{n}{n+1} P(n) = \frac{n-1}{n} P(n-1) + \frac{2n-1}{n(n+1)}$
$s_n = \frac{n}{n+1} P(n), \delta_n = \frac{2n-1}{n(n+1)}$
$s_n = s_{n-1} + \delta_n$
$s_{n-1} = s_{n-1} + \delta_{n-1}$
$\ldots$
$s_2 = s_1 + \delta_2$
$s_1 = s_0 + \delta_1$
$s_n = \sum_{j=1}^{n} \delta_j = \sum_{j=1}^{n} \frac{2j-1}{j(j+1)} = \sum_{j=1}^{n} (2j-1)(\frac{1}{j} - \frac{1}{j+1})$
$= \sum_{j=1}^{n} \frac{2j-1}{j} - \frac{2j-1}{j+1}$
$= \sum_{j=1}^{n} \frac{2j-1}{j} - \frac{2j+2-3}{j+1}$
$= \sum_{j=1}^{n} \frac{-1}{j} + \sum_{j=1}^{n} \frac{3}{j+1}$
$= \sum_{j=1}^{n} \frac{-1}{j} + \sum_{j=1}^{n} \frac{1+2}{j+1}$
$= \sum_{j=1}^{n} (\frac{-1}{j} + \frac{1}{j+1}) + \sum_{j=1}^{n} \frac{2}{j+1}$
$= -\frac{n}{n+1} + \sum_{j=1}^{n} \frac{2}{j+1}$
$P(n) = \frac{n+1}{n} s_n = -1 + 2\frac{n+1}{n} \sum_{j=1}^{n} \frac{1}{j+1} = -1 + 2\frac{n+1}{n}(\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n+1})$
$= 2\frac{n+1}{n}(\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}) + \frac{2}{n} - 1$
$= 2\frac{n+1}{n}(k-1) + \frac{2}{n} - 1 = 2\frac{n+1}{n}(\ln n + \gamma - 1) + \frac{2}{n} - 1 \leqslant 2\frac{n+1}{n} \ln n$

平衡二叉树是空树或者具有性质
(1)左右子树都是平衡二叉树
(2)左右子树高度差绝对值小于 1

平衡因子左子树高度减右子树高度

失衡调整
(1)单向右旋
(2)单向左旋
(3)双向旋转 (先左后右)
(4)双向旋转 (先右后左)

插入 e 算法描述
(1)空树,e 为根节点
(2)e 等于根节点，不插入
(3)e 小于根节点，左子树无 e, 插入左子树
    根节点平衡因子 =-1, 改为 0，树深度 +0
    根节点平衡因子 =0, 改为 1，树深度 +1
    根节点平衡因子 =1
        左子树根节点平衡因子 =1，单向右旋
        左子树根节点平衡因子 =-1，先左后右
(4)e 大于根节点，右子树无 e, 插入右子树
    根节点平衡因子 =1, 改为 0，树深度 +0
    根节点平衡因子 =0, 改为 1，树深度 +1
    根节点平衡因子 =-1
        右子树根节点平衡因子 =-1，单向左旋
        右子树根节点平衡因子 =1，先右后左

```
#define LH +1;
#define EH 0;
#define RH -1;

typedef struct BSTNode{
int bf;                          //节点平衡因子
struct BSTNode * lchild ,*rchild;   //左右孩子指针
}BSTNode,*BSTree;

void R_Rotate(BSTree &p){
    //对以*p为根的二叉排序树作右旋处理, 处理后p指向新的根节点, 即左子树根节点
    lc=p->lchild;               //lc指向*p的左子树的根节点
    p->lchild=lc->rchild;       //lc的右子树挂接为*p的左子树
    lc->rchild=p;p=lc;          //p指向新的根节点
}

void L_Rotate(BSTree &p){
    //对以*p为根的二叉排序树作左旋处理, 处理后p指向新的根节点, 即右子树根节点
    rc=p->rchild;               //rc指向*p的右子树的根节点
    p->rchild=rc->lchild;       //rc的左子树挂接为*p的右子树
    rc->lchild=p;p=rc;          //p指向新的根节点
}

void LeftBalance(BSTree &T){
    //对平衡二叉树T作左平衡处理, 结束时T指向新的根点
    lc=T->lchild;
    switch(lc->bf){
        case LH:
            T->bf=lc-bf=EH; R_Rotate(T);break;
        case RH:
            rd=lc->rchild;
            switch(rd->bf){
                case LH:T->bf=RH;lc->bf=Eh;break;
                case EH:T->bf=lc->bf=EH;break;
                case RH:T->bf=EH;lc->bf=LH;break;
            }
            rd->bf=EH;
            L_Rotate(T->rchild);
            R_Rotate(T);
    }
}
```

```
Status InsertAVL(BSTree &T,ElemType e,Boolean &taller){
    //平衡二叉树T不存在e，插入返回1，否则返回0
    //若插入失衡，则平衡处理，taller反映长高与否

    if(!T){
        T=(BSTree) malloc(sizeof(BSTNode));T->data=e;
        T->lchild=T->rchild=NULL;T->bf=EH;taller=TRUE;
    }
    else{
        if(EQ(e.key,T->data.key)){taller=FALSE;return 0;}
        if(LT(e.key,T->data.key)){
            if(!InsertAVL(T->lchild,e,taller)) return 0;
            if(taller) switch(T->bf){
                case LH:
                    LeftBalance(T);taller=FALSE;break;
                case EH:
                    T->bf=LH;taller=TRUE;break;
                case RH:
                    T->bf=EH;taller=FALSE;break;
            }
        }
        else{
            if(!InsertAVL(T->rchild,e,taller)) return 0;
            if(taller) switch(T->bf){
                case LH:
                    T->bf=EH;taller=FALSE;break;
                case EH:
                    T->bf=RH;taller=TRUE;break;
                case RH:
                    RightBalance(T);taller=FALSE;break;
            }
        }
    }
    return 1;
}
```

平衡二叉树查找的分析

比较次数不超过树的深度

N_h 深度为 h 的平衡二叉树的最少节点数

$N_{n+1}+1=N_n+1+N_{n-1}+1, N_0=0, N_1=1, N_2=2$

$b_{n+1}=b_n+b_{n-1}, b_0=1, b_1=2, b_2=3$

$b_{n+1}-sb_n=(1-s)(b_n-sb_{n-1})=(1-s)^n(b_1-sb_0)=(2-s)(1-s)^n$

$(s-1)s=1, s_1=\frac{1+\sqrt{5}}{2}, s_2=\frac{12\sqrt{5}}{2}$

$b_{n+1}-s_1b_n=(2-s_1)(1-s_1)^n$

$b_{n+1}-s_2b_n=(2-s_2)(1-s_2)^n$

$-(s_2-s_1)b_n=(2-s_2)(1-s_2)^n-(2-s_1)(1-s_1)^n$

$s_2-s_1=-\sqrt{5}, 2-s_2=\frac{3+\sqrt{5}}{2}, 2-s_1=\frac{3-\sqrt{5}}{2}, 1-s_2=\frac{1+\sqrt{5}}{2}, 1-s_1=\frac{1-\sqrt{5}}{2}$

$b_n=\frac{(2-s_2)}{-(s_2-s_1)}(1-s_2)^n-\frac{(2-s_1)}{-(s_2-s_1)}(1-s_1)^n$

$b_n=\frac{1}{\sqrt{5}}\left[(1+\frac{1+\sqrt{5}}{2})(\frac{1+\sqrt{5}}{2})^n-(1+\frac{1-\sqrt{5}}{2})(\frac{1-\sqrt{5}}{2})^n\right]=F_n+F_{n+1}$

$N_n=b_n-1=F_n+F_{n+1}-1=F_{n+2}-1$

$F_h\approx\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^h=c(\varphi)^h$

$N_h=c(\varphi)^{h+2}-1$

$\log(\frac{N_h+1}{c})=(h+2)\log\varphi$

$\frac{\log(\frac{N_h+1}{c})}{\log\varphi}-2=\log_\varphi(\frac{N_h+1}{c})-2=h$

先排序，再构造次优查找树，生成树是二叉排序树

9.2.2 B-树和 B+ 树

B-树及其查找

B-树是平衡的多路查找树，

m 阶 **B-树**是空树或具有性质

(1)每个节点至多有 m 棵子树
(2)若根节点不是叶子节点，则至少有两棵子树
(3)除根结点之外的所有非终端节点，至少包含$\lceil \frac{m}{2} \rceil$棵子树
(4)所有的非终端节点，包含信息
  $(n, A_0, k_1, A_1, K_2, \ldots, K_n, A_n), K_i$为关键字,$A_i$指向根节点的指针
  A_{i-1}指向子树的所有节点小于K_i, A_{i+1}指向子树的所有节点大于K_i
  关键字个数$n, \lceil \frac{m}{2} \rceil - 1 \leqslant n \leqslant m - 1$
(5)所有的叶子节点都出现在同一层次上，且不带信息

```
#define m 3                      //B-树的阶
typedef struct BTNode{
    int keynum;                  //关键字个数，即节点大小
    struct BTNode * parent;      //双亲结点
    KeyType key[m+1];            //关键字向量，0号单元未用
    struct BTNode * ptr[m+1];    //子树指针向量
    Record *recptr[m+1];         //记录指针向量，0号单元未用
}BTNode,*BTree;

typedef struct{
    BTNode *pt;          //指向找到的节点
    int i;               //在节点中的关键字号
    int tag;             //1成功，0失败
}Result;                 //B-树查找结果类型


Result SearchBTree(BTree T,KeyType K){
    //在m阶B-树T上查找K, 返回(pt,i,tag)
    //成功返回位置，失败插入返回插入位置
    p=T;q=NULL;found=FALSE;i=0;     //初始化，p指向待查节点，q指向p的双亲节点
    while(p && !found){
        i=Search(p,k);              //在p->key [1...keynum]中查找
        if(i>0 && p->key[i]==k) found =TRUE;    //查到关键字
        else{q=p;p=p->ptr[i];}
    }
    if(found) return (p,i,1);
    else return (q,i,0);
}
```

B-树查找分析

磁盘节点，内存顺序

$1, 2, 2\lceil \frac{m}{2} \rceil, 2\lceil \frac{m}{2} \rceil^2, \ldots, 2\lceil \frac{m}{2} \rceil^{n-2}, \ldots$

N 关键字 B-树，深度 l+1(叶子算深度)，N+1 叶子节点

$N + 1 \geqslant 2\lceil \frac{m}{2} \rceil^{l+1-2}$

$\log_{\lceil \frac{m}{2} \rceil}(N + 1) \geqslant \log_{\lceil \frac{m}{2} \rceil} 2 + l - 1$

$\log_{\lceil \frac{m}{2} \rceil}(\frac{N+1}{2}) + 1 \geqslant l$

B-树插入和删除

最底层非终端节点添加，添加后关键字个数不超过 m-1 完成，超过分裂

节点分裂 $*p$ 节点含 m-1 关键字，插入后节点信息 $(m, A_0, K_1, A_1, K_2, A_2, \ldots, K_m, A_m)$

$*p_1, (\lceil \frac{m}{2} \rceil - 1, A_0, K_1, A_1, K_2, A_2, \ldots, K_{\lceil \frac{m}{2} \rceil - 1}, A_{\lceil \frac{m}{2} \rceil - 1})$

$*p_2, (m - \lceil \frac{m}{2} \rceil, A_{\lceil \frac{m}{2} \rceil}, K_{\lceil \frac{m}{2} \rceil + 1}, A_{\lceil \frac{m}{2} \rceil + 1}, K_{\lceil \frac{m}{2} \rceil + 2}, A_{\lceil \frac{m}{2} \rceil + 2}, \ldots, K_m, A_m)$

$key, (K_{\lceil \frac{m}{2} \rceil}, *p_2)$ 合并到双亲

```
Status InsertBtree(BTree &T ,KeyType T,BTree q ,int i){
    //m阶B-树, *q的key[i],key[i+1]之间插入关键字k
    //插入后节点过大则分裂
    x=k;ap=NULL;finished=FALSE;
    while(q && !finished){
        Insert(q,i,x,ap);       //将x,ap分别插入q->key[i+1],q->ptr[i+1],
        if(q->keynum<m) finished=TRUE;      //插入完成
        else{                               //分裂节点*q
            s=m/2+1;splite(q,s,ap);x=q->key[s];
            //移动相应元素q->key[s+1..m], q->ptr[s..m]q->recptr[s+1..m]到新节点*ap
            q=q->parent;
            if(q) i=Search(q,x);
        }
    }
    if(!finished)               //T是空树或者节点已分裂为节点*p, *ap
        NewRoot(T,q,x,ap);      //生成含信息(T,x,ap)的新的根节点*T,原T和ap为子树指针
    return OK;
}
```

最底层非终端节点删除，删除后关键字个数不小于 $\lceil \frac{m}{2} \rceil$ 完成，小于合并

非终端节点 K_i 用指针 A_i 子树最小关键字 Y 替代 K_i 再删除 Y

非终端删除情况 (关键字个数)
(1)所在节点大于等于$\lceil \frac{m}{2} \rceil$
(2)所在节点等于$\lceil \frac{m}{2} \rceil - 1$, 存在兄弟节点大于$\lceil \frac{m}{2} \rceil - 1$ 双亲借兄弟，靠近给自己
(3)所在节点等于$\lceil \frac{m}{2} \rceil - 1$, 兄弟节点都等于$\lceil \frac{m}{2} \rceil - 1$ 毁灭自己，留给兄弟

9.2.3 键树

键树又称数字查找树，度大于 2 的树。元素是组成关键字的符号。关键字是数值，单词。

键树是有序树，结束符 \$ 小于任何字符

键树存储结构

(1) 孩子兄弟链表，分支节点 $(symbol, first, next)$，叶子节点 $infoptr$ 域，双链树

```
#define MAXKEYLEN 16        //关键字最大长度
typedef struct{
    char ch[MAXKEYLEN];     //关键字
    int num;                //关键字长度
}KeysType;                  //关键字类型

typedef enum{LEAF,BRANCH} NodeKind;     //节点种类:{叶子, 分支}
typedef struct DLTNode{
    char symbol;
    struct DLTNode *next;               //指向兄弟节点的指针
    NodeKind kind;
    union{
        Record *infoptr;               // 叶子节点的记录指针
        struct DLTNode *first          // 分支节点的孩子链指针
    }
}DLTNode,*DLTree;                      //双链树的类型

Record * SearchDLTree(DLTree T,KeysType K){
    //在非空双链树T中查找K, 存在返回记录指针, 失败返回空指针
    p=-T->first;i=0;
    while(p&& i<k.num){
        while(p&& p->symbol !=K.ch[i]) p=p->next;      //查找第i
        if(p&& i<K.num-1) p=p->first;                  //准备查找下一位
        ++i;
    }
    if(!p) return NULL;                                //查找成功
    else return p->infoptr;                            //查找失败
}
```

键树节点最大度 d，深度 h，双链树平均查找长度 $\frac{h}{2}(1+d)$

插入删除节点，等于在树中某个节点插入删除子树

(2) 多重链表，单支树压缩为叶子节点，

```
typedef struct TrieNode{
    NodeKind kind;
    union{
        struct{KeysType k;Record * infoptr;} lf;   //叶子节点
        struct{TrieNode *ptr[27]; int num;} bh;    //分支节点
    }
}TrieNode,*TrieTree;

Record * SearchTrie(TrieTree T,KeysType k){
    //在键树T中查找关键字等于K的记录
    for(p=T,i=0;                                  //对k的每个字符逐个查找
        p&& p->kind==BRANCH && i<K.num;            //*p为分支节点
        p=p->bh.ptr[ord(K.ch[i])],++i);            //ord 求字符在字母表中序号,$为0
    if(p=&& p->kind==LEAF&& p->lf.k==k) return p->lf.infoptr;  //查找成功
    else return NULL;
}
```

多重链表键树分割，无查找分析

9.3 哈希表

9.3.1 哈希表

關鍵字k, 象$f(k)$, 對應關係f, **哈希函數**

(1)哈希函數是映像

(2)不同關鍵字同象**衝突**

(3)關鍵字象做位置，以哈希函數，衝突處理辦法映射關鍵字到連續地址**哈希表**

(4)映射過程**散列** 存儲位置**哈希地址或散列地址**

9.3.2 哈希函數構造方法

關鍵字映射到地址等概率**均勻哈希函數**

(1)直接定地址$H(key) = a \cdot key + b$

(2)數字分析

(3)平方取中

(4)折疊法,(移位折疊，間界折疊)

(5)除留餘數$H(key) = keyMODp$(質數，不小於 20 質因數的合數),$p \leqslant m$(表長)

(6)隨機數$H(key) = random(key)$

9.3.3 衝突處理方法

地址序列

(1)開放定址法$H_i = (H(key) + d_i)\, MOD\, m \quad i = 1, 2, \ldots, k(k \leqslant m)$

d_i取法　(1)$d_i = 1, 2, 3, \cdots, m-1$，線性探測再散列

(2)$d_i = 1^2, -1^2, 2^2, -2^2, 3^2, \cdots, \pm k^2$，二次探測再散列

(3)$d_i =$ 偽隨機數列，隨機探測再散列

處理同義詞衝突產生非同義詞衝突，**二次聚集**

線性能填滿；平方形如$m = 4j + 3$的素數能填滿；隨機數列

(2)再哈希法$H_i = RH_i(key) \quad i = 1, 2, \ldots, k$

(3)鏈地址法$ChainChainHash[i]$

(4)公共溢出區$HashTable[0..m-1], OverTable[0..v]$

9.3.4 哈希表的查找及其分析

有記錄，且記錄等於關鍵字則查找成功

```
\\---開放地址哈希表的存儲結構
int Hashsize[]={997,...};        //哈希容量遞增表，一個合適的素數序列
typedef struct{
    ElemType * elem;             //數據元素存儲基址，動態分配數組
    int count;                   //當前數據元素個數
    int sizeindex;               //Hashsize[sizeindex]為當前容量
}HashTable;

#define SUCCESS 1
#define UNSUCCESS 0
#define DUPLICATE -1

Status SearchHash(HashTable H,KeyType k,int &p,int &c){
    //在開放地址哈希表中查找關鍵碼為k的元素
    //成功p指向节点，返回SUCCESS。失敗p指向插入位置，返回UNSUCCESS
    //c用作衝突計數，初始值0，供建表插入時參考
    p=Hash(k);
    while(H.elem[p].key !=NULLKEY &&          //有記錄
        !EQ(k,H.elem[p].key))                 //關鍵字不等
        colision(p,++c);                      //求得下一探查地址
    if(EQ(k,H.elem[p].key)) return SUCCESS; //成功，p返回位置
    else return UNSUCCESS;                    //失敗，p返回插入位置
}

Status InsertHash(HashTable &H,ElemType e){
    //查找不成功插入e到H，返回OK
    //衝突次數過大則重建哈希表
    c=0;
    if(SearchHash(H,e.key,c)) return DUPLICATE;     //表中有e
    else if(c<Hashsize[H.sizeindex]/2){             //衝突次數未達到上限
        H.elem[p]=e;++H.count;return OK;            //插入e
    }
    else{
        RecreateHashTable(H);return UNSUCCESS;      //重建哈希表
    }
}
```

哈希表裝填因子 $\alpha = \frac{\text{表中填入記錄數}}{\text{哈希表長度}}$

成功時平均查找長度 $S_{nl} \approx \frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$ 線性探測再散列

$S_{nr} \approx -\frac{1}{\alpha}\ln(1-\alpha)$ 偽隨機探測，二次探測，再哈希

$S_{nc} \approx 1+\frac{\alpha}{2}$ 鏈地址法

失敗時平均查找長度 $U_{nl} \approx \frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$ 線性探測再散列

$U_{nr} \approx \frac{1}{1-\alpha}$ 偽隨機探測，二次探測，再哈希

$U_{nc} \approx \alpha + e^{-\alpha}$ 鏈地址法

随机探测公式推导

哈希表长度 m, 已裝入 n 个记录，哈希函数均匀，处理冲突后产生地址随机

p_i, 再填入一个记录 i 次地址均发生冲突

p_i, 再填入一个记录 i-1 次地址均发生冲突，第 i 次成功

$p_1 = \frac{n}{m}$ $\quad q_1 = (1-\frac{n}{m})$

$p_2 = \frac{n}{m}\cdot\frac{n-1}{m-1}$ $\quad q_2 = \frac{n}{m}\cdot(1-\frac{n-1}{m-1})$

$\vdots$ $\quad \vdots$

$p_n = \frac{n}{m}\cdot\frac{n-1}{m-1}\cdots\frac{n-(n-1)}{m-(n-1)}$ $\quad q_n = \frac{n}{m}\cdot\frac{n-1}{m-1}\cdots(1-\frac{n-(n-1)}{m-(n-1)})$

$p_{n+1} = \frac{n}{m}\cdot\frac{n-1}{m-1}\cdots\frac{n-(n-1)}{m-(n-1)}\cdot\frac{n-n}{m-n} = 0$ $\quad q_{n+1} = p_{n+1} = \frac{n}{m}\cdot\frac{n-1}{m-1}\cdots\frac{n-(n-1)}{m-(n-1)}\cdot(1-\frac{n-n}{m-n})$

$q_i = p_{i-1} - p_i, p_0 = 1$

$$\begin{aligned}
\text{查找失败时}U_n &= \textstyle\sum_{i=1}^{n+1} q_i C_i = \sum_{i=1}^{n+1} (p_{i-1} - p_i) i \\
&= \textstyle\sum_{i=1}^{n+1} p_{i-1} i - \sum_{i=1}^{n+1} p_i i \qquad = \sum_{i=0}^{n} p_i (i+1) - \sum_{i=1}^{n+1} p_i i \\
&= \textstyle\sum_{i=0}^{n} p_i i - \sum_{i=1}^{n+1} p_i i + \sum_{i=0}^{n} p_i \qquad = \sum_{i=1}^{n} p_i i - \sum_{i=1}^{n+1} p_i i + \sum_{i=1}^{n} p_i + 1 \\
&= \textstyle -p_{n+1}(n+1) + \sum_{i=1}^{n} p_i + 1 \qquad = \sum_{i=1}^{n} p_i + 1
\end{aligned}$$

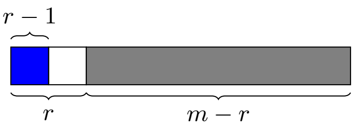
$C_m^n = \binom{m}{n}$

$\binom{m}{n} = \frac{m}{n}\binom{m-1}{n-1} \Rightarrow \frac{m}{n} = \frac{\binom{m}{n}}{\binom{m-1}{n-1}} \Rightarrow \frac{n}{m} = \frac{\binom{m-1}{n-1}}{\binom{m}{n}}$

$\binom{m}{n} = \binom{m-1}{n} + \binom{m-1}{n-1} \Rightarrow \binom{m}{n} - \binom{m-1}{n} = \binom{m-1}{n-1} \Rightarrow \binom{m+1}{n+1} - \binom{m}{n+1} = \binom{m}{n}$

$$\begin{aligned}
&\textstyle\sum_{i=1}^{n} p_i + 1 = * \\
&= 1 + \frac{n}{m} + \frac{n}{m}\cdot\frac{n-1}{m-1} + \cdots + \frac{n}{m}\cdots\frac{n-(n-1)}{m-(n-1)} \\
&= 1 + \frac{\binom{m-1}{n-1}}{\binom{m}{n}} + \frac{\binom{m-1}{n-1}}{\binom{m}{n}}\cdot\frac{\binom{m-2}{n-2}}{\binom{m-1}{n-1}} + \cdots + \frac{\binom{m-1}{n-1}}{\binom{m}{n}}\cdot\frac{\binom{m-2}{n-2}}{\binom{m-1}{n-1}}\cdots\cdot\frac{\binom{m-(n-1)-1}{n-(n-1)-1}}{\binom{m-(n-1)}{n-(n-1)}} \\
&= 1 + \frac{1}{\binom{m}{n}}\left[\binom{m-1}{n-1} + \binom{m-2}{n-2} + \cdots + \binom{m-n}{n-n}\right] \\
&= 1 + \frac{1}{\binom{m}{n}}\left[\binom{m-1}{m-n} + \binom{m-2}{m-n} + \cdots + \binom{m-n+1}{m-n} + \binom{m-n}{m-n}\right] \\
&= 1 + \frac{1}{\binom{m}{n}}\left[\binom{m}{m-n+1} \underbrace{-\binom{m-1}{m-n+1} + \binom{m-1}{m-n+1} - \binom{m-2}{m-n+1} + \cdots + \binom{m-n+2}{m-n+1}} - \binom{m-n+1}{m-n+1} + \binom{m-n}{m-n}\right] \\
&= 1 + \frac{1}{\binom{m}{n}}\cdot\binom{m}{m-n+1} = 1 + \frac{n!(m-n)!}{m!}\cdot\frac{m!}{(m-n+1)!(n-1)!} \\
&= 1 + \frac{n}{m-n+1} = \frac{m+1}{m-n+1} = \frac{1}{1-\frac{n}{m+1}} \\
&\approx \frac{1}{1-\alpha}
\end{aligned}$$

$$\begin{aligned}
\text{查找成功时}S_n &= \textstyle\sum_{i=1}^{n} p_i C_i = \sum_{i=0}^{n-1} p_i U_i \\
&\xlongequal{p_i=\frac{1}{n}} \frac{1}{n}\textstyle\sum_{i=0}^{n-1} U_i = \frac{1}{n}\sum_{i=0}^{n-1} \frac{1}{1-\frac{i}{m}} \\
&\approx \frac{1}{n}\int_0^n \frac{1}{1-\frac{i}{m}}di = \frac{m}{n}\int_0^n \frac{1}{1-\frac{i}{m}}d(\frac{i}{m}) = \frac{m}{n}\int_0^{\frac{n}{m}} \frac{1}{1-x}d(x) = \frac{1}{\alpha}\int_0^{\alpha}\frac{1}{1-x}dx = -\frac{1}{\alpha}\ln(1-x)|_0^{\alpha} \\
&\approx -\frac{1}{\alpha}\ln(1-\alpha)
\end{aligned}$$



$P_r = \frac{\binom{m-r}{n-r+1}}{\binom{m}{n}}, \sum_{r=1}^{n+1} P_r = \sum_{r=1}^{n+1} \frac{\binom{m-r}{n-r+1}}{\binom{m}{n}} = 1$

$$\begin{aligned}
U_n &= \textstyle\sum_{r=1}^{n+1} r P_r \\
&= \textstyle m + 1 - (m+1)\sum_{r=1}^{n+1} P_r + \sum_{r=1}^{n+1} r P_r \\
&= \textstyle m + 1 - \sum_{r=1}^{n+1} (m+1-r) P_r \\
&= m + 1 - \textstyle\sum_{r=1}^{n+1} (m+1-r) \frac{\binom{m-r}{n-r+1}}{\binom{m}{n}} \\
&= m + 1 - \frac{1}{\binom{m}{n}} \textstyle\sum_{r=1}^{n+1} (m+1-r) \binom{m-r}{m-n-1} \\
&= m + 1 - \frac{1}{\binom{m}{n}} \textstyle\sum_{r=1}^{n+1} (m-n) \binom{m-r+1}{m-n} \\
&= m + 1 - \frac{(m-n)}{\binom{m}{n}}\left[\binom{m}{m-n} + \cdots + \binom{m-n+1}{m-n} + \binom{m-n}{m-n}\right] \\
&= m + 1 - \frac{(m-n)}{\binom{m}{n}}\left[\binom{m+1}{m-n+1} \underbrace{-\binom{m}{m-n+1} + \binom{m}{m-n+1} - \binom{m-1}{m-n+1} + \cdots + \binom{m-n+2}{m-n+1}} - \binom{m-n+1}{m-n+1} + \binom{m-n}{m-n}\right] \\
&= m + 1 - \frac{(m-n)}{\binom{m}{n}}\binom{m+1}{m-n+1} = m + 1 - (m-n)\frac{m+1}{m+1-n} = (m+1)(1 - \frac{m-n}{m-n+1}) \\
&= \frac{m+1}{m-n+1} = \frac{1}{1-\frac{n}{m+1}} \approx \frac{1}{1-\alpha} \\
S_n &= \frac{1}{n}\textstyle\sum_{k=0}^{n-1} U_k \\
&= \frac{1}{n}\left[\frac{m+1}{m+1} + \frac{m+1}{m} + \cdots + \frac{m+1}{m-n+2}\right] \\
&= \frac{m+1}{n}(H_{m+1} - H_{m-n+1}) \\
&= \frac{m+1}{n}(\ln\frac{m+1}{m+1-n}) \\
&\approx -\frac{1}{\alpha}\ln(1-\alpha)
\end{aligned}$$

非链地址处理冲突的哈希表中删除记录，填入特殊符号

10 内部排序

10.1 概述

```
#define MAXSIZE 20
typedef int KeyType;
typedef struct{
    KeyType key;
    InfoType ontherinfo;
}RedType;
typedef struct{
    RedType r[MAXSIZE+1];
    int length;
}Sqlist;
```

10.2 插入排序

10.2.1 直接插入

```
void InsertSort(Sqlist &L){
    //对顺序表做直接插入排序
    for(i=2;i<=L.length;i++){
        if(LT(L.r[i].key,L.r[i-1].key)){ //"<",需将L.r[i]插入有序子表
            L.r[0]=L.r[i];              //复制为哨兵
            L.r[i]=L.r[i-1];
            for(j=i-2;LT(L.r[0].key,L.r[j].key);--j)
                L.r[j+1]=L.r[j];        //记录后移
            L.r[j+1]=L.r[0];            //插入正确位置
            printf("vine");
        }
    }
}//InsertSort
```

10.2.2 其他插入

折半插入

```
void BInsertSort(Sqlist &L){
    //对顺序表做折半插入排序
    for(i=2;i<=L.length;i++){
        L.r[0]=L.r[i];                  //将L.r[i]暂存到L.r[0]
        low=1,high=i-1;
        while(low<=high){               //在L.r[low...high]中折半查找有序插入位置
            m=(low+high)/2;             //折半
            if(LT(L.r[0].key,L.r[m].key)) high=m-1; //插入点在高
            else low =m+1;                          //插入点在低
        }
        for(j=i-1;j>=high+1;--j) L.r[j+1]=L.r[j];   //记录后移
        L.r[high+1]=L.r[0];                         //插入
    }
}//BInsertSort
```

二路插入

表插入

希尔排序

```
void ShellInsort(Sqlist &L, int dk){
    //对顺序表做希尔插入排序
    //1.位置增量dk
    //2.L.r[0]是暂存不是哨兵，j<=0时插入位置已找到
    for(i=dk+1;i<=L.length;i++){
        if(LT(L.r[i].key,L.r[i-dk].key)){ //"<",需将L.r[i]插入有序子表
            L.r[0]=L.r[i];                  //暂存L.r[0]
            for(j=i-dk;j>0 && LT(L.r[0].key,L.r[j].key);j-=dk)
                L.r[j+dk]=L.r[j];            //记录后移,查找插入位置
            L.r[j+dk]=L.r[0];                //插入正确位置
        }
    }
}//ShellInsort

void ShellSort(Sqlist &L, int dk[],int t){
    //按增量序列 dk[0...t-1]对顺序表做希尔插入排序
    for(k=0;k<t;k++){
        ShellInsort(L,dk[k]);      //一趟增量为dk[k]的插入排序
    }
}//ShellSort
```

10.3 快速

10.3.1 起泡排序

```
void BubbleSort(int a[],int n){
    for(i=n-1,change=TRUE;i>=1 && change;--i){
        change=FALSE;
        for(j=0;j<i,j++){
            if(a[j]>a[i]){SWAP(a[j],a[i]);change=TRUE;}
        }
    }
}
```

10.3.2 快速排序

```
void Partition(Sqlist &L,int low,int high){
    //交换顺序表L中子序列L.r[low...high]的记录， 枢轴记录到位， 返回位置此时
    //在枢轴前（后）记录不大于（不小于）它
    L.r[0]=L.r[low];          //第一个记录做枢轴
    pivotkey=L.r[low].key;    //枢轴记录关键字
    while(low<high){          //从表的两端交替向中间扫描
        while(low<high && L.r[high].key>=pivotkey) --high;
        L.r[low]=L.r[high];   //小的左移
        while(low<high && L.r[low]<=pivotkey) ++low;
        L.r[high]=L.r[low];   //大的右移
    }
    L.r[low]=L.r[0];          //枢轴到位
    return low;               //返回枢轴位置
}

void QSort(Sqlist &L,int low,int high){
    //对顺序表L中子序列L.r[low...high]作快速排序
    if(low<high){                         //长度大于1
        pivotkey=Partition(L,low,high);   //将L.r[low...high]一分为二
        QSort(L,low,pivotkey-1);          //低子表递归
        QSort(L,pivotkey+1,high);         //高子表递归
    }
}

void QuickSort(Sqlist &L){
    //对顺序表L作快速排序
    QSort(L,1,L.length)
}
```

10.4 选择排序

10.4.1 简单选择排序

```
void SelectSort(Sqlist &L ){
    for(i=1;i<L.length;i++){        //选择第i小的记录，并交换到位
        j=SelectMinKey(L,i);        //在L.r[i...L.length]中选择key最小的记录
        if(i!=j) SWAP(L.r[i],L.r[j]) //与第i个记录交换
    }
}
```

10.4.2 树形排序

10.4.3 堆排序

```
void HeapAdjust(HeapType &H,int s ,int m ){
    //已知H.r[s...m]中记录除H.r[s]外均满足堆的定义
    //调整H.r[s]使得H.r[s...m]称为大顶堆
    rc=H.r[s];
    for(j=2*s;j<=m;j*=2){    //沿key较大的孩子节点向下筛选
        if(j<m && LT(H.r[j].key,H.r[j+1].key)) ++j; //j为key较大的记录的下标
        if(!LT(rc.key,H.r[j].key)) break;           //rc插入s
        H.r[s]=H.r[j];s=j;                           //插入
    }
    H.r[s]=rc;
}

void HeapSort(HeapType &H ){
    for(i=H.length/2;;i>0;--i)            //把H.r[1...H.length]建成大顶堆
        HeapAdjust(H,i,H.length);
    for(i=H.length;i>1;--i){               //堆顶记录和未经排序子序列H.r[1...i]中最后一个记录交换
        SWAP(H.r[1],H.r[i]);               //将[1...i-1]建成大顶堆
        HeapAdjust(H,1,i-1);
    }
}
```

10.5 归并排序

```
void Merge(RcdType SR[],RcdType & TR[],int i,int m,int n ){
    //将有序的SR[i...m],SR[m+1,n]归并为有序的TR[i...n]
    for(j=m+1,k=i;i<=m && j<=n;++k){  //将SR中记录从小到大并入TR
        if(LQ(SR[i].key,SR[j].key))  TR[k]=SR[i++];
        else TR[k]=SR[j++];
    }
    if(i<=m) TR[K...n]=SR[i...m];       //将剩余的SR[i...m]复制到TR[K...n]
    if(j<n)  TR[k...n]=SR[j...n];       //将剩余的SR[j...n]复制到TR[K...n]
}

void Msort(RcdType SR[],RcdType & TR1[],int s,int t){
    //将SR[s...t]归并为TR1[s...t]
    if(s==t) TR1[s]=SR[s];
    else{
        m=(s+t)/2;            //将SR[s...t]平分为SR[s...m], SR[m+1...t]
        Msort(SR,TR2,s,m);    //递归SR[s...m] 为有序 TR2[s...m]
        Msort(SR,TR2,m+1,t);  //递归SR[m+1...t]为有序 TR2[m+1...t]
        Merge[TR2,TR1,s,m.t]; //将TR2[s...m],TR2[m+1...t] 归并到 TR1[s...t]
    }
}

void MergeSort(Sqlist &L){
    Msort(L.r,L.r,1,L.length);
}
```

10.6 基数排序

10.6.1 多关键字的排序

10.6.2 链式基数排序

```
#define MAX_NUM_OF_KEY 8
#define RADIX 10
#define MAX_SPACE 10000
typedef struct{
    KeysType Keys[MAX_NUM_OF_KEY];
    InfoType otheritems;
    int next;
}SLCell;

typedef struct{
    SLCell r[MAX_SPACE];
    int keynum;
    int recnum;
}SLList;

typedef int ArrType[RADIX];


void Distribute(SLCell &r,int i,ArrType &f,ArrType &e){
    //静态链表L的r域中记录已按keys[0]...keys[i-1]有序
    //本算法按第i个关键字keys[i]建立RADIX个子表，使得同一子表中记录的keys[i]相同
    //f[0...RADIX-1],e[0...RADIX-1]分别指向各子表中第一个和最后一个记录
    for(j=0;j<Radix;++j) f[j]=0 //各子表初始化为空
    for(p=r[0].next;p;p=r[p].next){
        j=ord(r.[p].keys[i]);   //ord将记录中第i个关键字映射到[0...RADIX-1]
        if(!f[j]) f[j]=p;
        else r[e[j]].next=p;
        e[j]=p;                 //将p指向的结点插入第j个子表中
    }
}//Distribute

void Collect(SLCell &r,int i,ArrType f,ArrType e){
    //本算法按keys[i]从小至大地将f[0...RADIX-1]所指个子表依次链接成一个链表
    //e[0...RADIX-1]为各子表的尾指针
    for(j=0;!f[j];j=succ(j));    //找到第一个非空子表，succ为求后继函数
    r[0].next=f[j];t=e[j];       //r[0].next指向第一个非空子表中第一个节点
    while(j<RADIX){
        for(j=succ(j);j<RADIX-1 && !f[j];j=succ(j));  //找到下一个非空子表
        if(f[j] {r[t].next=f[j];t=e[j];})             //链接两个非空子表
    }
    r[t].next=0;                                      //t指向最后一个非空子表中的最后一个节点
}//Collect

void RadixSort(SLList &L){
    //L是采用静态链表表示的顺序表
    //对L作基数排序，使得L成为按关键字自小到大的有序静态链表，L.r[0]为头节点
    for(i=0;i<L.recnum;++i) L.r[i].next=i+1;
    L.r[L.recnum].next=0;          //将改造为静态链表
    for(i=0;i<L.keynum;++i){       //按最低位优先依次对各关键字进行分配和收集
        Distribute(L.r,i,f,e);     //第i趟分配
        Collect(L.r,i,f,e);        //第i趟收集
    }
}//RadixSort
```

10.7 各内部排序方法的比较讨论

排序方法	平均时间	最坏情况	辅助存储
简单排序	$O(n^2)$	$O(n^2)$	$O(1)$
快速排序	$O(nlogn)$	$O(n^2)$	$O(logn)$
堆排序	$O(nlogn)$	$O(nlogn)$	$O(1)$
归并排序	$O(nlogn)$	$O(nlogn)$	$O(n)$
基数排序	$O(d(n+rd))$	$O(d(n+rd))$	$O(rd)$

简单排序包括除希尔排序之外所有插入排序，起泡排序，简单选择排序，直接插入排序

地址向量重排算法

```
void Rearrange(Sqlist &L,int adr[] ){
    //adr给出顺序表的有序次序， 即L.r[adr[i]]是第i小记录
    //本算法按adr重排L.r使其有序
    for(i=1;i<L.length;++i){
        if(adr[i]!=i){
            j=i;L.r[0]=L.r[i];            //暂存记录
            while(adr[j]!=i){             //调整L.r[adr[j]]的记录到位直到adr[j]=i为止
                k=adr[j];L.r[j]=L.r[k];
                adr[j]=j;j=k;
            }
            L.r[j]=L.r[0];adr[j]=j;       //记录按序到位
        }
    }
}
```

11　外部排序

12 文件