Data Structure

Vine

2022 年 7 月 4 日

目录

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

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

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

9.1.2 有序表的查找

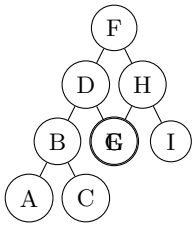
折半查找

```
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
}
```

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

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

```
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;
}
```

9.1.4 索引顺序表的查找

9.2 动态查找表

9.3 哈希表

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 ontheritems;
   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