10.6 基数排序

基数排序是一种借助“多关键字排序”的思想来实现“单关键字排序”的内部排序算法。

10.6.1 多关键字的排序

10.6.2 链式基数排序

10.6.1 多关键字的排序

n **个记录的序列** $\{R_1, R_2, \ldots, R_n\}$ **对关键字** $(K_i^0, K_i^1, \ldots, K_i^{d-1})$ **有序**是指：

对于序列中任意两个记录 R_i 和 $R_j(1 \le i < j \le n)$ 都**满足**下列**（词典）有序**关系：

$$(K_i^0, K_i^1, \ldots, K_i^{d-1}) < (K_j^0, K_j^1, \ldots, K_j^{d-1})$$

其中: K^0 **被称为 “最主（高）” 位关键字**

K^{d-1} **被称为 “最次（低）” 位关键字**

实现多关键字排序通常有两种作法:

最高位优先法（MSD）：

先对K^0进行排序，并按K^0的不同值将记录序列分成若干子序列之后，分别对K^1进行排序，……，依次类推，直至最后对最次位关键字排序完成为止。

最低位优先法（LSD）：

先对K^{d-1}进行排序，然后对K^{d-2}进行排序，依次类推，直至对最主位关键字K^0排序完成为止。

例如:学生记录含三个关键字:**系别、班号**和**班内的序列号**，其中以系别为最主位关键字。

LSD的排序过程如下:

无序序列	3,2,30	1,2,15	3,1,20	2,3,18	2,1,20
对K^2排序	1,2,15	2,3,18	3,1,20	2,1,20	3,2,30
对K^1排序	3,1,20	2,1,20	1,2,15	3,2,30	2,3,18
对K^0排序	1,2,15	2,1,20	2,3,18	3,1,20	3,2,30

排序过程中不需要根据“前一个”关键字的排序结果，将记录序列分割成若干个(“前一个”关键字不同的)子序列。

10.6.2 链式基数排序

假如多关键字的记录序列中，每个关键字的取值范围相同，则按LSD法进行排序时，可以采用“**分配-收集**”的方法，其好处是不需要进行关键字间的比较。

对于数字型或字符型的**单关键字**，可以**看成**是由多个数位或多个字符构成的**多关键字**，此时可以**采用**这种“**分配-收集**”的办法**进行排序**，**称作基数排序法**。

在描述算法之前，尚需定义新的数据类型

```
#define MAX_NUM_OF_KEY  8 //关键字项数的最大值
#define RADIX   10
        //关键字基数，此时是十进制整数的基数
#define MAX_SPACE  10000
 typedef struct {
    KeysType keys[MAX_NUM_OF_KEY];//关键字
    InfoType otheritems;        //其他数据项
    int next;
}SLCell;                    //静态链表的结点类型
```

```
typedef struct {
  SLCell  r[MAX_SPACE];
  //静态链表的可利用空间，r[0]为头结点
  int  keynum;  //记录的当前关键字个数
  int  recnum;  //静态链表的当前长度
}SLList;          //静态链表类型
typedef int ArrType[RADIX];//指针数组类型
```

例如：对下列这组关键字

{209, 386, 768, 185, 247, 606, 230, 834, 539 }

首先按其 “个位数” 取值分别为 0, 1, …, 9，**“分配”** 成 10 组，之后按从 0 至 9 的顺序将 它们 **“收集”** 在一起；

然后按其 “十位数” 取值分别为 0, 1, …, 9 **“分配”** 成 10 组，之后再按从 0 至 9 的顺序将它们 **“收集”** 在一起；

最后按其“百位数”重复一遍上述操作。

在计算机上实现基数排序时，为减少所需辅助存储空间，应采用链表作存储结构，即链式基数排序，具体作法为：

1．待排序记录以指针相链，构成一个链表；

2．“分配”时，按当前“关键字位”所取值，将记录分配到不同的“链队列”中，每个队列中记录的“关键字位”相同；

3．“收集”时，按当前关键字位取值从小到大将各队列首尾相链成一个链表；

4．对每个关键字位均重复 2）和 3）两步。

例如：

p→369→367→167→239→237→138→230→139

进行第一次分配

f[0] →230←r[0]

f[7]→367→167→237← r[7]

f[8] →138←r[8]

f[9]→369→239→139←r[9]

进行第一次收集

p→230→367→167→237 →138 →369→239→139

p→230→367→167→237→138→369→239→139

进行第二次分配

f[3]→230 →237 →138 →239 →139 ←r[3]

f[6] →367 →167 →369 ← r[6]

进行第二次收集

p→230→237→138→239→139 →367→167→369

p→230→237→138→239→139→367→167→369

进行第三次分配

f[1] →138 →139 →167 ← r[1]

f[2] →230 →237 →239 ← r[2]

f[3] →367 →369 ← r[3]

进行第三次收集之后便得到记录的有序序列

p→138→139→167 →230→237→239 →367→369

提醒注意：

1．“分配”和“收集”的实际操作仅为修改链表中的指针和设置队列的头、尾指针；

2．为查找使用，该链表尚需应用算法Arrange将它调整为有序表。

基数排序的时间复杂度为O(d(n+rd))

其中：分配为O(n)

收集为O(rd)(rd为“基”)

d为“分配-收集”的趟数

```
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，使同一子表中记录的keys[i]相同
.f[0..RADIX-1]和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;          //各子表初始化为空表
    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所指的结点插入第j个子表
    }
}// Distribute
```

算法 10.15　链式基数排序中一趟分配的算法

```
void Collect (SLCell &r, int i, ArrType f, ArrType e ) {
//本算法按keys[i]自小至大的将f[0..RADIX-1]所指各子表依次链接成
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结点
  while ( j<RADIX){
      for ( j = succ(j); j<RADIX-1 && !f[j]; j = succ(j) );
                 //找下一个非空子表
      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的最后一个结点
}// Collect
```

算法 10.16　一趟收集的算法

```
void RadixSort (SList  &L ) {
// L是采用静态链表表示的顺序表。对L做基数排序，使
//得L成为按关键字自小到达的有序静态链表，L.r[0]为
//头结点。
    for (i=0; i<L.recnum; ++i)  L.r[i].next = i+1;
    L.r[L.recnum].next = 0;        //将L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
```

算法 10.17　链式基数排序的算法

10.6 基数排序

基数排序是一种借助“多关键字排序”的思想来实现“单关键字排序”的内部排序算法。

10.6.1 多关键字的排序

10.6.2 链式基数排序

10.6.1 多关键字的排序

n **个记录的序列** $\{R_1, R_2, \ldots, R_n\}$ **对关键字** $(K_i^0, K_i^1, \ldots, K_i^{d-1})$ **有序**是指：

对于序列中任意两个记录 R_i 和 $R_j(1 \le i < j \le n)$ 都**满足**下列**（词典）有序**关系：

$$(K_i^0, K_i^1, \ldots, K_i^{d-1}) < (K_j^0, K_j^1, \ldots, K_j^{d-1})$$

其中: K^0 **被称为 “最主（高）” 位关键字**

K^{d-1} **被称为 “最次（低）” 位关键字**

实现多关键字排序通常有两种作法:

最高位优先法（MSD）：

先对K^0进行排序，并按K^0的不同值将记录序列分成若干子序列之后，分别对K^1进行排序，……，依次类推，直至最后对最次位关键字排序完成为止。

最低位优先法（LSD）：

先对K^{d-1}进行排序，然后对K^{d-2}进行排序，依次类推，直至对最主位关键字K^0排序完成为止。

例如:学生记录含三个关键字:**系别、班号**和**班内的序列号**，其中以系别为最主位关键字。

LSD的排序过程如下:

无序序列	3,2,30	1,2,15	3,1,20	2,3,18	2,1,20
对K^2排序	1,2,15	2,3,18	3,1,20	2,1,20	3,2,30
对K^1排序	3,1,20	2,1,20	1,2,15	3,2,30	2,3,18
对K^0排序	1,2,15	2,1,20	2,3,18	3,1,20	3,2,30

排序过程中不需要根据“前一个”关键字的排序结果，将记录序列分割成若干个(“前一个”关键字不同的)子序列。

10.6.2 链式基数排序

假如多关键字的记录序列中，每个关键字的取值范围相同，则按LSD法进行排序时，可以采用“**分配-收集**”的方法，其好处是不需要进行关键字间的比较。

对于数字型或字符型的**单关键字**，可以**看成**是由多个数位或多个字符构成的**多关键字**，此时可以**采用**这种“**分配-收集**”的办法**进行排序**，**称作基数排序法**。

在描述算法之前，尚需定义新的数据类型

```
#define MAX_NUM_OF_KEY  8 //关键字项数的最大值
#define RADIX   10
        //关键字基数，此时是十进制整数的基数
#define MAX_SPACE  10000
 typedef struct {
    KeysType keys[MAX_NUM_OF_KEY];//关键字
    InfoType otheritems;        //其他数据项
    int next;
}SLCell;                    //静态链表的结点类型
```

```
typedef struct {
  SLCell  r[MAX_SPACE];
  //静态链表的可利用空间，r[0]为头结点
  int  keynum;  //记录的当前关键字个数
  int  recnum;  //静态链表的当前长度
}SLList;          //静态链表类型
typedef int ArrType[RADIX];//指针数组类型
```

例如：对下列这组关键字

{209, 386, 768, 185, 247, 606, 230, 834, 539 }

首先按其 “个位数” 取值分别为 0, 1, …, 9，**“分配”** 成 10 组，之后按从 0 至 9 的顺序将 它们 **“收集”** 在一起；

然后按其 “十位数” 取值分别为 0, 1, …, 9 **“分配”** 成 10 组，之后再按从 0 至 9 的顺序将它们 **“收集”** 在一起；

最后按其“百位数”重复一遍上述操作。

在计算机上实现基数排序时，为减少所需辅助存储空间，应采用链表作存储结构，即链式基数排序，具体作法为：

1．待排序记录以指针相链，构成一个链表；

2．“分配” 时，按当前“关键字位”所取值，将记录分配到不同的 “链队列” 中，每个队列中记录的 “关键字位” 相同；

3．“收集”时，按当前关键字位取值从小到大将各队列首尾相链成一个链表；

4．对每个关键字位均重复 2）和 3）两步。

例如：

p→369→367→167→239→237→138→230→139

进行第一次分配

f[0] →230←r[0]

f[7]→367→167→237← r[7]

f[8] →138←r[8]

f[9]→369→239→139←r[9]

进行第一次收集

p→230→367→167→237 →138 →369→239→139

p→230→367→167→237→138→369→239→139

进行第二次分配

f[3]→230 →237 →138 →239 →139 ←r[3]

f[6] →367 →167 →369 ← r[6]

进行第二次收集

p→230→237→138→239→139 →367→167→369

p→230→237→138→239→139→367→167→369

进行第三次分配

f[1] →138 →139 →167 ← r[1]

f[2] →230 →237 →239 ← r[2]

f[3] →367 →369 ← r[3]

进行第三次收集之后便得到记录的有序序列

p→138→139→167 →230→237→239 →367→369

提醒注意：

1．“分配”和“收集”的实际操作仅为修改链表中的指针和设置队列的头、尾指针；

2．为查找使用，该链表尚需应用算法Arrange将它调整为有序表。

基数排序的时间复杂度为O(d(n+rd))

其中：分配为O(n)

收集为O(rd)(rd为“基”)

d为“分配-收集”的趟数

```
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，使同一子表中记录的keys[i]相同
.f[0..RADIX-1]和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;          //各子表初始化为空表
    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所指的结点插入第j个子表
    }
 }// Distribute
```

算法 10.15　链式基数排序中一趟分配的算法

```
void Collect (SLCell &r, int i, ArrType f, ArrType e ) {
//本算法按keys[i]自小至大的将f[0..RADIX-1]所指各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结点
  while ( j<RADIX){
      for ( j = succ(j); j<RADIX-1 && !f[j]; j = succ(j) );
                 //找下一个非空子表
      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的最后一个结点
}// Collect
```

算法 10.16　一趟收集的算法

```
void RadixSort (SList  &L ) {
// L是采用静态链表表示的顺序表。对L做基数排序，使
//得L成为按关键字自小到达的有序静态链表，L.r[0]为
//头结点。
      for (i=0; i<L.recnum; ++i)  L.r[i].next = i+1;
      L.r[L.recnum].next = 0;        //将L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
```

算法 10.17　链式基数排序的算法