

# Data structures Final Assignment

Yaoming Shi, Hanlin Qian, Yizhou Xue

UCAS

2025.6.25

# 多关键字排序

在本次大作业中，一共运用了三种方式对多关键字进行排序

- **简单 LSD 排序**：用 LSD 办法，和简单的冒泡排序对多关键字序列进行稳定的原地排序
- **基数排序**：基于分配和收集和 LSD 的思想，对多关键字序列进行稳定的非原地排序
- **MSD 排序**：根据拓展的要求，用 MSD 进行递归排序，来进行效果的比较，

# LSD 排序

根据关键词的优先级从高到低分别进行三次遍历的排序

```
1 void lsd_sort(int ** arr_record,int r,int k){
2     for(int i = k - 1; i >= 0; i--){ //对每个关键字进行排序, 先从优先级低的排序, 采用冒泡排序
3         int* temp;
4         int j = r-1;
5         int sorted = FALSE;
6         while(!sorted){
7             sorted = TRUE;
8             for(int m = 0; m < j; m++){
9                 if(arr_record[m+1][i] < arr_record[m][i]){
10                     temp = arr_record[m+1];
11                     arr_record[m+1] = arr_record[m];
12                     arr_record[m] = temp;
13                     sorted = FALSE; //表示遇到了逆序对, sorted置为0
14                 }
15             }
16             j--;
17         }
18     }
19 }
```

# 基数排序

严格按照课上的思路，将关键字分成若干个链表，最后再 collect 起来

```
1 void RadixSort(SlList *L){
2     int* f;
3     int* e;
4     //为f,e分配空间
5     f = (int *)malloc(RADIX * sizeof(int));
6     e = (int *)malloc(RADIX * sizeof(int));
7     for(int i = L->keynum - 1; i >= 0; i--){//按照关键字排序，最右侧的先优先级最低
8         Distribute(L->array,i,f,e);
9         Collect(L->array,i,f,e);
10    }
11    free(f); free(e);
12 }
```

## 基数排序

```
1 void Distribute(SlCell* array, int i, int* f, int* e){
2     for(int j = 0; j < RADIX; j++) f[j] = 0; //初始化f数组
3     for(int p = array[0].next; p; p = array[p].next){
4         //将p所指的结点插入到相应的子链表当中
5         int m = array[p].keys[i];
6         if(!f[m]) {
7             f[m] = p; //如果f[j]为空，连接p
8         }
9         else{
10            array[e[m]].next = p;
11        }
12        e[m] = p;
13    }
14 }
15 void Collect(SlCell* array, int i, int* f, int* e){
16     int j;
17     for(j = 0; !f[j]; j++)//找到第一个非空子表
18     array[0].next = f[j];
19     int t = e[j];
20     while(j < RADIX){
21         for(j++; j < RADIX; j++)//找下一个非空的子表
22             if(f[j]){
23                 array[t].next = f[j]; //将两个表合并
24                 t = e[j];
25             }
26     }
27     array[t].next = 0;
28     //最后一个结点指向表头
29 }
30 }
```

## 分配和收集函数

# MSD 排序


首先将最高优先级的关键字排序, 之后对相同关键字的元素进行递归排序, 直到所有关键字都被处理完毕

```
1 void msd_sort(int** array_record, int left, int right, int k, int key_num){
2     if(key_num >= k || right <= left + 1) return ;//表示如果当前关键字已经排序完成, 或者当前同一关键字中只有1个元素
3     bubblesort(array_record, left, right, key_num); //对key_num进行排序
4
5     int start = left; //如果相同关键字有多个, 则递归调用msd, 对下一关键字进行排序
6     while(start < right){
7         int end = start + 1;
8         while(end < right && array_record[start][key_num] == array_record[end][key_num]){
9             end++;
10        }
11        msd_sort(array_record, start, end, k, key_num + 1); //对更低优先级的关键字排序
12        start = end; //继续向下做排序
13    }
14 }
15
```

msd 算法

# 其它关键代码-随机数生成

为了防止每次运行生成的随机数都相同, 于是利用 `srand` 函数生成了随机数种子



```
1  srand((unsigned)time(0)); // 添加这行以随机化种子
2  int r = (rand() % MAX_RECORD) + 1;
3  int k = (rand() % MAX_KEY) + 1;
```

# 循环运行取平均值

为了让运行的结果更加有统计意义, 在 main 函数中重复运行 LOOP 次, 最终取平均值结果

```
1 //在程序中实现多次计算取平均值
2 while(1 < LOOP){
3     //中间代码段
4     l++;
5 }
6
7 printf("=====\n");
8 printf("LSD_BUBBLE_AVRAGE_TIME = %.7f\n", time_lsd/LOOP);
9 printf("=====\n");
10
11 printf("=====\n");
12 printf("MSD_BUBBLE_AVRAGE_TIME = %.7f\n", time_msd/LOOP);
13 printf("=====\n");
14
15 printf("=====\n");
16 printf("LSD_DISTRIBUTE&COLLET = %.7f\n", time_cnd/LOOP);
17 printf("=====\n");
18
19 return 0;
20 }
```

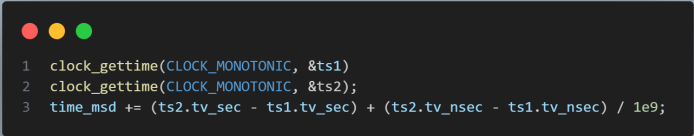
重复运行

Woring on sort [1]	record number[3964]	keys per record[5]
Woring on sort [2]	record number[4618]	keys per record[2]
Woring on sort [3]	record number[5338]	keys per record[5]
Woring on sort [4]	record number[6176]	keys per record[5]
Woring on sort [5]	record number[8257]	keys per record[1]
Woring on sort [6]	record number[1098]	keys per record[2]
Woring on sort [7]	record number[3329]	keys per record[1]
Woring on sort [8]	record number[3431]	keys per record[5]
Woring on sort [9]	record number[1049]	keys per record[3]
Woring on sort [10]	record number[1137]	keys per record[3]
Woring on sort [11]	record number[7442]	keys per record[5]
Woring on sort [12]	record number[4455]	keys per record[3]
Woring on sort [13]	record number[8216]	keys per record[5]
Woring on sort [14]	record number[4529]	keys per record[3]
Woring on sort [15]	record number[7060]	keys per record[5]
Woring on sort [16]	record number[9195]	keys per record[4]
Woring on sort [17]	record number[4235]	keys per record[2]
Woring on sort [18]	record number[2580]	keys per record[4]
Woring on sort [19]	record number[1266]	keys per record[3]
Woring on sort [20]	record number[9161]	keys per record[5]
Woring on sort [21]	record number[319]	keys per record[5]
Woring on sort [22]	record number[7818]	keys per record[2]
Woring on sort [23]	record number[1744]	keys per record[5]
Woring on sort [24]	record number[2411]	keys per record[3]
Woring on sort [25]	record number[9771]	keys per record[2]

运行时的过程显示

# 提高时间精度

由于在题目给的输入规模下, `time.h` 中的 `clock()` 函数的精度太低, 最多只能读到毫秒, 于是用 `timespec` 来达到纳秒级, 获取更高精度的结果



```
1 clock_gettime(CLOCK_MONOTONIC, &ts1)
2 clock_gettime(CLOCK_MONOTONIC, &ts2);
3 time_ms += (ts2.tv_sec - ts1.tv_sec) + (ts2.tv_nsec - ts1.tv_nsec) / 1e9;
```

提高时间精度的代码段



# 最终结果

最终运行 1000 次取平均值的结果 (记录数不超过 10000, 关键字数不超过 5), 运行时间  $LSD > MSD > \text{基数排序}$  这符合我们的预期, 因为这三者的最差时间复杂度分别为  $O(n \times k)$ ,  $O(n \times k)$ ,  $O(n \times \log k)$ , 其中  $n$  为记录数,  $k$  为关键字数, 而 MSD 的平均复杂度更小, 因为所有的高优先级关键字都相同的概率显然更小

```
=====
LSD_BUBBLE_AVRAGE_TIME = 0.0961649
=====
=====
MSD_BUBBLE_AVRAGE_TIME = 0.0308656
=====
=====
LSD_DISTRIBUTE&COLLET = 0.0000581
=====
```

运行结果截图