# 7.7 Bin Sort & Radix Sort

分配排序&基数排序

## Binsort (1)

设A[n]为待排序数组，B[n]为临时数组

- Simple Binsort

称B[ ]中每个元素对应位置为一个Bin

*for i=0; i<n; i++*   *B[A[i]]=A[i]*

*for i=0; i<n; i++*   *A[i]=B[i]*

A[ ] = { 9, 5, 1, 7, 0, 6, 8, 2, 4, 3 }

- *Simple* Binsort 适用条件

太窄

① 关键字取值不能重复，因一个bin中只能放一个记录

② 关键字最大取值 <= 待排序数组尺寸 – 1

③ 关键字只能为非负整数

只能对关键值取值为 [0,n-1]的n个记录排序

## Binsort (2)

**Make each bin a list**
**Make B的长度为 Max(A[])+1**

- Extented Binsort

① 确定关键字的最大取值Max

② 定义数组 LList<Elem> B[Max+1]

③ 将关键字为K的记录依次放入链表B[K]中 O(*n*)

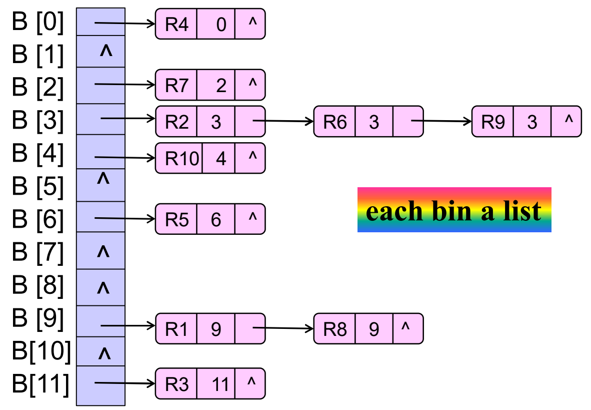
④ 从B[0]出发，顺序访问每个bin的每个记录，并将其放入A[ ]中 O(max(Max,*n*))

O (*n* + max(Max,n))

缺点：若Max很大，B[ ] 维数会很大（>> n）

记录序列： A[ ] = { R1, R2, R3, R4, R5, R6, R7, R8, R9, R10}
对应关键字 K[ ] = { 9, 3, 11, 0, 6, 3, 2, 9, 3, 4}

关键字 的取值范围 [0~11], 大于记录个数10， 且关键字有重复



## Binsort (3) —extented algorithm

```
template <class Elem>
void binSort(Elem A[], int n) {
 LList<Elem>  B[MaxKeyValue];  //允许 重复关键字，关键字取
   值范围可大于记录个数
 for (i=0; i<n; i++)  B[ A[i] ].append(A[i]);

 for (i=0;j=0; i<MaxKeyValue; i++)
   for (B[i].moveToStart(); B[i].currPos()<B[i].Length; B[i].next())
       { A[j] = B[i].getValue() ; j++;  }
}
```

**Cost:** O ($n$ + max(MaxKeyValue,n))
could be $\Theta(n^2)$ or arbitrarily worse

69

69

## BinSort Cost

- **In general the BinSort cost is:**

  **O ($n$ + max(MaxKeyValue,n))**

  ✓ **If range of keys is small, the cost is O($n$).**

  ✓ **If  range of keys is very large, the cost could be O($n^2$) or arbitrarily worse**

- **BinSort 需要额外空间**

  ✓ **n个data 和 n+MaxkeyValue 个 pointer 附加空间**

- **BinSort is stable**

70

70

## Radix Sort 基数排序 (1)

- 思路：按位排序

  ① 确定关键字的最大取值Max及其位数k

  ② 从低位到高位，进行k趟，第i趟根据第i位对上一趟结果进行Binsort，原始序列记为0趟结果

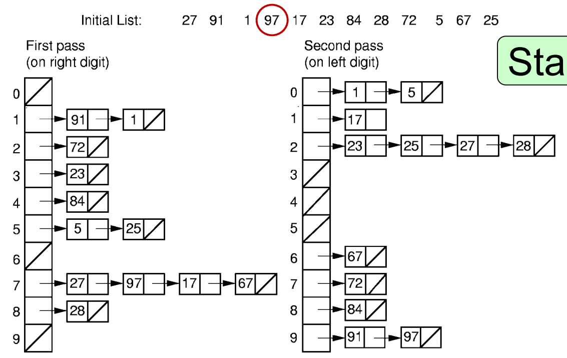
每趟 B[ ] 维数等于基数，如10（十进制数）

**O($kn$)**

71

71

## Radix Sort 基数排序 (2)--Llist based


Initial List: 27 91 1 97 17 23 84 28 72 5 67 25
First pass (on right digit)
0
1 91 1
2 72
3 23
4 84
5 5 25
6
7 27 97 17 67
8 28
9
Second pass (on left digit)
0 1 5
1 17
2 23 25 27 28
3
4
5
6 67
7 72
8 84
9 91 97

Stable？

**First pass result: ??**

**Second pass result: ??**

72

72

## Radix Sort 基数排序 (2)--array based

原序列：27 91 1 97 17 23 84 28 72 5 67 25

First pass: 91 1 72 23 84 5 25 27 97 17 67 28

Second pass: 1 5 17 23 25 27 28 67 72 84 91 97

Stable？

73

73

## Radix Sort (3)—Llist based

```
template <class Elem>
void radixSort(Elem A[], LList<Elem>  B[],  int n, int k, int r) {
    Elem item;
    int i, j, m, rtok;
    for (i=0, rtok=1; i<k; i++, rtok*=r)
      for (j=0; j<r; j++)   B[j].clear;
      for (j=0; j<n; j++)   B[(A[j]/rtok)%r ].append(A[j]);
      for (m=0, j = 0; m<r; m++)
        for (B[m].moveToStart(); B[m].currPos() < B[m].Length);  B[m].next())
           {  A[j] = B[m].getValue( ); j++;  }
}
```

记录数

基数（进制数）

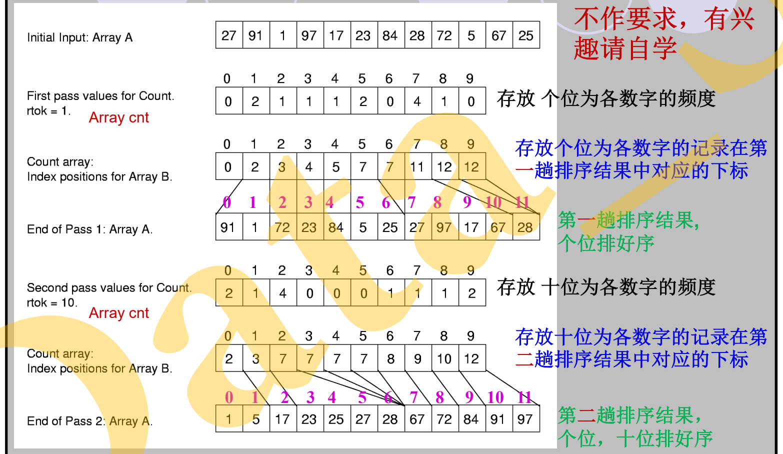
A[n]最大数的位数

$O(kn)$

74

74

## Radix Sort (4)--- array-based



Initial Input: Array A

| 27 | 91 | 1 | 97 | 17 | 23 | 84 | 28 | 72 | 5 | 67 | 25 |
|---|---|---|---|---|---|---|---|---|---|---|---|

First pass values for Count. rtok = 1. Array cnt

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 2 | 1 | 1 | 1 | 2 | 0 | 4 | 1 | 0 |

存放 个位为各数字的频度

Count array: Index positions for Array B.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 2 | 3 | 4 | 5 | 7 | 7 | 11 | 12 | 12 |

存放个位为各数字的记录在第一趟排序结果中对应的下标

End of Pass 1: Array A.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 91 | 1 | 72 | 23 | 84 | 5 | 25 | 27 | 97 | 17 | 67 | 28 |

第一趟排序结果，个位排好序

Second pass values for Count. rtok = 10. Array cnt

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 1 | 4 | 0 | 0 | 0 | 1 | 1 | 1 | 2 |

存放 十位为各数字的频度

Count array: Index positions for Array B.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 3 | 7 | 7 | 7 | 7 | 8 | 9 | 10 | 12 |

存放十位为各数字的记录在第二趟排序结果中对应的下标

End of Pass 2: Array A.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 5 | 17 | 23 | 25 | 27 | 28 | 67 | 72 | 84 | 91 | 97 |

第二趟排序结果，个位，十位排好序

不作要求，有兴趣请自学

75

75

## Radix Sort (4) --- array-based

```
template <class Elem>
void radix(Elem A[], Elem B[], int n, int k, int r, int cnt[]) {
// n:  # of records;   k:  关键字的最大位数；  r：  基数
//  A[n]:  待排序记录序列；cnt[r]，  B[n]: 临时数组；
  int i, j, rtok;
  for (i=0, rtok=1; i<k; i++, rtok*=r) {
     for (j=0; j<r; j++)   cnt[j] = 0;
     for(j=0; j<n; j++)   cnt[(A[j]/rtok)%r]++;
     for (j=1; j<r; j++)     cnt[j] = cnt[j-1] + cnt[j];
     for (j=n-1; j>=0; j--)    B[--cnt[(A[j]/rtok)%r] ] = A[j];
     for (j=0; j<n; j++)   A[j] = B[j];
  }
}
```

不作要求，有兴趣请自学

$O(kn)$

How do $n$, $k$, and $r$ relate to?

76

76

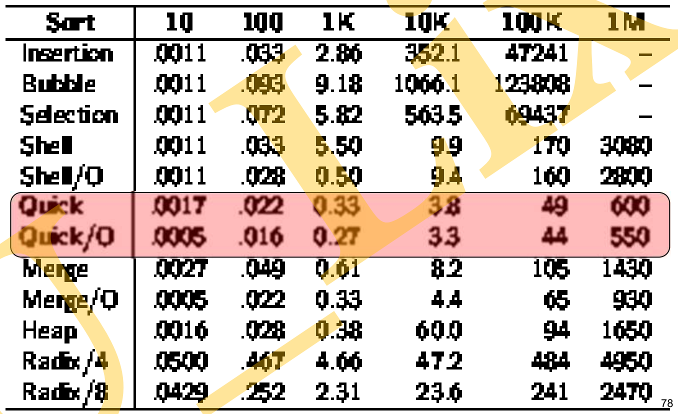
## Radix Sort Cost

- ➢ In general the Radix Sort cost is: O($kn$)
  - ✓ If range of keys is 0~$n$ , then k = cell($\log_r n$). thus, the cost is O($n \log_r n$)
  - ✓ If key range is small, that is k is small, then the cost can be O($n$).
- ➢ Radix Sort 需要额外空间
  - ✓LList-based: n个data 和n+r个pointer附加空间
  - ✓*Array-based: n+r* 个data 附加空间
- ➢ Radix Sort is stable

77

77

## Empirical Comparison (Windows)

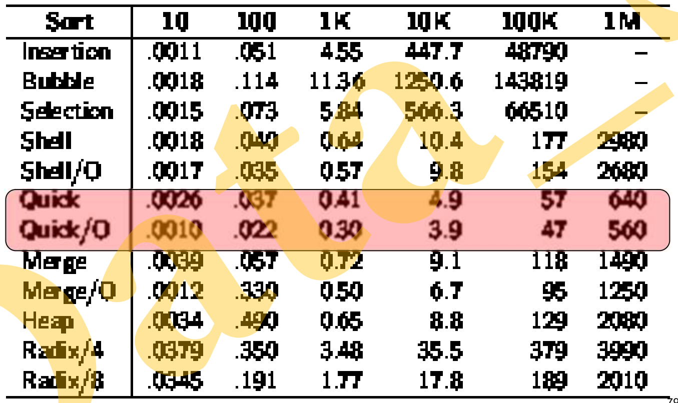


| Sort | 10 | 100 | 1K | 10K | 100K | 1M |
|---|---|---|---|---|---|---|
| Insertion | .0011 | .033 | 2.86 | 352.1 | 47241 | – |
| Bubble | .0011 | .093 | 9.18 | 1066.1 | 123808 | – |
| Selection | .0011 | .072 | 5.82 | 563.5 | 69437 | – |
| Shell | .0011 | .033 | 5.50 | 9.9 | 170 | 3080 |
| Shell/O | .0011 | .028 | 0.50 | 9.4 | 160 | 2800 |
| Quick | .0017 | .022 | 0.33 | 3.8 | 49 | 600 |
| Quick/O | .0005 | .016 | 0.27 | 3.3 | 44 | 550 |
| Merge | .0027 | .049 | 0.61 | 8.2 | 105 | 1430 |
| Merge/O | .0005 | .022 | 0.33 | 4.4 | 65 | 930 |
| Heap | .0016 | .028 | 0.38 | 60.0 | 94 | 1650 |
| Radix/4 | .0500 | .467 | 4.66 | 47.2 | 484 | 4950 |
| Radix/8 | .0429 | .252 | 2.31 | 23.6 | 241 | 2470 |

78

78

## Empirical Comparison (Linux)



| Sort | 10 | 100 | 1K | 10K | 100K | 1M |
|---|---|---|---|---|---|---|
| Insertion | .0011 | .051 | 4.55 | 447.7 | 48790 | – |
| Bubble | .0018 | .114 | 11.36 | 1250.6 | 143819 | – |
| Selection | .0015 | .073 | 5.84 | 566.3 | 66510 | – |
| Shell | .0018 | .040 | 0.64 | 10.4 | 177 | 2980 |
| Shell/O | .0017 | .035 | 0.57 | 9.8 | 154 | 2680 |
| Quick | .0026 | .037 | 0.41 | 4.9 | 57 | 640 |
| Quick/O | .0010 | .022 | 0.30 | 3.9 | 47 | 560 |
| Merge | .0039 | .057 | 0.72 | 9.1 | 118 | 1490 |
| Merge/O | .0012 | .330 | 0.50 | 6.7 | 95 | 1250 |
| Heap | .0034 | .490 | 0.65 | 8.8 | 129 | 2080 |
| Radix/4 | .0379 | .350 | 3.48 | 35.5 | 379 | 3990 |
| Radix/8 | .0345 | .191 | 1.77 | 17.8 | 189 | 2010 |

79

79

## 内部排序方法分类 *Review*

排序方法
- 简单插入排序
- 冒泡排序
- 简单选择排序
- 希尔排序
- 快速排序
- 归并排序
- 堆排序
- 基数排序

80

80

# *Chapter 8 File Processing and External Sort*

1

## Content

8.1 File Processing
8.2 External Sorting

2

## 8.1 File Processing

8.1.1 Primary vs. Secondary Storage
8.1.2 Disk Drives
8.1.3 Buffers and Buffer pools

3

## 8.1.1 Primary vs. Secondary Storage

**Primary storage: Main memory (RAM主存)**

**Secondary Storage: Peripheral devices(外存)**
- ***Disk*** **drives (硬盘，U盘，光盘)**
- **Tape drives (磁带)**

文件存储位置

**RAM is usually volatile(易失性)， expensive.**
**RAM is about million times faster than disk.**

4

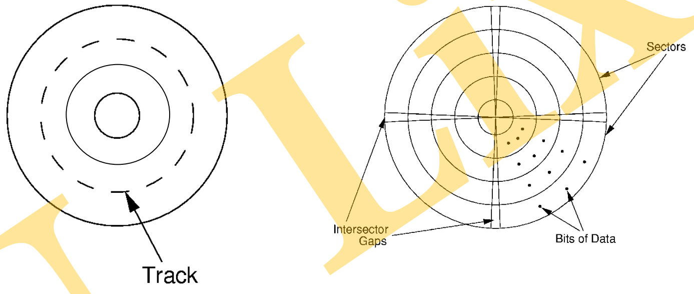
## Golden Rule of File Processing

- **Minimize the number of disk accesses!**
  1. **Arrange information so that you get what you want with few disk accesses.**
  2. **Arrange information to minimize future disk accesses.**

- **Compress information to save processing time by reducing disk accesses.**

5

## 8.1.2 Disk Drives



**Sector(扇形)/Block is the basic unit of I/O.**

6

## Other terms about disk

- **Locality of Reference(引用的局部性):**

  **When data is read from disk, next request is likely to come from near the same place in the file. 两次存取的位置一般比较接近**

- **Block(块）:**

  **Another name of Sector, Smallest unit of record allocation(分配) in windows OS**

- **Cluster(簇):**

  **Smallest unit of file allocation(分配), usually contain several Contiguous (连续的) sectors. Smallest unit of file allocation(分配) in windows OS**

7

## Disk Access Cost(磁盘存取花费)

**=Track seek time + Rotational Delay + Transfer time**

- **Track seek time(寻道时间): Time for I/O head to reach desired track. Largely determined by distance between I/O head and desired track.**
  - **Track-to-track time: Minimum time to move from one track to an adjacent track.**
  - **Average Seek time: Average time to reach a track for random access**
- **Rotational Delay or Latency(旋转延迟): Time for data to rotate under I/O head.**
  - **One half of a rotation time on average.**
  - **At 7200 rpm, this is 8.3/2 = 4.2ms. At 5400 rpm, 11.1/2=5.6ms.**
- **Transfer/read time: Time for data to move under the I/O head.**
  - **Number of sectors scan /Number of sectors per track * rotation time on average**

8

8

## Hard Disk Spec Example

16.8 GB disk on 10 platters = 1.68GB/platter
13,085 tracks/platter
256 sectors/track
512 bytes/sector
Track-to-track seek time: 2.2 ms
Average seek time: 9.5ms
4KB/cluster, 8 sectors/cluster , 32 clusters/track.
5400 RPM

Read a 1MB file divided into 2048 records of 512 bytes each.

1. all records are on 8 sequential tracks 158ms
2. all clusters(256) are randomly spread across the disk. 3942ms

9

## Disk Access Cost Example1

- Read a 1MB file divided into 2048 records of 512 bytes each.
  - ✓ Assume all records are on 8 sequential tracks.
  - ✓ First track: 9.5 + 11.1/2 + 11.1 = 26.2 ms
  - ✓ Remaining 7 tracks: 2.2 + 11.1/2 + 11.1 = 18.9 ms.
  - ✓ Total: 26.2 + 7 * 18.9 = 158.5ms

10

## Disk Access Cost Example2

- Read a 1MB file divided into 2048 records of 512 bytes each.
  - ✓ Assume all clusters are randomly spread across the disk.
  - ✓ 需存放 256 clusters（随机，不连续）.
  - ✓ locate 1 Cluster time: 9.5+ 11.1/2 ms
  - ✓ 1 Cluster read time: 11.1x 8 /256 = 0.35ms
  - ✓ Total: 256* (9.5 + 11.1/2 +0.35) = 3942.4 ms.

Random access is MUCH worse than the sequential access

11

## How Much time needed to Read?

- Read time for one track:
  9.5 + 11.1/2 + 11.1 = 26.2ms.
- Read time for one sector (512bytes):
  9.5 + 11.1/2 + (1/256) x11.1 = 15.1ms.
- Read time for one byte:
  9.5 + 11.1/2 = 15.05 ms.
- Nearly all disk drives read/write one sector at every I/O access.
  A sector Also referred to as a page.

Sector（block） 是I/O的最小（基本）单位

12

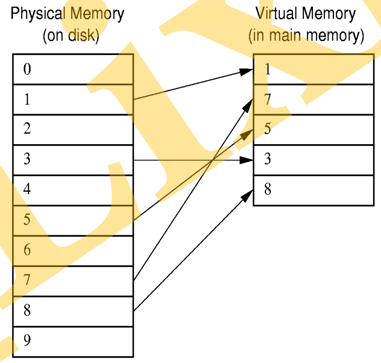
## 8.1.3 Buffers(缓冲区) and Buffer pools

- The information in a sector is stored in a buffer or cache.
- If the next I/O access is to the same buffer, then no need to go to disk.
- There are usually one or more input buffers and one or more output buffers.

13

## Buffer Pools

- A series of buffers used by an application to cache disk data is called a buffer pool.
- Virtual memory uses a buffer pool to imitate greater RAM memory by actually storing information on disk and “swapping” between disk and RAM.



14

## Organizing Buffer Pools

Which buffer should be replaced when new data must be read?

1) First-in, First-out: Replace the first one on the queue.

2) Least Frequently Used (LFU): Count buffer accesses, replace the least used. 替换使用频度最少的

3) Least Recently used (LRU): Keep buffers on a linked list. When buffer is accessed, bring it to front. Replace the one at end. 替换没用时间最久的的

For example: 5 buffers

读取Sector顺序要求：9 0 1 7 6 6 8 1 3 5 1 7 1

15

## OS designer’s View of Files

- Would rather sequential access than Random access
- Using buffer and buffer pool
- Organizing Buffer Pools reasonable

**Minimize the number of disk accesses**

16

## 8.2 External Sorting

8.2.1 External Sorting
8.2.2 Simple Approaches to external sorting
8.2.3 Replacement Selection(置换选择)
8.2.4 Multiway Merging（多路归并）

17

17

## 8.2.1 External Sorting

- Problem: Sorting data sets too large to fit into main memory.
- To sort, portions（部分） of the data must be brought into main memory, processed, and returned to disk.
- An external sort should minimize disk accesses.
  - ✓ Prefer sequential accesses than random accesses

更关注的是I/O操作次数，而不是运算次数

18

## Model of External Computation

- Secondary memory(disk) is divided into equal-sized blocks (512, 1024, etc…)
- A basic I/O operation transfers the contents of one disk block to/from main memory.

与内部排序不同，外部排序关注的是I/O操作次数，而不是运算次数

19

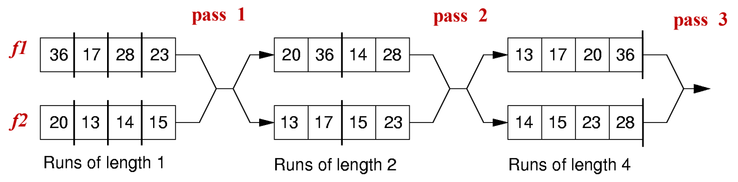
## 8.2.2 Simple Approaches to external sorting

1. Using internal sorting method if your PC support virtual memory
2. Adapting (修改) an internal sorting method to external sorting
   - Quicksort requires random access to the entire set of records.
   - MergeSort is better-- sequnece access a subset of records
     - Process *n* elements in log*n* passes.
     - A group of sorted records is called a *run*.

20

## Simple External two-way Mergesort ----an example

*f*  36 17 28 23 20 13 14 15



Each pass through the files provides larger runs.

In general, external Mergesort consists of two phases:

1. Break the files into initial runs
2. Merge two runs together into a single run.

## Simple two-way External Mergesort—内外存要求

- RAM
  - Two input buffers
  - Two output buffers
- Disk
  - Two input files,
  - Two output files

## Simple two-way External Mergesort ---步骤

one pass

1. Split(分割) the file into two input files.
2. Read in one block from each input file into input buffers.
3. Repeat steps 1) - 2) until two input files are exhausted
   1) Take the odd run from each input buffer, output them to the first output buffer in sorted order .
   2) Take the even run from each input buffer, output them to the second output buffer in sorted order .
4. Alternating between output files and input files.
5. Repeat steps 2-4, except the second input file is empty.
6. the first input file is the sorted file

For step 3， whenever the input buffer is exhausted, read in one block from the appropriate input file, and whenever the output buffer is full, write it out to the appropriate output file

Each pass through the files provides larger and larger runs until only one run remains.

## Problems with Simple two-way MergeSort ?

- 简单 2路归并排序初始runs的长度为多少？ **=*1***
- 简单 2路归并排序初始runs的个数为多少？ ***=记录个数N***
- How many passes for n initial runs? **=$\log_2 n$**
- How many times I/O access are needed in each pass？

**2N/M, N: 待排序文件记录数， M: Block中可存放记录数**

- 简单 2路归并排序总的I/O 次数为多少？ **$\log_2 N$ *2*N/M***
- Is RAM well used? **No， just four buffers**

**初始runs个数越少( 长度越长), MergeSort所需 的pass数越少， 需要的I/O操作越少，速度也越快。**

需要改进简单**2**路外部归并排序

## Better External sorting based on the following two step

1. Breaking a File into some larger initial Runs
   - Read as much of the file into memory as possible.
     ---use RAM better
   - Perform an internal sort.---Create larger initial runs
   - Output the group of sorted records( an initial run) to a single file.
2. Merge n initial runs together to form a single sorted file
   - 2-way merge 趟数：$\text{Log}_2 n$
   - Multi-way merge 趟数: $\log_B n$

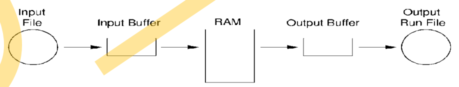
25

## 8.2.3 Replacement Selection /置换选择法

---use the RAM better & Creating larger initial runs

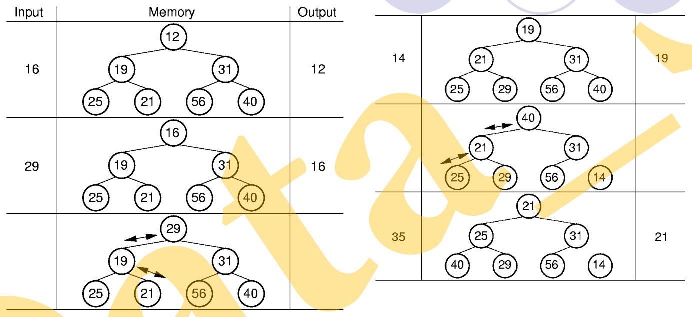
1. Break available memory into an input buffer, an output buffer and an array of size K records for the heap .
2. Fill the array from disk file.
3. Set LAST=K-1, Build a min-heap H.
4. Repeat 1)-3) until LAST= -1 (H is empty)
   1) output the record at the root to Output Buffer;
   2) suppose R be the next record in the input buffer
      - ✓ If R'key is greater than the record'key just output , place R at the root;
      - ✓ Else replace record at the root using the record at LAST position , place R at the LAST position, and let LAST=LAST-1 (add R to a new heap ).
   3) Operate Siftdown at root
5. Repeat step3-4 until the disk file is exhausted

Create one initial run



26

## RS Example



Run的最小长度： K
Run的最大长度： N
Run的平均长度： 2K

K: length of an array for the heap
N: length of the file

27

## RS—内外存使用情况及I/O次数

- RAM—有多少用多少，假设有B+2 个block可用
  - 1 input buffers
  - 1 output buffers
  - B buffer for heap
- Disk
  - n个文件---each initial run 存一个（长度约 2BM）
- I/O times
  - 2N/M.
    N: 待排序文件记录数，
    M: 1个Block中可存放记录数

28

28

## RS+2-way merge—内外存使用情况及I/O次数

- RAM， 假设有B+2 block可用 利用率高，有多少用多少
  - RS： 2 (input&output buffer)； B (heap)
  - 2-way： 2 input buffers ＋1 output buffer 利用率不好
- I/O times RS 2-way merge
  - $2N/M + 2N/M * \text{Log}_2 n$
  - n: 初始run的个数， ≈ $N/(2MB)$
  - N: 待排序文件记录数， M: 1个Block中可存放记录
- Disk---每个run存一个文件
  - 第1趟： n 个输入文件，n / 2个输出文件
  - 第i趟： n / $2^{i-1}$个输入文件，n / $2^i$ 个输出文件

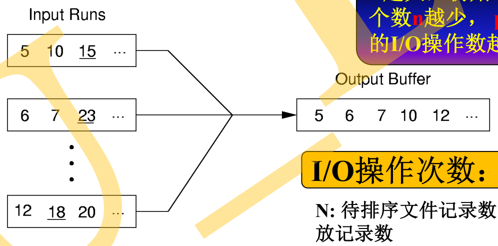
29

29

## 8.2.4 Multiway Merge

- With RS, each initial run may be several blocks long.
- each run is placed in separate file.
- Read the first block from each file into memory and perform a *B*-way merge.
- When a buffer becomes empty, read a block from the appropriate run file.
- Each record is read only once from disk during the one pass merge process.

B越大，初始run的平均长度L越长，个数n越少， pass 次数越少， 需要的I/O操作数越少，速度也越快。



I/O操作次数： $2\ \text{Log}_B n * N/M$

N: 待排序文件记录数， M: 1个Block中可存放记录数

30

## B-way merge—内外存使用情况及I/O次数

- RAM—有多少用多少， 假设有B+1 个block可用
  - B input buffers
  - 1 output buffers
- Disk---每个run存一个文件
  - 第1趟： n个输入文件， n /B 个 输出文件
  - 第i趟： n / $B^{i-1}$个输入文件， n / $B^i$ 个 输出文件
- I/O times
  - $2\ N/M\ * \text{Log}_B n$
  - n: 初始run的个数
  - N: 待排序文件记录数，
  - M: 1个Block中可存放记录数

31

31

## RS+B-way merge—内外存使用情况及I/O次数

- RAM---有多少用多少， 假设有B+2 block可用
  - RS： 2 (input&output buffer)； B (heap)
  - B-way： B input buffers ＋1 output buffer
- I/O times RS B-way merge
  - $2N/M + 2N/M * \text{Log}_B n$
  - n: 初始run的个数 ≈ $N/(2MB)$
  - N: 待排序文件记录数， M: 1个Block中可存放记录数
- Disk---每个run存一个文件
  - 第1趟： n个输入文件， n /B 个 输出文件
  - 第i趟： n / $B^{i-1}$个输入文件， n / $B^i$ 个 输出文件

32

32

Assume *B* blocks were allocated to heap for RS, followed by a B-way merge, one Block can store M records

- How long is the average length of the initial runs 2BM
- How many runs can be processed at one time? B
- If a file have n initial runs, how many pass needed to sort it? $LOG_B n$
- How big a file can be merged in one pass? $2B^2 M$
- How big a file can be merged in two pass? $2B^3 M$

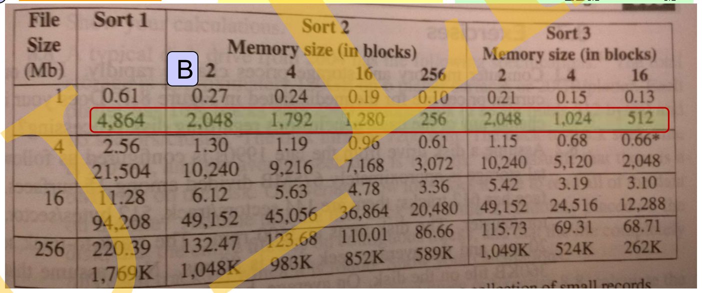
33

## A comparison of three external sort methods

Sort1: Simple two-way merge
Sort2: RS + two-way merge
Sort3: RS + B-way merge

N: 待排序文件记录个数,
M: 1个Block中可存放记录(关键字)个数
B: RS时可分配给堆的block/buffer个数

I/o: $2Log_2 N * \frac{N}{M}$ $(1 + Log_2(\frac{N}{2BM})) * \frac{2N}{M}$ $(1 + Log_B(\frac{N}{2BM})) * \frac{2N}{M}$



| File Size (Mb) | Sort 1 | Sort 2 Memory size (in blocks) | | | | Sort 3 Memory size (in blocks) | | |
|---|---|---|---|---|---|---|---|---|
| | | B 2 | 4 | 16 | 256 | 2 | 4 | 16 |
| 1 | 0.61 | 0.27 | 0.24 | 0.19 | 0.10 | 0.21 | 0.15 | 0.13 |
| | 4,864 | 2,048 | 1,792 | 1,280 | 256 | 2,048 | 1,024 | 512 |
| 4 | 2.56 | 1.30 | 1.19 | 0.96 | 0.61 | 1.15 | 0.68 | 0.66* |
| | 21,504 | 10,240 | 9,216 | 7,168 | 3,072 | 10,240 | 5,120 | 2,048 |
| 16 | 11.28 | 6.12 | 5.63 | 4.78 | 3.36 | 5.42 | 3.19 | 3.10 |
| | 94,208 | 49,152 | 45,056 | 36,864 | 20,480 | 49,152 | 24,516 | 12,288 |
| 256 | 220.39 | 132.47 | 123.68 | 110.01 | 86.66 | 115.73 | 69.31 | 68.71 |
| | 1,769K | 1,048K | 983K | 852K | 589K | 1,049K | 524K | 262K |

34

Chapter8 end

35

35