Sorting

- **Radix sort**
  - sort according to the first lower-end digit (LED), the second LED, the third LED ...

unsorted	79	11	43	71	47	23	67	59		
first LED	*0	*1	*2	*3	*4	*5	*6	*7	*8	*9
		11		43				47		79
		71		23				67		59
sorted by first LED	11	71	43	23	47	67	79	59		
second LED	0*	1*	2*	3*	4*	5*	6*	7*	8*	9*
		11	23		43	59	67	71		
					47			79		
sorted by second LED	11	23	43	47	59	67	71	79		

Sorting

- **Radix sort**
  - sort according to the first lower-end digit (LED), the second LED, the third LED ...

```
// radix sort: r (radix i.e. base of digits), d (max number of digits)
void radixsort(int s[],int n,int r,int d){
        int *sb=new int[n],*cr=new int[r];
        for(int k,i=0,ri=1;i<d;i++,ri*=r){for(k=0;k<r;k++) cr[k]=0;
                for(k=0;k<n;k++) cr[(s[k]/ri)%r]++;
                for(k=1;k<r;k++) cr[k]+=cr[k-1];
                for(k=n-1;k>=0;k--) sb[--cr[(s[k]/ri)%r]]=s[k];
                for(k=0;k<n;k++) s[k]=sb[k];}
        delete[] sb;delete[] cr;}
void radixsort(int s[],int n,int r,int d1,int d2){
        int *sb=new int[n],*cr=new int[r];int k,i,ri=1;
        for(i=0;i<d1;i++) ri*=r;
        for(;i<d2;i++,ri*=r){for(k=0;k<r;k++) cr[k]=0;
                for(k=0;k<n;k++) cr[(s[k]/ri)%r]++;
                for(k=1;k<r;k++) cr[k]+=cr[k-1];
                for(k=n-1;k>=0;k--) sb[--cr[(s[k]/ri)%r]]=s[k];
                for(k=0;k<n;k++) s[k]=sb[k];}
        delete[] sb;delete[] cr;}
// END radix sort
```

Sorting

- **Radix sort**
  - sort according to the first lower-end digit (LED), the second LED, the third LED ...

s[] unsorted	79	11	43	71	47	23	67	59		
first LED	*0	*1	*2	*3	*4	*5	*6	*7	*8	*9
cr[] (count radix)	0	0	0	0	0	0	0	0	0	0

```
for(int k,i=0,ri=1;i<d;i++,ri*=r){for(k=0;k<r;k++) cr[k]=0;
        for(k=0;k<n;k++) cr[(s[k]/ri)%r]++;
        for(k=1;k<r;k++) cr[k]+=cr[k-1];
        for(k=n-1;k>=0;k--) sb[--cr[(s[k]/ri)%r]]=s[k];
        for(k=0;k<n;k++) s[k]=sb[k];}
```

Sorting

- **Radix sort**
  - sort according to the first lower-end digit (LED), the second LED, the third LED ...

s[] unsorted	79	11	43	71	47	23	67	59		
first LED	*0	*1	*2	*3	*4	*5	*6	*7	*8	*9
cr[] (count radix)	0	0	0	0	0	0	0	0	0	1

```
for(int k,i=0,ri=1;i<d;i++,ri*=r){for(k=0;k<r;k++) cr[k]=0;
    for(k=0;k<n;k++) cr[(s[k]/ri)%r]++;
    for(k=1;k<r;k++) cr[k]+=cr[k-1];
    for(k=n-1;k>=0;k--) sb[--cr[(s[k]/ri)%r]]=s[k];
    for(k=0;k<n;k++) s[k]=sb[k];}
```

Sorting

- **Radix sort**
  - sort according to the first lower-end digit (LED), the second LED, the third LED ...

s[] unsorted	79	11	43	71	47	23	67	59		
first LED	*0	*1	*2	*3	*4	*5	*6	*7	*8	*9
cr[] (count radix)	0	1	0	0	0	0	0	0	0	1

```
for(int k,i=0,ri=1;i<d;i++,ri*=r){for(k=0;k<r;k++) cr[k]=0;
        for(k=0;k<n;k++) cr[(s[k]/ri)%r]++;
        for(k=1;k<r;k++) cr[k]+=cr[k-1];
        for(k=n-1;k>=0;k--) sb[--cr[(s[k]/ri)%r]]=s[k];
        for(k=0;k<n;k++) s[k]=sb[k];}
```

Sorting

- **Radix sort**
  - sort according to the first lower-end digit (LED), the second LED, the third LED ...

s[] unsorted	79	11	43	71	47	23	67	59		
first LED	*0	*1	*2	*3	*4	*5	*6	*7	*8	*9
cr[] (count radix)	0	1	0	1	0	0	0	0	0	1

```
for(int k,i=0,ri=1;i<d;i++,ri*=r){for(k=0;k<r;k++) cr[k]=0;
    for(k=0;k<n;k++) cr[(s[k]/ri)%r]++;
    for(k=1;k<r;k++) cr[k]+=cr[k-1];
    for(k=n-1;k>=0;k--) sb[--cr[(s[k]/ri)%r]]=s[k];
    for(k=0;k<n;k++) s[k]=sb[k];}
```

Sorting

- **Radix sort**
  - sort according to the first lower-end digit (LED), the second LED, the third LED ...

s[] unsorted	79	11	43	71	47	23	67	59		
first LED	*0	*1	*2	*3	*4	*5	*6	*7	*8	*9
cr[] (count radix)	0	2	0	1	0	0	0	0	0	1

```
for(int k,i=0,ri=1;i<d;i++,ri*=r){for(k=0;k<r;k++) cr[k]=0;
    for(k=0;k<n;k++) cr[(s[k]/ri)%r]++;
    for(k=1;k<r;k++) cr[k]+=cr[k-1];
    for(k=n-1;k>=0;k--) sb[--cr[(s[k]/ri)%r]]=s[k];
    for(k=0;k<n;k++) s[k]=sb[k];}
```

Sorting

- **Radix sort**
  - sort according to the first lower-end digit (LED), the second LED, the third LED ...

s[] unsorted	79	11	43	71	47	23	67	59		
first LED	*0	*1	*2	*3	*4	*5	*6	*7	*8	*9
cr[] (count radix)	0	2	0	1	0	0	0	1	0	1

```
for(int k,i=0,ri=1;i<d;i++,ri*=r){for(k=0;k<r;k++) cr[k]=0;
        for(k=0;k<n;k++) cr[(s[k]/ri)%r]++;
        for(k=1;k<r;k++) cr[k]+=cr[k-1];
        for(k=n-1;k>=0;k--) sb[--cr[(s[k]/ri)%r]]=s[k];
        for(k=0;k<n;k++) s[k]=sb[k];}
```

Sorting

- **Radix sort**
  - sort according to the first lower-end digit (LED), the second LED, the third LED ...

s[] unsorted	79	11	43	71	47	23	67	59		
first LED	*0	*1	*2	*3	*4	*5	*6	*7	*8	*9
cr[] (count radix)	0	2	0	2	0	0	0	1	0	1

```
for(int k,i=0,ri=1;i<d;i++,ri*=r){for(k=0;k<r;k++) cr[k]=0;
    for(k=0;k<n;k++) cr[(s[k]/ri)%r]++;
    for(k=1;k<r;k++) cr[k]+=cr[k-1];
    for(k=n-1;k>=0;k--) sb[--cr[(s[k]/ri)%r]]=s[k];
    for(k=0;k<n;k++) s[k]=sb[k];}
```

Sorting

- **Radix sort**
  - sort according to the first lower-end digit (LED), the second LED, the third LED ...

s[] unsorted	79	11	43	71	47	23	67	59		
first LED	*0	*1	*2	*3	*4	*5	*6	*7	*8	*9
cr[] (count radix)	0	2	0	2	0	0	0	2	0	1

```
for(int k,i=0,ri=1;i<d;i++,ri*=r){for(k=0;k<r;k++) cr[k]=0;
        for(k=0;k<n;k++) cr[(s[k]/ri)%r]++;
        for(k=1;k<r;k++) cr[k]+=cr[k-1];
        for(k=n-1;k>=0;k--) sb[--cr[(s[k]/ri)%r]]=s[k];
        for(k=0;k<n;k++) s[k]=sb[k];}
```

Sorting

- **Radix sort**
  - sort according to the first lower-end digit (LED), the second LED, the third LED ...

s[] unsorted	79	11	43	71	47	23	67	59		
first LED	*0	*1	*2	*3	*4	*5	*6	*7	*8	*9
cr[] (count radix)	0	2	0	2	0	0	0	2	0	2

```
for(int k,i=0,ri=1;i<d;i++,ri*=r){for(k=0;k<r;k++) cr[k]=0;
        for(k=0;k<n;k++) cr[(s[k]/ri)%r]++;
        for(k=1;k<r;k++) cr[k]+=cr[k-1];
        for(k=n-1;k>=0;k--) sb[--cr[(s[k]/ri)%r]]=s[k];
        for(k=0;k<n;k++) s[k]=sb[k];}
```

Sorting

- **Radix sort**
  - sort according to the first lower-end digit (LED), the second LED, the third LED ...

s[] unsorted	79	11	43	71	47	23	67	59		
first LED	*0	*1	*2	*3	*4	*5	*6	*7	*8	*9
cr[] (count radix)	0	2	0	2	0	0	0	2	0	2
cr[] (count radix)	0	2	2	4	4	4	4	6	6	8

```
for(int k,i=0,ri=1;i<d;i++,ri*=r){for(k=0;k<r;k++) cr[k]=0;
        for(k=0;k<n;k++) cr[(s[k]/ri)%r]++;
        for(k=1;k<r;k++) cr[k]+=cr[k-1];
        for(k=n-1;k>=0;k--) sb[--cr[(s[k]/ri)%r]]=s[k];
        for(k=0;k<n;k++) s[k]=sb[k];}
```

Sorting

- **Radix sort**
  - sort according to the first lower-end digit (LED), the second LED, the third LED ...

s[] unsorted	79	11	43	71	47	23	67	59		
first LED	*0	*1	*2	*3	*4	*5	*6	*7	*8	*9
cr[] (count radix)	0	2	0	2	0	0	0	2	0	2
cr[] (count radix)	0	2	2	4	4	4	4	6	6	8
sb[] sorted by 1st LED										

```
for(int k,i=0,ri=1;i<d;i++,ri*=r){for(k=0;k<r;k++) cr[k]=0;
        for(k=0;k<n;k++) cr[(s[k]/ri)%r]++;
        for(k=1;k<r;k++) cr[k]+=cr[k-1];
        for(k=n-1;k>=0;k--) sb[--cr[(s[k]/ri)%r]]=s[k];
        for(k=0;k<n;k++) s[k]=sb[k];}
```

Sorting

- **Radix sort**
  - sort according to the first lower-end digit (LED), the second LED, the third LED ...

s[] unsorted	79	11	43	71	47	23	67	59		
first LED	*0	*1	*2	*3	*4	*5	*6	*7	*8	*9
cr[] (count radix)	0	2	0	2	0	0	0	2	0	2
cr[] (count radix)	0	2	2	4	4	4	4	6	6	7
sb[] sorted by 1st LED								59		

```
for(int k,i=0,ri=1;i<d;i++,ri*=r){for(k=0;k<r;k++) cr[k]=0;
        for(k=0;k<n;k++) cr[(s[k]/ri)%r]++;
        for(k=1;k<r;k++) cr[k]+=cr[k-1];
        for(k=n-1;k>=0;k--) sb[--cr[(s[k]/ri)%r]]=s[k];
        for(k=0;k<n;k++) s[k]=sb[k];}
```

Sorting

- **Radix sort**
  - sort according to the first lower-end digit (LED), the second LED, the third LED ...

s[] unsorted	79	11	43	71	47	23	67	59		
first LED	*0	*1	*2	*3	*4	*5	*6	*7	*8	*9
cr[] (count radix)	0	2	0	2	0	0	0	2	0	2
cr[] (count radix)	0	2	2	4	4	4	4	5	6	7
sb[] sorted by 1st LED						67		59		

```
for(int k,i=0,ri=1;i<d;i++,ri*=r){for(k=0;k<r;k++) cr[k]=0;
        for(k=0;k<n;k++) cr[(s[k]/ri)%r]++;
        for(k=1;k<r;k++) cr[k]+=cr[k-1];
        for(k=n-1;k>=0;k--) sb[--cr[(s[k]/ri)%r]]=s[k];
        for(k=0;k<n;k++) s[k]=sb[k];}
```

Sorting

- **Radix sort**
  - sort according to the first lower-end digit (LED), the second LED, the third LED ...

s[] unsorted	79	11	43	71	47	23	67	59		
first LED	*0	*1	*2	*3	*4	*5	*6	*7	*8	*9
cr[] (count radix)	0	2	0	2	0	0	0	2	0	2
cr[] (count radix)	0	2	2	3	4	4	4	5	6	7
sb[] sorted by 1st LED				23		67		59		

```
for(int k,i=0,ri=1;i<d;i++,ri*=r){for(k=0;k<r;k++) cr[k]=0;
        for(k=0;k<n;k++) cr[(s[k]/ri)%r]++;
        for(k=1;k<r;k++) cr[k]+=cr[k-1];
        for(k=n-1;k>=0;k--) sb[--cr[(s[k]/ri)%r]]=s[k];
        for(k=0;k<n;k++) s[k]=sb[k];}
```

Sorting

- **Radix sort**
  - sort according to the first lower-end digit (LED), the second LED, the third LED ...

s[] unsorted	79	11	43	71	47	23	67	59		
first LED	*0	*1	*2	*3	*4	*5	*6	*7	*8	*9
cr[] (count radix)	0	2	0	2	0	0	0	2	0	2
cr[] (count radix)	0	2	2	3	4	4	4	4	6	7
sb[] sorted by 1st LED				23	47	67		59		

```
for(int k,i=0,ri=1;i<d;i++,ri*=r){for(k=0;k<r;k++) cr[k]=0;
        for(k=0;k<n;k++) cr[(s[k]/ri)%r]++;
        for(k=1;k<r;k++) cr[k]+=cr[k-1];
        for(k=n-1;k>=0;k--) sb[--cr[(s[k]/ri)%r]]=s[k];
        for(k=0;k<n;k++) s[k]=sb[k];}
```

Sorting

- **Radix sort**
  - sort according to the first lower-end digit (LED), the second LED, the third LED ...

s[] unsorted	79	11	43	71	47	23	67	59		
first LED	*0	*1	*2	*3	*4	*5	*6	*7	*8	*9
cr[] (count radix)	0	2	0	2	0	0	0	2	0	2
cr[] (count radix)	0	1	2	3	4	4	4	4	6	7
sb[] sorted by 1st LED		71		23	47	67		59		

```
for(int k,i=0,ri=1;i<d;i++,ri*=r){for(k=0;k<r;k++) cr[k]=0;
        for(k=0;k<n;k++) cr[(s[k]/ri)%r]++;
        for(k=1;k<r;k++) cr[k]+=cr[k-1];
        for(k=n-1;k>=0;k--) sb[--cr[(s[k]/ri)%r]]=s[k];
        for(k=0;k<n;k++) s[k]=sb[k];}
```

Sorting

- **Radix sort**
  - sort according to the first lower-end digit (LED), the second LED, the third LED ...

s[] unsorted	79	11	43	71	47	23	67	59		
first LED	*0	*1	*2	*3	*4	*5	*6	*7	*8	*9
cr[] (count radix)	0	2	0	2	0	0	0	2	0	2
cr[] (count radix)	0	1	2	2	4	4	4	4	6	7
sb[] sorted by 1st LED		71	43	23	47	67		59		

```
for(int k,i=0,ri=1;i<d;i++,ri*=r){for(k=0;k<r;k++) cr[k]=0;
        for(k=0;k<n;k++) cr[(s[k]/ri)%r]++;
        for(k=1;k<r;k++) cr[k]+=cr[k-1];
        for(k=n-1;k>=0;k--) sb[--cr[(s[k]/ri)%r]]=s[k];
        for(k=0;k<n;k++) s[k]=sb[k];}
```

Sorting

- **Radix sort**
  - sort according to the first lower-end digit (LED), the second LED, the third LED ...

s[] unsorted	79	11	43	71	47	23	67	59		
first LED	*0	*1	*2	*3	*4	*5	*6	*7	*8	*9
cr[] (count radix)	0	2	0	2	0	0	0	2	0	2
cr[] (count radix)	0	0	2	2	4	4	4	4	6	7
sb[] sorted by 1st LED	11	71	43	23	47	67		59		

```
for(int k,i=0,ri=1;i<d;i++,ri*=r){for(k=0;k<r;k++) cr[k]=0;
        for(k=0;k<n;k++) cr[(s[k]/ri)%r]++;
        for(k=1;k<r;k++) cr[k]+=cr[k-1];
        for(k=n-1;k>=0;k--) sb[--cr[(s[k]/ri)%r]]=s[k];
        for(k=0;k<n;k++) s[k]=sb[k];}
```

Sorting

- **Radix sort**
  - sort according to the first lower-end digit (LED), the second LED, the third LED ...

s[] unsorted	79	11	43	71	47	23	67	59		
first LED	*0	*1	*2	*3	*4	*5	*6	*7	*8	*9
cr[] (count radix)	0	2	0	2	0	0	0	2	0	2
cr[] (count radix)	0	0	2	2	4	4	4	4	6	6
sb[] sorted by 1st LED	11	71	43	23	47	67	79	59		

```
for(int k,i=0,ri=1;i<d;i++,ri*=r){for(k=0;k<r;k++) cr[k]=0;
        for(k=0;k<n;k++) cr[(s[k]/ri)%r]++;
        for(k=1;k<r;k++) cr[k]+=cr[k-1];
        for(k=n-1;k>=0;k--) sb[--cr[(s[k]/ri)%r]]=s[k];
        for(k=0;k<n;k++) s[k]=sb[k];}
```

Sorting

- **Radix sort**
  - sort according to the first lower-end digit (LED), the second LED, the third LED ...

s[] unsorted	79	11	43	71	47	23	67	59		
first LED	*0	*1	*2	*3	*4	*5	*6	*7	*8	*9
cr[] (count radix)	0	2	0	2	0	0	0	2	0	2
cr[] (count radix)	0	0	2	2	4	4	4	4	6	6
sb[] sorted by 1st LED	11	71	43	23	47	67	79	59		
s[] sorted by 1st LED	11	71	43	23	47	67	79	59		

```
for(int k,i=0,ri=1;i<d;i++,ri*=r){for(k=0;k<r;k++) cr[k]=0;
        for(k=0;k<n;k++) cr[(s[k]/ri)%r]++;
        for(k=1;k<r;k++) cr[k]+=cr[k-1];
        for(k=n-1;k>=0;k--) sb[--cr[(s[k]/ri)%r]]=s[k];
        for(k=0;k<n;k++) s[k]=sb[k];}
```

Sorting

- **Radix sort**
  - sort according to the first lower-end digit (LED), the second LED, the third LED ...

unsorted	79	11	43	71	47	23	67	59		
first LED	*0	*1	*2	*3	*4	*5	*6	*7	*8	*9
		11		43				47		79
		71		23				67		59
sorted by first LED	11	71	43	23	47	67	79	59		

```
for(int k,i=0,ri=1;i<d;i++,ri*=r){for(k=0;k<r;k++) cr[k]=0;
        for(k=0;k<n;k++) cr[(s[k]/ri)%r]++;
        for(k=1;k<r;k++) cr[k]+=cr[k-1];
        for(k=n-1;k>=0;k--) sb[--cr[(s[k]/ri)%r]]=s[k];
        for(k=0;k<n;k++) s[k]=sb[k];}
```

Sorting

- **Radix sort**
  - sort according to the first lower-end digit (LED), the second LED, the third LED ...

unsorted	79	11	43	71	47	23	67	59		
first LED	*0	*1	*2	*3	*4	*5	*6	*7	*8	*9
		11		43				47		79
		71		23				67		59
sorted by first LED	11	71	43	23	47	67	79	59		
second LED	0*	1*	2*	3*	4*	5*	6*	7*	8*	9*
		11	23		43	59	67	71		
					47			79		
sorted by second LED	11	23	43	47	59	67	71	79		

Sorting

- **Radix sort**
  - sort according to the first lower-end digit (LED), the second LED, the third LED ...

```
DEMO : radix sort =>
unsorted sequence: 42 92 96 79 93 4 85 66 68 76 74 63 39 17 71 3
radix sort: 3 4 17 39 42 63 66 68 71 74 76 79 85 92 93 96
radix sort (radix 10, 1st l.e.d): 71 42 92 93 63 3 4 74 85 96 66 76 17 68 79 39
radix sort (radix 10, 2nd l.e.d): 3 4 17 39 42 63 66 68 71 74 76 79 85 92 93 96
radix sort (radix 5, 1st l.e.d): 85 96 66 76 71 42 92 17 93 68 63 3 79 4 74 39
radix sort (radix 5, 2nd l.e.d): 76 3 79 4 85 63 39 66 42 92 17 93 68 96 71 74
radix sort (radix 5, 3rd l.e.d): 3 4 17 39 42 63 66 68 71 74 76 79 85 92 93 96
```

```
for(int k,i=0,ri=1;i<d;i++,ri*=r){for(k=0;k<r;k++) cr[k]=0;
        for(k=0;k<n;k++) cr[(s[k]/ri)%r]++;
        for(k=1;k<r;k++) cr[k]+=cr[k-1];
        for(k=n-1;k>=0;k--) sb[--cr[(s[k]/ri)%r]]=s[k];
        for(k=0;k<n;k++) s[k]=sb[k];}
```

```
cout<<"DEMO : radix sort =>\n";cp<int>(iA,iTab,ni);
cout<<"unsorted sequence: ";show<int>(iA,ni);
cout<<"radix sort: ";radixsort(iA,ni,10,2);show<int>(iA,ni);
cp<int>(iA,iTab,ni);
cout<<"radix sort (radix 10, 1st l.e.d): ";radixsort(iA,ni,10,0,1);show<int>(iA,ni);
cout<<"radix sort (radix 10, 2nd l.e.d): ";radixsort(iA,ni,10,1,2);show<int>(iA,ni);
cp<int>(iA,iTab,ni);
cout<<"radix sort (radix 5, 1st l.e.d): ";radixsort(iA,ni,5,0,1);show<int>(iA,ni);
cout<<"radix sort (radix 5, 2nd l.e.d): ";radixsort(iA,ni,5,1,2);show<int>(iA,ni);
cout<<"radix sort (radix 5, 3rd l.e.d): ";radixsort(iA,ni,5,2,3);show<int>(iA,ni);
```

Sorting

- **Radix sort**
  - sort according to the first lower-end digit (LED), the second LED, the third LED ...
  - complexity: O((n+r)d)
    - if n values are densely distributed, radixsort tends to be very efficient
    - if n values are sparsely distributed, radixsort tends to have poor performance

```
// radix sort: r (radix i.e. base of digits), d (max number of digits)
void radixsort(int s[],int n,int r,int d){
        int *sb=new int[n],*cr=new int[r];
        for(int k,i=0,ri=1;i<d;i++,ri*=r){for(k=0;k<r;k++) cr[k]=0;
                for(k=0;k<n;k++) cr[(s[k]/ri)%r]++;
                for(k=1;k<r;k++) cr[k]+=cr[k-1];
                for(k=n-1;k>=0;k--) sb[--cr[(s[k]/ri)%r]]=s[k];
                for(k=0;k<n;k++) s[k]=sb[k];}
        delete[] sb;delete[] cr;}
void radixsort(int s[],int n,int r,int d1,int d2){
        int *sb=new int[n],*cr=new int[r];int k,i,ri=1;
        for(i=0;i<d1;i++) ri*=r;
        for(;i<d2;i++,ri*=r){for(k=0;k<r;k++) cr[k]=0;
                for(k=0;k<n;k++) cr[(s[k]/ri)%r]++;
                for(k=1;k<r;k++) cr[k]+=cr[k-1];
                for(k=n-1;k>=0;k--) sb[--cr[(s[k]/ri)%r]]=s[k];
                for(k=0;k<n;k++) s[k]=sb[k];}
        delete[] sb;delete[] cr;}
// END radix sort
```

THANK YOU

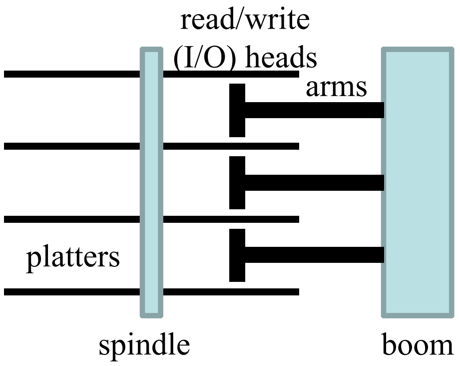
上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

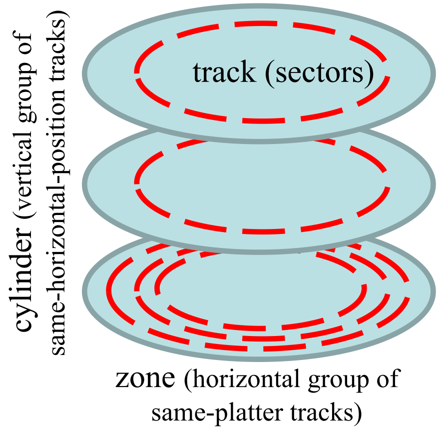
File Processing & External Sorting

- **Primary (main) memory vs. secondary (peripheral) storage**
  - primary or main memory
    - e.g. **random access memory** (RAM), registers, cache, video memories
  - secondary or peripheral storage
    - e.g. **hard disk drives**, solid state drives, removable USB drives, CDs, DVDs
  - memory & storage access speed
    - RAM access speed is $10^5 \sim 10^6$ faster than disk drive access speed
- **Disk-based applications - minimize the number of disk accesses**
  - organize file structures
  - save information previously retrieved

File Processing & External Sorting

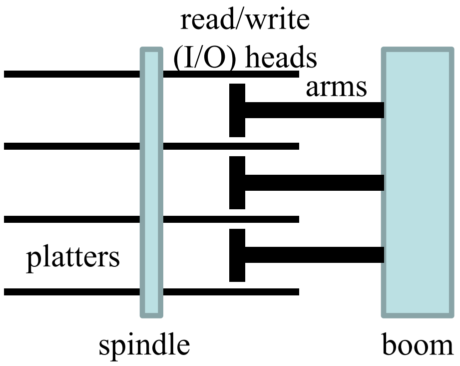
- **Disk drives**
  - logical file vs. physical file
  - disk drive architecture
    - spindle & platter
    - read/write (I/O) head
    - boom & arm
    - track & cylinder & zone
    - sector & inter-sector gap
      - each track is divided into sectors with inter-sector gaps
      - tracks in the same zone have inter-sector gaps at same angular positions

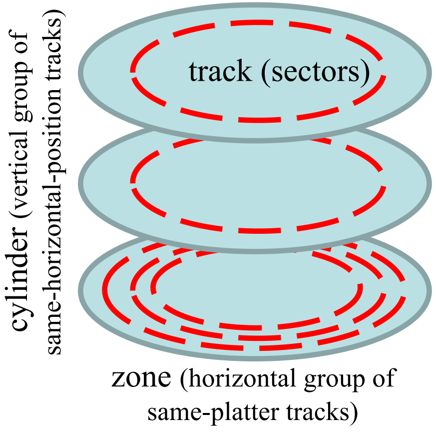




File Processing & External Sorting

- **Disk drives**
  - logical file vs. physical file
  - disk drive architecture
    - spindle & platter
    - read/write (I/O) head
    - boom & arm
    - track & cylinder & zone
    - sector & inter-sector gap

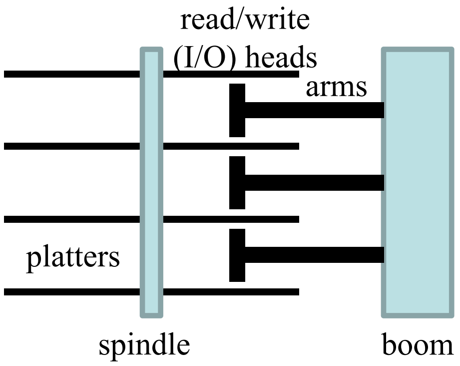


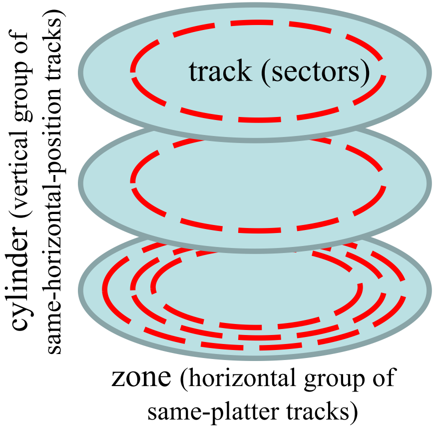


  - disk drive acess
    - *seek* : to position the I/O head over the track containing target data
    - *preparatory rotation* : to rotate the sector containing target data to come under the I/O head
      - e.g. disk spin rates are 1200~15000 RPM (rotations per minute); typical spin rates are 7200 RPM
      - *rotational delay/latency* : time waited for the desired sector to come under the I/O head
    - *data transfer* : to actually read or write target data

File Processing & External Sorting

- **Disk drives**
  - logical file vs. physical file
  - disk drive architecture
    - spindle & platter
    - read/write (I/O) head
    - boom & arm
    - track & cylinder & zone
    - sector & inter-sector gap

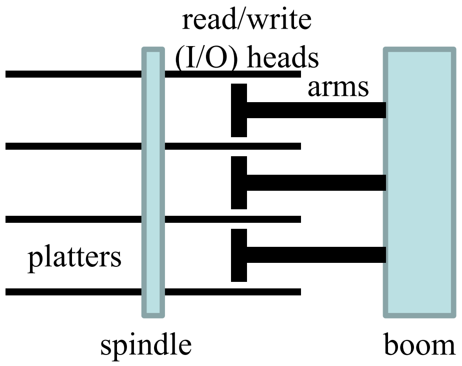


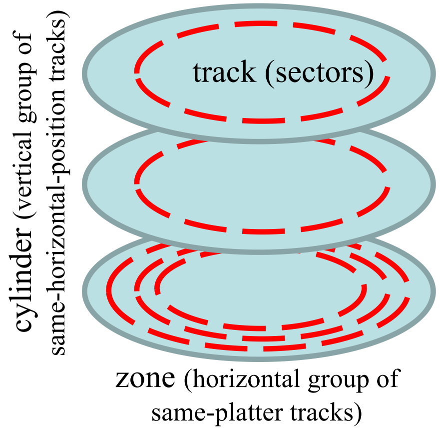


  - disk drive acess - *seek & preparatory rotation & data transfer*
  - aggregate a file's sectors on as few tracks as possible
    - seek time is slow (typically the most expensive part of I/O operations)
    - *locality of reference* assumption : if one sector is read, the next sector tends to be read

File Processing & External Sorting

- **Disk drives**
  - logical file vs. physical file
  - disk drive architecture
    - spindle & platter
    - read/write (I/O) head
    - boom & arm
    - track & cylinder & zone
    - sector & inter-sector gap

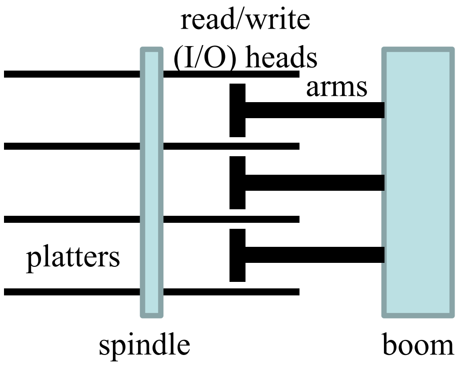


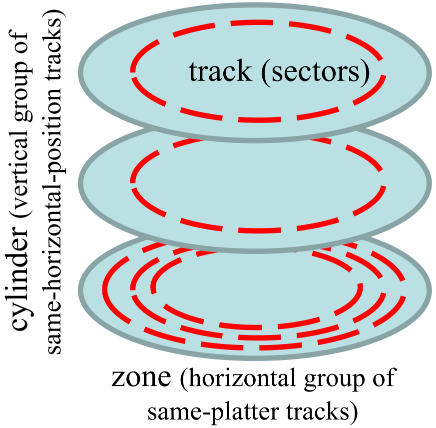


  - disk drive acess - *seek & preparatory rotation & data transfer*
  - aggregate a file’s sectors on as few tracks as possible
  - *smallest file allocation unit*
    - Unix just uses a *sector* (a.k.a. *block*) as the smallest file allocation unit
    - Windows use a *cluster* (group of contiguous sectors) as the smallest file allocation unit

File Processing & External Sorting

- **Disk drives**
  - logical file vs. physical file
  - disk drive architecture
    - spindle & platter
    - read/write (I/O) head
    - boom & arm
    - track & cylinder & zone
    - sector & inter-sector gap

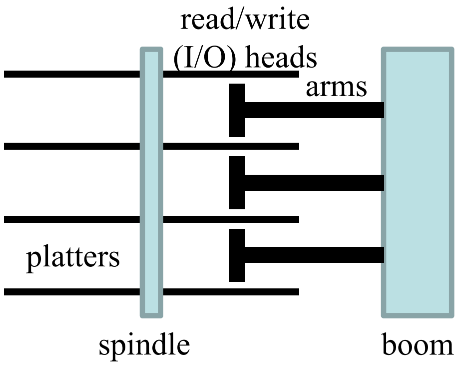


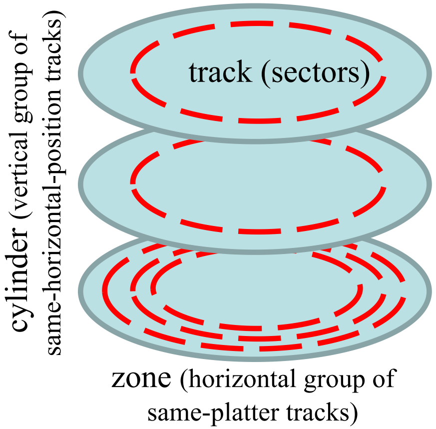


  - disk drive acess - *seek & preparatory rotation & data transfer*
  - aggregate a file's sectors on as few tracks as possible
  - *smallest file allocation unit* - consistently called *cluster* here, be it a sector or an actual one
    - Unix just uses a *sector* (a.k.a. *block*) as the smallest file allocation unit
    - Windows use a *cluster* (group of contiguous sectors) as the smallest file allocation unit

File Processing & External Sorting

- **Disk drives**
  - logical file vs. physical file
  - disk drive architecture
    - spindle & platter
    - read/write (I/O) head
    - boom & arm
    - track & cylinder & zone
    - sector & inter-sector gap

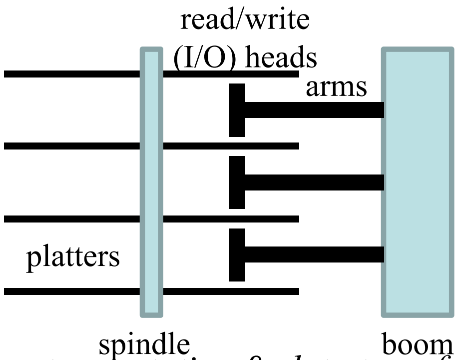


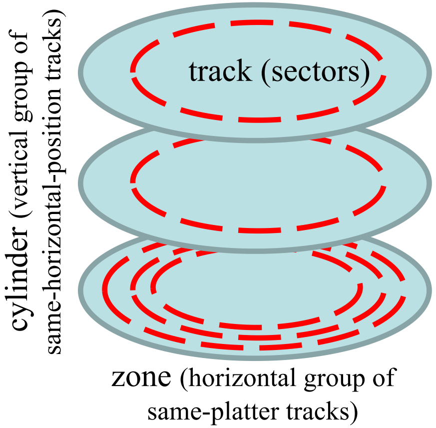


  - disk drive acess - *seek & preparatory rotation & data transfer*
  - aggregate a file's sectors on as few tracks as possible
  - *smallest file allocation unit* - consistently called *cluster* here, be it a sector or an actual one
  - file fragmentation
    - *extent* : a group of contiguous clusters from a file (ideally a file consists of one extent)
    - a file may consist of multiple (even far-away) extents, which is called file fragmentation

File Processing & External Sorting

- **Disk drives**
  - logical file vs. physical file
  - disk drive architecture
    - read/write (I/O) head
    - track & cylinder & zone
    - sector & inter-sector gap

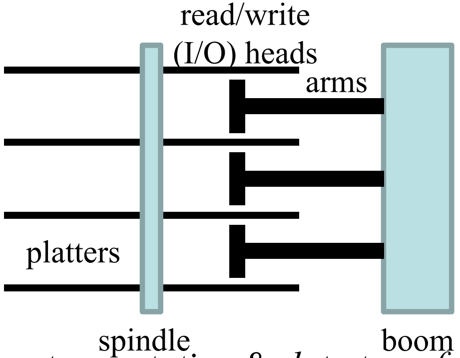


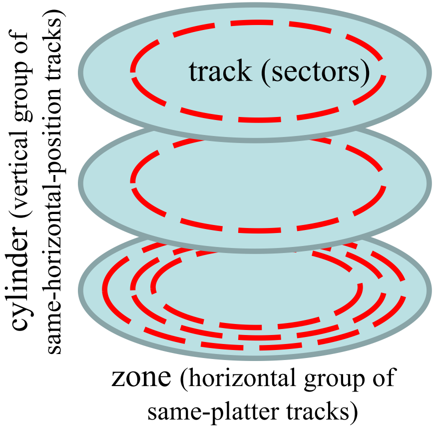


  - disk drive acess - *seek & preparatory rotation & data transfer*
  - aggregate a file's sectors on as few tracks as possible
  - *smallest file allocation unit* - consistently called *cluster* here, be it a sector or an actual one
  - file fragmentation
  - *file allocation table*
    - store information about which sectors belong to which file

File Processing & External Sorting

- **Disk drives**
  - logical file vs. physical file
  - disk drive architecture
    - read/write (I/O) head
    - track & cylinder & zone
    - sector & inter-sector gap

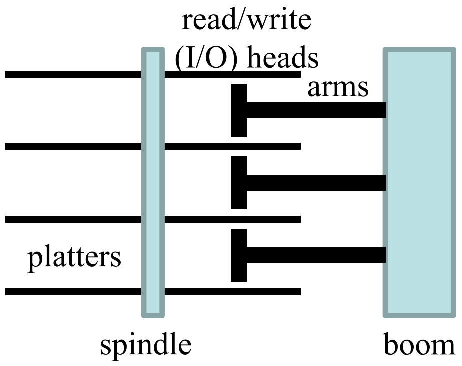


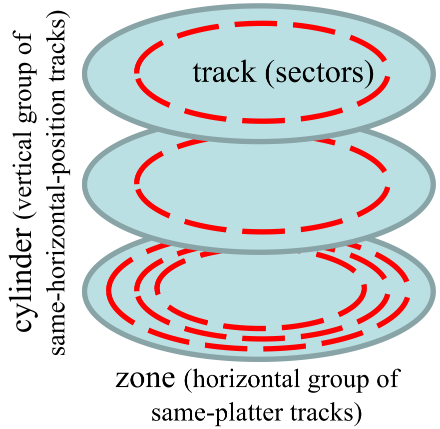


  - disk drive acess - *seek* & *preparatory rotation* & *data transfer*
  - aggregate a file’s sectors on as few tracks as possible
  - *smallest file allocation unit* - consistently called *cluster* here, be it a sector or an actual one
  - file fragmentation
  - *file allocation table*
  - *sector header*
    - store sector address, error detection code for sector contents, sector condition etc.

File Processing & External Sorting

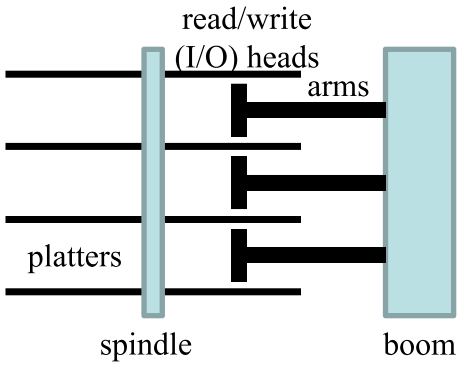
- **Disk drives**
  - disk drive architecture
  - disk drive acess
    - seek
    - preparatory rotation
    - data transfer
  - disk drive access time (e.g.)
    - a disk drive has 20G spread among 10 platters (so 2G/platter)
    - each platter contains 16384 tracks (2G i.e. $2x1024^2$K/16384 tracks=128K/track)
    - each track contains 256 sectors (128K/256 sectors=0.5K/sector=512 bytes/sector)
      - sector sizes are typically a power of two, in the range 512 (2^9) to 16K (2^{14}) bytes
    - a cluster consists of 8 sectors (8x0.5K/sector=4K/cluster); a track contains 32 clusters
    - track-to-track seek time is 2.0ms; average random access seek time is 8.0ms
    - disk spin/rotation rate is 7200 RPM (60/7200=8.33ms/rotation)

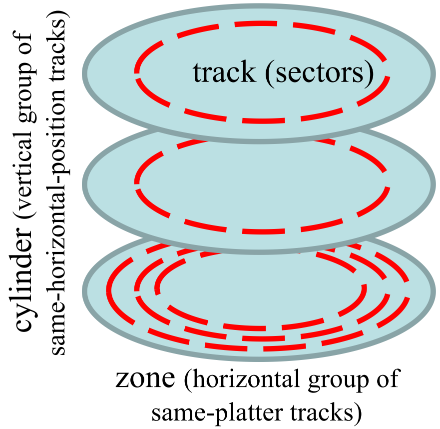




File Processing & External Sorting

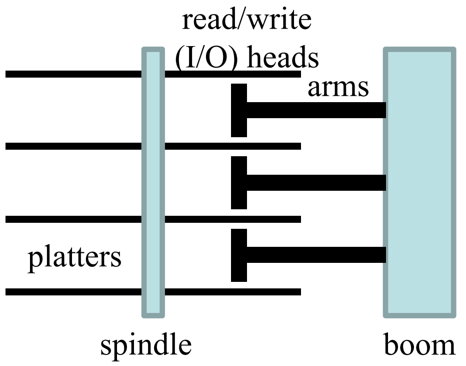
- **Disk drives**
  - disk drive architecture
  - disk drive acess
    - seek
    - preparatory rotation
    - data transfer
  - disk drive access time (e.g.)
    - disk drive (20G) = 10 platters ; platter = 16384 tracks
    - track (128K) = 32 clusters = 256 sectors ; sector = 0.5K ; cluster = 4K
    - track-to-track seek = 2.0ms ; random seek = 8.0ms ; rotation = 8.33ms (7200 RPM)
    - **how long will it take to read a file of 1M** (1024K = 8 tracks = 256 clusters)?

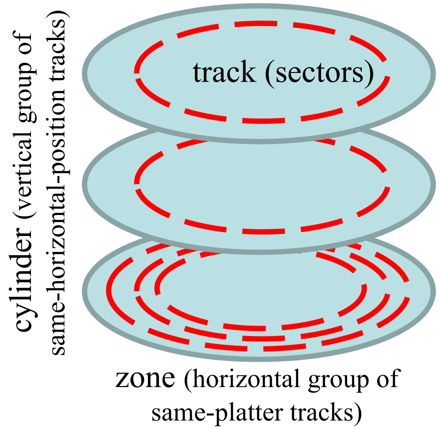




File Processing & External Sorting

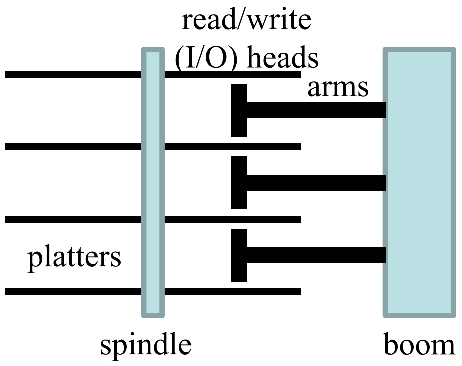
- **Disk drives**
  - disk drive architecture
  - disk drive acess
    - seek
    - preparatory rotation
    - data transfer
  - disk drive access time (e.g.)
    - disk drive (20G) = 10 platters ; platter = 16384 tracks
    - track (128K) = 32 clusters = 256 sectors ; sector = 0.5K ; cluster = 4K
    - track-to-track seek = 2.0ms ; random seek = 8.0ms ; rotation = 8.33ms (7200 RPM)
    - **how long will it take to read a file of 1M** (1024K = 8 tracks = 256 clusters)?
    - if the file fills 8 adjacent tracks
      - [8.0+8.33x(0.5+1)]+7x[2.0+8.33x(0.5+1)]=20.5+7x14.5=122ms
    - if the file fills 256 clusters spread randomly across the disk

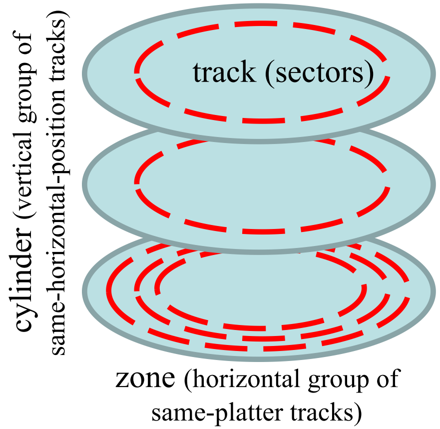




File Processing & External Sorting

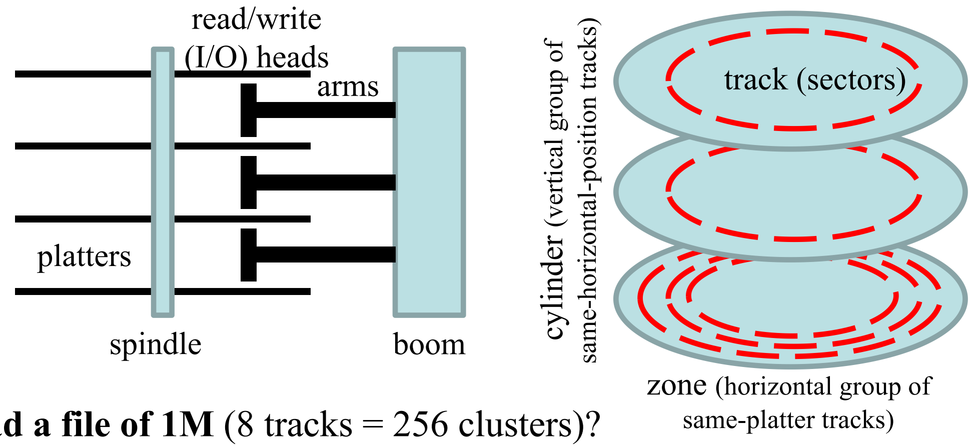
- **Disk drives**
  - disk drive architecture
  - disk drive acess
    - seek
    - preparatory rotation
    - data transfer
  - disk drive access time (e.g.)
    - disk drive (20G) = 10 platters ; platter = 16384 tracks
    - track (128K) = 32 clusters = 256 sectors ; sector = 0.5K ; cluster = 4K
    - track-to-track seek = 2.0ms ; random seek = 8.0ms ; rotation = 8.33ms (7200 RPM)
    - **how long will it take to read a file of 1M** (1024K = 8 tracks = 256 clusters)?
    - if the file fills 8 adjacent tracks : 122ms
    - if the file fills 256 clusters spread randomly across the disk
      - 256x[8.0+8.33x(0.5+8/256)]=256x(8.0+4.43)=3182ms





File Processing & External Sorting

- **Disk drives**
  - disk drive architecture
  - disk drive acess
    - seek
    - preparatory rotation
    - data transfer
  - disk drive access time (e.g.)
    - **how long will it take to read a file of 1M** (8 tracks = 256 clusters)?
    - if the file fills 8 adjacent tracks : 122ms
    - if the file fills 256 clusters spread randomly across the disk : 3182ms
  - importance of aggregating a file's sectors & avoiding file fragmentation
    - file fragmentation happens most commonly when the disk is nearly full, and when the file manager must search for free space for a created or changed file



File Processing & External Sorting

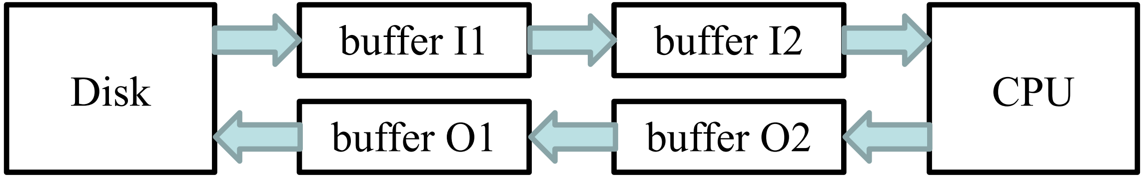
- **Buffering & caching**
  - disk drive access time (e.g.)
    - disk drive (20G) = 10 platters ; platter = 16384 tracks
    - track (128K) = 32 clusters = 256 sectors ; sector = 0.5K ; cluster = 4K
    - track-to-track seek = 2.0ms ; random seek = 8.0ms ; rotation = 8.33ms (7200 RPM)
    - access a track : 8.0+8.33x(0.5+1)=20.50ms
    - access a sector : 8.0+8.33x(0.5+1/256)=12.20ms (saves almost a half of track-access time)
    - access a byte : 8.0+8.33x(0.5+1/128K)=12.17ms (save almost none of sector-access time)

File Processing & External Sorting

- **Buffering & caching**
  - disk drive access time (e.g.)
    - disk drive (20G) = 10 platters ; platter = 16384 tracks
    - track (128K) = 32 clusters = 256 sectors ; sector = 0.5K ; cluster = 4K
    - track-to-track seek = 2.0ms ; random seek = 8.0ms ; rotation = 8.33ms (7200 RPM)
    - access a track : 8.0+8.33x(0.5+1)=20.50ms
    - access a sector : 8.0+8.33x(0.5+1/256)=12.20ms (saves almost a half of track-access time)
    - access a byte : 8.0+8.33x(0.5+1/128K)=12.17ms (save almost none of sector-access time)
  - **automatic reading/writing of an entire sector**
  - *buffering* a.k.a. *caching*
    - take/send additional information from/to disk to satisfy (potential) future requests
    - most operating systems maintain at least two buffers, one for input, one for output

File Processing & External Sorting

- **Buffering & caching**
  - disk drive access time (e.g.)
    - access a track : 8.0+8.33x(0.5+1)=20.50ms
    - access a sector : 8.0+8.33x(0.5+1/256)=12.20ms (saves almost a half of track-access time)
    - access a byte : 8.0+8.33x(0.5+1/128K)=12.17ms (save almost none of sector-access time)
  - automatic reading/writing of an entire sector
  - *buffering* a.k.a. *caching*
    - take/send additional information from/to disk to satisfy (potential) future requests
    - most operating systems maintain at least two buffers, one for input, one for output
  - double buffering



File Processing & External Sorting

- **Buffering & caching**
  - automatic reading/writing of an entire sector
  - *buffering* a.k.a. *caching*
    - take/send additional information from/to disk to satisfy (potential) future requests
    - most operating systems maintain at least two buffers, one for input, one for output
  - double buffering
  - **buffer pool** (collection of buffers)
    - may store many buffers of information taken from *backing storage* such as disk files
    - using buffers as intermediary between a user & a disk file is called *buffering the file*
    - the information stored in a buffer is called a *page*
    - maintenance (update) of buffer pool - decisions based on *heuristics*
      - e.g. LFU (least frequently used)
    - virtual memory

File Processing & External Sorting

- **C++ programmer's logical view of files**
  - logical view of a random access file is a single stream of bytes
  - file interaction viewed as a communication channel for issuing three instructions
    - read bytes from the current position in the file
    - write bytes to the current position in the file
    - move the current position within the file
  - random access
    - process records in an order independent of their logical order within the file
  - sequential access
    - process records in order of their logical appearance within the file

File Processing & External Sorting

- **C++ programmer's logical view of files**
  - logical view of a random access file is a single stream of bytes
  - file interaction viewed as a communication channel for issuing three instructions
    - read bytes from the current position in the file
    - write bytes to the current position in the file
    - move the current position within the file
  - sequential access better than random access
  - C++ mechanisms for manipulating disk files (e.g. **fstream** class)
    - *open*(char *name, openmode flags)
    - *read*(char *buff, int count)
    - *write*(char *buff, int count)
    - *seekg*(int pos) & *seekp*(int pos) : for "get" (read) & "put" (write) positions
    - *close*()

File Processing & External Sorting

- **External sorting**
  - sort collections of records too large to fit in main memory
    - e.g. process payrolls & other large business databases
  - external divide & conquer
    - read some records from disk, do some rearranging, then write them back to disk; repeat until the file is sorted, with each record read perhaps many times (no choice other than this)
    - primary goal of an external sorting algorithm is to *minimize the number of times when information must be read from or written to disk*
    - since reading/writing a block from disk takes the order of 10^6 times longer than a memory access, can be reasonably expected that a single block's records can be sorted by an internal sorting algorithm such as *Quicksort* in less time than is required to read/write the block

File Processing & External Sorting

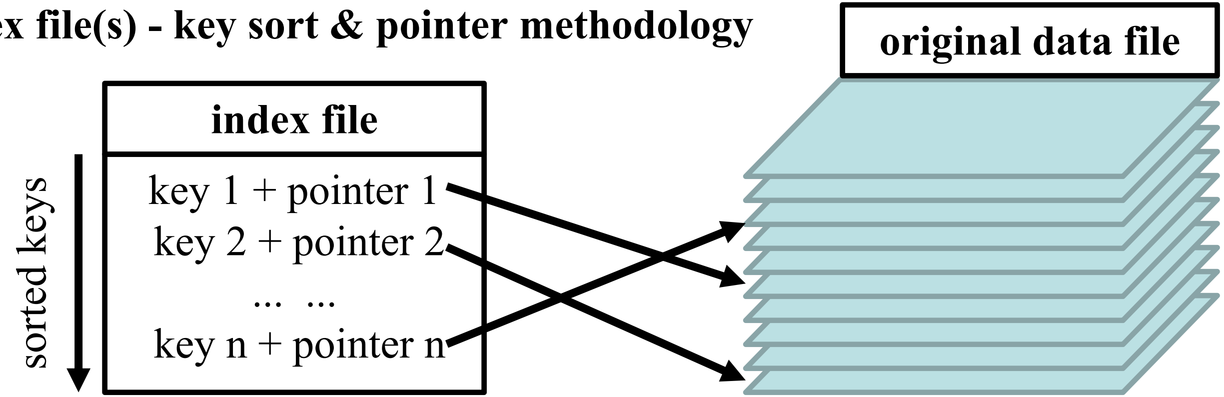
- **External sorting**
  - sort collections of records too large to fit in main memory
  - external divide & conquer
    - read some records from disk, do some rearranging, then write them back to disk; repeat until the file is sorted, with each record read perhaps many times
    - *minimize the number of times when information must be read from or written to disk*
  - sequential processing seems to be obviously faster; however
    - required that blocks making up a file are indeed stored on disk in sequential order & close together, preferably filling a small number of contiguous tracks
    - required that the disk I/O head remains positioned over the file, which may not happen
      - if competition for the disk I/O head exists (e.g. on a multi-user time-shared system)
      - if reading from an input file & writing to an output file are alternated frequently and consequently the disk I/O head will continuously seek between the input & output files

File Processing & External Sorting

- **External sorting**
  - sort collections of records too large to fit in main memory
  - external divide & conquer
    - read some records from disk, do some rearranging, then write them back to disk; repeat until the file is sorted, with each record read perhaps many times
    - *minimize the number of times when information must be read from or written to disk*
  - sequential processing seems to be obviously faster; however, there is usually no such ideal thing as efficient sequential processing of a data file
  - usually perform a smaller number of non-sequential disk operations
    - rather than a larger number of logically sequential disk operations that require a large number of seeks in practice

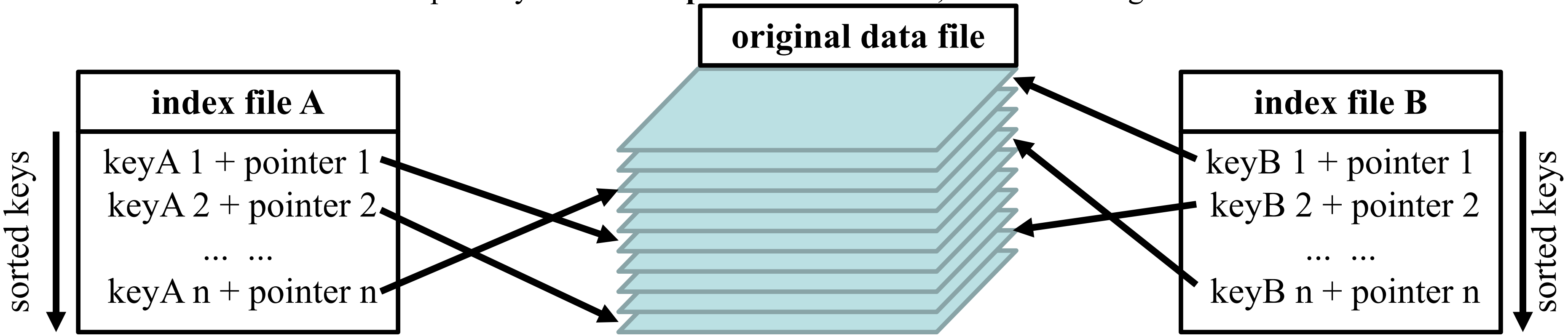
File Processing & External Sorting

- **External sorting**
  - sort collections of records too large to fit in main memory
  - external divide & conquer
    - read some records from disk, do some rearranging, then write them back to disk; repeat until the file is sorted, with each record read perhaps many times
    - *minimize the number of times when information must be read from or written to disk*
  - usually perform a smaller number of non-sequential disk operations
  - **index file(s) - key sort & pointer methodology**



File Processing & External Sorting

- **External sorting**
  - sort collections of records too large to fit in main memory
  - external divide & conquer
  - usually perform a smaller number of non-sequential disk operations
  - **index file(s) - key sort & pointer methodology**
    - no need to reorder records in the original database file
    - especially when **multiple index files** exist, there is no single "sorted" order of all records



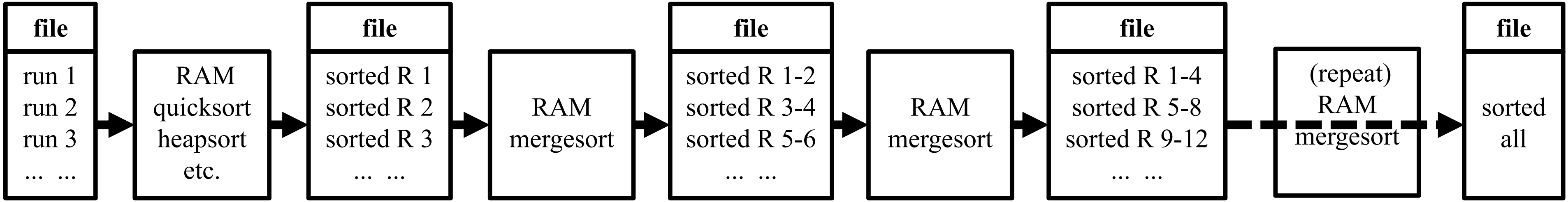
File Processing & External Sorting

- **External sorting**
  - sort collections of records too large to fit in main memory
  - external divide & conquer
  - **external mergesort**
    - merge two *sorted* sub-sequences into *sorted* one
    - get the min elements of both sub-sequences immediately & pop the smaller one

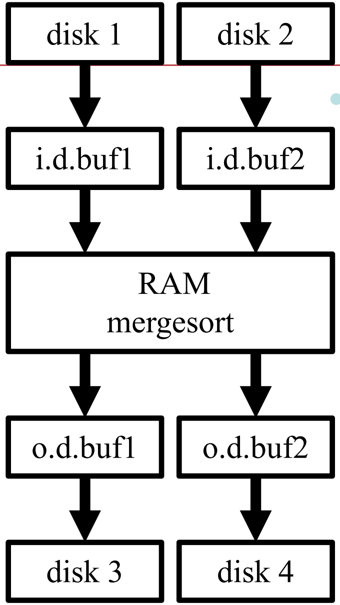
unsorted	89	11	43	71	37	23	67	59
merge every two	11	89	43	71	23	37	59	67
merge every four	11	43	71	89	23	37	59	67
merge every eight (all)	11	23	37	43	59	67	71	89

File Processing & External Sorting

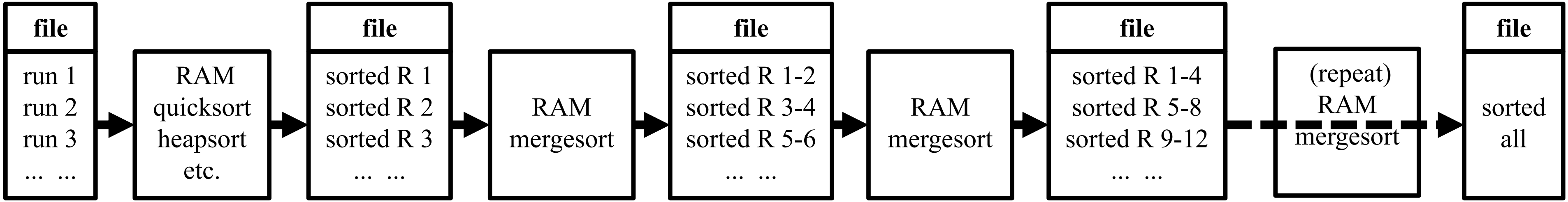
- **External sorting**
  - sort collections of records too large to fit in main memory
  - external divide & conquer
  - **external mergesort**
    - merge two *sorted* sub-sequences into *sorted* one
    - get the min elements of both sub-sequences immediately & pop the smaller one
    - **break the file into large initial runs** (which can be sorted by internal sorts say *Quicksort*)
    - **merge the runs together to form a single sorted file**



File Processing & External Sorting

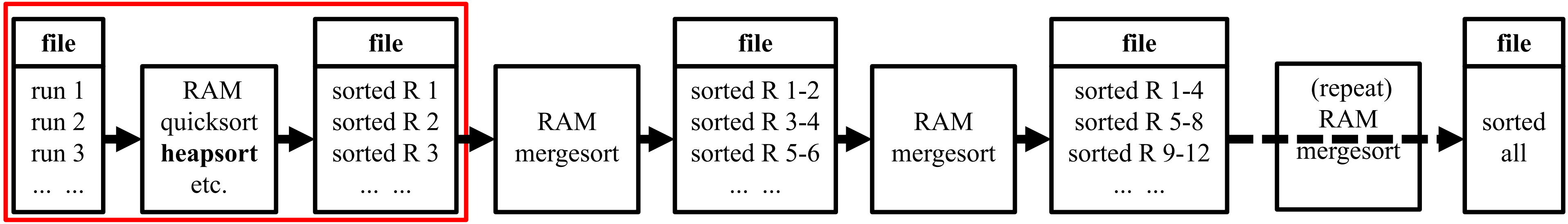


- **External sorting**
  - sort collections of records too large to fit in main memory
  - external divide & conquer
  - **external mergesort**
    - merge two *sorted* sub-sequences into *sorted* one
    - get the min elements of both sub-sequences immediately & pop the smaller one
    - **break the file into large initial runs** (which can be sorted by internal sorts say *Quicksort*)
    - **merge the runs together to form a single sorted file**
    - can easily take advantage of sequential processing & double buffering
      - better have a total of four disk drives for maximum efficiency



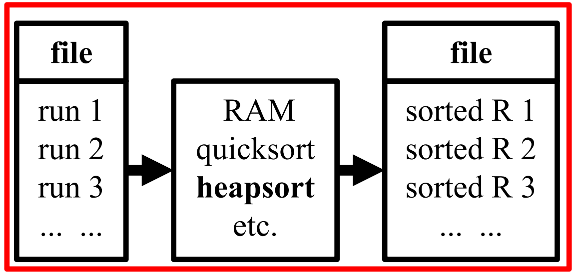
File Processing & External Sorting

- **External sorting**
  - sort collections of records too large to fit in main memory
  - external divide & conquer
  - external mergesort
    - break the file into large initial runs
    - merge the runs together to form a single sorted file
    - can easily take advantage of sequential processing & double buffering
  - **replacement selection**



File Processing & External Sorting

- **External sorting**
  - external mergesort
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R's key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap



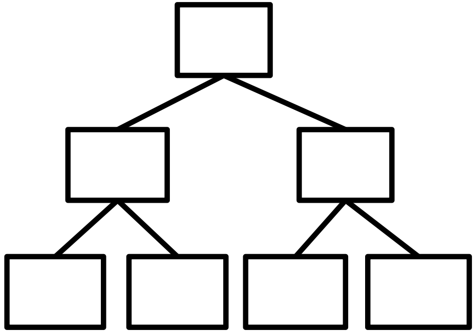
File Processing & External Sorting

- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R's key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer

42, 92, 96, 79, 93, 4, 85, 66, 68, 63, 39, 80, 76, 74, 17, 71, 100, 97, 3

output buffer

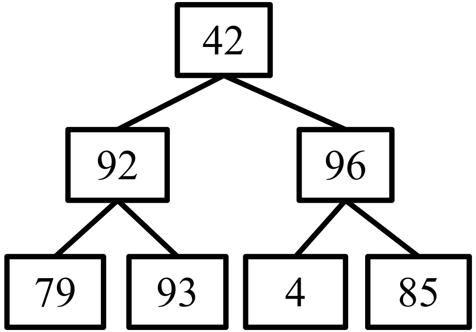


File Processing & External Sorting

- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R's key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer

66, 68, 63, 39, 80, 76, 74, 17, 71, 100, 97, 3, ...



output buffer

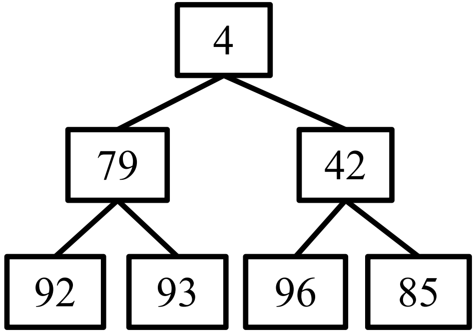
File Processing & External Sorting

- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R's key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer

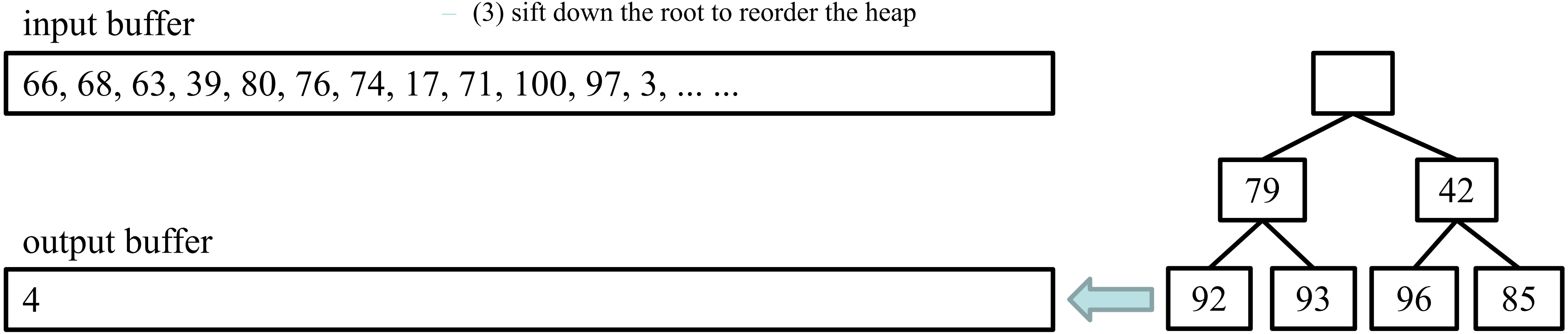
66, 68, 63, 39, 80, 76, 74, 17, 71, 100, 97, 3, ...

output buffer



File Processing & External Sorting

- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - » (a) if R's key value > the root key value just output, then place R at the heap root
        - » (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

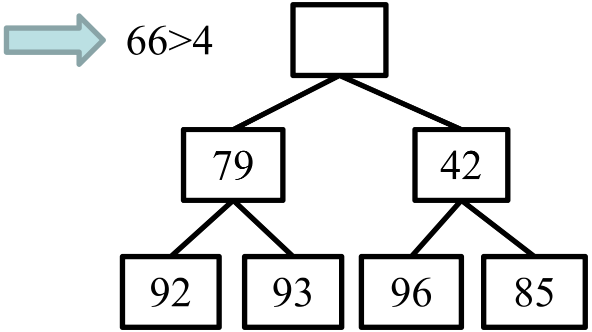


File Processing & External Sorting

- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R's key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer

68, 63, 39, 80, 76, 74, 17, 71, 100, 97, 3, ...



output buffer

4

File Processing & External Sorting

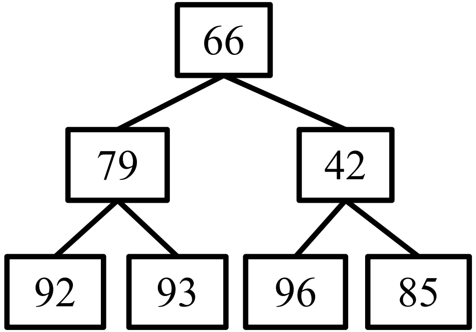
- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R's key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer

68, 63, 39, 80, 76, 74, 17, 71, 100, 97, 3, ...

output buffer

4



File Processing & External Sorting

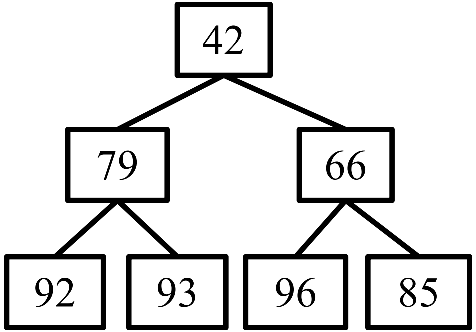
- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R's key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer

68, 63, 39, 80, 76, 74, 17, 71, 100, 97, 3, ...

output buffer

4



File Processing & External Sorting

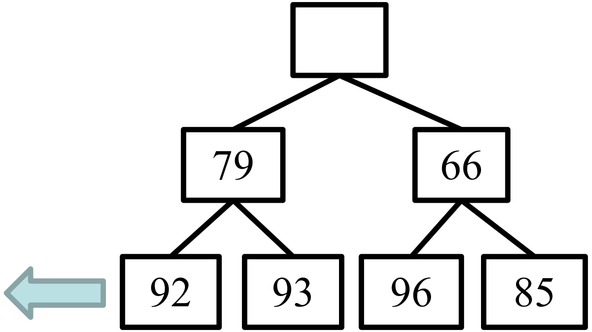
- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R's key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer

68, 63, 39, 80, 76, 74, 17, 71, 100, 97, 3, ...

output buffer

4, 42



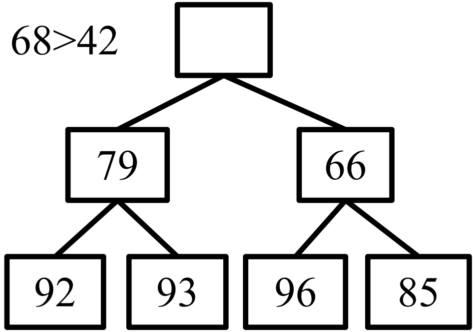
File Processing & External Sorting

- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R's key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer

63, 39, 80, 76, 74, 17, 71, 100, 97, 3, ...

68>42

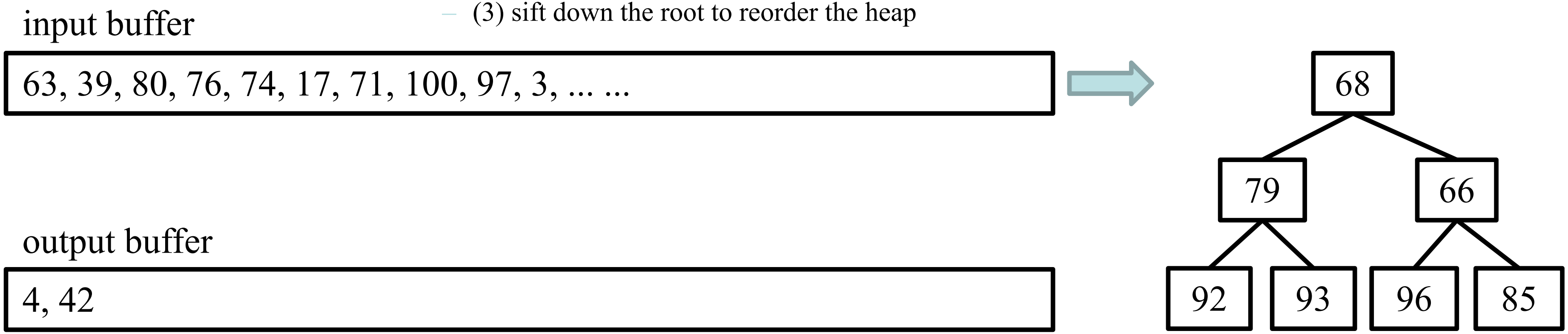


output buffer

4, 42

File Processing & External Sorting

- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R's key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap



File Processing & External Sorting

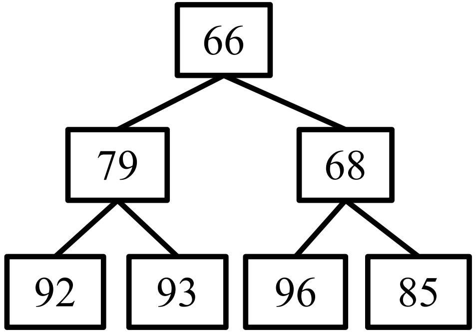
- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R's key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer

63, 39, 80, 76, 74, 17, 71, 100, 97, 3, ...

output buffer

4, 42



File Processing & External Sorting

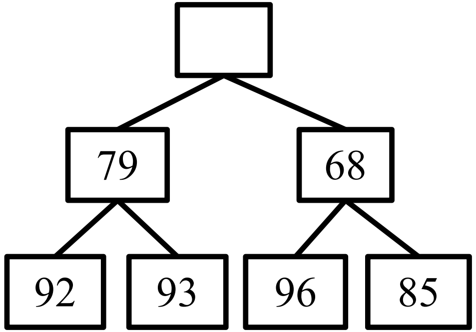
- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R's key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer

63, 39, 80, 76, 74, 17, 71, 100, 97, 3, ...

output buffer

4, 42, 66



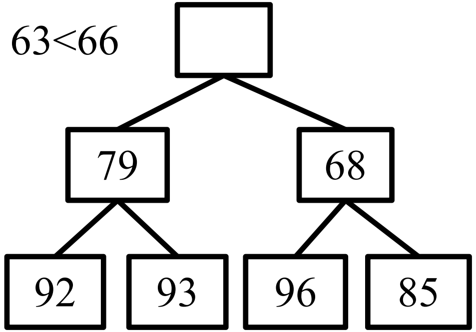
File Processing & External Sorting

- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R's key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer

39, 80, 76, 74, 17, 71, 100, 97, 3, ...

63<66

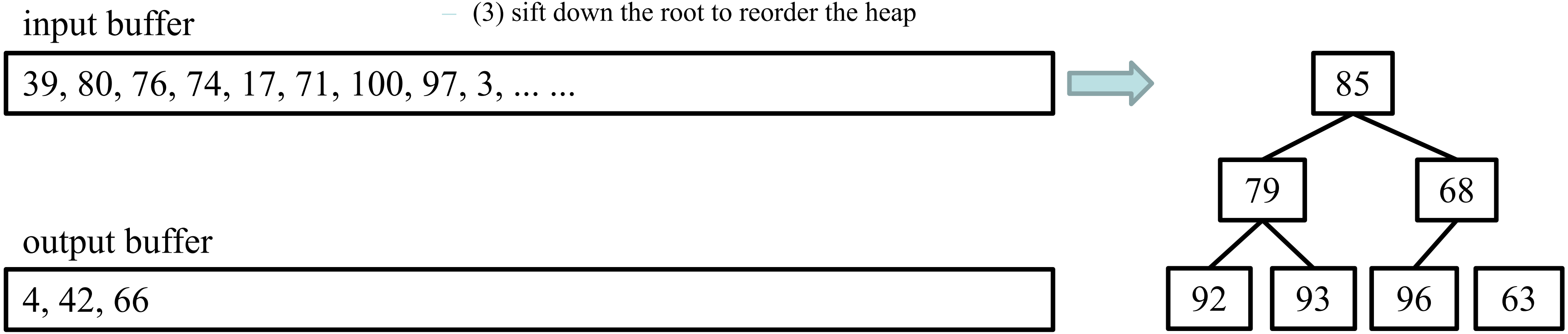


output buffer

4, 42, 66

File Processing & External Sorting

- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R’s key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap



File Processing & External Sorting

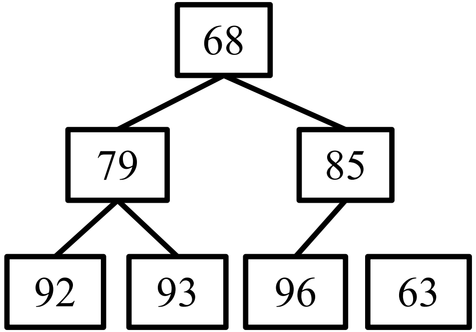
- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R's key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer

39, 80, 76, 74, 17, 71, 100, 97, 3, ...

output buffer

4, 42, 66



File Processing & External Sorting

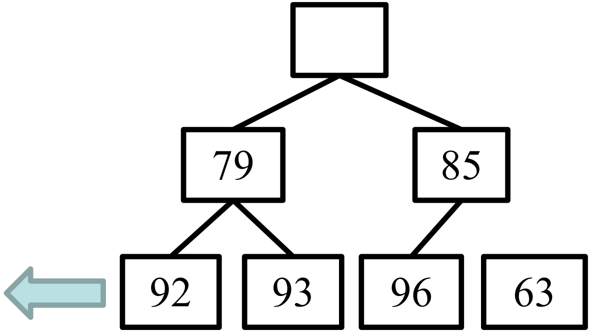
- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R's key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer

39, 80, 76, 74, 17, 71, 100, 97, 3, ...

output buffer

4, 42, 66, 68



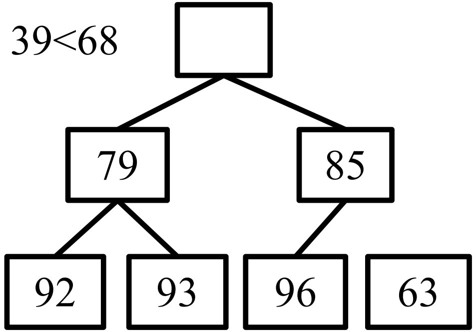
File Processing & External Sorting

- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R's key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer

80, 76, 74, 17, 71, 100, 97, 3, ...

39<68



output buffer

4, 42, 66, 68

File Processing & External Sorting

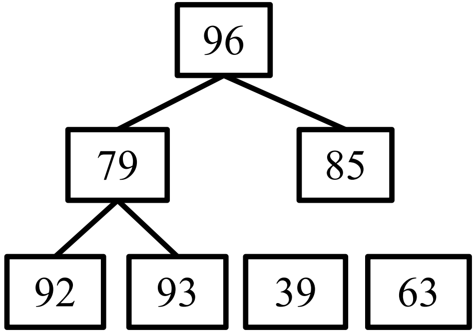
- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R’s key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer

80, 76, 74, 17, 71, 100, 97, 3, ...

output buffer

4, 42, 66, 68



File Processing & External Sorting

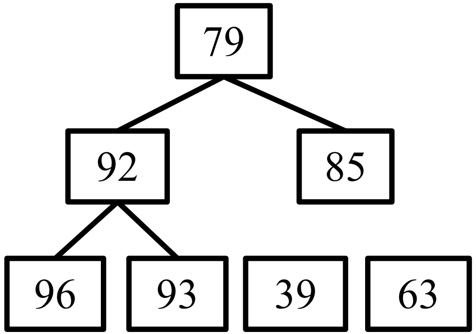
- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R's key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer

80, 76, 74, 17, 71, 100, 97, 3, ...

output buffer

4, 42, 66, 68



File Processing & External Sorting

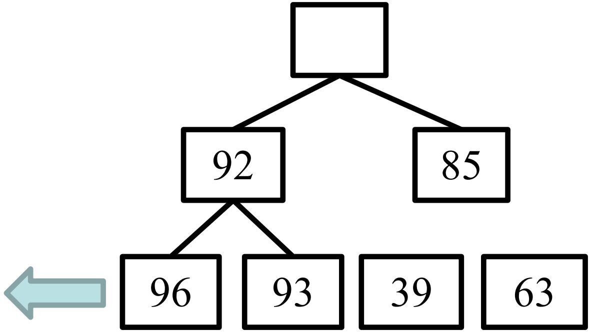
- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R's key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer

80, 76, 74, 17, 71, 100, 97, 3, ...

output buffer

4, 42, 66, 68, 79



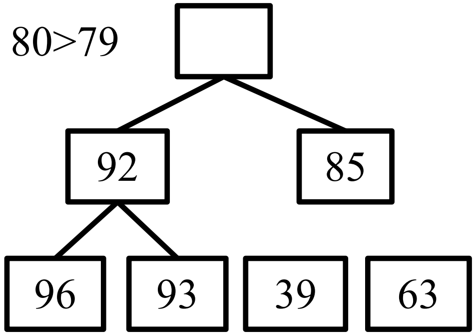
File Processing & External Sorting

- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R's key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer

76, 74, 17, 71, 100, 97, 3, ...

80>79



output buffer

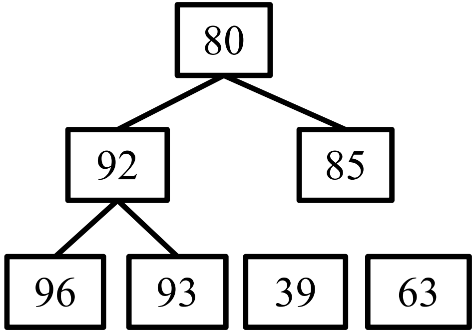
4, 42, 66, 68, 79

File Processing & External Sorting

- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R’s key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer

76, 74, 17, 71, 100, 97, 3, ...



output buffer

4, 42, 66, 68, 79

File Processing & External Sorting

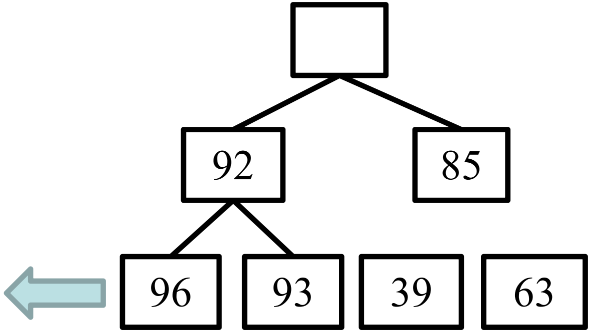
- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R's key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer

76, 74, 17, 71, 100, 97, 3, ...

output buffer

4, 42, 66, 68, 79, 80

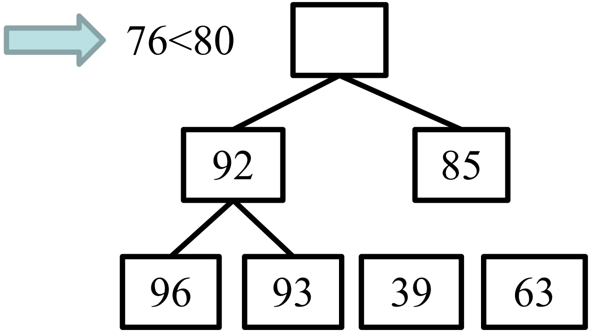


File Processing & External Sorting

- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R's key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer

74, 17, 71, 100, 97, 3, ...



output buffer

4, 42, 66, 68, 79, 80

File Processing & External Sorting

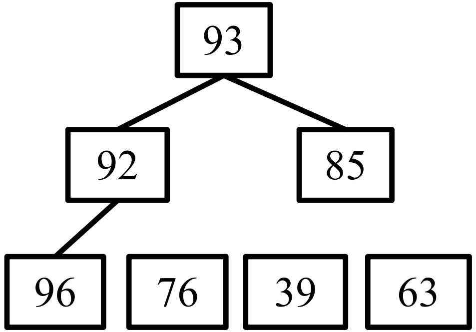
- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R's key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer

74, 17, 71, 100, 97, 3, ...

output buffer

4, 42, 66, 68, 79, 80



File Processing & External Sorting

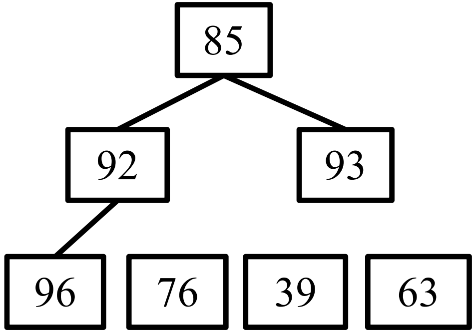
- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R's key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer

74, 17, 71, 100, 97, 3, ...

output buffer

4, 42, 66, 68, 79, 80



File Processing & External Sorting

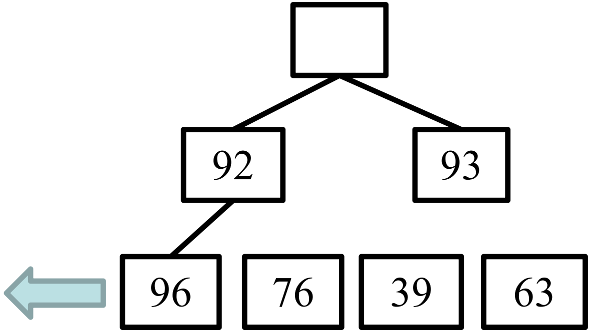
- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R's key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer

74, 17, 71, 100, 97, 3, ...

output buffer

4, 42, 66, 68, 79, 80, 85

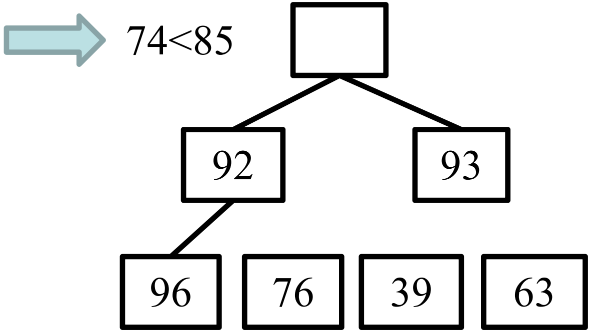


File Processing & External Sorting

- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R's key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer

17, 71, 100, 97, 3, … …



output buffer

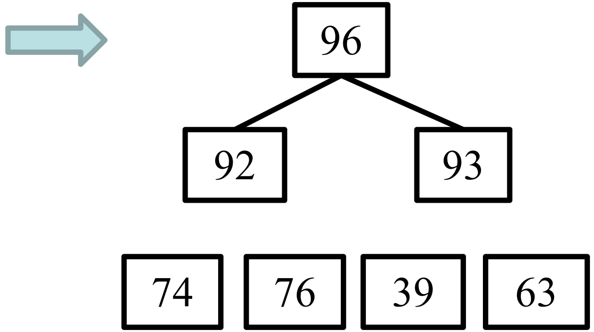
4, 42, 66, 68, 79, 80, 85

File Processing & External Sorting

- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R's key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer

17, 71, 100, 97, 3, ...



output buffer

4, 42, 66, 68, 79, 80, 85

File Processing & External Sorting

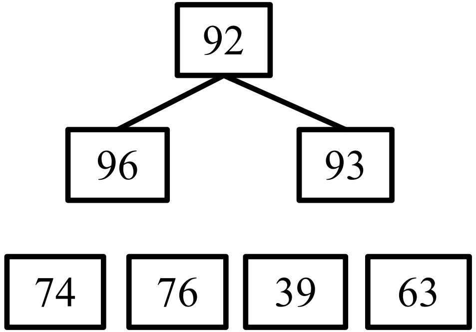
- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R's key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer

17, 71, 100, 97, 3, ...

output buffer

4, 42, 66, 68, 79, 80, 85



File Processing & External Sorting

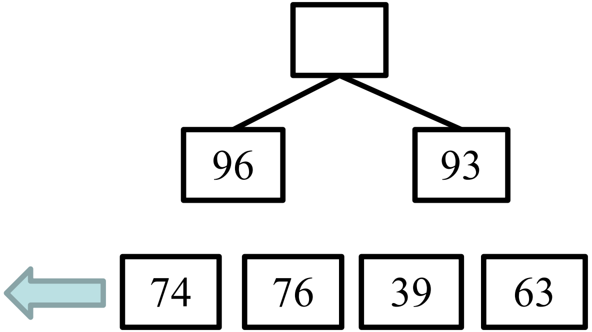
- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R's key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer

17, 71, 100, 97, 3, ...

output buffer

4, 42, 66, 68, 79, 80, 85, 92

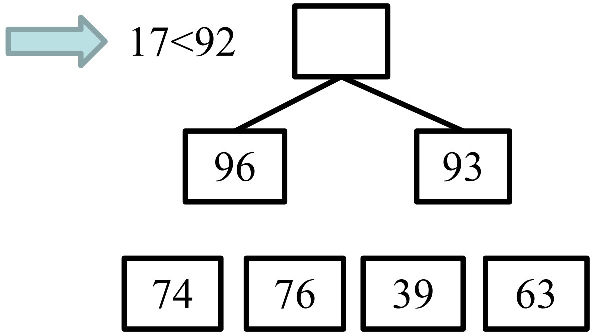


File Processing & External Sorting

- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R's key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer

71, 100, 97, 3, ...



output buffer

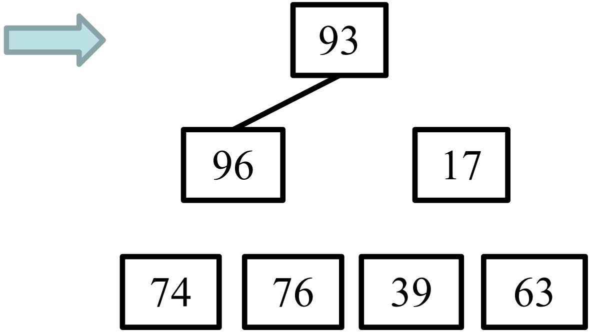
4, 42, 66, 68, 79, 80, 85, 92

File Processing & External Sorting

- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R's key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer

71, 100, 97, 3, … …



output buffer

4, 42, 66, 68, 79, 80, 85, 92

File Processing & External Sorting

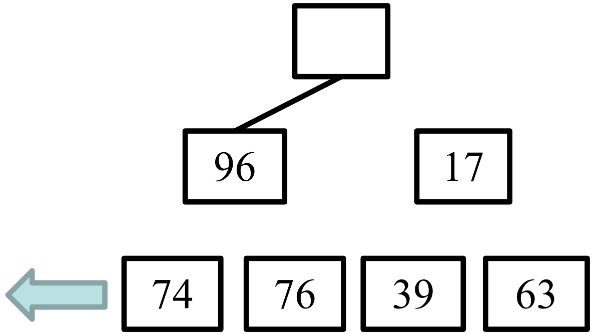
- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R's key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer

71, 100, 97, 3, ...

output buffer

4, 42, 66, 68, 79, 80, 85, 92, 93



File Processing & External Sorting

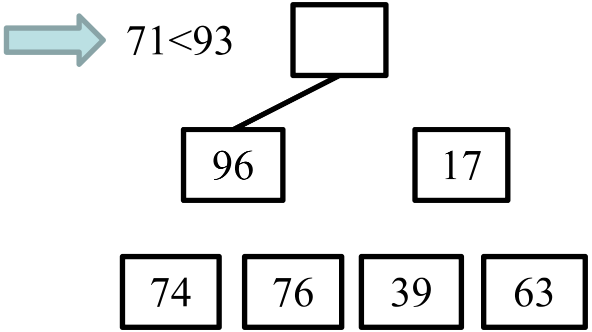
- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R's key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer

100, 97, 3, ...

output buffer

4, 42, 66, 68, 79, 80, 85, 92, 93



File Processing & External Sorting

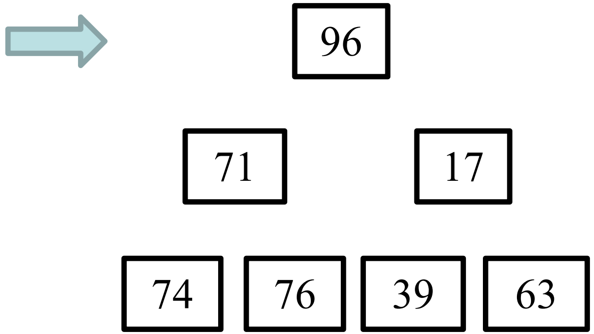
- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R's key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer

100, 97, 3, … …

output buffer

4, 42, 66, 68, 79, 80, 85, 92, 93



File Processing & External Sorting

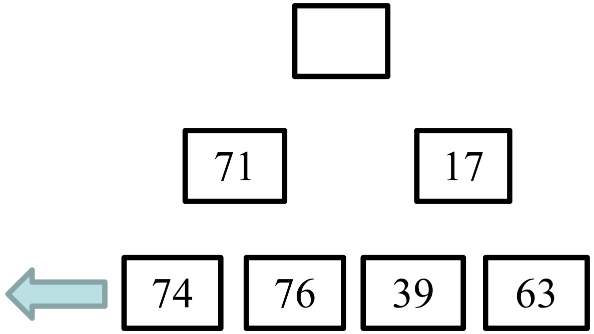
- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R's key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer

100, 97, 3, … …

output buffer

4, 42, 66, 68, 79, 80, 85, 92, 93, 96



File Processing & External Sorting

- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - » (a) if R's key value > the root key value just output, then place R at the heap root
        - » (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer

97, 3, … …

100>96

71 17

74 76 39 63

output buffer

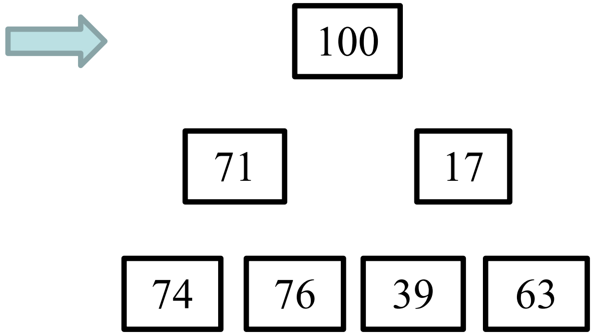
4, 42, 66, 68, 79, 80, 85, 92, 93, 96

File Processing & External Sorting

- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R’s key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer

97, 3, … …



output buffer

4, 42, 66, 68, 79, 80, 85, 92, 93, 96

File Processing & External Sorting

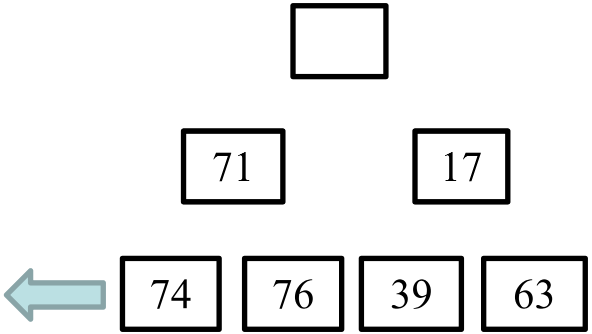
- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R's key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer

97, 3, … …

output buffer

4, 42, 66, 68, 79, 80, 85, 92, 93, 96, 100



File Processing & External Sorting

- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R's key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer

3, … …

97<100

71 17

74 76 39 63

output buffer

4, 42, 66, 68, 79, 80, 85, 92, 93, 96, 100

File Processing & External Sorting

- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R's key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer

3, … …

heap empty! 97

71 17

74 76 39 63

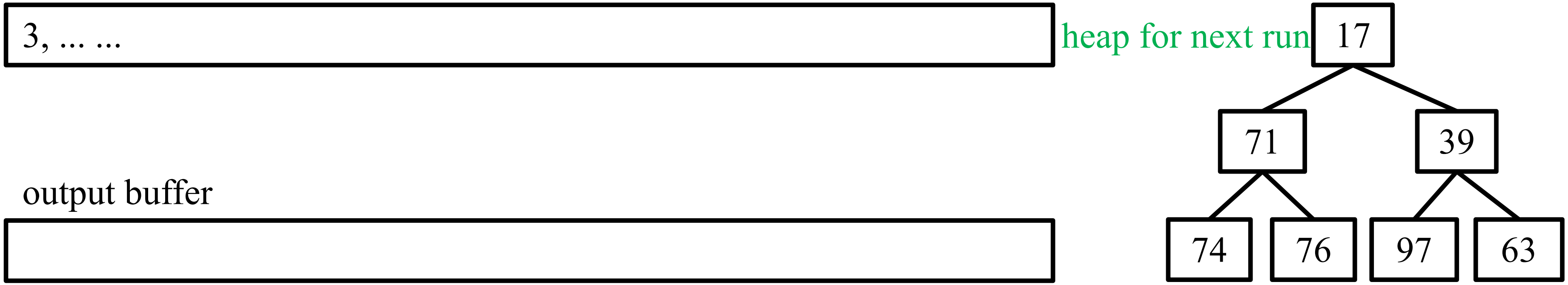
output buffer - the first sorted run is done

4, 42, 66, 68, 79, 80, 85, 92, 93, 96, 100

File Processing & External Sorting

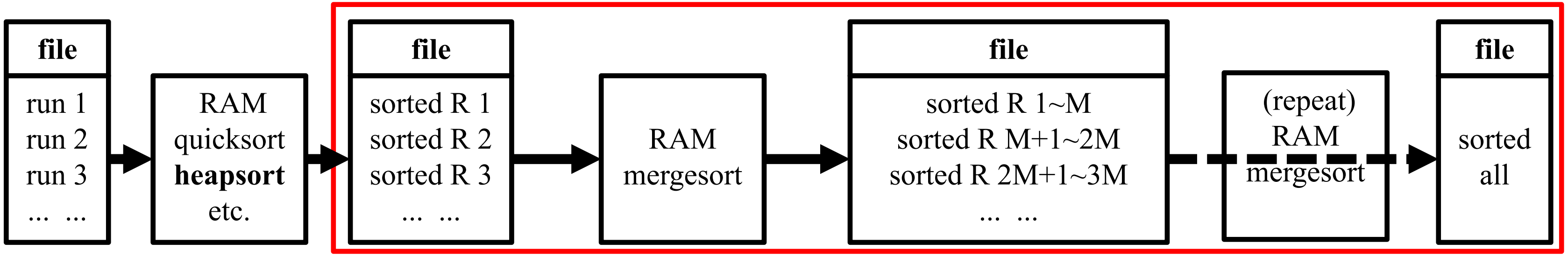
- **External sorting**
  - **replacement selection**
    - **Init**: fill the array (of size n) from disk & initialize a min-heap; set L=n-1
    - **Loop**: repeat until the array is empty
      - (1) send the root record (i.e. that with the minimum key value) to the output buffer
      - (2) take the next record R from the input buffer
        - (a) if R's key value > the root key value just output, then place R at the heap root
        - (b) otherwise, replace the heap root with record at position L; place R at position L; set L=L-1
      - (3) sift down the root to reorder the heap

input buffer



File Processing & External Sorting

- **External sorting**
  - sort collections of records too large to fit in main memory
  - external divide & conquer
  - external mergesort
  - replacement selection
  - **multiway merging**
    - if we have M sorted runs to merge, with a block from each run available in memory, then the M-way merge algorithm looks at M runs' front-most values & selects the smallest one to output (if M is not small, a heap can be used to maintain the M runs' front-most values)



THANK YOU

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY