CHAPTER 7

SORTING

All the programs in this file are selected from

Ellis Horowitz, Sartaj Sahni, and Susan Anderson-Freed

“Fundamentals of Data Structures in C”,

Internal Sorting

Insertion sort	(page 337)
Quick sort	(page 340)
Merge sort	(page 346)
Heap sort	(page 352)
Radix sort	(page 356)
External sort	(page 376)
Bubble sort	
Selection sort	(page 9)

Sequential Search

- Example

  44, 55, 12, 42, 94, 18, 06, 67
- unsuccessful search
  - n times
- successful search

$$\left(\sum_{i=1}^{n} i\right) / n = \frac{n(n+1)}{2n} = \frac{n+1}{2}$$

Program 7.1:Sequential search (p.334)

```
int seqsearch( element a[], int k, int n)
{
/*search a[1:n]; return the least i such that a[i].key = k; return 0,
if k is not l the array.*/
    int i;
    for (i=1; i<= n && a[i].key != k; i++)
      ;
    if (i > n) return 0;
    return i;
}
```

left right

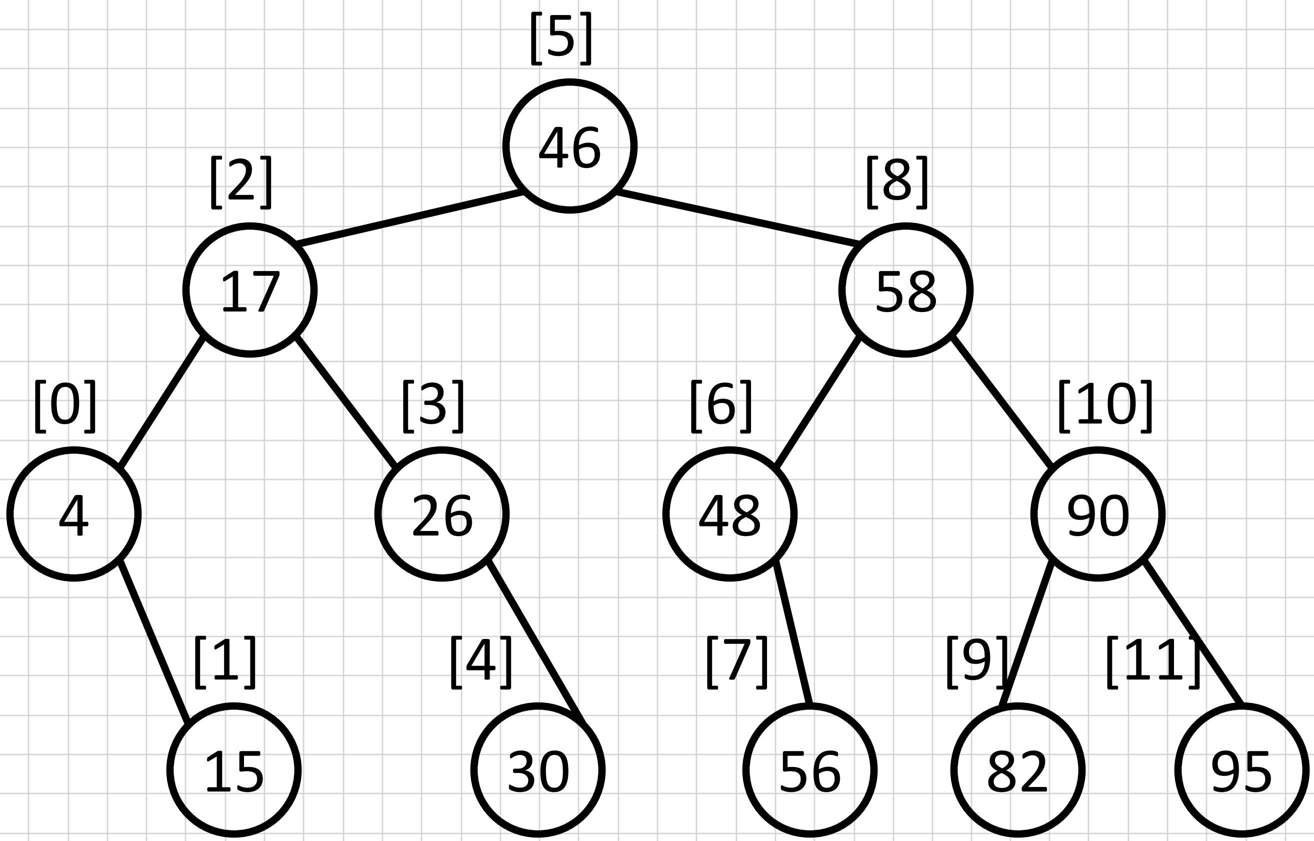
4, 15, 17, 26, 30, 46, 48, 56, 58, 82, 90, 95

Binary search

```
int binsearch(element list[ ], int searchnum, int n)
{
/* search list [0], ..., list[n-1]*/
    int left = 0, right = n-1, middle;
    while (left <= right) {
        middle = (left+ right)/2;
    switch (COMPARE(list[middle].key, searchnum)) {
        case -1: left = middle +1;
                    break;
        case  0: return middle;
        case  1:right = middle - 1;
        }
      }
      return -1;
}
```

$O(\log_2 n)$

***Figure :** Decision tree for binary search



4, 15, 17, 26, 30, 46, 48, 56, 58, 82, 90, 95

List Verification

- Compare two lists to verify that they are identical or identify the discrepancies.
- example
  - international revenue service (e.g., employee vs. employer)
- complexities
  - random order: $O(mn)$
  - ordered list: $O(tsort(n)+tsort(m)+m+n)$

__Program 7.2:__ verifying using a sequential search(p.336)

```
void verify1(element list1[], element list2[ ], int n, int m)
{
int i, j;
int marked[MAX_SIZE];

for(i = 0; i<m; i++)
    marked[i] = FALSE;
for (i=0; i<n; i++)
  if ((j = seqsearch(list2, m, list1[i].key)) < 0)
    printf("%d is not in list 2\n ", list1[i].key);
 else
    marked[j] = TRUE;
 for ( i=0; i<m; i++)
      if (!marked[i])
          printf("%d is not in list1\n", list2[i]key);

}
```

compare two unordered lists list1 and list2

(a) all records found in list1 but not in list2
(b) all records found in list2 but not in list1
(c) all records that are in list1 and list2 with the same key but have different values for different fields.

check each of the other fields from list1[i] and list2[j], and print out any discrepancies

***Program 7.3:** Fast verification of two lists (p.337)

```
void verify2(element list1[ ], element list2 [ ], int n, int m)
{
   int i, j;
   sort(list1, n);
   sort(list2, m);
   i = j = 0;
   while (i < n && j < m)
        if  (list1[i].key < list2[j].key) {
             printf (“%d is not in list 2 \n”, list1[i].key);
             i++;
        }
        else if (list1[i].key == list2[j].key) {
             i++; j++;
        }
        else {
             printf(“%d is not in list 1\n”, list2[j].key);
             j++;
        }
       for(; i < n; i++)
            printf (“%d is not in list 2\n”, list1[i].key);
       for(; j < m; j++)
            printf(“%d is not in list 1\n”, list2[j].key);
}
```

Same task as verify1, but list1 and list2 are sorted

compare list1[i] and list2[j] on each of the other field and report any discrepancies

List1: 1 2 4 9 11

List2: 2 3 5 7 13

Sorting Problem

- Definition
  - given $(R_0, R_1, \ldots, R_{n-1})$, where each record R_i has key value K_i, find a permutation σ, such that $K_{\sigma(i-1)} \leq K_{\sigma(i)}$, $0<i<n-1$
- sorted
  - $K_{\sigma(i-1)} \leq K_{\sigma(i)}$, $0<i<n-1$
- stable
  - if $i < j$ and $K_i = K_j$ then R_i precedes R_j in the sorted list
- internal sort vs. external sort
- criteria
  - # of key comparisons
  - # of data movements

Insertion Sort

- Hypothesis: we know how to sort n-1 elements
- Induction on the n-th element
  - sort n-1 elements
  - put the n-th element in its correct place by scanning the n-1 sorted elements
  - movements: O(n^2), comparison: O(n^2)
  - improvement
    - use binary search in finding correct place
    - comparison: O(nlogn), movement: O(n^2)

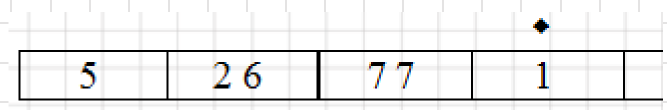
Insertion Sort

Find an element smaller than K.

26	5	77	1	61	11	59	15	48	19
5	26	77	1	61	11	59	15	48	19
5	26	77	1	61	11	59	15	48	19
1	5	26	77	61	11	59	15	48	19
1	5	26	61	77	11	59	15	48	19
1	5	11	26	61	77	59	15	48	19
1	5	11	26	59	61	77	15	48	19
1	5	11	15	26	59	61	77	48	19
1	5	11	15	26	48	59	61	77	19
5	26	1	61	11	59	15	48	19	77

Insertion Sort

```
void insertionSort(element a[], int n){
  int j;
  for(j = 2; j <= n; j++){
    element temp = a[j];
    insert(temp, a, j - 1);
  }
}
void insert(element e, element a[], int i){
  a[0] = e;
  while (e.key < a[i].key){
    a[i + 1] = a[i];
    i--;
  }
  a[i + 1] = e;
}
```



worse case

left out of order (LOO)

i	0	1	2	3	4
-	5	4	3	2	1
1	4	5	3	2	1
2	3	4	5	2	1
3	2	3	4	5	1
4	1	2	3	4	5

$$O(\sum_{j=0}^{n-2} i) = O(n^2)$$

best case

i	0	1	2	3	4
-	2	3	4	5	1
1	2	3	4	5	1
2	2	3	4	5	1
3	2	3	4	5	1
4	1	2	3	4	5

O(n)

R_i is LOO if $R_i < \max_{0 \le j < i}\{R_j\}$

k: # of records LOO

Computing time: O((k+1)n)

44	55	12	42	94	18	06	67
		*	*		*	*	*

Variation

- Binary insertion sort
  - sequential search --> binary search
  - reduce # of comparisons,
    # of moves unchanged
- List insertion sort
  - array --> linked list
  - sequential search, move --> 0

Selection Sort

- Hypothesis: we know how to sort n-1 elements
- Induction on the n-th element
  - sort n-1 elements
  - select the minimal element from unsorted as the n-th element
  - put it in a correct place by swapping
  - movements: O(n-1), comparison: O(n^2)

Example of Selection Sort

	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
Original	34	8	64	51	32	21
after pass 0	8	34	64	51	32	21
after pass 1	8	21	64	51	32	34
after pass 2	8	21	32	51	64	34
after pass 3	8	21	32	34	64	51
after pass 4	8	21	32	34	51	64

Algorithm of Selection Sort

```
void SelectionSort(ElemenType A[ ], int N) {
        int j,p;
        Element Type Min;
        for (P=0; P <= N-2; P++)  {
                Min=P;
                for (j=P+1; j <= N-1 ; j++) {
                        if (A[j] < A[Min])
                        Min=j;
                }
                exchange (A[P], A[Min]);
        }
 }
```

Recursive sorting algorithm

The best of the internal sorting in practical use

Worst case:

$O(n^2)$

Average case:

$O(n \log n)$

Quick Sort

Chapter 7.3

Page 340

Watch the video!

Quick Sort (C.A.R. Hoare)

- Given $(R_0, R_1, \ldots, R_{n-1})$

  K_i: pivot key

  if K_i is placed in $S(i)$,

  then $K_j \leq K_{s(i)}$ for $j < S(i)$,

  $K_j \geq K_{s(i)}$ for $j > S(i)$.

- $\underbrace{R_0, \ldots, R_{S(i)-1}}, R_{S(i)}, \underbrace{R_{S(i)+1}, \ldots, R_{S(n-1)}}$

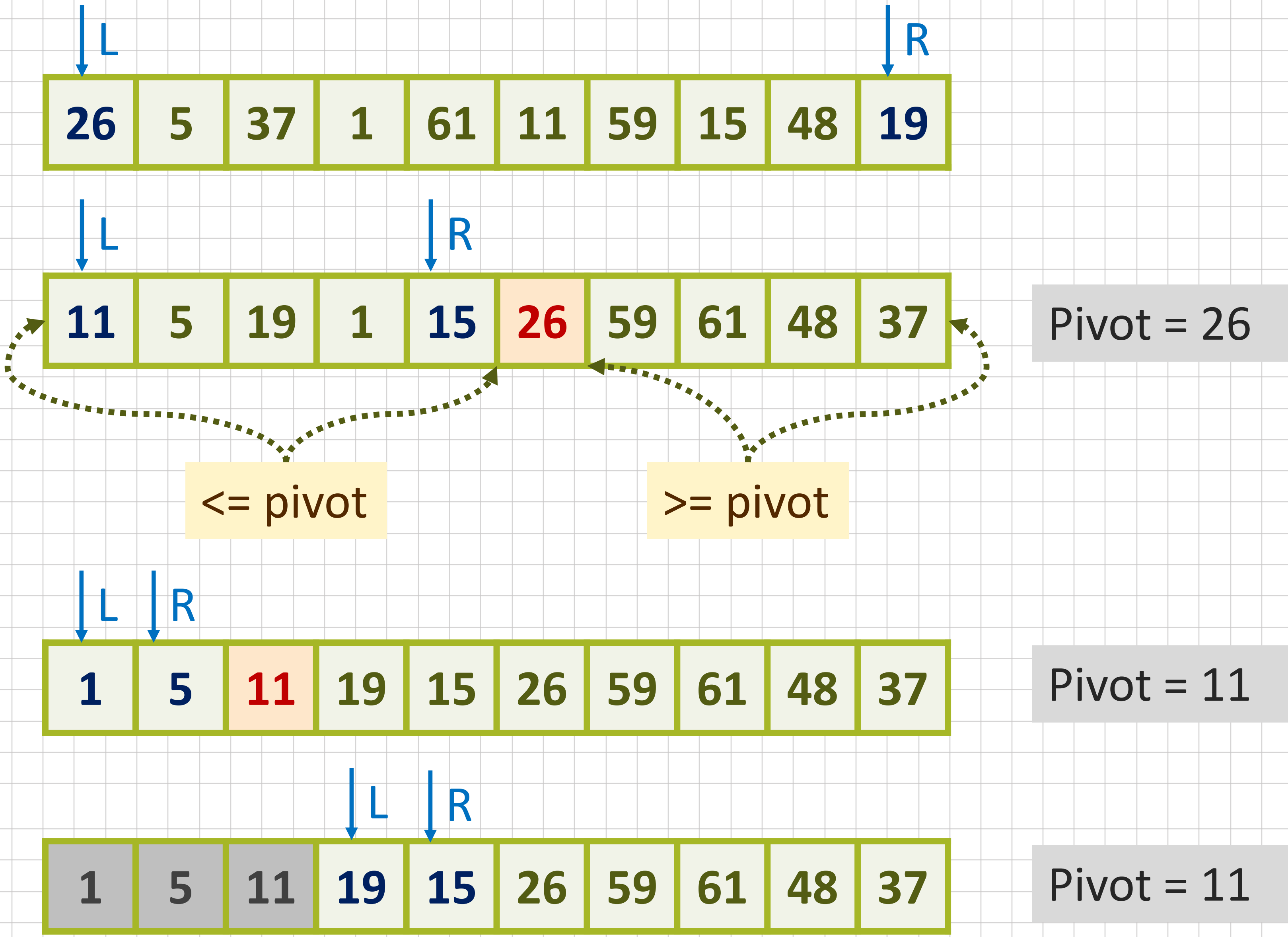
two partitions

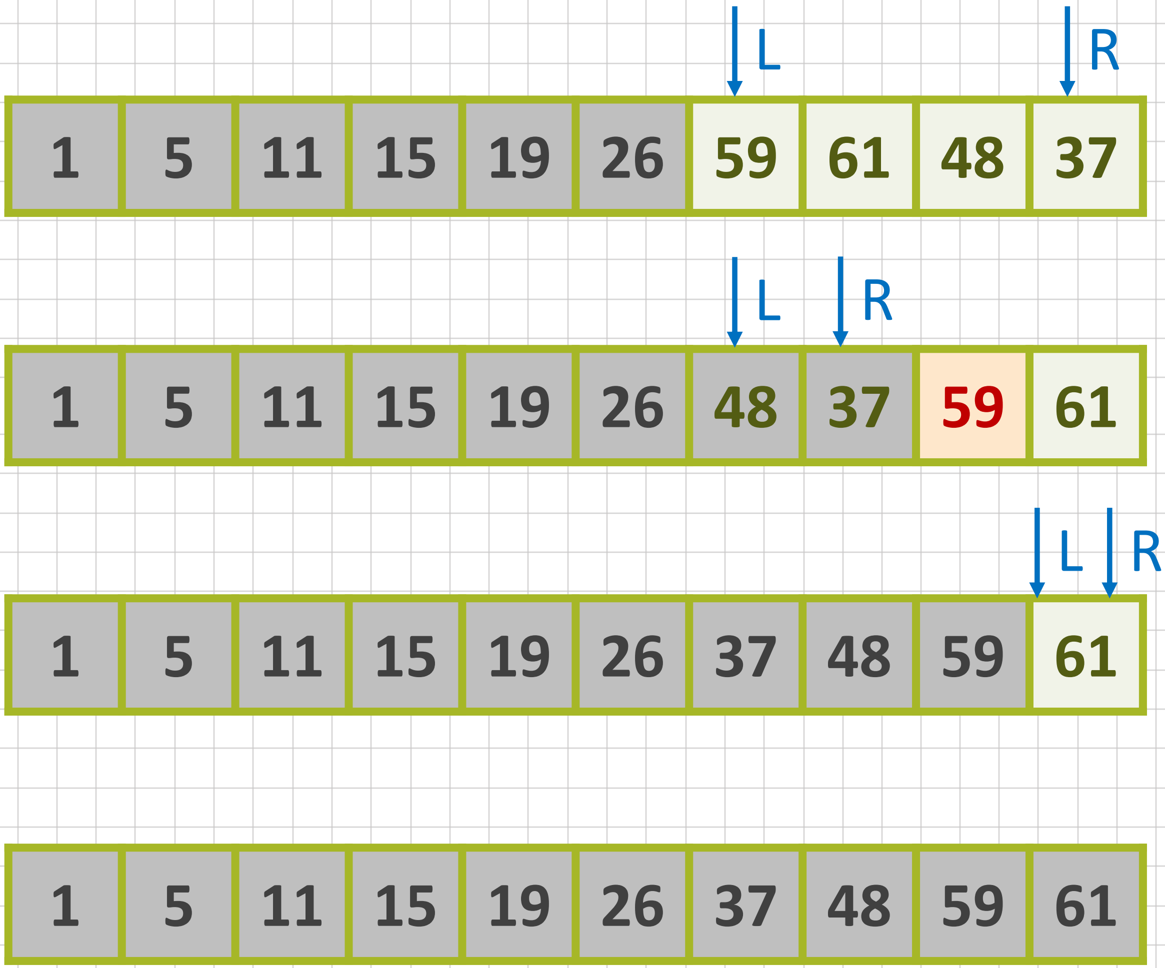
Quick Sort

Sort a[left : right] into non-decreasing order on the field

1. arbitrarily chosen a pivot key
2. Make a[left].key <= a[right + 1].key
   a. Search for keys from the left and right sublists
   b. if(elements are out of order)**swap the elements**
   c. if(the left and right boundaries cross or meet)**stop**
3. quickSort(the left sublist)
4. quickSort(the right sublist)

Let left boundary = L;   right boundary = R




L
R
1 5 11 15 19 26 59 61 48 37
L
R
1 5 11 15 19 26 48 37 59 61
L
R
1 5 11 15 19 26 37 48 59 61
1 5 11 15 19 26 37 48 59 61

Quick Sort

```
void quickSort(element list[], int
left, int right){
  int pivot, i, j;
  element temp;
  if (left < right) {
    i = left;    j = right+1;
    pivot = list[left].key;
    do {
      do i++; while (list[i].key < pivot);
      do j--; while (list[j].key > pivot);
      if (i < j)
        SWAP(list[i], list[j], temp);
    } while (i < j);
    SWAP(list[left], list[j], temp);
    quicksort(list, left, j-1);
    quicksort(list, j+1, right);
  }
}
```

Example for Quick Sort

R0	R1	R2	R3	R4	R5	R6	R7	R8	R9	left	right
{ 26	5	37	1	61	11	59	15	48	19}	0	9
{ 11	5	19	1	15}	26	{ 59	61	48	37}	0	4
{ 1	5}	11	{ 19	15}	26	{ 59	61	48	37}	0	1
1	5	11	15	19	26	{ 59	61	48	37}	3	4
1	5	11	15	19	26	{ 48	37}	59	{ 61}	6	9
1	5	11	15	19	26	37	48	59	{ 61}	6	7
1	5	11	15	19	26	37	48	59	61	9	9
1	5	11	15	19	26	37	48	59	61		

Picking the Pivot

- A wrong way: the first or the last element
- A safe maneuver: choose pivot randomly with the cost of random number generation.
- Median-of-Three Partitioning:
  - base choice: the median value (how to find?)
  - estimate:
    - pick three elements randomly and use the median.
    - use the median of the left, right, and center element.

Analysis for Quick Sort

- Assume that each time a record is positioned, the list is divided into the rough same size of two parts.
- Worst case: $O(n^2)$
- Average case and best case: $O(n \log n)$

$T(n)$ is the time taken to sort n elements

$T(n) <= cn + 2T(n/2)$ for some c

$<= cn + 2(cn/2 + 2T(n/4))$

...

Worst Case 1

Choose the biggest one as the pivot

57	5	37	1	61	11	59	15	48	19

5	37	1	61	11	59	15	48	19	57

Source: https://sites.google.com/site/mzshieh/courses/data-structures-summer-2013

Worst Case 2

Choose the smallest one as the pivot

1	5	37	50	61	11	59	15	48	19

1	5	37	50	61	11	59	15	48	19

Source: https://sites.google.com/site/mzshieh/courses/data-structures-summer-2013

Worst Case 3

Every elements are the same

1	1	1	1	1	1	1	1	1	1

1	1	1	1	1	1	1	1	1	1

Source: https://sites.google.com/site/mzshieh/courses/data-structures-summer-2013

Merge Sort

Chapter 7.5

Page 346

[Watch the video!]

Recursive sorting algorithm

Key point:

- Merge two sorted list into one

Complexity :

- O(nlogn)

Weakness:

- $\Omega(n)$ extra space

Merge Sort

- Given two sorted lists
  (list[i], …, list[m])
  (list[m+1], …, list[n])
  generate a single sorted list by merge
  (sorted[i], …, sorted[n])

Example of Merge Sort

	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
Original	11	8	14	7	6	8	23	4
after pass 1	8	11	7	14	6	8	4	23
after pass 2	7	8	11	14	4	6	8	23
after pass 3	4	6	7	8	8	11	14	23

Example of Merge

	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
after pass 2	7	8	11	14	4	6	8	23
p=0 q=4	4							
p=0 q=5	4	6						
p=0 q=6	4	6	7					
p=1 q=6	4	6	7	8				
p=2 q=6	4	6	7	8	8			
p=2 q=7	4	6	7	8	8	11		
p=3 q=7	4	6	7	8	8	11	14	
	4	6	7	8	8	11	14	23

Merge function

merge list[i..m] and list[m+1…n]

```
void merge(element list[], element sorted[],
                int i, int m, int n)
{
   int j, k, t;
   j = m+1;
   k = i;
   while (i<=m && j<=n) {
      if (list[i].key<=list[j].key)
         sorted[k++]= list[i++];
      else sorted[k++]= list[j++];
   }
   if (i>m) for (t=j; t<=n; t++)
      sorted[k+t-j]= list[t];
   else for (t=i; t<=m; t++)
      sorted[k+t-i] = list[t];
}
```

additional space: n-i+1

# of data movements: M(n-i+1)

Iterative Merge Sort

Sort 26, 5, 77, 1, 61, 11, 59, 15, 48, 19

	[26] [5] [77] [1] [61] [11] [59] [15] [48] [19]				
Pass 1	[5 26]	[1 77]	[11 61]	[15 59]	[19 48]
Pass 2	[1 5 26 77]		[11 15 59 61]		[19 48]
Pass 3	[1 5 11 15 26 59 61 77]				[19 48]
Pass 4	[1 5 11 15 19 26 48 59 61 77]				

Merge_Sort

```
void merge_sort(element list[], int n)
{
    int s=1;
    element sorted[MAX_SIZE];

    while (s<n) {
        merge_pass(list, sorted, n, s);
        s*= 2;
        merge_pass(sorted, list, n, s);
        s*= 2;
    }
}
```

Merge_Pass

```
void merge_pass(element list[], element
                 sorted[],int n, int s)
{
  int i, j;
  for (i=1; i<=n-2*s+1; i+=2*s)
    merge(list,sorted,i,i+s-1, i+2*s-1);
```

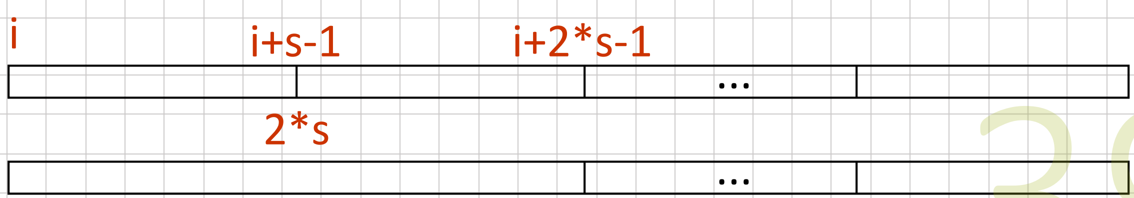
length

One complement segment and one partial segment
e.g., merge(list, sorted, 1, 8,10)

```
  if (i+s-1<n)
    merge(list, sorted, i, i+s-1, n);
  else
```

Only one segment

```
    for (j=i; j<=n; j++) sorted[j]=
  list[j];
}
```



Analysis of Merge Sort

Complexity: O(n log n)

Total passes: log n

For each merge: O(n)

Therefore, the complexity is O(n log n)

Heap Sort

Chapter 736

Page 352

Max-heap structure

Worst case:

O(n log n)

Average case:

O(n log n)

Slightly Slower than merge sort

Extra space needed:

only O(1) (constant)

Heap Sort

- Structure property: complete binary tree
  - complete binary tree:

    Maximum tree-depth: $\lceil log_2(n+1) \rceil$
  - array implementation of heap
- Order property: for each node X, the key in the parent of X is smaller than or equal to the key in X.

Heap Sort

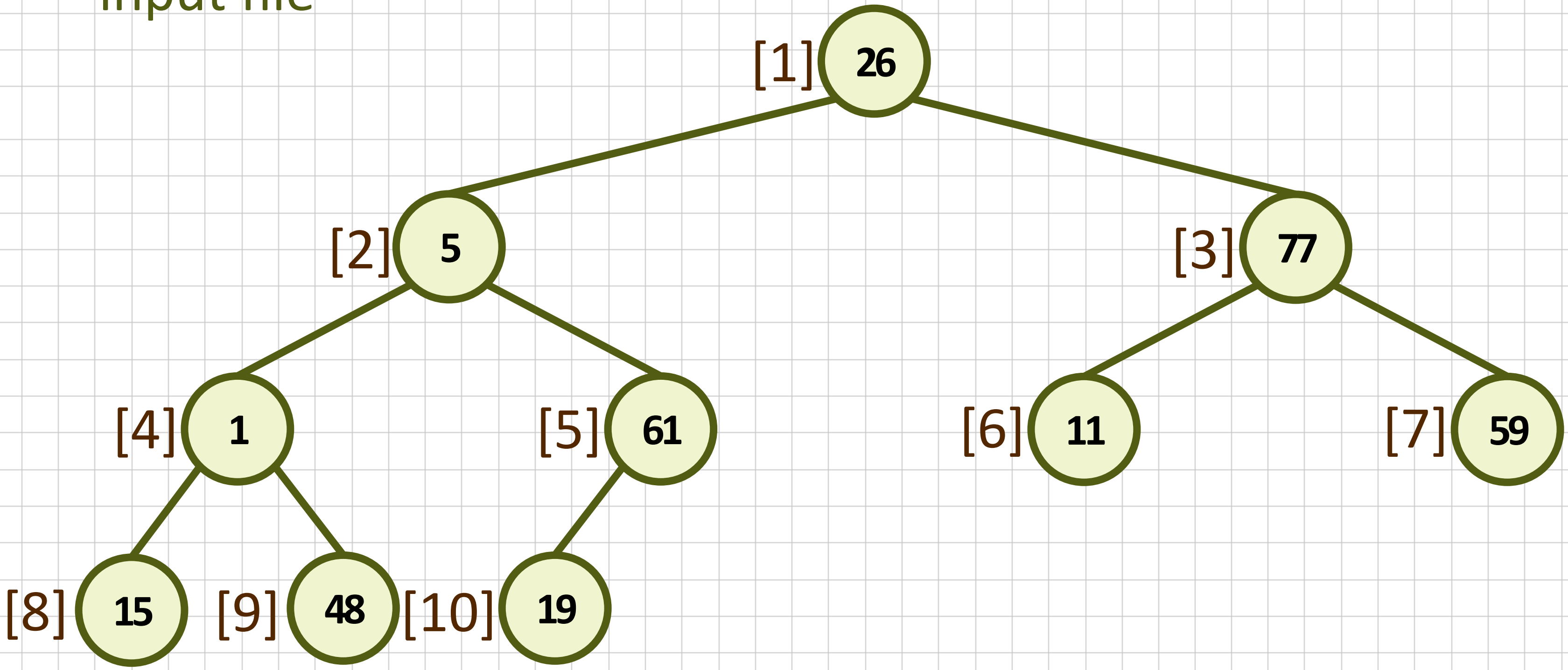
Algorithm (Increasing order)

1. Build heap (Min-heap)
2. for (i=0; i <N; i++)
3. DeleteMin;

Heap Sort

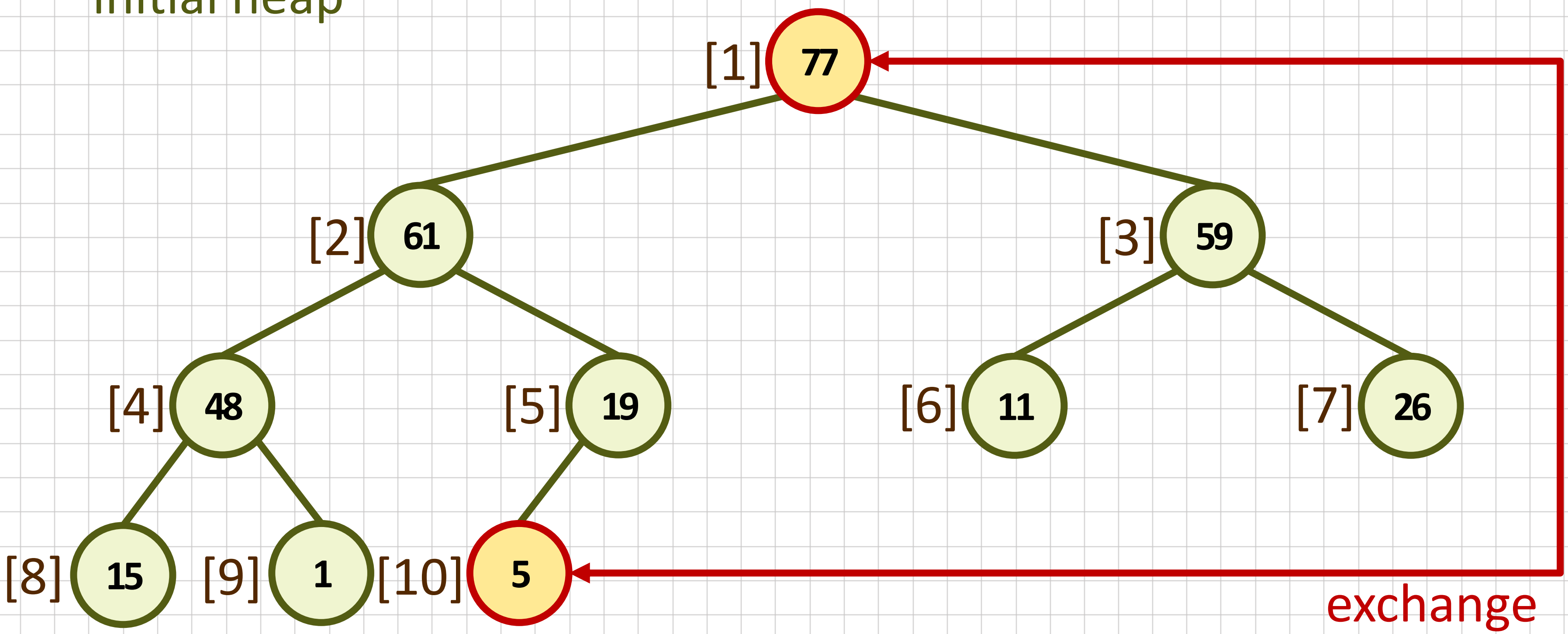
26, 5, 77, 1, 61, 11, 59, 15, 48, 19

input file

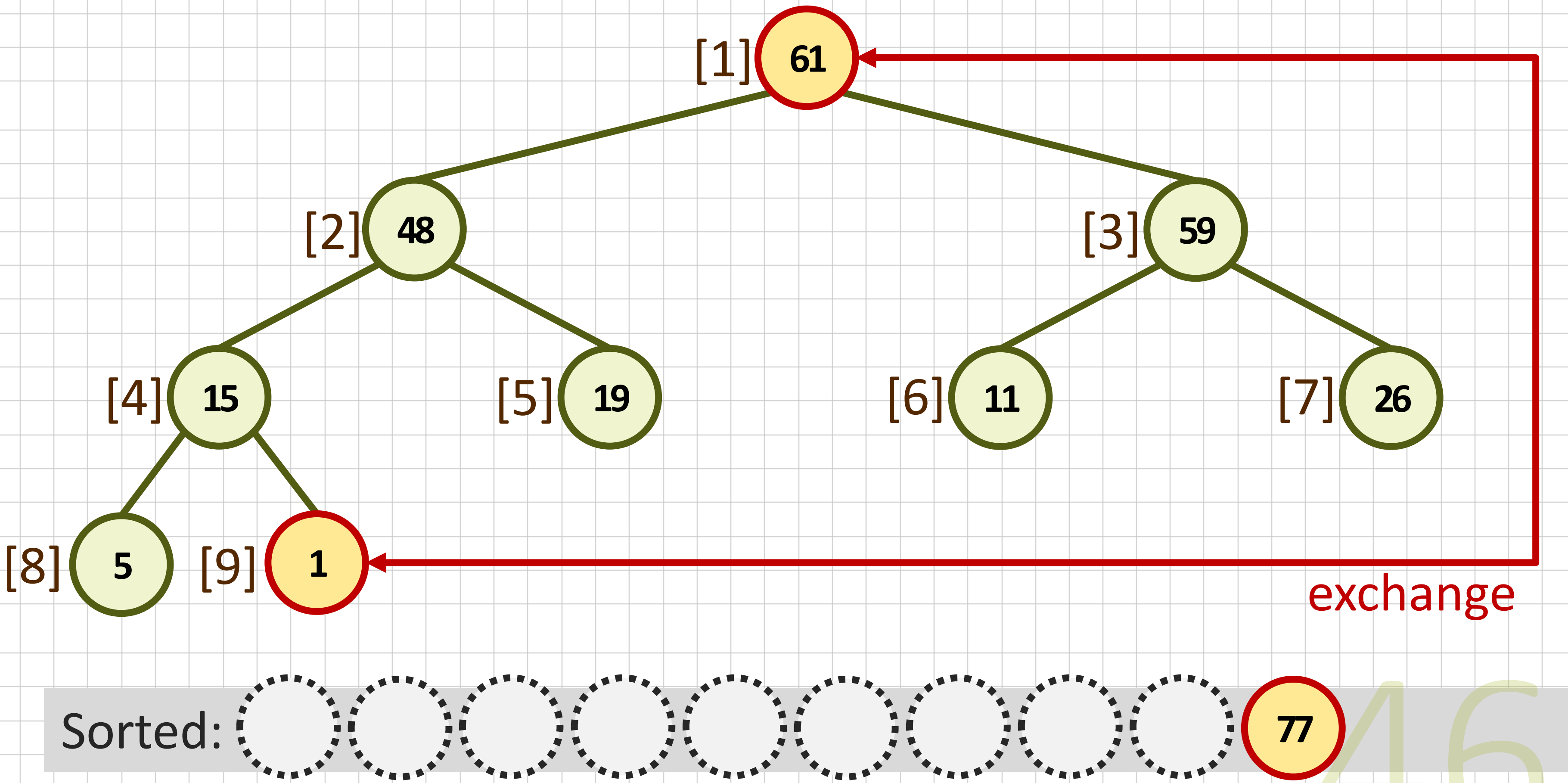


Heap Sort

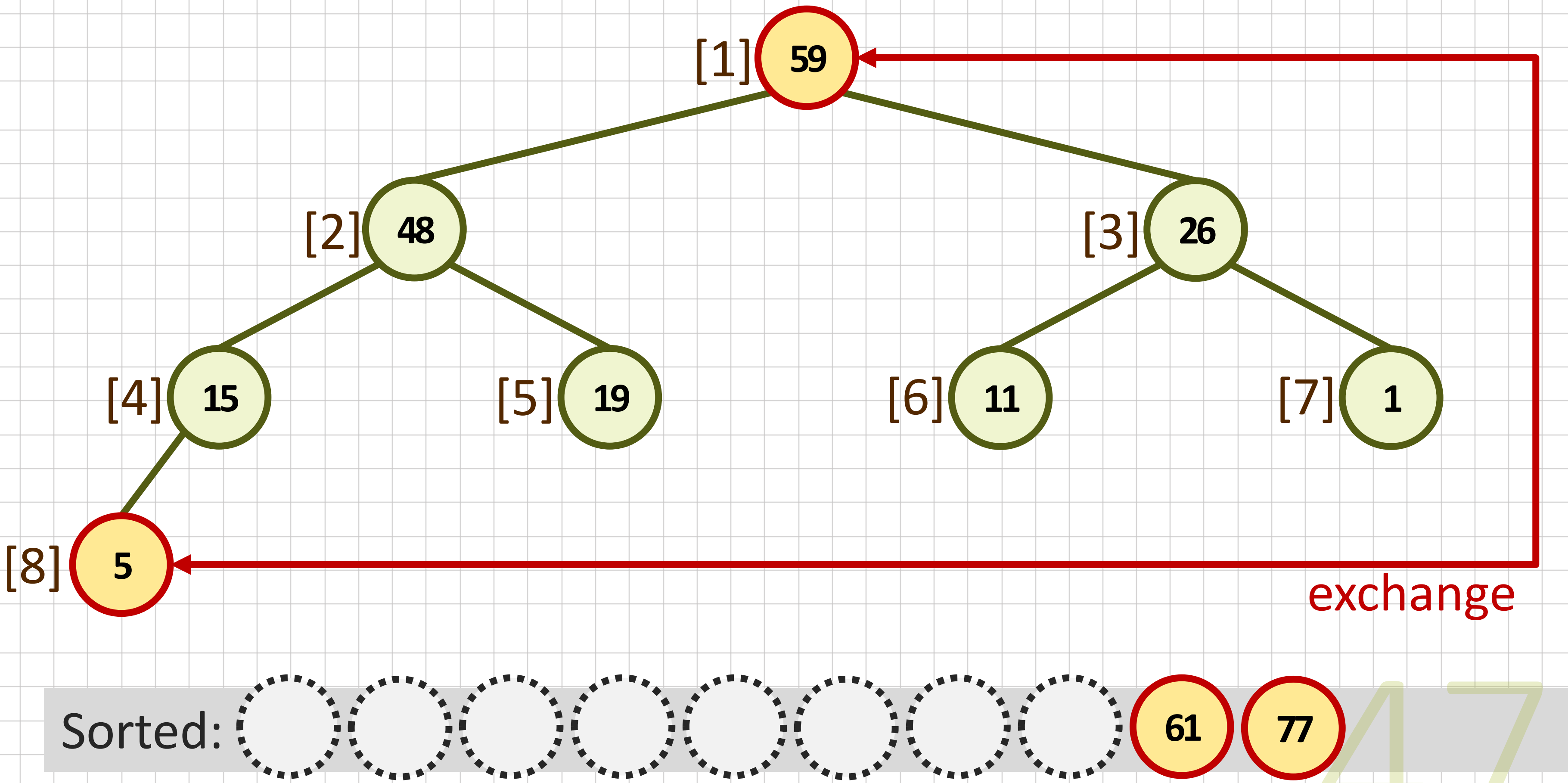
initial heap



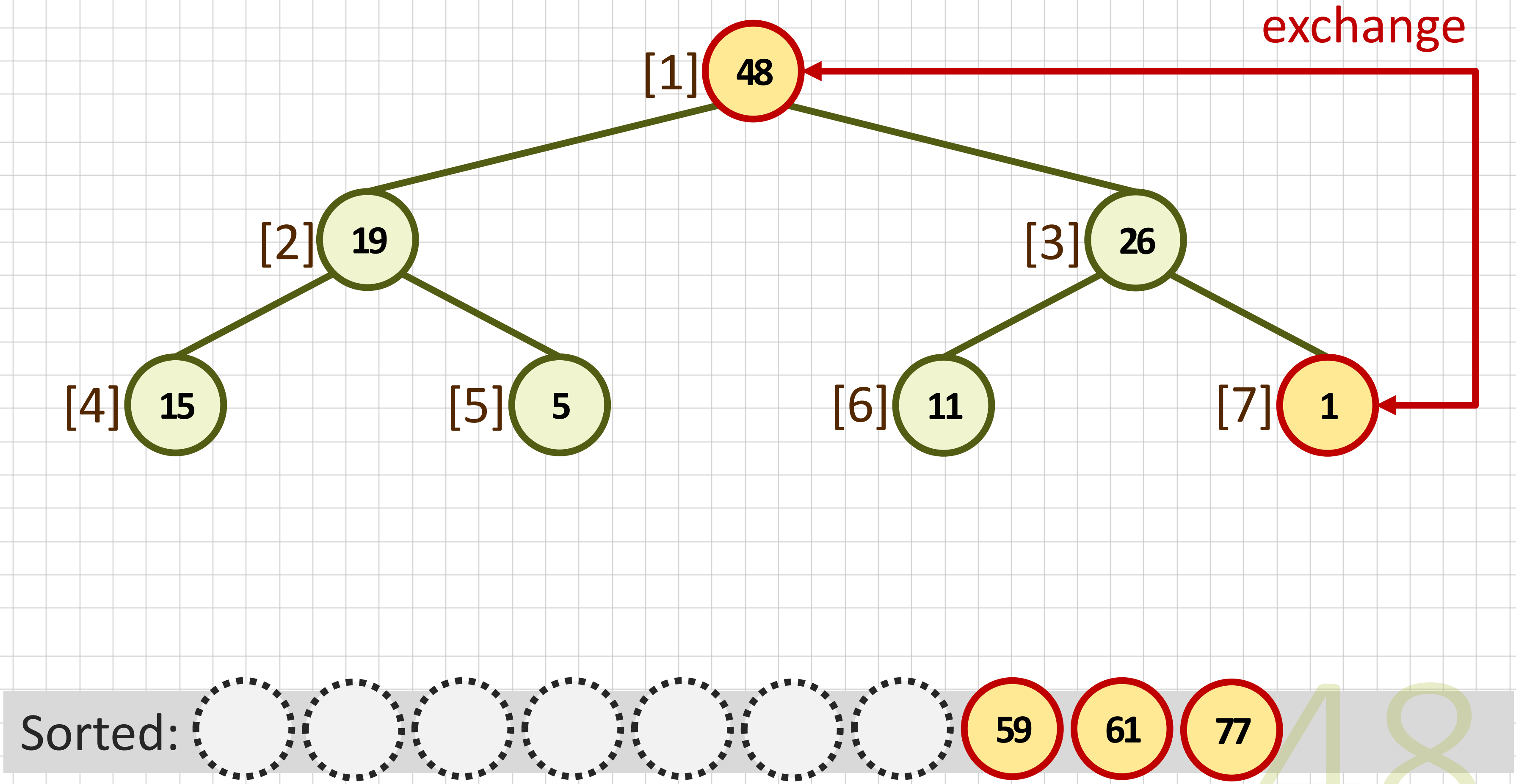
Heap Sort



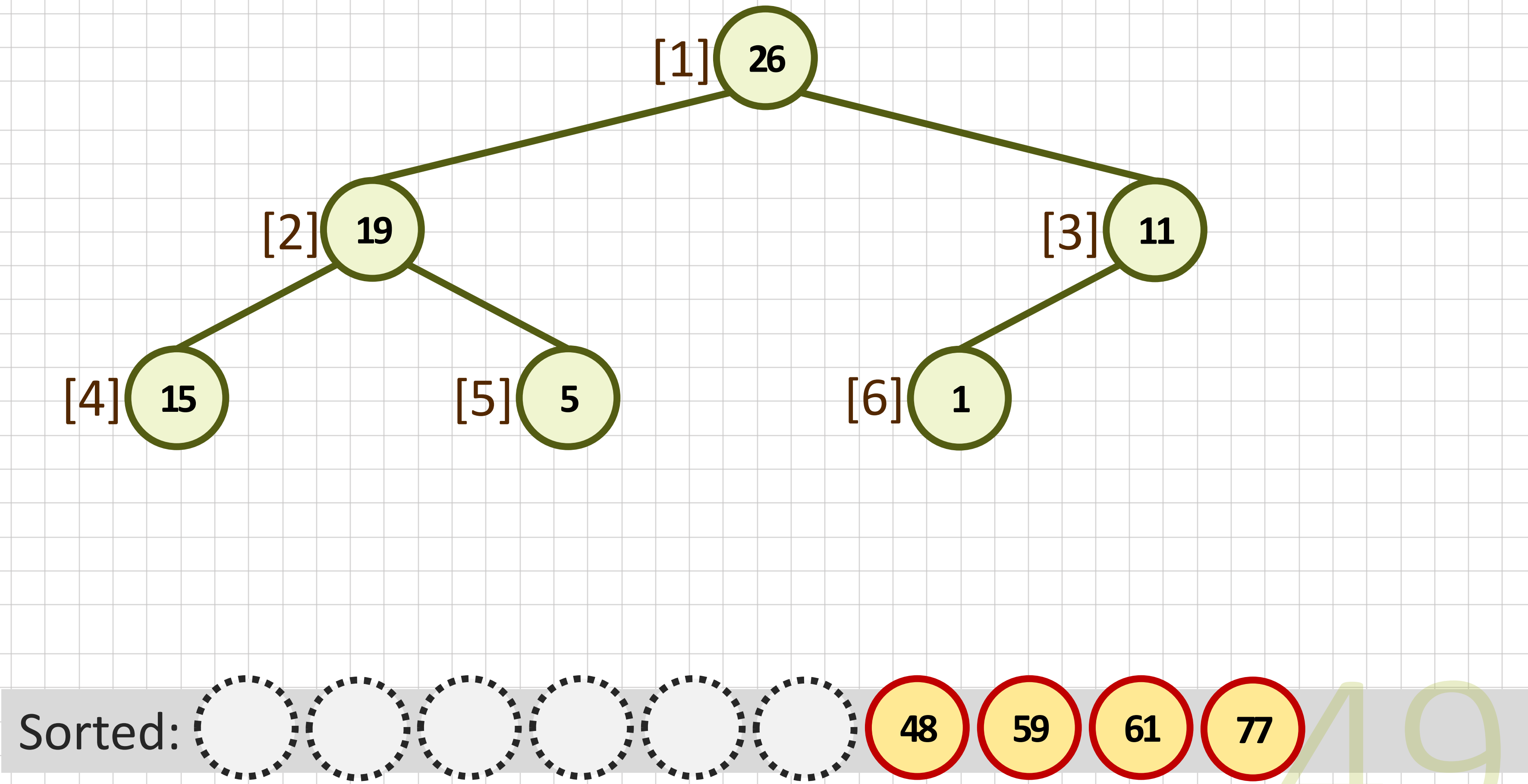
Heap Sort



Heap Sort



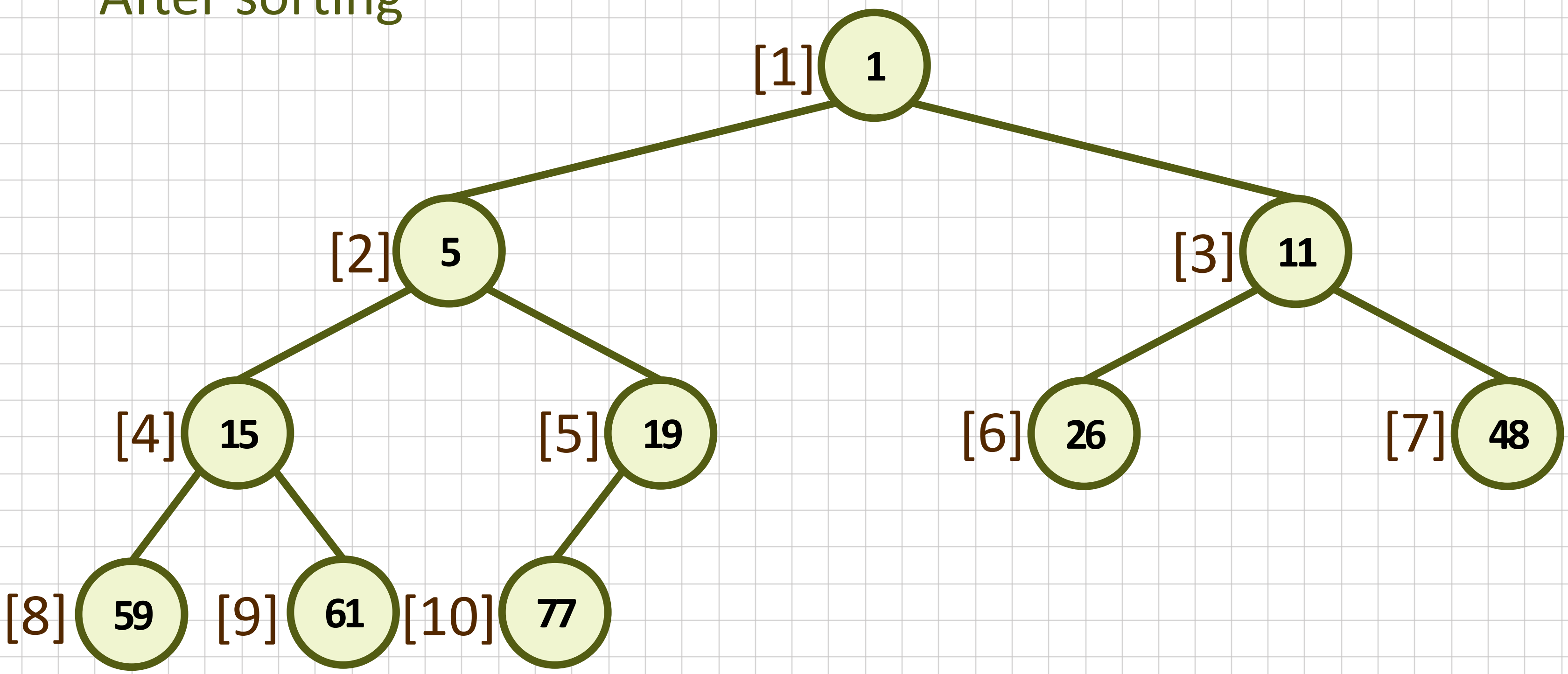
Heap Sort



Heap Sort

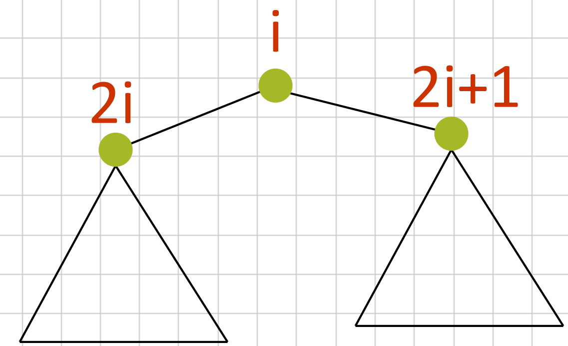
26, 5, 77, 1, 61, 11, 59, 15, 48, 19

After sorting



Sorted: 1 5 11 15 19 26 48 59 61 77

Heap Sort




```
void adjust(element list[], int root, int n)
{
  int child, rootkey;
  element temp;
  temp=list[root];
  rootkey=list[root].key;
  child=2*root;
  while (child <= n) {
    if ((child < n) &&
       (list[child].key < list[child+1].key))
          child++;
    if (rootkey > list[child].key)
      break;
    else {
       list[child/2] = list[child];
       child *= 2;
    }
  }
  list[child/2] = temp;
}
```

Heap Sort

```
void heapsort(element list[], int n)    ascending order (max heap)
{
    int i, j;
    element temp;
    for (i=n/2; i>0; i--) adjust(list, i, n);  bottom-up
    for (i=n-1; i>0; i--) {  n-1 cylces
        SWAP(list[1], list[i+1], temp);
        adjust(list, 1, i);  top-down
    }
}
```

Complexity of Heap Operations

Insertion: precolate up, O(logn)

DeleteMin: precolate down, O(logn)

Heap Sort: O(nlogn)

Bubble Sort

補充

Complexity: $O(n^2)$

- For N elements A[0], …,A[N-1], Bubble sort consists of **(N-1) passes** (Pass 0 through N-2).
- In pass P, adjacent elements in A[P], ..,A[N-1] are compared & exchanging if necessary.
- After pass P, the first P elements have been placed in the correct position.

Example of Bubble Sort

	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
Original	**2**	**4**	**7**	**1**	**5**	**3**
after pass 0	**2**	**4**	**1**	**5**	**3**	**7**
after pass 1	**2**	**1**	**4**	**3**	**5**	**7**
after pass 2	**1**	**2**	**3**	**4**	**5**	**7**
afetr pass 3	**1**	**2**	**3**	**4**	**5**	**7**
after pass 4	**1**	**2**	**3**	**4**	**5**	**7**

Algorithm of Bubble Sort

```
BubbleSort( int a[], int n)
Begin
    for i = 1 to n-1
        sorted = true
        for j = 0 to n-1-i
            if a[j] > a[j+1]
                temp = a[j]
                a[j] = a[j+1]
                a[j+1] = temp
                sorted = false
        end for
        if sorted
            break from i loop
    end for
End
```

Bucket Sort

Chapter 7.7

Page 356

- Bucket sort
  - ✓ allocate sufficient number of buckets &
    - ➢ put element in corresponding buckets
    - ➢ at the end, scan the buckets in order & collect all elements
- n elements, ranges from 1 to m → m buckets

Radix Sort

- Drawback of bucket sort: waste buckets (space)
- Radix sort
  - use several passes of bucket sort
  - more than one number could fall into the <u>same bucket</u>
- Two approaches
  - most significant bit (MSB): radix-exchange sort
  - least significant bit (LSB): straight-radix sort

Radix Sort

Sort by keys

K^0, K^1, ..., K^{r-1}

Most significant key

Least significant key

R_0, R_1, ..., R_{n-1} are said to be sorted w.r.t. K_0, K_1, ..., K_{r-1} iff

$$(k_i^0, k_i^1, \ldots, k_i^{r-1}) <= (k_{i+1}^0, k_{i+1}^1, \ldots, k_{i+1}^{r-1}) \quad 0 \le i < n-1$$

Most significant digit first: sort on K^0, then K^1, ...

Least significant digit first: sort on K^{r-1}, then K^{r-2}, ...

Radix-Exchange Sort (MSB)

- Given n elements represented by k-digits
  - hypothesis: we know how to sort elements with left k digits
  - induction use bucket sort

Radix-Exchange Sort (MSB)

Given

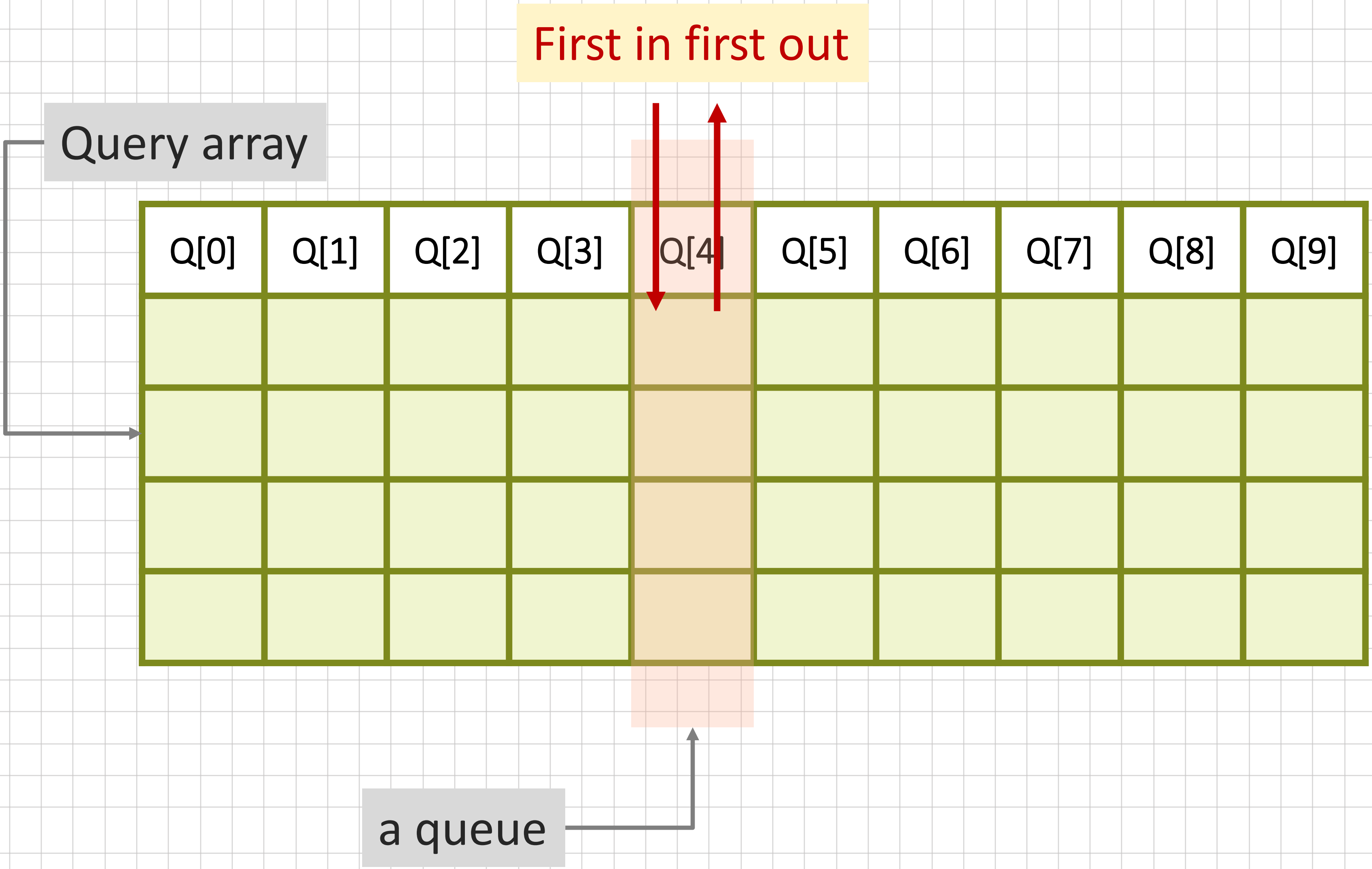
(064, 008, 216, 512, 027, 729, 000, 001, 343, 125)

0	1	2	3	4	5	6	7	8	9
001	125	216	343		512		728		
000									
027									
008									
064									

Consider the most significant digit first

Straight-Radix Sort (LSB)

- Given n elements represented by k-digits
  - Hypothesis: sort elements with < digits
  - Induction
    - ignore the most significant bit & sort the n elements according to their **k-1 least significant bits**
    - scan all the elements & use bucket sort on the most significant bit
    - collect all the buckets in order


First in first out
Query array
Q[0]
Q[1]
Q[2]
Q[3]
Q[4]
Q[5]
Q[6]
Q[7]
Q[8]
Q[9]
a queue

LSD

179 208 306 93 859 984 55 9 271 33

Q[0]	Q[1]	Q[2]	Q[3]	Q[4]	Q[5]	Q[6]	Q[7]	Q[8]	Q[9]

Input sequence: 179,208,306,93,859,984,55,9,271,33

LSD

179 208 306 93 859 984 55 9 271 33

Q[0]	Q[1]	Q[2]	Q[3]	Q[4]	Q[5]	Q[6]	Q[7]	Q[8]	Q[9]
	271		093	984	055	306		208	179
			033						859
									009

Input sequence: 179,208,306,93,859,984,55,9,271,33

LSD

Q[0]	Q[1]	Q[2]	Q[3]	Q[4]	Q[5]	Q[6]	Q[7]	Q[8]	Q[9]
	271		093	984	055	306	208		179
			033						859
									009

Output sequence: 271,93,33,984,55,306,208,179,859,9

Input sequence: 271,93,33,984,55,306,208,179,859,9

Q[0]	Q[1]	Q[2]	Q[3]	Q[4]	Q[5]	Q[6]	Q[7]	Q[8]	Q[9]
306			033		055		271	984	093
208					859		179		
009									

Output sequence: 306,208,09,33,55,859,271,179,984,93

Input sequence: 306,208,9,33,55,859,271,179,984,93

Q[0]	Q[1]	Q[2]	Q[3]	Q[4]	Q[5]	Q[6]	Q[7]	Q[8]	Q[9]
009	179	208	306					859	984
033		271							
055									
093									

Output sequence: 9,33,55,93,179,208,271,306,859,984

Another Example

Given (64, 8, 216, 512, 27, 729, 0, 1, 343, 125)

0	1	2	3	4	5	6	7	8	9
0	1	512	343	64	125	216	27	8	729

⇒ (0, 1, 512, 343, 64, 125, 216, 27, 8,729)

0	1	2	3	4	5	6	7	8	9
8 1 0	216 512	729 27 125		343		64			

⇒ (0, 1, 8, 512, 216, 125, 27, 729, 343, 64)

Another Example (Cont.)

⇒ (0, 1, 8, 512, 216, 125, 27, 729, 343, 64)

64	**125**	**216**	**343**		**512**		**729**		
27									
8									
1									
0									
0	1	2	3	4	5	6	7	8	9

⇒ (0, 1, 8, 27, 64, 125, 216, 343, 512, 729)

External Sort

Chapter 7.10

Page 376

The lists are so large that an entire list cannot be contained in the internal memory.

External Sorting

Very large files

(overheads in disk access)

- seek time

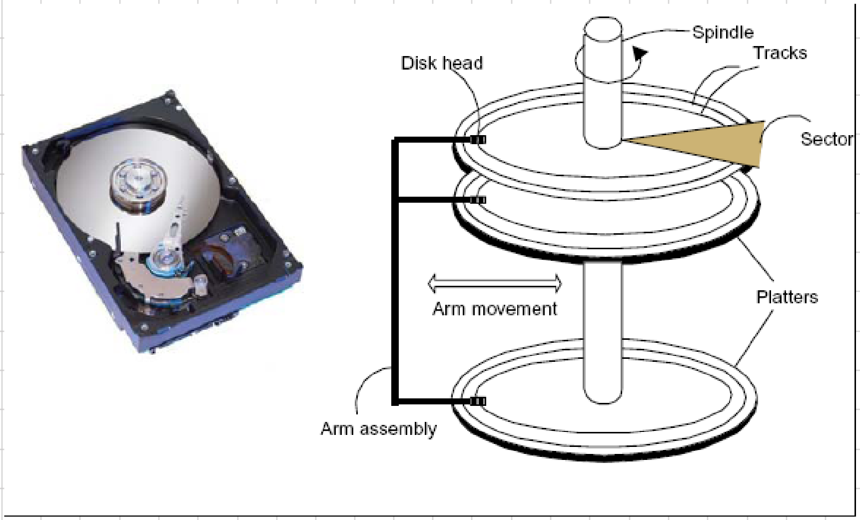
  measure of how long the disk heads take to arrive a particular track.

- latency time

  the time it takes for the block to rotate under the head.

- transmission time

  read and write data in the block once the head is positioned.



External Sorting

- merge sort
  - phase 1

    Segment the input file & sort the segments (runs)
  - phase 2

    Merge the runs

File: 4500 records, A1, …, A4500

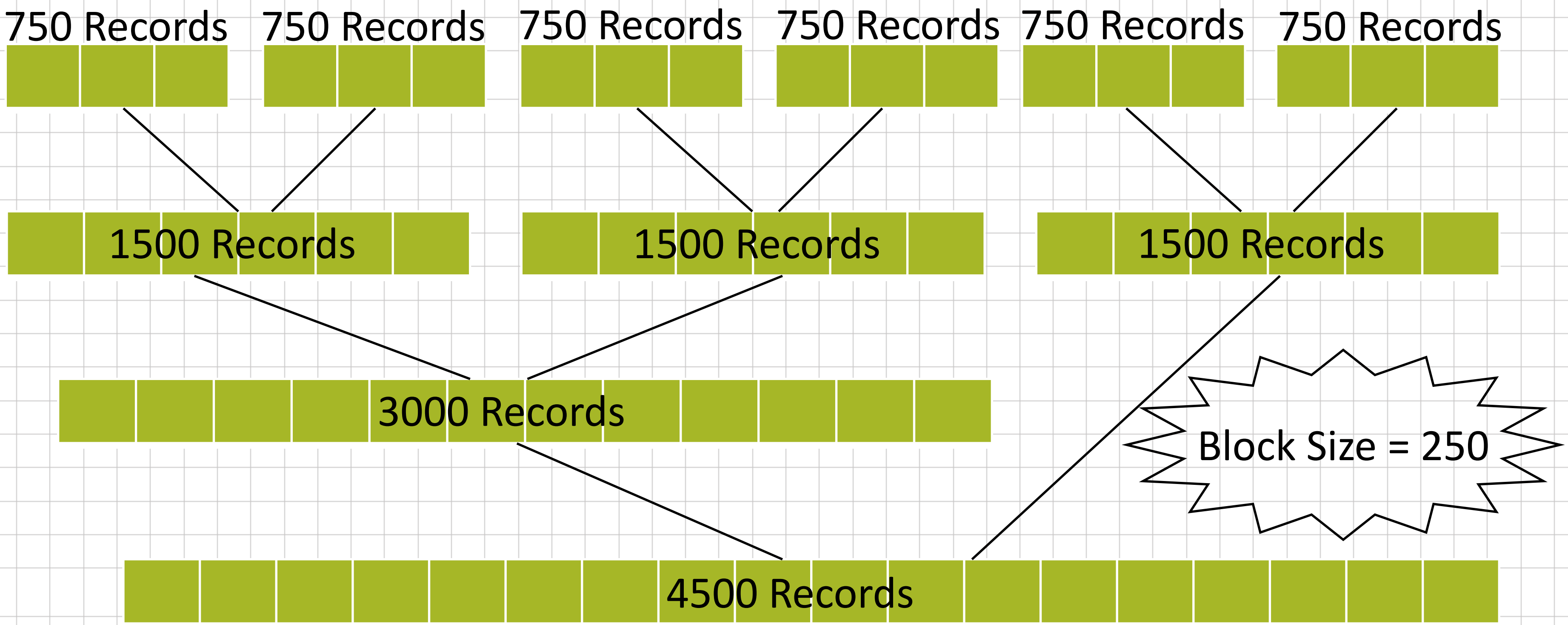
internal memory: 750 records (3 blocks)

block length: 250 records

input disk vs. scratch pad (disk)

(1) sort three blocks at a time and write them out onto **scratch pad**

(2) three blocks: two input buffers & one output buffer



Time Complexity of External Sort

- input/output time

t_s = maximum seek time

t_l = maximum latency time

t_{rw} = time to read/write one block of 250 records

$t_{IO} = t_s + t_l + t_{rw}$

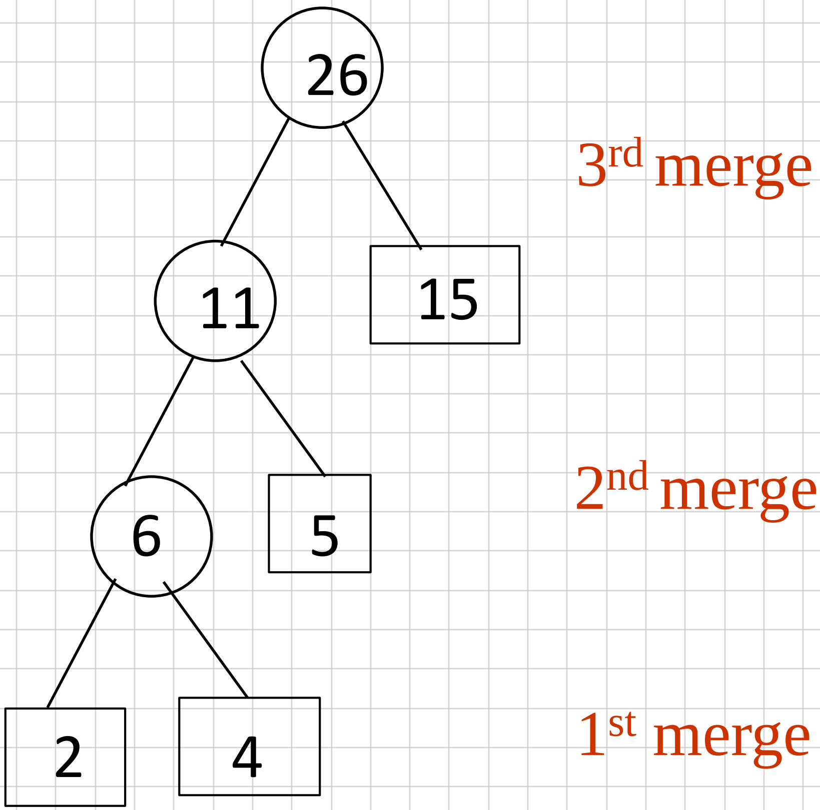
- cpu processing time

t_{IS} = time to internally sort 750 records

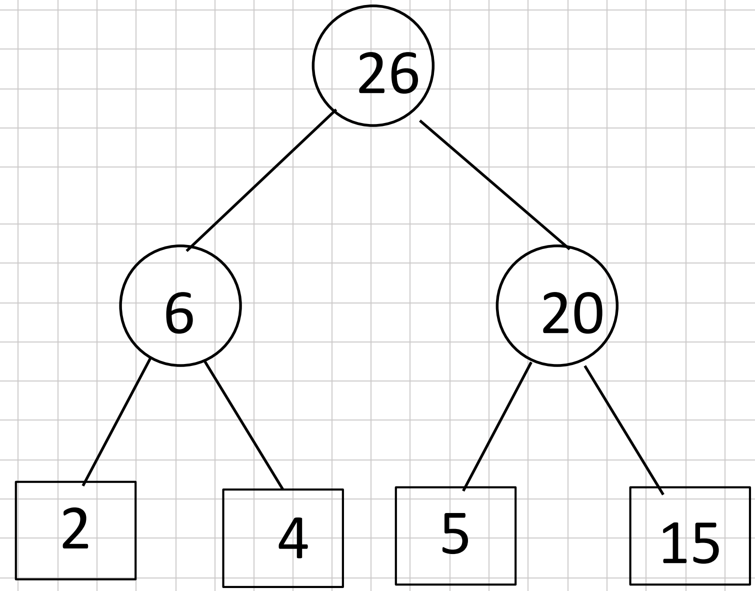
nt_m = time to merge *n* records from input buffers to the output buffer

Optimal Merging of Runs

When runs are of difference size, it is more important to determine the run merge strategy



weighted external path length
= 2*3 + 4*3 + 5*2 + 15*1 = 43

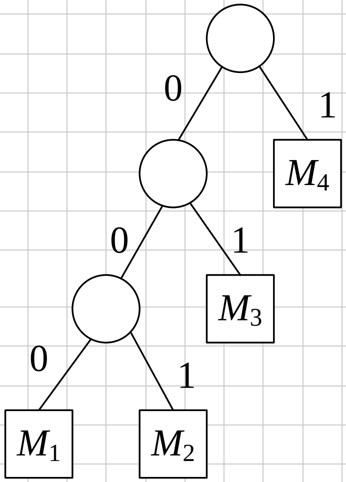


weighted external path length
= 2*2 + 4*2 + 5*2 + 15*2 = 52

Huffman Code

- Assume we want to obtain an optimal set of codes for messages M1, M2, …, Mn+1. Each code is a binary string that will be used for transmission of the corresponding message.
- At receiving end, a decode tree is used to decode the binary string and get back the message.
- A zero is interpreted as a left branch and a one as a right branch.
- If q_i is the relative frequency with which message Mi will be transmitted, the expected decoding time is

$$\sum q_i d_i$$



M1: 000
M2: 001
M3: 01
M4: 1