



DATA STRUCTURE AND ALGORITHMS

Chapter 7: Sorting



MOTIVATION OF SORTING

- The term list here is a collection of records.
- Each record has one or more fields.
- Each record has a key to distinguish one record with another.
 - For example, the phone directory is a list. Name, phone number, and even address can be the key, depending on the application or need.

SORTING

- Two ways to store a collection of records
 - Sequential
 - Non-sequential
- Assume a sequential list f . To retrieve a record with key $f[i].key$ from such a list, we can do search in the following order:
 - $f[n].key, f[n-1].key, \dots, f[1].key \Rightarrow$ sequential search

EXAMPLE OF AN ELEMENT OF A SEARCH LIST

* (P.320)

```
# define MAX-SIZE 1000/* maximum size of list plus one */
typedef struct {
    int key;
    /* other fields */
} element;
element list[MAX_SIZE];
```

SEQUENTIAL SEARCH

***Program 7.1:** Sequential search (p.321)

```
int seqsearch( int list[ ], int searchnum, int n )
{
/*search an array, list, that has n numbers. Return i, if list[i]=searchnum.
Return -1, if searchnum is not in the list */
    int i;
    list[n]=searchnum; sentinel
    for (i=0; list[i] != searchnum; i++)
        ;
    return (( i<n) ? i : -1);
}
```

SEQUENTIAL SEARCH (CONT'D)

- The number of comparisons for a record key l is $n-i+1$.
- The average number of comparisons for a successful search is

$$\sum_{1 \leq i \leq n} \frac{n-i+1}{n} = \frac{n+1}{2}$$

- There is a better way than this.
 - Binary search



BINARY SEARCH

- A binary search only takes $O(\log n)$ time to search a sequential list with n records.
- An interpolation scheme relies on an ordered list.



SORTING APPLICATION

- So far we have seen two uses of sorting
 - An aid in **searching**
 - A means for **matching** entries in lists
- Sorting is used in other applications.
 - Estimated **25% of all computing time** is spent on sorting
- **No one sorting method is the best** for all initial orderings of the list being sorted.

FORMAL DESCRIPTION OF SORTING

- Given a list of records (R_1, R_2, \dots, R_n) , each record R_i has a key value K_i ,
- The sorting problem is finding a permutation, σ , such that $K_{\sigma(i)} \leq K_{\sigma(i+1)}$.
- When the list has several key values that are identical, the permutation, σ , is not unique.

FORMAL DESCRIPTION OF SORTING (CONT'D)

- That permutation, σ_s , be the permutation with the following properties:
 - $K_{\sigma(i)} \leq K_{\sigma(i+1)}, i \leq n - 1$
 - If $i < j$ and $K_i == K_j$ in the input list, then R_i precedes R_j in the sorted list
- A sorting method that generates the permutation σ_s is *stable*



FORMAL DESCRIPTION OF SORTING (CONT'D)

- A sorting algorithm is said to be *in-place* if it requires very little additional space besides the initial array holding the elements that are to be sorted.
- Normally “very little” is taken to mean that for sorting n elements, extra space $O(\log n)$ is required.

CATEGORY OF SORTING METHOD

- **Internal Method:** Methods to be used when the list to be sorted is small enough so that the entire sort list can be carried out in the main memory.
 - Insertion sort, quick sort, merge sort, heap sort and radix sort.
- **External Method:** Methods to be used on larger lists.
 - Only a portion of data will be loaded into main memory

INSERTION SORT

```
void insertion_sort(element list[], int n)
{
    int i, j;
    element next;
    for (i=1; i<n; i++) {
        next= list[i];
        for (j=i-1; j>=0&&next.key<list[j].key;j--)
            list[j+1] = list[j];
        list[j+1] = next;
    }
}
```

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

INSERTION SORT EXAMPLE #1

- Record R_i is left out of order (LOO) iff
 - $R_i < \max_{1 \leq j < i} \{R_j\}$
- Example 7.1: Assume $n = 5$ and the input key sequence is 5, 4, 3, 2, 1

j	[1]	[2]	[3]	[4]	[5]
-	5	4	3	2	1
2	4	5	3	2	1
3	3	4	5	2	1
4	2	3	4	5	1
5	1	2	3	4	5

INSERTION SORT EXAMPLE #2


- Example 7.2: Assume $n = 5$ and the input key sequence is 2, 3, 4, 5, 1

j	[1]	[2]	[3]	[4]	[5]	
-	2	3	4	5	1	
2	2	3	4	5	1	$O(1)$
3	2	3	4	5	1	$O(1)$
4	2	3	4	5	1	$O(1)$
5	1	2	3	4	5	$O(n)$



INSERTION SORT VARIATIONS

- Binary insertion sort:
 - Using **binary search** to **reduce** the number of comparisons in an insertion sort.
 - The number of records moves remains the same.
- List insertion sort:
 - The number of record moves becomes zero because only the **link** fields require adjustment.



OTHER $O(N^2)$ SORTING ALGORITHMS

- Selection sort
 - Chapter 1
- Bubble sort

SELECTION SORT

- Given the list of records with key (26, 5, 47, 19, 6), using selection sort to sort the list in increasing order.
 - Pass1: 5, 26, 47, 19, 6
 - Pass2: 5, 6, 47, 19, 26
 - Pass3: 5, 6, 19, 47, 26
 - Pass4: 5, 6, 19, 26, 47

BUBBLE SORT

```
void bubble_sort(T arr[], int len)
{
    int i, j;
    for (i = 0; i < len - 1; i++)
        for (j = 0; j < len - 1 - i; j++)
            if (arr[j] > arr[j + 1])
                swap(arr[j], arr[j + 1]);
}
```

BUBBLE SORT (CONT'D)

- Given the list of records with key (26, 5, 47, 19, 6), using selection sort to sort the list in increasing order.
 - Pass1: 5, 26, 19, 6, **47**
 - Pass2: 5, 19, 6, **26**, 47
 - Pass3: 5, 6, **19**, 26, 47
 - Pass4: 5, **6**, 19, 26, 47

QUICK SORT

- Quick sort is developed by C. A. R. Hoare. The quick sort scheme has **the best average behavior** among the sorting methods.
- Quick sort differs from insertion sort in that the **pivot** key K_i is placed at the correct spot with respect to the whole list. $K_j \leq K_{s(i)}$ for $j < s(i)$ and $K_j \geq K_{s(i)}$ for $j > s(i)$.
- Therefore, the sublist to the left of $s(i)$ and to the right of $s(i)$ can be sorted independently.

QUICK SORT (CONT'D)

```
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);
    }
}
```

QUICK SORT EXAMPLE

- Example 7.3: The input list has 10 records with keys (26, 5, 37, 1, 61, 11, 59, 15, 48, 19).

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		

QUICK SORT (CONT'D)

- In QuickSort(), list[n+1] has been set to have a key at least as large as the remaining keys.
- Analysis of QuickSort
 - The worst case $O(n^2)$
 - If each time a record is correctly positioned, the sublist of its left is of the same size of the sublist of its right. Assume $T(n)$ is the time taken to sort a list of size n :

$$T(n) \leq cn + 2T\left(\frac{n}{2}\right), \text{ for some constant } c$$

$$\leq cn + 2\left(\frac{cn}{2} + 2T\left(\frac{n}{4}\right)\right)$$

$$\leq 2cn + 4T\left(\frac{n}{4}\right)$$

$$\leq cn \cdot \log_2 n + T(1)$$

$$= O(n \log n)$$



LEMMA 7.1

- Lemma 7.1: Let $T_{avg}(n)$ be the expected time for function QuickSort to sort a list with n records. Then there exists a constant k such that $T_{avg}(n) \leq kn \cdot \log_e n$ for $n \geq 2$.



ANALYSIS OF QUICK SORT

- Unlike insertion sort (which only needs additional space for a record), quick sort needs stack space to implement the recursion.
- Space complexity:
 - Average case and best case: $O(\log n)$
 - Worst case: $O(n)$
- Time complexity:
 - Average case and best case: $O(n \log n)$
 - Worst case: $O(n^2)$

QUICK SORT VARIATIONS

- Quick sort using a median of three: Pick the median of the first, middle, and last keys in the current sublist as the pivot.
 - $pivot = median\{K_l, K_{(l+r)/2}, K_r\}$.
- Quick sort is unstable

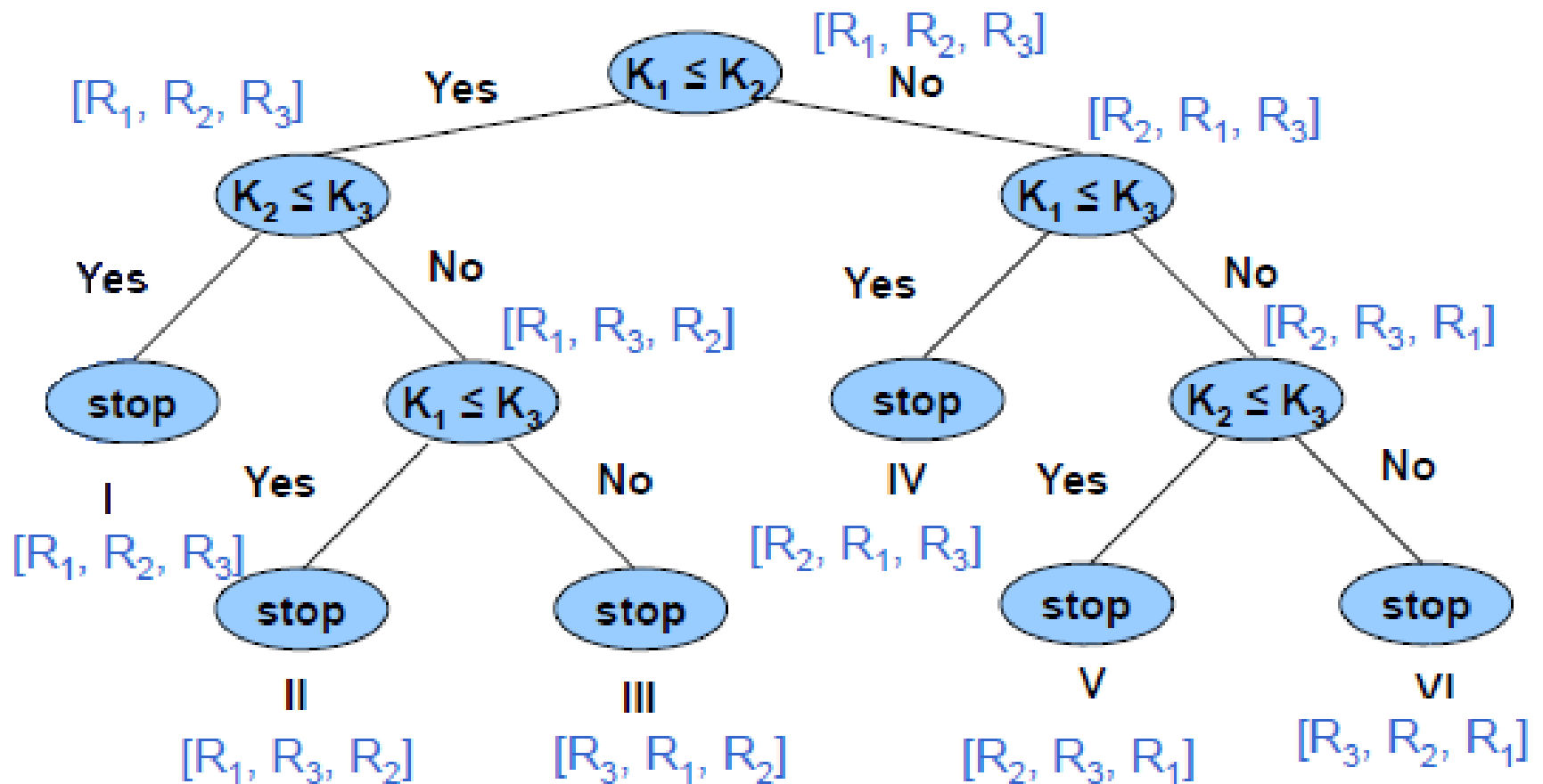


DECISION TREE

- So far both insertion sorting and quick sorting have worst-case complexity of $O(n^2)$.
- If we restrict the question to sorting algorithms in which the only operations permitted on keys are comparisons and interchanges, then $O(n \log n)$ is the best possible time.
- This is done by using a tree that describes the sorting process. Each vertex of the tree represents a key comparison, and the branches indicate the result. Such a tree is called **decision tree**.

DECISION TREE FOR INSERTION SORT

Apply insertion sort on R_1, R_2, R_3



DECISION TREE (CONT'D)

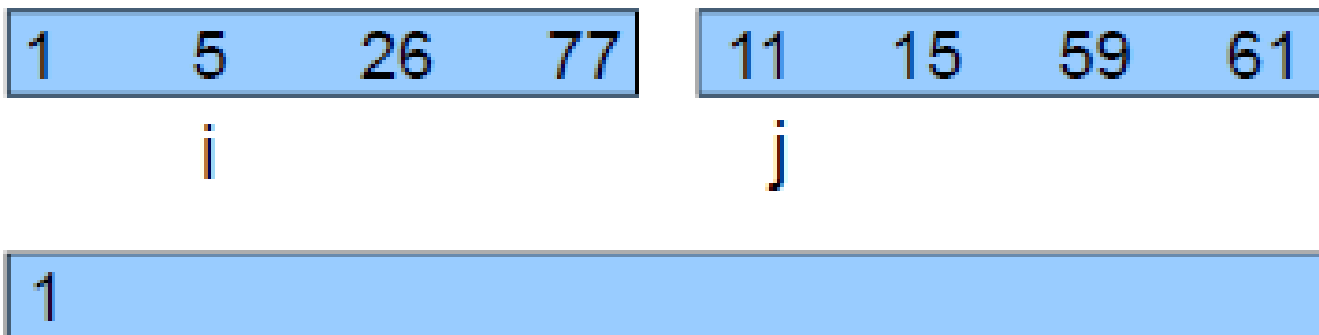
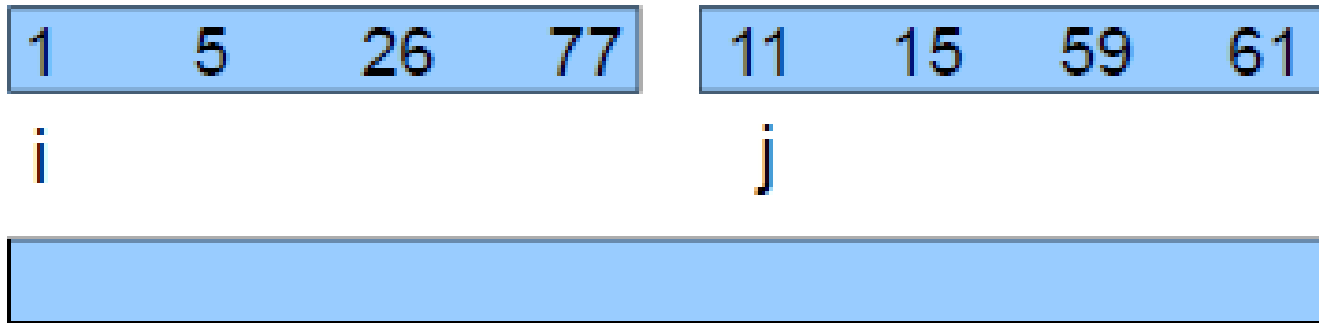
- Theorem 7.1: Any decision tree that sorts n distinct elements has a height of at least $\log_2(n!) + 1$
 - When sorting n elements, there are $n!$ different possible results (permutations).
 - Thus, every decision tree for sorting must have least $n!$ leaves.
 - A decision tree is also a binary tree, which can have at most 2^{k-1} leaves if its height is k
 - The height must be at least $\log_2 n! + 1$

DECISION TREE (CONT'D)

- Corollary: Any algorithm that sorts only by comparisons must have a worst-case computing time of $\Omega(n \log n)$
 - By the above theorem, there is a path of length $\log_2 n!$
 - $n! = n(n-1) \dots (2)(1) \geq (n/2)^{n/2}$
 - $\log_2 n! = (n/2) \log_2 (n/2) = \Omega(n \log n)$

SIMPLE MERGE (CONT'D)

- Example



SIMPLE MERGE (CONT'D)

1	5	26	77
---	---	----	----

i

11	15	59	61
----	----	----	----

j

1	5								
---	---	--	--	--	--	--	--	--	--

1	5	26	77
---	---	----	----

i

11	15	59	61
----	----	----	----

j

1	5	11							
---	---	----	--	--	--	--	--	--	--

1	5	26	77
---	---	----	----

i

11	15	59	61
----	----	----	----

j

1	5	11	15						
---	---	----	----	--	--	--	--	--	--

• • •

ANALYSIS OF SIMPLE MERGE

- If an array is used, additional space for $n - l + 1$ records is needed.
 - $n - l + 1$ = the number of elements to be merged
 - Time complexity of Simple Merge is linear
- If linked list is used instead, then additional space for $n - l + 1$ links is needed.

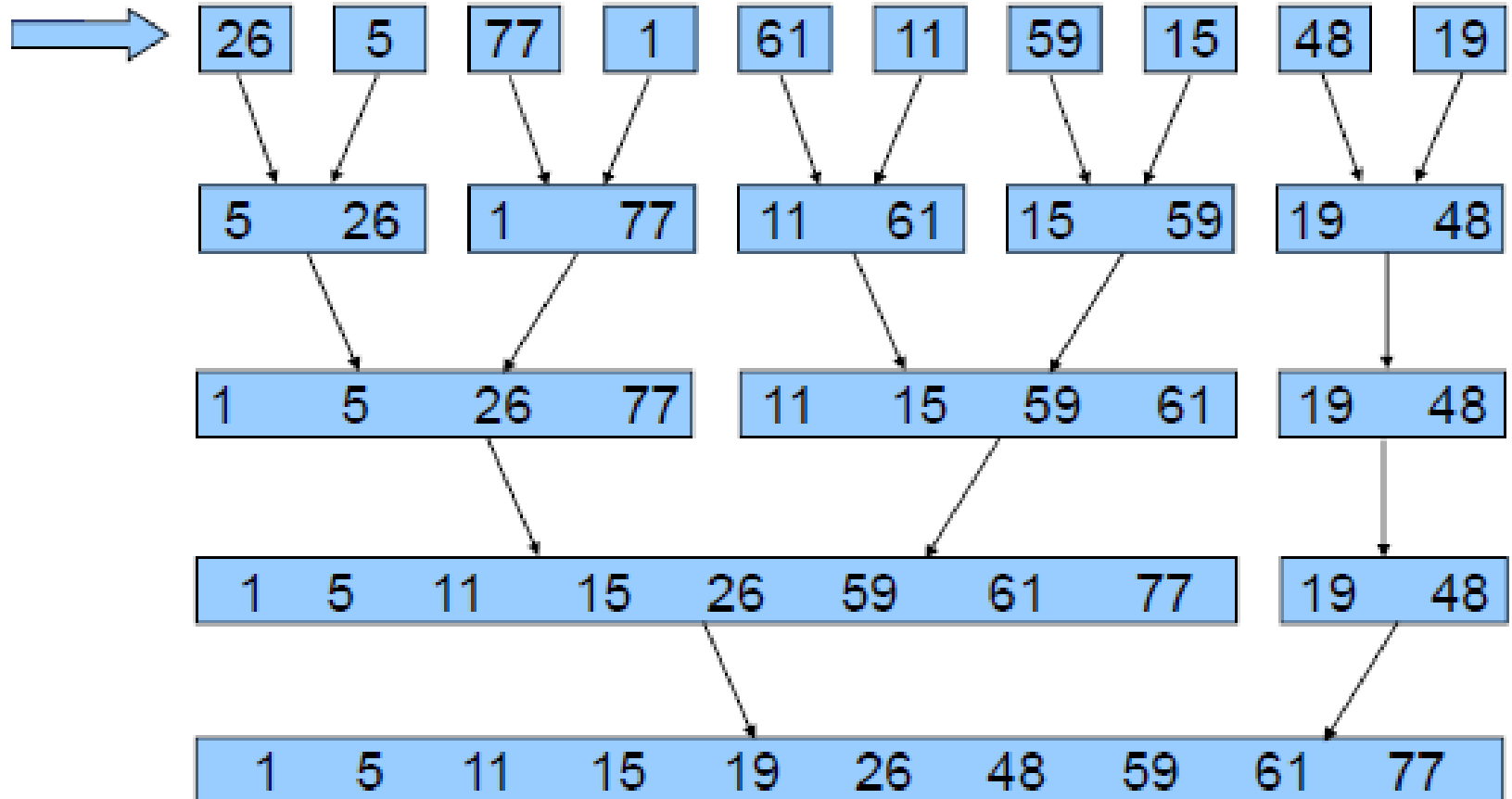


MERGE SORT

- Treat the input as n sorted lists, each of length 1.
- Lists are merged by pairs to obtain $n/2$ lists, each of size 2 (if n is odd, the one list is of length 1).
- The $n/2$ lists are then merged by pairs, and so on until we are left with only one list.

MERGE TREE

Merge pass



ITERATIVE MERGE SORT

$O(n)$ space

```
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];
}
```

addition space: $n-i+1$

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

MERGE PASS

```
void merge_pass(element list[], element sorted[],int n,  
                int length)  
{  
    int i, j;  
    for (i=0; i<n-2*length; i+=2*length)  
        merge(list,sorted,i,i+length-1, i+2*length-1);  
    if (i+length<n)  
        merge(list, sorted, i, i+length-1, n-1);  
    else  
        for (j=i; j<n; j++) sorted[j]= list[j];  
}
```



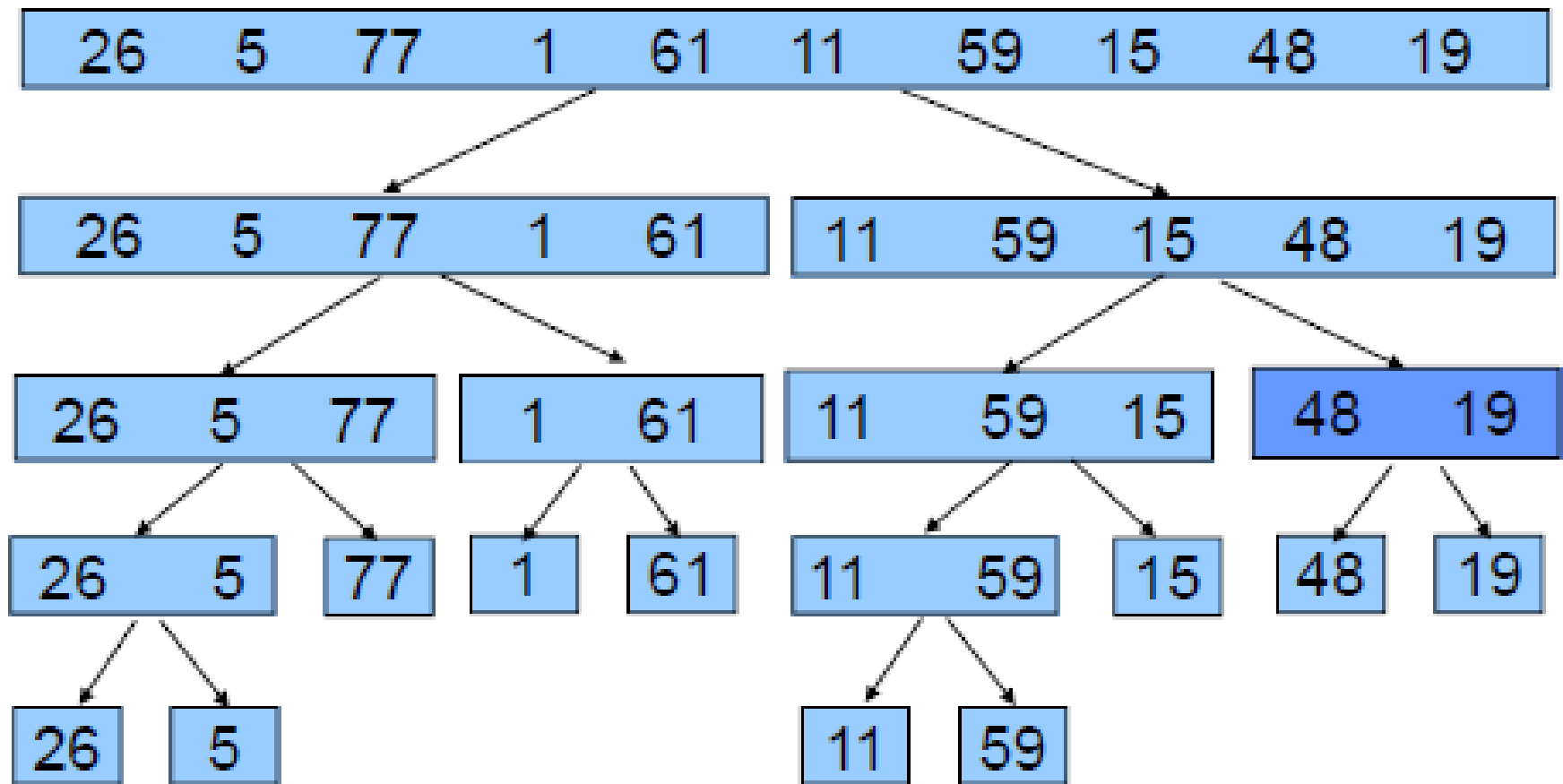
ANALYSIS OF MERGE SORT

- Total of $\lceil \log_2 n \rceil$ passes are made over the data. Each pass of merge sort takes $O(n)$ time.
- The total of computing time is $O(n \log n)$
- MergeSort results in a stable sorting function

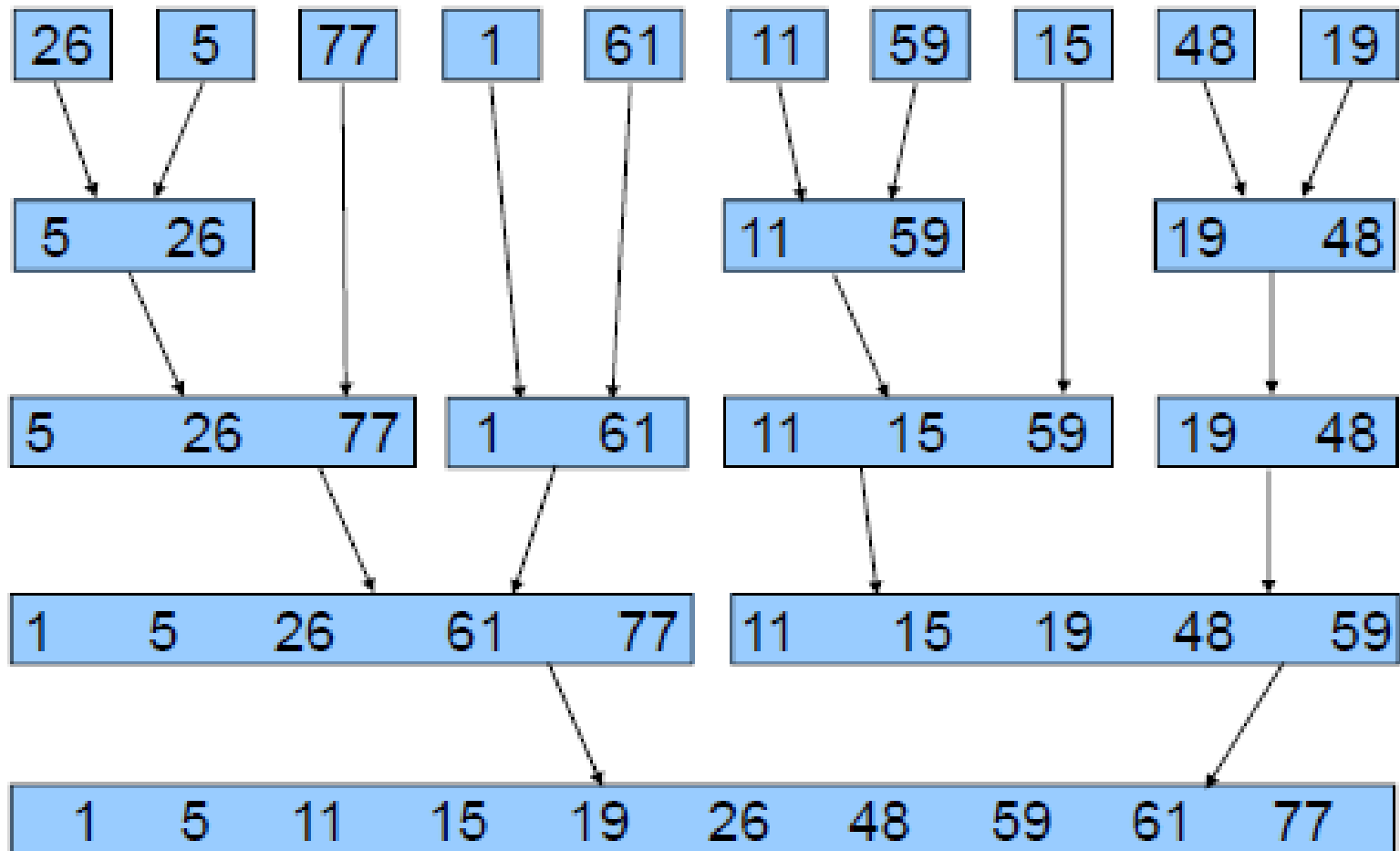
RECURSIVE MERGE SORT

- Recursive merge sort divides the list to be sorted into two roughly equal parts:
 - the left sublist $[\text{left} : (\text{left} + \text{right}) / 2]$
 - the right sublist $[(\text{left} + \text{right}) / 2 + 1 : \text{right}]$
- These sublists are sorted recursively, and the sorted sublists are merged.
- To avoid copying, the use of a linked list (integer instead of real link) for sublist is desirable.
- The recursive merge sort is stable

SUBLIST PARTITIONING FOR RECURSIVE MERGE SORT



SUBLIST PARTITIONING FOR RECURSIVE MERGE SORT (CONT'D)



RECURSIVE MERGE SORT

lower upper



A horizontal orange bar representing an array segment from index 'lower' to 'upper'.

Point to the start of sorted chain



An upward-pointing arrow from the text 'Point to the start of sorted chain' to the middle of the orange bar above.

```
int rmerge(element list[], int lower, int upper)
{
    int middle;
    if (lower >= upper) return lower;
    else {
        middle = (lower+upper)/2;
        return listmerge(list,
            rmerge(list,lower,middle),
            rmerge(list,middle, upper));
    }
}
```

lower middle upper



A horizontal orange bar representing an array split into two segments. The left segment is labeled 'lower' and the right segment is labeled 'upper'. An arrow points from the 'middle' label to the boundary between the two segments.

MERGED LINKED LIST

```
int listmerge(element list[], int first, int  
                second)
```

```
{
```

```
    int start=n;
```

```
    while (first!=-1 && second!=-1) {
```

```
        if (list[first].key<=list[second].key) {
```

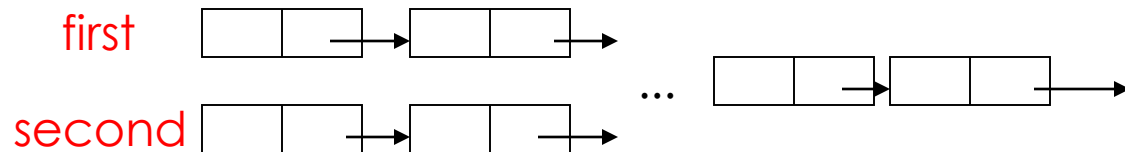
```
            /* key in first list is lower, link this  
             element to start and change start to  
             point to first */
```

```
            list[start].link= first;
```

```
            start = first;
```

```
            first = list[first].link;
```

```
}
```



```
else {  
    /* key in second list is lower, link this  
       element into the partially sorted list */  
    list[start].link = second;  
    start = second;  
    second = list[second].link;  
}  
}  
if (first == -1) first is exhausted.  
    list[start].link = second;  
else second is exhausted.  
    list[start].link = first;  
return list[n].link;  
}
```

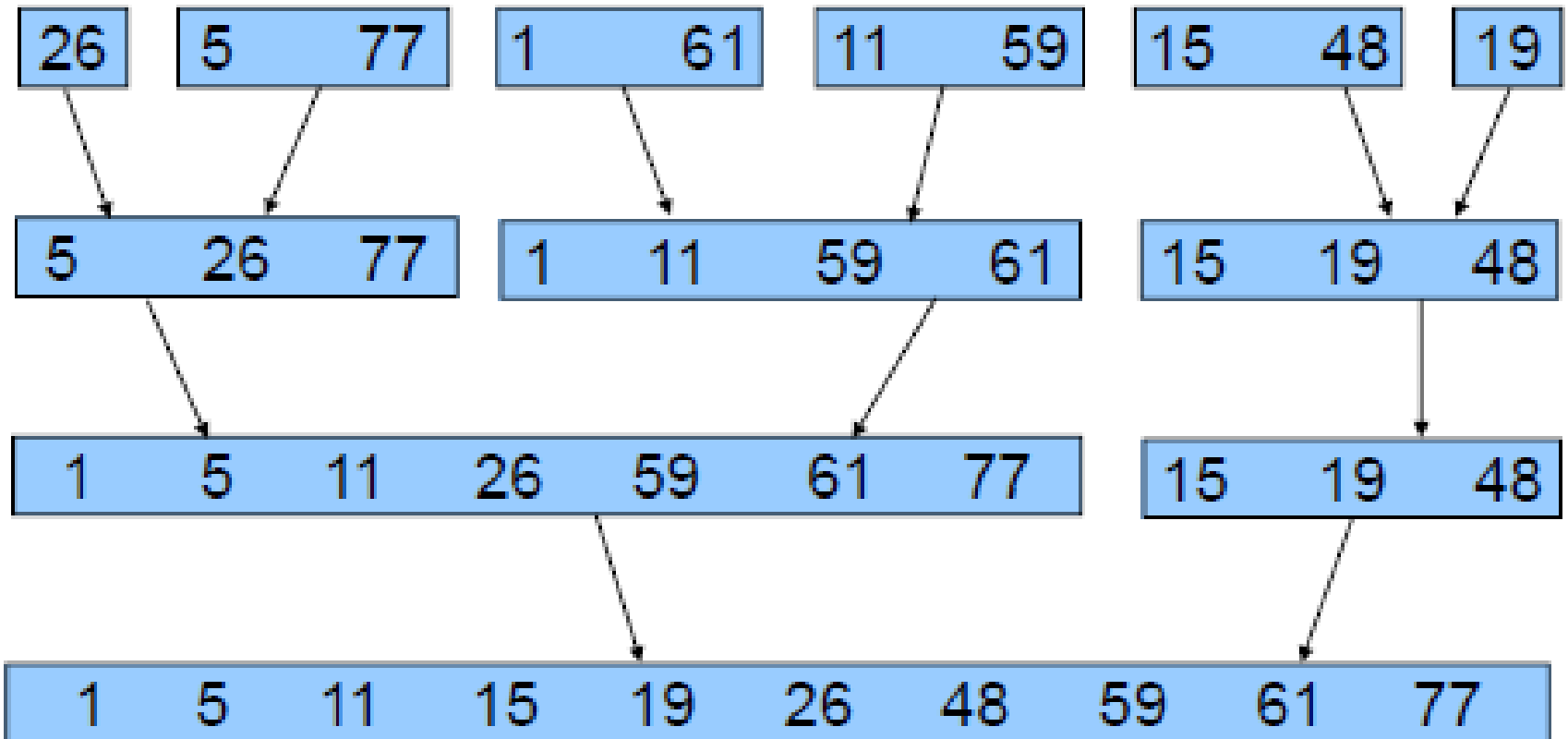
$O(n \log_2 n)$



NATURAL MERGE SORT

- Natural merge sort takes advantage of the prevailing order within the list before performing merge sort.
- It runs an initial pass over the data to determine the sublists of records that are in order.
- Then it uses the sublists for the merge sort.

NATURAL MERGE SORT EXAMPLE

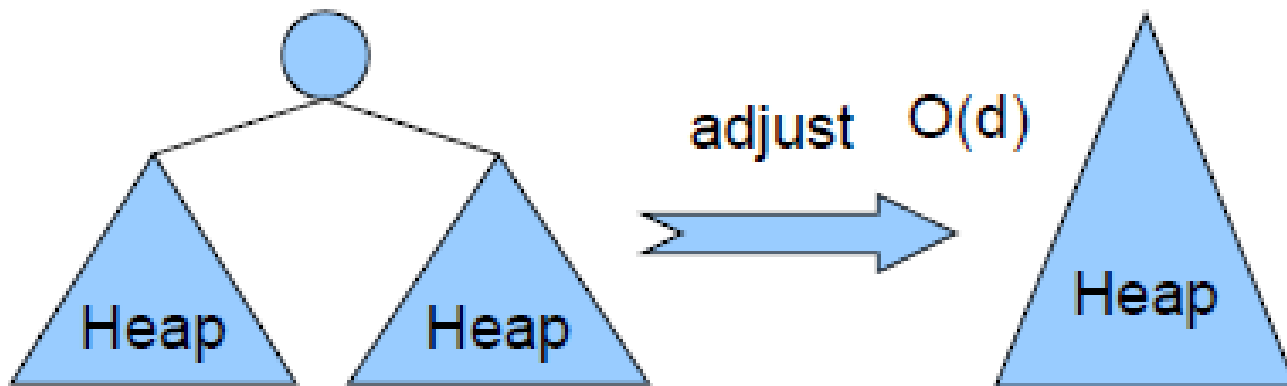


HEAP SORT

- Merge sort needs additional storage space proportional to the number of records in the file being sorted, even though its computing time is $O(n \log n)$
- $O(1)$ merge only needs $O(1)$ space but the sorting algorithm is much slower.
- We will see that heap sort only requires a fixed amount of additional storage and achieves worst case and average computing time $O(n \log n)$.
- Heap sort uses the max-heap structure.
- Heap sort is unstable

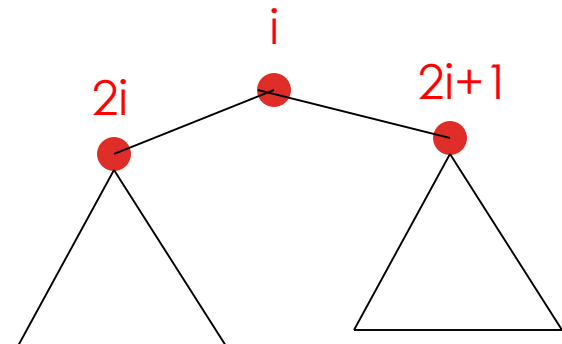
HEAP SORT (CONT'D)

- For heap sort, first of all, the n records are inserted into an empty heap. (Build heap via bottom-up)
- Next, the records are extracted from the heap one at a time.
- With the use of a special function `adjust()`, we can create a heap of n records faster.



ADJUSTING A MAX HEAP

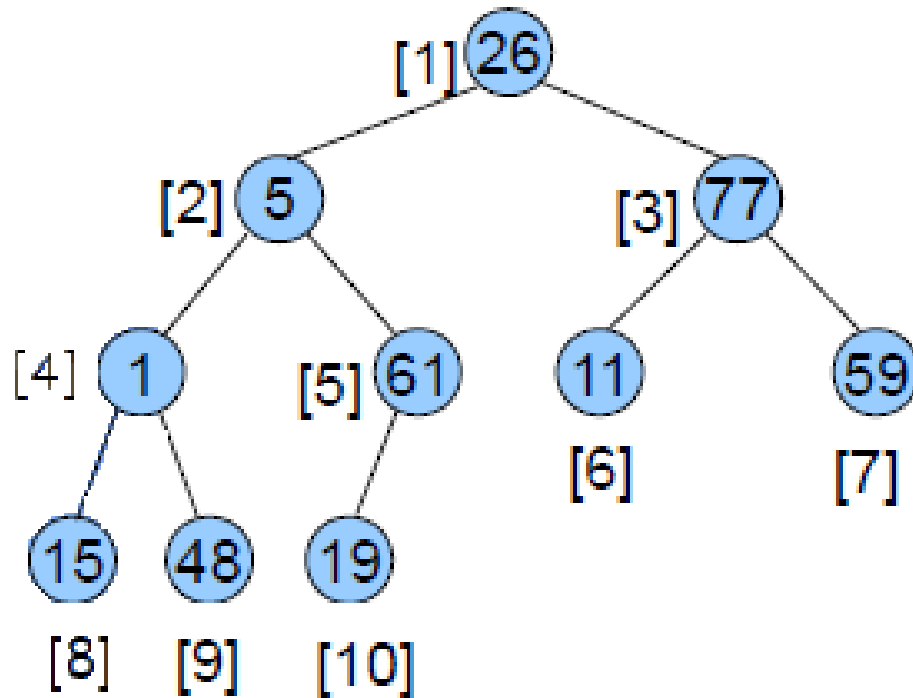
```
void adjust(element list[], int root, int n)
{
    int child, rootkey;    element temp;
    temp=list[root];      rootkey=list[root].key;
    child=2*root;
    while (child <= n) {
        if ((child < n) &&
            (list[child].key < list[child+1].key))
            child++;
        if (rootkey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



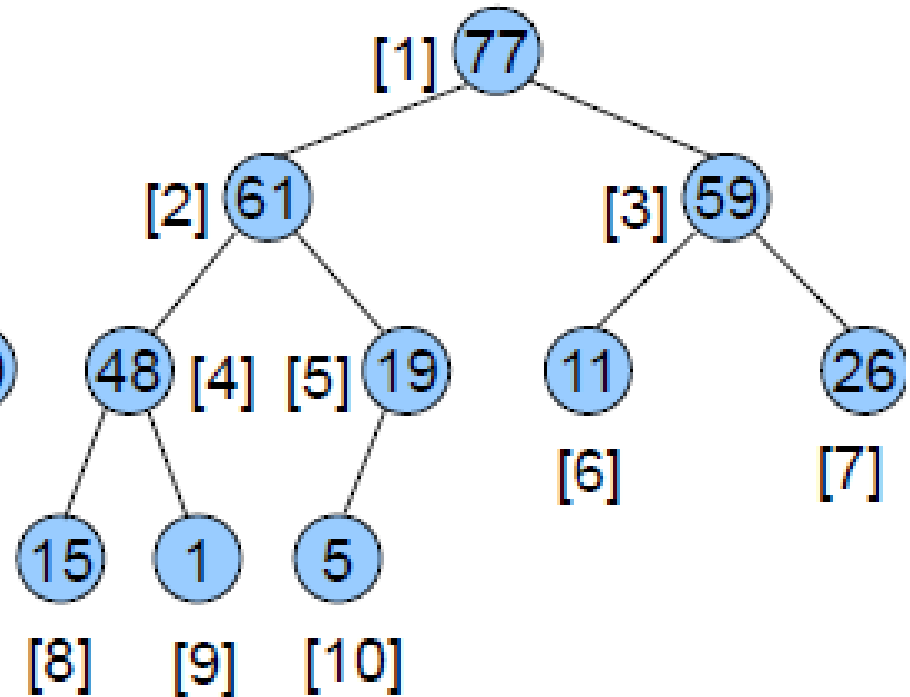
PROGRAM 7.14 (HEAP SORT)

```
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);
    for (i=n-1; i>0; i--) {
        SWAP(list[1], list[i+1], temp);
        adjust(list, 1, i);
    }
}
```

CONVERTING AN ARRAY INTO A MAX HEAP

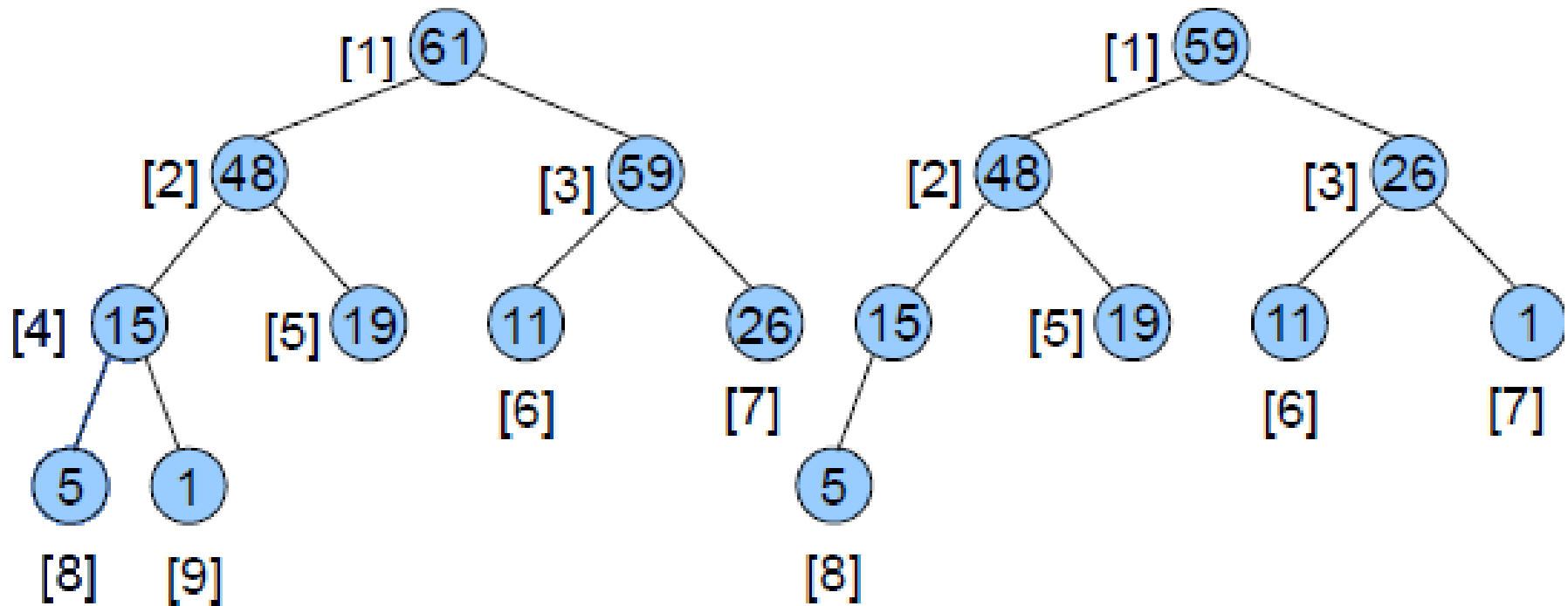


(a) Input array



(b) Initial heap

HEAP SORT EXAMPLE



Heap size = 9
Sorted = [77]

Heap size = 8
Sorted = [61, 77]

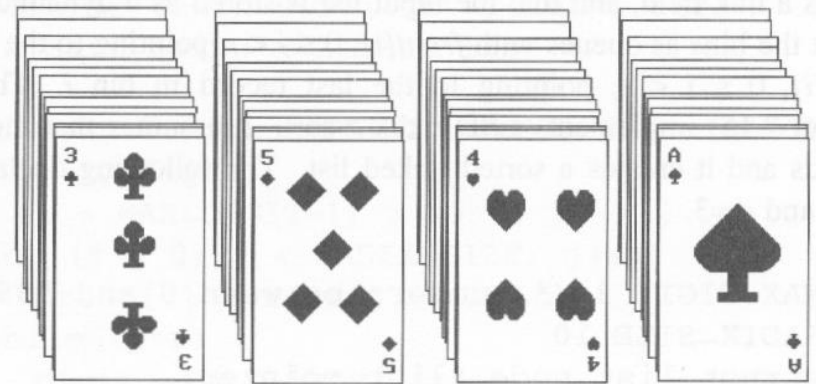
SUMMARY OF COMPARISON SORT

Name	Average Case	Worst Case	Extra Memory	Stable
Selection sort	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Insertion sort	$O(n^2)$	$O(n^2)$	$O(1)$	No
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n)$ $O(1)^*$	Yes
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(1)$	No
Quicksort	$O(n \log n)$	$O(n^2)$	$O(\log n)$	No

From Wikipedia

SORTING SEVERAL KEYS

- A list of records are said to be sorted with respect to the keys K^1, K^2, \dots, K^r iff for every pair of records i and j , $i < j$ and $(K^1_i, K^2_i, \dots, K^r_i) \leq (K^1_j, K^2_j, \dots, K^r_j)$.
- The r -tuple (x^1, x^2, \dots, x^r) is less than or equal to the r -tuple (y^1, y^2, \dots, y^r) iff either $x^i = y^i$, $1 \leq i \leq j$, and $x^{j+1} < y^{j+1}$ for some $j < r$ or $x^i = y^i$, $1 \leq i \leq r$.
- Example, sorting a deck of cards: suite and face value.



Suits: ♣ < ♦ < ♥ < ♠

Face values: 2 < 3 < 4 < ... < J < Q < K < A



SORTING SEVERAL KEYS (CONT'D)

- Two popular ways to sort on multiple keys
 - sort on the most significant key into multiple piles. For each pile, sort on the second significant key, and so on. Then piles are combined. This method is called sort on most-significant-digit-first (MSD).
 - The other way is to sort on the least significant digit first (LSD).

SORTING SEVERAL KEYS (CONT'D)

- LSD and MSD only define the order in which the keys are to be sorted. But they do not specify how each key is sorted.
- LSD and MSD can be used even when there is only one key.
 - E.g., if the keys are numeric, then each decimal digit may be regarded as a subkey. → Radix sort.

Sort by keys

K^0, K^1, \dots, K^{r-1}

Most significant key

Least significant key

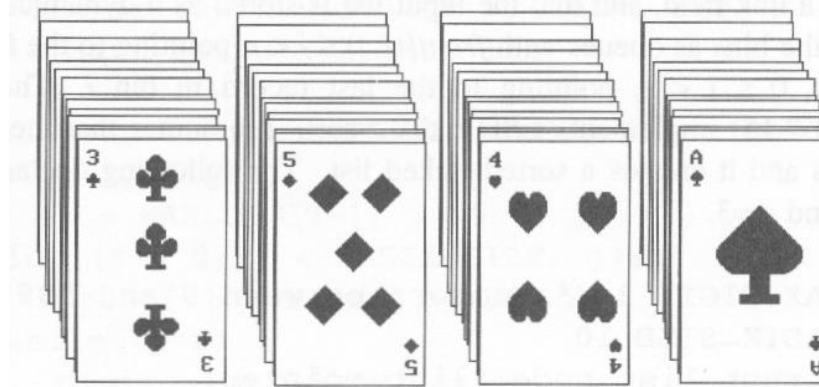
R_0, R_1, \dots, R_{n-1} are said to be sorted w.r.t. K_0, K_1, \dots, K_{r-1} iff

$$(k_i^0, k_i^1, \dots, k_i^{r-1}) \leq (k_{i+1}^0, k_{i+1}^1, \dots, k_{i+1}^{r-1}) \quad 0 \leq i < n-1$$

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

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

Arrangement of cards after first pass of an MSD sort(p.353)



Suits: ♣ < ♦ < ♥ < ♠

Face values: 2 < 3 < 4 < ... < J < Q < K < A

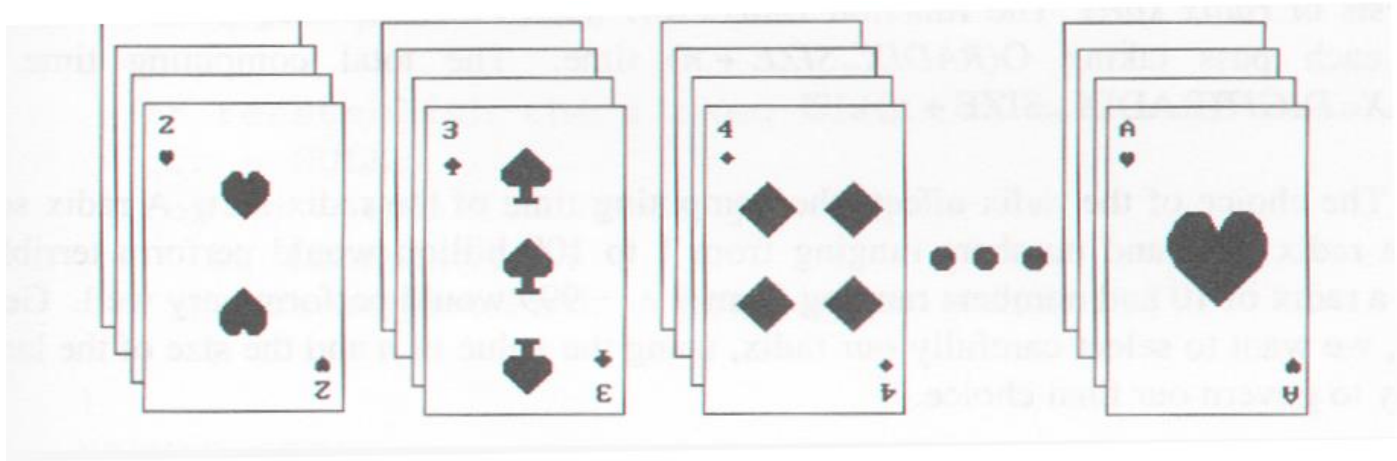
(1) MSD sort first, e.g., bin sort, four bins ♣ ♦ ♥ ♠

LSD sort second, e.g., insertion sort

(2) LSD sort first, e.g., bin sort, 13 bins 2, 3, 4, ..., 10, J, Q, K, A

MSD sort, e.g., bin sort four bins ♣ ♦ ♥ ♠

Arrangement of cards after first pass of LSD sort (p.353)





RADIX SORT

- In radix sort, we decompose the sort key using some radix r .
 - The number of bins needed is r .
- Assume the records R_1, R_2, \dots, R_n to be sorted based on a radix of r . Each key has d digits in the range of 0 to $r-1$.

RADIX SORT (CONT'D)

- Assume each record has a link field. Then the records in the same bin are linked together into a chain:
 - $f[i], 0 \leq i \leq r - 1$ (the pointer to the first record in bin i)
 - $e[i]$, (the pointer to the end record in bin i)
 - The chain will operate as a queue.

DATA STRUCTURES FOR LSD RADIX SORT

- An LSD radix r sort,
- R_0, R_1, \dots, R_{n-1} have the keys that are d -tuples $(x_0, x_1, \dots, x_{d-1})$

```
#define MAX_DIGIT 3
#define RADIX_SIZE 10
typedef struct list_node *list_pointer;
typedef struct list_node {
    int key[MAX_DIGIT];
    list_pointer link;
}
```

```
list_pointer radix_sort(list_pointer ptr)
{
    list_pointer front[RADIX_SIZE],
                rear[RADIX_SIZE];
    int i, j, digit;
    for (i=MAX_DIGIT-1; i>=0; i--) {
        for (j=0; j<RADIX_SIZE; j++)
            front[j]=rear[j]=NULL; Initialize bins to be empty queue.
        while (ptr) { Put records into queues.
            digit=ptr->key[I];
            if (!front[digit]) front[digit]=ptr;
            else rear[digit]->link=ptr;
```

$O(n)$

```

    rear[dit]=ptr;
    ptr=ptr->link; Get next record.

```

```

}

```

```

/* reestablish the linked list for the next pass */

```

```

ptr= NULL;

```

 $O(d(n+r))$

```

for (j=RADIX_SIZE-1; j>=0; j++)

```

```

    if (front[j]) {

```

 $O(r)$

```

        rear[j]->link=ptr;

```

```

        ptr=front[j];

```

```

    }

```

```

}

```

```

return ptr;

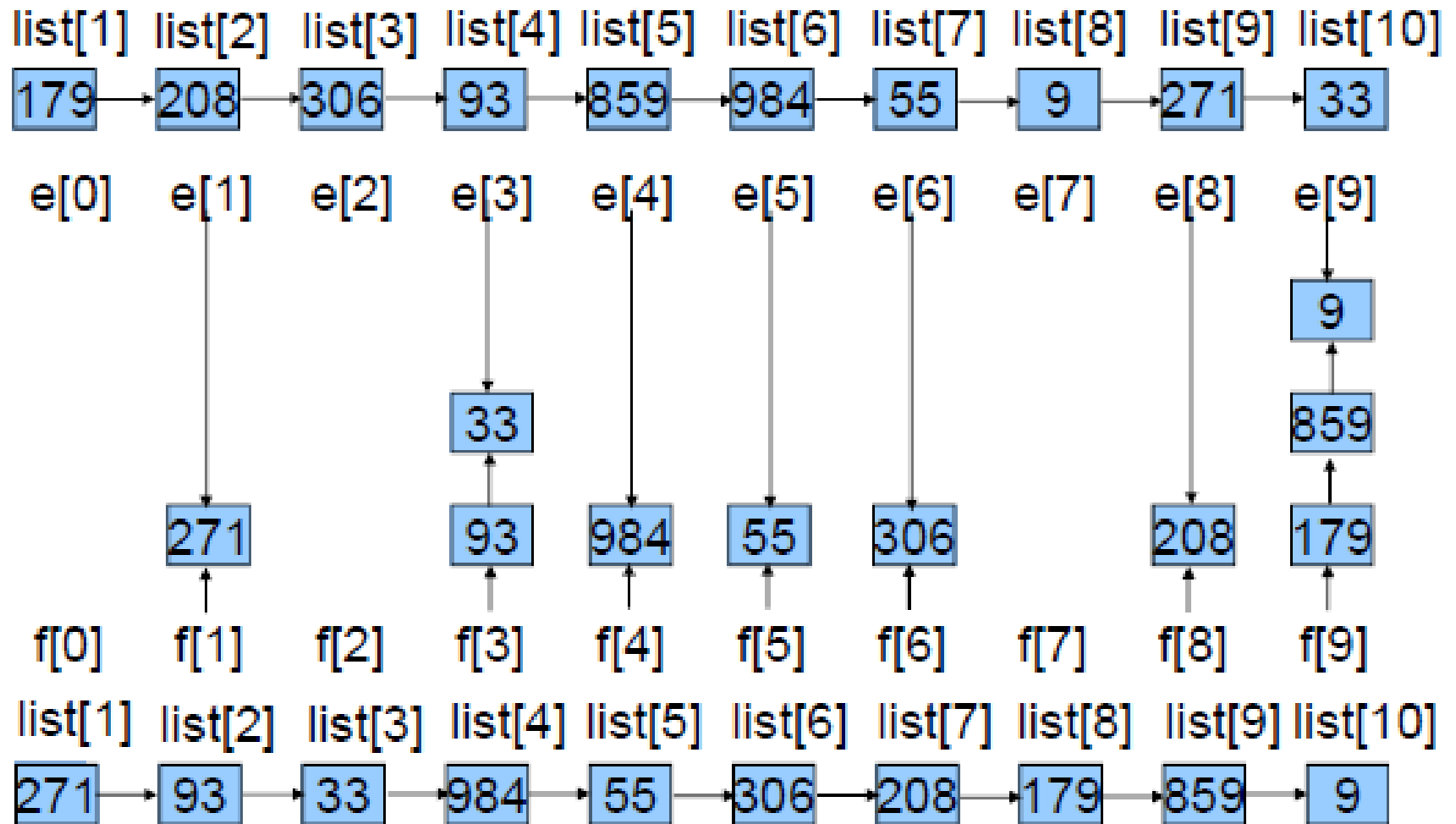
```

```

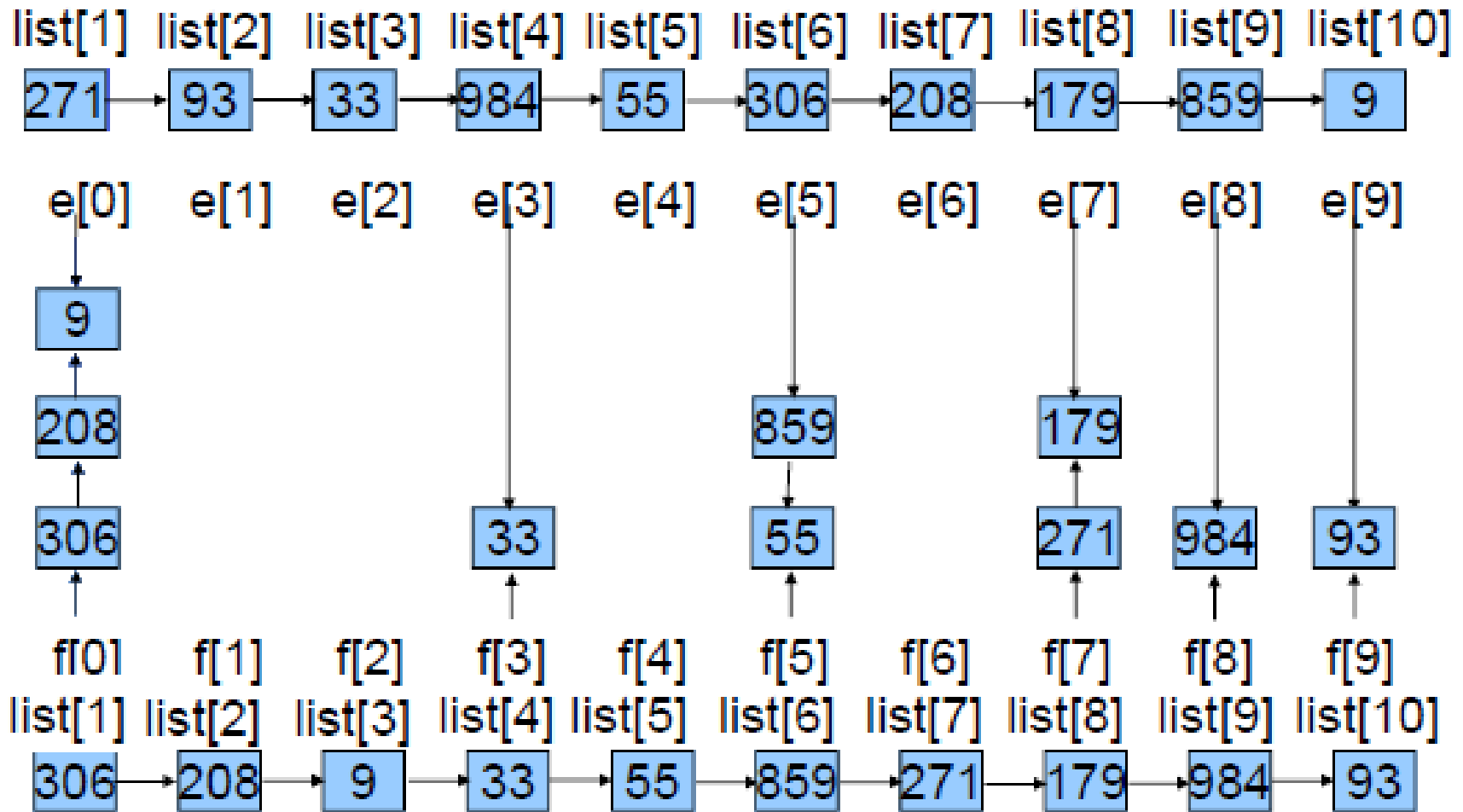
}

```

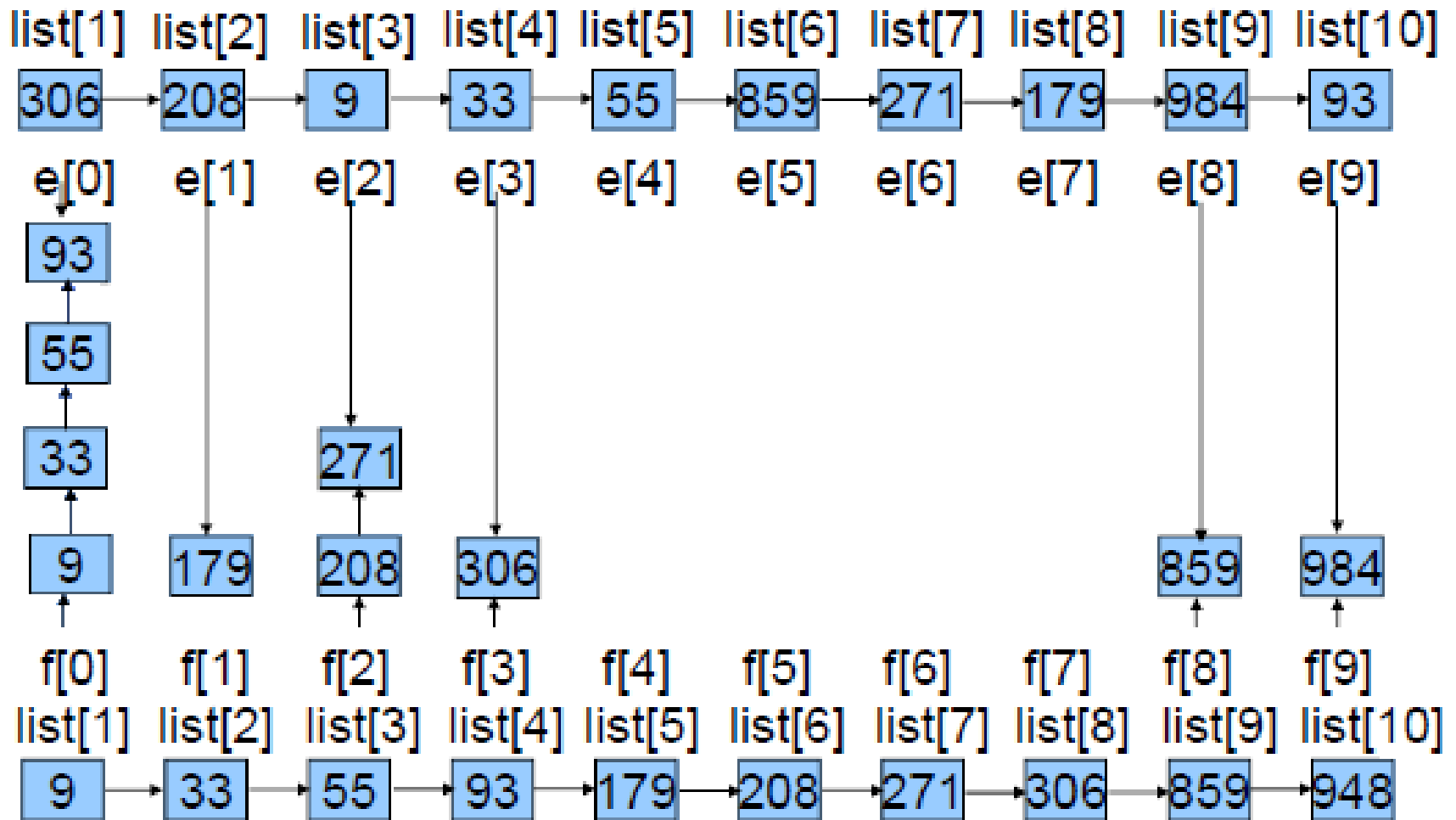

RADIX SORT EXAMPLE



RADIX SORT EXAMPLE (CONT'D)



RADIX SORT EXAMPLE (CONT'D)



SUMMARY OF INTERNAL SORTING

- No one method is best for all conditions.
 - Insertion sort is good when the list is **already partially ordered**. And it is **best for small number** of n .
 - Merge sort has the best worst-case behavior but needs more storage than heap sort.
 - Quick sort has the best average behavior, but its worst-case behavior is $O(n^2)$.
 - The behavior of radix sort depends on the size of the keys and the choice of r .

COMPLEXITY OF SORT

	stability	space	time		
			best	average	worst
Bubble Sort	stable	little	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion Sort	stable	little	$O(n)$	$O(n^2)$	$O(n^2)$
Quick Sort	unstable	$O(\log n)$	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Merge Sort	stable	$O(n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Heap Sort	unstable	little	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Radix Sort	stable	$O(np)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
List Sort	?	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Table Sort	?	$O(n)$	$O(1)$	$O(n)$	$O(n)$



EXTERNAL SORTING

- There are some lists that are too large to fit in the memory of a computer. So internal sorting is not possible.
- Some records are stored in the disk (tape, etc.). System retrieves a block of data from a disk at a time. A block contains multiple records.
- The most popular method for sorting on external storage devices is merge sort.
 - Segments of the input list are sorted.
 - Sorted segments (called runs) are written onto external storage.
 - Runs are merged until only one run is left.

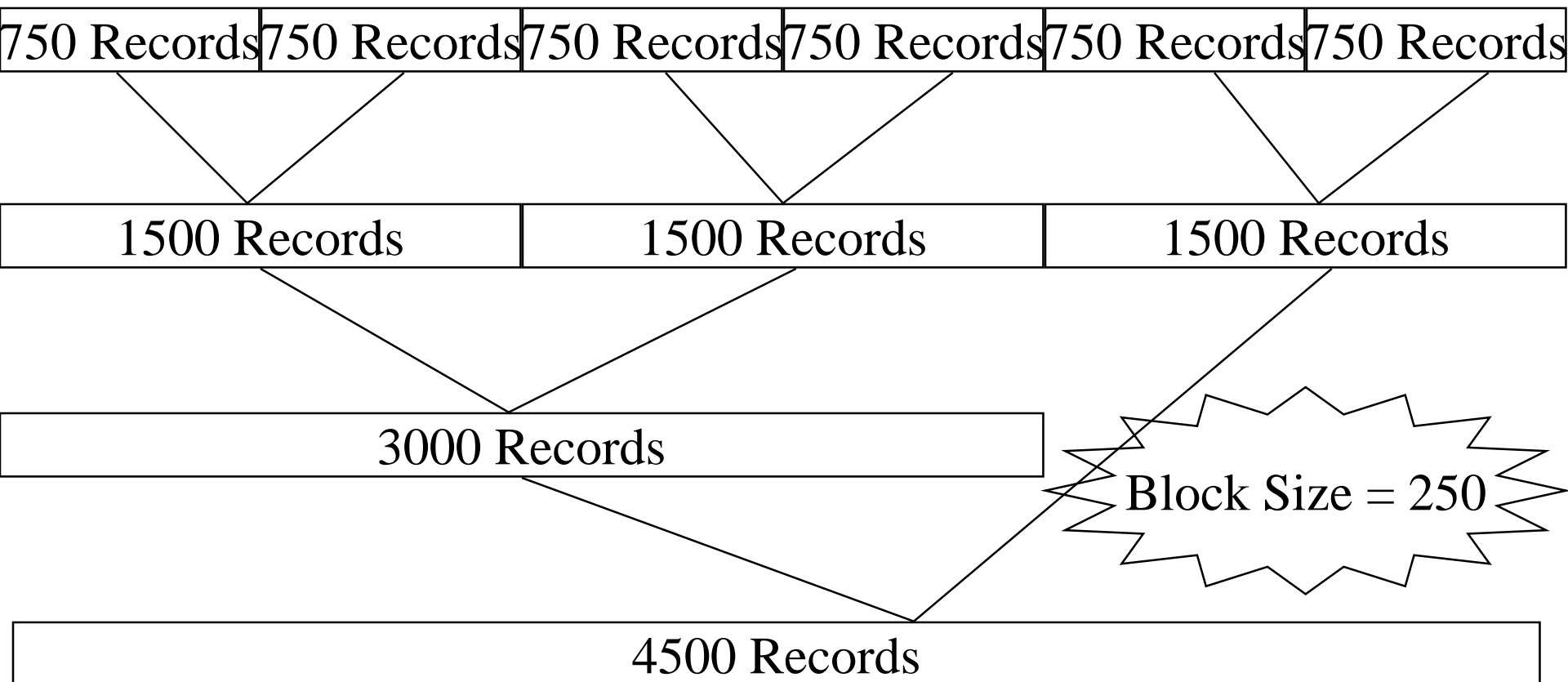


EXTERNAL SORTING

- Consider a computer which is capable of sorting 750 records is used to sort 4500 records.
- Six runs are generated with each run sorting 750 records.
- Allocate three 250-record blocks of internal memory for performing merging runs. Two for input run 2 runs and the last one is for output.
- Three factors contributing to the read/write time of disk:
 - seek time
 - latency time
 - transmission time

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



2 2/3 passes

TIME COMPLEXITY OF EXTERNAL SORT

- $t_{IO} = t_s + t_l + t_{rw}$
 - t_s = maximum seek time
 - t_l = maximum latency time
 - t_{rw} = time to read or write on block of 250 records
- t_{IS} = time to internal sort 750 records
- nt_m = time to merge n records from input buffers to the output buffer

TIME COMPLEXITY OF EXTERNAL SORT

Operation	time
(1) read 18 blocks of input , $18t_{IO}$, internally sort, $6t_{IS}$, write 18 blocks, $18t_{IO}$	$36 t_{IO} + 6 t_{IS}$
(2) merge runs 1-6 in pairs	$36 t_{IO} + 4500 t_m$
(3) merge two runs of 1500 records each, 12 blocks	$24 t_{IO} + 3000 t_m$
(4) merge one run of 3000 records with one run of 1500 records	$36 t_{IO} + 4500 t_m$
Total Time	$96 t_{IO} + 12000 t_m + 6 t_{IS}$

Critical factor: number of passes over the data

Runs: m , pass: $\lceil \log_2 m \rceil$



K-WAY MERGING

- To merge m runs via 2-way merging will need $\lceil \log_2 m \rceil + 1$ passes where m is the number of runs
- If we use higher order merge, the number of passes over would be reduced.
- With k -way merge on m runs, we need $\lceil \log_k m \rceil$ passes over.

K-WAY MERGING (CONT'D)

- But is it always true that the higher order of merging, the less computing time we will have?
 - Not necessary!
 - $k-1$ comparisons are needed to determine the next output.
 - The number of comparisons is $n(k-1)\log_k m$
 - If loser tree is used to reduce the number of comparisons, we can achieve complexity of $O(n\log_2 m)$
 - The data block size reduced as k increases. Reduced block size implies the increase of data passes over (seek and latency times)

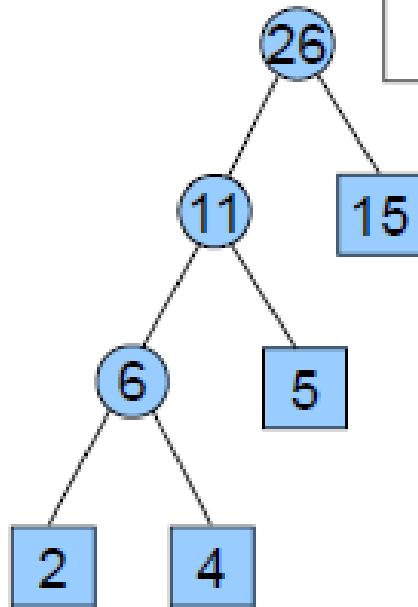


BUFFER HANDLING FOR PARALLEL OPERATION

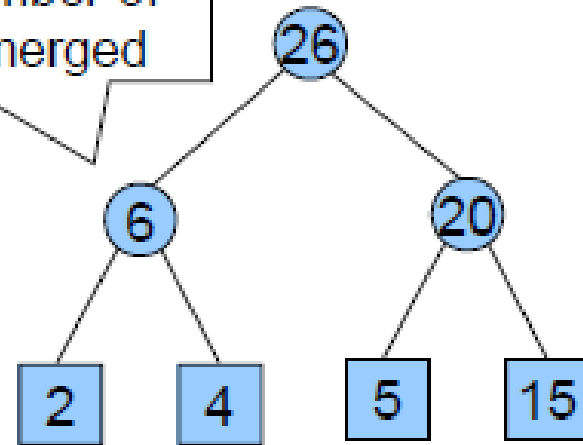
- To achieve better performance, multiple input buffers and two output buffers are used to avoid idle time.
- Evenly distributing input buffers among all runs may still have idle time problem. Buffers should be dynamically assigned to whoever needs to retrieve more data to avoid halting the computing process.
- We should take advantage of task overlapping and keep computing process busy and avoid idle time.

OPTIMAL MERGING OF RUNS

The time for merge is linear in the number of records to be merged



$$\begin{aligned}\text{weighted external path length} \\ &= 2 \cdot 3 + 4 \cdot 3 + 5 \cdot 2 + 15 \cdot 1 \\ &= 43 (=6+11+26)\end{aligned}$$



$$\begin{aligned}\text{weighted external path length} \\ &= 2 \cdot 2 + 4 \cdot 2 + 5 \cdot 2 + 15 \cdot 2 \\ &= 52 (=6+20+26)\end{aligned}$$

OPTIMAL MERGING OF RUNS (CONT'D)

- The cost of a k -way merge of n runs of length $q_i, 1 \leq i \leq n$, is minimized by using a merge tree of degree k that has minimum weighted external path length.

HUFFMAN CODE

- Assume we want to obtain an optimal set of codes for messages M_1, M_2, \dots, M_{n+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. These codes are called Huffman codes.
- The cost of decoding a code word is proportional to the number bits in the code. This number is equal to the distance of the corresponding external node from the root node.

HUFFMAN CODE (CONT'D)

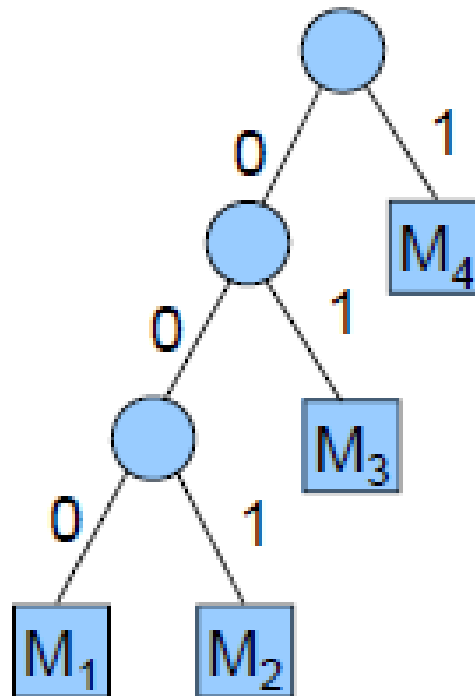
- If q_i is the relative frequency with which message M_i will be transmitted, then the expected decoding time is

$$\sum_{1 \leq i \leq n+1} q_i d_i$$

where d_i is the distance of the external node for message M_i from the root node.

HUFFMAN CODE (CONT'D)

- The expected decoding time is minimized by choosing code words resulting in a decode tree with minimal weighted external path length.



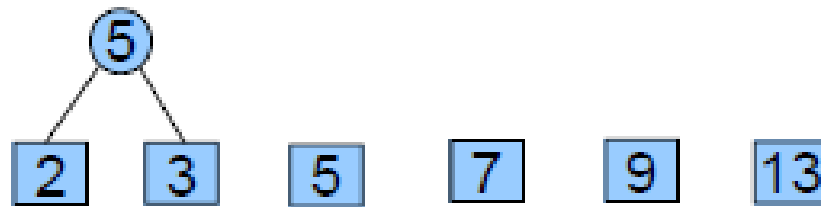


HUFFMAN TREE EXAMPLE

- text
 - ABBABCCDDE..
- Scan the text to calculate the number of appearances of each character
 - A: 2, B: 3, C: 5, D: 7, E: 9, F: 13
- Build Huffman tree
- Encode text
 - 1000 1001 1001.....

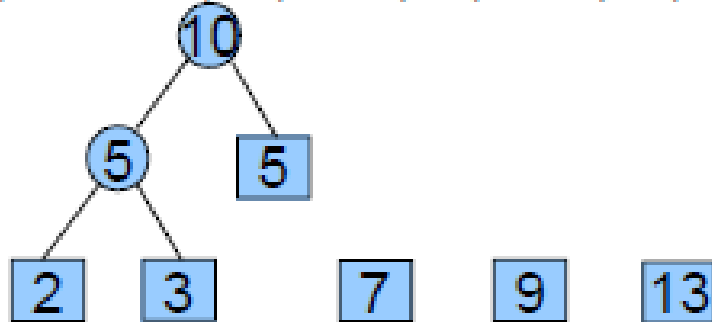
HUFFMAN TREE EXAMPLE

$\{(A: 2), (B: 3), (C: 5), (D: 7), (E: 9), (F: 13)\}$



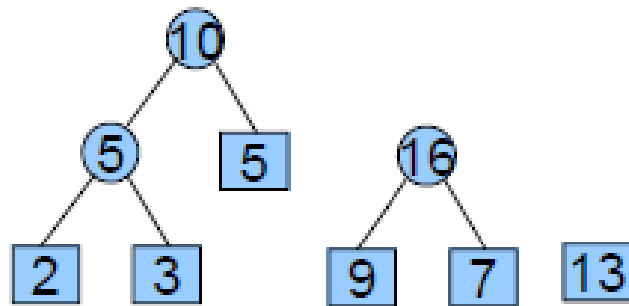
(a)

$\{(A, B: 5), (C: 5), (D: 7), (E: 9), (F: 13)\}$

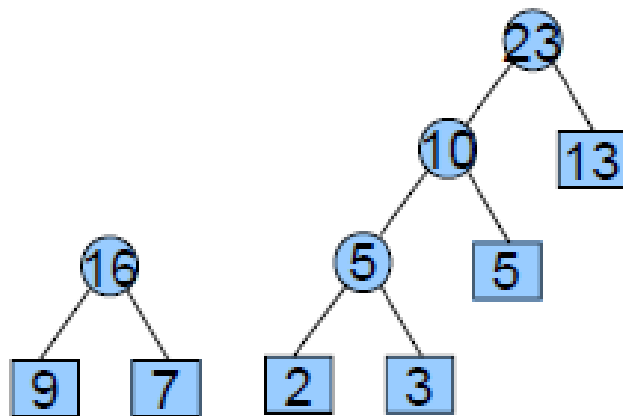


(b)

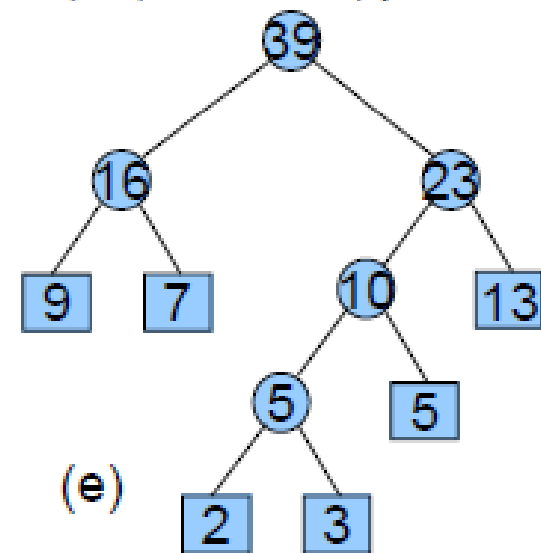
$\{(D: 7), (E: 9), (A, B, C: 10), (F: 13)\}$



(c) $\{(A,B,C: 10), (F: 13), (D, E: 16)\}$



(d)
 $\{(D, E: 16), (A,B,C,F: 23)\}$



(e)
 $\{(A,B,C,D,E,F: 39)\}$