POSTGRADUATE DEPARTMENT OF COMPUTER APPLICATIONS,
GOVERNMENT ARTS COLLEGE(AUTONOMOUS),
COIMBATORE 641018.

DATA STRUCTURES AND ALGORITHMS

The contents in this E material are from

Ellis Horowitz, Sartaj Sahni, and Susan Anderson-Freed
“Fundamentals of Data Structures in C”,
Computer Science Press, 1992.

UNIT 4

FACULTY

Dr.R.A.ROSELINE M.Sc.M.Phil.,Ph.D,

Associate Professor and Head,

Postgraduate Department of Computer Applications,

Government Arts College(Autonomous),

Coimbatore 641018.

Symbol Table

name	attributes
:	:

- Collect attributes when a name is declared.
- Provide information when a name is used.

Two issues:

1. interface: how to use symbol tables
2. implementation: how to implement it.

Symbol Table

A symbol table class

A symbol table class provides the following functions:

1. create() : symbol_table

2. destroy (symbol_table)

3. enter (name, table) : pointer to an entry

4. find (name, table) : pointer to an entry

5. set_attributes (*entry, attributes)

6. get_attributes (*entry) : attributes

Symbol Table

basic implementation techniques

- basic operations: enter() and find()
- considerations: number of names
  storage space
  retrieval time
- organizations:

<1> unordered list (linked list/array)

<2> ordered list

» binary search on arrays

» expensive insertion

(+) good for a fixed set of names
(e.g. reserved words, assembly
opcodes)

<3> binary search tree

» On average, searching takes
O(log(n)) time.

» However, names in programsare
not chosen randomly.

<4> hash table: most common

(+) constant time

Symbol Table

binary search tree

- For balanced tree, search takes O(log(n)) time.

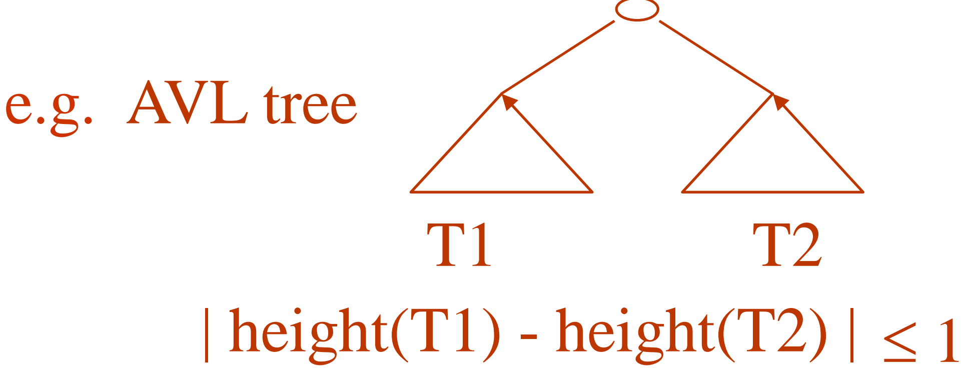
- For random input, search takes O(log(n)) time.
  - » However, average search time is 38% greater than that for a balanced tree.

- In worst case, search takes O(n) time.

e.g. (A B C D E) ⟹ linear list
(A E B D C) ⟹ linear list

Symbol Table

- Solution : keep the tree approximately balanced.



- Insertion/deletion may need to move some subtrees to keep the tree approximately balanced.

Symbol Table

(+) space = O(n)

[compare] hash table needs a fixed size regardless of the number of entries.

Application: Sometimes, we may use multiple symbol tables. AVL is better than hashing in this case.

Symbol Table

hash tables

hash : name $\longrightarrow$ address

- hash is easy to compute

- hash is uniform and randomizing.

Ex. $(C_1 + C_2 + \ldots\ldots + C_n) \bmod m$

$(C_1 * C_2 * \ldots\ldots * C_n) \bmod m$

$(C_1 + C_n) \bmod m$

$(C_1 \oplus C_2 \oplus \ldots\ldots \oplus C_n) \bmod m$

Symbol Table

- conflicts :

<1> linear resolution

  - Try h(n), h(n)+1, h(n)+2, ...
  - Problematic if the hash table did not reserve enough empty slots.

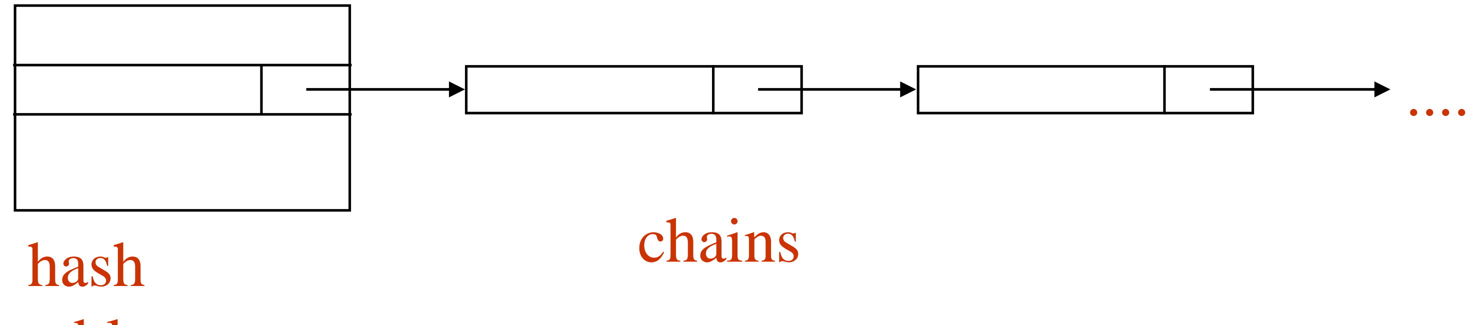
<2> add-the-hash rehash

  - Try h(n), (2*h(n)) mod m, (3*h(n)) mod m, ...
  - m must be prime.

<3> quadratic rehash

  - Try h(n), (h(n)+1) mod m, (h(n)+4) mod m, ...

<4> chaining



Symbol Table

- Advantages of chaining:

  (+) less space overhead of the hash table

  (+) does not fail catastrophically when the hash table is almost full.

- The chains may be organized as search trees, rather than linear lists.

- More importantly, we can remove all names defined in a scope when the scope is closed.

```
{  var  a;

      {   var a, b, c;
            ...
            ...
       }
```

Symbol Table

String space

Should we store the identifiers in the symbol table?

- Name lengths differ significantly.
  e.g. i, account_receivable,
  a_very_very_long_name

- Space is wasted if space of max size is reserved.

- Solution : store the identifiers in string space, and store an index and length in the symbol table.

Symbol Table

Ex.

1 8 13 20

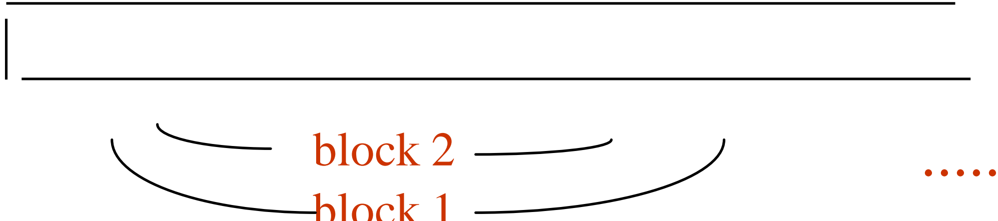
a c c o u n t t e m p l b a l a n c e ......

string space

symbol table

1	7	.....
13	7	.....
8	5	.....

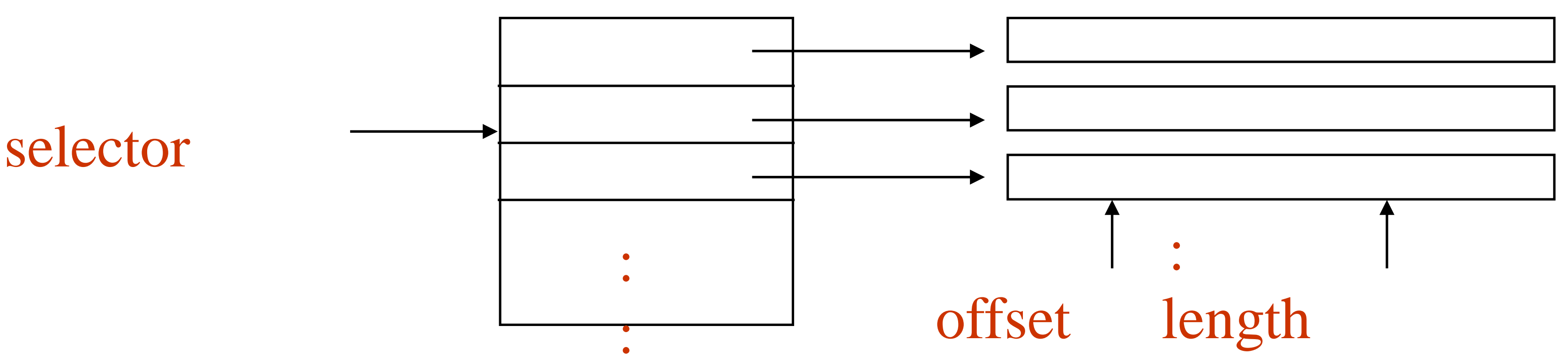
- Usually, symbol table manages the string space.
- Names in string space may be re-used.
- Individual scopes may use different string space. However, in block- structured languages, a single string space is good for re-claiming space (when the name becomes invisible).



Symbol Table

- How large is the string space?
  - too big —— waste space
  - too small —— run out of space

Solution : segmented string space

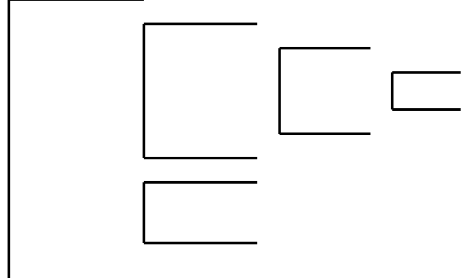


We need (selector, offset, length) to identify a name.

Symbol Table

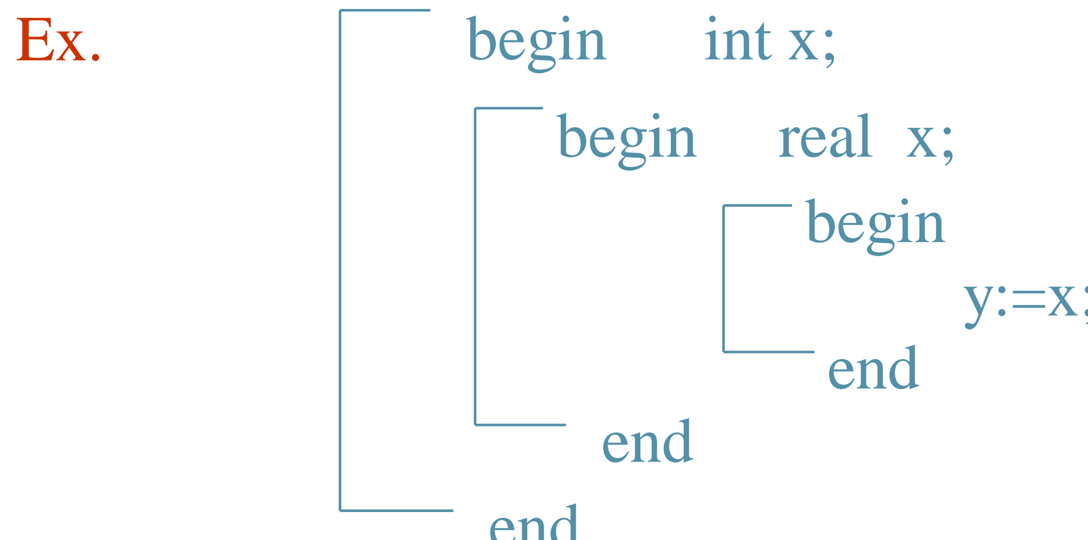
block-structured symbol table

- Scopes may be nested.



- Typical visibility rules specify that a use of name refers to the declaration in the innermost enclosing scope.

Ex.



Symbol Table

- Symbol tables in block-structured languages:

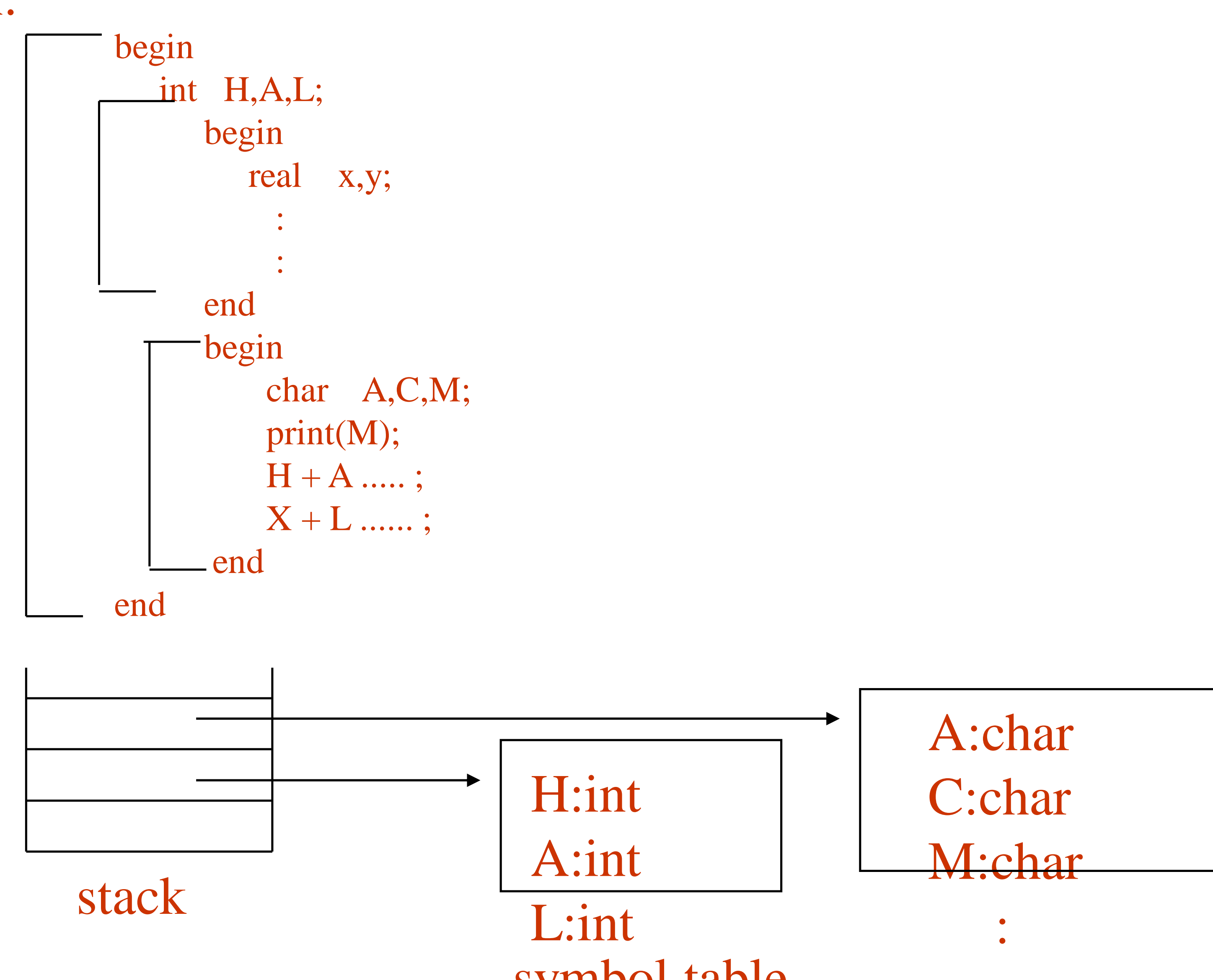
    1. many small tables
    2. one global table

<1> many small tables

- one symbol table per scope.
- use a stack of tables.
- The symbol table for the current scope is on stack top.
- The symbol tables for other enclosing scopes are placed under the current one.
- Push a new table when a new scope is entered.
- Pop a symbol table when a scope is closed.

Symbol Table

Ex.



Symbol Table

- To search for a name, we check the symbol tables on the stack from top to bottom.

  (-) We may need to search multiple tables.

  E.g. A global name is defined in the bottom-most symbol table.

  (-) Space may be wasted if a fixed-sized hash table is used to implement symbol tables.

  - hash table too big -- waste
  - hash table too small -- collisions

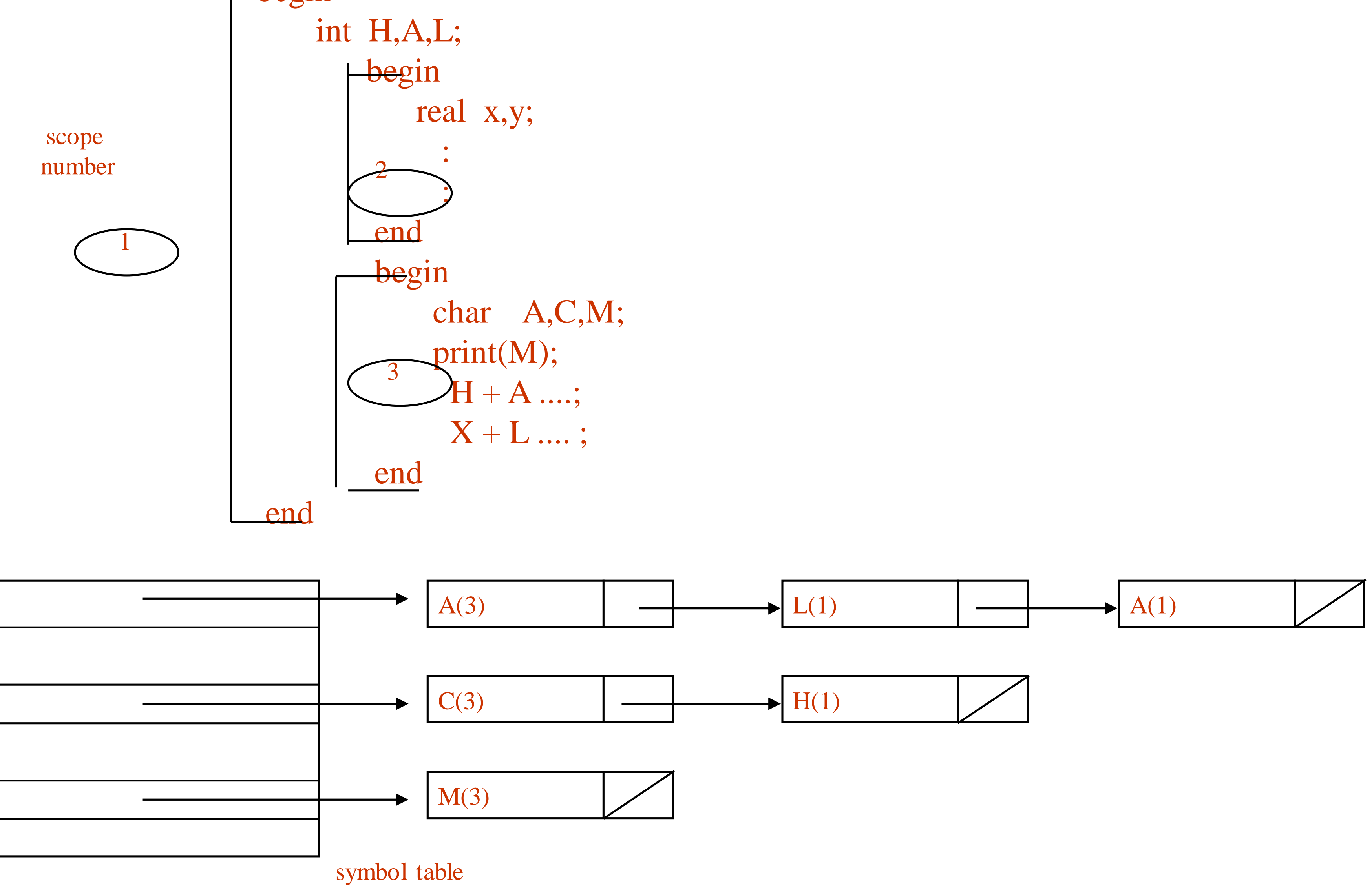
Symbol Table

<2> one global table

- All names are in the same table.
- What about the same name is declared several times?
    - Each name is given a scope number.
    - <name, scope number> should be unique in the table.

Symbol Table

Ex.



Symbol Table

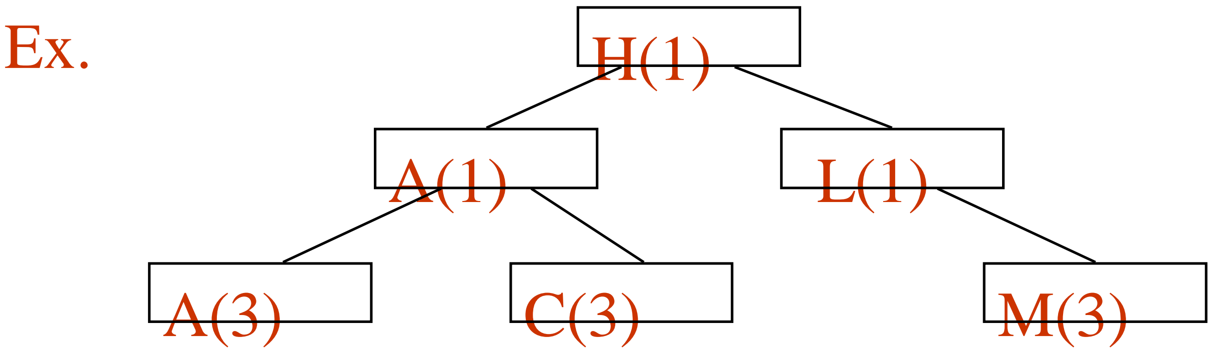
- To search a name is easy.
- New names are placed at the front of lists.
- To close a scope, we need to remove all entries defined in that scope.

  (We need to examine each list.)

Symbol Table

- One global table cannot be implemented with binary trees easily.
  - It is not easy to delete all entries declared in a scope when the scope is closed.
  - To find a name, we need to search the last entry (rather than the first entry).

Ex.



Consider the case when A is being searched for.

Symbol Table

Comparisons:

many small tables	one global table
simpler	more complicated
slower	faster search
less efficient (for hash tables)	efficient storage use
	Need space for scope numbers
hash table or trees	Use hash tables
Good for keeping	Good for 1-pass

SORTING

Sequential Search

- Example

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

- unsuccessful search
  - n+1
- successful search

$$\sum_{i=0}^{n-1}(i+1)/n = \frac{n+1}{2}$$

* (P.320)

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

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

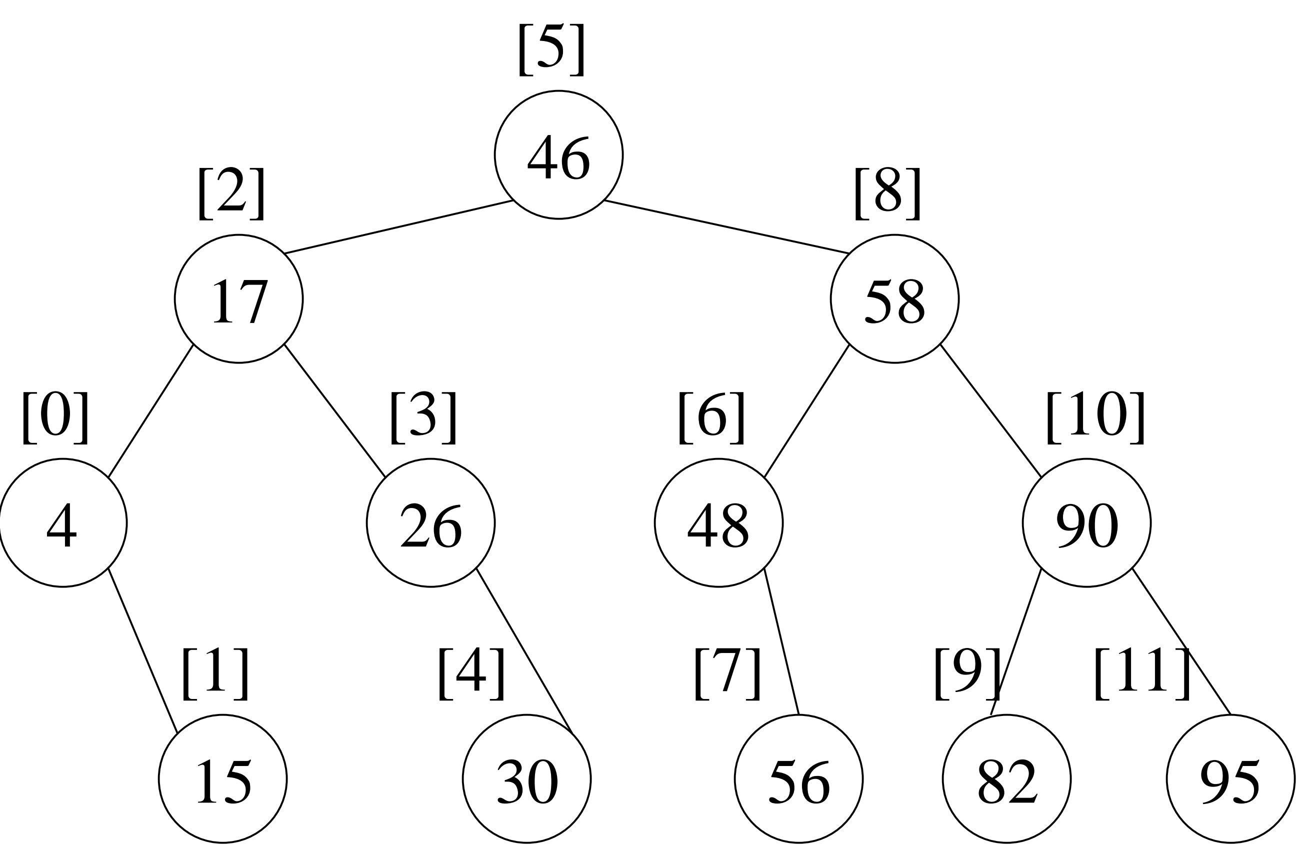
```
int seqsearch( int list[ ], int searchnum, int n )
{
/*search an array, list, that has n numbers. Return i, if
list[i]=searchnum. Return -1, if searchnum is not in the list */
    int i;
    list[n]=searchnum;    sentinel
    for (i=0; list[i] != searchnum; i++)
        ;
    return (( i<n) ? i :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

***Program 7.2:** Binary search (p.322)

```
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 7.1:** Decision tree for binary search (p.323)



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

List Verification

- Compare 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 7.3:** verifying using a sequential search(p.324)

```
void verify1(element list1[], element list2[ ], int n, int m)
/* compare two unordered lists list1 and list2 */
{
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
 /* check each of the other fields from list1[i] and list2[j], and
print out any discrepancies */
```

(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.

```
        marked[j] = TRUE;
for ( i=0; i<m; i++)
        if (!marked[i])
            printf("%d is not in list1\n", list2[i]key);
}
```

***Program 7.4:** Fast verification of two lists (p.325)

```
void verify2(element list1[ ], element list2 [ ], int n, int m)
/* Same task as verify1, but list1 and list2 are sorted */
{
   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) {
        /* compare list1[i] and list2[j] on each of the other field
           and report any discrepancies */
        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);
}
```

Sorting Problem

- Definition
  - given $(R_0, R_1, \ldots, R_{n-1})$, where R_i = key + data
    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

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 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;
  }
}
```

insertion_sort(list,n)

worse case

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

left out of order (LOO)

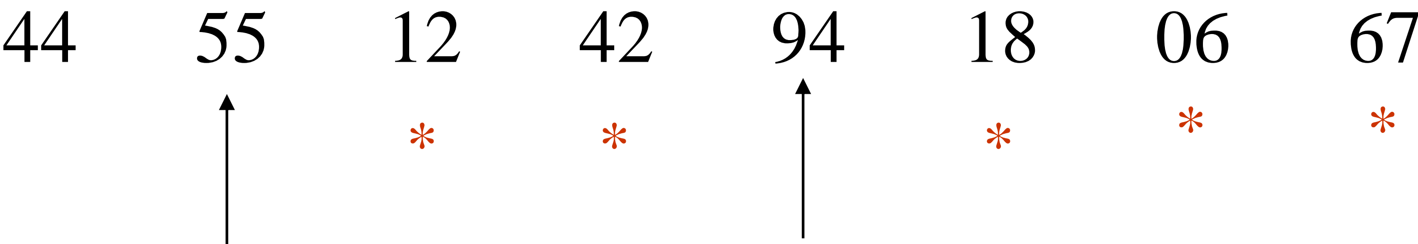
i	0	1	2	3	4
-	2	3	4	5	1
1	2	3	4	5	1
2	2	3	4	5	1
3	2	3	4	5	1
4	1	2	3	4	5

O(n)

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

k: # of records LOO

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



Variation

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

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

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

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

two partitions

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		

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);
  }
}
```

Analysis for Quick Sort

- Assume that each time a record is positioned, the list is divided into the rough same size of two parts.
- Position a list with n element needs $O(n)$
- $T(n)$ is the time taken to sort n elements

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

$$
\begin{aligned}
&<= cn + 2(cn/2 + 2T(n/4)) \\
&\ldots \\
&<= cn \log n + nT(1) = O(n \log n)
\end{aligned}
$$

Time and Space for Quick Sort

- 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: $O(n^2)$

Merge Sort

- Given two sorted lists
  (list[i], …, list[m])
  (list[m+1], …, list[n])
  generate a single sorted list
  (sorted[i], …, sorted[n])
- O(n) space vs. O(1) space

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];
}
```

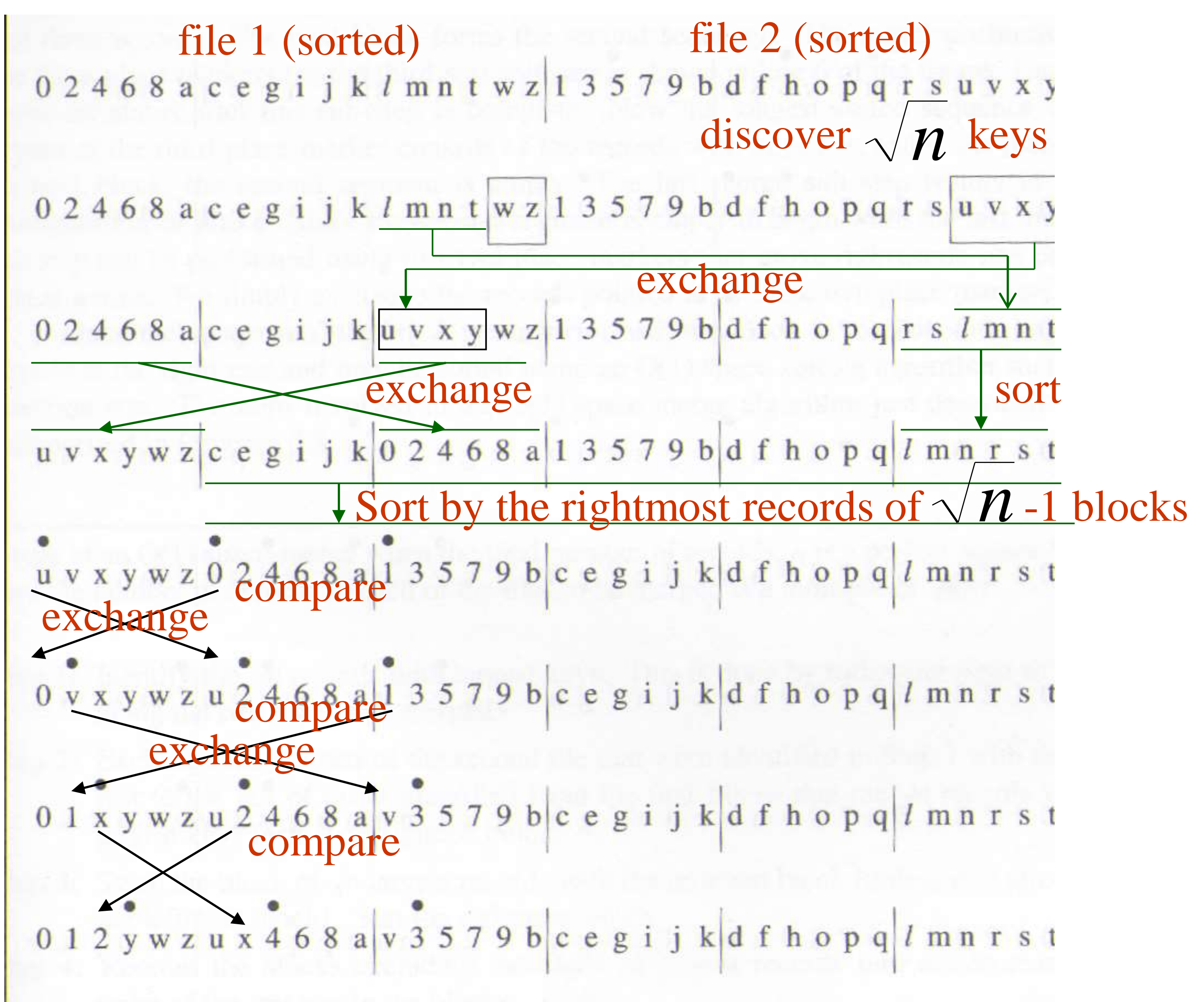
addition space: n-i+1

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

Analysis

- array vs. linked list representation
  - array: O(M(n-i+1)) where M: record length
    for copy
  - linked list representation: O(n-i+1)
    (n-i+1) linked fields

*Figure 7.5: First eight lines for O(1) space merge example (p337)



0 1 2 y w z u x 4 6 8 a v 3 5 7 9 b c e g i j k d f h o p q l m n r s t

0 1 2 3 w z u x 4 6 8 a v y 5 7 9 b c e g i j k d f h o p q l m n r s t

0 1 2 3 4 z u x w 6 8 a v y 5 7 9 b c e g i j k d f h o p q l m n r s t

0 1 2 3 4 5 u x w 6 8 a v y z 7 9 b c e g i j k d f h o p q l m n r s t

*Figure 7.6: Last eight lines for O(1) space merge example(p.338)

0 1 2 3 4 5 u x w 6 8 a | v y z 7 9 b | c e g i j k | d f h o p q | *l* m n r s t

6, 7, 8 are merged

0 1 2 3 4 5 6 7 8 u w a | v y z x 9 b | c e g i j k | d f h o p q | *l* m n r s t

Segment one is merged (i.e., 0, 2, 4, 6, 8, a)

0 1 2 3 4 5 6 7 8 9 a w | v y z x u b | c e g i j k | d f h o p q | *l* m n r s t

Change place marker (longest sorted sequence of records)

0 1 2 3 4 5 6 7 8 9 a w v y z x u b c e g i j k | d f h o p q | *l* m n r s t

Segment one is merged (i.e., b, c, e, g, i, j, k)

0 1 2 3 4 5 6 7 8 9 a b c d e f g h i j k v z u | y x w o p q | *l* m n r s t

Change place marker

0 1 2 3 4 5 6 7 8 9 a b c d e f g h i j k v z u y x w o p q | *l* m n r s t

Segment one is merged (i.e., o, p, q)

0 1 2 3 4 5 6 7 8 9 a b c d e f g h i j k *l* m n o p q y x w | v z u r s t

No other segment. Sort the largest keys.

0 1 2 3 4 5 6 7 8 9 a b c d e f g h i j k *l* m n o p q r s t | v z u y x w

***Program 7.8:** O(1) space merge (p.339)

Steps in an O(1) space merge when the total number of records, n is a perfect square */
and the number of records in each of the files to be merged is a multiple of $\sqrt{n}$ */

Step1: Identify the $\sqrt{n}$ records with largest key. This is done by following right to left along the two files to be merged.

Step2: Exchange the records of the second file that were identified in Step1 with those just to the left of those identified from the first file so that the $\sqrt{n}$ record with largest keys form a contiguous block.

Step3: Swap the block of $\sqrt{n}$ largest records with the leftmost block (unless it is already the leftmost block). Sort the rightmost block.

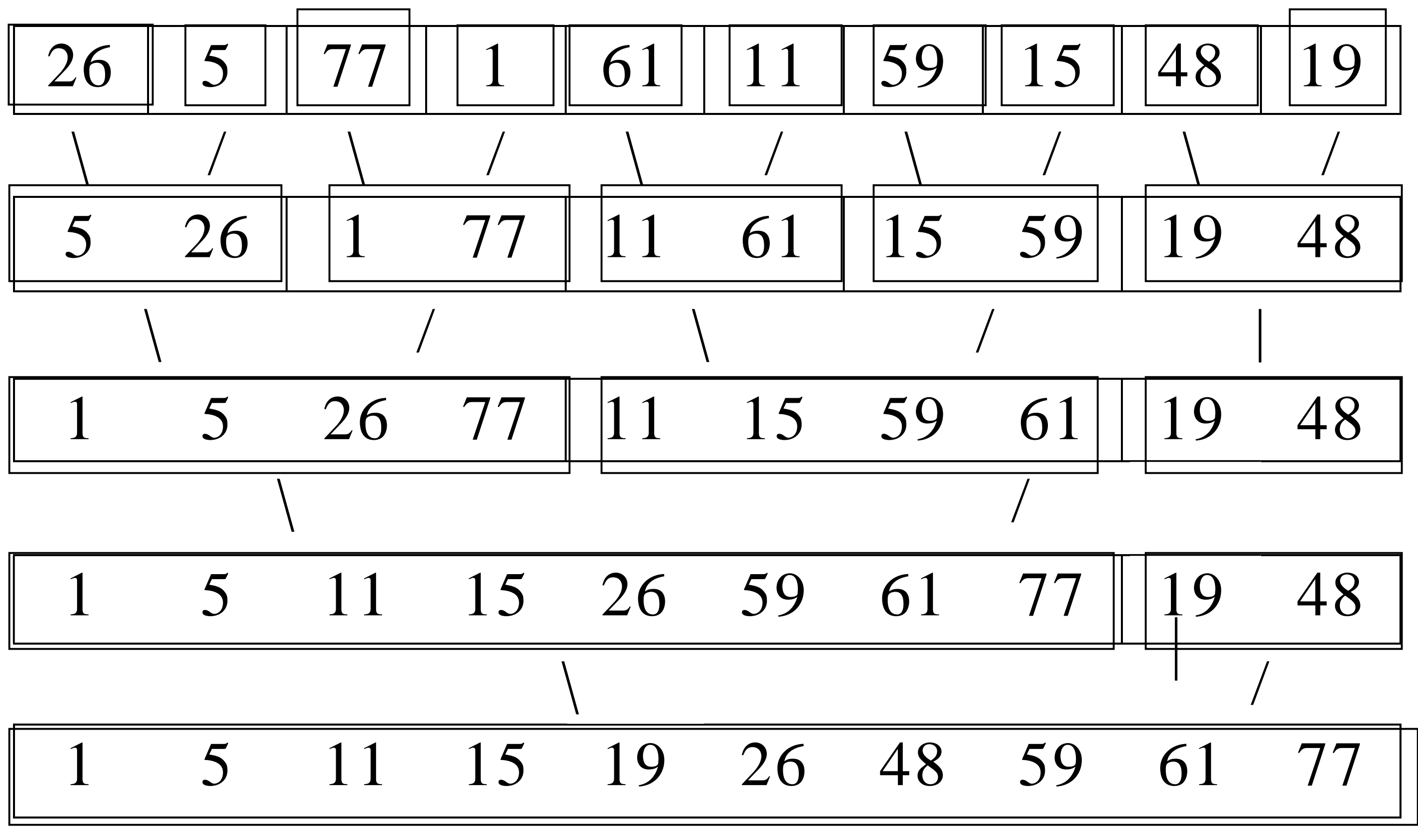
Step4: Reorder the blocks excluding the block of largest records into nondecreasing order of the last key in the blocks.

Step5: Perform as many merge sub steps as needed to merge the $\sqrt{n}-1$ blocks other than the block with the largest keys.

Step6: Sort the block with the largest keys.

Interactive Merge Sort

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



O(nlog$_2$n): $\lceil \log_2 n \rceil$ passes, O(n) for each pass

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*lenght-1);
  if (i+length<n)
    merge(list, sorted, i, i+lenght-1, n-1);
  else
    for (j=i; j<n; j++) sorted[j]= list[j];
}
```

One complement segment and one partial segment

Only one segment

i   i+length-1   i+2length-1

		...	

2*length

	...	

Merge_Sort

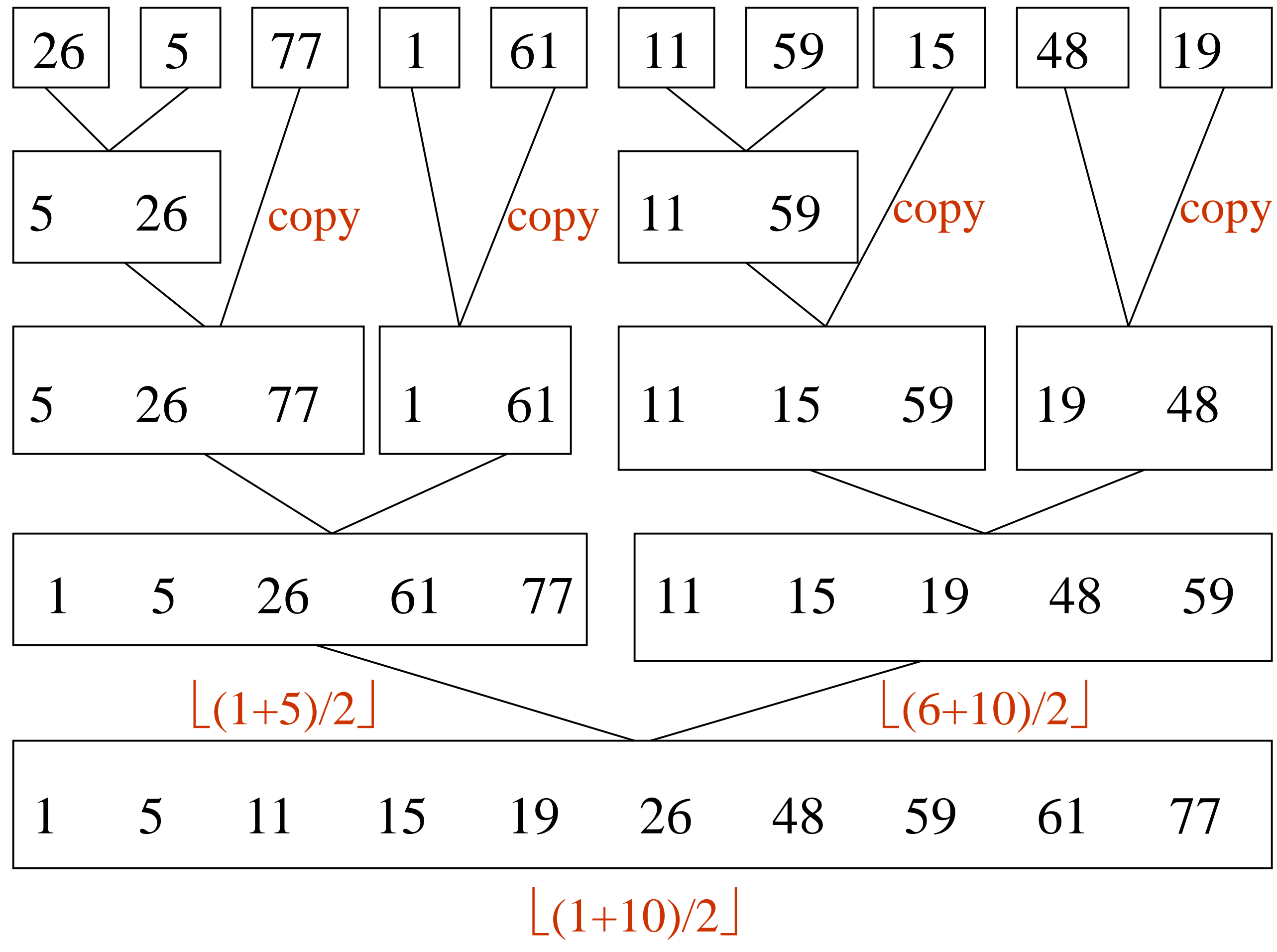
```
void merge_sort(element list[], int n)
{
    int length=1;
    element extra[MAX_SIZE];

    while (length<n) {
        merge_pass(list, extra, n, length);
        length *= 2;
        merge_pass(extra, list, n, length);
        length *= 2;
    }
}
```

1 1 1 1 ...

21 21 ...

41 ...

Recursive Formulation of Merge Sort



Data Structure: array (copy subfiles) vs. linked list (no copy)

***Figure 7.9:** Simulation of merge_sort (p.344)

start=3

i	R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9
key	26	5	77	1	61	11	59	15	48	19
link	8	5	-1	1	2	7	4	9	6	0

Recursive Merge Sort

lower upper

Point to the start of sorted chain

```
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

ListMerge

```
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;
      }
```

first

second

...

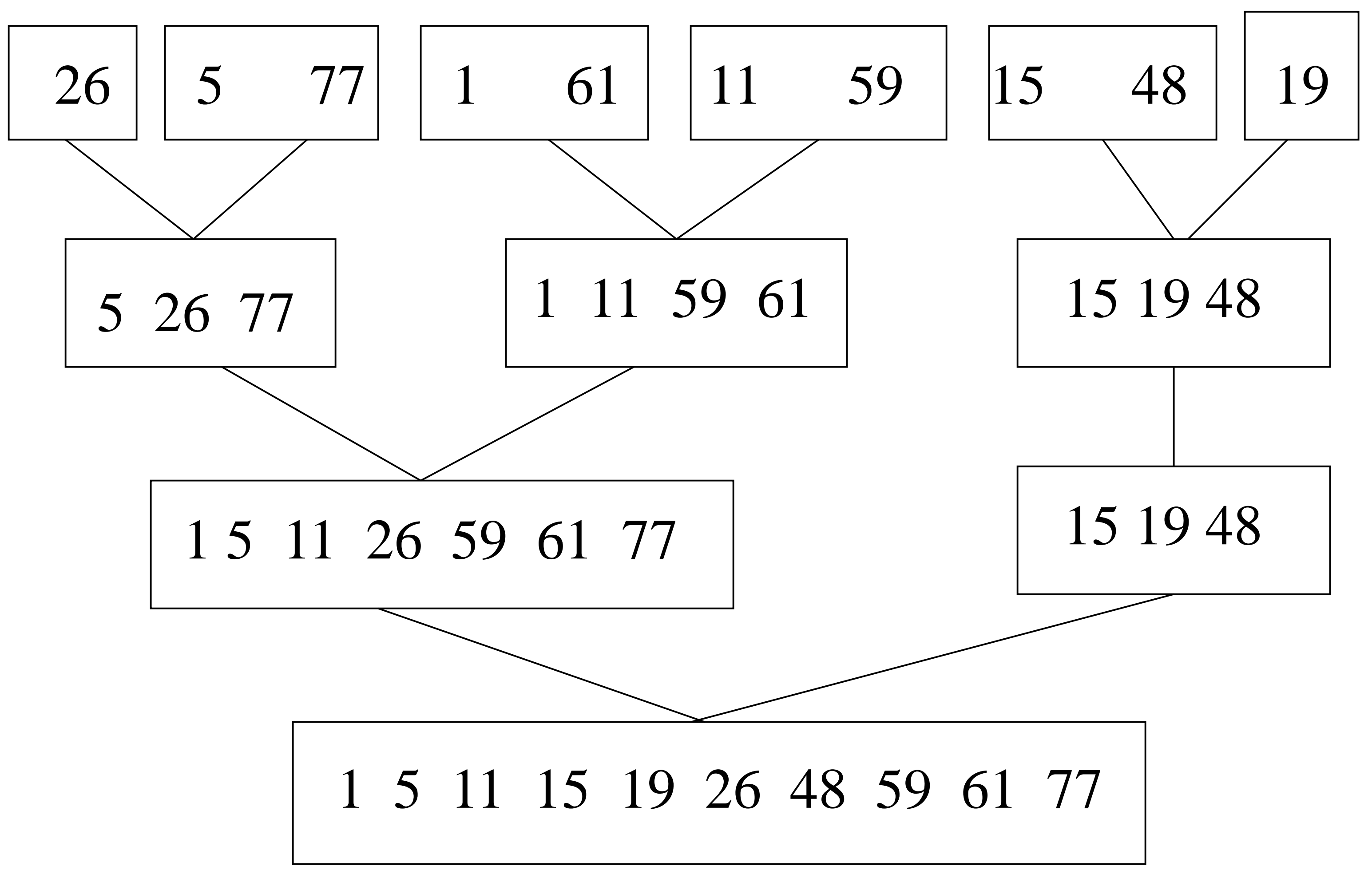
```
    else {
  /* key in second list is lower, link this
     element into the partially sorted list */
      list[start].link = second;
      start = second;
      second = list[second].link;
    }
  }
  if (first==-1)
    list[start].link = second;
  else
    list[start].link = first;
  return list[n].link;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

first is exhausted.

second is exhausted.

$O(n\log_2 n)$

Nature Merge Sort



Heap Sort

***Figure 7.11:** Array interpreted as a binary tree (p.349)

1	2	3	4	5	6	7	8	9	10
26	5	77	1	61	11	59	15	48	19

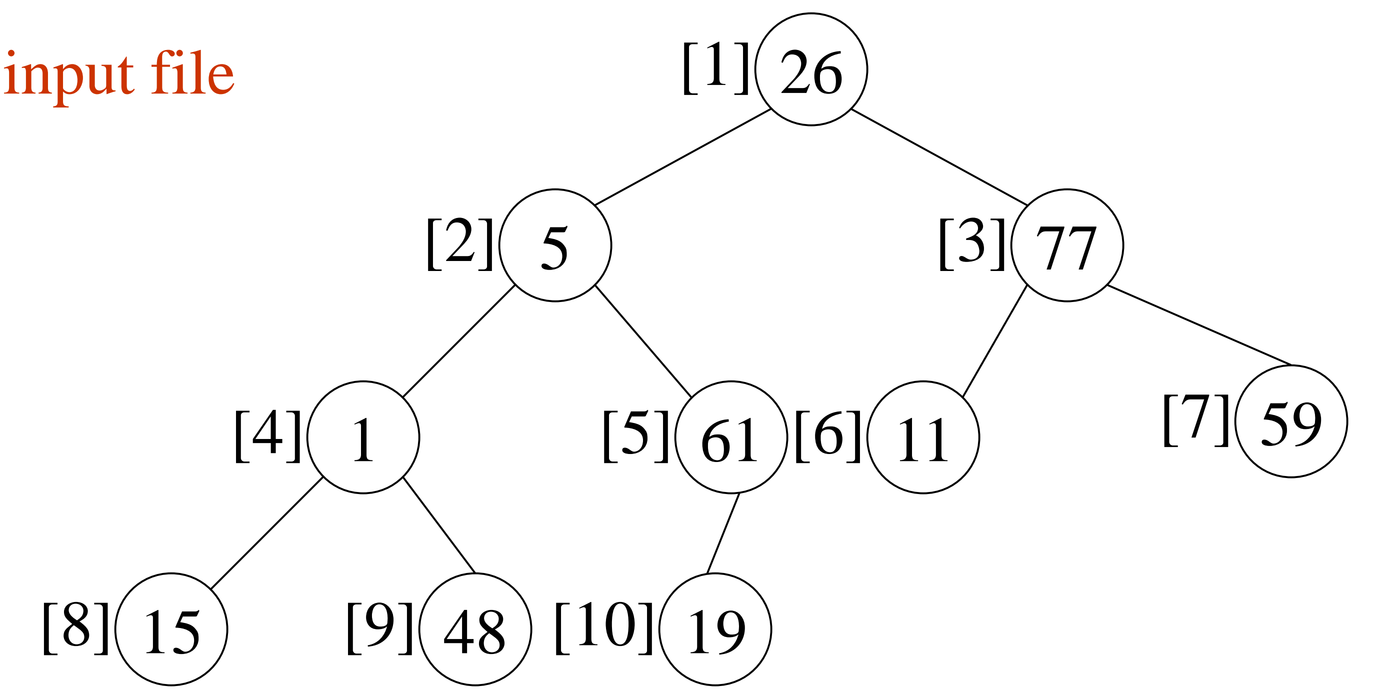


*Figure 7.12: Max heap following first **for** loop of *heapsort*(p.350)

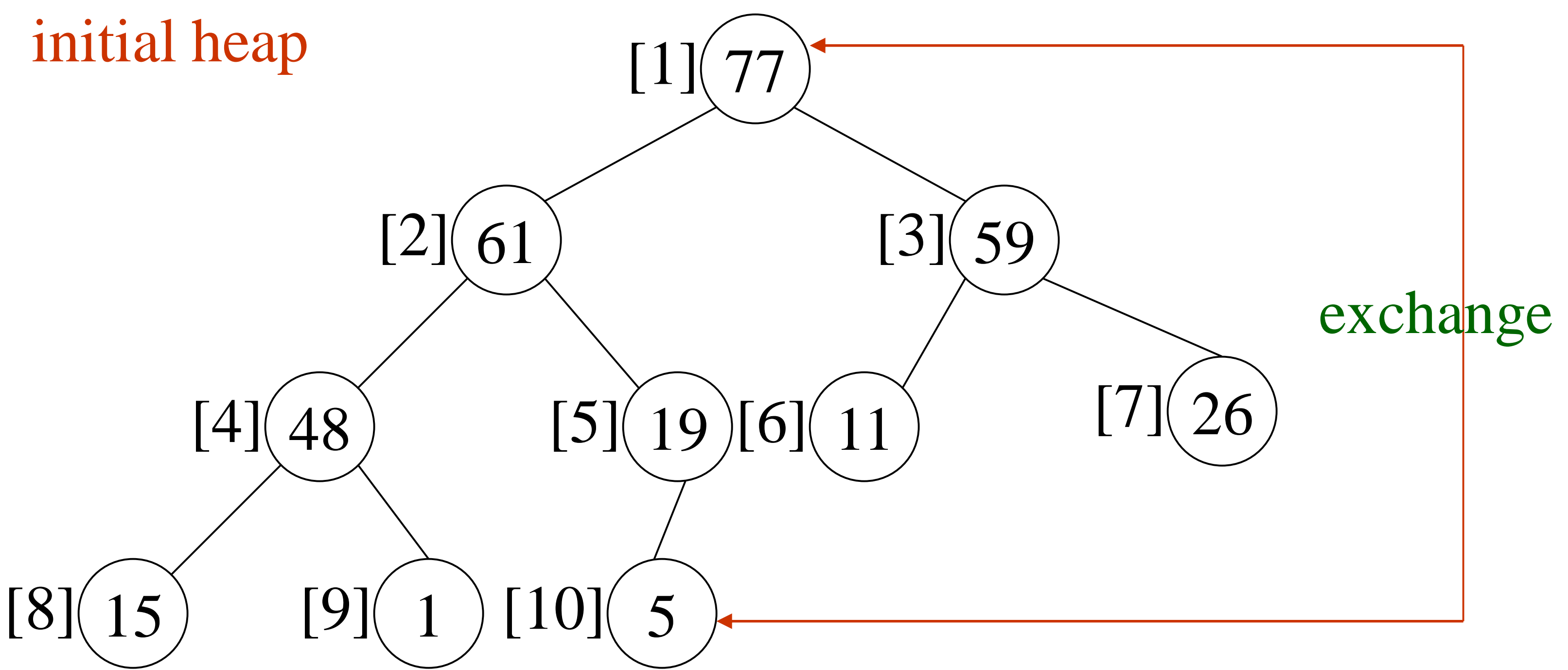


Figure 7.13: Heap sort example(p.351)

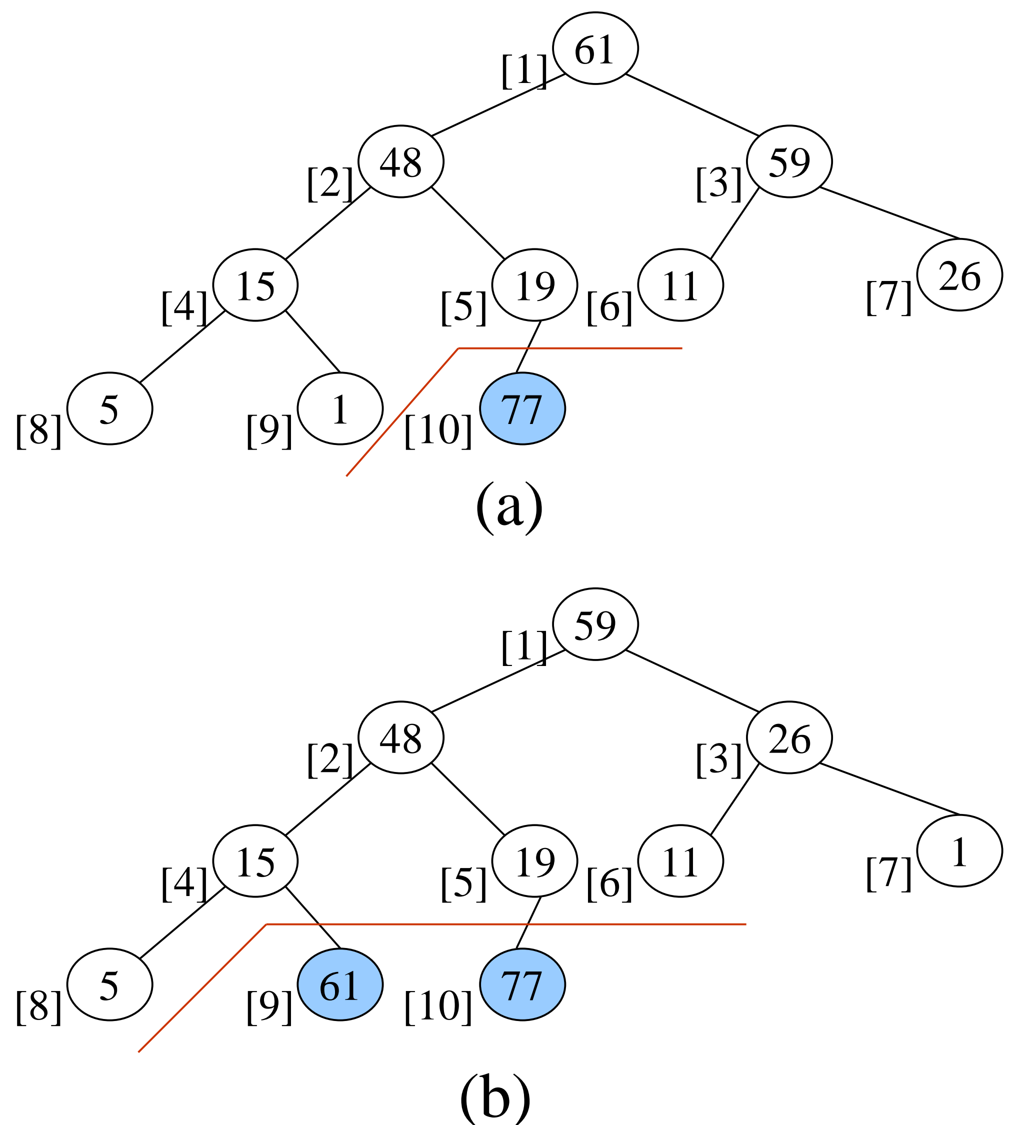
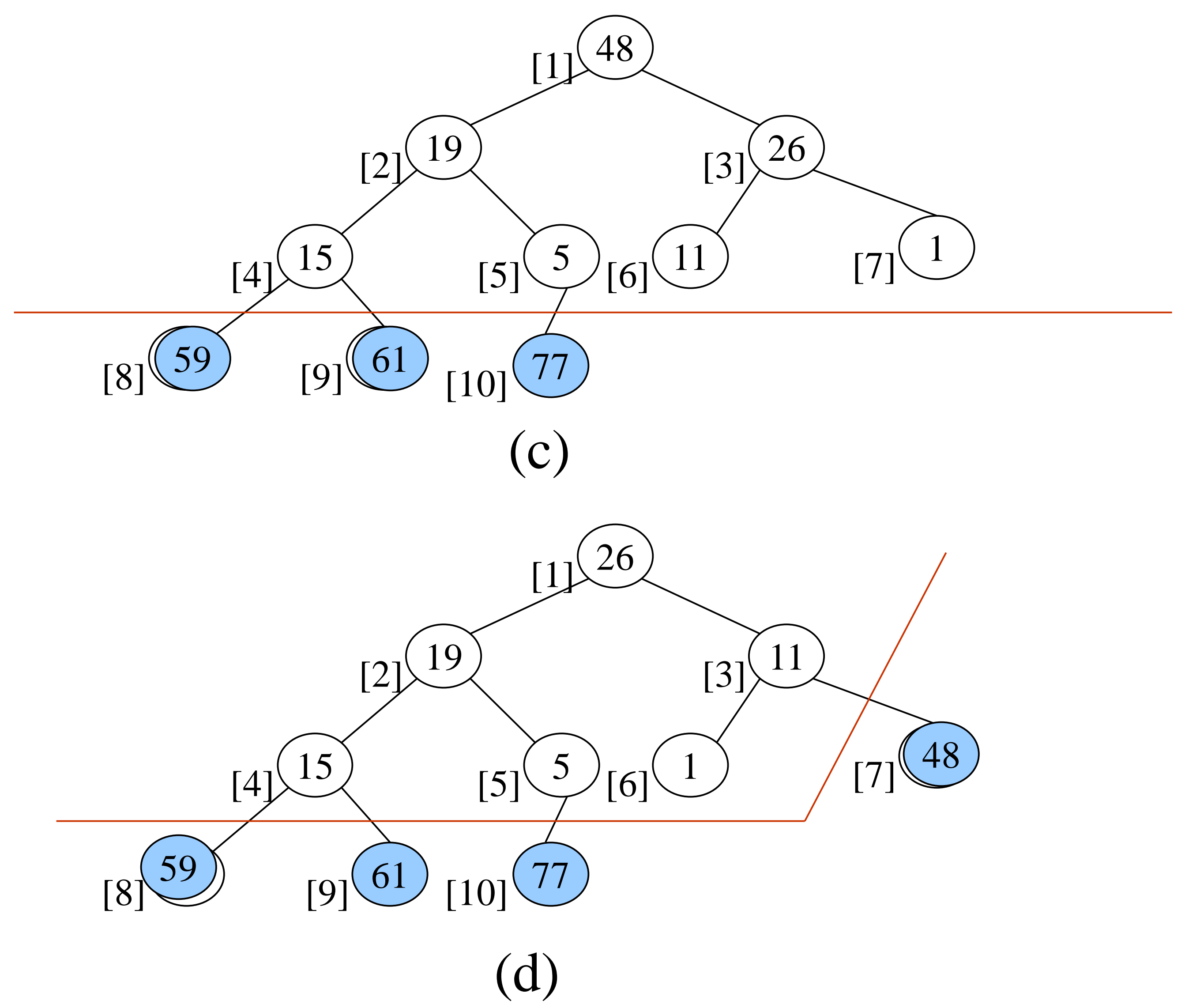


Figure 7.13(continued): Heap sort example(p.351)



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;
}
```

i

2i

2i+1

Heap Sort

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

Radix Sort

Sort by keys

K^0, K^1, …, K^{r-1}

Most significant key (points to K^0)

Least significant key (points to K^{r-1})

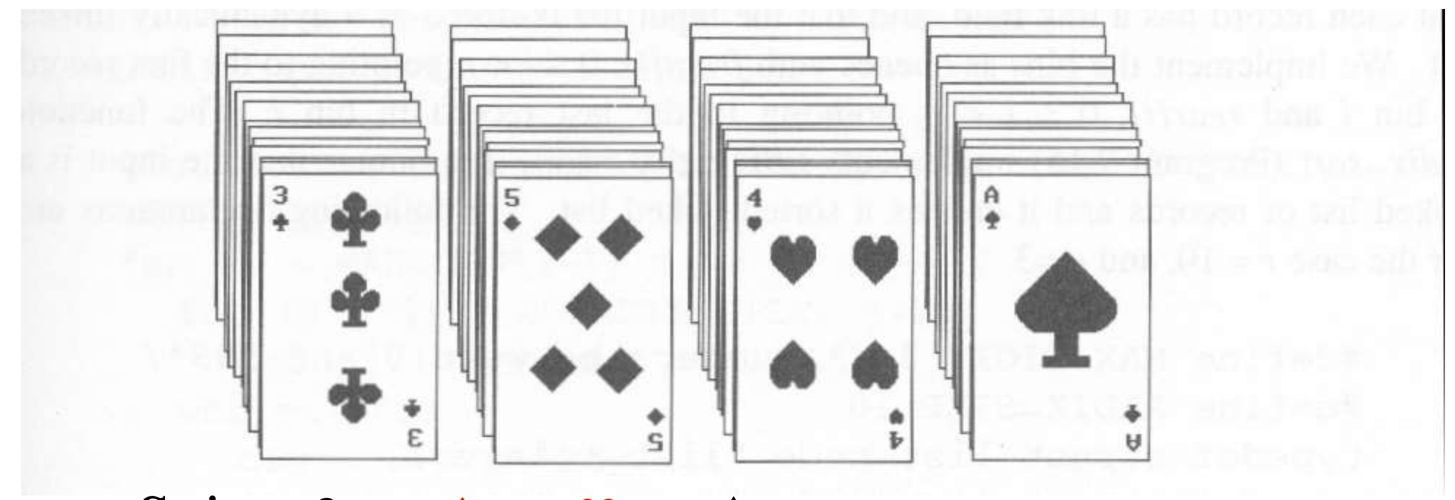
$R_0, R_1, \ldots, R_{n-1}$ are said to be sorted w.r.t. $K_0, K_1, \ldots, K_{r-1}$ iff

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

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

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

Figure 7.14: 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 ♣ ♦ ♥ ♠

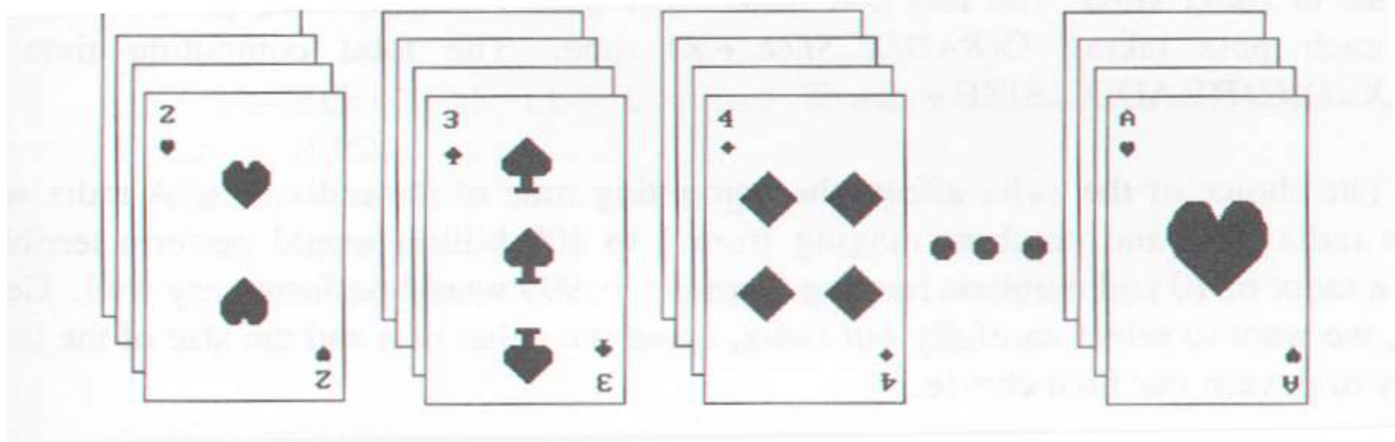
LSD sort second, e.g., insertion sort

(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 ♣ ♦ ♥ ♠

Figure 7.15: Arrangement of cards after first pass of LSD sort (p.353)



Radix Sort

$0 \leq K \leq 999$

$(K^0, \quad K^1, \quad K^2)$

MSD		LSD
0-9	0-9	0-9

radix 10 sort

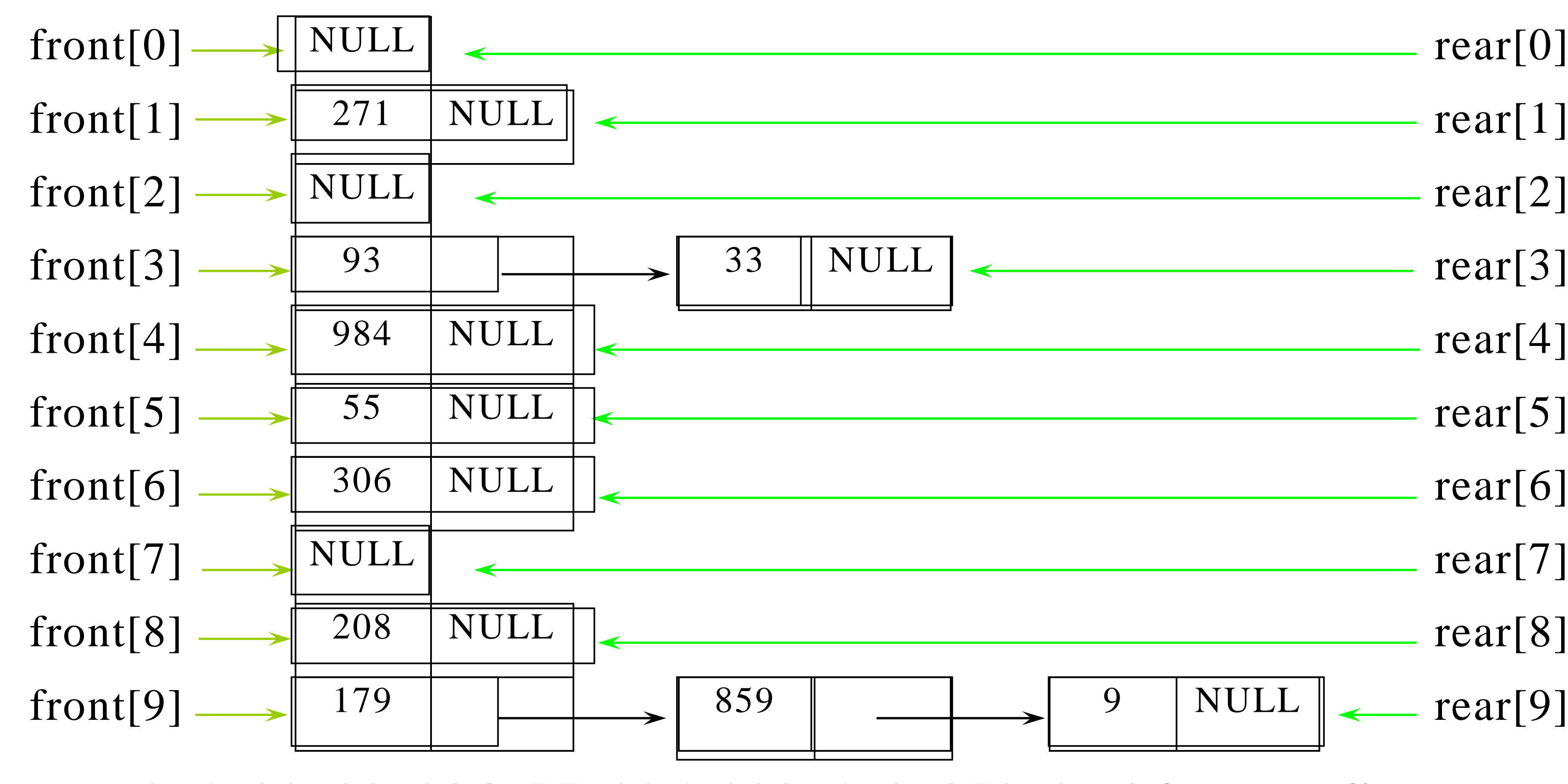
radix 2 sort

Example for LSD Radix Sort

d (digit) = 3, r (radix) = 10 ascending order

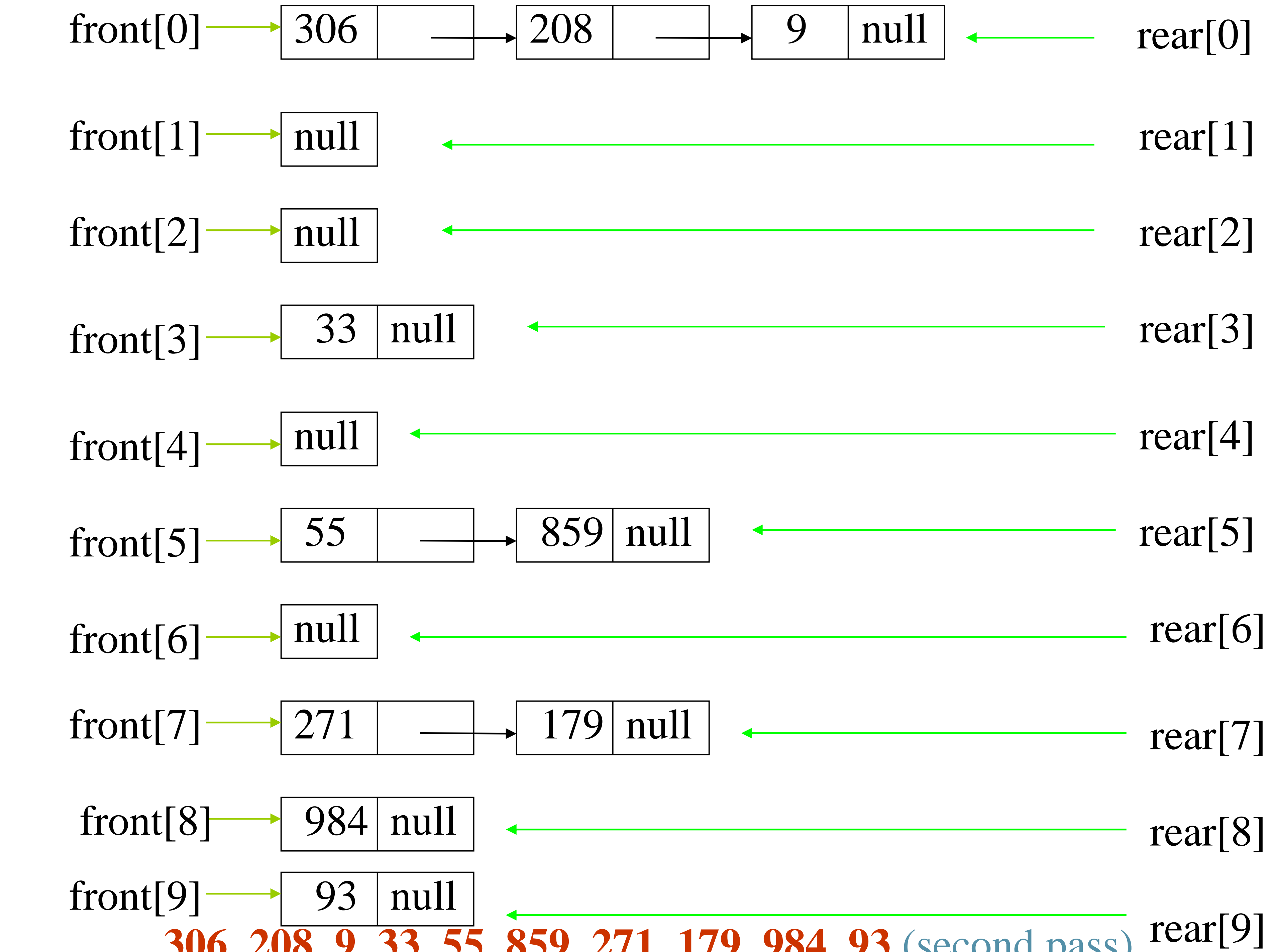
179, 208, 306, 93, 859, 984, 55, 9, 271, 33

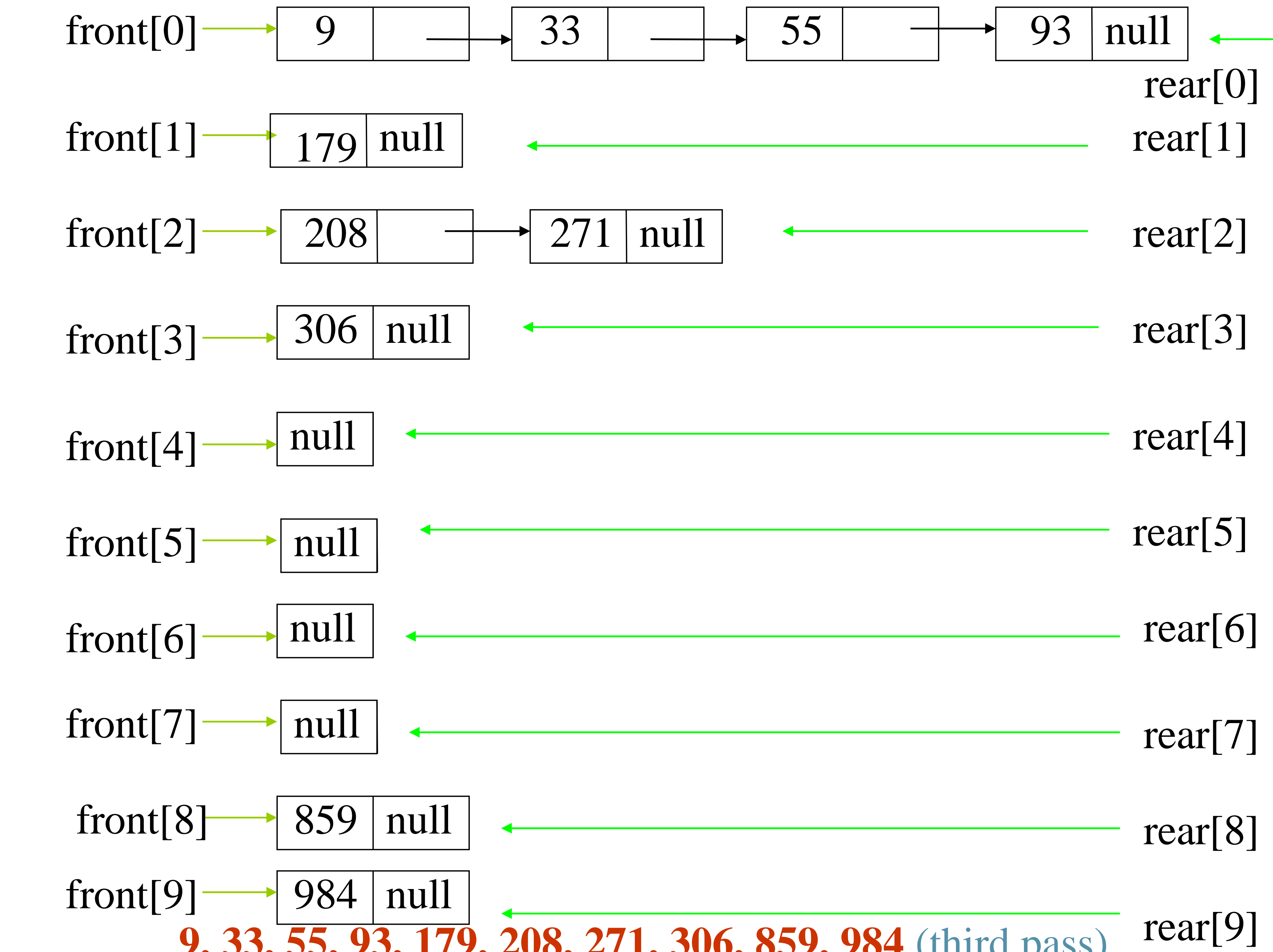
Sort by digit



concatenate

271, 93, 33, 984, 55, 306, 208, 179, 859, 9 After the first pass


front[0]
306
208
9
null
rear[0]
front[1]
null
rear[1]
front[2]
null
rear[2]
front[3]
33
null
rear[3]
front[4]
null
rear[4]
front[5]
55
859
null
rear[5]
front[6]
null
rear[6]
front[7]
271
179
null
rear[7]
front[8]
984
null
rear[8]
front[9]
93
null
rear[9]
306, 208, 9, 33, 55, 859, 271, 179, 984, 93 (second pass)


front[0]
9
33
55
93
null
rear[0]
front[1]
179
null
rear[1]
front[2]
208
271
null
rear[2]
front[3]
306
null
rear[3]
front[4]
null
rear[4]
front[5]
null
rear[5]
front[6]
null
rear[6]
front[7]
null
rear[7]
front[8]
859
null
rear[8]
front[9]
984
null
rear[9]
9, 33, 55, 93, 179, 208, 271, 306, 859, 984 (third pass)

Data Structures for LSD Radix Sort

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

```
#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;
}
```

LSD Radix Sort

```
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]=read[j]=NULL;
    while (ptr) {
      digit=ptr->key[I];
      if (!front[digit]) front[digit]=ptr;
      else rear[digit]->link=ptr;
```

Initialize bins to be empty queue.

Put records into queues.

```
        rear[digit]=ptr;
        ptr=ptr->link;        Get next record.
      }
      /* reestablish the linked list for the next pass */
      ptr= NULL;
      for (j=RADIX_SIZE-1; j>=0; j++)
        if (front[j]) {
          rear[j]->link=ptr;
          ptr=front[j];
        }
    }
    return ptr;
}
```

O(n)

O(d(n+r))

O(r)

Practical Considerations for Internal Sorting

- Data movement
  - slow down sorting process
  - insertion sort and merge sort --> linked file
- perform a linked list sort + rearrange records

List and Table Sorts

- Many sorting algorithms require excessive data movement since we must physically move records following some comparisons
- If the records are large, this slows down the sorting process
- We can reduce data movement by using a linked list representation

List and Table Sorts

- However, in some applications we must physically rearrange the records so that they are in the required order
- We can achieve considerable savings by first performing a linked list sort and then physically rearranging the records according to the order specified in the list.

In Place Sorting

i	R0	R1	R2	R3	R4	R5	R6	R7	R8	R9
key	26	5	77	1	61	11	59	15	48	19
link	8	5	-1	1	2	7	4	9	6	0

R0 <--> R3

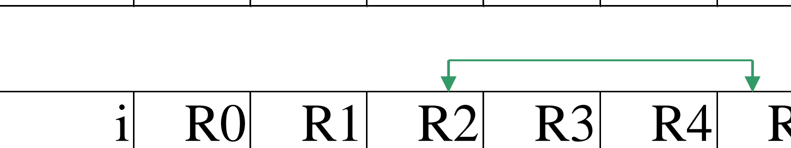
1	5	77	26	61	11	59	15	48	19
1	5	-1	8	2	7	4	9	6	3

How to know where the predecessor is ?
I.E. its link should be modified.

Doubly Linked List

i	R0	R1	R2	R3	R4	R5	R6	R7	R8	R9
key	26	5	77	1	61	11	59	15	48	19
link	8	5	-1	1	2	7	4	9	6	0
linkb	9	3	4	-1	6	1	8	5	0	7

i	R0	R1	R2	R3	R4	R5	R6	R7	R8	R9
key	1	5	77	26	61	11	59	15	48	19
link	1	5	-1	8	2	7	4	9	6	3
linkb	-1	3	4	9	6	1	8	5	3	7



i	R0	R1	R2	R3	R4	R5	R6	R7	R8	R9
key	1	5	77	26	61	11	59	15	48	19
link	1	5	-1	8	2	7	4	9	6	3
linkb	-1	3	4	9	6	1	8	5	3	7

i	R0	R1	R2	R3	R4	R5	R6	R7	R8	R9
key	1	5	77	26	61	11	59	15	48	19
link	1	5	-1	8	2	7	4	9	6	3
linkb	-1	3	4	9	6	1	8	5	3	7

Algorithm for List Sort

```
void list_sort1(element list[], int n,
                int start)
{
  int i, last, current;
  element temp;
  last = -1;
  for (current=start; current!=-1;
       current=list[current].link) {
    list[current].linkb = last;
    last = current;
  }
```

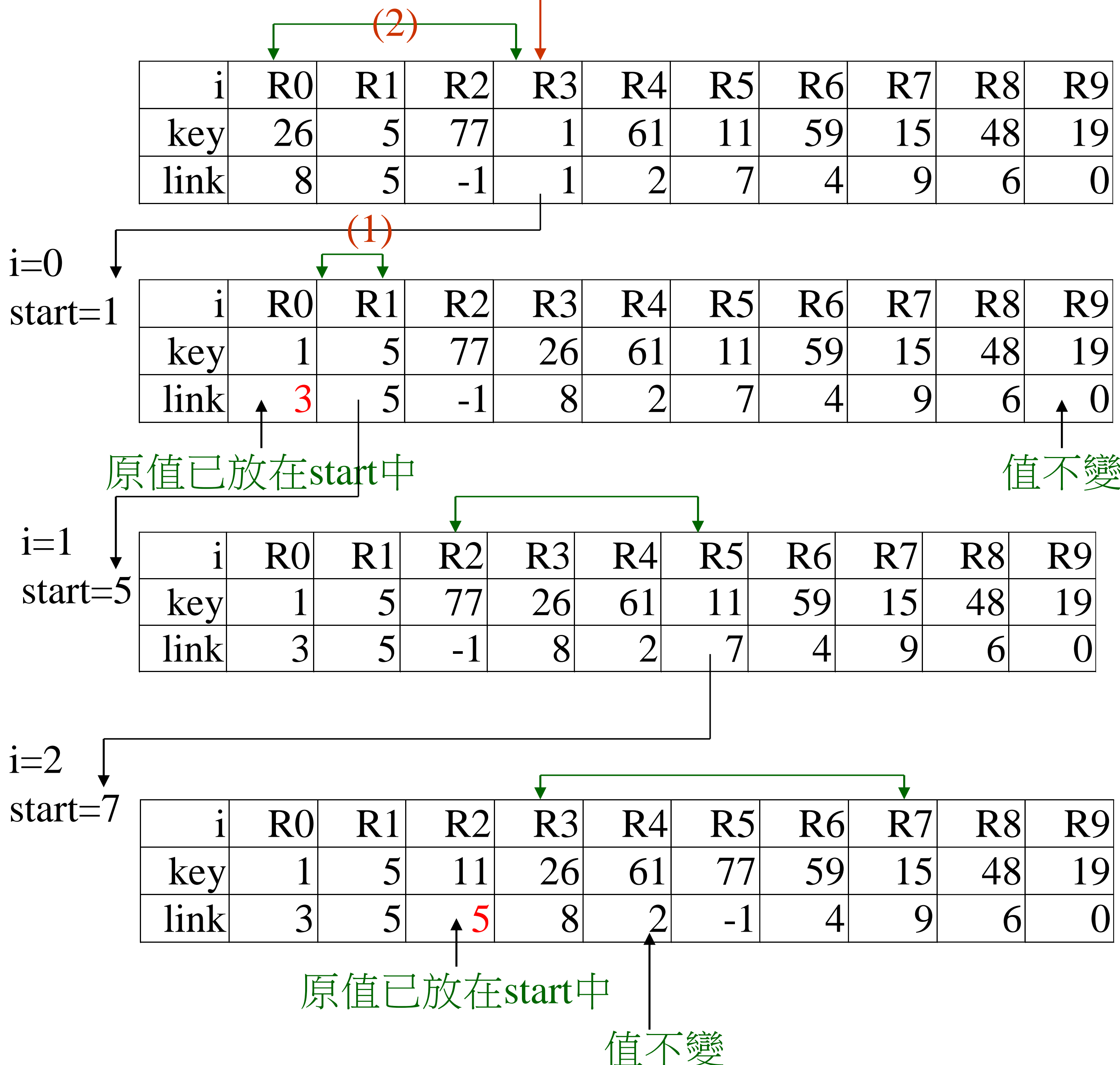
Convert start into a doubly lined list using linkb

O(n)

```
for (i=0; i<n-1; i++) {
  if (start!=i) {
    if (list[i].link+1)
      list[list[i].link].linkb= start;
    list[list[i].linkb].link= start;
    SWAP(list[start], list[i].temp);
  }
  start = list[i].link;
}
}
```

list[i].link≠-1

O(mn)



i	R0	R1	R2	R3	R4	R5	R6	R7	R8	R9
key	26	5	77	1	61	11	59	15	48	19
link	8	5	-1	1	2	7	4	9	6	0

i	R0	R1	R2	R3	R4	R5	R6	R7	R8	R9
key	1	5	77	26	61	11	59	15	48	19
link	3	5	-1	8	2	7	4	9	6	0

i	R0	R1	R2	R3	R4	R5	R6	R7	R8	R9
key	1	5	77	26	61	11	59	15	48	19
link	3	5	-1	8	2	7	4	9	6	0

i	R0	R1	R2	R3	R4	R5	R6	R7	R8	R9
key	1	5	11	26	61	77	59	15	48	19
link	3	5	5	8	2	-1	4	9	6	0

i=3
start=9

i	R0	R1	R2	R3	R4	R5	R6	R7	R8	R9
key	1	5	11	15	61	77	59	26	48	19
link	3	5	5	7	2	-1	4	8	6	0

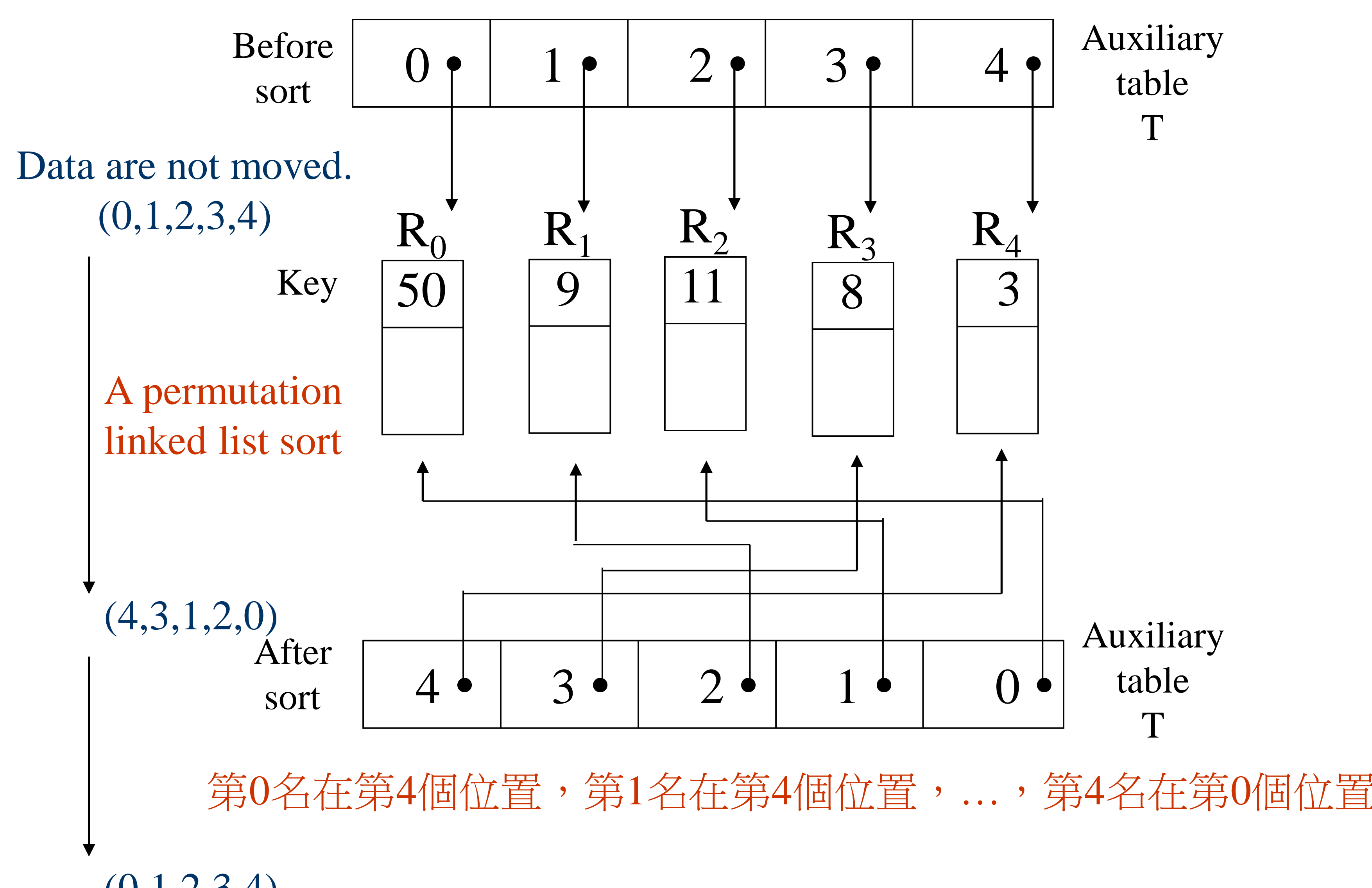
i=4
start=0

i	R0	R1	R2	R3	R4	R5	R6	R7	R8	R9
key	1	5	11	15	19	77	59	26	48	61
link	3	5	5	7	9	-1	4	8	6	2

i=5
start=8

i	R0	R1	R2	R3	R4	R5	R6	R7	R8	R9
key	1	5	11	15	19	77	59	26	48	61
link	3	5	5	7	9	-1	4	8	6	2

Table Sort

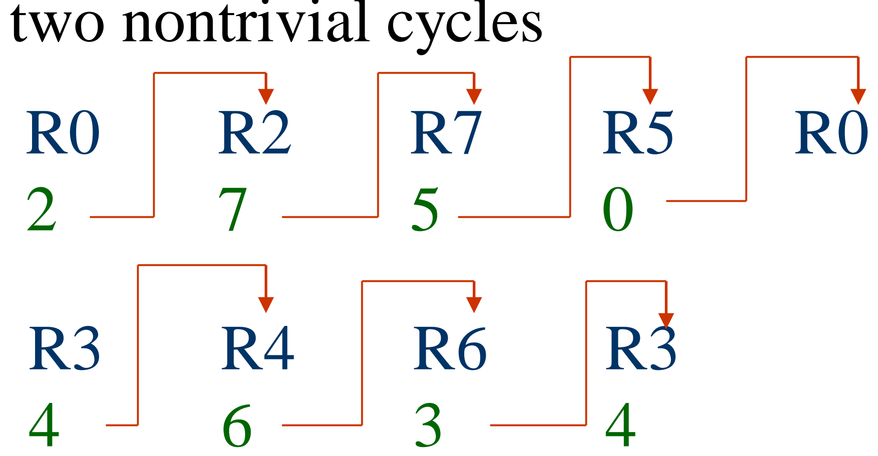


Every permutation is made of disjoint cycles.

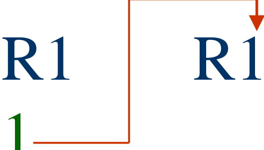
Example

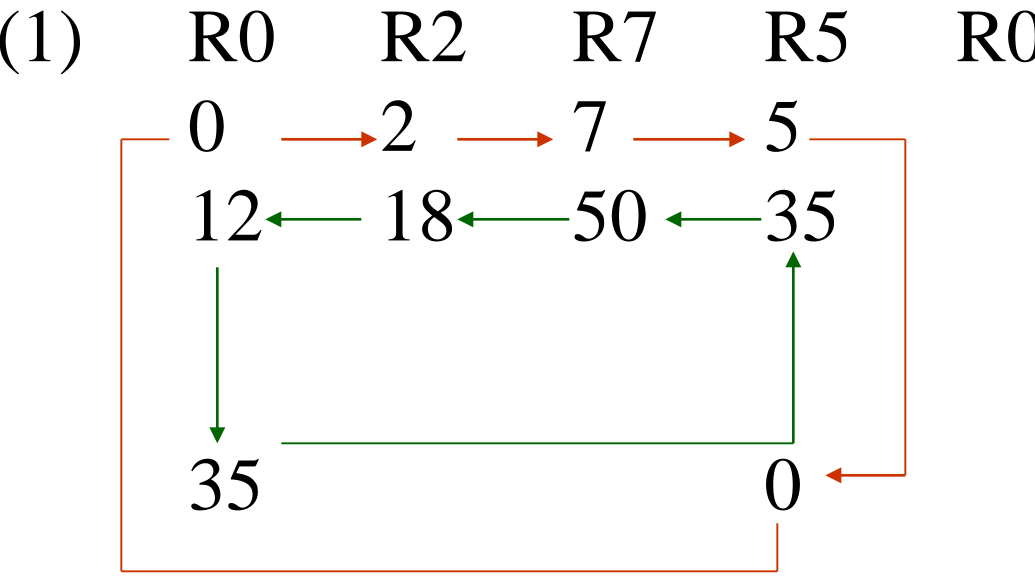
	R0	R1	R2	R3	R4	R5	R6	R7
key	35	14	12	42	26	50	31	18
table	2	1	7	4	6	0	3	5

two nontrivial cycles

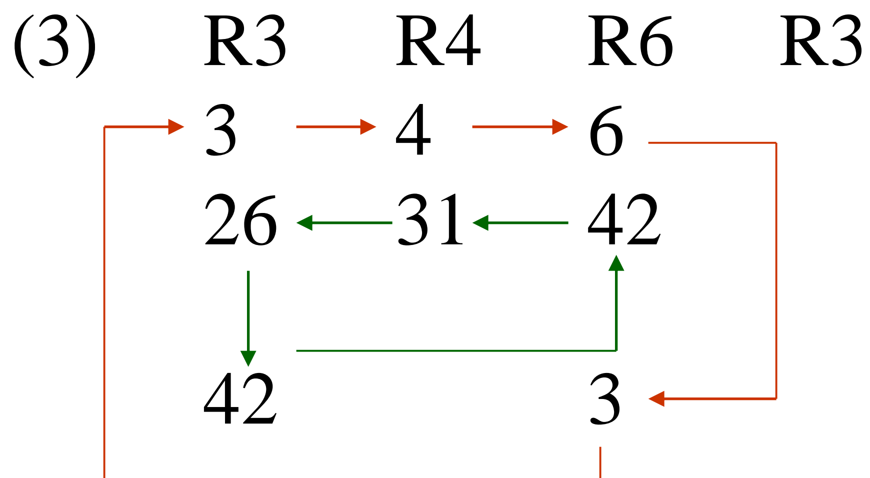


trivial cycle




(1)
R0
R2
R7
R5
R0
0
2
7
5
12
18
50
35
35
0

(2)
i=1,2
t[i]=i


(3)
R3
R4
R6
R3
3
4
6
26
31
42
42
3

Algorithm for Table Sort

```
void table_sort(element list[], int n,
                int table[])
{
    int i, current, next;
    element temp;
    for (i=0; i<n-1; i++)
       if (table[i]!=i) {
          temp= list[i];
          current= i;
```

```
            do {
                next = table[current];
                list[current] = list[next];
                table[current]= current;
                current = next;
            } while (table[current]!=i);
            list[current] = temp;
            table[current] = current;
        }
}
```

形成cycle

Complexity of Sort

	stability	space	time		
			best	average	worst
Bubble Sort	stable	little	O(n)	O(n2)	O(n2)
Insertion Sort	stable	little	O(n)	O(n2)	O(n2)
Quick Sort	untable	O(logn)	O(nlogn)	O(nlogn)	O(n2)
Merge Sort	stable	O(n)	O(nlogn)	O(nlogn)	O(nlogn)
Heap Sort	untable	little	O(nlogn)	O(nlogn)	O(nlogn)
Radix Sort	stable	O(np)	O(nlogn)	O(nlogn)	O(nlogn)
List Sort	?	O(n)	O(1)	O(n)	O(n)
Table Sort	?	O(n)	O(1)	O(n)	O(n)

Comparison

$n < 20$: insertion sort

$20 \leq n < 45$: quick sort

$n \geq 45$: merge sort

hybrid method: merge sort + quick sort
merge sort + insertion sort

External Sorting

- Very large files (overheads in disk access)
  - seek time
  - latency time
  - transmission time
- merge sort
  - phase 1
    Segment the input file & sort the segments (runs)
  - phase 2
    Merge the runs

File: 4500 records, A1, …, A4500

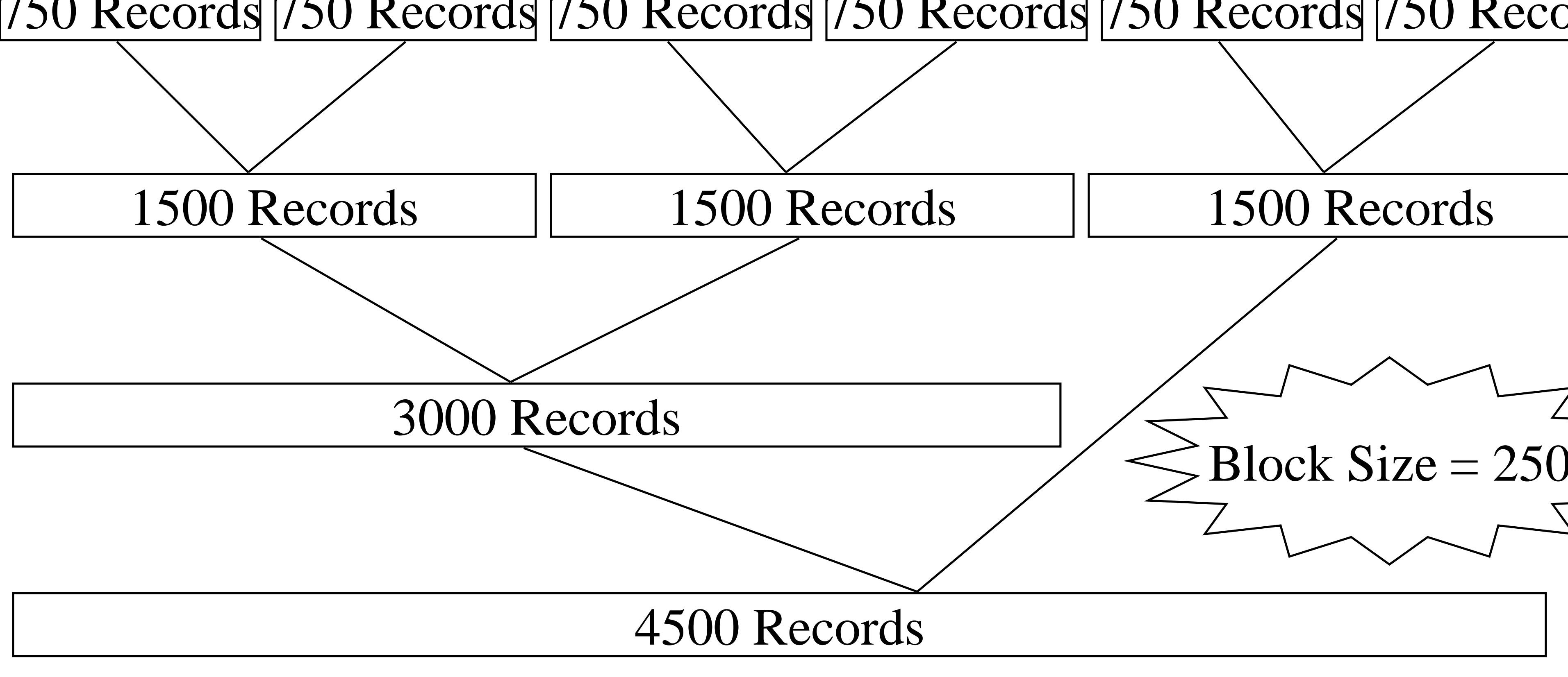
internal memory: 750 records (3 blocks)

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



2 2/3 passes

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

Time Complexity of External Sort

(Continued)

	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	$96\ t_{IO} + 12000\ t_m + 6\ t_{IS}$

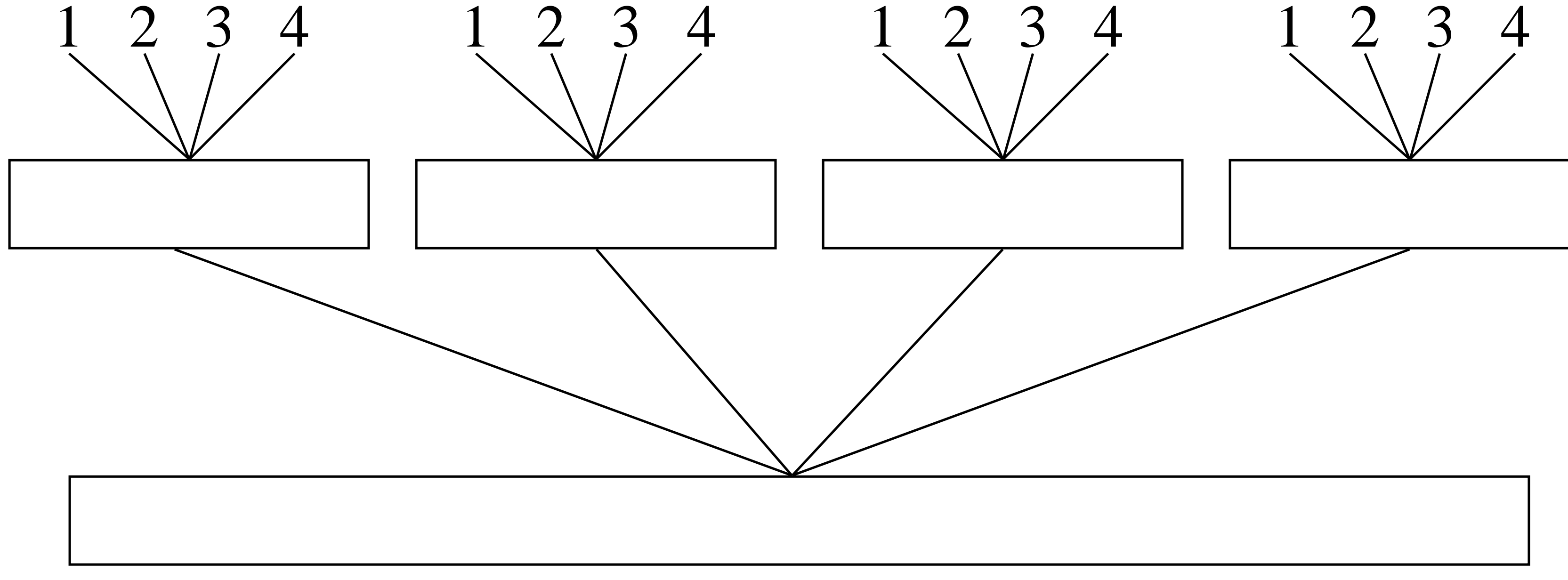
Critical factor: number of passes over the data

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

Consider Parallelism

- Carry out the CPU operation and I/O operation in parallel
- 132 t_{IO} = 12000 t_m + 6 t_{IS}
- Two disks: 132 t_{IO} is reduced to 66 t_{IO}

K-Way Merging



A 4-way merge on 16 runs

2 passes (4-way) vs. 4 passes (2-way)

Analysis

- $\lceil \log_k m \rceil$ passes
- 1 pass: $\lceil n\log_2 k \rceil$
- I/O time vs. CPU time
  - reduction of the number of passes being made over the data
  - efficient utilization of program buffers so that input, output and CPU processing is overlapped as much as possible
  - run generation
  - run merging

$O(n\log_2 k * \log_k m)$