Chapter 7:

SORTING

7.1 Searching and list verification

- Introduction
  - Often, when searching a list of records, we wish to examine the records based on some field that serves to identify the record. This field is known as a *key*.
  - The efficiency of a searching strategy depends on the assumptions we make about the arrangement of records in the list.
    - If the records are ordered by the key field, we can search the list very efficiently. On the other hand, if the records are in random order based on the key field, we must start the search at one end of the list and examine each record until we either find the desired dey or we reach the other end of the list.

- Sequential search
  - We search the list by examining the key values *list*[*0*].*key, ... , list*[*n-1*].*key*, in that order, until the correct record is located, or we have examined all the records in the list.

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

```
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;
   for (i = 0; list[i] != searchnum; i++)
      ;
   return ((i < n ) ? i : -1);
}
```

Program 7.1 Sequential search

– **Analysis of *seqsearch*:**

- The average number of comparisons for a successful search is:

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

- Binary search
  - Binary search assumes that the list is ordered on the key field such that *list*[0].*key* ≤ *list*[1]. *key* ≤ … ≤ *list*[*n*-1]. *key*.
  - This search begins by comparing *searchnum* (a search key) and *list*[*middle*].*key* where *middle*=(*n*-1)/2.
  - There are three possible outcomes:
    - ***searchnum* < *list[middle].key:*** In this case, we discard the records between *list*[*middle*] and *list*[*n*-1], and continue the search with the records between *list*[0] and *list*[*middle*].
    - ***searchnum = list[middle].key:*** In this case, the search terminates successfully.
    - ***searchnum* < *list[middle].key:*** In this case, we discard the records between *list*[0] and *list*[*middle*], and continue the search with the records between *list*[*middle+1*] and *list*[*n-1*].

```
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;
}
```

Program 7.2: Binary search

– **Analysis of *binsearch*:**

- *binsearch* makes no more than O(log *n*) comparisons.

- List verification
  - Typically, we compare lists to verify that they are identical, or to identify the discrepancies. Thus, the problem of list verification is an instance of repeatedly searching one list, using each key in the other list as the search key.
    - Example: the problem of the Internal Revenue Service (IRS).
    - Analysis of function *verify1* and *verify2*:
      - verify1 (program 7.3): O(mn)
      - verify2 (program 7.4): O(tsort(n) + tsort(m) + m + n)

        $\Rightarrow$ O(max[n log n, m log m]).

        , because it is possible to sort n records in O(n log n).

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

Program 7.3: Verifying using a sequential search

```
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
      fields 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);
}
```

Program 7.4: Fast verification of two lists

7.2 Definitions

- Two important applications of sorting:
  - An aid to search
  - matching entries in lists.
- Sorting is also used in the solution of many other more complex problems.
- Formally state the problem:

  We are given a list of records (R_0, R_1, …, R_{n-1}), in which each record, R_i, has a key value, K_i. In addition, there is an ordering relation (<) on the keys such that for any two key values x and y, either $x = y$ or $x < y$ or $y < x$. This ordering relation is transitive, that is , for any three values x, y, and z, $x < y$ and $y < z$ implies $x < z$. We define the sorting problems as finding a permutation such that $K_{\sigma(i-1)} \leq K_{\sigma(i)}$, $0 < i \leq n$-1. The desired ordering is then ($R_{\sigma(0)}$, $R_{\sigma(1)}$, …, $R_{\sigma(n)}$).

– Since a list could have several identical key values, the permutation is not unique. In some applications, we are interested in finding the unique permutation, σ_s, that has the following properties:
  - [sorted] $K_{\sigma(i\text{-}1)} \leq K_{\sigma(i)}$, for $0 < i \leq n\text{-}1$.
  - [stable] 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*.
  - Stability is only one criterion that we use to distinguish between sorting methods.

– We can characterize sorts based on both location and the sorting technique employed.
  - *Internal sort*
  - *External sort*

7.3 Insertion sort

– Concept:

The basic step in this method is to insert a record R into a sequence of ordered records, $R_1, R_2, \ldots, R_i$ ($K_1 \le K_2 \le, \ldots, \le K_i$) in such a way that the resulting sequence of size i+1 is also ordered.

```
void insertion_sort(element list[], int n)
/* perform a insertion sort on the list */
{
   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;
   }
}
```

Program 7.5: *insertion_sort*

– ***Analysis of InsertionSort:***

- The worst-case time is $O(n^2)$.
- The average time is $O(n^2)$.

– Example 7.1

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**

– Example 7.2

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**

– **Variations**

- ***Binary insertion sort:*** we can reduce the number of comparisons made in an insertion sort by replacing the sequential searching technique used in *insertion_sort* with binary search. The number of record moves remains unchanged.
- ***List insertion sort:*** The elements of the list are represented as a dynamically linked list rather than as an array. The number of record moves becomes zero becomes zero because only the link fields require adjustment. However, we must retain the sequential search used in *insertion_sort*.

7.4 Quick Sort

Algorithm:

Procedure QSORT (m, n)

/*sort records R_m, …,R_n into nondecreasing order on key K. Key K_m is arbitrarily chosen as the control key(*pivot*). Pointers i and j are used to partition the subfile so that any time $K_l \le K$, $l < j$ and $K_l \ge K$, $l > j$. It is assumed that $K_m \le K_{n+1}$ */

```
  if m < n then
            [ i ← m; j ← n+1; k ← k;
              loop
                  repeat i ← i+1 until k ≥ k;
                  repeat j ← j-1 until k ≤ k;
                  if i < j then call INTERCHANGE (R_i, R_j);
                           else exit;
              forever
              call INTERCHANGE (R_m, R_j);
              call QSORT (m, j-1);
              call QSORT (j+1, n);  ]
end QSORT

Procedure INTERCHANGE (R, S)  /* change the value of R and S */
            temp ← R;     R ← S;     S ← temp;
end INTERCHANGE
```

– Example 7.3

```
void quicksort(element list[], int left, int right)
/* sort list[left], . . . , list[right] into nondecreasing
order on the key field. list[left].key is arbitrarily
chosen as the pivot key.  It is assumed that
list[left].key ≤ list[right+1].key. */
{
   int pivot,i,j;
   element temp;
   if (left < right) {
      i = left;     j = right + 1;
      pivot = list[left].key;
      do {
      /* search for keys from the left and right sublists,
      swapping out-of-order elements until the left and
      right boundaries cross or meet */
         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);
   }
}
```

Program 7.6: *quicksort* function

R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	*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	[19	15]	26	[59	61	48	37]	3	4
1	5	11	15	19	26	[59	61	48	37]	6	9
1	5	11	15	19	26	[48	37]	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		

Figure 7.2: Simulation of *quicksort*

– **Analysis of *quicksort*:**

- The worst case behavior of this algorithm is shown to be $O(n^2)$.
- Lemma 7.1 shows that the average computing time for quick sort is $O(n \log_2 n)$.

– **Lemma 7.1:**

Let $T_{avg}(n)$ be the expected time for quicksort to sort a file with n records. Then there exists a constant k such that $T_{avg}(n) \leq kn\log_e n$ for $n \geq 2$.

– **Variation**

- ***qucksort using a median of three:*** Our version of quick sort always picked the key of the first record in the current sublist as the pivot. A better choice for this pivot is the median of the first, middle, and last keys in the current sublist. Thus, *pivot* = *median* $\{K_{left}, K_{(left+right)/2}, K_{right}\}$. For example, median {10, 5, 7} =7 and median {10, 7, 7} = 7.

7.5 Optimal sorting time

– How quickly can we hope to sort a list of n objects?
  - The best possible time is $O(n \log_2 n)$.

– **Theorem 7.1:**

  Any decision tree that sorts n distinct elements has a height of at least $\log_2(n!) + 1$.

– **Corollary:**

  Any algorithm that sorts by comparisons only must have a worst case computing time of $\Omega(n \log n)$.

– 課本這個Corollary 的寫法容易讓人誤會。建議另一寫法（The Design and Analysis of Computer Algorithms, by Aho/Hopcroft/Ullman, pp.87)

Any algorithm for sorting by comparisons requires at least $n \log n$ comparisons to sort n elements.

– Proof.

For $n > 1$

$$n! \geqq n(n-1)(n-2)\cdots\cdots\left(\frac{n}{2}\right) \geqq \left(\frac{n}{2}\right)^{n/2}$$

So, $\log n! \geqq n \log n$.

7.6 Merge sort

- Merging
  - Before looking at the merge sort algorithm to sort n records let us see how one may merge two sorted lists to get a single sorted list.
    - The first one, Program 7.7, uses O(*n*) additional space.
    - It merges the sorted lists (*list*[*i*], … , *list*[*m*]) and (*list*[*m+1*], …, *list*[*n*]), into a single sorted list, (*sorted*[*i*], … , *sorted*[*n*]).

```
void merge(element list[], element sorted[], int i, int m,
                                              int n)
/* merge two sorted files: list[i],...,list[m], and
list[m+1],..., list[n]. These files are sorted to
obtain a sorted list: sorted[i],..., sorted[n] */
{
   int j,k,t;
   j = m+1;          /* index for the second sublist */
   k = i;            /* index for the sorted list */

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

Program 7.7: Merging two sorted lists

- Iterative merge sort

  1. We assume that the input sequence has n sorted lists, each of length 1.

  2. We merge these lists pairwise to obtain *n*/2 lists of size 2.

  3. We then merge the n/2 lists pairwise, and so on, until a single list remains.

```
void merge_pass(element list[], element sorted[], int n,
                                                int length)
{
/* perform one pass of the merge sort.  It merges adjacent
pairs of subfiles from list into sorted.  n is the
number of elements in the list. length is the length of the
subfile */
   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];
}
```

Program 7.9: *merge_pass*

```
void merge_sort(element list[], int n)
/* perform a merge sort on the file */
{
   int length = 1; /* current length being merged */
   element extra[MAX_SIZE];

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

Program 7.10: *merge_sort*

- **Analysis of *merge_sort*:**
  - The total computing time is O(n log n).
- Example 7.5:

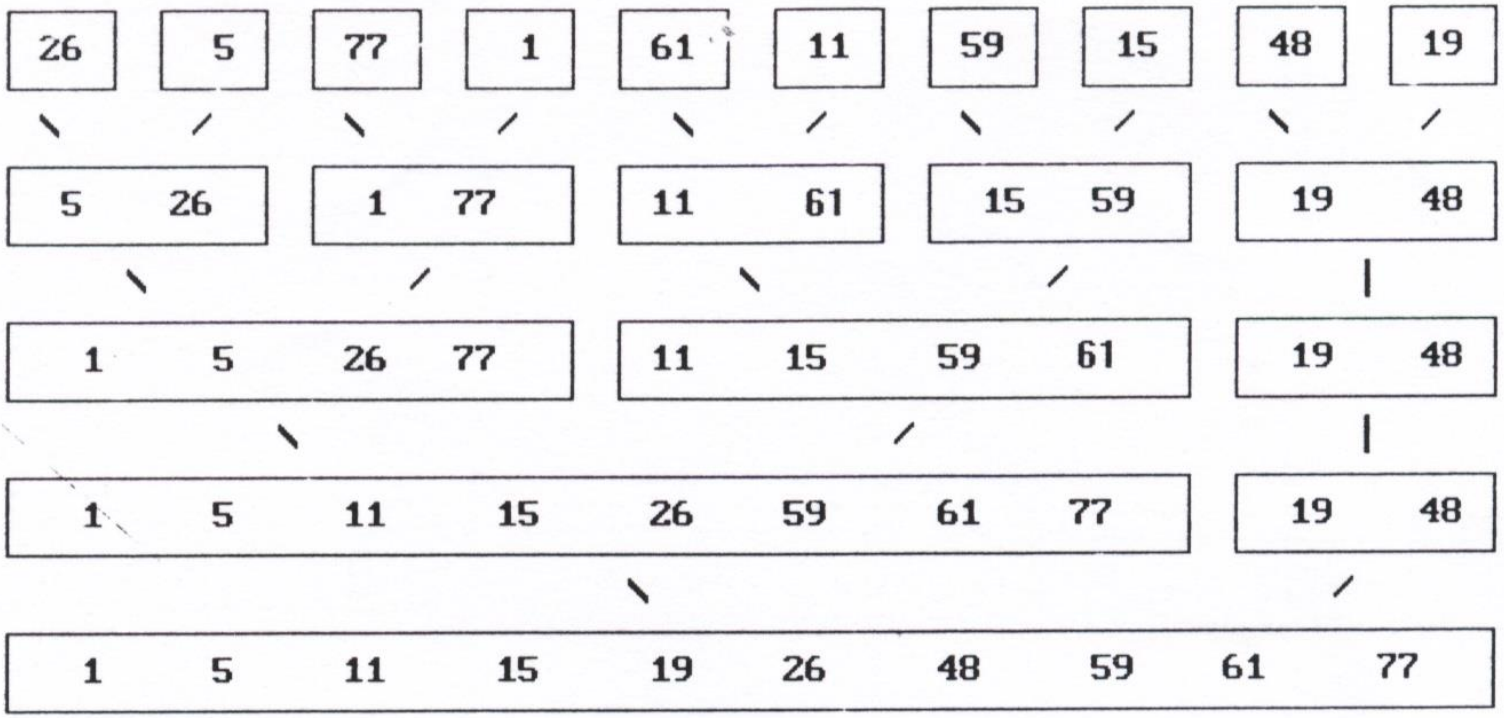


Figure 7.7: Merge tree for iterative merge sort

- Recursive merge sort
  - For the recursive version, we modify our record structure to accommodate a link field. The new structure is defined as:

```
typedef struct {
        int key;
        /* other fields */
        int link;
        } element;
```

```
int rmerge(element list[], int lower, int upper)
/* sort the list, list[lower],..., list[upper]. The link
field in each record is initially set to -1. */
{
   int middle;
   if (lower >= upper)
      return lower;
   else {
      middle = (lower + upper) / 2;
      return listmerge(list,rmerge(list,lower,middle),
                           rmerge(list,middle+1,upper));
   }
}
```

Program 7.11: Recursive merge sort

```
int listmerge(element list[], int first, int second)
/* merge lists pointed to by first and 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 second list is lower, link this element into
      the partially sorted list */
         list[start].link = second;
         start = second;
         second = list[second].link;
      }
   /* move remainder */
   if (first == -1)
      list[start].link = second;
   else
      list[start].link = first;
   return list[n].link; /* start of the new list */
}
```

Program 7.12: Merging linked lists

– **Analysis of *rmerge*:**

  - The total computing time is O(n log n).

– Example 7.6

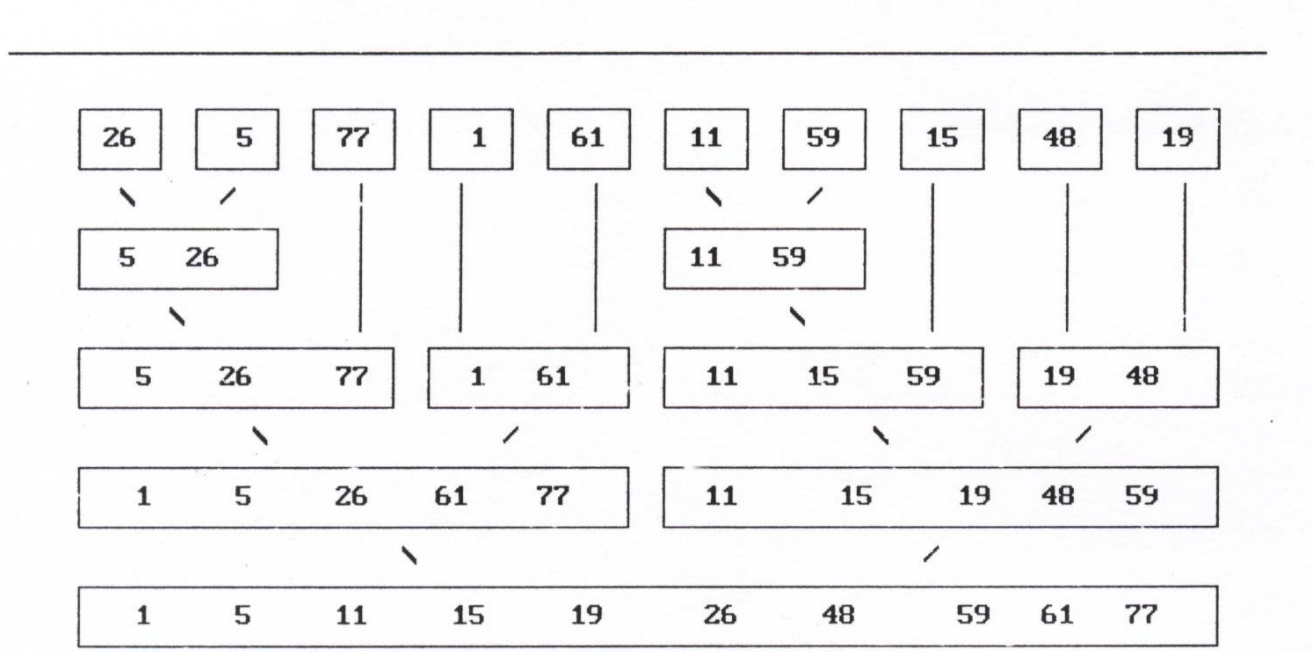


Figure 7.8: Sublist partitioning for a recursive merge sort

– **Variation:**

- ***Natural merge sort:*** We can modify *merge_sort* to take into account the prevailing order within the input list. In this implementation we make an initial pass over the data to determine the sequences of records that are in order. The merge sort then uses these initially ordered sublists for the remainder of the passes.
  - Figure 7.10 shows the results of a natural merge sort using the input sequence found in Example 7.6

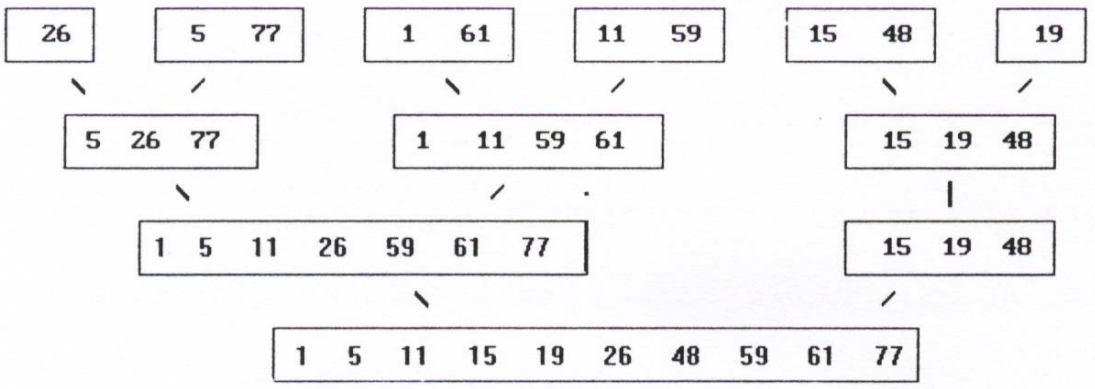


Figure 7.10: Merge sort starting with sorted sublists

7.7 Heap sort

– What advantage does the heap sort have?

- The merge sort has a computing time of O(n log n) both in the worst case and as average behavior, however, it requires additional storage proportional to the number of records in the file being sorted.

- The heap sort algorithm will require only a fixed amount of additional storage and at the same time will have as its worst case and average computing time O(n log n).

```
void adjust(element list[], int root, int n)
/* adjust the binary tree to establish the heap */
{
   int child,rootkey;
   element temp;
   temp = list[root];
   rootkey = list[root].key;
   child = 2 * root;        /* left child */
   while (child <= n) {
      if ((child < n) &&
      (list[child].key < list[child+1].key))
         child++;
      if (rootkey > list[child].key) /* compare root and
                                        max. child */
         break;
      else {
         list[child / 2] = list[child]; /* move to parent */
         child *= 2;
      }
   }
   list[child/2] = temp;
}
```

Program 7.13: Adjusting a max heap

```
void heapsort(element list[], int n)
/* perform a heapsort on the array */
{
   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);
   }
}
```

Program 7.14: Heap sort

– **Analysis of *heapsort*:**
  - The total computing time is O(n log n).
– Example 7.7

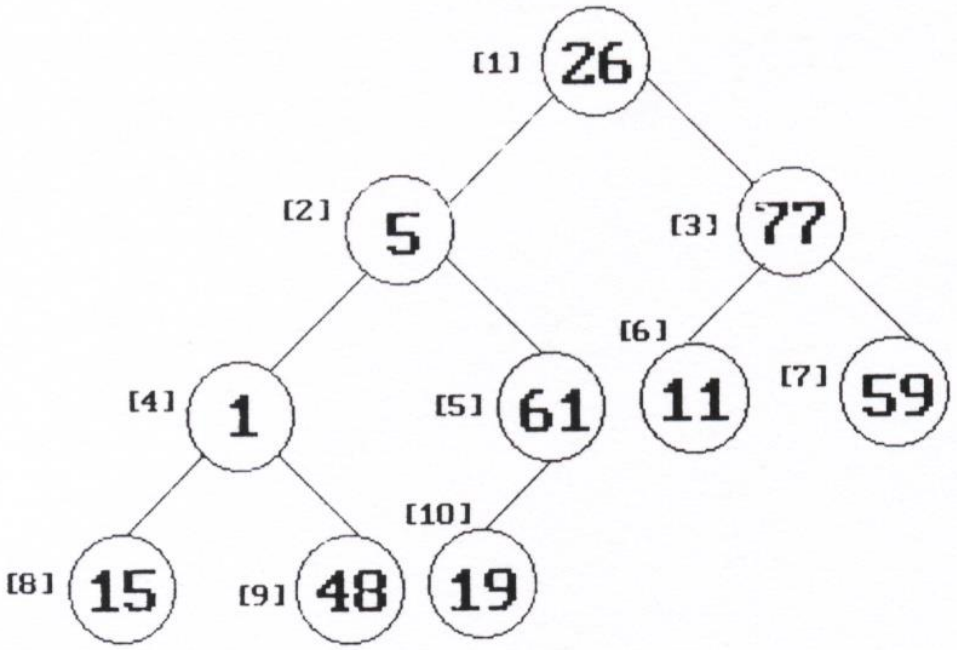


Figure 7.11: Array interpreted as a binary tree

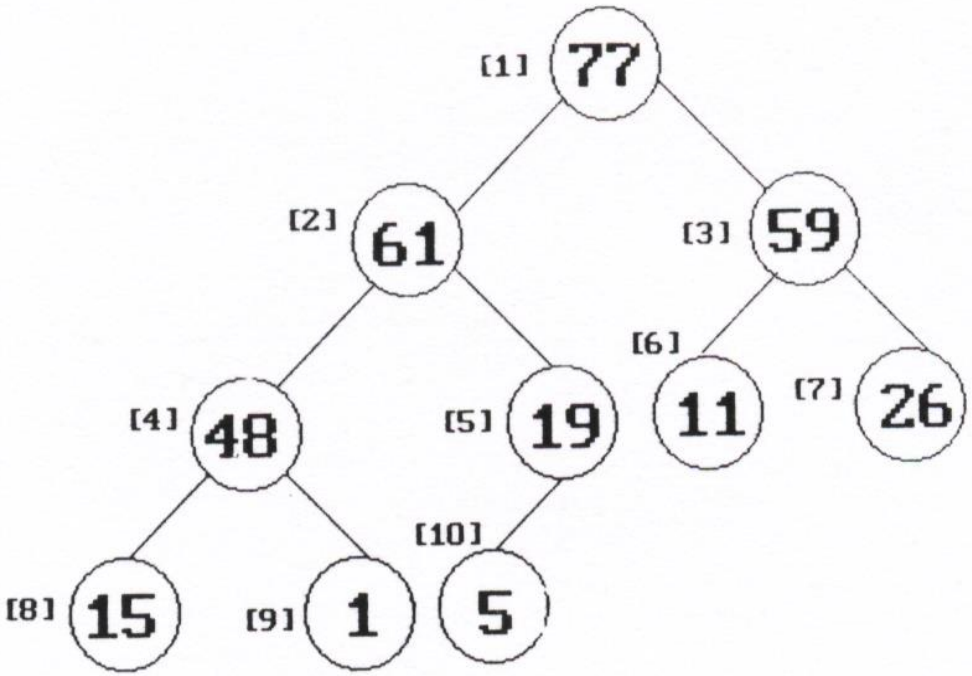
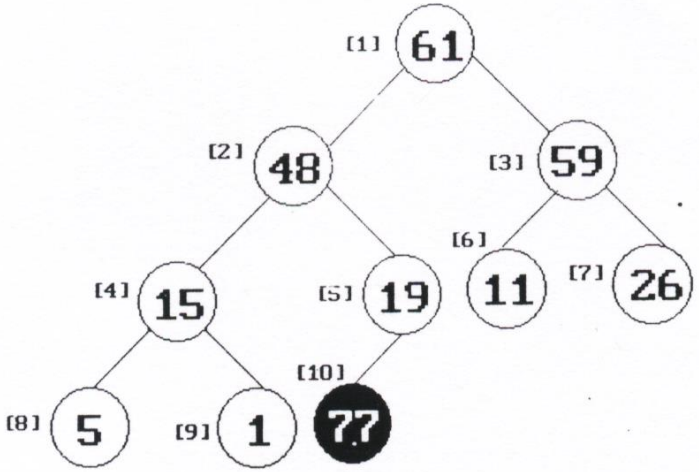
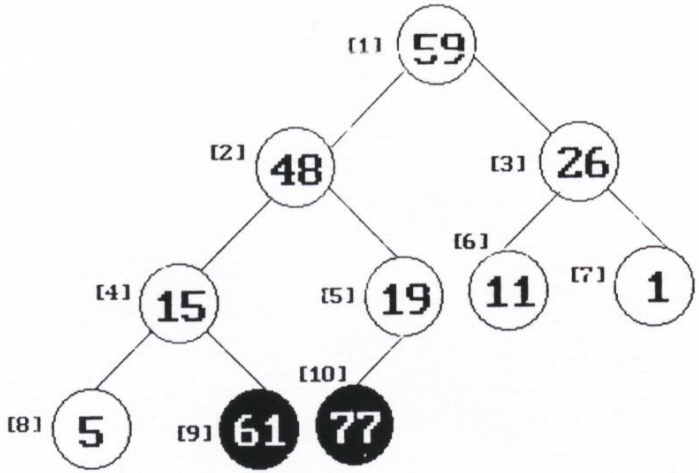


Figure 7.12: Max heap following first **for** loop of *heapsort*

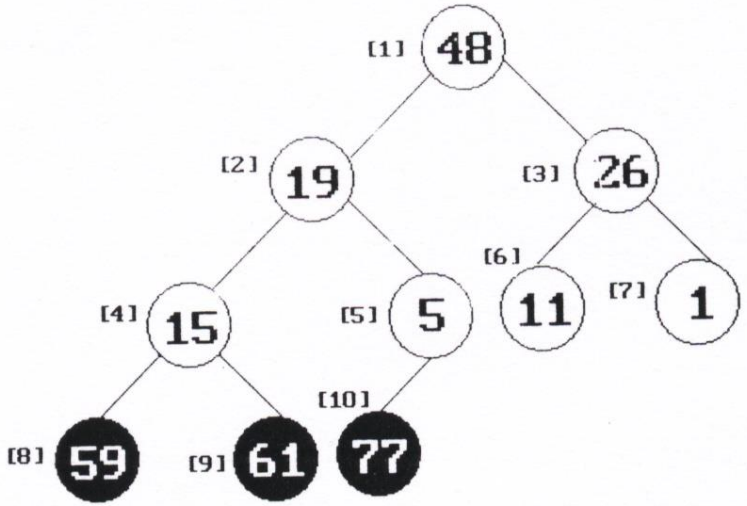


(a)

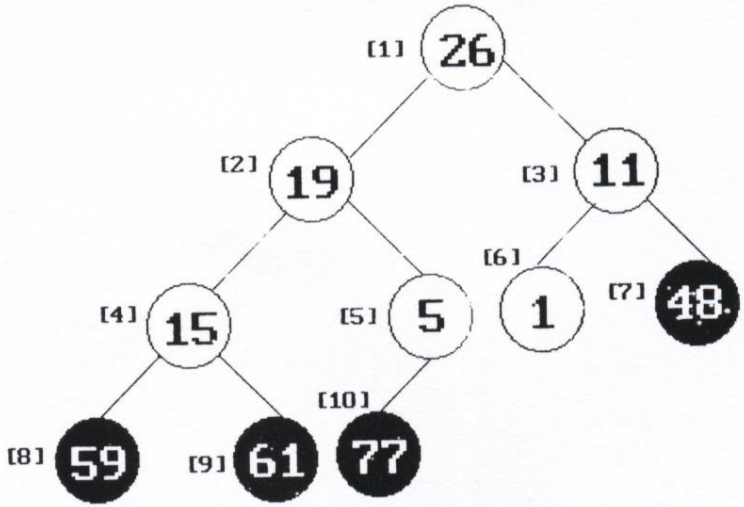


(b)

Figure 7.13: Heap sort example



(c)



(d)

Figure 7.13 (continued): Heap sort example

7.8 Radix sort

– We considers the problem of sorting records that have several keys.
  - These keys are labeled K^0, K^1, … , K^{r-1}, with K^0 being the most significant key and K^{r-1} the least. Let K_j^j denote key K^j of record R_i.
  - A list of records R_0, … , R_{n-1}, is lexically sorted with respect to the keys K^0, K^1, … , K^{r-1} *iff* $(K_i^0, K_i^1, \ldots, K_i^{r-1}) \leq (K^0_{i+1}, K^1_{i+1}, \ldots, K^{r-1}_{i+1})$, $0 \leq i < n-1$.
  - We say that the r-tuple $(x_0, x_1, \ldots, x_{r-1})$ is less than or equal to the r-tuple $(y_0, \ldots, y_{r-1})$ *iff* either $x_i = y_i$, $0 \leq i \leq j$ and $x_{j+1} < y_{j+1}$ for some $j < r-1$ or $x_i = y_i$, $0 \leq i < r$.

– Example

sorting a deck of cards on two keys, suit and face value, in which the keys have the ordering relation:

K^0[Suit]: ♣ < ♦ < ♥ < ♠

K^1[Face value]: 2 < 3 < 4 < … < 10 < J < Q < K < A

Thus, a sorted deck of cards has the ordering:

2♣, …, A♣, … , 2♠, … , A♠

– Two approaches to sort:

- MSD(Most Significant Digit)
- LSD(Least Significant Digit)

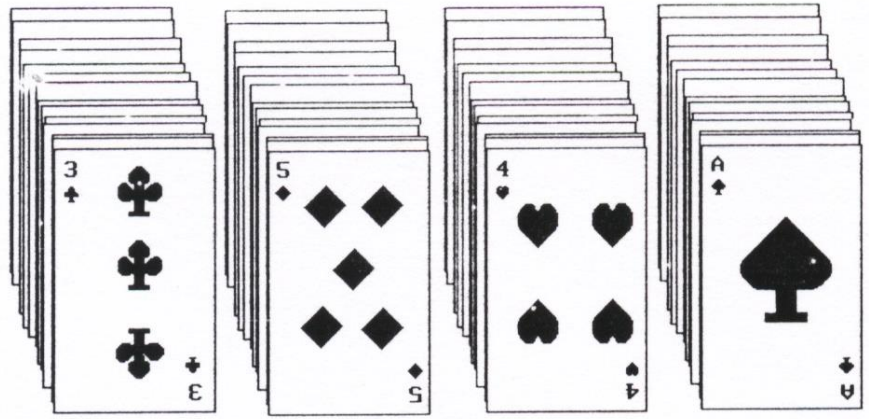


Figure 7.14: Arrangement of cards after first pass of an MSD sort

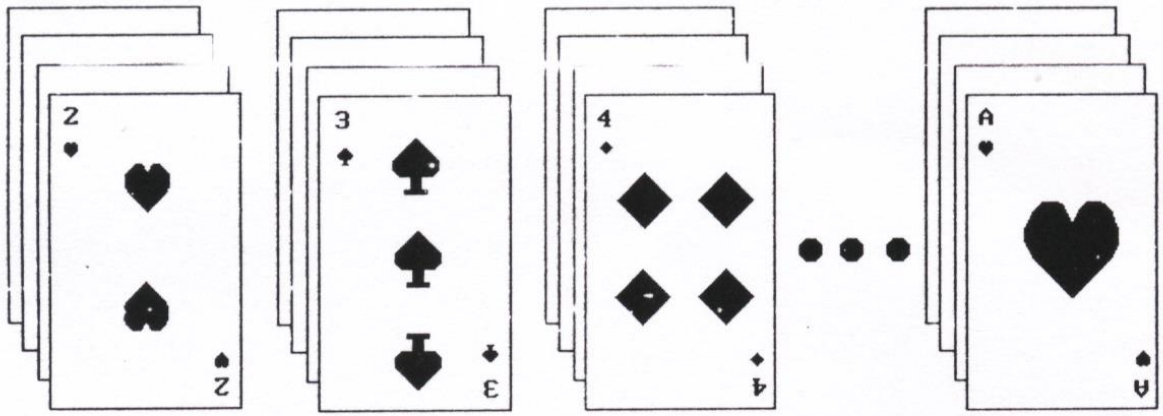


Figure 7.15: Arrangement of cards after first pass of LSD sort

- We also can use an LSD or MSD sort when we have only one logical key, if we interpret this key as a composite of several keys.
- In a *radix sort*, we decompose the sort key into digits using a radix r.
  - Ex: When r =10, we get the common base 10 or decimal decomposition of the key.
- LSD radix r sort
  - We assume that the records, $R_0, \ldots, R_{n-1}$, have keys that are d-tuples $(x_0, x_1, \ldots, x_{d-1})$ and that $0 \leq x_i < r$. Each record has a link field, and that the input list is stored as a dynamically linked list.
  - We implement the bins as queues with *front[i]*, $0 \leq i < r$, pointing to the first record in bin i and *rear[i]*, $0 \leq i < r$, pointing to the last record in bin i.

```
#define MAX_DIGIT 3 /* numbers between 0 and 999*/
#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)
/*Radix Sort using a linked list */
{
   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;
      while (ptr) {
         digit = ptr->key[i];
         if (!front[digit])
            front[digit] = ptr;
         else
            rear[digit]->link = ptr;
         rear[digit] = ptr;
         ptr = ptr->link;
      }
      /* 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;
}
```

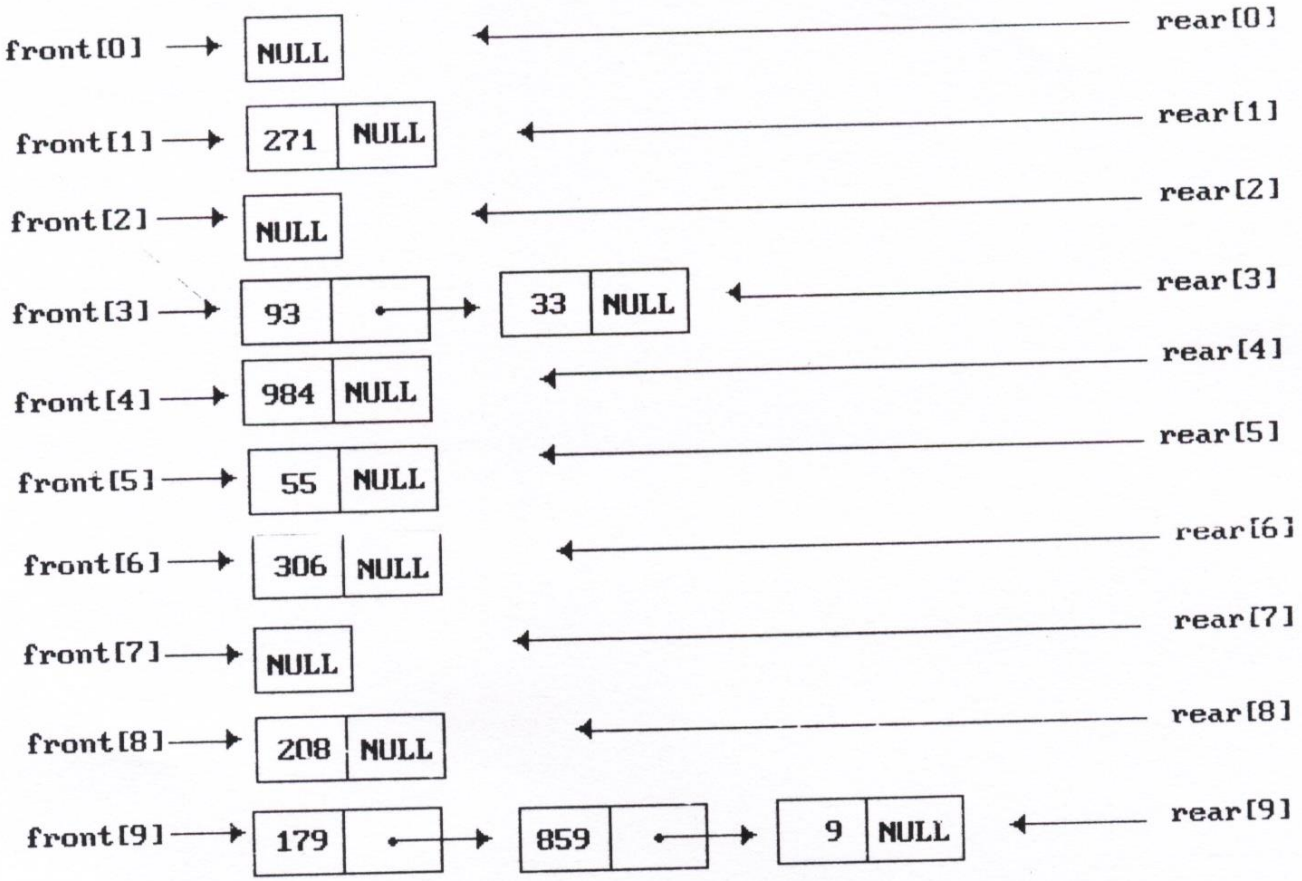
Program 7.15: LSD radix sort

– Example 7.8

- The input sequence is (179, 208, 306, 93, 859, 984, 55, 9, 271, 33). The radix is 10, and since all number are in the range [0 … 999], the number of digits is 3. The list elements are label R_0, …, R_9. Figure 7.16 illustrates the sort at each pass.

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

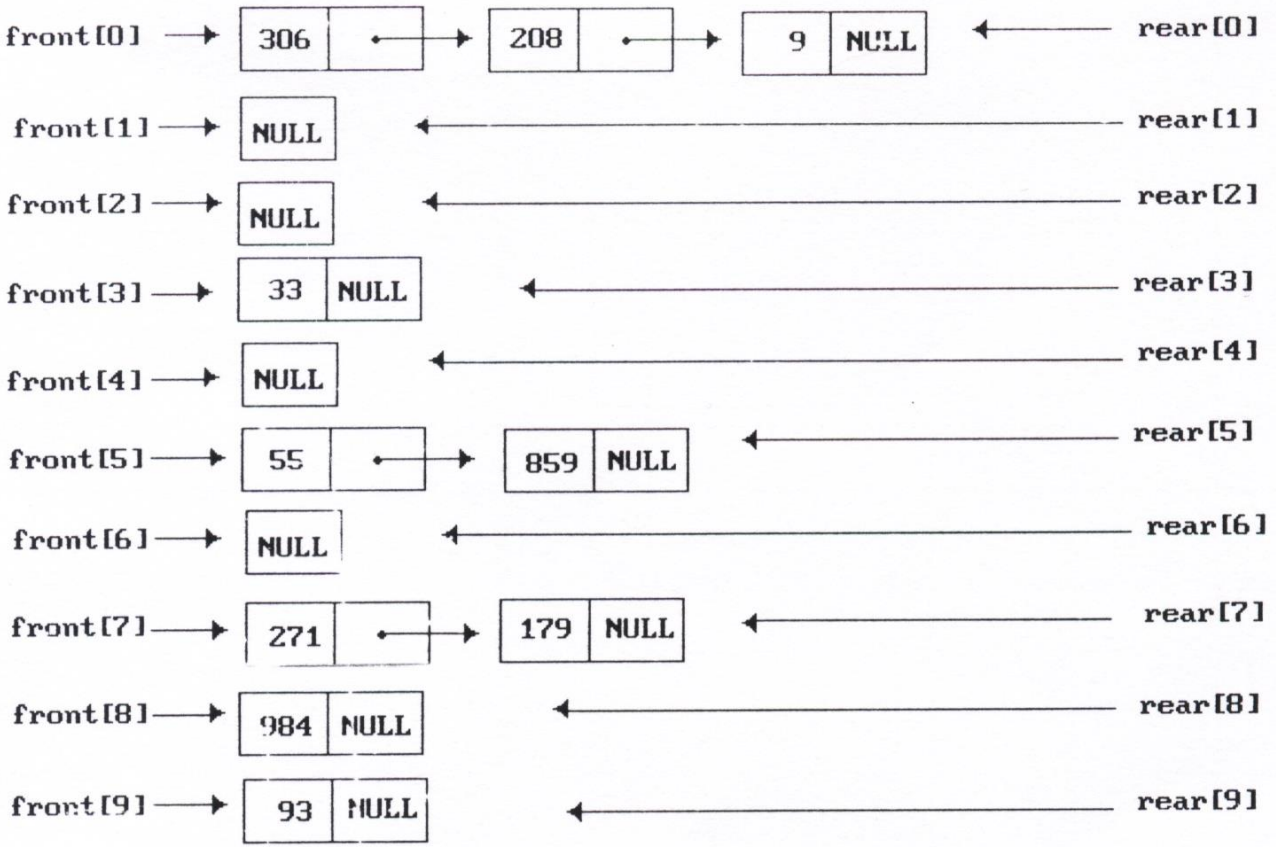
Initial input



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

Chain after first pass, $i = 2$.

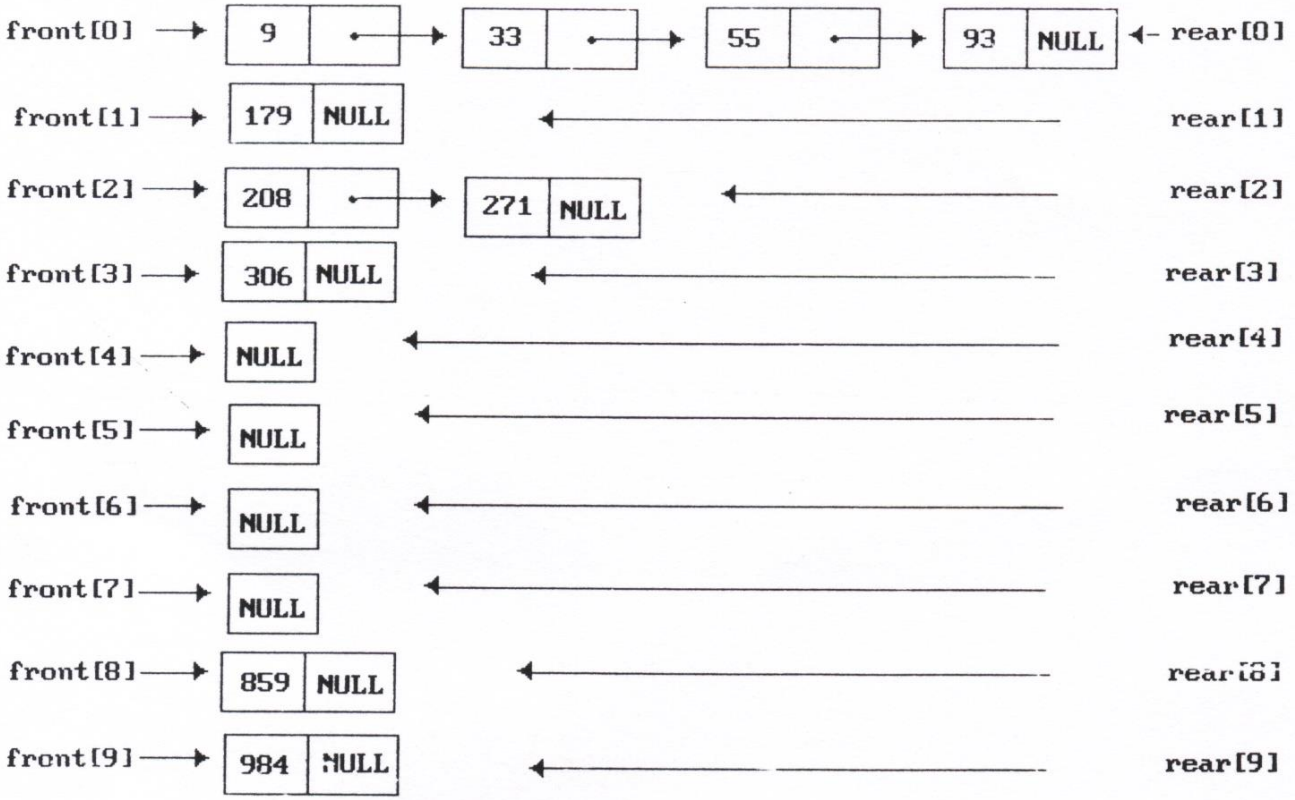
Figure 7.16: Simulation of *radix_sort*



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

Chain after second pass, $i = 1$

Figure 7.16 (continued): Simulation of *radix_sort*



$9 \rightarrow 33 \rightarrow 55 \rightarrow 93 \rightarrow 179 \rightarrow 208 \rightarrow 271 \rightarrow 306 \rightarrow 859 \rightarrow 984$

Chain after third pass, $i = 0$

Figure 7.16 (continued): Simulation of *radix_sort*

Analysis of *radix sort*:

Let the radix size be r, and the number of digits be d.

The function *radix_sort* makes d passes over the data, each pass taking $O(r + n)$ time. Hence, the total computing time is $O(d\ (r + n))$.

7.9 List and table sorts

- Apart from the radix sort and recursive merge sort, all the sorting methods we have looked at require excessive data movement since we must physically move records following some comparisons.
- If the records are large, this slows down the sorting process. Therefore, when sorting lists with large records we modify our sorting methods to minimize data movement.
- We can reduce data movement by using a linked list representation. The sort does not physically rearrange the list, but modifies the link fields to show the sorted order.

– Assume that the linked list has been sorted and that *start* points to the record that contains the smallest key. This record's link field points to the record with the second smallest key, and so on.
– *list_sort1* algorithm will physically rearrange these records into nondecreasing order.
  • The data structure used in *list_sort1* algorithm is as following:

```
typedef struct {
        int key;
        int link;
        int linkb;   /* back link */
        } element;
```

```
void list_sort1(element list[], int n, int start)
/* start is a pointer to the list of n sorted elements,
linked together by the field link.  linkb is assumed to be
present in each element.  The elements are rearranged
so that the resulting elements list[0],..., list[n-1] are
consecutive and sorted.  */
{
   int i,last,current;
   element temp;

   last = -1;
   for (current = start; current != -1;
   current = list[current].link) {
   /* establish the back links for the list */
      list[current].linkb = last;
      last = current;
   }
   for (i = 0; i < n-1; i++) {
   /* move list[start] to position i while maintaining the
   list */
      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;
   }
}
```

Program 7.16: *list_sort1*

– **Analysis of *list_sort1*:**
  - If there are *n* records in the list and each record is *m* words long, the total time is O(*mn*).
– Example 7.9

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	i	61	11	59	15	48	19
link	8	5	−1	1	2	7	4	9	6	0
linkb										

Figure 7.17 Linked list following a list sort

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
linkb	9	3	4	−1	6	1	8	5	0	7

Figure 7.18 Doubly linked list resulting from list of Figure 7.15

start = 1

i	$\mathbf{R_0}$	R_1	R_2	$\mathbf{R_3}$	R_4	R_5	R_6	R_7	R_8	R_9
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

Figure 7.19 Configuration after first iteration of the **for** loop of function *list_sort*

$i = 2$
$start = 5$

i	R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9
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

$i = 3$
$start = 7$

i	R_0	R_1	**R_2**	R_3	R_4	**R_5**	R_6	R_7	R_8	R_9
key	1	5	**11**	26	61	**77**	59	15	48	19
link	1	5	**7**	8	5	**−1**	4	9	6	3
linkb	−1	3	**1**	9	6	**4**	8	5	3	7

$i = 4$
$start = 9$

i	R_0	R_1	R_2	**R_3**	R_4	R_5	R_6	**R_7**	R_8	R_9
key	1	5	11	**15**	61	77	59	**26**	48	19
link	1	5	7	**9**	5	−1	4	**8**	6	7
linkb	−1	3	1	**5**	6	4	8	**9**	**7**	7

Figure 7.20 Example for *list_sort*1

7.10 Summary of internal sorting

– Of the several sorting methods we have studied no one method is best. Some methods are good for small *n*, others for large *n*.

Times in hundredths of a second

n	quick	merge	heap	insert
0	0.041	0.027	0.034	0.032
10	1.064	1.524	1.482	0.775
20	2.343	3.700	3.680	2.253
30	3.700	5.587	6.153	4.430
40	5.085	7.800	8.815	7.275
50	6.542	9.892	11.583	10.892
60	7.987	11.947	14.427	15.013
70	9.587	15.893	17.427	20.000
80	11.167	18.217	20.517	25.450
90	12.633	20.417	23.717	31.767
100	14.275	22.950	26.775	38.325
200	30.775	48.475	60.550	148.300
300	48.171	81.600	96.657	319.657
400	65.914	109.829	134.971	567.629
500	84.400	138.033	174.100	874.600
600	102.900	171.167	214.400	
700	122.400	199.240	255.760	
800	142.160	230.480	297.480	
900	160.400	260.100	340.000	
1000	181.000	289.450	382.250	

Figure 7.26: Average times for sort methods

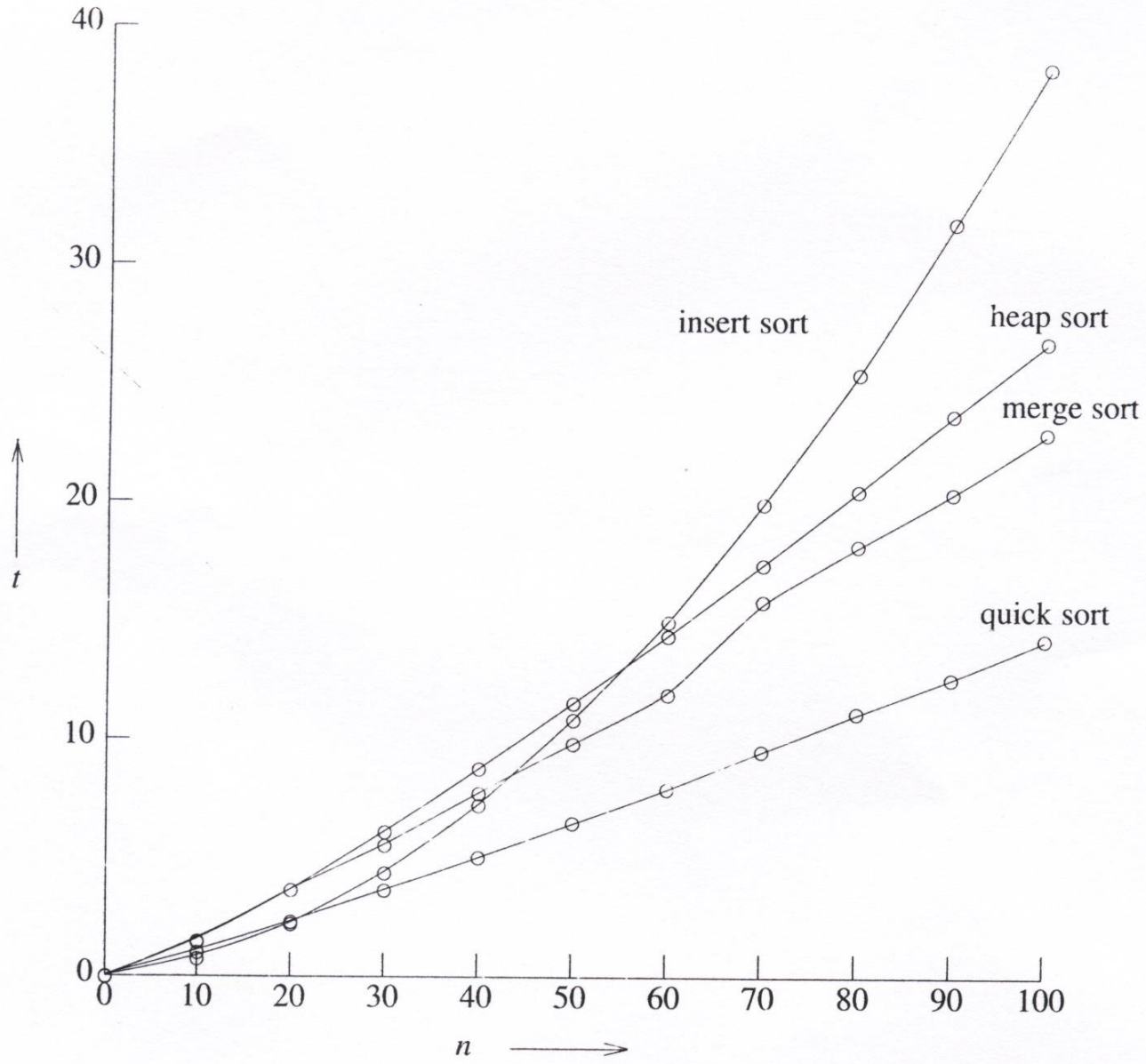


Figure 7.27: Plot of average times

7.11 External sorting

- In this section, we consider techniques to sort large files. The files are assumed to be so large that the whole file cannot be be contained in the internal memory of a computer, making an internal sort impossible.
- We shall assume that the file to be sorted resides on a disk.
- When reading or writing from/to a disk, the following overheads apply:
  - *Seek time.*
  - *Latency time.*
  - *Transmission time*.

– The most popular method for sorting on external storage devices is *merge sort*. This method consists of essentially two distinct phases.

  (1) Segments of the input file are sorted using a good internal sort method. These sorted, segments, known as runs, are written out onto external storage as they are generated.

  (2) The runs generated in phase one are merged together following the merge tree pattern of Figure 7.7, until only one run is left.

– Example:

  • A file containing 4500 records, A_1, …, A_{4500}, is to be sorted using a computer with an internal memory capable of sorting at most 750 records. The input file is maintained on disk and has a block length of 250. We have available another disk that may be used as a scratch pad. Figure 7.28 ~ 7.29 illustrate the approach outlined above.

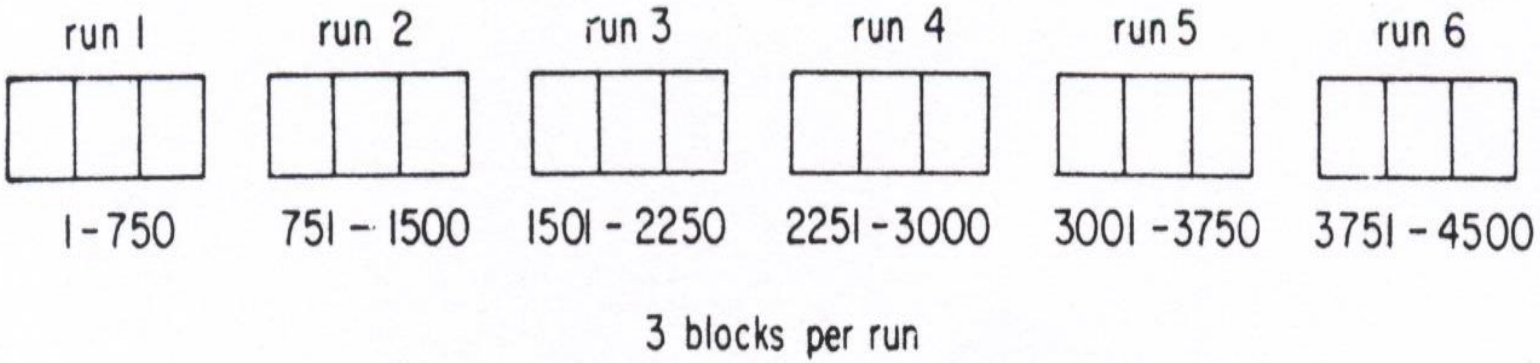


Figure 7.28: Blocked runs obtained after internal sorting

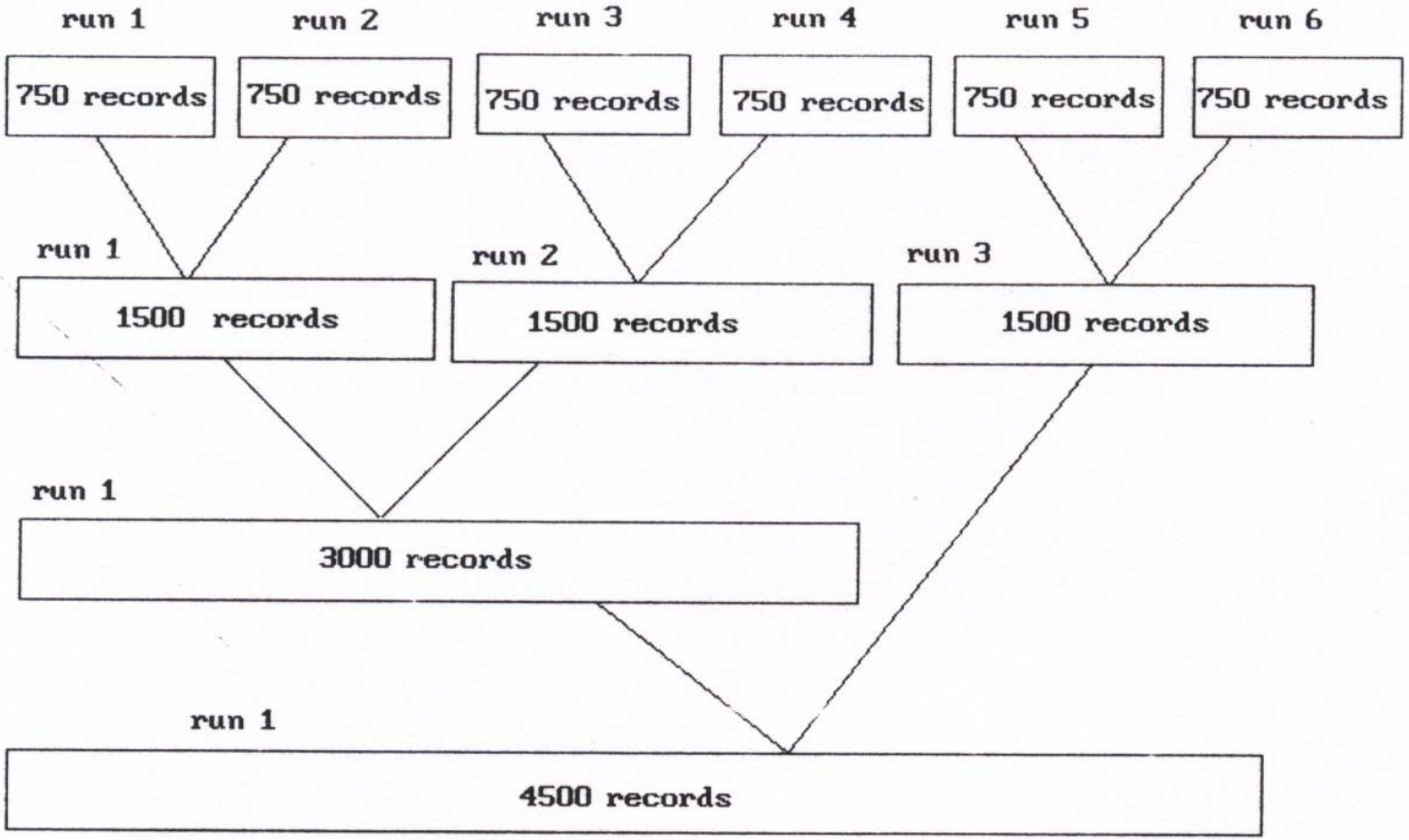


Figure 7.29: Merging the six runs

– To determine the time required by the external sort, we use the following notation:

t_s=maximum seek time

t_l=maximum latency time

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

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

t_{IS}=time to internally sort 750 records

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

	operation	time
(1)	read 18 blocks of input, $18t_{IO}$, internally sort, $6t_{IS}$, write 18 blocks, $18t_{IO}$	$36t_{IO}+6t_{IS}$
(2)	merge runs 1-6 in pairs	$36t_{IO}+4500t_m$
(3)	merge two runs of 1500 records each, 12 blocks	$24t_{IO}+3000t_m$
(4)	merge one run of 3000 records with one run of 1500 records	$36t_{IO}+4500t_m$
	total time	$132t_{IO} + 12000t_m + 6t_{IS}$

Figure 7.30 Computing times for disk sort example

- *k*-way merging
  - If we started with m runs, then the merge tree would have $\lceil \log_2 m \rceil + 1$ levels for a total of $\lceil \log_2 m \rceil$ passes over the data file.
  - The number of passes over the data can be reduced by using a higher order merge.
    - Figure 7.31 illustrates a 4-way merge on 16 runs. The number of passes over the data is now 2, versus 4 passes in the case of a 2-way merge.
  - In general, a k-way merge on m runs requires at most $\lceil \log_2 m \rceil$ passes over the data (Figure 7.32). Thus, the input/output time may be reduced by using a higher order merge.

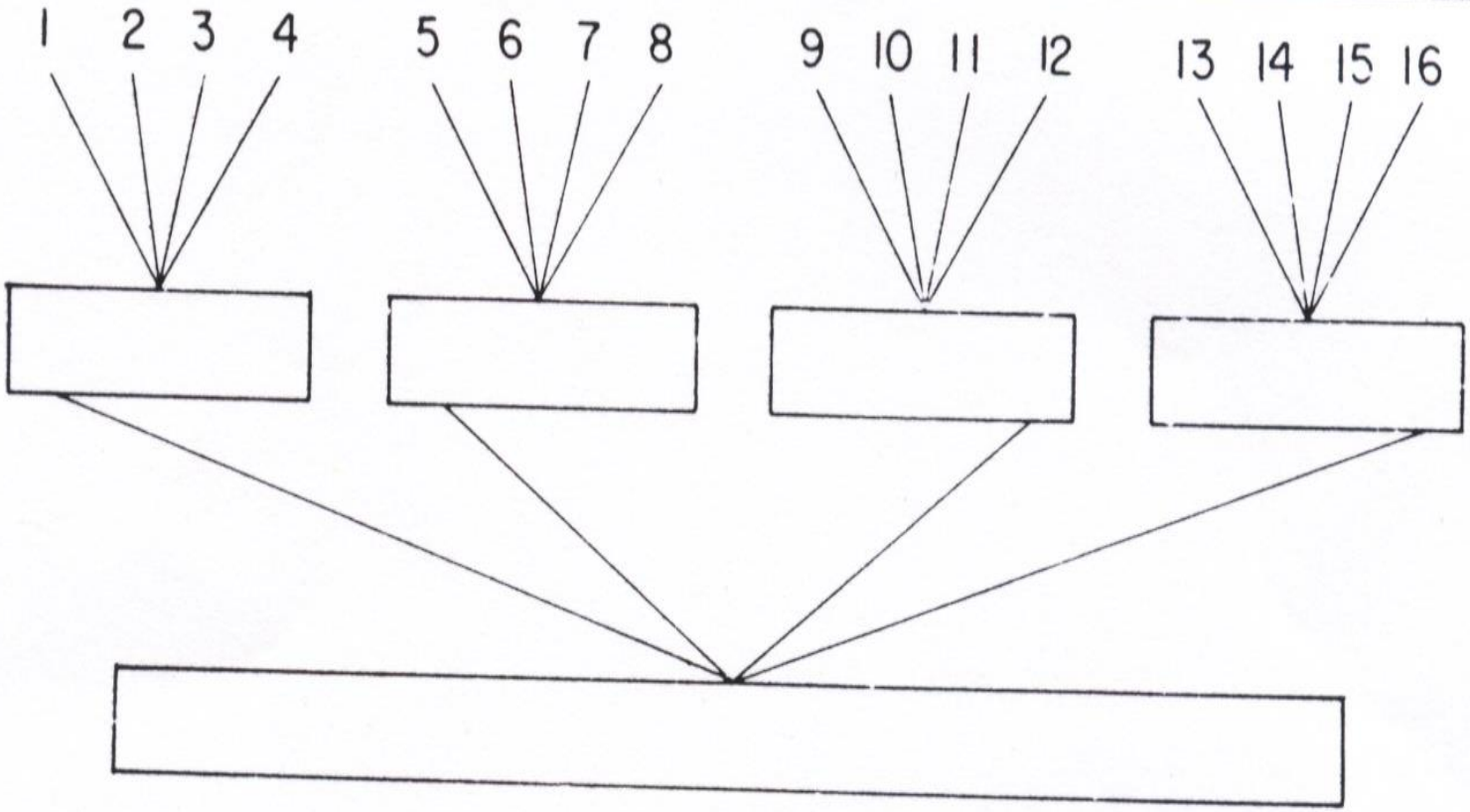


Figure 7.31: A 4-way merge on 16 runs

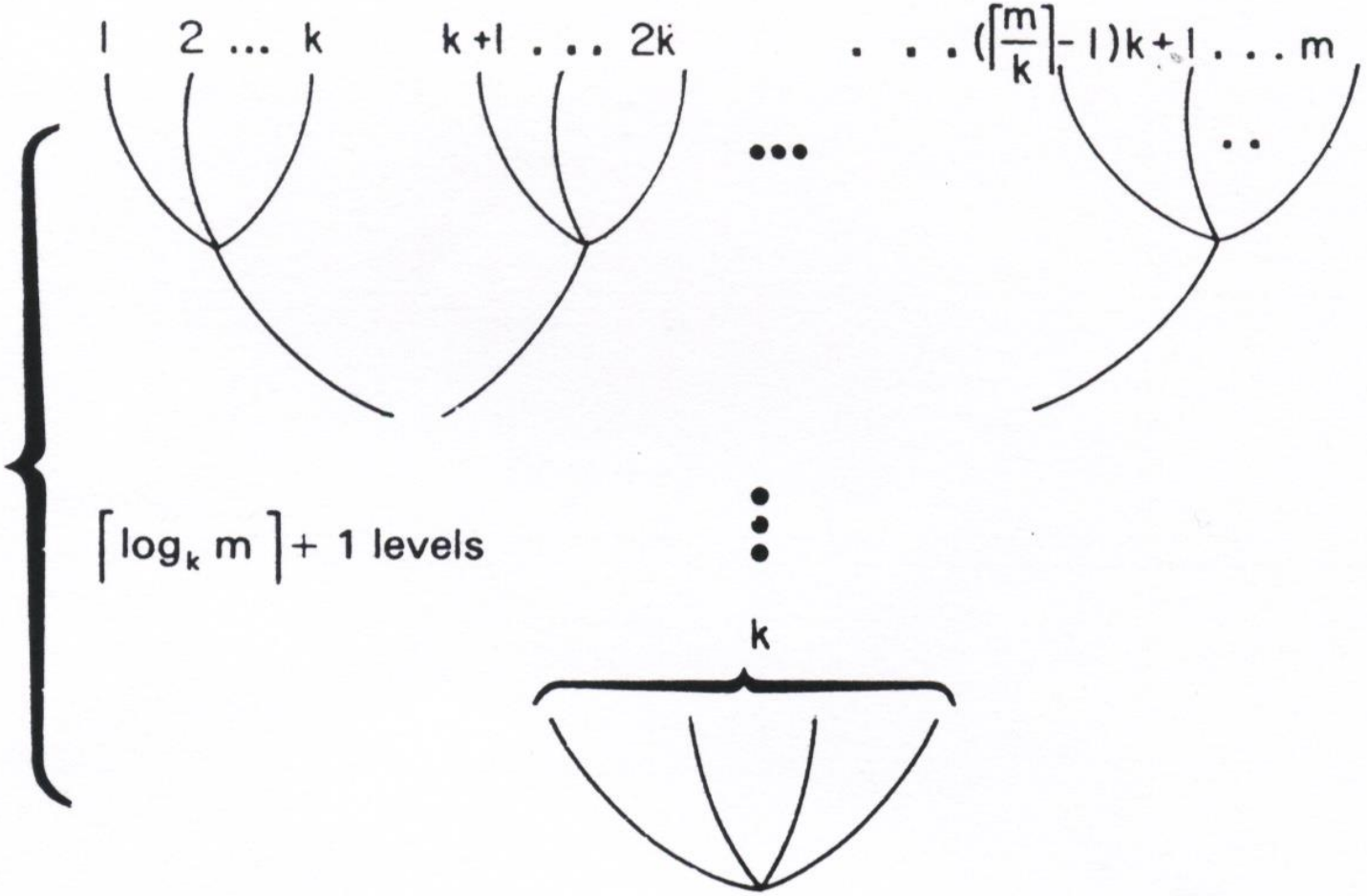


Figure 7.32: A *k*-way merge

– The use of a higher order merge, however, has some other effects on the sort. As *k* increases, the reduction in input/output time will be offset by the resulting increase in CPU time needed to perform the *k*-way merge.
– For large *k* (say, $k \geq 6$) we can achieve a significant reduction in the number of comparisons needed to find the next smallest elements by using a *loser tree* with *k* leaves.
– Beyond a certain *k* value the input/output time would actually increase despite the decrease in the number of passes being made. The optimal value for *k* clearly depends on disk parameters and the amount of internal memory available for buffers.

- Section 7.11.5 Optimal merging of runs
  - The runs generated by run_generation may not be of the same size. When runs are of different size, the run merging strategy employed so far does not yield minimum merge times.

    Example: For the tow trees of Figure 7.36, the respective weighted external path lengths are:

    2*3+4*3+5*2+15*1 = 43

    2*2+4*2+5*2+15*2 = 52

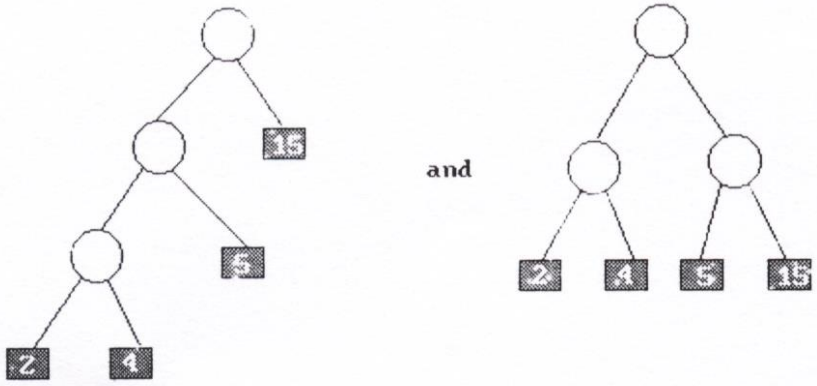


Figure 7.36: Example trees

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

– A very nice solution to the problem of finding a binary tree with minimum weighted external path length has been given by D. Huffman.

  • Example: We have the weights $q_1 = 2$, $q_2 = 3$, $q_3 = 5$, $q_4 = 7$, $q_5 = 9$, and $q_6 = 13$. The sequence of trees we would get is given in Figure 7.37.

```
typedef struct tree_node *tree_pointer;
typedef struct tree_node {
        tree_pointer left_child;
        int          weight;
        tree_pointer right_child;
};
tree_pointer tree;
int n;
```

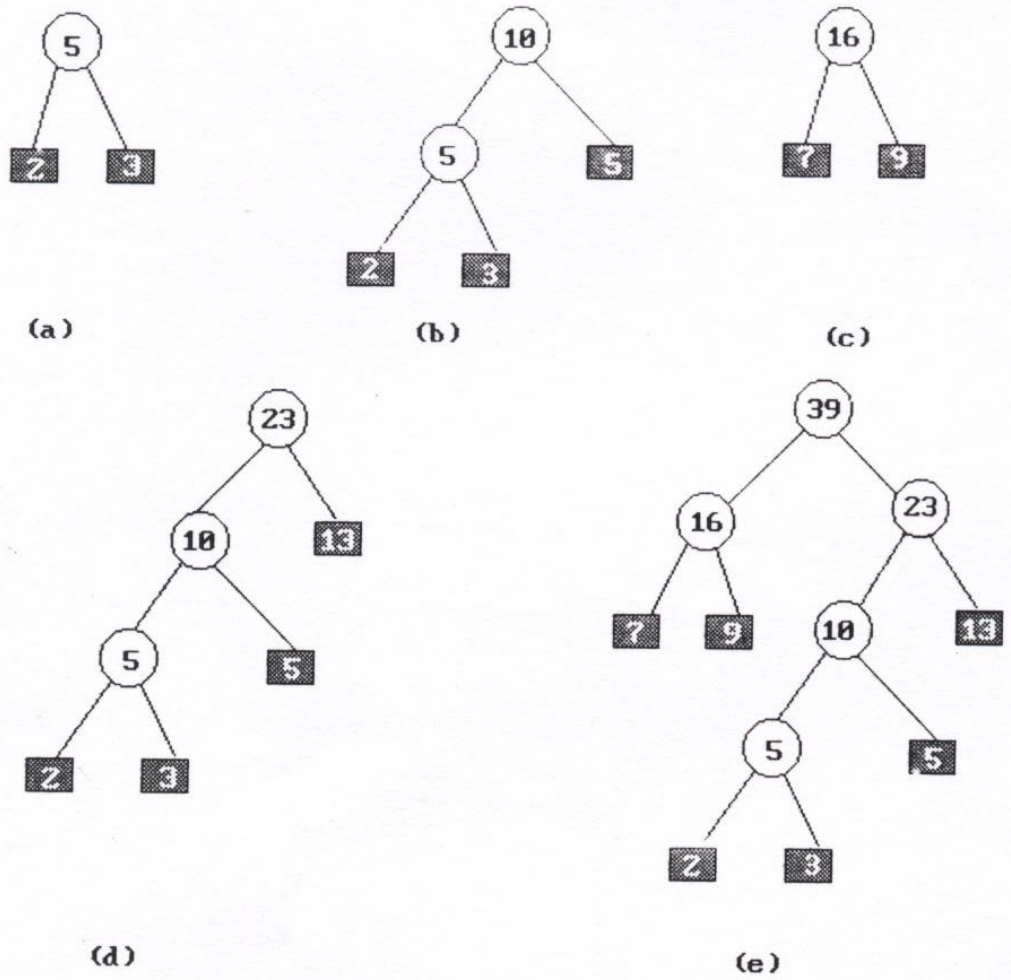


Figure 7.37: Construction of a Huffman tree

```
void huffman(tree_pointer heap[], int n)
{
/* heap is a list of n single node binary trees */
   tree_pointer tree;
   int i;
   /* initialize min heap */
   initialize(heap, n);
   /* create a new tree by combining the trees with the
   smallest weights until one tree remains */

   for (i = 1; i < n; i++) {
      tree = (tree_pointer)
                                malloc(sizeof(tree_node));
      if (IS_FULL(tree)) {
         fprintf(stderr, "The memory is full\n");
         exit(1);
      }
      tree ->left_child = least(heap, n-i+1);
      tree->right_child = least(heap, n-i);
      tree->weight = tree->left_child->weight +
      tree->right_child->weight;
      insert(heap,n-i-1,tree);
   }
}
```

Program 7.23: Huffman function

– **Analysis of *huffman*:**

  • The total computing time of *huffman function* is O(*n* log *n*).