Unit 8 Searching and Sorting Techniques

Structure:

8.1 Introduction

In the previous unit we discussed the application of graphs, which includes topological sorting, weighted shortest path, minimum spanning tree and introduction to NP completeness

In this unit you are going to learn about one of the important concepts in data structure that is "**sorting".** Sorting is one of the most fundamental algorithmic problems. Even 25 percentages of CPU cycles are spent for sorting. While doing insertion, searching or selection, it makes the processes easier and reduced the execution time. Sorting is an operation that segregates items into groups according to specified criterion and it can be applied to either numerical or alphanumerical data. Here we are going to discuss bubble, merge, selection and heap sorting techniques.

We are also going to discuss one more concept in this unit which is "**searching"**. Searching is the operation which finds the location in memory of some given ITEM, then the process is success. Otherwise it sends message about the non availability of the particular ITEM. The search is said to successful or unsuccessful according to whether the ITEM is present or not. The searching algorithm that is used depends mainly on the type of data structure that is used. In this unit you will learn about two search patterns that is linear and binary search.

Objectives:

After studying this unit, you should be able to:

- explain the definition, notations and concepts of sorting
- discuss some of the sorting techniques
- describe sequential and binary search techniques.

8.2 Sorting

Sorting is the process of arranging a collection of items in some specified order. The items/records in a file or data structures in memory, consist of one or more fields or members. One of these fields is designated as the "sort key" which means the records will be ordered according to the value of that field. In general, a key can be of any sequence of characters, and the ordering is imposed by sorting depends on the collating sequence associated with the particular character set which is being used. Sorting sequence may be ascending or descending numerical, lexical ordering, or date. Sorting is the subject of a great deal of study since it is a common operation which can consume a lot of computer time. There are many well-known sorting we are going to discuss few algorithm in the coming up subsections.

8.2.1 Notations and concepts

Data can occur in any form and it is assumed that we are given a collection of elements. Each element is represented by a record contains the related information. The collection of records called table which represents the information where operation of sorting can be performed. The organization of table is application dependent may contains different types of data.

A table is assumed to be the order of n records R_1, R_2 .... R_n. Each record contains more than one key, but here we assume each record is having single key and other information. Sorting is the process of arranging the content in same order based on ordering criterion. Sorting can be done on any type of data like numeric or alphanumeric. Numerical sorting results in arranging the records in either ascending or descending order. In general key can be of any sequence but for our convenience we are going to assume the key only as numeric key. Most of the sorting algorithms involve movement of the records from one place to another in the table. In certain applications records are quite long and really expensive to move, the records can be organized in such a manner as to minimize this moving cost while performing the sort.

One method of reducing the cost of moving the records is to arrange the table as a simple linked list. The movement of records is efficient when such a representation is used. Another method is to use a pointer vector, each element of which contains the address of one record.

8.2.2 Bubble sort

Bubble sort is a straightforward and simplistic method of sorting data that is used very commonly. The algorithm starts at the beginning of the data set. It compares the first two elements, and if the first is greater than the second, then it swaps them. It continues doing this for each pair of adjacent elements to the end of the data set. It then starts again with the first two elements, repeating until no swaps have occurred on the last pass. This algorithm is highly inefficient, and is rarely used except as a simplistic example. For example, if we have 100 elements then the total number of comparisons will be 10000. Bubble sort may, however, be efficiently used on a list that is already sorted, except for a very small number of elements. For example, if only one element is not in order, bubble sort will only take *2n* time. If two elements are not in order, bubble sort will only take at most *3n* time. Bubble sort average case and worst case are both $O(n^2)$.

Procedure: BUBBLE_SORT (K,N). Given vector K and N elements this procedure sorts the elements into ascending order using the method just described. The variables PASS and LAST denote the pass counter and position of the last unsorted element, respectively. Variable I is used as index of vector elements variable EXCHS is used to count the number of exchanges.

```
1. [Initialize ] LAST←N (entire list is unsorted at this point)
2. [Loop on pass index]
           Repeat thru step 5 for PASS = 1, 2, ... N-1
3. [initialize exchanges counter for this pass]
   EXCHS←0
4. [perform pairwise comparisons on unsorted elements]
   Repeat for 1 = 1, 2, ... LAST-1
```

```
            If K[1] > K[I+1]
            then K[I] <-> K[I+1]
            EXCHS <- EXCHS +1
    5.  [were any exchanges made on this pass]
        If EXCHS=0
        then Return
        else LAST<-LAST-1
    6.  [Finished]
        Return (maximum number of passes required)
```

This algorithm is straightforward. Before each pass, the interchange marker EXCHS is initialized to zero. This marker is incremented each time an interchange is made. If, at the end of a pass, EXCHS has a value of zero, the sort is complete. We will now discuss the bubble sort with simple example, consider an array:

45, 67, 12, 34, 25, 39

In the first step, the focus is on the first two elements (in **bold**) which are compared and swapped, if necessary. In this case, since the element at index 1 is larger than the one at index 0, no swap takes place.

45**, 67**, **12**, 34, 25, 39

Then the focus move to the elements at index 1 and 2 which are compared and swapped, if necessary. In our example, 67 is larger than 12 so the two elements are swapped. The result is that the largest of the first three elements is now at index 2.

45, 12, **67, 34**, 25, 39

The process is repeated until the focus moves to the end of the array, at which point the largest of all the elements ends up at the highest possible index. The remaining steps and result are:

45, 12, 34**, 67, 25**, 39

45, 12, 34**,** 25, **67, 39**

45, 12, 34, 25, 39, 67

You can observe that one complete bubble step moves the largest element to the last position. Now the effective size of the array is reduced by 1 and the process repeats until the size becomes zero.

8.2.3 Merge sort

Divide-and conquer is a general algorithm design paradigm:

Divide: divide the input data ***S*** in two disjoint subsets $S_1$and S_2

Recur: solve the sub problems associated with S_1 and S_2

Conquer: combine the solutions for S_1 and S_2 into a solution for ***S***

The base case for the recursion is sub problems of size 0 or 1

Merge-sort on an input sequence ***S*** with ***n*** elements consists of three steps:

Divide: partition ***S*** into two sequences ***S***1 and ***S***2 of about ***n***/2 elements each

Recur: recursively sort ***S***1 and ***S***2

Conquer: merge ***S***1 and ***S***2 into a unique sorted sequence.

Merge sort algorithm proceeds as follows:
1) Divide the array with the size of m as middle= m/2.
2) Update left from 1 to middle.
3) Update right from middle+1 to m
4) Repeat 1 to 3 until m≤1
5) Call merge [left, right].

```
merge_sort[m]
    if length[m] ≤ 1
        return m
     middle ← length[m] / 2
    for x ← 1 to middle
         add x to left
    for x ← middle to m
         add x to right
    left ← merge_sort[left]
    right ← merge_sort[right]
    result ← merge[left, right]
    return result
```

```
merge[left, right]
    while length[left] > 0 or length[right] > 0
        if length[left] > 0 and length[right] > 0
            if first[left] ≤ first[right]
                append first[left] to result
                left← rest[left]
            else
                append first[right] to result
                right ← rest[right]
        else if length[left] > 0
            append first[left] to result
            left ← rest[left]
        else if length[right] > 0
            append first[right] to result
            right ← rest[right]
    end while
    return result
```

Given Input with the size of m=8

7 2 9 4 3 8 6 1

Algorithm starts with dividing the array until m<=1.

7 2 9 4 | 3 8 6 1

7 2 | 9 4 | 3 8 | 6 1

7 | 2 | 9 | 4 | 3 | 8 | 6 | 1

After dividing the given elements, process proceeds with merge sort. While merging, the elements will also be sorted in the order (highlighted) and the result will be stored in the resultant array. Figure 8.1 explores the step by step procedure of merging process.

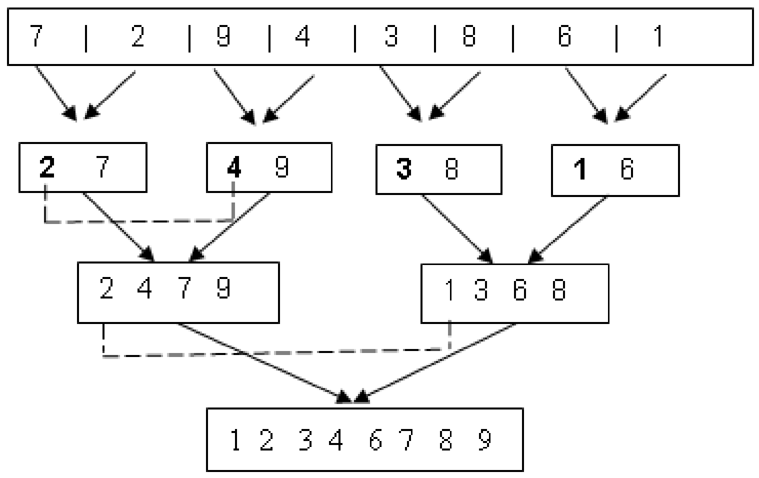


Figure 8.1: Merging after sorting elements

8.2.4 Selection sort

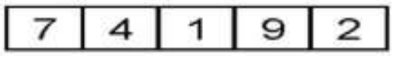
Selection sort is one of the sorting techniques that is typically used for sequencing small lists. It starts by comparing the entire list for the lowest item and moves it to the #1 position. It then compares the rest of the list for the next-lowest item and places it in the #2 position and so on until all items are in the required order. Selection sorts perform numerous comparisons, but fewer data movements than other methods.

Algorithm for selection sort:

1) Find the minimum value in the list
2) Swap it with the value in the first position
3) Repeat the steps above for the remainder of the list (starting at the second position and advancing each time).

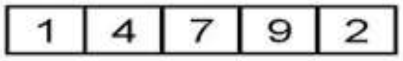
```
Selection sort (A, n)
For i = 0 to n-1
 MIN← i
    For j = i + 1 to n
     if (A[j] < A[i])
     MIN← j
     End if
Swap(A[j], A[i]; // swap min to front
 Return
```

As an example, begin with the following array.



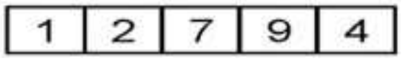
7	4	1	9	2

The selection sort marks the first element (7). It then goes through the remaining data to find the smallest number (1). It swaps with the first element (7) and the smallest element (1) which is placed in its correct position.



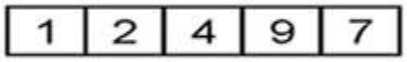
1	4	7	9	2

It then marks the second element (4) and looks through the remaining data for the next smallest number (2). These two numbers are then swapped.



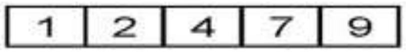
1	2	7	9	4

Marking the third element (7) and looking through the remaining data for the next smallest number (4). These two numbers are now swapped.



1	2	4	9	7

Lastly it marks the fourth element (9) and looks through the remaining data for the next smallest number (7). These two numbers are then swapped.



1	2	4	7	9

8.2.5 Heap sort

Heap: A balanced, left-justified binary tree in which no node has a value greater than the value in its parent.

***Heap Property*:** In a heap, for every node *i* other than the root, the value of a node is greater than or equal (at most) to the value of its parent A[PARENT (*i*)] ≥ A[*i*].

Heap sort: Heap sort is a much more effective type of selection sort, and is one of the fastest sorting algorithms. Programmers often use this when faced with very large arrays they know to be in an unsorted state. The largest or smallest element of the list is determined and then placed at the end or beginning of the list. This process continues with the rest of the list, and is accomplished by using a data structure called a heap, which is a special type of binary tree. Once the data list has been made into a heap, the root node is guaranteed to be the largest element. When it is removed and placed at the end of the list, the heap is rearranged so the largest element remaining moves to the root.

***Heapify procedure*:** Heapify picks the largest child key and compare it to the parent key. If parent key is larger than heapify quits, otherwise it swaps the parent key with the largest child key. So that the parent is now becomes larger than its children. Swap may destroy the heap property of the subtree rooted at the largest child node. If this is the case, Heapify calls itself again using largest child node as the new root.

```
Heapify (A, i)
        l ← left [i]
        r ← right [i]
        if l ≤ heap-size [A] and A[l] > A[i]
            then largest ← l
            else largest ← i
        if r ≤ heap-size [A] and A[i] > A[largest]
            then largest ← r
        if largest  ≠ i
            then exchange A[i] ↔ A[largest]
.               Heapify (A, largest)
```

HEAPSORT() starts with finding the maximum element of the array stored at the root $A[1]$, it can be put into its correct final position by exchanging it with $A[n]$ (the last element in A). If we now discard node n from the heap than the remaining elements can be made into heap. Note that the new element at the root may violate the heap property, to maintain the heap status heapify will be called.

```
HEAPSORT(A)
for i ← length (A) down to 2 do
exchange      A[1]      ↔      A[i]
heap-size [A] ← heap-size [A] -
1
Heapify (A, 1)
```

For example take an array with the size of 8 with following elements.

2 7 3 5 6 4 8 1

The level order binary tree will be constructed as below in the figure 8.2.

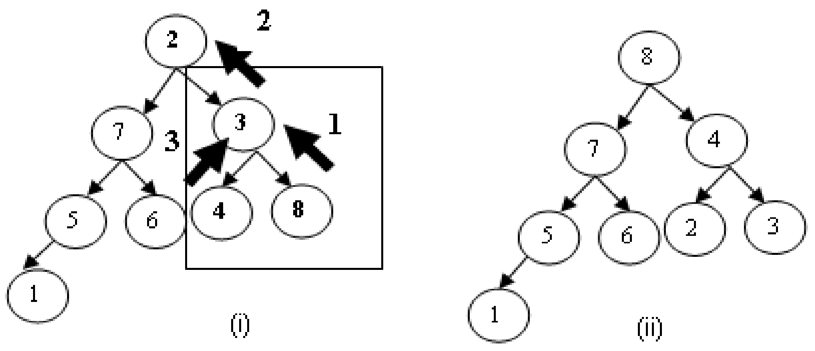


Figure 8.2 (a)

Here tree in figure 8.2(a) (i) is not heap because the heap property is A[PARENT (*i*)] ≥ A[*i*].

The node 3 is having its child with highest value so it requires exchange. You can observe exchange takes place in the specified order as 1, 2 and 3 in figure 8.2(a) (ii). Now the tree is heap and the highest value available in the root can be removed.

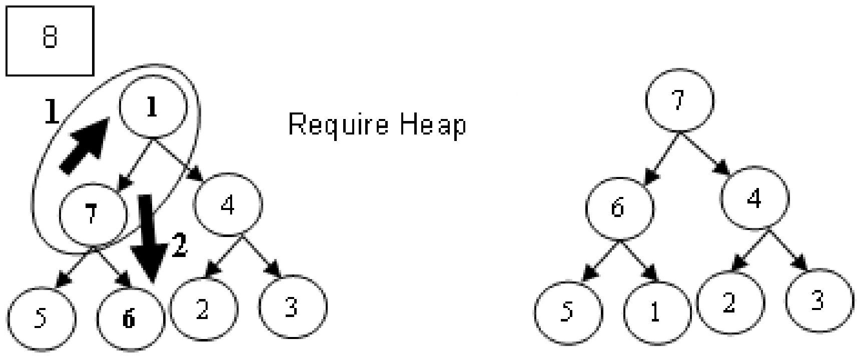
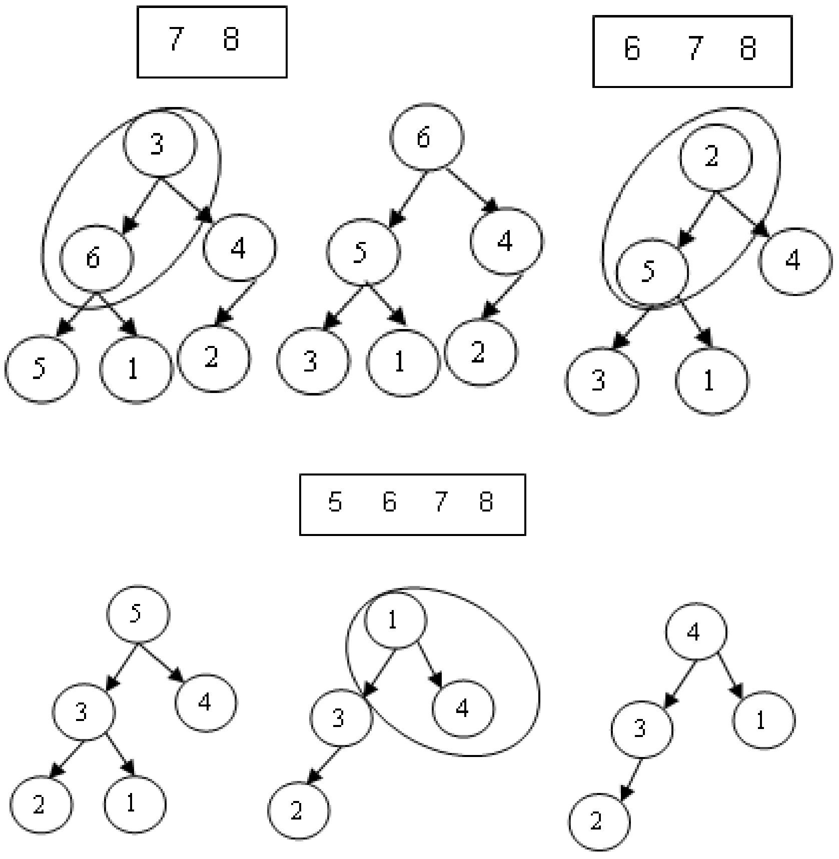


Figure 8.2 (b)

While removing the root it should be replaced with the last element, and the last element is identified as 1.

Again we need to apply the heapify algorithm to make the tree heap as shown in figure 8.2(c). Thus the value is exchanged with the value 7 and further 1 is exchanged with 5. Now the tree is in heap so the highest value from the tree can be removed and update with the result array. Now the root will be replaced with the last node that is 3. If we continue with the procedure we will get the sorted array.



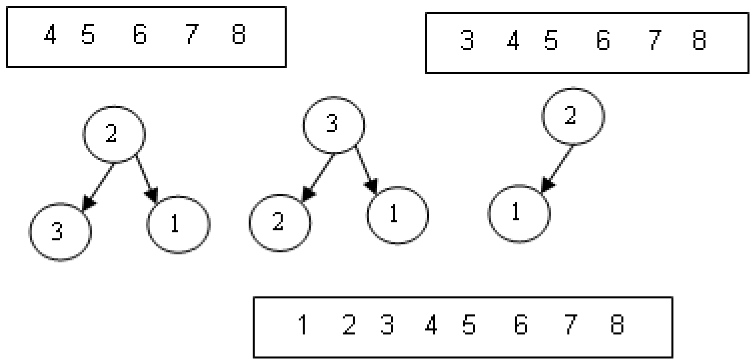


Figure 8.2 (c)

Self Assessment Questions

1. Sorting is the process of arranging the content in some order based on __________ criterion.
2. Identify the algorithm given below:

```
For k = 0 to n-1
 MIN← k
    For m = k + 1 to n
     if (A[m] < A[k]
     MIN← m
     End if
Swap(A[m], A[k]); // swap min to front
Return
```

   Ans: __________________________
3. Merge sort uses ___________________ technique to sort the elements.
4. _________________ uses left-justified binary tree to sort the elements.
5. Write down the array status after the 3rd step of sorting the array 'A[ ]' given below , using different sorting techniques:
   5, 2, 6, 1, 8, 10.
   i) bubble: _________________________.
   ii) selection: __________________________.

8.3 Searching

Computer systems are often used to store large amounts of data from which individual records must be retrieved according to some search criterion.

Thus the efficient storage of data to facilitate fast searching is an important issue. In this section, we shall investigate the performance of some searching algorithms and the data structures which they use.

8.3.1 Sequential searching

The simplest type of searching process is the **sequential search or linear search.** In the sequential search, each element of the array is compared to the key, in the order it appears in the array, until the first element matching the key is found. If you are looking for an element that is near the front of the array, the sequential search will find it quickly. The more data that must be searched, the longer it will take to find the data that matches the key using this process.

For a list with *n* items, the best case is when the value is equal to the first element of the list, in which case only one comparison is needed. The worst case is when the value is not in the list (or occurs only once at the end of the list), in which case *n* comparisons are needed. The input to a search algorithm is an array of objects A, the number of objects n, and the key value being sought x. Here we are going to discuss linear search for ordered and unordered list.

Unordered linear search

If the given array is not sorted while searching an element in the unordered list, we need to continue the search until we find the value or till the end of the array. This might correspond, for example, to collection exams which have not yet been sorted alphabetically. If a student wanted to obtain her exam score, how could she do so? She would have to search through the entire collection of exams, one-by-one, until her exam was found. This corresponds to the unordered linear search algorithm.

```
Unordered-Linear-Search[A, n, x]
1 for i ← 1 to n
2    do if A[i] = x
3       then return i
4       else i ← i + 1
5 return "x not found"
```

Note that in order to determine that an object does not exist in the collection, one needs to search through the entire collection. Now consider the following array:

i	1	2	3	4	5	6	7	8
A	34	16	12	11	54	10	65	37

Now we will discuss the Unordered-Linear-Search [A, 8, 54] . The variable i would initially be set to 1, and since A[1] (i.e., 34) is not equal to x (i.e., 54), i would be incremented by 1 in Line 4. Since A[2] 6= x, i would again be incremented, and this would continue until i = 5. At this point, A[5] = 54 = x, so the loop would terminate and 5 would be returned in Line 3. 1.

Now consider executing the pseudocode Unordered-Linear-Search [A, 8, 53]. Since 53 does not exist in the array, i would continue to be incremented until it exceeded the bounds of the for loop, at which point the for loop would terminate. In this case, “x not found” would be returned in Line 5.

Ordered linear search

Now suppose that the given array is sorted. In this case, we need not necessarily search through the entire list to find a particular object or determine that it does not exist in the collection. For example, if the collection of exams were sorted by name, one need not search beyond the “P”s to determine that the exam for “Peterson” does or does not exist in the collection. A simple modification of the above algorithm yields the ordered linear search algorithm.

```
Ordered-Linear-Search[A, n, x]
  1. 1 for i← 1 to n
  2. do if A[i] = x
  3. then return i
  4. elseif A[i] < x
  5. then i ← i + 1
  6. else return “x not found”
  7. return “x not found”
```

Note that the search can now be terminated early if and when it is determined that A[i] > x in Line 6. Now consider the following array, the sorted version of the array used in our previous example:

i	1	2	3	4	5	6	7	8
A	10	11	12	16	34	37	54	65

Now consider executing the pseudocode Ordered-Linear-Search[A, 8, 54]. The variable i would initially be set to 1, and since A[1] (i.e., 10) is not equal to x (i.e., 54), a test would be performed in Line 4 to see if A[1] is less than x. Since A[1] is less than x, i would be incremented by 1 in Line 5. Since A[2] < x, i would again be incremented, and this would continue until i = 7. At this point, A[7] = 54 = x, so the loop would terminate and 7 would be returned in Line 3. Now consider executing the pseudocode Ordered-Linear-Search [A, 8, 53]. Since 53 does not exist in the array, i would continue to be incremented until i = 7, at which point A[i] (i.e., 54) is no longer less than x (i.e., 53) so the loop would terminate in Line 6 returning "x not found."

8.3.2 Binary search

We are going to discuss one more search which is very efficient especially where you have more elements to search. Binary search is one of the fastest ways to search the element in a sorted array. The idea is to look at the element in the middle. If the key is equal to that element, then the search is finished. If the key is less than the middle element, do a binary search on the first half. If it's greater than the middle element, do a binary search of the second half.

The advantage of a binary search over a linear search is, amazing when you apply binary search for large numbers. For an array of a million elements, binary search, O(log N), will find the target element with a worst case of only 20 comparisons. Linear search, O(N), on average will take 500,000 comparisons to find the element. But to apply binary search on a particular array, it must be sorted first. Because sorting is not a fast operation, it may not be worth the effort to sort when there are only a few searches.

Now consider the following idea for a search algorithm using our phone book example. Select a page roughly in the middle of the phone book. If the name being sought is on this page, your search is over. If the name being sought is occurs alphabetically before this page, repeat the process on the "first half" of the phone book; otherwise, repeat the process on the "second half" of the phone book. Note that in each iteration, the size of the remaining portion of the phone book to be searched is ***divided in half***; the algorithm applying such a strategy is referred to as binary search.

```
Binary-Search[A, n, x]
 1.  low ← 1
 2.  high ← n
 3.  while low ≤ high
 4.  do mid← [(low + high)/2]
 5.  if A[mid] = x
 6.  then return mid
 7.  elseif A[mid] < x
 8.  then low← mid + 1
 9.  else high   mid − 1
10.  return "x not found
```

Now consider the following array for binary search.

i	1	2	3	4	5	6	7	8
A	10	11	12	16	34	37	54	65

Consider executing the algorithm Binary-Search[A, 8, 34]. The variable high is initially set to 8, and since low (i.e., 1) is less than or equal to high, mid is set to **(low + high)/2 = 9/2 = 4**.

Since A[mid] (i.e., 16) is not x (i.e., 34), and since A[mid] < x, low is reset to mid + 1 (i.e., 5) in Line 8. Since low ≤ high, a new mid = [(5 + 8)/2] = 6 is calculated, and since A[mid] (i.e., 37) is greater than x, high is now reset to mid −1 (i.e., 5) in Line 9. It is now the case that low and high are both 5; mid will then be set to 5 as well, and since A[mid] (i.e., 34) is equal to x, mid = 5 will be returned in Line 6.

Now consider executing the algorithm Binary-Search [A, 8, 33]. Binary-Search will behave exactly as described above until the point when mid = low = high. Since A[mid] (i.e., 34) is greater than x (i.e., 33), high will be reset to mid − 1 = 4 in Line 9. Since it is no longer the case that low ≤ high, the while loop terminates, returning "x not found" in Line 10.

Self Assessment Questions

6. _______________ searching technique can be applied if the given set is unordered.
7. *Pick the correct option:*
   Which algorithm uses MID value to search:
   a) Binary
   b) Linear
   c) Heap
   d) Merge
8. Given the array 'A[ ]' below, predict the result (found at <pos>/ not found) after using linear and binary search techniques, and indicate which is faster.
   i) 2, 4, 7. 19, 20. Where key = 2.
   Answer: a) linear:_______________ b) binary:_______________
   ii) 13, 12, 6, 8, 4, 3. Where key = 12
   Answer: a) linear:_______________ b) binary:_______________
   iii) 1, 2, 3, 4, 5, 6, 7, 8, 9. Where key = 9
   Answer: a) linear:_______________ b) binary:_______________
   iv) 23, 13, 25, 35, 56, 78, 94. Where key = 100
   Answer: a) linear:_______________ b) binary:_______________

8.4 Summary

In this unit we discussed two important concepts in data structure called sorting and searching. The efficiency of the data can be substantially increased if the data are stored according to some criteria or order. In this unit we discussed four different algorithms for sorting. The selection of sorting relies on the nature of data structure. Information retrieval is one of the most important applications of computers. Most of the information is available in the form of record. Each record will have a unique data or field called key to identify the record. Sorting technique makes the searching process easy which in turn makes the algorithm very efficient.

8.5 Terminal Questions

1. Explain sorting with its notation and concept.
2. Discuss bubble sort with example.

3. Elaborate merge sort with example.
4. Explain selection sort with example.
5. Describe heap sort with example.
6. Explain sequential search in detail with example.
7. Discuss binary search with example.

8.6 Answers

Self Assessment Questions

1. ordering
2. selection sort
3. divide-and-conquer
4. heap sort
5. i) bubble: 2, 5, 1, 6, 8, 10.
   ii) selection:1, 2, 5, 6, 8, 10.
6. linear search
7. a) Binary
8. i) a) linear: found at A[0] b) binary: A[0], linear search is faster.
   ii) a) linear: found at A[1] b) binary: not found. Linear search is faster.
   iii) a) linear: found at A[8] b) binary: A[8]. Binary search is faster.
   iv) a) linear: not found b) binary: not found. Binary search is faster.

Terminal Questions

1. To arrange a collection of items in some specified order. The items/records in a file or data structures in memory, consist of one or more fields or members. Data can occur in any form and it is assumed that we are given a collection of elements. Each element is represented by a record contains the related information. The collection of records called table which represents the information where operation of sorting can be performed. (Refer section 8.2 for detail)
2. *Bubble sort* is a straightforward and simplistic method of sorting data that is used very commonly. The algorithm starts at the beginning of the data set. It compares the first two elements, and if the first is greater than the second, then it swaps them. (Refer sub-section 8.2.2 for detail).
3. Merge sort for which, Divide-and conquer is a general algorithm design paradigm:

   Divide: divide the input data ***S*** in two disjoint subsets $S_1$and S_2

Recur: solve the sub problems associated with $\boldsymbol{S_1}$ and $\boldsymbol{S}_2$

Conquer: combine the solutions for $\boldsymbol{S_1}$ and $\boldsymbol{S}_2$ into a solution for $\boldsymbol{S}$

The base case for the recursion is sub problems of size 0 or 1.

(Refer sub-section 8.2.3 for detail).

4. Selection sort is one of the sorting techniques that is typically used for sequencing small lists. It starts by comparing the entire list for the lowest item and moves it to the #1 position. It then compares the rest of the list for the next-lowest item and places it in the #2 position and so on until all items are in the required order. (Refer sub-section 8.2.4 for detail).
5. Heap sort is a much more effective type of selection sort, and is one of the fastest sorting algorithms. Programmers often use this when faced with very large arrays they know to be in an unsorted state. (Refer sub-section 8.2.5 for detail).
6. The simplest type of searching process is the sequential search or linear search. In the sequential search, each element of the array is compared to the key, in the order it appears in the array, until the first element matching the key is found. (Refer sub-section 8.3.1 for detail).
7. Binary search is one of the fastest ways to search the element in a sorted array. The idea is to look at the element in the middle. If the key is equal to that element, then the search is finished. If the key is less than the middle element, do a binary search on the first half. If it's greater than the middle element, do a binary search of the second half. (Refer sub-section 8.3.2 for detail).