UNIT – II: SORTING AND SEARCHING TECHNIQUES

Structure

2.1 INTRODUCTION

Sorting and Searching are fundamental operations in computer science. Sorting refers to the operation of arranging data in some given order. Searching refers to the operation of searching the particular record from the existing information. Normally, the information retrieval involves searching, sorting and merging. In this chapter we will discuss the searching and sorting techniques in detail.

2.2 OBJECTIVES

After going through this unit you will be able to:

- Know the fundamentals of sorting techniques
- Know the different searching techniques
- Discuss the algorithms of internal sorting and external sorting
- Difference between internal sorting and external sorting
- Complexity of each sorting techniques
- Discuss the algorithms of various searching techniques
- Discuss Merge sort
- Discuss algorithms of sequential search, binary search and binary tree search.
- Analyze the performance of searching methods

2.3 SORTING

Sorting is very important in every computer application. Sorting refers to arranging of data elements in some given order. Many Sorting algorithms are available to sort the given set of elements.

We will now discuss two sorting techniques and analyze their performance. The two techniques are:

- Internal Sorting
- External Sorting

2.3.1 Internal Sorting

Internal Sorting takes place in the main memory of a computer. The internal sorting methods are applied to small collection of data. It means that, the entire collection of data to be sorted in small enough that the sorting can take place within main memory. We will study the following methods of internal sorting

1. Insertion sort
2. Selection sort
3. Merge Sort
4. Radix Sort
5. Quick Sort
6. Heap Sort
7. Bubble Sort

2.3.1.1 Insertion Sort

In this sorting we can read the given elements from 1 to n, inserting each element into its proper position. For example, the card player arranging the cards dealt to him. The player picks up the card and inserts them into the proper position. At every step, we insert the item into its proper place.

This sorting algorithm is frequently used when n is small. The insertion sort algorithm scans A from A[l] to A[N], inserting each element A[K] into its proper position in the previously sorted subarray A[l], A[2], . . . , A[K-1]. That is:

Pass 1. A[l] by itself is trivially sorted.
Pass 2. A[2] is inserted either before or after A[l] so that: A[l], A[2] is sorted.
Pass 3. A[3] is inserted into its proper place in A[l], A[2], that is, before A[l], between A[l] and A[2], or after A[2], so that: A[l], A[2], A[3] is sorted.
Pass 4. A[4] is inserted into its proper place in A[l], A[2], A[3] so that: A[l], A[2], A[3], A[4] is sorted.

……

Pass N. A[N] is inserted into its proper place in A[l], A[2], . . . , A[N - 1] so that: A[l], $A[2],\ldots,A[N]$ is sorted.

Example 2.1

Pass	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
K = 1:	−∞	(77)	33	44	11	88	22	66	55
K = 2:	−∞	77	(33)	44	11	88	22	66	55
K = 3:	−∞	33	77	(44)	11	88	22	66	55
K = 4:	−∞	33	44	77	(11)	88	22	66	55
K = 5:	−∞	11	33	44	77	(88)	22	66	55
K = 6:	−∞	11	33	44	77	88	(22)	66	55
K = 7:	−∞	11	22	33	44	77	88	(66)	55
K = 8:	−∞	11	22	33	44	66	77	88	(55)
Sorted:	−∞	11	22	33	44	55	66	77	88

Insertion sort for *n* = 8 items

Algorithm 2.1

INSERTION (A , N)

This algorithm sorts the array A with N elements

1. Set A[0] := --∞ . [initializes the element]
2. Repeat Steps 3 to 5 for K= 2,3, ……,N
3. Set TEMP := A[K] and PTR:= K-1
4. Repeat while TEMP < A[PTR]
   (a) Set A[PTR +1]:=A[PTR] [Moves element forward]
   (b) Set PTR := PTR-1
   [End of loop].
5. Set A[PTR+1] := TEMP [inserts element in proper place]
   [End of Step 2 loop]
6. Return

Complexity of Insertion Sort:

The insertion sort algorithm is a very slow algorithm when n is very large.

Algorithm	Worst Case	Average Case
Insertion Sort	$\frac{n(n-1)}{2} = O(n^2)$	$\frac{n(n-1)}{4} = O(n^2)$

Worst Case

The Worst Case occurs when the array A is in reverse order and the inner loop must use the maximum number of K-1 of comparisons.

$$f(n) = 1 + 2 + \ldots\ldots + (n-1) = \frac{n(n-1)}{2} = O(n^2)$$

Average Case

The average case occurs when there is (K-1) /2 comparisons in the inner loop.

$$F(n) = \tfrac{1}{2} + 2/2 + \ldots\ldots\ldots + \frac{(n-1)}{2} = \frac{n(n-1)}{4} = O(n^2)$$

2.3.1.2 Selection Sort

In this sorting we find the smallest element in this list and put it in the first position. Then find the second smallest element in the list and put it in the second position. And so on**.**

Pass 1. Find the location LOC of the smallest in the list of N elements A[l], A[2], . . . , A[N], and then interchange A[LOC] and [1] . Then A[1] is sorted.

Pass 2. Find the location LOC of the smallest in the sublist of N – 1 Elements A[2], A[3],. . . , A[N], and then interchangeA[LOC] and A[2]. Then:A[l], A[2] is sorted, since A[1]<A[2].

Pass 3. Find the location LOC of the smallest in the sublist of N – 2 elements A[3], A[4], . . . , A[N], and then interchange A[LOC] and A[3]. Then: A[l], A[2], . . . , A[3] is sorted, since A[2] < A[3].

……

Pass N - 1. Find the location LOC of the smaller of the elements A[N - 1), A[N], and then interchange A[LOC] and A[N- 1]. Then: A[l], A[2], . . . , A[N] is sorted, since A[N - 1] < A[N].
Thus A is sorted after N - 1 passes.

Example 2.2

Suppose an array A contains 8 elements as follows:

77, 33, 44, 11, 88, 22, 66, 55

Pass	A[l]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
K = l, LOC = 4	**77**	33	44	**11**	88	22	66	55
K = 2, LOC = 6	11	**33**	44	77	88	**22**	66	55
K = 3, LOC = 6	11	22	**44**	77	88	**33**	66	55
K = 4, LOC = 6	11	22	33	**77**	88	**44**	66	55
K = 5,LOC = 8	11	22	33	44	**88**	77	66	**55**
K = 6, LCjC = 7	11	22	33	44	55	**77**	**66**	88
K = 7, LOC = 7	11	22	33	44	55	66	**77**	88
Sorted:	11	22	33	44	55	66	77	88

Algorithm 2.2:

1. To find the minimum element

MIN (A, K , N, LOC)

An array A is in memory. This procedure finds the location LOC of the smallest element among A[K] , A[K+1],…….A[N].

1. Set MIN:= A[K] and LOC := K [Initializes pointers]
2. Repeat for J = K +1, K+2…
   If MIN > A [J] , then : Set MIN := A[J] and LOC := A[j] and LOC: = J
3. Return

2. To Sort the elements

SELECTION (A, N)

1. Repeat Steps 2 and 3 form K= 1,2, ….., N – 1
2. Call MIN(A,K,N,LOC)
3. [Interchange A[K] and A[LOC]]
   Set TEMP: = A [K], A [K]:= A [LOC] and A [LOC]:=TEMP
4. Exit.

Complexity of the Selection Sort Algorithm

First note that the number f(n) of comparisons in the selection sort algorithm is independent of the original order of the elements. Observe that MIN(A, K, N, LOC) requires n - K comparisons. That is, there are n - 1 comparisons during Pass 1 to find the smallest element, there are n - 2 comparisons during Pass 2 to find the second smallest element, and so on. Accordingly,

$f(n) = (n - 1) + (n - 2) + \ldots.. + 2 + 1 = n(n(-1)/2 = O(n^2)$

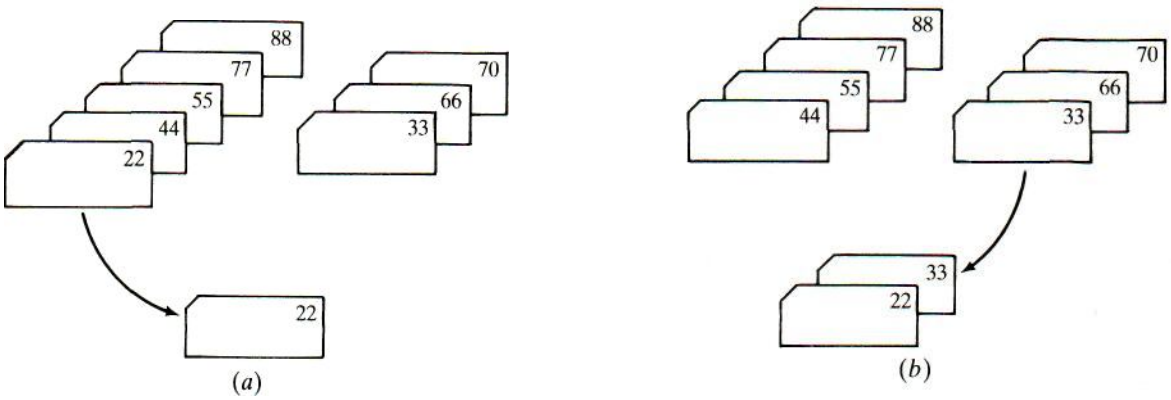
The above result is summarized in the following table:

Algorithm	Worst Case	Average Case
Selection Sort	$\frac{n(n-1)}{2} = O(n^2)$	$\frac{n(n-1)}{2} = O(n^2)$

2.3. 1.3 Merge Sort

Combing the two lists is called as merging. For example A is a sorted list with r elements and B is a sorted list with s elements. The operation that combines the elements of A and B into a single sorted list C with n = r + s elements is called merging. After combing the two lists the elements are sorted by using the following merging algorithm

Suppose one is given two sorted decks of cards. The decks are merged as in Fig. 2.1. That is, at each step, the two front cards are compared and the smaller one is placed in the combined deck. When one of the decks is empty, all of the remaining cards in the other deck are put at the end of the combined deck. Similarly, suppose we have two lines of students sorted by increasing heights, and suppose we want to merge them into a single sorted line. The new line is formed by choosing, at each step, the shorter of the two students who are at the head of their respective lines. When one of the lines has no more students, the remaining students line up at the end of the combined line.



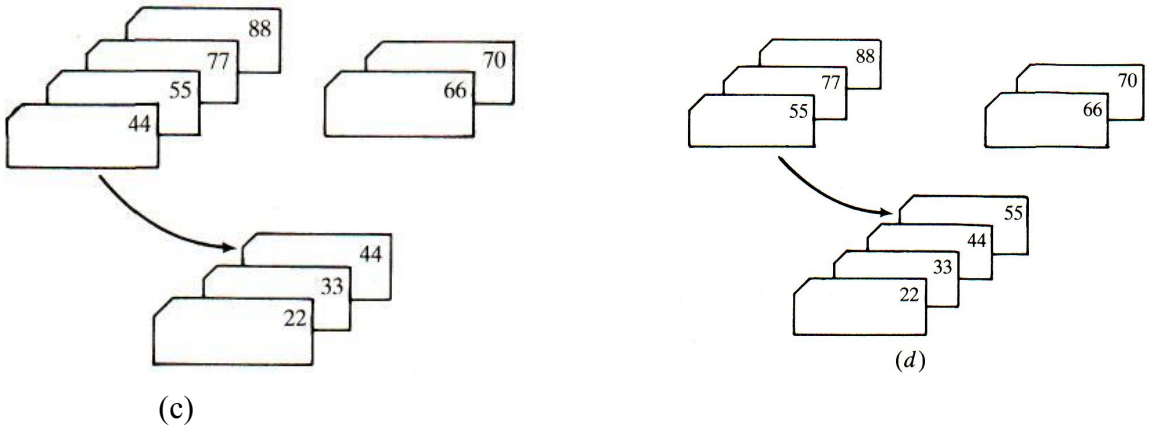


Fig 2.1

The above discussion will now be translated into a formal algorithm which merges a sorted r-element array A and a sorted s-element array B into a sorted array C, with n = r + s elements. First of all, we must always keep track of the locations of the smallest element of A and the smallest element of B which have not yet been placed in C. Let NA and NB denote these locations, respectively. Also, let PTR denote the location in C to be filled. Thus, initially, we set NA : = 1, NB : = 1 and PTR : = 1. At each step of the algorithm, we compare A[NA] and B[NB] and assign the smaller element to C[PTR]. Then we increment PTR by setting PTR:= PTR + 1, and we either increment NA by setting NA: = NA + 1 or increment NB by setting NB: = NB + 1, according to whether the new element in C has come from A or from B. Furthermore, if NA> r, then the remaining elements of B are assigned to C; or if NB > s, then the remaining elements of A are assigned to C.

Algorithm 2.3

MERGING (A, R, B, S, C)

Let A and B be sorted arrays with R and S elements. This algorithm merges A and B into an array C with N = R + S elements.

1. [Initialize] Set NA : = 1 , NB := 1 AND PTR : = 1
2. [Compare] Repeat while NA <= R and NB <= S
   If A[NA] < B[NB] , then
   (a)[Assign element from A to C] set C[PTR] := A[NA]
   (b)[Update pointers] Set PTR := PTR +1 and NA := NA +1
   Else
   (a) [Assign element from B to C] Set C[PTR] := B[NB]
   (b) [Update Pointers] Set PTR := PTR +1 and NB := NB +1
   [End of loop]
3. [Assign remaining elements to C]
   If NA > R , then
   Repeat for K = 0 ,1,2,……..,S- NB
   Set C[PTR+K] := B[NB+K]
   [End of loop]
   Else
   Repeat for K = 0,1,2,……,R-NA
   Set C[PTR+K] := A[NA+K]
   [End of loop]
4. Exit

The total computing time = $O(n \log_2 n)$.

The disadvantages of using mergesort is that it requires two arrays of the same size and type for the merge phase

2.3.1.4 Radix Sort

Radix sort is the method that many people intuitively use or begin to use when alphabetizing a large list of names. Specifically, the list of names is first sorted according to the first letter of each name. That is, the names are arranged in 26 classes, where the first class consists of those names that begin with "A," the second class consists of those names that begin with "B," and so on. During the second pass, each class is alphabetized according to the second letter of the name. And so on. If no name contains, for example, more than 12 letters, the names are alphabetized with at most 12 passes.

The radix sort is the method used by a card sorter. A card sorter contains 13 receiving pockets labeled as follows:

9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 11, 12, R (reject)

Each pocket other than R corresponds to a row on a card in which a hole can be punched. Decimal numbers, where the radix is 10, are punched in the obvious way and hence use only the first 10 pockets of the sorter. The sorter uses a radix reverse-digit sort on numbers. That is, suppose a card sorter is given a collection of cards where each card contains a 3-digit number punched in columns 1 to 3. The cards are first sorted according to the units digit. On the second pass, the cards are sorted according to the tens digit. On the third and last pass, the cards are sorted according to the hundreds digit. We illustrate with an example.

Example 2.3

Suppose 9 cards are punched as follows:

348, 143, 361, 423, 538, 128, 321, 543, 366

Given to a card sorter, the numbers would be sorted in three phases, as pictured in Fig. 2.2:

(a) In the first pass, the units digits are sorted into pockets. (The pockets are pictured upside down, so 348 is at the bottom of pocket 8.) The cards are collected pocket by pocket, from pocket 9 to pocket 0. (Note that 361 will now be at the bottom of the pile and 128 at the top of the pile.) The cards are now reinput to the sorter.

(b) In the second pass, the tens digits are sorted into pockets. Again the cards are collected pocket by pocket and reinput to the sorter.

(c) In the third and final pass, the hundreds digits are sorted into pockets.

Input	0	1	2	3	4	5	6	7	8	9
348									348	
143				143						
361		361								
423				423						
538									538	
128									128	
321		321								
543				543						
366							366			

(a) First pass.

Input	0	1	2	3	4	5	6	7	8	9
361							361			
321			321							
143					143					
423			423							
543					543					
366					543					
366							366			
348					348					
538				538						
128			128							

(b) Second pass.

Input	0	1	2	3	4	5	6	7	8	9
321				321						
423					423					
128		128								
538						538				
143		143								
543						543				
348				348						
361				361						
366				366						

(c) Third pass.

Figure: 2.2

When the cards are collected after the third pass, the numbers are in the following order:

128, 143, 321, 348, 361, 366, 423, 538, 543

Thus the cards are now sorted.

The number C of comparisons needed to sort nine such 3-digit numbers is bounded as follows:

$C \leq 9 * 3 * 10$

The 9 comes from the nine cards, the 3 comes from the three digits in each number, and the 10 comes from radix d = 10 digits.

Complexity of Radix Sort

Suppose a list A of n items $A_1, A_2, \ldots, A_n$ is given. Let d denote the radix (e.g., d = 10 for decimal digits, d = 26 for letters and d = 2 for bits), and suppose each item A_i is represented by means of s of the digits:

$A_i = d_{i1}\, d_{i2} \ldots. d_{is}$

The radix sort algorithm will require 5 passes, the number of digits in each item. Pass K will compare each d_{ik} with each of the d digits. Hence the number C(n) of comparisons for the algorithm is bounded as follows:

$C\{n) \leq d * s * n$

Although d is independent of n, the number s does depend on n. In the worst case, s = n, so C(n) = $O(n^2)$. In the best case, s = $\log_d n$, so C(n) = O(n log n). In other words, radix sort performs well only when the number s of digits in the representation of the A_i's is small.

Another drawback of radix sort is that one may need d*n memory locations. This comes from the fact that all the items may be "sent to the same pocket" during a given pass. This drawback may be minimized by using linked lists rather than arrays to store the items during a given pass. However, one will still require 2*n memory locations.

2.3.1.5 Quick Sort

This is the most widely used internal sorting algorithm. It is based on divide-and-conquer type i.e. Divide the problem into sub-problems, until solved sub problems are found.

Example 2.4

Suppose A is the following list of 12 numbers:

(44) 33, 11, 55, 77, 90, 40, 60, 99, 22, 88, 66

The quick sort algorithm finds the final position of one of the numbers; in this illustration, we use the first number, 44. This is accomplished as follows. Beginning with the last number, 66, scan the list from right to left, comparing each number with 44 and stopping at the first number less than 44. The number is 22. Interchange 44 and 22 to obtain the list

(22) 33, 11, 55, 77, 90, 40, 60, 99, (44) 88, 66

(Observe that the numbers 88 and 66 to the right of 44 are each greater than 44.) Beginning with 22, next scan the list in the opposite direction, from left to right, comparing each number with 44 and stopping at the first number greater than 44. The number is 55. Interchange 44 and 55 to obtain the list

22, 33, 11, (44) 77, 90, 40, 60, 99, (55), 88, 66 .

(Observe that the numbers 22, 33 and 11 to the left of 44 are each less than 44.) Beginning this time with 55, now scan the list in the original direction, from right to left, until meeting the first number less than 44. It is 40. Interchange 44 and 40 to obtain the list

22, 33, 11, (40) 77, 90, (44) 60, 99, 55, 88, 66*

(Again, the numbers to the right of 44 are each greater than 44.) Beginning with 40, scan the list from left to right. The first number greater than 44 is 77. Interchange 44 and 77 to obtain the list

22, 33, 11, 40, (44) 90, (77) 60, 99, 55, 88, 66

(Again, the numbers to the left of 44 are each less than 44.) Beginning with 77, scan the list from right to left seeking a number less than 44. We do not meet such a number before meeting 44. This means all numbers have been scanned and compared with 44. Furthermore, all numbers less than 44 now form the sublist of numbers to the left of 44, and all numbers greater than 44 now form the sublist of numbers to the right of 44, as shown below:

22, 33, 11, 40, (44) 90, 77, 60, 99, 55, 88, 66

First sublist Second sublist

Thus 44 is correctly placed in its final position, and the task of sorting the original list A has now been reduced to the task of sorting each of the above sublists.

The above reduction step is repeated with each sublist containing 2 or more elements.

Algorithm 2.4

This algorithm sorts an array A with N elements

1. [Initialize] TOP := NULL
2. If N >1 , then TOP := TOP +1 , LOWER[1] := 1 , UPPER[1] :=N
3. Repeat Steps 4 to 7 while TOP ≠ NULL
4. Set BEG := LOWER[TOP] , END := UPPER[TOP],TOP:=TOP-1
5. Call QUICK(A,N,BEG,END,LOC)
6. If BEG < LOC -1 then
   TOP := TOP +1 , LOWER [TOP] := BEG
   UPPER[TOP] = LOC-1
   End If
7. If LOC +1 < END then
   TOP := TOP +1 , LOWER[TOP] := LOC +1
   UPPER[TOP] := END
   End If
8. Exit

The Quick sort algorithm uses the $O(N \log_2 N)$ comparisons on average.

2.3.1.6 Heap Sort

A heap is a complete binary tree, in which each node satisfies the heap condition.

Heap condition

The key of each node is greater than or equal to the key in its children. Thus the root node will have the largest key value.

MaxHeap

Suppose H is a complete binary tree with n elements. Then H is called a heap or maxheap, if the value at N is greater than or equal to the value at any of the children of N.

MinHeap

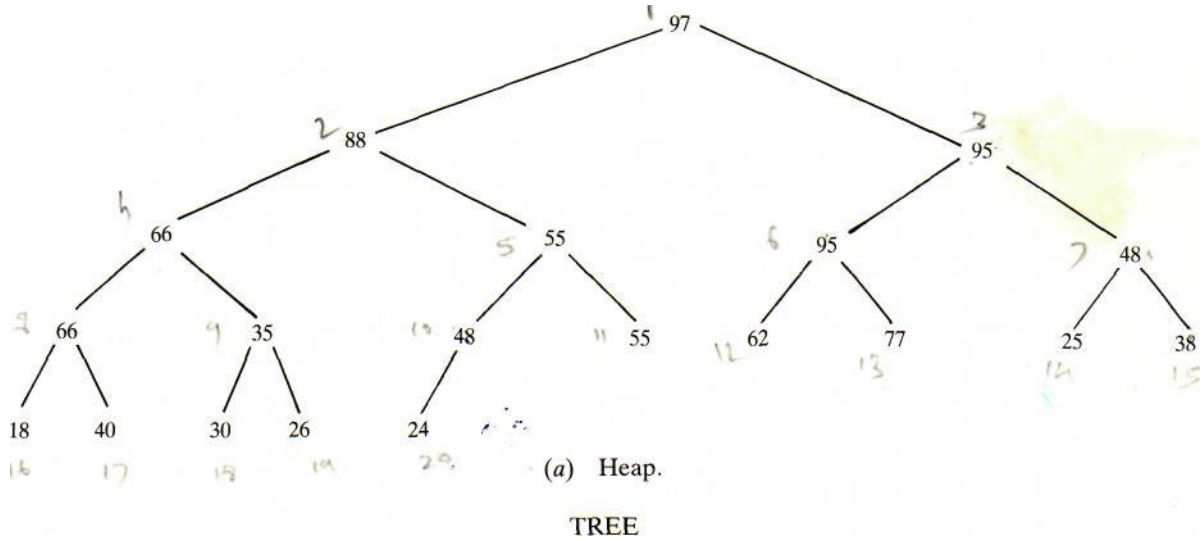
The value at N is less than or equal to the value at any of the children of N.

The operations on a heap

(i) New node is inserted into a Heap
(ii) Deleting the Root of a Heap

Example 2.5

Consider the complete tree H in Fig.2.3 (a) . Observe that H is a heap. Figure 2.3 (b) shows the sequential representation of H by the array TREE. That is, TREE[1] is the root of the tree H, and the left and right children of node TREE[K] are, respectively, TREE[2K] and TREE[2K + 1]. This means, in particular, that the parent of any nonroot node TREE[J] is the node TREE[J / 2] (where J / 2 means integer division). Observe that the nodes of H on the same level appear one after the other in the array TREE.



(a) Heap.

TREE

The sequential representation of H by the array TREE

97	88	95	66	55	95	48	66	35	48	55	62	77	25	38	18	40	30	26	24
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

(b) **Sequential representation**

Fig. 2.3

TREE[1] is the root of the tree H, and the left and right children of node TREE[K] are, respectively, TREE[2K] and TREE[2K + 1]. This means, in particular, that the parent of any nonroot node TREE[J] is the node TREE[J / 2] (where J / 2 means integer division). Observe that the nodes of H on the same level appear one after the other in the array TREE.

Inserting into a Heap

Inserting a new element into a heap tree. Suppose H is a heap with N elements. We insert ITEM into the heap H as follows:

(i) First join ITEM at the end of H.

(ii) Then the ITEM rise to its "appropriate place" in H.

Example 2.6

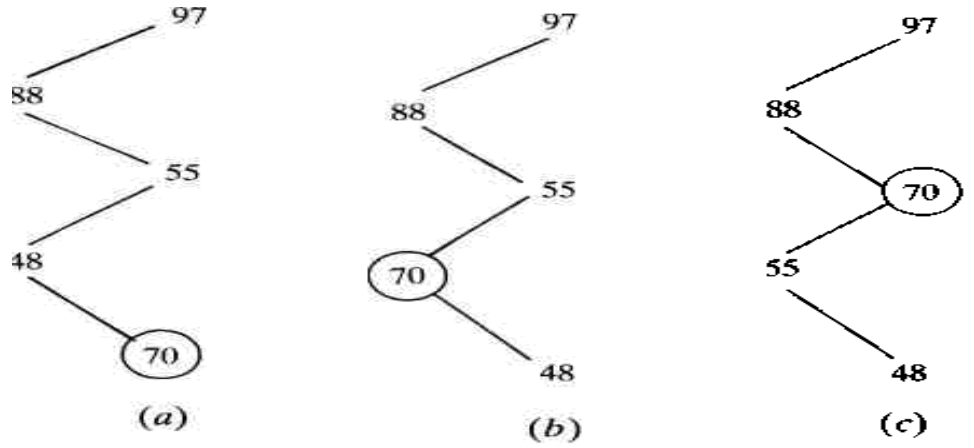
To add ITEM = 70 to H. First we adjoin 70 as the next element in the complete tree; that is, we set TREE[21] = 70. Then 70 is the right child of TREE[10] = 48.The path from 70 to the root of H is pictured in Fig 2.4(a). We now find the appropriate place of 70 in the heap as follows:

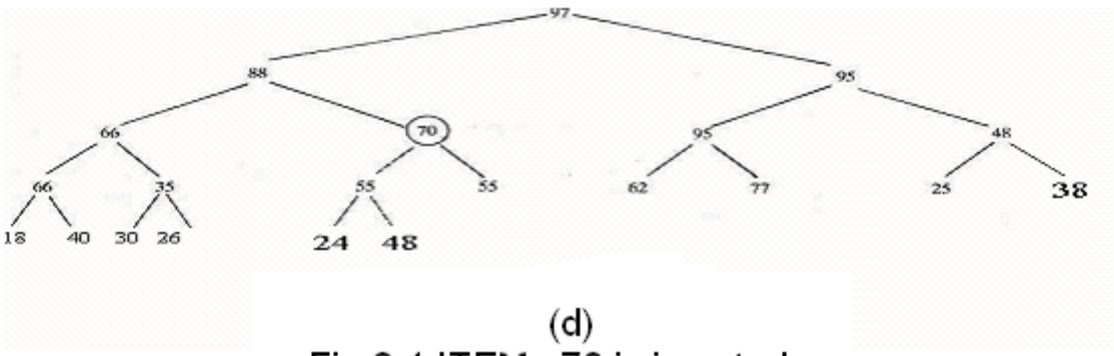
(a) Compare 70 with its parent, 48. Since 70 is greater than 48, interchange 70 and 48; Fig. 2.4(b).

(b) Compare 70 with its new parent, 55. Since 70 is greater than 55, interchange 70 and 55; the path will now look like Fig 2.4(c)

(c) Compare 70 with its new parent, 88. Since 70 does not exceed 88, ITEM = 70 has risen to its appropriate place in H.

The following figure shows the final tree. A dotted line indicates that an exchange has taken place.





(d)

Fig 2.4 ITEM =70 is inserted

Algorithm 2.5

INSHEAP (TREE, N, ITEM)

A heap H with N elements is stored in the array TREE, and an ITEM of information is given. This procedure inserts ITEM as a new element of H . PTR gives the location of ITEM as it rises in the tree, and PAR denotes the location of the parent of ITEM.

1. [Add new node to H and initialize PTR].
   Set N := N +1 and PTR := N
2. [Find location to insert ITEM]
   Repeat steps 3 to 6 while PTR <1
3. Set PAR := [PTR/2] [Location of parent node]
4. If ITEM <= TREE[PAR], then
   Set TREE[PTR] := ITEM , and return
   End If
5. Set TREE[PTR] := TREE[PAR] [Moves node down]
6. Set PTR := PAR [Updates PTR]

End Loop

7. [Assign ITEM as the root of H]
8. Return

Deleting the Root of a Heap

Deleting the root element from a heap tree. Suppose H is a heap with N elements. We delete the root R from the heap H as follows:

(i) Assign the root R to some variable ITEM
(ii) Replace the deleted node R by the last node L of H
(iii) L sinks to its appropriate place in H.

Example 2.7

Consider the heap H in Fig 2.5(a), where R = 95 is the root and L = 22 is the last node of the tree. Step 1 of the above procedure deletes R = 95, and Step 2 replaces R = 95 by L = 22.

(a) Compare 22 with its two children, 85 and 70. Since 22 is less than the larger child, 85, interchange 22 and 85..(Fig. 2.5 (c))
(b) Compare 22 with its two new children, 55 and 33. Since 22 is less than the larger child, 55, interchange 22 and 55 . (Fig.2.5 (d))
(c) Compare 22 with its new children, 15 and 20.Since 22 is greater than both children, node 22 has dropped to its appropriate place in H.

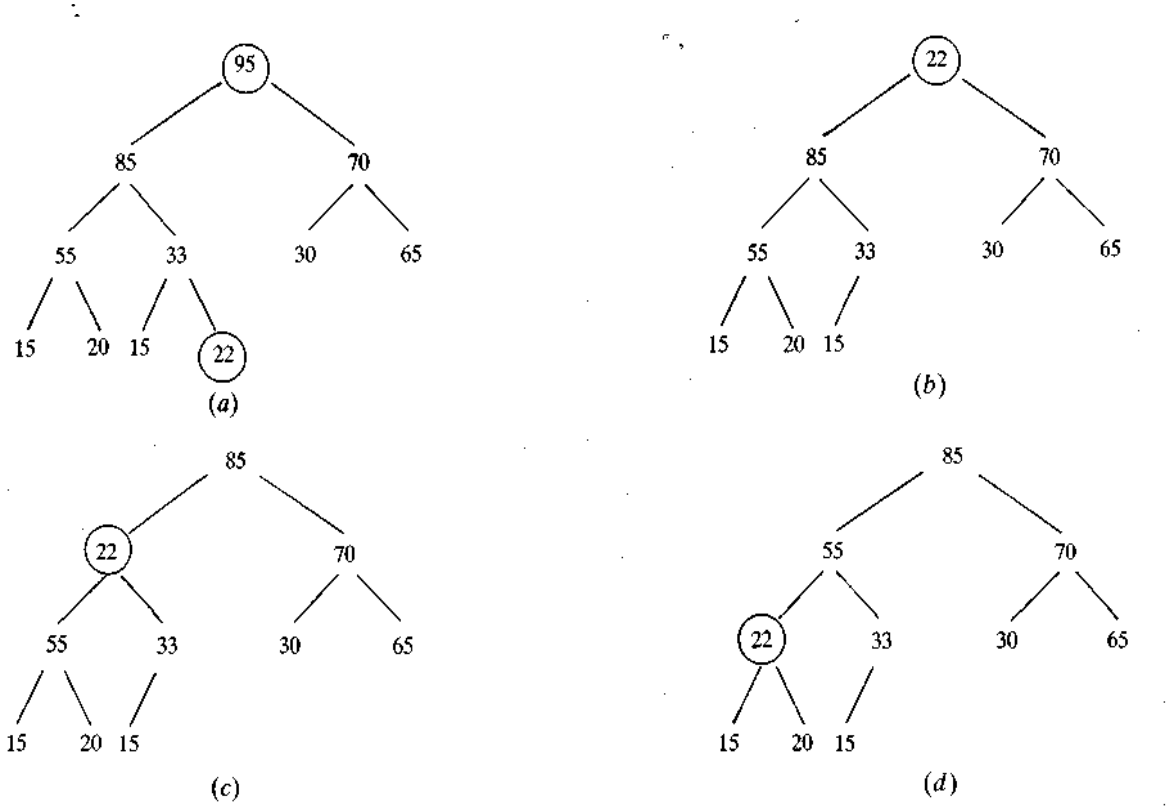


Fig. 2.5 Reheaping

Algorithm 2.6

DELHEAP (TREE, N, ITEM)

A heap H with N elements is stored in the array TREE. This procedure assigns the root TREE [1] of H to the variable ITEM and then reheaps the remaining elements. The variable LAST saves the value of the original last node of H. The pointers PTR, LEFT and RIGHT give the locations of LAST and its left and right children as LAST sinks in the tree.

1. Set ITEM := TREE[1] [removes root of H]
2. Set LAST := TREE [N] and N:= N-1 {Removes last node of H]
3. Set PTR := 1 , LEFT := 2 and RIGHT := 3 [Initializes pointers]
4. Repeat steps 5 to 7 while RIGHT <= N
5. If LAST >= TREE [LEFT] and LAST >=TREE [RIGHT], then
   Set TREEE [PTR] := LAST and return
   End if
6. If TREE [RIGHT] <= TREE [LEFT], then
   Set TREE [PTR] := TREE[LEFT] and PTR := LEFT
   Else
   Set TREE [PTR]:= TREE [RIGHT] and PTR:= RIGHT
   End If
7. Set LEFT := 2 * PTR AND RIGHT := LEFT +1
   [End of Step 4 loop]
8. If LEFT=N and if LAST < TREE[LEFT] , then Set PTR := LEFT
9. Set TREE[PTR]:=LAST
10. Return.

2.3.3.6 Bubble Sort

In this sorting algorithm, multiple swapping take place in one iteration. Smaller elements move or 'bubble' up to the top of the list. In this method ,we compare the adjacent members of the list to be sorted , if the item on top is greater than the item immediately below it, they are swapped.

Example 2.8

Suppose the following numbers are stored in an array A:

32, 51, 27, 85, 66, 23, 13, 57

We apply the bubble sort to the array A. We discuss each pass separately.

Pass 1. We have the following comparison:

(a) Compare Al and A2 Since 32 < 51, the list is not altered.
(b) Compare A2 and A3 Since 51 > 27, interchange 51 and 27 as follows:
32, (27), (51), 85, 66, 23, 13, 57
(c) Compare A3 and A4. Since 51 < 85, the list is not altered.
(d) Compare A4 and A5. Since 85 > 66, interchange 85 and 66 as follows:
32, 27, 51, (66), (85), 23, 13, 57
(e) Compare A5 and A6. Since 85 > 23, interchange 85 and 23 as follows:
32, 27, 51, 66, (23), (85), 13, 57
(f) Compare A6 and A7. Since 85 > 13, interchange 85 and 13 as follows:
32, 27, 51, 66, 23, (13), (85), 57
(g) Compare A7 and A8. Since 85 > 57, interchange 85 and 57 as follows:
32, 27, 51, 66, 23, 13, (57), (**85**)

At the end of this first pass, the largest number, 85, has moved to the last position. However, the rest of the numbers are not sorted, even though some of them have changed their positions. For the remainder of the passes, we show only the positions of the

numbers in the array.

At the end of Pass 2, the second largest number, 66, has moved its way down to the next-to-last position.
27, 33, 51, 23, 13, 57, **66**, **85**

At the end of Pass 3, the third largest number, 57, has moved its way down to its position in the list.
27, 33, 23, 13, 51, **57**, **66**, **85**

At the end of Pass 4, the fourth largest number, 51, has moved its way down to its position in the list.
27, 23, 13, 33, **51**, **57**, **66**, **85**

At the end of Pass 5, the fifth largest number, 33, has moved its way down to its position in the list.
23, 13, 27, **33**, **51**, **57**, **66**, **85**

At the end of Pass 6, the sixth largest number, 27, has moved its way down to its position in the list.
13, 23, **27**, **33**, **51**, **57**, **66**, **85**

At the end of Pass 7 (last pass), the seventh largest number, 23, has moved its way down to its position in the list.
13, **23**, **27**, **33**, **51**, **57**, **66**, **85**

Algorithm 2.7

BUBBLE (DATA, N)

Here DATA is an array with N elements. This algorithm sorts the elements in DATA.

1. Repeat Steps 2 and 3 for K = 1 to N-1
2. Set PTR := 1 [Initialize pass pointer PTR]
3. Repeat while PTR <= N-K : [Execute Pass]
   (a) If DATA[PTR] > DATA[PTR+1], then
   Interchange DATA[PTR] and DATA[PTR+1]
   End if
   (b) Set PTR := PTR+1
   [End of inner loop]
   [End of step 1 outer loop]
4. Exit.

The total number of comparisons in Bubble sort are:

= (N-1)+(N-2)….+2+1

=(N-1)*N/2=O(N^2)

The time required to execute the bubble sort algorithm is proportional to n^2 , where n is the number of input items. The Bubble sort algorithm uses the O(n^2) comparisons on average

2.3.2 EXTERNAL SORT

The External sorting methods are applied only when the number of data elements to be sorted is too large. These methods involve as much external processing as processing in the CPU. To study the external sorting, we need to study the various external devices used for storage in addition to sorting algorithms. This sorting requires auxiliary storage.The following are the examples of external sorting

- Sorting with Disk

- Sorting with Tapes

2.3.2.1 Sorting with Disks

We will first illustrate merge sort using disks. The following example illustrate the concept of sorting with disks

The file F containing 600 records is to be sorted. The main memory is capable of sorting of 1000 records at a time. The input file F is stored on one disk and we have in addition another scratch disk. The block length of the input file is 500 records.

We see that the file could be treated as 6 sets of 1000 records each. Each set is sorted and stored on the scratch disk as a run. These 5 runs will then be merged as follows.

Allocate 3 blocks of memory each capable of holding 500 records. Two of these buffers B1 and B2 will be treated as input buffers and the third B3 as the output buffer. We have now the following

1) 6 run R1, R2, R3, R4, R5, R6 on the scratch disk.
2) 3 buffers B1,B2 and B3
    - Read 500 records from R1 into B1.
    - Read 500 records from R2 into B2.
    - Merge B1 and B2 and write into B3.
    - When B3 is full- write it out to the disk as run R11.
    - Similarly merge R3 and R4 to get run R12.
    - Merge R5 and R6 to get run R13.

Thus, from 6 runs of size 1000 each, we have now 3 runs of size 2000 each.
The steps are repeated for steps R11 and R12 to get a run of size 4000.
This run is merged with R13 to get a single sorted run of size 6000.

2.3.2.2 Sorting with Tapes

Sorting with tapes is essentially similar to the merge sort used for sorting with disks. The differences arise due to the sequential access restriction of tapes. This makes the selection time prior to data transmission an important factor, unlike seek time and latency time. Thus is sorting with tapes we will be more concerned with arrangement of blocks and runs on the tape so s to reduce the selection or across time.

Example

A file of 6000 records is to be sorted. It is stored on a tape and the block length is 500. The main memory can sort unto a 1000 records at a time. We have in addition 4 search tapes T1-T4.

Check Your Progress 1

1. Define Sorting.
2. What is internal sorting?
3. Given 2 sorted list of size ‘m’ and ‘n’ respectively. The number of comparisons needed in the worst case by the merge sort algorithm will be
   (a) mn (b) max(m,n) (c) min(m,1) (d) m+n-1
4. Sorting is useful for
   (a) report generation (b) minimizing the storage needed
   (c) making searching easier and efficient (d) responding to queries easily
5. Choose the correct statement
   (a)Internal sorting is used if the number of items to be sorted is very large
   (b)External sorting is used if the number of items to be sorted in very large
   (c)External sorting needs auxiliary storage
   (d)Internal sorting needs auxiliary storage.
6. The way a card game player arranges his cards as he picks them up one by one is an example of
   (a) bubble sort (b) selection sort (c) insertion sort (d) merge sort

7. You are asked to sort 15 randomly .you should prefer
   (a) bubble sort (b) quick sort (c) merger sort (d) heap sort
8. Describe Heap.
9. The maximum number of comparisons needed to sort 7 items (each item is 4 digit) using radix sort is
   (a) 280 (b) 40 (c) 47 (d) 38
10. What is the difference between MinHeap and MaxHeap?
11. Which of the following algorithm design technique is used in the quick sort?
   (a) Dynamic programming
   (b) BackTracking
   (c) Divide and conquer
   (d) Greedy method
12. The number of swapping needed to sort the numbers 8, 22,7,9,31,19,5,13 in ascending order, using bubble sort is
   (a) 11 (b) 12 (c) 13 (d) 14
13. What is merging?

2.4 SEARCHING

Searching refers to the operation of finding the location of a given item in a collection of items. The search is said to be successful if ITEM does appear in DATA and unsuccessful otherwise.

The following searching algorithms are discussed in this chapter.

1. Sequential Searching
2. Binary Search
3. Binary Tree Search

2.4.1 Sequential Search

This is the most natural searching method. The most intuitive way to search for a given ITEM in DATA is to compare ITEM with each element of DATA one by one. .The algorithm for a sequential search procedure is now presented

Algorithm 2.8

```
SEQUENTIAL SEARCH
INPUT :  List of Size N.  Target Value T
OUTPUT : Position of T in the list-1
BEGIN
   Set FOUND = false
   Set I := 0
    While (I <= N) and (FOUND is false)
        IF List[i] ==t THEN
                FOUND = true
        ELSE
                I = I+1
    IF FOUND==false THEN
            T is not present in the List
END
```

2.4.2 Binary Search

Suppose DATA is an array which is sorted in increasing numerical order. Then there is an extremely efficient searching algorithm, called binary search, which can be used to find the

location LOC of a given ITEM of information in DATA.

The binary search algorithm applied to our array DATA works as follows. During each stage of our algorithm, our search for ITEM is reduced to a segment of elements of DATA: DATA[BEG], DATA[BEG + 1], DATA[BEG + 2], ... DATA[END].

Note that the variable BEG and END denote the beginning and end locations of the segment respectively. The algorithm compares ITEM with the middle element DATA[MID] of the segment, where MID is obtained by

MID = INT((BEG + END) / 2)

(We use INT(A) for the integer value of A.) If DATA[MID] = ITEM, then the search is successful and we set LOC: = MID. Otherwise a new segment of DATA is obtained as follows:

(a) If ITEM < DATA[MID], then ITEM can appear only in the left half of the segment: DATA[BEG],DATA[BEG + 1],….. ,DATA[MID - 1]
So we reset END := MID - 1 and begin searching again.

(b) If ITEM > DATA[MID], then ITEM can appear only in the right half of the segment: DATA[MID + 1], DATA[MID + 2],....,DATA[END]
So we reset BEG := MID + 1 and begin searching again.
Initially, we begin with the entire array DATA; i.e. we begin with BEG = 1 and END = n, If ITEM is not in DATA, then eventually we obtain END< BEG

This condition signals that the search is unsuccessful, and in this case we assign LOC: = NULL. Here NULL is a value that lies outside the set of indices of DATA. We now formally state the binary search algorithm.

Algorithm 2.9: (Binary Search) BINARY(DATA, LB, UB, TEM, LOC)
Here DATA is sorted array with lower bound LB and upper bound UB, and ITEM is a given item of information. The variables BEG, END and MID denote, respectively, the beginning, end and middle locations of a segment of elements of DATA. This algorithm finds the location LOC of ITEM in DATA or sets LOC=NULL.

```
1. [Initialize segment variables.]
    Set BEG := LB, END := UB and MID = INT((BEG + END)/ 2).
    2. Repeat Steps 3 and 4 while BEG ≤ END and
                                    DATA[MID] ≠ ITEM.
        3. If ITEM<DATA[MID], then:
                Set END := MID - 1.
           Else:
                Set BEG := MID + 1.
           [End of If structure]
        4. Set MID := INT((BEG + END)/2).
    [End of Step 2 loop.]
    5. If DATA[MID] :=ITEM, then:
            Set LOC:=MID.
       Else:
            Set LOC := NULL.
       [End of If structure.]
     6. Exit.
```

Example 2.9

Let DATA be the following sorted 13-element array:

DATA: 11, 22, 30, 33, 40, 44, 55, 60, 66, 77, 80, 88, 99

We apply the binary search to DATA for different values of ITEM.

(a) Suppose ITEM = 40. The search for ITEM in the array DATA is pictured in where the values of DATA[BEG] and DATA[END] in each stage of the algorithm are indicated by parenthesis and the value of DATA[MID] by a bold. Specifically, BEG, END and MID will have the following successive values:

(1) Initially, BEG = 1 and END 13. Hence,

MID = INT[(1 + 13) / 2] = 7 and so DATA[MID] = 55

(2) Since 40 < 55, END = MID – 1 = 6. Hence,

MID = INT[(1 + 6) / 2] = 3 and so DATA[MID] = 30

(3) Since 40 > 30, BEG = MID + 1 = 4. Hence,

MID = INT[(4 + 6) / 2] = 5 and so DATA[MID] = 40

The search is successful and LOC = MID = 5.

(1) (11), 22, 30, 33, 40, 44, **55**, 60, 66, 77, 80, 88, (99)
(2) (11), 22, **30**, 33, 40, (44), 55, 60, 66, 77, 80, 88, 99
(3) 11, 22. 30, (33), **40**, (44), 55, 60, 66, 77, 80, 88, 99 [Successful]

Complexity of the Binary Search Algorithm

The complexity is measured by the number of comparisons f(n) to locate ITEM in DATA where DATA contains n elements. Observe that each comparison reduces the sample size in half. Hence we require at most f(n) comparisons to locate ITEM where

$f(n) = [\log_2 n] + 1$

That is the running time for the worst case is approximately equal to $\log_2 n$. The running time for the average case is approximately equal to the running time for the worst case.

Limitations of the Binary Search Algorithm

The algorithm requires two conditions:

(1) the list must be sorted and

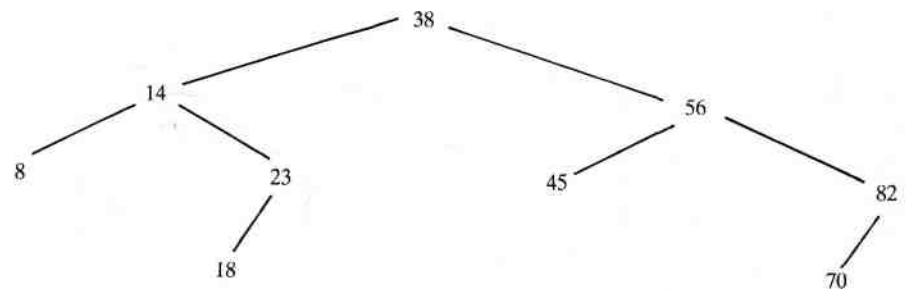
(2) one must have direct access to the middle element in any sublist.

2.4.3 Binary Search Tree

Suppose T is a binary tree. Then T is called a binary search tree if each node N of T has the following property:

"The value at N is greater than every value in the left subtree of N and is less than every value in the right subtree of N.

Example 2.10



Binary Search Tree(T)

Fig. 2.6

SEARCHING AND INSERTING IN BINARY SEARCH TREES

Suppose an ITEM of information is given. The following algorithm finds the location of ITEM in the binary search tree T, or inserts ITEM as a new node in its appropriate place in the tree.

(a) Compare ITEM with the root node N of the tree.
- (i) If ITEM < N, proceed to the left child of N.
- (ii) If ITEM > N, proceed to the right child of N.

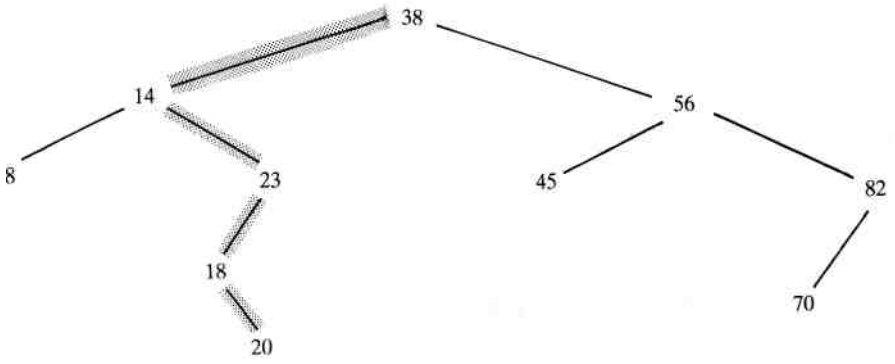
(b) Repeat Step (a) until one of the following occurs:
- (i) We meet a node N such that ITEM = N. In this case the search is successful.
- (ii) We meet an empty subtree, which indicates that the search is unsuccessful, and we insert ITEM in place of the empty subtree.

In other words, proceed from the root R down through the tree T until finding ITEM in T or inserting ITEM as a terminal node in T.

Example 2.11

Consider the binary search tree T in Fig. 2.6 . Suppose ITEM = 20 is given. Compare ITEM = 20 with the root, 38, of the tree T. Since 20 < 38, proceed to the left child of 38, which is 14.

1. Compare ITEM = 20 with 14. Since 20 > 14, proceed to the right child of 14, which is 23.
2. Compare ITEM = 20 with 23. Since 20 < 23, proceed to the left child of 23, which is 18.
3. Compare ITEM = 20 with 18. Since 20 > 18 and 18 does not have a right child, insert 20 as the right child of 18.



ITEM = 20 inserted.

DELETING IN A BINARY SEARCH TREE

Suppose T is a binary search tree, and suppose an ITEM of information is given. This section gives an algorithm which deletes ITEM from the tree T.

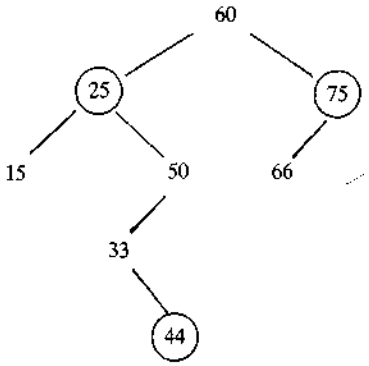
Case 1. N has no children. Then N is deleted from T by simply replacing the location of N in the parent node P(N) by the null pointer.

Case 2. N has exactly one child. Then N is deleted from T by simply replacing the location of

N in P(N) by the location of the only child of N.

Case 3. N has two children. Let S(N) denote the inorder successor of N. (The reader can verify that S(N) does not have a left child.) Then N is deleted from T by first deleting S(N) from T (by using Case 1 or Case 2) and then replacing node N in T by the node S(N).

Observe that the third case is much more complicated than the first two cases. In all three cases, the memory space of the deleted node N is returned to the AVAIL list.



(a) Before deletions.

ROOT

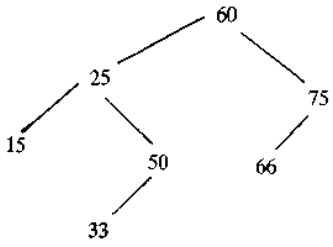
3

AVAIL

5

1	33	0	9
2	25	8	10
3	60	2	7
4	66	0	0
5		6	
6		0	
7	75	4	0
8	15	0	0
9	44	0	0
10	50	1	0

(b) Linked representation.



(a) Node 44 is deleted

ROOT

3

AVAIL

9

	INFO	LEFT	RIGHT
1	33	0	0
2	25	8	10
3	60	2	7
4	66	0	0
5		6	
6		0	
7	75	4	0
8	15	0	0
9		5	
10	50	1	0

(b) Linked representation

Check Your Progress 2

1. What is searching?
2. Distinguish between sequential search and binary search.
3. Define binary search tree.
4. Which of the following traversal techniques list the nodes of a binary search tree in ascending order?

   (a) post-order (b) in-order (c) pre-order (d) none of the above
5. The average successful search time for sequential search time taken by binary search on a sorted array of 10 item is

   (a) 2.6 (b) 2.7 (c) 2.8 (d) 2.9

2.5 SUMMARY

Sorting is an important application activity. Many internal sorting algorithms are available. Each algorithm is most effective and efficient for a particular situation or a particular set of data. The choice of selecting the particular algorithm based on the performance of the application.

External sorting is an important activity in large businesses. The choice of an external Sorting algorithm is depends on external system considerations.

The sequential search method was seen to be easy to implement and relatively efficient to use small lists. But very time consuming for long unsorted lists. The binary search method is an improvement, in that it eliminates half the list from consideration at each iteration.

2.6 SOLUTIONS/ANSWERS

Check your progress 1

1. Sorting refers to arranging of data elements in some given order.
2. Internal Sorting takes place in the main memory of a computer. The internal sorting methods applied to small collection of data.
3. d) m+n-1
   Each comparison puts 1 element in the final sorted array. So, in worst case m+n-1 comparison are necessary
4. (a) report generation (c) making searching easier and efficient (d) responding to queries easily
5. (b) External sorting is used if the number of items to be sorted is very large (c) External sorting needs auxiliary storage
6. (c) Insertion sort
7. (a) bubble sort
   Reason: n is small
8. A heap is a complete binary tree, in which each node satisfies the heap condition. The key of each node is greater than or equal to the key in its children. Thus the root node will have the largest key value.
9. (a) 280
   Reason: The maximum number of comparison is number of items * radix * number of digits
   7 * 10 * 4 = 280
10. **MaxHeap:** Suppose H is a complete binary tree with n elements. Then H is called a heap or maxheap, if the value at N is greater than or equal to the value at any of the children of N.
    MinHeap: The value at N is less than or equal to the value at any of the children of N.
11. (c) Divide and conquer
    Reason: Quick sort is based on divide-and-conquer type i.e. Divide the problem into sub-problems, until solved sub problems are found.
12. (d) 14
13. Combing the two lists is called as merging

Check your progress 2

1. Searching refers to the operation of finding the location of a given item in a collection of items.
2. Sequential Search: To compare ITEM with each element of DATA one by one
   Binary Search: search for ITEM is reduced to a segment of elements of DATA
3. Binary Search Tree: The value at N is greater than every value in the left subtree of N and is

less than every value in the right subtree of N.

4. (b) in-order
5. (d) 2.9

2.7 FURTHER READINGS

1. Fundamentals of Data Structures in C by Horowitz & Sahni
2. Fundamentals of Computer Algorithms by Horowitz & Sahni
3. Data structures and algorithms by Alfred V.Aho, John E.Hopcroft & Jeffrey D.Ullman