UNIT – V

Searching and Sorting: Searching – sequential searching, binary searching, hashing. Sorting – selection sort, bubble sort, quick sort, heap sort, merge sort, and insertion sort, efficiency considerations.

UNIT – V

CbH

Chapter ▸ 6

Searching

6.1 Introduction

Searching is a procedure to search a particular element in a DATA (i.e. Collection of data elements in memory)

Suppose we have a list of items, in which a specific Item of information is given. Searching refers to the operation of finding the locations LOC of ITEM in list, or printing some message that ITEM does not appear there. The search is said to be successful if Item does appear in list and unsuccessful otherwise.

Number of algorithm are defined for searching. The complexity of searching algorithms is measured in terms of the number f (n) of comparisions required to find ITEM in list where list contain n elements.

6.2 Linear Search

Also known as sequential search. A linear search is the basic and simple search algorithm.A linear search searches an element or value in a list/array till the desired element or value is not found and it searches in a sequence order. It compares the element with all other elements given in the list and if the element is matched it returns the value index else it return -1 linear search is applied on the unsorted list taken there are fewer elements in a list.

Example :

8	3	6	2	5

To search the element 5 in above list, it will go step by step in a sequence order.

Program in C.

```
function find Index (value, target)
{   For (var i = 0; i < value length; i + +)
    { if (value [i] = target)
{ return   i ;
   }
}
return -1;
}
```

Write a program to search an element in an array using Linear search

```
# include <stdio.h>
# include <conio.h>
int main()
{
```

```
int arr[10], num, I,n,beg,end,mid,found=0;
clrscr();
printf(" \n Enter the number of elements in the array: ");
scanf(" %d", &n);
printf(" \n Enter the elements: ");
for(i=0;i<n;i++)
{
scanf(" %d", &arr[i]);

}
printf("\n Enter the number that has to be searched :");
scanf("%d", &num);
for(i=0;i<n;i++)
{
 if (arr[i]== num)
 {
  found=-1;
  Pos=I;
  printf(" \n %d  is found in the array at position =%d", num,i);
  break;
 }
 if (found==0)
  printf(" \n %d doesnot exist in the array", num);
 getch();
 return 0;
}
```

6.2.1 Complexity of Linear Search

Since this algorithm Compares every element to find the required element in list.

6.2.1.1 Worst Case

Worst case, occur when one must search through the entire list/array i.e. when ITEM does not appear in List. In this case, algorithm requires comparision.

$f_1(n) = n + 1$ $f(n)$ = no's of comparision
(i.e. one more comparision required than the total no's of elements).
thus, in worst case, the running time is proportional to n.

6.2.1.2 Average Case

Let P_k is the probability that ITEM appear in list [k], and suppose q is the probability that ITEM does not appear in list (then $P_1 + P_2 + \ldots + q = 1$]. Since the algorithm uses k, Comparisions when ITEM appears in List [k], the average number of comparision is given by:

$$f(n) = 1 \cdot p_1 + 2 \cdot p_2 + \ldots n \cdot p_n + (n+1) \cdot q$$

In particular, suppose q is very small and ITEM appears with equal probability in each element of List, Then q=0 and each p_i = 1/n accordingly.

$$f(n) = 1 \cdot \frac{1}{n} + 2 \cdot \frac{1}{n} + \ldots . n \cdot \frac{1}{n} + (n+1) \cdot 0 = (1 + 2 + \ldots . + n) \frac{1}{n}$$

$$= \frac{n(n+1)}{2} \cdot \frac{1}{n} = \frac{n+1}{2}$$

That is, in this special case, the average number of Comparisions required to find the location of ITEM is approx, equal to half the number of elements in the list. Complexity in average case is O (n).

6.3 Binary Search

Generally, to find a value in unsorted array, we should look through elements of an array one by one, until searched value is found. In case of searched value is absent from array, we go through all elements. In average, Complexity of such an element is proportional to the length of array.

Situation changes significantly, when array is sorted. If we know it, random access capability can be utilized very efficiently to find searched value quick. Cost of searching algorithm reduce to binary logarithm of the array length. For reference, $\log_2(1000000) = 20$. It means, that in worst case, algorithm makes 20 steps to find a value in sorted array of a million elements or to say, that it doesn't present it the array.

6.3.1 Binary Search Algorithm

Binary search is a searching algorithm that works efficiently with a sorted list. The mechanism of binary search can be better understood by an analogy of a telephone directory. When we are searching for a particular name in directory ,we first open the directory from the middle and then decide whether to look for the name in the first part of the directory or in the second part of the directory. Again, we open some page in the middle and the whole process is repeated until we finally find the right name.

Algorithm work as follow. Algorithm done either recursively or iteratively.

1. Get the middle element of sorted array [DATA].
   `Mid = INT (BEG + END) / 2`
   If DATA (MID) = ITEM, then search is successful and we set LOC: = MID. Otherwise a new segment of DATA is obtained as follows :
2. If ITEM < DATA (MID), the ITEM can appear only in the left half of the segment.
   so, END becames MID-1, and begin searching again.
3. If ITEM > DATA (MID), then ITEM can appear only in right half of the segment.
   so, BEG : = MID +1 and begin searching again.

Examples: find 6 in {-1, 5, 6, 18, 19, 25, 46, 78, 102, 114}

Step 1. Middle element is 10/2 = 5, at 5 location we found 19.
`6 < 19 [Item < DATA (MID)]`
So, END become MID-1 and begin searching again.

Step 2. New array is {-1, 5, 6, 18}
Middle element is 4/2 = 2, at 2nd location we have 5.
`ITEM > DATA (MID)`
So, BEG = MID+1, and begin searching again

Step 3. New array is {6, 18}
6 is ultimately find.

Write a program to search an element in an array using Binary search

```
# include <stdio.h>
# include <conio.h>
int main()
{
  int arr[10], num, I,n,beg,end,mid,found=0;
  clrscr();
  printf(" \n Enter the number of elements in the array: ");
  scanf(" %d", &n);
  printf(" \n Enter the elements: ");
  for(i=0;i<n;i++)
  {
  scanf(" %d", &arr[i]);

  }
  printf("\n Enter the number that has to be searched :");
  scanf("%d", &num);
  beg=0, end=n-1;
  while(beg<=end)
  {
    mid=(beg+end)/2;
    if(arr[mid]==num)
    {
      printf("\n %d is present in the array at position=%d,num,mid ");
      found=1;
      break:

    }
  else if(arr[mid]>num)
    end=mid-1;
  else
    beg=mid+1;
    }
    if (beg> end && found==0)
      printf("\n %d does not exist  in the array", num);
    getch();
    return 0;
}
```

6.3.2 Complexity of Binary Search Algorithm

As we above already discuss that cost searching is reduce to binary logarithmic of array length. In Binary search algorithm, we find that each comparision reduces the array size in half. Hence we require a most f (n) Comparisons to locate ITEM where.

$2f(n) > n$ or equivalently $f(n)=\lfloor \log_2 n \rfloor+1$

Running time for worst case is approx. equal to $\log_2 n$.

Similarly, can also show that the running time for average case is approximately to the running time for the worst or.

6.3.3 Interpolation Search

Also know as an extra polation search is a searching technique that finds a specified value in a sorted array. Concept of interpolation search is similiar to how we search name in a telephone directory or for keys by which a book's entries are ordered. For example, when looking for a name "Bharat" in a telephone directory, we know that it will be near the extreme left, so appling a binary search technique by dividing the list in two holves each time is not a good idea. we must start the extreme left in the frist pass it's.

In each step of Interpolation search, the remaining search space to the value to be found is calculated. the calculation done on the basis of values at the bounds of the search space and the value to be searched. The value found at this estimater position is then compared with the value being searched for. If the two values are equal, them the search is complete.

Example: -

Given a list of numbers a [] = 1,3,5,7,9,11,13,15,17,19,21
Search for value 19 using interpolation search technique.
Solution:
Low = 0, High = 10, VAL = 19, a [low] = 1, a [High] = 21
Middle = low=(High- Low)× {[VAL-a(low)]/[a(High)-a(low)]}
= 0+(10-0)×(19-1)/(21-1)
= 0+10×0.9=9
a[middle]=a[9]=19 which is equal to value to be searched.

6.3.3.1 Complexity of Interpolation Search

when n elements of a list to be sorted are uniformaly distributed (average, case), interpolation search makes about log(logn) comparisons. How ever, in worst case, that is when the elements increase exponentially, the algorithm can make up to O(n) compaisions.

6.3.4 Difference b/w Linear Search and Binary Search

Linear	Binary search
1. Data in any order	1. Data must be in sorted order
2. Time Complexity O (n)	2. Time Complexity O (log n)
3. Access is slower	3. Access is faster
4. Single/multi dimensional array	4. Only single dimensional array

■ ■ ■

Very Short Questions

1. Define Searching.
2. Write down the different methods of searching.
3. What is Linear Search method?
4. Define Binary search method
5. What is the running time complexity of Linear search?
6. What is the running time complexity of Binary search?

Short Questions

1, What is the difference between Linear search and Binary search?
2. What type of data structure used in Binary search and why?

Long Questions

1. Define Linear search with an example .
2. Define Binary search with an example and write down its limitations.

Chapter ▸ 7

Hashing and Collision

7.1 Introduction

In chapter we discussed two search algorithms: linear search and binary search.Linear search has a running time proportional to O(n), while binary search take time proportional to O(logn),where n is the number of elements in the array. Binary search and binary search tree are efficient algorithm to search for an element .But what if we want to perform the search operation in time proportional to O(1)? In other words,is there a way to search an element in an array within constant time,irrespective its size?

7.2 Hashing

As in array and linked list the linear search requires O (n) comparisons. In a sorted array the binary search require. O (logn) comparision and in binary search tree searching required O (logn) comparision. However, there are applications which requires O (l) i.e. constant time searching capability. Ideally it may not be possible. but still we can achieve a performance very close to it using the data structure called HASH TABLE.

The search time of each algorithm discussed so far depends on the number of 'n' elements in the collection of data (array, linked list). But now we discuss a technique called hashing or hash addressing, which is essentially independent of the number n.

7.3 Hash table

It is a data structure in which location of data item is determined directly as a function of the data its of rather than by a sequence of comparision under ideal condition the time required to locate a data item is O (1) and does not depend upto the number of data stored. Hash table becomes very effective since it uses an array of size proportional to the key actually used. [key is which uniquiely determined the searching item in data collection].

7.4 Hash Function

Basic application of Hashing is done in file management. We assume that there is a file *f* of *n* records with a set *k* of keys which unquiely determine the records of *f*. Secondly, we assume that f is maintained in memory by a Hash table 'T' of 'm' memory locations and that L is the set of memory addresses of the location in T. The general idea of using key is to determine the address of a record is and excellent idea, but is must be modified so that a great deal of space is not wasted. This modification so that a great deal of space is not wasted. This modification takes the form of a function H frm the set k of keys into the set L of memory address. Such a function,

$H:k \rightarrow L$

Is called as Hash function or hashing function.

Some times, such function H may not yield different values, it is possible that two different keys K_1 and K_2 will yield the some hash address, this situation is called ''Collision".

Load Factor

Load factor of hash table is the ratio of element to buckets. The default load factor is 1 0(means table is full) generally provide best balance between speed and size.

7.4.1 Characteresicties of Hash Function

(1) It should be possible to compute efficiently
(2) It should distribute the keys uniformly across the hash table i.e. it should keep the number of collision as minimum as possible.
(3) The cost of executing a hash function must be small,so that using the hashing technique become preferable over other approaches.

7.5 Different types of Hash function

In this section, we will discuss the hash function which use numeric keys.However,there can be cases in real-world applications where we can alphanumeric keys

7.5.1 Division Method

choose a number m larger than the number n of keys in K (m in usually chosen to be a prime number or a number without small division, since this frequently minimizes the number of collisions)

n (k) = K (mod m) k= keys

{ k (mod m) denotes by m the remainder when k divides m= size of hash table.}

Example : If m= 9 then key k= 132 is mapped as

h (132) = 132 mod 9 = 7

7.5.2 Mid square Method

The mid-square method squares the key value, and then takes out middle 'r' bits of the result, giving a value in the range 0 to 2^r-1. This works will become most or all bits of the key value contribute to the result.

Example:

Consider a record whose keys are 4-digit numbers in base 10. The goal is to hash these key values to a table of size 100 (i.e.a range of 0 to 99). This range is equivalent to two digits in base 10. That is r=2. If the input is the number 4567, squaring yields an 8 digit number, 20857489 the middle two digits of this result is "57".

7.5.3 Folding Method

The key value K is divided into number of parts $K_1, K_2, K_3, \ldots\ldots\ldots\ldots k_r$ where each part has same no. of digits except last part. Now these parts are added and the hash value is obtained by ignoring the last carry if any.

Example:

If m=100 K= 9235, 714, 71458

K:	9235	714	71458
Parts:	92,35	71,4	71,45,8
Sum:	127	75	114
	⊔	⊔	⊔
h (k):	27	75	14 $\{h(k) = k_1 + k_2 + k_3 + \ldots\ldots K_r\}$

7.6 Collisions

As discussed earlier in this chapter ,collisions occur when the hash function maps two different keys to the same location.Obviously,two records cannot be stored in the same location. Therefore, a method used to solve the problem of collision,also called collision resolution technique, is applied.

7.6.1 Collision Resolution

Suppose we want to add a new read in a file F, but the memory location address H (k) is already occupied. This situation is called collision. Collision cannot be climmated altogether but we can keep collision to a certain minimum level.

We can use two strategies:

(a) Separate Chaining

(b) Open Addressing
- Linear probing
- Double hashing
- Quadratic Probing

a) Separate Chaning

In this scheme all those, elements whose key values map to same hash table, slots are put in one linked list. Thus, slot [i] in the hash table contains the painter to the head of the linked list of all elements that has to value [i]. If there is no such element that hash to value [i] the slot [i] contain Null value.

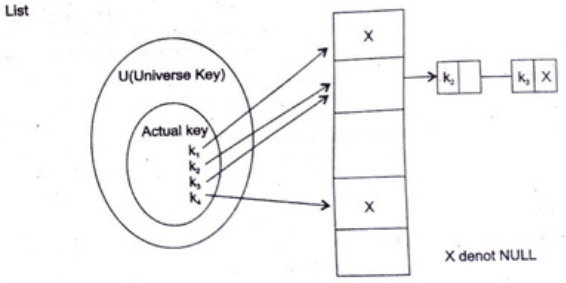


Figure 7.1 Separate Chaning

Example:

Consider a hash table having I slots and it uses hash function h (k) = K mod 9 and the key values are 2, 8, 19, 15, 20, 33, 12, 17, 10, 5

h (5) = 5 mod 9 = 5
h (28) = mod 9 = 1
h (19) = 19 mod 9 = 1
h (20) = 20 mod 9=2
h (33) = 33 mod 9 = 3
h (12) = 12 mod 9 = 3
h (17) = 17 mod 9 = 8
h (10) = 10 mod 9=1

They are mapped as follow in hash table-

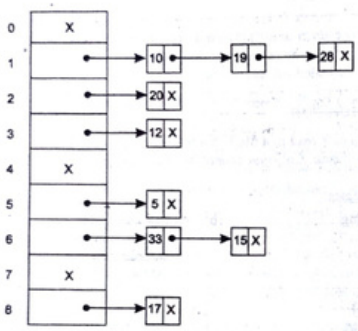


Figure 7.2

b) Open Addressing

In open addressing, each position in the array is one of three states, EMPTY, DELETED or OCCUPIED if a position is OCCUPIED, it contains a legitimate value (key and data), otherwise, it contain no value. Position are initially EMPTY. If the value in a position is deleted, the position is marked as Deleted, not Emply : the need for this distinction is explained below:

The algorithm for inseuting a value Vinto a table is this :-

1. Compute the position at which V ought to be stored, P=H (Key (v)).
2. If position P is not OCCUPIED, Store V there.
3. If position P is Occupied, Compute another position in the table, set P to this value and repeat (2)-(3).

How should we ''compute another position'' is step (3)

- **Linear Probing**

Process of examining the slots in the hash table is called probing.

Linear probing P= (1+ P) mod Table- Size.

This has the virtues that it is very past to compute and at ''probes'' (i.e. look, at) every position in the hash table.

Example: Let table size 10, keys our 2- digit integers, and H (k) = k mod 10. Initially all the positions in the table are Emply.

0	1	2	3	4	5	6	7	8
Empty	Empty	Empty	Empty	Empty	Empty	Empty	Empty	Empty

I we now insert 15, 17 and 8 there is no collision, the result is -

0	1	2	3	4	5	6	7	8	9
Empty	Empty	Empty	Empty	Empty	15	Empty	17	8	Empty

When we insert 35, we compute P- H (35)= 35 mod 10=5 position 5 is occupied (by 15). So we must loop elsewhere for a position in which to store 35. Using Linear probing, the position we would by next (5+1) mod 10=6. Position 6 is empty, so 35 is inserted there-

0	1	2	3	4	5	6	7	8	9
Empty	Empty	Empty	Empty	Empty	15	35	17	8	Empty

Now suppose we insert 25, we first by position (25 mod 10). It is occupied. Using Linear probing, we next by position 6 (5+1 mod 10) it is too occupied, as in the next one we by, position 8 (7+1 mod 10) Poition 9 is next (8+1 mod 10); it is empty so 25 is inserted there.

0	1	2	3	4	5	6	7	8	9
Empty	Empty	Empty	Empty	Empty	15	35	17	8	25

Now suppose we insert 75. The first position we try is 5. Then we will try position 6, then 7, then 8, then 9, finally we try position O (9+1 mod 10); it is empty so 75 is stored there

0	1	2	3	4	5	6	7	8	9
15	Empty	Empty	Empty	Empty	15	35	17	8	25

Now, suppose we insert 11, it is easily entered at 1 position, in this way linear probing works.

Note: Linear probing is easy to implement out but suffer from a problem called primary clustering. Clusters are blocks of occupied slots. So avoid primary clustering we use quadratic probing.

7.7 Quadratic Probing

Here, if a value already stored at location generated by h(k), then following hash function is used to resolve collision.

$H(k,i) = [h'(k)+c_1 i+c_2 i^2] \bmod m$

Where m is size of the hash table, h'(k)=(k mod m) ,I is probe number that varies from 0 to m-1 and c_1 and c_2 are constants such that c_1 and $c_2 \neq 0$.

Quadratic probing eliminates the primary clustering phenomenon of linear probimg because instead of doing linear search ,it does a quadratic search. For a given key k, first location generated by h'(k) mod m is probed .If location is free,the value is stored in it, else subsequent locations probed are offsets that depend in a quadratic manner on the probe number i.

Example: Consider a hash table of size 10 .Using quadratic probing, insert key 72,27,36,24,63,81, and 101 into the table. Take c_1 =1and c_2=3

Solution:

Let h'(k)= k mod m, m=10

Initially , the hash table can be given as:

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

We have,

$H(k,i)=[h'(k)+c_1i+c_2i^2] \bmod m$

Step 1 Key=72

H(72,0)= [72 mod 10 + 1×0+3×0] mod 10
=[72 mod 10]mod 10
=2 mod 10
=2

Since T[2] is vacant , insert the key 72 in T[2]. The hash table now become:

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	-1	-1	-1

Step 2 Key =27

H(27,0)= [27 mod 10 + 1×0+3×0] mod 10
=[27 mod 10]mod 10
=7mod 10
=7

Since T[7]is vacant , insert the key 27 in T[7]. The hash table now become:

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	27	-1	-1

Step 3 Key=36

H(36,0)= [36 mod 10 + 1×0+3×0] mod 10
=[36 mod 10]mod 10
=6mod 10
=6

Since T[6] is vacant , insert the key 36 in T[6]. The hash table now become:

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	36	27	-1	-1

Step 4 Key=24

H(24,0)= [24 mod 10 + 1×0+3×0] mod 10
=[24 mod 10]mod 10
=4 mod 10
=4

Since T[4] is vacant , insert the key 24 in T[4]. The hash table now become:

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	24	-	36	27	-1	-1

Step 5 Key=63

H(63,0)= [63 mod 10 + 1×0+3×0] mod 10
=[63 mod 10]mod 10
=3 mod 10
=3

Since T[3] is vacant , insert the key 63 in T[3]. The hash table now become:

0	1	2	3	4	5	6	7	8	9
-1	-1	72	63	24	-1	36	27	-1	-1

Step 6 Key=81

H(81,0)= [81 mod 10 + 1×0+3×0] mod 10
=[81 mod 10]mod 10
=1 mod 10
=1

Since T[1] is vacant , insert the key 81 in T[1]. The hash table now become:

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	-1	36	27	-1	-1

Step 7 Key=101

H(101,0)= [101 mod 10 + 1×0+3×0] mod 10
=[101 mod 10]mod 10
=101 mod 10
=1

Since T[1] is already occupied , the key 101 cannot stored in T[1]. Therefore, try again for the next location. Thus probe i=1.

Key =101

H(101,1)= [101 mod 10 + 1×1+3×1] mod 10
=[101 mod 10+4]mod 10
=[1+4]mod 10
=5 mod 10
=5

Since T[5] is vacant , insert the key 101 in T[5]. The hash table now become:

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	101	36	27	-1	-1

Quadratic probing is widely applied in Berkeley Fast File System to allocate free blocks.

7.8 Double Hashing

To start with, double hashing uses one hash value and then repeatedly steps forward an interval until an empty location is reached. The interval is decided using a second, independent hash function, hence the name *double hashing*.In double hashing ,we use two hash function rather than a single function. The hash function in the case of double hashing can be given as:

$h(k,i)=[h_1(k)+ih_2(k)] \bmod m$

Where m is the size of hash table ,$h_1(k)$and $h_2(k)$ are two hash function given as $h_1(k)=k \bmod m$, $h_2(k)=k \bmod m$ ‘,I is probe number that varies from 0 to m-1 and m’ is chosen to be less than m . We can choose m’=m-1 or m-2.

When we have to inset a key k in the hash table,we first probe the location given by applying $[h_1(k)=k \bmod m]$ because during the first probe ,i=0.If the location is vacant ,the key is inserted into it,else subsequent probes generate locations that are at an offset of $[h_2(k)=k \bmod m]$ from the previous location. .since the offset may vary with every probe depending on the value generated by the second hash function ,the performance of double hashing is very close to the performance of the odeal scheme of uniform hashing.

Double hashing minimizes repeated collisions and the effects of clustering .That is,double hashing is free from problems associated with primary clustering as well as secondary clustering.

Double Hashing minimizes repeated collisions and the effects of clustering .That is, double hashing is free from problems associated with primary clustering as well as secondary clustering.

Example: consider a hash table of size =10. Using double hashing ,insert the keys 72,27,36,24,63,81,92,and 101 into the table. Let h_1=(k mod 10) and h_2=(k mod 8).

Solution

Let m=10

Initially,the hash table can be given as

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

We have ,

H(k,i)=[h_1(k)+ih_2(k)]mod m

Step 1 Key=72

H(72,0) = [72 mod 10 +(0 ×72 mod 8)] mod 10
= [2 +(0×0)] mod 10
= 2 mod 10
= 2

Since T[2] is vacant , insert the key 72 in T[2].The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	-1	-1	-1

Step 2 key =27

H(27,0) = [27 mod 10 +(0 ×27 mod 8)] mod 10
= [7 + (0×3)] mod 10
= 7 mod 10
= 7

Since T[7] is vacant, insert the key 27 in T[7]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	27	-1	-1

Step 3 Key=36

H(36,0) = [36 mod 10 +(0 ×36 mod 8)] mod 10
= [6 + (0×4)]mod 10
= 6 mod 10
= 6

Since T[6] is vacant, insert the key 36 in T[6]. The hash table now becomes

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	36	27	-1	-1

Step 4 key=24

H(24,0) = [24 mod 10 +(0 ×24 mod 8)] mod 10
= [4 +(0 × 0)] mod 10
= 4 mod 10
= 4

Since T[4] is vacant, insert the key 24 in T[4]. The hash table now becomes

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	24	-1	36	27	-1	-1

Step 5 key=63

H(63,0) = [63 mod 10 +(0 × 63 mod 8)] mod 10
= [3 + (0 × 7)]mod 10
= 3 mod 10
= 3

Since T[3] is vacant, insert the key 63 in T[3]. The hash table now becomes

0	1	2	3	4	5	6	7	8	9
-1	-1	72	63	24	-1	36	27	-1	-1

Step 6 Key = 81

H(81,0) = [81 mod 10 +(0 × 81 mod 8)] mod 10
= [1 + (0×1)] mod 10
= 1 mod 10
= 1

Since T[1] is vacant, insert the key 81 in T[1]. The hash table now becomes

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	-1	36	27	-1	-1

Step 7 Key=92

H(92,0) = [92 mod 10 +(0 × 92 mod 8)] mod 10
= [2 + (0× 4)] mod 10
= 2 mod 10
= 2

Now T[2] is occupied ,so we cannot store key=92 in T[2]. Therefore, try again for the next location.Thus probe, i=1, this time.

Key=92

H(92,1) = [92 mod 10 +(1 × 92 mod 8)] mod 10
=[2 + (1 × 4)]mod 10
= (2+4) mod 10
= 6 mod 10
= 6

Now T[6] is occupied ,so we cannot store key=92 in T[6]. Therefore, try again for the next location.Thus probe, i=2, this time.

Key=92

H(92,2) = [92 mod 10 +(2 × 92 mod 8)] mod 10
= [2 + (2× 4)] mod 10
= [2 + 8] mod 10
= 10 mod 10
= 10

Since T[0] is vacant, insert the key 92 in T[0]. The hash table now becomes

0	1	2	3	4	5	6	7	8	9
92	81	72	63	24	-1	36	27	-1	-1

Step 8 Key=101

H(101,0) = [101 mod 10 +(0 × 101 mod 8)] mod 10
= [1 + (0×5)]mod 10
= 1 mod 10
= 1

Now T[1] is occupied ,so we cannot store key=101 in T[1]. Therefore, try again for the next location. Thus probe, i=1, this time.

Key=101

H(101,1) = [101 mod 10 +(1 × 101 mod 8)] mod 10
= [1 + (1× 5)]mod 10
= [1 + 5]mod 10
= 6

Now T[6] is occupied , so we cannot store key=101in T[6]. Therefore,we try for the next location with probe i=2.Repeat the entire process until a vacant location found.You will see that we have to probe many times to insert the key 101 in the hash table. Although double hashing is a very efficient algorithm, it always require m to be a prime number. In our case m=10, which is not a prime number , hence, the degradation in performance. Had m been equal to 11, the algorithm would have worked efficiently . Thus, we can say that the performance of the technique is sensitive to the value of m.

7.9 Rehashing

When the Hashtable is half full or nearly full, the number of collision is increases,results in poor performance of insertion and search operations. In such cases, it is better to create a new hash table with size double of original hash table.

All entries in the original hash table will then have to be moved to the new hash table .This is done by taking each entry ,computing its new hash value,and then inserting it in the new hash table

7.10 Application of Hashing

- Hashing table is uded for database indexing.
- Hashing technique is used to implement compiler symbol tables in C++.
- Hashing widely used for Internet search engines.
- Hashing technique used in real world applications like , CD-Database, Drivers License/Insurance cards, sparse matrix,filesignatures,Game boards ,Graphics so on.

■ ■ ■

Very Short Questions

1. Define Hash table.
2. What do you understand by Hash function?
3. State down the properties of a good hash function
4. How hash table is better than a direct access table(array)?
5. What is collision?
6. Give some applications of Hashing
7. Is a good hash function completely eliminates collision.
8. What is Load factor?

Short Questions

1. Write short notes on
   (a) Linear probing (b) Quadratic probing
2. Explain chaining with an example
3. State down different types of Hash function with an example of each.
4. What is collision . How can we prevent collision?
5. Write a short note on Rehashing?

Long Questions

1. Consider a hash table with size=10. Using linear probing , insert the keys 27,72,63,42,36,18,29 and 101 in table.
2. Consider a hash table with size=10. Using Quadratic probing , insert the keys 27,72,63,42,36,18,29 and 101 in table.Take c_1=1 and c_2=3.
3. Consider a hash table with size=11. Using double hashing , insert the keys 27,72,63,42,36,18,29 and 101 in table.Take h_1=k mod 10 and h_2=k mod 8
4. Calçulate hash values of keys: 1892,1921,2007,3456 using different methods of hashing.
5. What is collision resolution? State down the methods of collision resolution with example.

Chapter ▸ 8

Sorting

8.1 Interoduction

Sorting is nothing but storage of data in sorted order either in Ascending or Descending order. In other word sorting refers to the operation of arranging data in some given order such as increasing data or decreasing with numerical data, or alphabetically, with character data. The term sorting comes into picture with the term searching there are so many things in our life that we need to search like a particular record in database, roll number in merit list, a particular telephone number, any particular page in book etc.

Sorting arranges data in a sequence which make searching easier. Sorting and searching apply to a file of records. Every record which is going to sorted will contain one key called **Primary key (k)** which is uniquely determine the records in the file for e.g. suppose we have a record of students every such record will have the following data :

- Roll No.
- Name
- Age
- Class.

Here student roll no. can be taken as key for sorting the records in ascending or descending order. Now suppose we want to search a student with roll no. 15, we don't need to search the complete record we will simply search between the students with roll no. 10 to 20.

8.2 Sorting

Let A be a list of n elements $A, A_2, \ldots\ldots$ An in memory. Sorting refues to operation of rearranging the contents of A so that they are decreasing in order that is, so that

$A > A_2 > A_3 > \ldots\ldots A_n$

Since A has n elements, there are n ! ways that the contents can appear in A.

Example:

Suppose on array DATA contain 8 elements as follows :-
DATA : 99, 66, 44, 33, 11, 22
After sorting, DATA must appear in memory as follows :-
DATA : 11, 22, 33, 44, 66, 99

Since DATA consists 6 elements, there are 6!=720 ways that the numbers 11, 22,.....99 can appear in DATA.

8.2.1 Types of Sorting

(i) **Internal sorting:** If the size of data that is required to be sorted can be accommodated in the main memory then the sorting procedure applied on that data list is called internal sorting.
(ii) **External sorting:** It is a counter part of Internal sorting. This is applied on those data list which size cannot be accommodated in the main memory.
(iii) **In place Sorting:** Sorting procedure is in place if it does not was any extra space during sorting.
(iv) **Stable Sorting:** If a sorting procedure maintain the relative order if indices of equal keys then they this shoarting procedure is stable. For example:

"2 1 4 10 3 7 5 2" then after stable sorting list will be like this 1 2' 2" 3 4 5 7 10 .

8.3 Algorithms for Sorting

8.3.1 Bubble Sort

Bubble sort is simple method that sorts the array elements by repeaatedly moving the largest element to the highest index position of the array segment .In bubble sorting consecutive adjacent pair of elements in the array are compared with each other. If the element at the lower index is greater than the element at the higher index , the two elements are interchanged so that the element is placed before the bigger one.This process will continue till the list of unsorted elements exhausts.

This algorithm named bubble sorting because elements 'bubble'to the top of the list.Note that at the end of the first pass the largest element in the list will be placed at its proper position.

Consider a File:

25,57,48,37,12,92,86,33
In first pass following comparisions are made
X[0] with x[1] (25,with 57) no interchange
X[1] with x[2] (57, with 48) interchange
X[2] with x[3] (57,with 37) interchange
X[3] with x[4] (57,with 12) interchange
X[4] with x[5] (57,with 92) no interchange
X[5] with x[6] (92,with 86) interchange
X[6] with x[7] (92,with 33) interchange

And after the first pass the file is

25,57 48,37,12,57,86,33,**92**(largest)

Pass 0: 25,57,48,37,12,92,86,33
Pass 1: 25,48,57,37,12,86,33,**92**
Pass 2: 25,37,12,48,57,33,**86,92**
Pass 3: 25,12,37,48,33,**57,86,98**
Pass 4: 12,25,37,33,**48,57,86,98**
Pass 5: 12,**25,33,37,48,57,86,98**

Pass 6: 12,25,33,37,48,57,86,98
Pass 7: 12,25,33,37,48,57,86,98

C program for algorithm

```
Void Bubble(int x[],int n)
{
  int hold,J,pass;
  int flag=TRUE;
  for (pass=0;((pass<n-1)&&(flag==True));pass++)
  {
     flag=False;
     for(J=0;J<n-pass-1;J++)
       if(x[J]>x[J+1])
         swap(x[J],x[J+1])
  }

}
```

8.3.1 Complexity of Bubble Sort

Complexity of any sorting algorithm depends upon the number of comparisons.In Bubble sort,we have seen that there are n-1 passes in total.In first pass, n-1 comparisons are made to place the highest element in its correct position.Then,in Pass 2, there are n-2 comparisions and the second highest element is placed in its position.Therefore, to compute the complexity of bubble sort we need to calculate the total number of comparisons. It can be given as:

$F(n)=(n-1)+(n-2)+(n-3)+(n-4)+....+3+2+1$
$F(n)=n(n-1)/2$
$F(n)=n^2/2+O(n)=O(n^2)$

therefore,the complexity of bubble sort algorithm is $O(n^2)$ .It means the time required to execute bubble sort is proportional to n^2,where n is the total number of elements in the array.

8.4 Insertion Sort

If the few objects are already sorted, an unsorted object can be inserted in the sorted set in proper place. In the algorithm, pick a particular value and insert it at appropriate place in sorted sublist i.e. during the k^{th} intervention a [k] is inserted in the proper place in the sorted sub-array a [1], a [2], a [3] ..... a[k-1]. This task is accomplished by comparing a[k] with a [k-1], a [k-2] ..... and so on until the first element a [j] such that $a[j] \leq a[k]$ is ..... then reach of element a [k-1], a [k-2] ..... a [j +1] are moved one position up and then element a [k] is inserted in [j + 1], a[k-2] ..... a [j+1] are moved one position up and then element a [k] is inserted in $j+1^{th}$ position in the array it also requiring (n-1) passes to sort a file.

Write a program to sort an array using insertion sort algorithm

```
# include <stdio.h>
# include <conio.h>
void insertion_sort(int arr[], int n)
void main()
{
 int arr[10], i,n;
 clrscr();
 printf(" \n Enter the number of elements in the array: ");
 scanf(" %d", &n);
 printf(" \n Enter the elements of the array ");
   for(i=0;i<n;i++)
   {
     scanf(" %d", &arr[i]);

   }
   insertion_sort(arr,n);
   printf("\n  The sorted array is:\n");
   for(i=0;i<n;i++)
      printf("%d\t", arr[i]);
    getch();
}
void insertion_sort(int arr[],int n)
{
  int i,j,temp;
  for(i=0;i<n;i++)
  {
    temp=arr[i];
    j=i-1;
    while((temp<arr[j]) && (j>=0))
    {
      arr[j+1]=arr[j];
      j--;
    }
    arr[j+1]=temp;

  }

}
```


Consider a file

35	20	40	100	3	10	15
0	1	2	3	4	5	6

Pass-1 Since a [1] < a [0] insert a [1] before a [0]

20	35	40	100	3	10	15
0	1	2	3	4	5	6

Pass-2 Since a [2] > a [1] no action is taken.

20	35	40	100	3	10	15
0	1	2	3	4	5	6

Pass-3 Since a [3]> a [2] no action is taken

20	35	40	100	3	10	15
0	1	2	3	4	5	6

Pass-4 Since a [4] is less that a [3], a [1] and a [0], therefore shifting all a [3], a [2], a [1],

a [0] one position right and placing a [4] to a [0].

3	25	40	100	35	10	.15
0	1	2	3	4	5	6

Pass-5 a [5] is less than a [4], a [3], a [2], a [1], so shifting the one position right and a [5] is placed at a [i]:

3	10	25	35	40	100	15
0	1	2	3	4	5	6

Pass-6 a [6] is less than a [5], a [4], a [2] so shifting then one position right and placing a [6] is placed at a [2]:

3	10	15	25	35	40	100
0	1	2	3	4	5	6

 # C program for algorithm:

```
Void Insertion sort (in a [  ], int n)
{
  int k, temp, J;
  for (k=1; k < n, k+1)
  {
    temp = a [k];
    J= k-1
  while ((J>o) and (temp < a [J]))
    { a[J+1] = a [J];
    }
  {
    a [J+1]= a [J];
    J---;
  }
  a [ J+1] = temp;
  }
}
```

8.4.1 Complexity of Insertion Sort

Time: The number f (n) of Comparisions in the insertion sort algorithm can be easily computed. In worst case, elements of list are in reverse order, use k-1 number of comparision Hence.

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

Average case :- There will be apporximately
(k-1)/2 Comparisions in the inner loop.

$$f(n) = \frac{1}{2} + \frac{2}{2} + \cdots \frac{n-1}{2} = \frac{n(n-1)}{4} = 0(n^2)$$

Best case: The best case time complexity is when the array is already sorted, and is O (n).

Space: The space is O (1), just a few scalar variables

8.4.2 Advantages of Insertion Sort

- The advantages of this sorting algorithm are as follows:
- It is easy to implement and efficient to use on small sets of data.
- It can be efficiently implemented on data sets that are already substantially sorted.
- It performs better than algorithm like selection sort and bubble sort,Insertion sort algorithm is simpler than others,with only a small trade-off in efficiency.It is over twice as fast as the bubble sort and almost 40 percent faster than selection sort.
- It require less memory space (only O(1)) of additional memory space.

8.5 Selection Sorting

Suppose an array A with n elements A [1], A [2]...A (N] is in memory. In selection sort, first pass starts by finding the smallest element in the entire array and swap with the first element. In the second pass find the smallest element from the remaining array and swap with the 2nd element and so on.

Consider a file-

Data	20	35	40	100	2	11	15
Index	0	1	2	3	4	5	6

Pass 1. Intercharge the elements a [0] and a [4]

2	35	40	100	20	11	15
0	1	2	3	4	5	6

Pass 2. Swap a [1] with a [5]

2	11	40	100	20	35	15
0	1	2	3	4	5	6

Pass 3. Swap a [2] with a [6]

2	11	15	100	20	35	40
0	1	2	3	4	5	6

Pass 4. Swap a [3] with a [4]

2	11	15	20	100	35	40
0	1	2	3	4	5	6

Pass 5. Swap a [4] with a [5]

2	11	15	20	35	100	40
0	1	2	3	4	5	6

Pass 6. Swap a [5] with a [6]

2	11	15	20	35	40	100
0	1	2	3	4	5	6

Selection sort requires (n-1) passes to sort an array-

C programe is selection Sort-

```
Void Selection Sort (int a [   ], int Size)
{
int i., j, min, temp;
for (i=0; i < size-1; I + 1)
}
min = i ;
for (j= i + 1); j < size; j + 1)
{
if… (a [j] < a [min]  …. if element at j is less than element at min position
  {
  min= j; …… then set min as]
  }
}
temp = a [i];
a [i] = a [min];
a [min] = a [temp];
  }
}
```

8.5.1 Complexity of Selection Sort

In selection sort, selecting the lowest element require scanning all n elements (this take n-1 comparisions) and then swapping it into the first position. Finding the next lowest element require scanning the remaining n-1 elements and so on.

(n-1) + (n-2) + (n-3)...+ 2 + 1 = n (n-1)/2
Θ (n^2) comparisions

Space Complexity: Selection sort algorithm uses a stack. In worst case, the size of the stack can be large as the size of the data array to be sorted. So the space complexity of the proposed algorithm is O(n).

8.5.2 Advantage of Selection Sort

- It is simple and easy to implement.
- It can be used for small data sets.
- It is 60 per cent more efficient than bubble sort.
- However, in case of large data sets, the efficiency of selection sort drops as compared to insertion sort.

8.6 Radix Sort

Radix sort is one of the linear sorting algorithms for integers. It functions by sorting the input numbers on each digit, for each of the digits in the numbers. However the process adopted by this sourt method is some what counter intentive, in the sense the numbers are sorted on the least significant digit first, followed by the second-least significant digit and so on till the most significant digit.

For example Consider the set of numbers-

33101	26440	16341	20101	801

Since Unit place digits are in order 1 0 1 1 1
after using redix sort to sort these numbers

26440	33101	16341	20101	801

Notice that the integers that ends with 1 are in the same order as in the original array because radix sort **is stable.** Now sort acc. to 10 digit after using sorting new array like.

33101	20101	801	26440	16341

Now sort array acc. to 100th digit, now array appears as .

33101	20101	16341	26440	801

Against sorting based on 1000th digit

20101	801	33101	16341	26440

Again sourting by most significant digit (mean 10,000th digit)

801	16341	20101	26440	33101

The number of Comparisions needed to sort 5 such 5 digit numbers is bounded as follow :-

C < 5 ×5 ×10

The 5 comes from the five digit number (801 can be written as. 00801) and the 10 comes from radix d=10 **digit** (decimal systems).

Write a program to implement radix sort algorithm

```
# include <stdio.h>
# include<conio.h>
int largest(int arr[], int n)
void radix_sort(int ar[], int n )
void main()
{
  int arr[10], i,n;
  clrscr();
  printf(" \n Enter the number of elements in the array: ");
  scanf(" %d", &n);
  printf(" \n Enter the elements of the array  ");
  for(i=0;i<n; i++)
  {
     scanf(" %d", &arr[i]);

  }
  radix_sort(arr, n);
     printf("\n  The sorted array is:\n");
  for(i=0;i<n;i++)
     printf("%d\t", arr[i]);
    getch();
}
int largest (int arr[], int n)
{
  int large=arr[0],I;
  for(i=1;i<n;i++)
  {
```

```
        if(arr[i]>large)
          large=arr[i];
        }
      return large;
    }
    void radix_sort(int arr[], int n)
    {
      int bucket[10][10], bucket_count[10];
      int i,j,k, remainder, NOP=0, divisor=1,large,pass;
      large=largest(arr,n);
      while (large>0)
      {
        NOP++;
        large/=10;
        }
      for (pass=0;pass<NOP;pass++)        //initialize the buckets
      {
        for(i=0;i<10;i++)
          bucket_count[i]=0;
        for(i=0;i<n;i++)
        {
        // sort the numbers according to the digit at passth place remainder=(||arr[i]/
divisior)%10;
          bucket[remainder][bucket_count[remainder]]=arr[i];
          bucket_count[remainder]+=1;
          }
          //collect the numbers after PASS pass
          i=0;
          for(k=0;k<10;k++)
          {
          for(j=0;j<bucket_count[k];j++)
          {
            arr[j]=bucket[k][j];
            i++;
          }
          }
        divisor *=10;
      }
    }
```

8.6.1 Complexity of Radix Sort

Let us suppose a list A of n items A_1, A_2,...An is given. het d denote the radix (e.g., d=10 for decrimal digits, d=26 for alphabets and d=2 for bits), and suppose each item Ai is represented by means of S of the digits :

$A0 = d_{i1}\ d_{i2}\ ...d_{is}$

Radix algorithm will require & passes, the number of digits in each item. Pass k will compare each dik with each of the d digits hence.

$((n) \leq d * S * n)$

8.6.2 Worst Case

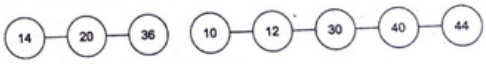
To calculate the complexity of radix sort algorithm,assume that there are n number that have to be soerted and k is the number of digits in the largest number.In this case,the radix sort algorithm is called a total of k times .The inner loop is executed n times. Hence,the entire radix sort algorithm takes O(kn)time to execute .When radix sort is applied on a data set of finite size (very small set of numbers), then the algorithm runs in O(n) asymptotic time.

Radix sort is simple and one of the fastest sorting algorithm but there are certain trades-off for this that can make it less preferable as compared to other sorting algorithm. It takes more space than other. Other drawback of it is that algorithm is dependent on digits or letters.That's why for every different data type,algorithm has to be rewritten.

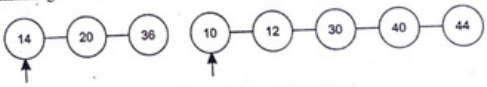
8.7 Merge Sort

Merge Sort.... based on divide and conquer technique (meanse rearsively break down a problem into two or more sub-problems of the same type (divide). Until these become Simple enough to be solved directly (conquer) Merge sort divides the array to be sortedinto two nearly equally parts each part is then sourted recursively, the two sorted halves are then Merge into one sourted array.

Example : Suppose that the goal is to merge the sorted array.

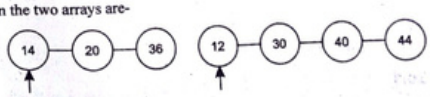


We begin by examining the first element in each array.

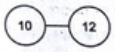


Since, 14 > 10, 10 is the smallest element it is copied the output array.

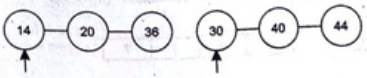
Again the two arrays are-



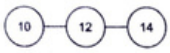
as 14 > 12, 12 is copied to out array.



Again two arrays are-



14 < 30, so 14 is copied to output array-



Again the two arrays are-

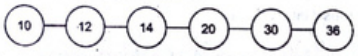
20<30, ...so 20 is copied to out put array-

and the two arrays are-

30<36, so 30 is copied to output array-

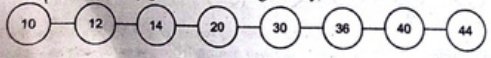
and the two arrays are-

36< 40, so 36 is copied to output array-



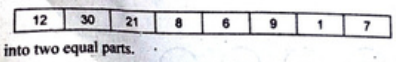
And the first array become empty and second array is

appending the second array gives the one merged array-

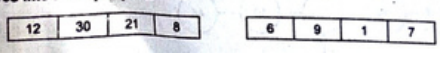


Example

Consider a file

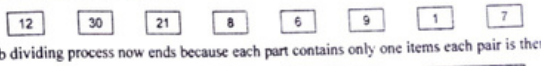


first the array is dived into two equal parts.



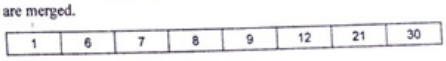
by merying sort... the process begins by dividing each part into equal parts.

then each of these part into equal part.

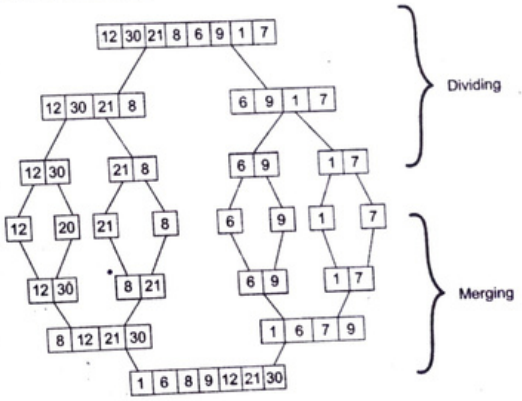


this sub dividing process now ends because each part contains only one items each pair is then merge.

Each of these pair are merged.



all these process can be drown in as fig-



After Pass k, the array A will be partitioned into sorted subarrays where each subarray, except the last, will contain exactly $L=2^k$ elements, thence the algorithm requires at most log n passes to sort an n-element of array A.

Write a program to implement merge sort algorithm

```
# include <stdio.h>
# include <conio.h>
void merge(int ar[], int,int ,int )
void merge_sort(int a[], int,int )
void main()
{
```

```
    int arr[10], i,n;
    clrscr();
    printf(" \n Enter the number of elements in the array: ");
    scanf(" %d", &n);
    printf(" \n Enter the elements of the array  ");
    for(i=0;i<n;i++)
    {
    scanf(" %d", &arr[i]);

    }
    merge_sort(arr,0, n-1);
     printf("\n  The sorted array is:\n");
    for(i=0;i<n;i++)
     printf("%d\t", arr[i]);
     getch();
   }
void merge(int arr[],int beg, int mid, int end)
{
  int i=beg, j=mid+1; index=beg. Temp[10],k;
  dcwhile ((i<=mid) && (j<=end))
  {
    if(arr[i]<arr[j])
    {
      temp[index]=arr[i];
      i++;


    }
    else
    {
      temp[index]=arr[j];
      j++;
    }
    index++;
  }
  if(i>mid)
{
  while(j<=end)
  {
    temp[index]=arr[j];
    j++;
    index;
  }
}
else
{
  while(i<=mid)
  {
    temp[index]=arr[i];
    i++;
    index++;
  }
```



```
  }
  for(k=beg; k<index;k++)
  arr[k]=temp[k];
}
void merge_sort(int arr[], int beg, int end)
{
  int mid;
  if(beg<end)
  {
    mid=(beg+end)/2;
    merge_sort(arr, beg, end);
    merge_sort(arr, mid+1, end);
    merge(arr, beg, mid, end);
  }
}
```

8.7.1 Complexity of Merge sort Agorithm

Let f (n) denote the number of comparisions needed to sort an n-element array A using merge-sort algorithm this algorithm requires at most log n passes. Moreover, each pass merges a total n elements, each pass will require at most n comparisions. Accordingly for both the worst case and average case.

$f(n) \leq n \log n$

Here, we see that this algorithm has the same order as heap sort and the same average order as quick sort. Main draw back of Merge sort is that is requires an auxiliary array with n element. Extra memory require of order O (n). Because merging two sorted subarray in place is more complicated and would need more comparisions and more operations.

Two sorted list of size 'm' and 'n' ,require m+n-1 number of comparision in the worst case of merge sort.

Note: Merge sort is more efficient than quick sort.

8.8 Quick Sorting

Quick sort is a sorting algorithm which uses the idea of divide and Conquer. Pick one element, called Pivot, from the array which has to sort, with help of this pivot element, array divide into two halves in such a way that elements in the left sub-arrary are less than and the elements in the rights subarray which are greater than the pivot element Suppose A is the following array:

25, 10, 30, 15, 20, 28

There the dividing element we choose is the first element of the array i.e. a [0] we call it pivot to Begin with set Pivot=0, left= 0, right = n-1 (here n=6, so..... n-1=5)

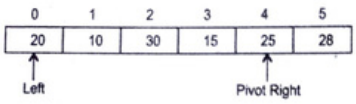
25	10	30	15	20	28
0	1	2	3	4	5

Starting scanning elements from right since a [Pivot]< a (right) we decrease the value of valuable right to get.

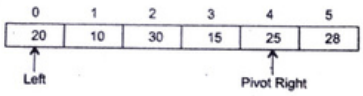
0	1	2	3	4	5
25	10	30	15	20	28

Pivot Left — Right

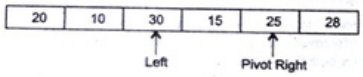
Now a [Pivot]> a (right), inter change these elements to3 get



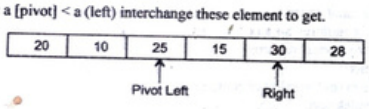
Now Start scanning elements from left since a [Pivot) > a [left] we increase the value of left of get.



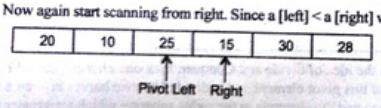
since again a [Pivot] > a (left) increase the value of left.



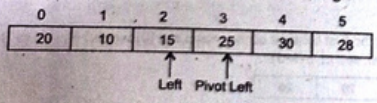
Now a [pivot] < a (left) interchange these element to get.



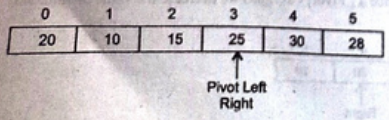
Now again start scanning from right. Since a [left] < a [right] we decrease the value of valuable right to get.



Now a (Pivot) > a [Right] inter change these element, to get



ow start scanning from left since a [pivot]> a (left) increase left

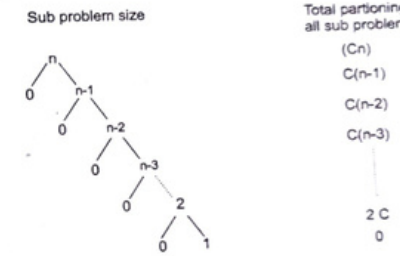


Since left becomes equal to right we stop pivot element 25 is placed at its final position and it divides the array into two parts.

8.8.1 Complexity of Quicksort

8.8.1.1 Worst case

Suppose partition of array are really unbalanced. The one of the partitions will contain no element and other partition will contain. n-1 elements and one is pivot element.



When we total up the portioning times for each level, we get

$Cn + C(n-1) + c(h-2) + \ldots + 2c$
$= c(n + (n-1) + (n-2) + \ldots + 2)$
$= c((n+1)(n/2)-1)$

The last line is because 1+2+3+....+ n is the airthmatic series (we substract 1 because for quicksort, the summation starts 2, not (1). We have some law-order terms and constant co-efficients, but whenwe use big Θ notation so worst case running time is $\Theta(n^2)$

Write a program to implement quick sort algorithm

```
# include <stdio.h>
# include <conio.h>
int partition(int ar[], int beg ,int end )
void quick_sort(int a[], int beg ,int end )
void main()
{
int arr[10], i,n;
clrscr();
printf(" \n Enter the number of elements in the array: ");
scanf(" %d", &n);
```

```
    printf(" \n Enter the elements of the array  ");
    for(i=0;i<n;i++)
    {
    scanf(" %d", &arr[i]);

    }
    quick_sort(arr,0, n-1);
     printf("\n  The sorted array is:\n");
    for(i=0;i<n;i++)
     printf("%d\t", arr[i]);
    getch();
  int partition (int a[], int beg,int end)
  {
    int left, right, temp, loc,flag;
    loc=left=beg;
    right=end;
    flag=0;
    while(flag!=1)
    {
      while((a[loc]<= a[right]) && (loc!= right))
      right --;
      if (loc==right);
      flag =1;
      else if(a[loc]>a[right])
    {
        temp=a[loc];
        a[loc]=a[right];
        a[right]=temp;
        loc=right;
      }
      if flag!=1
      {
        while(a[loc]>=a[left]) && (loc!=left))
        left++;
        if(loc==left);
        flag=1;
        else if (a[loc]< a[left])
        {
         temp = a[loc];
          a[loc]= a[left];
          a[left]= temp;
          loc=left;
      }
    }
    }
    return loc;
}
void quick_sort(int arr[],  int beg, int end)
{
  int loc;
  if(beg<end)
  {
```



```
    loc=partition(a, beg,end);
    quick_sort(a, beg, loc-1);
    quick_sort(a, loc+1, end);
  }
}
```

8.8.1.2 Best Case

Best case occurs when the partitions are evenly balanced as possible their sizes either are equal or within each other. 'The former case occurs of the sub array has an odd number of elements and pivot is right in the middle after portioning and each partition has (n-1)/2 elements. The latter case occurs if the sub-arrary has an even number n of elements and one partition has n/2 elements with the other having n/2-1. In either of these cases, each partition has atmost n/2 element, and the tree of sub-problem sizes looks a lot like the tree of sub-problem size for merge sort total portioning time for all sub-problems of this size.

Practically,the efficiency of quick sort depends on the element which is choosen as the pivot.Its worst case efficiency is given as $O(n^2)$. The worst case occurs when the array is already sorted and the left-most elements is choosen as the pivo

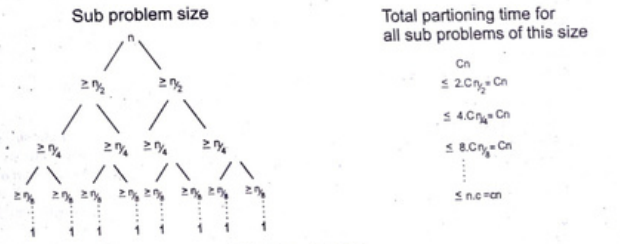


Figure 8.1 Best Case

Using big Θ notation, complexity is Θ (n logn),
Average case running time. It is also Θ (n logn).

8.9 Heapsort

We have already discussed binary heap in chapter 5. Therefore,we already know how to build a heap from an array,how to insert a new element in heap and similiarly delete an element from it.Now,using these basic concepts ,we will discuss the application of heaps to write an efficient algorithm of heapsort.

Usually not quite as fast as quick sort.It is true,``in-place``sort,requiring no auxillary space or storage. It use priority queue and sort n element of array in O(nlogn), it means each element to delete require log n running time.

Given an array of n element,the heap sort algorithm can be used to sort array in two phase:

- In phase1,build a heap using the element of array.
- In phase 2,repeatedly delete the root element of the heap formed in phase 1.

In a max heap ,we know that the largest value in heap always present at the root node.So in phase 2,when the root element is deleted,we are actually collecting the elements of array in decreasing order.The algorithm

of heapsort is shown in Fig.

Heapsort(Arr,N)

```
Step 1: [Build Heap H]
   Repeat for I=0 to N-1
     CALL Insert_Heap(Arr,N,Arr[I])
  [end of Loop]
Step 2: (Repeatdly delete the root element)
   Repeat while N>0
     CALL Delete_Heap(Arr,N,VAL)
     SET N=N+1
  (End Of Loop)
Step 3: END
```

8.9.1 Complexity of Heapsort

Heapsort use two major operations-Insertion and root deletion.Each element deleted from the heap is placed in the last memory location of an array..

In Phase 1, when we build a heap ,the number of comparisons ti find the right location of the new element I Heap cannot exceed the depth of Heap,Since ,Heap is a complete tree,its depth cannot exceed n,where n is the number of elements in the Heap.

Thus, the total number of comparison, f(n) to insert n elements of array in Heap is limited to:

f(n)<=(nlogn)3

In Phase 2, we have Heap which is complete tree with n elements having left and right sub-trees as heaps. Assuming L to be root of tree, *reheaping* the tree would need 4 comparisionto move L tone step down the tree .Since the depth of Heap cannot exceed O(logn),reheaping the tree will require a maximum of 4logn comparisonsto find the right location of L in Heap.

Since ,n element will be deleted from Heap ,reheaping will done n times.Therefore,the number of comparisons to delete n elements is limited to:

f(n)<=4nlogn

Hence, running time of the Phase 2, of the heapsort algorithm is O(nlogn).

Here,we see that each phase require O(nlogn). Therefore,the running time to sort an array of n elements in the worst case is proportional to O(nlogn)..

It is concluded that heapsort is simple,fast and stable sorting algorithm that can be used to sort large sets of data efficiently.

■ ■ ■

Very Short Questions

1. Define Sorting.
2. What is the importance of Sorting.
3. How many types of sorting?
4. Which sorting method is based on Divide and Conquer?
5. Which sorting method need an extra auxillary memory?
6. What to do you understand of complexity in Sorting methods?
7. A card game player arranges his cards and picks them one by one. Which sorting method is used?

8. In which sorting , consecutive adjacent pairs of elements in the array are compared with each other?
9. Which sorting algorithm sorts by moving the current data elements past the already sorted values and repeatedly interchanging it with the preceding value until it is in its correct place?
10. What is the portioning time of array in Quick sort.

Short Questions

1. Define Bubble Sorting with an example.
2. Define Insertion sorting with an example
3. Define selection sorting with an example
4. Define Quick sorting with an example
5. Define Merge sorting with an example. Discuss its time complexity also.
6. Define Radix sorting with an example.

Long Questions

1. Sort the elements 77,49,25,12,9,33,56,.81 using
   (a) Insertion sorting b) selection sorting
   (b) Bubble sorting (d) merge sort
2. If the following sequence of numbers is to be sorted using quick sort , then show iterations of the sorting processes.
   42,34,75,23,21,18,90,67,78
3. Compare the running time complexity of different sorting algorithms.
4. Define Heap. State down heapsort with an example.
5. Compare heap sort and Quick sort.

CbH