Unit 5

Searching, Sorting and Hashing Techniques

5.1. INTRODUCTION TO SEARCHING ALGORITHMS

Searching is an operation or a technique that helps finds the place of a given element or value in the list. Any search is said to be successful or unsuccessful depending upon whether the element that is being searched is found or not. Some of the standard searching technique that is being followed in data structure is listed below:

1. Linear Search
2. Binary Search

5.2. LINEAR SEARCH

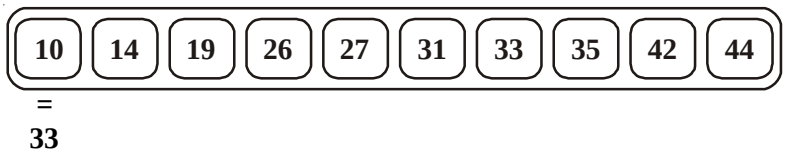
Linear search is a very basic and simple search algorithm. In Linear search, we search an element or value in a given array by traversing the array from the starting, till the desired element or value is found.

It compares the element to be searched with all the elements present in the array and when the element is **matched** successfully, it returns the index of the element in the array, else it return -1.

Linear Search is applied on unsorted or unordered lists, when there are fewer elements in a list.

For Example,

Linear Search



Algorithm

Linear Search (Array A, Value x)

Step 1: Set i to 1

Step 2: if i > n then go to step 7

Step 3: if A[i] = x then go to step 6

Step 4: Set i to i + 1

Step 5: Go to Step 2

Step 6: Print Element x Found at index i and go to step 8

Step 7: Print element not found

Step 8: Exit

Pseudocode

```
procedure linear_search (list, value)
  for each item in the list
    if match item == value
      return the item's location
    end if
  end for
end procedure
```

Linear Search Program

```
#include<stdio.h>
int main()
{
   int a[20],i,x,n;
   printf(—How many elements?‖);
   scanf(-%d‖,&n);
   printf(—Enter array elements:\n‖);
   for(i=0;i<n;++i)
      scanf(-%d‖,&a[i]); printf(-\nEnter
   element to search:‖); scanf(-%d‖,&x);
   for(i=0;i<n;++i)
      if(a[i]==x)
         break;
   if(i<n)
      printf(—Element found at index %d‖,i);
   else
      printf(—Element not found‖);
   return 0;
}
```

Features of Linear Search Algorithm

1. It is used for unsorted and unordered small list of elements.
2. It has a time complexity of **O(n)**, which means the time is linearly dependent on the number of elements, which is not bad, but not that good too.
3. It has a very simple implementation.

5.3. BINARY SEARCH

Binary Search is used with sorted array or list. In binary search, we follow the following steps:

1. We start by comparing the element to be searched with the element in the middle of the list/array.
2. If we get a match, we return the index of the middle element.
3. If we do not get a match, we check whether the element to be searched is less or greater than in value than the middle element.
4. If the element/number to be searched is greater in value than the middle number, then we pick the elements on the right side of the middle element(as the list/array is sorted, hence on the right, we will have all the numbers greater than the middle number), and start again from the step 1.
5. If the element/number to be searched is lesser in value than the middle number, then we pick the elements on the left side of the middle element, and start again from the step 1.

Binary Search is useful when there are large number of elements in an array and they are sorted. So a necessary condition for Binary search to work is that the list/array should be sorted.

Features of Binary Search

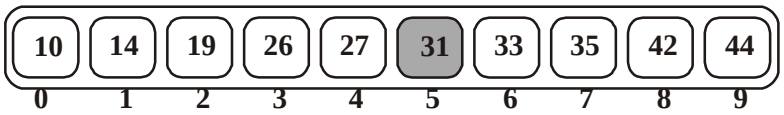
1. It is great to search through large sorted arrays.
2. It has a time complexity of **O(log n)** which is a very good time complexity. It has a simple implementation.

Binary search is a fast search algorithm with run-time complexity of Ï(log n). This search algorithm works on the principle of divide and conquers. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the sub array reduces to zero.

How Binary Search Works?

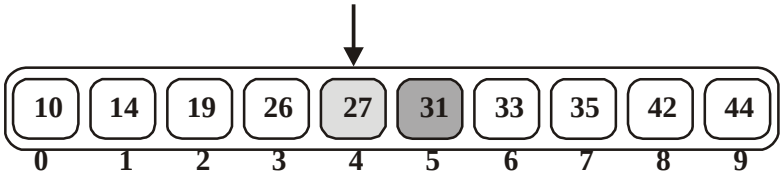
For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.



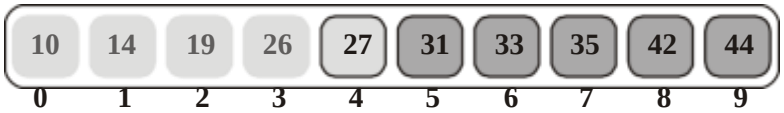
First, we shall determine half of the array by using this formula -

mid = low + (high - low) / 2

Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

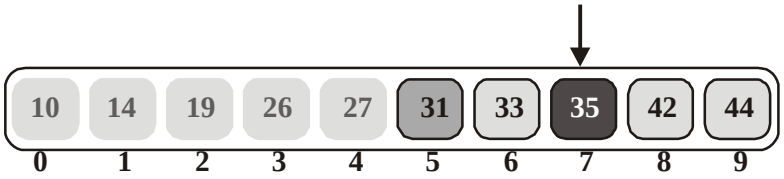


We change our low to mid + 1 and find the new mid value again.

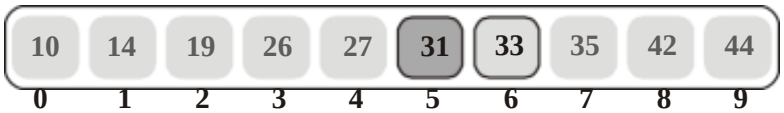
low = mid + 1

mid = low + (high - low) / 2

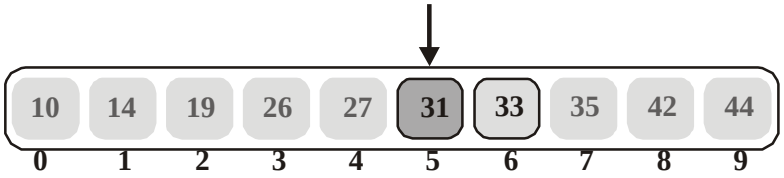
Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.



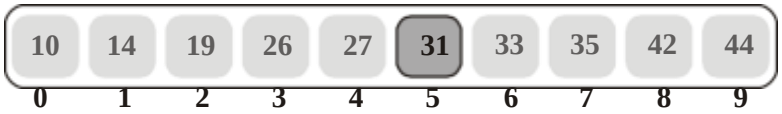
The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.



Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.



We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

Pseudocode

The pseudocode of binary search algorithms should look like this —

```
Procedure binary_search
  A ! sorted array
  n ! size of array
  x ! value to be searched
  Set lowerBound = 1
  Set upperBound = n
  while x not found
    if upperBound < lowerBound
      EXIT: x does not exists.
          set midPoint = lowerBound + ( upperBound - lowerBound ) / 2
     if A[midPoint] < x
      set lowerBound = midPoint + 1
     if A[midPoint] > x
      set upperBound = midPoint - 1
    if A[midPoint] = x
      EXIT: x found at location midPoint
  end while
end procedure
```

Binary Search Program

```
#include<stdio.h>
int main()
{
   int arr[50],i,n,x,flag=0,first,last,mid;
   printf(—Enter size of array:‖);
   scanf(–%d‖,&n);
   printf(–\nEnter array element(ascending order)\n‖);
   for(i=0;i<n;++i)
      scanf(–%d‖,&arr[i]);
   printf(–\nEnter the element to search:‖);
   scanf(–%d‖,&x);
   first=0;
   last=n-1;
   while(first<=last)
   {
      mid=(first+last)/2;
      if(x==arr[mid]){
         flag=1;
         break;
      }
      else
         if(x>arr[mid])
            first=mid+1;
         else
            last=mid-1;
   }
    if(flag==1)
      printf(–\nElement found at position %d‖,mid+1); else
      printf(–\nElement not found‖);
   return 0;
}
```

5.4 SORTING

Preliminaries

A sorting algorithm is an algorithm that puts elements of a list in a certain order. The most used orders are numerical order and lexicographical order. Efficient sorting is important to optimizing the use of other algorithms that require sorted lists to work correctly and for producing human - readable input.

Sorting algorithms are often classified by :

* Computational complexity (worst, average and best case) in terms of the size of the list (N).

  For typical sorting algorithms good behaviour is O(NlogN) and worst case behaviour is $O(N^2)$ and the average case behaviour is O(N).

* Memory Utilization
* Stability - Maintaining relative order of records with equal keys.
* No. of comparisions.
* Methods applied like Insertion, exchange, selection, merging etc.

Sorting is a process of linear ordering of list of objects.

Sorting techniques are categorized into

⇒ Internal Sorting

⇒ External Sorting

Internal Sorting takes place in the main memory of a computer.

eg : - Bubble sort, Insertion sort, Shell sort, Quick sort, Heap sort, etc.

External Sorting, takes place in the secondary memory of a computer, Since the number of objects to be sorted is too large to fit in main memory.

eg : - Merge Sort, Multiway Merge, Polyphase merge.

5.5 THE BUBBLE SORT

The **bubble sort** makes multiple passes through a list. It compares adjacent items and exchanges those that are out of order. Each pass through the list places the next largest value in its proper place. In essence, each item ―bubbles‖ up to the location where it belongs.

Fig. 5.1 shows the first pass of a bubble sort. The shaded items are being compared to see if they are out of order. If there are *n* items in the list, then there are n - 1n - 1 pairs of items that need to be compared on the first pass. It is important to note that once the largest value in the list is part of a pair, it will continually be moved along until the pass is complete.

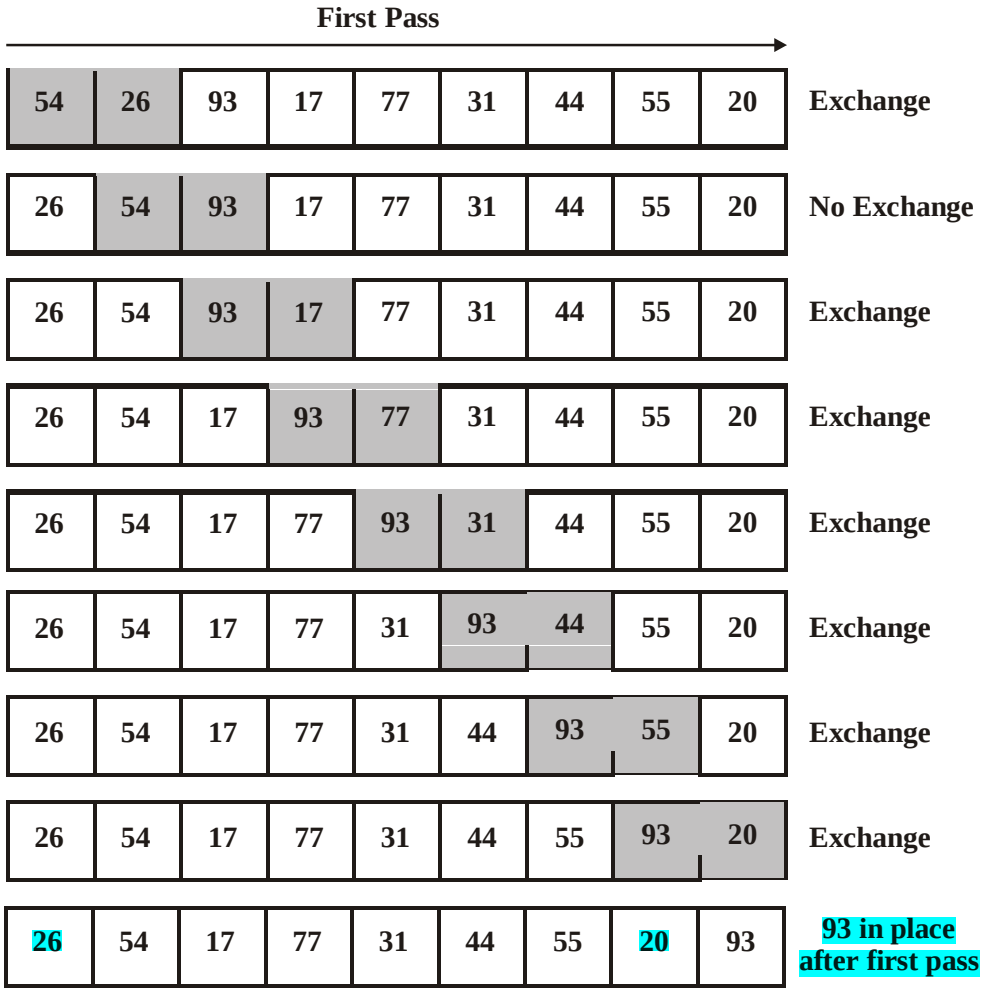


Fig. 5.1 Bubble Sort

At the start of the second pass, the largest value is now in place. There are n - 1n - 1 items left to sort, meaning that there will be n - 2 n - 2 pairs. Since each pass places the next largest value in place, the total number of passes necessary will be n - 1 n - 1. After completing the n - 1 n - 1 passes, the smallest item must be in the correct position with no further processing required. The exchange operation, sometimes called a ―swapl.

ROUTINE for Bubble sort

```
/ C program for implementation of Bubble sort
#include <stdio.h>
 void swap(int *xp, int *yp)
{
   int temp = *xp;
```

```
    *xp = *yp;
    *yp = temp;
}
// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
  int i, j;
  for (i = 0; i < n-1; i++)
      // Last i elements are already in place
      for (j = 0; j < n-i-1; j++)
         if (arr[j] > arr[j+1])
            swap(&arr[j], &arr[j+1]);
}
/* Function to print an array */
void printArray(int arr[], int size)
{
   int i;
   for (i=0; i < size; i++)
      printf("%d ", arr[i]);
   printf("n");
}
```

Program for bubble sort:

```
def bubbleSort(alist):
    for passnum in range(len(alist)-1,0,-1):
        for i in range(passnum):
            if alist[i]>alist[i+1]:
                temp = alist[i]
                alist[i] = alist[i+1]
                alist[i+1] = temp
alist = [54,26,93,17,77,31,44,55,20]
bubbleSort(alist)
print(alist)
```

Output:

[17, 20, 26, 31, 44, 54, 55, 77, 93]

Analysis:

To analyze the bubble sort, we should note that regardless of how the items are arranged in the initial list, n‖1n‖1 passes will be made to sort a list of size *n*. Table -1 shows the number of comparisons for each pass. The total number of comparisons is the sum of the first n - 1n - 1 integers. In the best case, if the list is already ordered, no exchanges will be made. However, in the worst case, every comparison will cause an exchange. On average, we exchange half of the time.

Pass	Comparisons
1	n - 1n - 1
2	n - 2n - 2
3	n - 3n - 3
...	...
n - 1n - 1	11

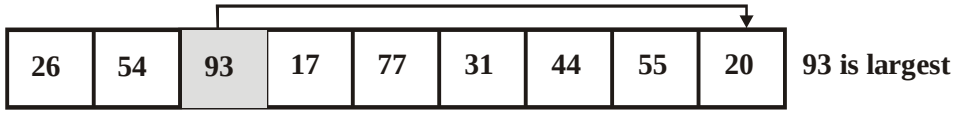
Disadvantages:

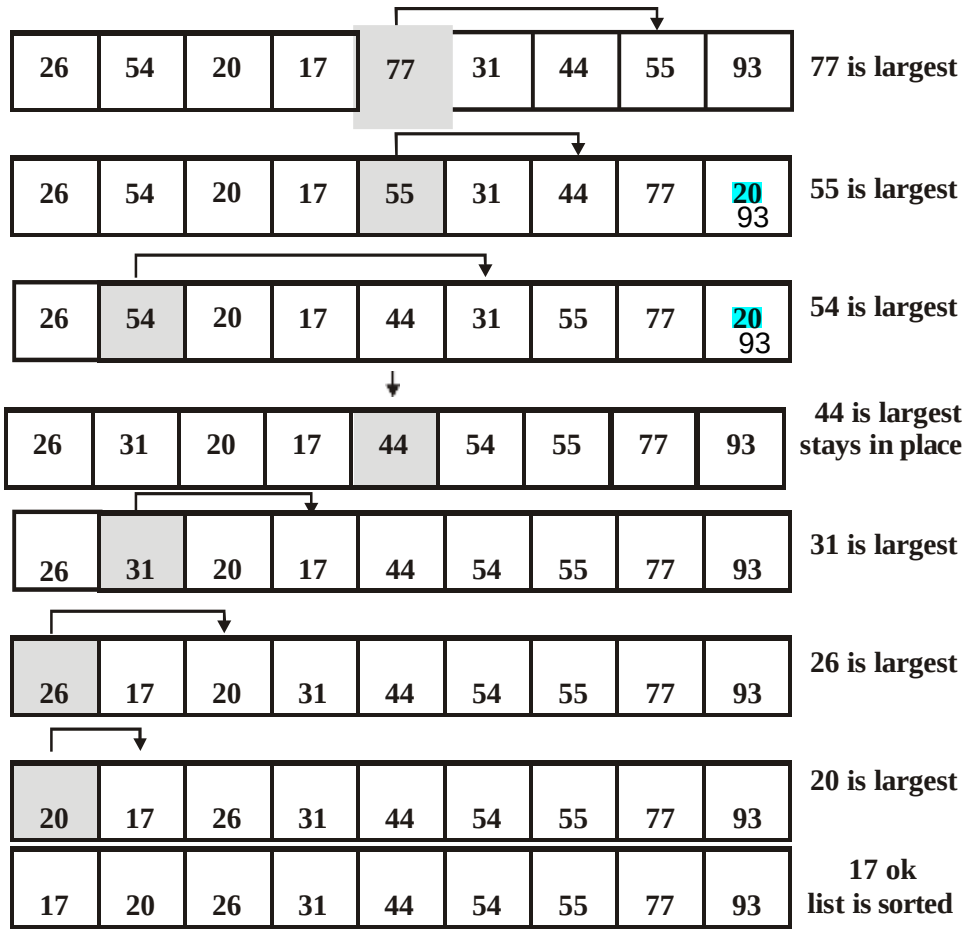
A bubble sort is often considered the most inefficient sorting method since it must exchange items before the final location is known. These ―wasted‖ exchange operations are very costly. However, because the bubble sort makes passes through the entire unsorted portion of the list, it has the capability to do something most sorting algorithms cannot. In particular, if during a pass there are no exchanges, then we know that the list must be sorted. A bubble sort can be modified to stop early if it finds that the list has become sorted. This means that for lists that require just a few passes, a bubble sort may have an advantage in that it will recognize the sorted list and stop

5.6. THE SELECTION SORT

The **selection sort** improves on the bubble sort by making only one exchange for every pass through the list. In order to do this, a selection sort looks for the largest value as it makes a pass and, after completing the pass, places it in the proper location. As with a bubble sort, after the first pass, the largest item is in the correct place. After the second pass, the next largest is in place. This process continues and requires n‖1n‖1passes to sort *n* items, since the final item must be in place after the (n‖1)(n‖1) last pass.

Figure shows the entire sorting process. On each pass, the largest remaining item is selected and then placed in its proper location. The first pass places 93, the second pass places 77, the third places 55, and so on.





ROUTINE for Selection sort

```
// C program for implementation of selection sort
#include <stdio.h>
 void swap(int *xp, int *yp)
{
   int temp = *xp;
   *xp = *yp;
   *yp = temp;
}
 void selectionSort(int arr[], int n)
{
```

```
    int i, j, min_idx;
     // One by one move boundary of unsorted subarray
    for (i = 0; i < n-1; i++)
    {
       // Find the minimum element in unsorted array
       min_idx = i;
       for (j = i+1; j < n; j++)
         if (arr[j] < arr[min_idx])
           min_idx = j;

       // Swap the found minimum element with the first element
       swap(&arr[min_idx], &arr[i]);
    }
}
 /* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
       printf("%d ", arr[i]);
    printf("\n");
}
 // Driver program to test above functions
int main()
{
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr)/sizeof(arr[0]);
    selectionSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

Program for Selection Sort:

```
def selectionSort(alist):
    for fillslot in range(len(alist)-1,0,-1):
        positionOfMax=0
        for location in range(1,fillslot+1):
            if alist[location]>alist[positionOfMax]:
                positionOfMax = location
        temp = alist[fillslot]
        alist[fillslot] = alist[positionOfMax]
        alist[positionOfMax] = temp
alist = [54,26,93,17,77,31,44,55,20]
selectionSort(alist)
print(alist)
```

Output:

```
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

5.7 INSERTION SORT

Insertion sorts works by taking elements from the list one by one and inserting them in their current position into a new sorted list. Insertion sort consists of N - 1 passes, where N is the number of elements to be sorted. The i^{th} pass of insertion sort will insert the i^{th} element A[i] into its rightful place among A[1], A[2] --- A[i - 1]. After doing this insertion the records occupying A[1]. A[i] are in sorted order.

Insertion Sort Procedure

```
void Insertion_Sort (int a[ ], int n)
{
  int i, j, temp ;
  for (i = 0; i < n ; i++)
  {
        temp = a[i] ;
        for (j = i ; j>0 && a[j-1] > temp ; j--)
```

```
            {
                a[j] = a[ j - 1] ;
            }
            a[j] = temp ;
        }
    }
```

Example

Consider an unsorted array as follows,

20 10 60 40 30 15

Passes of Insertion Sort

ORIGINAL	20	10	60	40	30	15	POSITIONS MOVED
After i = 1	10	20	60	40	30	15	1
After i = 2	10	20	60	40	30	15	0
After i = 3	10	20	40	60	30	15	1
After i = 4	10	20	30	40	60	15	2
After i = 5	10	15	20	30	40	60	4
Sorted Array	10	15	20	30	40	60	

Analysis Of Insertion Sort

WORST CASE ANALYSIS - $O(N^2)$

BEST CASE ANALYSIS - $O(N)$

AVERAGE CASE ANALYSIS - $O(N^2)$

Limitations Of Insertion Sort :

* It is relatively efficient for small lists and mostly - sorted lists.
* It is expensive because of shifting all following elements by one.

5.8 SHELL SORT

Shell sort was invented by Donald Shell. It improves upon bubble sort and insertion sort by moving out of order elements more than one position at a time. It works by arranging the data sequence in a two - dimensional array and then sorting the columns of the array using insertion sort.

In shell short the whole array is first fragmented into K segments, where K is preferably a prime number. After the first pass the whole array is partially sorted. In the next pass, the value of K is reduced which increases the size of each segment and reduces the number of segments. The next value of K is chosen so that it is relatively prime to its previous value. The process is repeated

until K = 1, at which the array is sorted. The insertion sort is applied to each segment, so each successive segment is partially sorted. The shell sort is also called the Diminishing Increment Sort, because the value of K decreases continuously.

Shell Sort Routine

```
void shellsort (int A[ ], int N)
{
    int i, j, k, temp;
    for (k = N/2; k > 0 ; k = k/2)
        for (i = k; i < N ; i++)
        {
            temp = A[i];
            for (j = i; j >= k & & A [ j - k] > temp ; j = j - k)
            {
                A[j] = A[j - k];
            }
                A[j] = temp;
        }
}
```

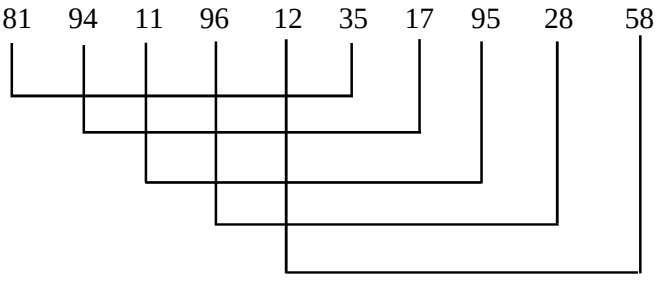
Example

Consider an unsorted array as follows.

81 94 11 96 12 35 17 95 28 58

Here N = 10, the first pass as K = 5 (10/2)

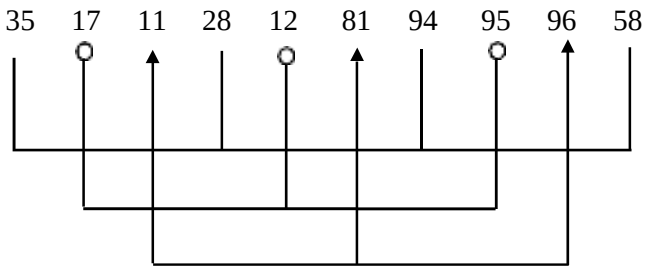
81 94 11 96 12 35 17 95 28 58



After first pass

35 17 11 28 12 81 94 95 96 58

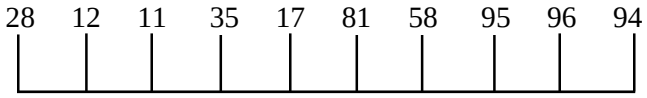
In second Pass, K is reduced to 3



After second pass,

28 12 11 35 17 81 58 95 96 94

In third pass, K is reduced to 1



The final sorted array is

11 12 17 28 35 58 81 94 95 96

Analysis Of Shell Sort :

WORST CASE ANALYSIS - $O(N^2)$
BEST CASE ANALYSIS- O(N log N)
AVERAGE CASE ANALYSIS - $O(N^{1.5})$

Advantages Of Shell Sort :

* It is one of the fastest algorithms for sorting small number of elements.
* It requires relatively small amounts of memory.

5.9. RADIX SORT

Radix sort is a small method used when alphabetizing a large list of names. Intuitively, one might want to sort numbers on their most significant digit. However, Radix sort works counter-intuitively by sorting on the least significant digits first. On the first pass, all the numbers are sorted on the least significant digit and combined in an array. Then on the second pass, the entire numbers are sorted again on the second least significant digits and combined in an array and so on.

Algorithm: Radix-Sort (list, n)

```
shift = 1
for loop = 1 to keysize do
  for entry = 1 to n do
    bucketnumber = (list[entry].key / shift) mod 10
    append (bucket[bucketnumber], list[entry])
  list = combinebuckets()
  shift = shift * 10
```

Analysis

Each key is looked at once for each digit (or letter if the keys are alphabetic) of the longest key. Hence, if the longest key has **m** digits and there are **n**keys, radix sort has order **O(m.n)**.

However, if we look at these two values, the size of the keys will be relatively small when compared to the number of keys. For example, if we have six-digit keys, we could have a million different records.

Here, we see that the size of the keys is not significant, and this algorithm is of linear complexity **O(n)**.

Example

Following example shows how Radix sort operates on seven 3-digits number.

Input	1st Pass	2nd Pass	3rd Pass
329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

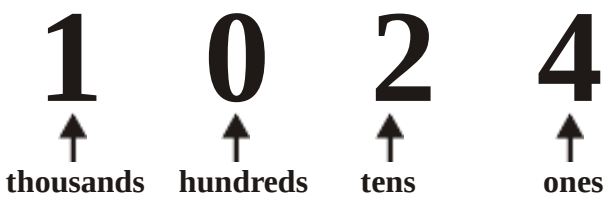
In the above example, the first column is the input. The remaining columns show the list after successive sorts on increasingly significant digits position. The code for Radix sort assumes that each element in an array ***A*** of ***n***elements has ***d*** digits, where digit ***1*** is the lowest-order digit and ***d*** is the highest-order digit.

Example

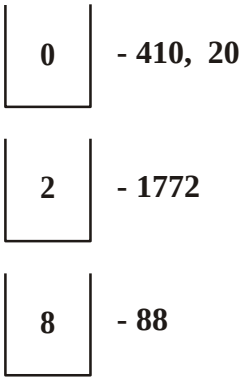
To show how radix sort works:

var array = [88, 410, 1772, 20]

Radix sort relies on the positional notation of integers, as shown here:



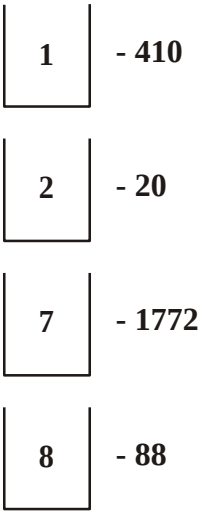
First, the array is divided into buckets based on the value of the least significant digit: the ***ones*** digit.



These buckets are then emptied in order, resulting in the following partially-sorted array:

array = [410, 20, 1772, 88]

Next, repeat this procedure for the ***tens*** digit:



The relative order of the elements didn‘t change this time, but you‘ve still got more digits to inspect.

The next digit to consider is the ***hundreds*** digit:

0	**- 20, 88**
4	**- 410**
7	**- 1772**

For values that have no hundreds position (or any other position without a value), the digit will be assumed to be ***zero***.

Reassembling the array based on these buckets gives the following:

array = [20, 88, 410, 1772]

Finally consider the ***thousands*** digit:

0	**- 20, 88, 410**
1	**- 1772**

Reassembling the array from these buckets leads to the final sorted array:

array = [20, 88, 410, 1772]

When multiple numbers end up in the same bucket, their relative ordering doesn‘t change. For example, in the zero bucket for the hundreds position, 20 comes before 88. This is because the previous step put 20 in a lower bucket than 80, so 20 ended up before 88 in the array.

5.10 HASHING

Hash Table

The hash table data structure is an array of some fixed size, containing the keys. A key is a value associated with each record.

Location	Slot 1
1	
2	92
3	43
4	
5	85
6	
7	
8	
9	
10	

Fig. 5.10 Hash Table

5.11 HASHING FUNCTION

A hashing function is a key - to - address transformation, which acts upon a given key to compute the relative position of the key in an array.

A simple Hash function

HASH (KEYVALUE) = KEYVALUE MOD TABLESIZE

Example : - Hash (92)

Hash (92) = 92 mod 10 = 2

The keyvalue _92‘ is placed in the relative location _2‘.

Routine For Simple Hash Function

```
Hash (Char *key, int Table Size)
{
        int Hashvalue = 0;
        while (* key ! = _\0‘)
                Hashval + = * key ++;
        return Hashval % Tablesize;
}
```

Some of the Methods of Hashing Function

1. Module Division
2. Mid - Square Method
3. Folding Method
4. PSEUDO Random Method
5. Digit or Character Extraction Method
6. Radix Transformation.

Collisions

A Collision occurs when two or more elements are hashed (mapped) to same value (i.e) When two key values hash to the same position.

Collision Resolution

When two items hash to the same slot, there is a systematic method for placing the second item in the hash table. This process is called collision resolution.

Some of the Collision Resolution Techniques

1. Seperate Chaining 2. Open Addressing 3. Multiple Hashing

5.12 SEPERATE CHAINING

Seperatechaining is an open hashingtechnique. Apointer field is added to each record location. When an overflow occurs this pointer is set to point to overflow blocks making a linked list.

In this method, the table can never overflow, since the linked list are only extended upon the arrival of new keys.

Insert : 10, 11, 81, 10, 7, 34, 94, 17

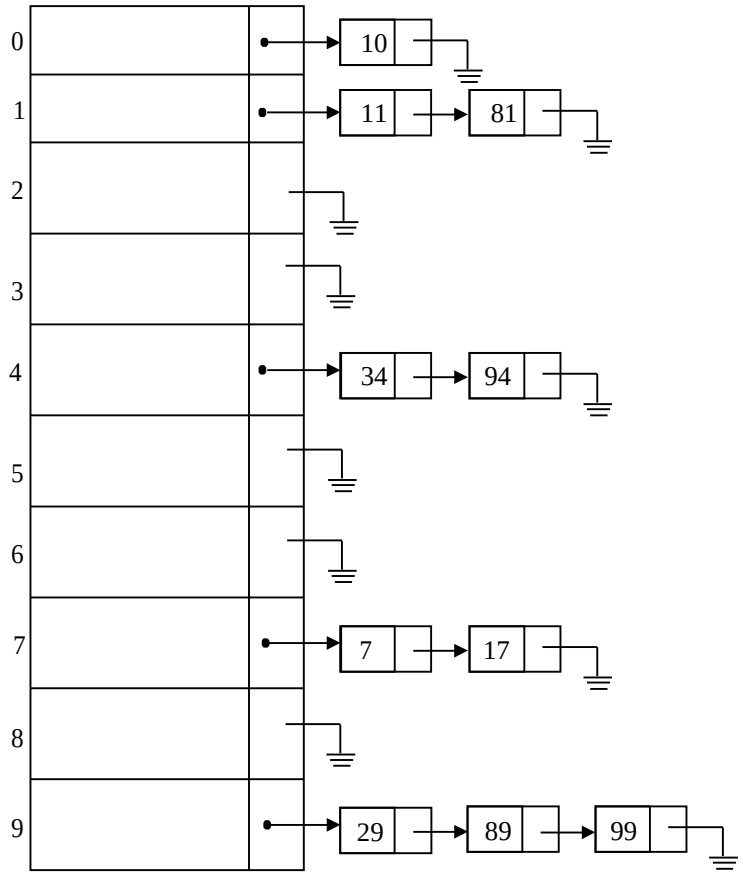


Fig. 5.12

Insertion

To perform the insertion of an element, traverse down the appropriate list to check whether the element is already in place.

If the element is new one, the inserted it is either at the front of the list or at the end of the list.

If it is a duplicate element, an extra field is kept and placed.

INSERT 10 :

Hash (k) = k% Tablesize

Hash (10) = 10 % 10

Hash (10) = 0

INSERT 11 :

Hash (11) = 11 % 10

Hash (11) = 1

INSERT 81 :

Hash (81) = 81% 10

Hash (81) = 1

The element 81 collides to the same hash value 1. To place the value 81 at this position perform the following.

Traverse the list to check whether it is already present.

Since it is not already present, insert at end of the list. Similarly the rest of the elements are inserted.

Routine To Perform Insertion

```
void Insert (int key, Hashtable H)
{
    Position Pos, Newcell;
    List L;
/* Traverse the list to check whether the key is already present */
    Pos = FIND (Key, H);
    If (Pos = = NULL) /* Key is not found */
    {
        Newcell = malloc (size of (struct ListNode));
        If (Newcell ! = NULL)
```

```
                {
                    L = H → Thelists [Hash (key, H → Tablesize)];
                    Newcell → Next = L → Next;
                    Newcell → Element = key;
                    /* Insert the key at the front of the list */
                     L → Next = Newcell;
                }
            }
        }
```

Find Routine

```
    Position Find (int key, Hashtable H)
    {
        Position P;
        List L;
        L = H → Thelists [Hash (key, H → Tablesize)];
        P = L → Next;
        while (P! = NULL && P → Element ! = key)
            P = p → Next;
        return p;
    }
```

Advantage

More number of elements can be inserted as it uses array of linked lists.

Disadvantage of Seperate Chaining

* It requires pointers, which occupies more memory space.
* It takes more effort to performa search, since it takes time to evaluate the hash function and also to traverse the list.

5.13 OPEN ADDRESSING

Open addressing is also called **closed Hashing**, which is an alternative to resolve the collisions with linked lists.

In this hashing system, if a collision occurs, alternative cells are tried until an empty cell is found. (ie) cells $h_0(x)$, $h_1(x)$, $h_2(x)$ .... are tried in succession.

INSERT : 42, 39, 69, 21, 71, 55, 33							
EMPTY TABLE	After 42	After 39	After 69	After 21	After 71	After 55	After 33
0			69	69	69	69	69
1				21	21	21	21
2	42	42	42	42	42	42	42
3					71	71	71
4							33
5						55	55
6							
7							
8							
9		39	39	39	39	39	39

There are three common collision resolution strategies. They are

(i) Linear Probing

(ii) Quadratic probing

(iii) Double Hashing.

5.13.1 Linear Probing

In linear probing, for the i^{th} probe the position to be tried is (h(k) + i) mod tablesize, where F(i) = i, is the linear function.

In linear probing, the position in which a key can be stored is found by sequentially searching all position starting from the position calculated by the hash function until an empty cell is found.

If the end of the table is reached and no empty cells has been found, then the search is continued from the beginning of the table. It has a tendency to create clusters in the table.

In fig 4.6.3 first collision occurs when 69 is inserted, which is placed in the next available spot namely spot 0, which is open.

The next collision occurs when 71 is inserted, which is placed in the next available spot namely spot 3. The collision for 33 is handled in a similar manner.

Advantage :

* It doesn‘t requires pointers

Disadvantage

* It forms clusters, which degrades the performance of the hash table for storing and retrieving data.

5.13.2. Quadratic probing

Quadratic Probing is similar to Linear probing. The difference is that if the element is inserted into a space that is filled , then 1^2 = 112=1 element away then 2^2 = 422=4 elements away, then 3^2 =932=9elements away then 4^2=1642=16 elements away and so on is checked.

With linear probing, there is always an open spot found, if one. However, this is not the case with quadratic probing unless choosing of the table size is considered.

For example

Table size is 16. First 5 pieces of data that all hash to index 2

- First piece goes to index 2.
- Second piece goes to 3 ((2 + 1)%16
- Third piece goes to 6 ((2+4)%16
- Fourth piece goes to 11((2+9)%16
- Fifth piece doesn‘t get inserted because (2+16)%16==2 which is full so we end up back where we started and we haven‘t searched all empty spots.

In order to guarantee that quadratic probes will hit every single available spots eventually, the table size must meet these requirements:

- Be a prime number
- never be more than half full (even by one element)

5.13.3. Double Hashing

Double Hashing works on a similar idea to linear and quadratic probing. A big table is used and hashed into it. Whenever a collision occurs, another spot is chosen in table to put the value. The difference here is that instead of choosing next opening, a second hash function is used to determine the location of the next spot. For example, given hash function H1 and H2 and key do the following:

- Check location hash1(key). If it is empty, put record in it.
- If it is not empty calculate hash2(key).
- check if hash1(key)+hash2(key) is open, if it is, put it in
- repeat with hash1(key)+2*hash2(key), hash1(key)+3*hash2(key) and so on, until an opening is found.

Like quadratic probing, choosing hash2 is considered . hash2 CANNOT ever return 0. hash2 must be done so that all cells will be probed eventually.

5.14 REHASHING

If the table gets too full, then the rehashing method builds new table that is about twice as big and scan down the entire original hash table, computing the new hash value for each element and inserting it in the new table.

Rehashing is very expensive operation, the running time is O(N), since there are N elements to rehash and the table size is roughly 2N.

Rehashing can be implemented in several ways with quadratic probing such as:

- Rehash, as soon as the table is half full.
- Rehash only when an insertion fails
- Rehash when the table reaches a certain load factor.

Routine for Rehashing for open addressing hash tables

```
HashTable Rehash (HashTable H)
{
    int i, oldsize;
    cell    *old cells;
    oldcells = H ® Thecells;
```

```
        oldsize    =  H ® Tablesize;
        H = InitializeTable (2 * oldsize);
        for (i = 0; i < oldsize ; I++)
            if (oldcells [i]. Info = = Legitimate)
                Insert (oldcells [i] . Element, H);
        free (oldcells);
        return H;
}
```

Suppose the elements 13, 15, 24 and 6 are inserted into an open addressing hash table of size 7 with hash function h(X) = X mod 7 and if linear probing is used to resolve collisions, then the resulting hash table appears as follows:

0	6
1	15
2	
3	24
4	
5	
6	13

Fig. 5.14.1:Open addressing hash table with linear probing with input 13, 15, 6, 24

If 23 is inserted into the table, the resulting table will be over 70 percent full.

0	6
1	15
2	23
3	24
4	
5	
6	13

Fig. 5.14.2

A new table is created, as table is so full. The size of this table is 17, as this is the first prime that as twice as large as theold table size. The new hash function is then h(X) = X mod 17. The old table is scanned and the elements 6, 15, 23, 24 and 13 are inserted into the new table.

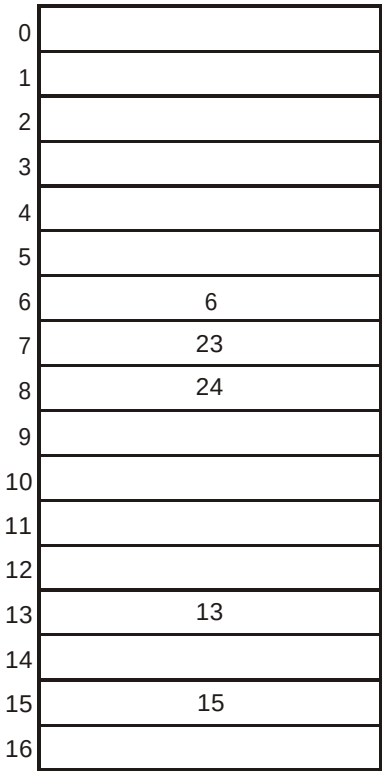


Fig. 5.14.3

Advantages

- Programmer doesn‘t worry about the table size.
- Simple to implement.
- Can be used in other data structures as well.

5.15 EXTENDIBLE HASHING

When open addressing hashing or separate chaining hashing is used, collisions could cause several blocks to be examined during a Find, even for a well-distributed hash table. Furthermore, when the table gets too full, an extremely expensive rehashing step must be performed, which requires O(N) disk accesses.

These problems can be overcomed by using extendible hashing.

Extendible hashing, allows a Find to be performed in two disk accesses. Insertions also requires few disk accesses.

Let us suppose, consider our data consists of several six bit integers. The root of the –tree contains four pointers determied by the leading two bits of the data. Each leaf has upto M = 4 elements. In each leaf the first two bits are identified; this is indicated by the number in parenthesis. D will represent the number of bits used by the root, also known as the directory. The number of entries in the directory is 2^D. d_L is the number of leading bits that all the elements of some leaf L have in common. $d_L \pounds D$. The extendible hashing scheme for these data is given below.

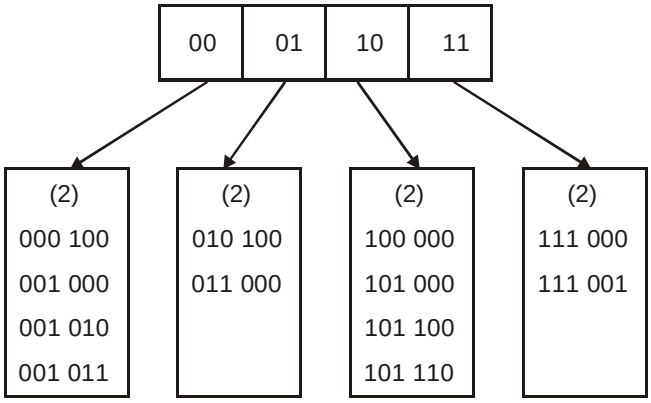


Fig. 5.15.1: Extendible hashing: Original data

Suppose that we want to insert the key 100100. This would go into the third leaf, but as the third leaf is already full, there is no room. We thus split this leaf into two leaves, which are now determined by the first three bits. Now the directory size is increased to 3.

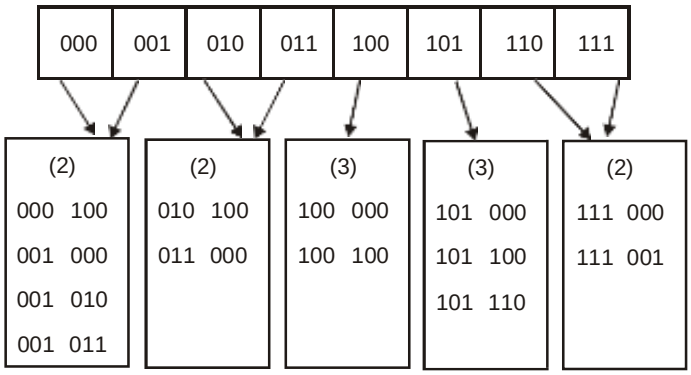


Fig. 5.15.2: Extendible hashing: after insertion of 100100 and directory split

Similarly, if the key 000000 is inserted, then the first leaf is split, generating two leaves with $d_L = 3$. The 000 and 001 pointers are updated.

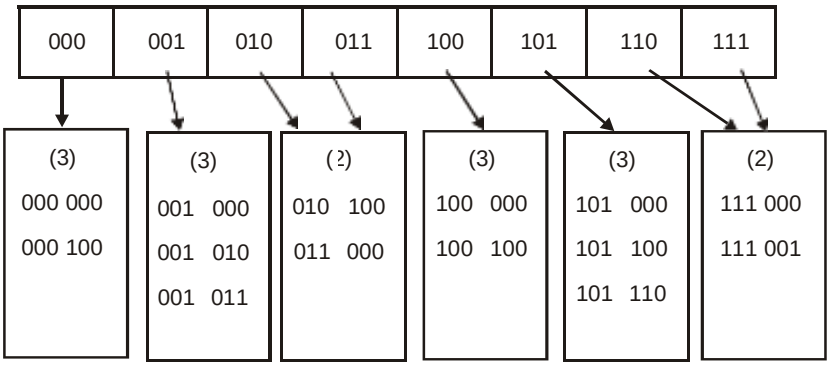


Fig. 5.15.3: Extendible hashing: after insertion of 000000 and leaf split

Advantage

- Provides quick access times for insert and Find operations on large databases.

Disadvantage

- This algorithm does not work if there are more than M duplicates.

If the elements in a leaf agree in more than D + 1 leading bits, then several directory splits is possible. The expected size of the directory is O ($N^{1+1/M}$ / M).

PART - A

1. Define Hashing.
2. Wrtie a routine to find to perform Hash function.
3. What is collision? What are the different collision resolving techniques?
4. What is open addressing?
5. What is separate chaining?
6. What do you mean by resolving?
7. What is an extendible hashing?

PART - B

1. Define Hash function. Write routines to find and insert an element in separate chaining.
2. Explain re-hashing techniques to avoid collision.
3. Explain extendible hashing to resolve collision.