UNIT - V

SORTING AND HASHING.

HASHING

a) Hashing is a technique used for performing insertions, deletions and Searching in Constant average time.

b) The Implementation of hash tables is frequently called as Hashing. Hash table is a ADT which supports only a subset of the operations allowed by binary search trees.

c) An ideal hash table data structure is merely an array of some fixed size, Containing the keys. Typically, a key is a String with an associated value (for example: Employee information)

d) we will refer to the table size as "TableSize" and it is a Common convention to have the table run from 0 to TableSize - 1.

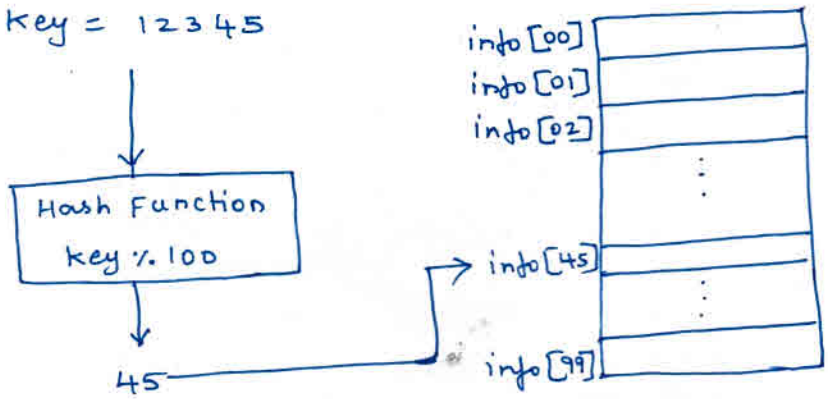
e) Each key is mapped into Some number in the range 0 to TableSize - 1 and placed in the appropriate cell.

f) This mapping is called a 'Hash function', which ideally should be simple to compute and should ensure that any two distinct keys get different cells.

g) Since there are a finite number of cells and virtually inexhaustible supply of keys, mapping to different cells all

(2) Example :-

consider a small company with 100 employees that uses its employees five digit ID number as the primary key. Now, the range of key values is from 00000 to 99999. obviously it is impractical to set up an array of 1,00,000 elements under the condition that at most only very few locations actually would be used.



b. The two most important Problem is,

i) Choosing a hash function and deciding what to do when two keys hash to the same value (this is known as Collision).

ii) Deciding on the table size.

Hash Function:

First Attempt:

a) If the input keys are integers, then simply returning (key mod Tablesize) is generally a reasonable strategy, unless key happens to have some undesirable properties.

b) In such case, the choice of hash function needs to be considered. For instance, if the table size is 10 and

c) To avoid situations like this, it is usually a good idea to ensure that the table size is prime. When the input keys are random integers, then this function is very simple to compute and also distributes the keys evenly.

d) usually, the keys are strings, in this case, the hash function needs to be chosen carefully. one option is to add up the ASCII values of the characters in the string.

Example:-

```
Hash (const char *key, int TableSize)
{
    Unsigned int HashVal = 0;
    While ( *key != '\0')
      HashVal += *key++;
    return HashVal % TableSize;
}
```

c) The hash function above is simple to implement and computes an answer quickly. However, if the table size is large, the function does not distribute the keys well.

d) Suppose that TableSize = 10,007 (10,007 is a prime number), and all the keys are eight or fewer characters long. Since a 'char' has an integer value that is always at most 127, the hash function can only assume values between 0 and 1,016 (which is 127 * 8). This is clearly not an equitable distribution.

(4)

Second Attempt :-

a) Another Hash function assumes that key has atleast two characters plus the NULL terminator.

Example:

```
Hash (Const Char *key, int Tablesize)
{
  return (key[0] + 27 * key[1] + 729 * key[2]) % Tablesize;
}
```

b) The value 27 represents the number of letters in the English Alphabet, plus the blank, and 729 is 27^2.

c) This function examines only the first three characters, but if these are random and the table size is 10,007, as before, then we would expect a reasonably equitable distribution.

d) Unfortunately English is not random. Although there are $26^3 = 17,576$ possible combinations of three characters (ignoring blanks), a check of a reasonably large on-line dictionary reveals that the number of different combinations is actually only 2,851.

e) Even if none of these combinations collide, only 28 percent of the table can only be hashed to. Thus, this function, although easily computable, it also not appropriate if the hash table is reasonably large.

Third Attempt.

a) This hash function involves all characters in the keys and can generally be expected to distribute well.
(it computes $\sum_{i=0}^{keysize-1} key[keysize-i-1] * 32^i$, and brings the result into proper range).

```
Hash(const char *key, int Tablesize)
{
    Unsigned int Hashval = 0;
    while (*key != '\0')
        Hashval = (Hashval << 5) + *key++;
    return Hashval % Tablesize;
}
```

b. The code computes a polynomial function (of 32) by use of Horner's rule.

~~c. The hash function designed~~

c. While designing hash function, the following points should be considered.

i) The hash function designed should be simple to implement.

ii) Computing time should be less.

iii) It should distribute the cells evenly.

(6)

Collision:

The main programming aspect is Collision resolution. If, when an element is inserted, it hashes to the same value as an already inserted element, then we have a collision and need to resolve it. There are several methods for dealing with collision, we will discuss two simplest techniques,

i) Seperate Chaining.

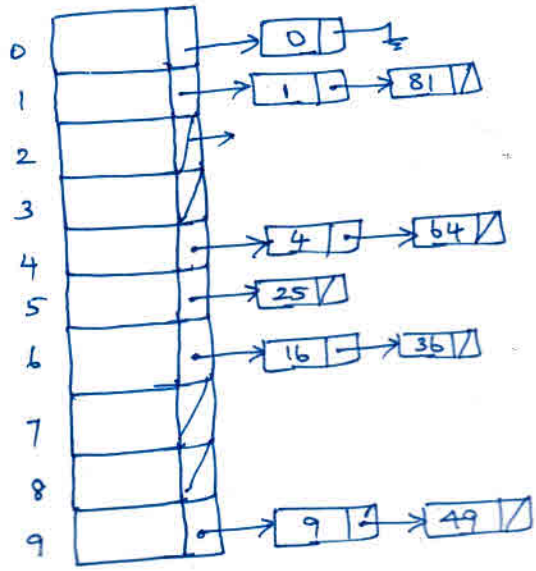
ii) Open addressing.

Separate Chaining:

a) seperate chaining is an collision resolution strategy in which all the elements that hash to the same value are kept in a list.

b) we assume that the keys are the first 10 perfect squares and that the hasing function is simply Hash(x) = x MOD 10.

A seperate chaining Hash table



c) To perform 'find', we use the hash function to determine which list to traverse. we then traverse the list in normal manner, returning the position where the item is found.

d) To perform an 'Insert', we traverse down the appropriate list to check whether the element is already in place. If the element turns out to be new, it is either inserted at the front of the list or at the end of the list.

Type declaration for seperate chaining hash table

```
Struct ListNode
{
  ElementType Element;
  Position Next;
};
typedef position List;
Struct HashTbl
{
   int TableSize;
   List *TheLists;
};
```

Note:

i) The ListNode structure is same as the linked declarations.

ii) The hash table structure contains an array of linked lists (and the number of lists in the array), which are dynamically allocated when the table is initialized.

iii) "TheLists" field is actually a pointer to pointer to a ListNode structure.

(8)

Initialization Routine for Seperate chaining Hash table.

```
HashTable InitializeTable (int Tablesize)
{
    HashTable H;
    int i;
Line 1  if (Tablesize < MinTableSize)
        {
Line 2    Error("Table Size is too small");
Line 3    return NULL;
        }
          /* Allocate Table */
Line 4  H = malloc(sizeof(struct HashTbl));
Line 5  if (H == NULL)
Line 6    fatalError("out of Space");
Line 7  H->Tablesize = Nextprime(Tablesize);
        /* Allocate array of lists */
Line 8  H->TheLists = malloc(Sizeof(List) * H->Tablesize);
Line 9  if (H->TheLists == NULL)
Line 10   FatalError("out of Space");
          /* Allocate List Headers */
Line 11 for(i=0; i< H->Tablesize; i++)
        {
Line 12   H->TheLists[i] = malloc(sizeof(struct ListNode));
Line 13   if (H->TheLists[i] == NULL)
Line 14     Fatal Error("out of Space");
          else
Line 15     H->TheLists[i]->Next = NULL;
        }
Line 16 return H;
}
```

Note:

i) The initialization function, which uses the same ideas that is used in the array implementation of stacks.

ii) Lines 4 through 6 allocate a hash table structure - If space is available, then 'H' will point to a structure containing an integer and a pointer to a list.

iii) Line 7 sets the table size to a prime number, and lines 8 through 10 attempt to allocate an array of lists.

iv) Lines 11 through 15 allocate one header per list and sets its Next field to NULL.

Find Routine for seperate chaining Hash Table.

```
Position Find (ElementType key, HashTable H)
{
    Position P;
    List L;
Line 1   L = H -> TheLists [Hash (key, H -> TableSize)];
Line 2   P = L -> Next;
Line 3   while (P != NULL && P -> Element != key)
Line 4   P = P -> Next;
Line 5   return P;
}
```

Note:

i) The call Find (key, H) will return a pointer to the cell containing key.

(10) Insert Routine for separate Chaining hash table

```
Void Insert (ElementType key, HashTable H)
{
    Position pos, NewCell;
    List L;
Line1   Pos = Find (key, H);
Line2   if ( Pos == NULL)   /* key is not found */
        {
Line3       NewCell = malloc ( sizeof ( struct ListNode));
Line4       if (NewCell == NULL)
Line5           FatalError ("out of space");
            else
            {
Line6           L = H -> TheLists [Hash( key, H -> TableSize)];
Line7           NewCell -> Next = L -> Next;
Line8           NewCell -> Element = key;
Line9           L -> Next = NewCell;
            }
        }
}
```

Note:

i) If the item to be inserted is already present, then we do nothing, otherwise we place it at the front of the list.

Load Factor (λ).

Load factor (λ) of a hash table to be the ratio of the number of elements in the hash table to the table size.

Note:-

Load factor is important factor to ~~[illegible]~~ traverse the list in an constant average time. The general rule for seperate chaining hashing is to make the table size about as large as the number of elements expected (in other words, [illegible])

OPen Addressing

a) Seperate Chaining hashing has the disadvantage of requiring Pointers. This in turn increases the time required to allocate new cells, and also essentially requires the Implementation of a Second data structure.

b) open addressing hashing is an alternative to resolving Collisions with linked lists.

c) In an open addressing hashing system, if a Collision occurs, alternative cells are tried until an empty cell is found.

d) more formally, cells $h_0(x)$, $h_1(x)$, $h_2(x)$, ... are tried ~~until an empty cell is found~~ in succession, where,

$$h_i(x) = (\text{Hash}(x) + F(i)) \bmod \text{Tablesize}, \text{ with } F(0)=0.$$

The Function 'F', is the Collision resolution strategy.

e) Because all the data go inside the table, a bigger table is needed for open addressing hashing than for seperate chaining hashing.

Generally, the load factor should be below $\lambda = 0.5$ for open addressing hashing.

Three Common Collision resolution strategy in open addressing Scheme are,

i) Linear probing
ii) Quadratic Probing

(12)

Linear Probing

a) In Linear probing, 'F' is a linear function of i, typically $F(i) = i$. This amounts to trying cells sequentially (with wraparound) in search of an empty cell.

Example:-

keys to be inserted :- {89, 18, 49, 58, 69} into the hash table using the hash function, $h_i(x) = (\text{Hash}(x) + F(i)) \bmod \text{Tablesize}$, and the collision resolution strategy, $F(i) = i$.

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Table size = 10

(i.e). To insert 89,

$h_0(89) = (89 + 0) \bmod 10 = 9$

→ To insert 18

$h_0(18) = (18 + 0) \bmod 10 = 8$

→ To insert 49

$h_0(49) = (49 + 0) \bmod 10 = 9$ [Collision, So try next slot]

$h_1(49) = (49 + 1) \bmod 10 = 0$

Similarly Proceed for other keys.

b) As long as the table is big enough, a free cell can always be found, but the time to do so can get quite large.

c) A worst situation is that, even if the table is relatavely empty, blocks of occupied cells start forming. This effect is known as "Primary Clustering" (i-e. Any key that hashes into the cluster will require several attempts to resolve the collision, and then it will add to the cluster.

Quadratic Probing

a) Quadratic probing is a collision resolution method that eliminates the Primary clustering problem of linear probing.

b) In Quadratic probing, the collision function is quadratic. The popular choice is $F(i) = i^2$.

Enamble:-

We will use the same keys for inserting {89, 18, 49, 58, 69}

	Empty table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

c) For quadratic probing, the situation is even more drastic (i.e) there is no guarantee of finding an empty cell once the table gets more than half full, or even before the table gets half full if the table size is not prime. This is because at most half of the table can be used as alternative locations to resolve collisions.

d) Other disadvantages in this scheme is that, if the table is even more than half full, the insertion could fail. If the table size is not prime, the number of alternative locations can be severly reduced.

```
Struct HashEntry
{
  ElementType Element;
  enum KindOfEntry Info;
};
typedef struct HashEntry Cell;

Struct HashTbl
{
  int TableSize;
  Cell *TheCells;
};
```

Routine to Initialize open Addressing Hash Table

```
HashTable InitializeTable (int TableSize)
{
   HashTable H;
   int i;
   if (Tablesize < MinTablesize)
   {
     Error ("Table size is too small");
     return NULL;
   }
     /* Allocate Table */
   H = malloc (sizeof (struct HashTbl));
   if (H == NULL)
     fatalError ("out of space");
   H -> TableSize = Nextprime (TableSize);
     /* Allocate Array of cells */
   H -> TheCells = malloc [sizeof(cell) * H -> Tablesize;
   if (H -> TheCells == NULL)
     fatalError ("out of space");
   for (i=0; i< H -> Tablesize; i++)
     H -> TheCells [i] . Info = Empty;
   return H;
}
```

(16)

Find routine for hashing with quadratic probing.

```
Position Find ( ElementType key, HashTable H)
{
    Position currentpos;
    int CollisionNum;
    CollisionNum = 0;
    currentpos = Hash (key, H->Tablesize);
    While (H->TheCells[currentpos].Info != Empty &&
                    H->TheCells[currentpos].Element != key)

    {
      currentpos += 2 * ++CollisionNum - 1;
      if (currentpos >= H->Tablesize)
      currentpos -= H->Tablesize;
    }
    return currentpos;
}
```

Insert Routine for Hash Tables with Quadratic probing

```
Void Insert( ElementType key, HashTable H)
{
    Position pos;
    pos = Find (key, H);
    if (H->TheCells[pos].Info != legitimate)
    {
      H->TheCells[pos].Info = legitimate;
      H->TheCells [pos] . Element = key;

    }
}
```

Double Hashing

a) The last collision resolution method is double hashing. For double hashing, one popular choice is $F(i) = i \cdot hash_2(x)$. This formula says that we apply a second hash function to x and probe at a distance $hash_2(x)$, $2hash_2(x)$ ... and so on.

Example: we will try to insert the keys {89, 18, 49, 58, 69}

$$h_i(x) = (Hash(x) + F(i)) \bmod TableSize$$

For double hashing

$$F(i) = i * hash_2(x)$$

we will use $hash_2(x) = R - (x \bmod R)$ with 'R' a Prime smaller than TableSize. we will use R = 7.

	Empty Table	After 89	After 18	After 49	After 58	After 69
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

b) If double hashing is Correctly implemented, Simulations imply that the expected number of probes is almost the the same as for a random Collision resolution strategy.

(18)

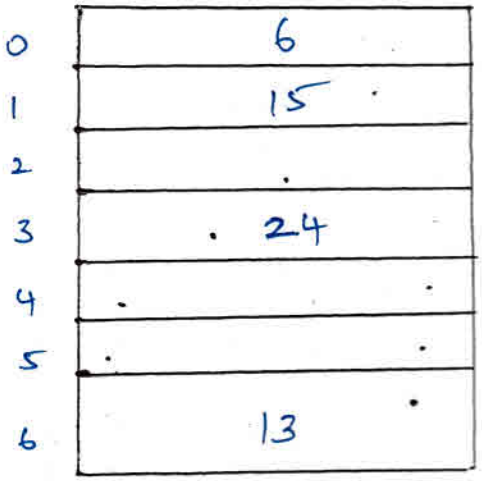
Rehashing

a) If the table gets too full, the running time for the operations will start taking too long and insertion might fail for open addressing hashing with quadratic resolution.

b) This can happen if there are too many removals intermixed with insertions.

c) A solution, then, is to build another table that is about twice as big (with an associated new hash function) and scan down the entire original hash table, computing the new hash value for each (non deleted) element and inserting it in the new table.

d) As an example, suppose the elements 13, 15, 24, and 6 are inserted into an open addressing hash table of size 7. The hash function is $h(x) = x \bmod 7$. Suppose linear Probing is used to resolve collision, then the resulting table appears as,



e) If now 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

f) Because the table is so full, a new table is created. The size of this table is 17, because this is the first Prime that is twice as large as the old table size.

g) The new hash function is then $h(x) = X \bmod 17$. The old table is scanned and elements 6, 15, 23, 24 and 13 are inserted into the new table.

0	
1	
2	
3	
4	
5	
6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	

(20)

b) This entire operation is called Rehashing. Rehashing can be implemented in several ways with quadratic probing. one alternative is to rehash as soon as the table is half full. The other extreme is to rehash when an insertion fails.

(15) Routine for Rehashing open addressing hash tables

```
HashTable Rehash(HashTable H)
{
    int i, oldsize;
    cell *oldcells;

    oldcells = H -> TheCells;
    oldsize = H -> TableSize;
    H = InitializeTable(2 * oldsize);  /* Get a new empty table */
    /* Scan through old table, reinserting into new */
    for (i=0; i < oldsize; i++)
        if (oldcells[i].Info == Legitimate)
            Insert(oldcells[i].Element, H);
    free(oldcells);
    return H;
}
```

Extendible Hashing

a) Suppose the amount of data is too large to fit in main memory, the main Consideration is that the number of disk accesses required to retreive data.

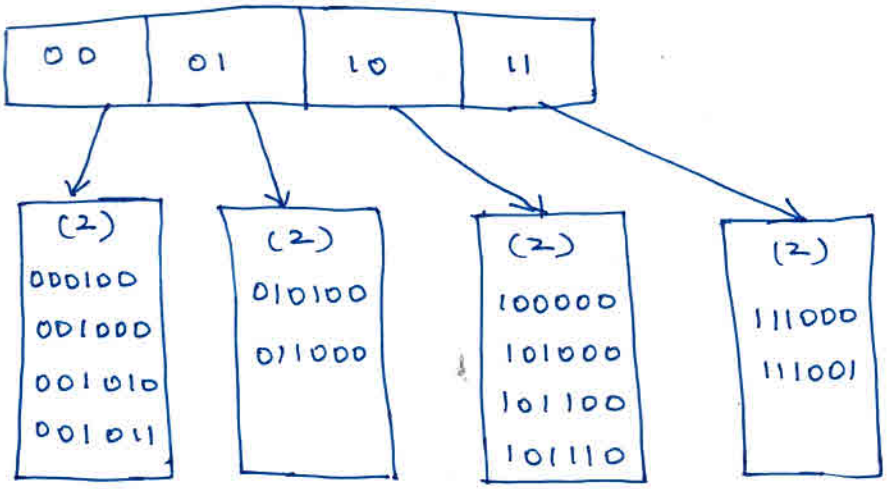
b) Assume that we have 'N' records to store, the value of 'N' changes over time. Also, at most 'M' records fit in one disk block i.e. M = 4

c) If either open Addressing or Seperate Chaining hashing is used, the major Problem is that Collisions Could Cause Several blocks to be examined during find. If the table goes full, an extremely expensive rehashing Step must be performed.

d) An alternative is known as Extendible Hashing, allows 'Find' to be performed in ~~two~~ few disk accesses. Insertion also required few disk accesses.

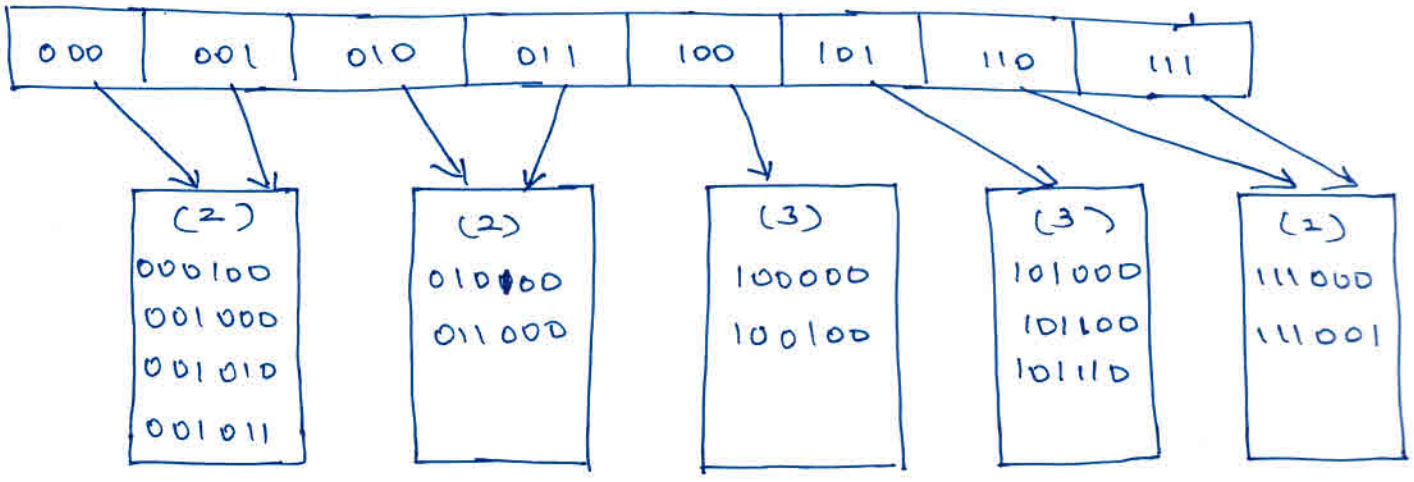
e) Suppose our data consits of Several six-bit data, The root (directory) of the tree Contains four pointers determined by the leading two bits of the data.

Each leaf has up to M=4 elements.



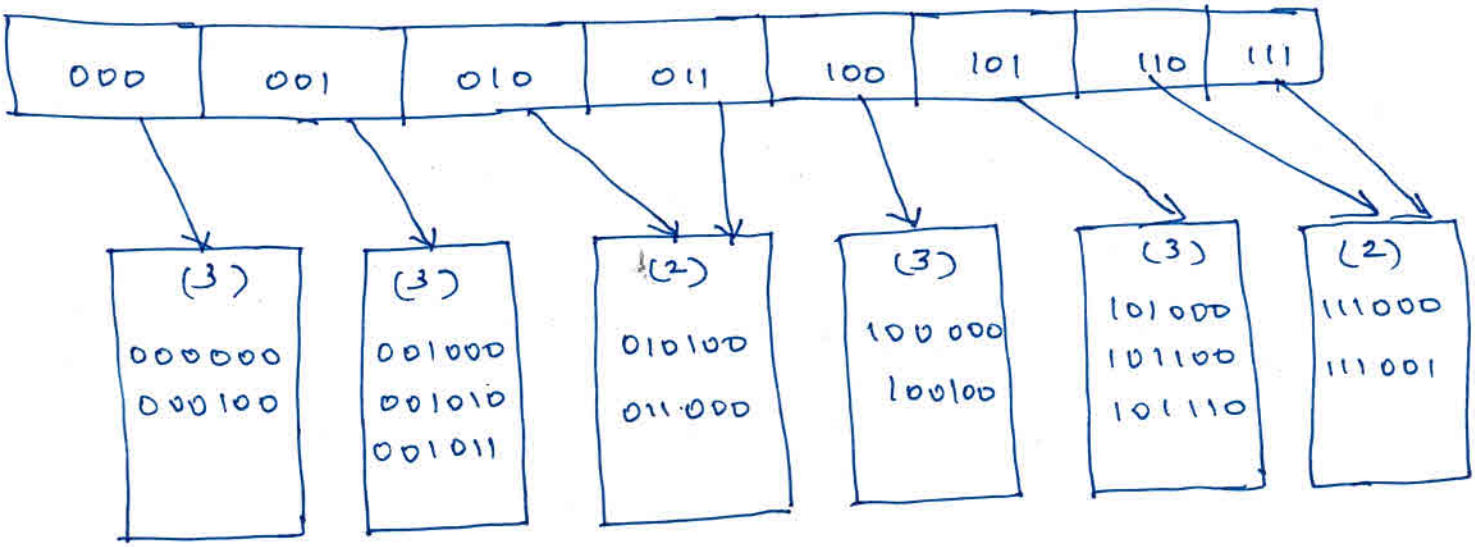
f) The first two bits are identical in each table: Suppose we want to insert the key 100100. This would go into the third leaf, but the third leaf is already full and there is no place. Thus, we would split this leaf into two leaves, which are now determined by first three bits. This requires increasing the directory bit size to 3.

Extendible Hashing after inserting of 100100



If the key 000000 is now inserted, then the first leaf is split. The only change required is the updating of the 000 and 001 pointers.

Extendible Hashing after Insertion of 000000



UNIT 5

Introduction to SEARCHING

Searching means to find whether a particular value is present in an array or not. If the value is present in the array, then searching is said to be successful and the searching process gives the location of that value in the array. However, if the value is not present in the array, the searching process displays an appropriate message and in this case searching is said to be unsuccessful.

There are two popular methods for searching the array elements: *linear search* and *binary search*. The algorithm that should be used depends entirely on how the values are organized in the array. For example, if the elements of the array are arranged in ascending order, then binary search should be used, as it is more efficient for sorted lists in terms ofcomplexity.

Linear Search

Linear search, also called as *sequential search*, is a very simple method used for searching an array for a particular value. It works by comparing the value to be searched with every element of the array one by one in a sequence until a match is found. Linear search is mostly used to search an unordered list of elements (array in which data elements are not sorted). For example, if an array A[] is declared and initialized as,

int A[] = {10, 8, 2, 7, 3, 4, 9, 1, 6, 5};

and the value to be searched is VAL = 7, then searching means to find whether the value '7' is present in the array or not. If yes, then it returns the position of its occurrence. Here, POS = 3 (index starting from0).

LINEAR_SEARCH(A, N, VAL)
Step 1: [INITIALIZE] SET POS = -1
Step 2: [INITIALIZE] SET I = 1
Step 3: Repeat Step 4 while I<=N
Step 4: IF A[I] = VAL
SET POS = I
PRINT POS
Go to Step 6
[END OF IF]
SET I = I + 1
[END OF LOOP]
Step 5: IF POS = –1
PRINT VALUE IS NOT PRESENT
IN THE ARRAY
[END OF IF]
Step 6: EXIT

In Steps 1 and 2 of the algorithm, we initialize the value of POS and I. In Step 3, a while loop is executed that would be executed till I is less than N (total number of elements in the array). In Step 4, a check is made to see if a match is found between the current array element and VAL. If a match is found, then the position of the array element is printed, else the value of I is incremented to match the next element with
VAL. However, if all the array elements have been compared with VAL and no match is found, then it means that VAL is not present in the array.

Complexity of Linear Search Algorithm

Linear search executes in O(n) time where n is the number of elements in the array. Obviously, the best case of linear search is when VAL is equal to the first element of the array. In this case, only one comparison will be made. Likewise, the worst case will happen when either VAL is not present in the array or it is equal to the last element of the array. In both the cases, n comparisons will have to be made. However, the performance of the linear search algorithm can be improved by using a sorted array.

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 a 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. Take another analogy. How do we find words in a dictionary? We first open the dictionary somewhere in the middle. Then, we compare the first word on that page with the desired word whose meaning we are looking for. If the desired word comes before the word on the page, we look in the first half of the dictionary, else we look in the second half. Again, we open a page in the first half of the dictionary and compare the first word on that page with the desired word and repeat the same procedure until we finally get the word. The same mechanism is applied in the binary search.

Now, let us consider how this mechanism is applied to search for a value in a sorted array.

Consider an array A[] that is declared and initialized as
int A[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

and the value to be searched is VAL = 9. The algorithm will proceed in the following manner.
BEG = 0, END = 10, MID = (0 + 10)/2 = 5

Now, VAL = 9 and A[MID] = A[5] = 5
A[5] is less than VAL, therefore, we now search for the value in the second half of the array. So,
we change the values of BEG and MID.
Now, BEG = MID + 1 = 6, END = 10, MID = (6 + 10)/2 =16/2 = 8

VAL = 9 and A[MID] = A[8] = 8
A[8] is less than VAL, therefore, we now search for the value in the second half of the segment.
So, again we change the values of BEG and MID.
Now, BEG = MID + 1 = 9, END = 10, MID = (9 + 10)/2 = 9
Now, VAL = 9 and A[MID] = 9.

In this algorithm, we see that BEG and END are the beginning and ending positions of the segment that we are looking to search for the element. MID is calculated as (BEG + END)/2. Initially, BEG = lower_bound and END = upper_bound. The algorithm will terminate when A[MID] = VAL. When the algorithm ends, we will set POS = MID. POS is the position at which the value is present in the array.
However, if VAL is not equal to A[MID], then the values of BEG, END, and MID will be changed depending on whether VAL is smaller or greater than A[MID].

(a) If VAL <A[MID], then VAL will be present in the left segment of the array. So, the value of END will be changed as END = MID –1.
(b) If VAL >A[MID], then VAL will be present in the right segment of the array. So, the value of BEG will be changed as BEG = MID +1.

Finally, if VAL is not present in the array, then eventually, END will be less than BEG. When this happens, the algorithm will terminate and the search will beunsuccessful.

BINARY_SEARCH(A, lower_bound, upper_bound, VAL)
Step 1: [INITIALIZE] SET BEG = lower_bound
END = upper_bound, POS = - 1
Step 2: Repeat Steps 3 and 4 while BEG <= END
Step 3: SET MID = (BEG + END)/2
Step 4: IF A[MID] = VAL
SET POS = MID
PRINTPOS
Go to Step6
ELSE IF A[MID] > VAL
SET END = MID - 1
ELSE
SET BEG = MID + 1
[END OF IF]
[END OF LOOP]
Step 5: IF POS = -1
PRINT “VALUE IS NOT PRESENT IN THE ARRAY”
[END OF IF]
Step 6:EXIT

In Step 1, we initialize the value of variables, BEG, END, and POS. In Step 2, a while loop is executed until BEG is less than or equal to END. In Step 3, the value of MID is calculated. In Step 4, we check if the array value at MID is equal to VAL (item to be searched in the array). If a match is found, then the value of POS is printed and
the algorithm exits. However, if a match is not found, and if the value of A[MID]
is greater than VAL, the value of END is modified, otherwise if A[MID] is greater
thanVAL, then the value of BEG is altered. In Step 5, if the value of POS = –1, then VAL is not present in the array and an appropriate message is printed on the screen before the algorithm exits.

Complexity of Binary Search Algorithm

The complexity of the binary search can be expressed as f(n), where n is the number of elements in the array. The complexity of the algorithm is calculated depending on the number of comparisons that are made. In the binary search algorithm, we see that with each comparison, the size of the segment where search has to be made is reduced to half. Thus, we can say that, in order to locate a particular value in the array, the total number of comparisons that will be made is given as $2f(n) > n$ or $f(n) = \log n$.

INTRODUCTION TO SORTING

Sorting means arranging the elements of an array so that they are placed in some relevant order which may be either ascending or descending. That is, if A is an array, then the elements of A are arranged in a sorted order (ascending order) in such a way that A[0] <A[1] <A[2] << A[N].
For example, if we have an array that is declared and initialized as
intA[] = {21, 34, 11, 9, 1, 0, 22};
Then the sorted array (ascending order) can be given as:
A[] = {0, 1, 9, 11, 21, 22, 34;

A sorting algorithm is defined as an algorithm that puts the elements of a list in a certain order, which can be either numerical order, lexicographical order, or any user-defined order. Efficient sorting algorithms are widely used to optimize the use of other algorithms like search and merge algorithms which require sorted lists to work correctly. There are two types ofsorting:

- ***Internal sorting*** which deals with sorting the data stored in the computer'smemory
- ***External sorting*** which deals with sorting the data stored in files. External sorting is applied when there is voluminous data that cannot be stored in thememory.

BUBBLE SORT

Bubble sort is a very simple method that sorts the array elements by repeatedly moving the largest element to the highest index position of the array segment (in case of arranging elements in ascending order). In *bubble sorting*, consecutive adjacent pairs 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 procedure of sorting is called 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 (i.e., at the end of the list).

NOTE:
If the elements are to be sorted in descending order, then in first pass the smallest element is moved to the highest index of the array.

Technique

The basic methodology of the working of bubble sort is given as follows:
(a) In Pass 1, A[0] and A[1] are compared, then A[1] is compared with A[2], A[2] is compared with A[3], and so on. Finally, A[N–2] is compared with A[N–1]. Pass 1 involves n–1 comparisons and places the biggest element at the highest index of thearray.

(b) In Pass 2, A[0] and A[1] are compared, then A[1] is compared with A[2], A[2] is compared with A[3], and so on. Finally, A[N–3] is compared with A[N–2]. Pass 2 involves n–2 comparisons and places the second biggest element at the second highest index of the array.

(c) In Pass 3, A[0] and A[1] are compared, then A[1] is compared with A[2], A[2] is compared with A[3], and so on. Finally, A[N–4] is compared with A[N–3]. Pass 3 involves n–3 comparisons and places the third biggest element at the third highest index of thearray.

(d) In Pass n–1, A[0] and A[1] are compared so that A[0]<A[1]. After this step, all the elements of the array are arranged in ascendingorder.

Example

To discuss bubble sort in detail, let us consider an array A[] that has the following elements:
A[] = {30, 52, 29, 87, 63, 27, 19, 54}

Pass 1:
(a) Compare 30 and 52. Since 30 < 52, no swapping isdone.
(b) Compare 52 and 29. Since 52 > 29, swapping is done.
30, **29, 52**, 87, 63, 27, 19, 54
(c) Compare 52 and 87. Since 52 < 87, no swapping isdone.
(d) Compare 87 and 63. Since 87 > 63, swapping is done.
30, 29, 52, **63, 87**, 27, 19, 54
(e) Compare 87 and 27. Since 87 > 27, swapping is done.
30, 29, 52, 63, **27, 87**, 19, 54
(f) Compare 87 and 19. Since 87 > 19, swapping is done.
30, 29, 52, 63, 27, **19**, **87**, 54
(g) Compare 87 and 54. Since 87 > 54, swapping is done.
30, 29, 52, 63, 27, 19, **54, 87**

Observe that after the end of the first pass, the largest element is placed at the highest index of the array. All the other elements are still unsorted.
Pass 2:
(a) Compare 30 and 29. Since 30 > 29, swapping isdone.
29, 30, 52, 63, 27, 19, 54, 87
(b) Compare 30 and 52. Since 30 < 52, no swapping isdone.
(c) Compare 52 and 63. Since 52 < 63, no swapping isdone.
(d) Compare 63 and 27. Since 63 > 27, swapping is done.
29, 30, 52,**27, 63**, 19, 54, 87
(e) Compare 63 and 19. Since 63 > 19, swapping is done.
29, 30, 52, 27, **19, 63**, 54, 87
(f) Compare 63 and 54. Since 63 > 54, swapping is done.
29, 30, 52, 27, 19, **54, 63**, 87

Observe that after the end of the second pass, the second largest element is placed at the second highest index of the array. All the other elements are still unsorted.
Pass 3:
(a) Compare 29 and 30. Since 29 < 30, no swapping isdone.

(b) Compare 30 and 52. Since 30 < 52, no swapping isdone.
(c) Compare 52 and 27. Since 52 > 27, swapping is done.
29, 30, **27, 52**, 19, 54, 63, 87
(d) Compare 52 and 19. Since 52 > 19, swapping is done.
29, 30, 27**, 19, 52**, 54, 63, 87
(e) Compare 52 and 54. Since 52 < 54, no swapping isdone.

Observe that after the end of the third pass, the third largest element is placed at the third highest index of the array. All the other elements are still unsorted.

Pass 4:

(a) Compare 29 and 30. Since 29 < 30, no swapping isdone.
(b) Compare 30 and 27. Since 30 > 27, swapping is done.
29, **27, 30**, 19, 52, 54, 63, 87
(c) Compare 30 and 19. Since 30 > 19, swapping is done.
29, 27, **19, 30**, 52, 54, 63, 87
(d) Compare 30 and 52. Since 30 < 52, no swapping isdone.

Observe that after the end of the fourth pass, the fourth largest element is placed at the fourth highest index of the array. All the other elements are still unsorted.

Pass 5:

(a) Compare 29 and 27. Since 29 > 27, swapping isdone.
27, 29, 19, 30, 52, 54, 63, 87
(b) Compare 29 and 19. Since 29 > 19, swapping is done.
27, **19, 29**, 30, 52, 54, 63, 87
(c) Compare 29 and 30. Since 29 < 30, no swapping isdone.

Observe that after the end of the fifth pass, the fifth largest element is placed at the fifth highest index of the array. All the other elements are still unsorted.

Pass 6:

(a) Compare 27 and 19. Since 27 > 19, swapping isdone.
19, 27, 29, 30, 52, 54, 63, 87
(b) Compare 27 and 29. Since 27 < 29, no swapping isdone.

Observe that after the end of the sixth pass, the sixth largest element is placed at the sixth largest index of the array. All the other elements are still unsorted.

Pass 7:

(a) Compare 19 and 27. Since 19 < 27, no swapping isdone.

BUBBLE_SORT(A, N)

Step 1: Repeat Step 2 For 1 = 0 to N-1
Step 2: Repeat For J = 0 to N - I
Step 3: IF A[J] > A[J + 1]
SWAP A[J] and A[J+1]
[END OF INNER LOOP]
[END OF OUTER LOOP]
Step 4: EXIT

In this algorithm, the outer loop is for the total number of passes which is N–1. The inner loop will be executed for every pass. However, the frequency of the inner loop will decrease with every pass because after every pass, one element will be in its correct position. Therefore, for every pass, the inner loop will be executed N–I times, where N is the number of elements in the array and I is the count of thepass.

Complexity of Bubble Sort

The 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 the first pass, N–1 comparisons are made to place the highest element in its correct position. Then, in Pass 2, there are N–2 comparisons 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) + + 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.

INSERTION SORT

Insertion sort is a very simple sorting algorithm in which the sorted array (or list) is built one element at a time. We all are familiar with this technique of sorting, as we usually use it for ordering a deck of cards while playing bridge.

The main idea behind insertion sort is that it inserts each item into its proper place in the final list. To save memory, most implementations of the insertion sort algorithm work by moving the current data element past the already sorted values and repeatedly interchanging it with the preceding value until it is in its correct place. Insertion sort is less efficient as compared to other more advanced algorithms such as quick sort, heap sort, and mergesort.

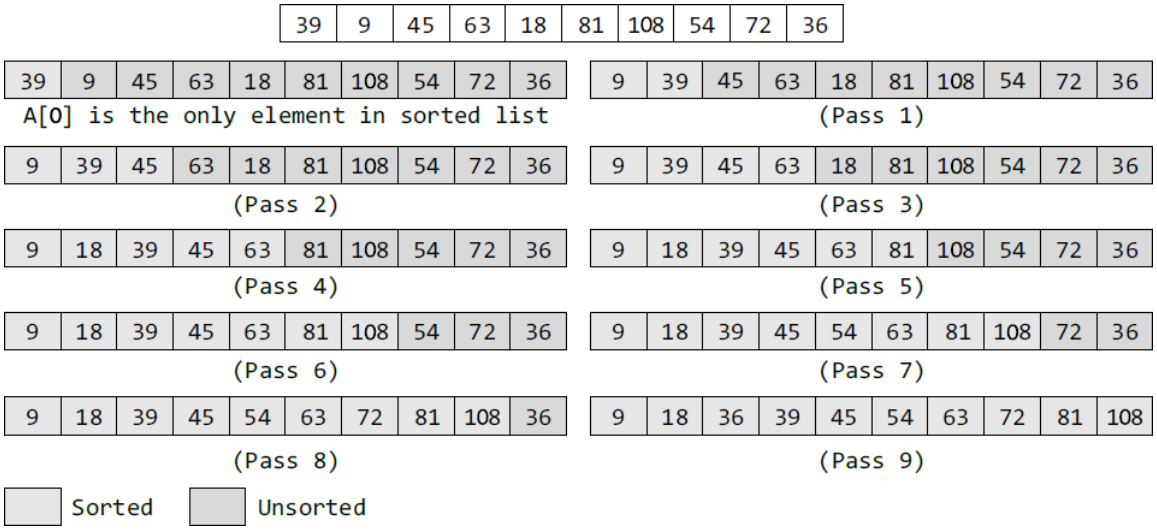
Technique

Insertion sort works as follows:

- The array of values to be sorted is divided into two sets. One that stores sorted values and another that contains unsortedvalues.
- The sorting algorithm will proceed until there are elements in the unsortedset.
- Suppose there are n elements in the array. Initially, the element with index 0 (assuming LB = 0) is in the sorted set. Rest of the elements are in the unsortedset.
- The first element of the unsorted partition has array index 1 (if LB =0).
- During each iteration of the algorithm, the first element in the unsorted set is picked up and inserted into the correct position in the sortedset.

Example

Consider an array of integers given below. We will sort the values in the array using insertion sort.



Initially, A[0] is the only element in the sorted set. In Pass 1, A[1] will be placed either before or after A[0], so that the array A is sorted. In Pass 2, A[2] will be placed either before A[0], in between A[0] and A[1], or after A[1]. In Pass 3, A[3] will be placed in its proper place. In Pass N–1, A[N–1] will be placed in its proper place to keep the arraysorted.

To insert an element A[K] in a sorted list A[0], A[1], ..., A[K–1], we need tocompare A[K] with A[K–1], then with A[K–2], A[K–3], and so on until we meet an element A[J] such that A[J] <= A[K]. In order to insert A[K] in its correct position, we need to move elements A[K–1], A[K–2], ..., A[J] by one position and then A[K] is inserted at the (J+1)th location.

INSERTION-SORT (ARR, N)
Step 1: Repeat Steps 2 to 5 for K = 1 to N – 1
Step 2: SET TEMP = ARR[K]
Step 3: SET J = K - 1
Step 4: Repeat while TEMP <= ARR[J]
SET ARR[J + 1] = ARR[J]
SET J = J - 1
[END OF INNER LOOP]
Step 5: SET ARR[J + 1] = TEMP
[END OF LOOP]
Step 6: EXIT

In the algorithm, Step 1 executes a for loop which will be repeated for each element in the array. In Step 2, we store the value of the Kth element in TEMP. In Step 3, we set the Jth index in the array. In Step 4, a for loop is executed that will create space for the new element from the unsorted list to be stored in the list of sorted elements. Finally, in Step 5, the element is stored at the (J+1)th location.

Complexity of Insertion Sort
For insertion sort, the best case occurs when the array is already sorted. In this case, the running time of the algorithm has a linear running time (i.e., O(n)). This is because, during each iteration, the first element from the unsorted set is compared only with the last element of the sorted set of the array. Similarly, the worst case of the insertion sort algorithm occurs when the array is sorted in the reverse order. In the worst case, the first element of the

unsorted set has to be compared with almost every element in the sorted set. Furthermore, every iteration of the inner loop will have to shift the elements of the sorted set of the array before inserting the next element. Therefore, in the worst case, insertion sort has a quadratic running time (i.e., $O(n^2)$). Even in the average case, the insertion sort algorithm will have to make at least (K–1)/2 comparisons. Thus, the average case also has a quadratic running time.

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 ofdata.
- It can be efficiently implemented on data sets that are already substantiallysorted.
- It performs better than algorithms like selection sort and bubble sort. Insertion sort algorithm is simpler than shell sort, with only a small trade-off in efficiency. It is over twice as fast as the bubble sort and almost 40 per cent faster than the selectionsort.
- it requires less memory space (only O(1) of additional memoryspace).
- It is said to be online, as it can sort a list as and when it receives newelements.

SELECTION SORT

Selection sort is a sorting algorithm that has a quadratic running time complexity of O(n2), thereby making it inefficient to be used on large lists. Although selection sort performs worse than insertion sort algorithm, it is noted for its simplicity and also has performance advantages over more complicated algorithms in certain situations. Selection sort is generally used for sorting files with very large objects (records) and smallkeys.

Technique

Example

We take the below depicted array for our example.

14, 33, 27, 10, 35, 19, 42, 44

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

Step 1:

14, 33, 27, 10, 35, 19, 42, 44

So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of sorted list.

10, 33, 27, 14, 35, 19, 42, 44

For the second position, where 33 is residing, we start scanning the rest of the list in linear manner.

Step 2:

10, 33, 27, 14, 35, 19, 42, 44

We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

10, 14, 27, 33, 35, 19, 42, 44

After two iterations, two least values are positioned at the the beginning in the sorted manner.

The same process is applied on the rest of the items in the array. We shall see an pictorial depiction of entire sorting process −

Step 3:
10, 14, 27, 33, 35, 19, 42, 44

Now replace 19 by 27

10, 14, 19, 33, 35, 27, 42, 44

Step 4:

10, 14, 19, 33, 35, 27, 42, 44

Now replace 27 by 33

10, 14, 19, 27, 35, 33, 42, 44

Step 5:

10, 14, 19, 27, 35, 33, 42, 44

Now replace 33 by 35

10, 14, 19, 27, 33, 35, 42, 44

Step 6:

10, 14, 19, 27, 33, 35, 42, 44

Its already at its position hence no need to replace
Step 7:

10, 14, 19, 27, 33, 35, 42, 44

Its already at its position hence no need to replace

Step 8:

10, 14, 19, 27, 33, 35, 42, 44

Its already at its position hence no need to replace

Finally we have sorted data:

10, 14, 19, 27, 33, 35, 42, 44

Algorithm:

```
for(j = 0; j < n-1; j++)
{
    intiMin = j;
    for( i = j+1; i< n; i++)
    {
      if(a[i] < a[iMin])
      {
         iMin = i;
      }
   }

   if(iMin != j)
   {
      swap(a[j], a[iMin]);
   }
}
```

RADIX SORT

Radix sort is a linear sorting algorithm for integers and uses the concept of sorting names in alphabetical order. When we have a list of sorted names, the *radix* is 26 (or 26 buckets) because there are 26 letters in the English alphabet. So radix sort is also known as bucketsort. Observe that words are first sorted according to the first letter of the name. That is, 26 classes are used to arrange the names, where the first class stores the names that begin with A, the second class contains the names with B, and soon.

During the second pass, names are grouped according to the second letter. After the second pass, names are sorted on the first two letters. This process is continued till the nth pass, where n is the length of the name with maximum number ofletters.

While sorting the numbers, we have ten buckets, each for one digit (0, 1, 2, …, 9) and the number of passes will depend on the length of the number having maximum number of digts.

Algorithm

Algorithm for RadixSort (ARR, N)
Step 1: Find the largest number in ARR as LARGE
Step 2: [INITIALIZE] SET NOP = Number of digits in LARGE
Step 3: SET PASS =0
Step 4: Repeat Step 5 while PASS <= NOP-1
Step 5: SET I =0 and INITIALIZE buckets
Step 6: Repeat Steps 7 to 9 while I<N-1
Step 7: SET DIGIT = digit at PASSth place inA[I]
Step 8: Add A[I] to the bucket numbered DIGIT
Step 9: INCEREMENT bucket count for bucket numbered DIGIT
[END OFLOOP]
Step 1 : Collect the numbers in the bucket
[END OFLOOP]
Step 11: END

Example

Sort the numbers given below using radix sort.
345, 654, 924, 123, 567, 472, 555, 808, 911
In the first pass, the numbers are sorted according to the digit at ones place. The buckets are pictured upside down as shown below.

Solution

Number	0	1	2	3	4	5	6	7	8	9
345						345				
654					654					
924					924					
123				123						
567								567		
472			472							
555						555				
808									808	
911		911								

After this pass, the numbers are collected bucket by bucket. The new list thus formed is used as an input for the next pass. In the second pass, the numbers are sorted according to the digit at the tens place. The buckets are pictured upside down.

Number	0	1	2	3	4	5	6	7	8	9
911		911								
472								472		
123			123							
654						654				
924			924							
345					345					
555						555				
567							567			
808	808									

In the third pass, the numbers are sorted according to the digit at the hundreds place. The buckets are pictured upside down.

Number	0	1	2	3	4	5	6	7	8	9
808									808	
911										911
123		123								
924										924
345				345						
654							654			
555						555				
567						567				
472					472					

The numbers are collected bucket by bucket. The new list thus formed is the final sorted result.
After the third pass, the list can be given as
123, 345, 472, 555, 567, 654, 808, 911, 924.

Complexity of Radix Sort

To calculate the complexity of radix sort algorithm, assume that there are n numbers that have to be sorted 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) asymptotictime.

Pros and Cons of Radix Sort

Radix sort is a very simple algorithm. When programmed properly, radix sort is one of the fastest sorting algorithms for numbers or strings of letters.
But there are certain trade-offs for radix sort that can make it less preferable as compared to other sorting algorithms. Radix sort takes more space than other sorting algorithms. Besides the array of numbers, we need 10 buckets to sort numbers, 26 buckets to sort strings containing only characters, and at least 40 buckets to sort a string containing alphanumeric characters.

Another drawback of radix sort is that the algorithm is dependent on digits or letters. This feature compromises with the flexibility to sort input of any data type. For every different data type, the algorithm has to be rewritten. Even if the sorting order changes, the algorithm has to be rewritten. Thus, radix sort takes more time to write and writing a general purpose radix sort algorithm that can handle all kinds of data is not a trivial task. Radix sort is a good choice for many programs that need a fast sort, but there are faster sorting algorithms available. This is the main reason why radix sort is not as widely used as other sorting algorithms.