**5. SEARCHING AND SORTING**

**Sorting:** Sorting is the operation to arranging the records of a table in to some sequential order(Ascending or Descending).

* The sort is performed according to the key value of each record. Depending on key, records sorted either numerically or alphanumerically.

**Classification of Sorting method:**

* **Internal sort:**
  1. Bubble sort
  2. Quick sort(Partition Exchange sort)
  3. Selection sort
  4. Heap sort
  5. Insertion sort
  6. Shell sort
* **External sort:**

1. Merge sort
2. Radix sort

**Bubble Sort**

* Bubble sort is easy to understand and simple sorting technique. That’s why it is the most widely known sorting method. But this sorting technique is not efficient in comparison to other sorting technique.
* Thefundamental thing in this technique is, for ascending order sorting during every pass the elements in the list comparing the next element and moves the top elements to the top of list.
* In bubble sort method during the first pass element 1 and 2 are compared and if they are out of order then they are interchanged. This process is repeated for elements 2 and 3 and so on.
* After the end of first pass the record with the largest value is placed at nth(last) position.
* During second pass the record with next largest value is placed at n-1 position and so on.
* After each pass we check for the interchange. If no interchanges occurred then the array must be sorted and no further pass is required.
* Consider the following example:

**BUBBLE\_SORT (K,N)**

* These function sorts elements of array K consist of N elements.
* PASS denotes the pass index.
* LAST denotes the last unsorted element.
* EXCHS counts the total number of exchange made during pass
  1. [Initialize]

LAST🡨N

* 1. [Loop on pass index]

Repeat thru step 5 for PASS = 1, 2… N-1

* 1. [Initialize exchange counter for this pass ]

EXCHS🡨0

* 1. [Perform pair wise comparisons on unsorted elements ]

Repeat for I = 1, 2… LAST-1

If K [I] > K [I+1] then

K[I] K[I+1]

EXCHS🡨EXCHS+1

* 1. [Exchange made on this pass?]

If EXCHS = 0 then

Write “Array already sorted”

Else

LAST🡨LAST-1

* 1. [Finished]

Return (Minimum number of pass required)

**Performance:**

* During first pass we required n-1 comparisons. During second pass we required n-2 comparisons and during ith pass we required n-i comparisons. So total number of comparisons is given as:

n-1

∑ (n-i) = n (n-1)/2

i=1

* Thus the order of comparisons is proportional to n2 i.e O(n2)

**Quick sort OR Partition Exchange sort**

* In this method we pick the first element from the array and place it in it’s proper position in the array such that all the elements preceding the element having smaller value and all the elements following the element having larger value.
* Thus the table is divided into two sub tables.
* The same process is then applied to each of the sub tables and repeated until all records are placed in their final position.
* Consider the following example:

**QUICK\_SORT(K,LB,UB)**

* These function sorts elements of array K consist of N elements.
* LB and UB denotes the lower and upper bound of the sub table being processed.
* FLAG is a logical variable.
* KEY is the value which is placed in its final position during pass.
  1. [Initialize]

FLAG🡨true

* 1. [Perform Sort]

If LB < UB then

I🡨LB

J🡨UB+1

KEY🡨K [LB]

Repeat while FLAG

I🡨I+1

Repeat while K [I] < KEY

I🡨I+1

J🡨J-1

Repeat while K [J] > KEY

J🡨J-1

If I < J then

K[I]K[J]

Else

FLAG🡨false

K[LB]K[J]

Call QUICK\_SORT (K, LB, J-1)

Call QUICK\_SORT (K, J+1, UB)

* 1. [Finished]

**Selection Sort**

* In this method selection sort starts from first element and searches the entire array until it find smallest element. Then smallest value interchanges with the first element.
* Now select second element and searches for the second smallest element from the array, if found then interchange with second element.
* So in this method after pass 1 smallest value arranged at first position then after pass 2 second minimum will arrange at second position and so on.
* This process continues until all the elements in the array are arranged in ascending order.
* Consider the following example:

**SELECTION\_SORT(K,N)**

* These function sorts elements of array K consist of N elements.
* PASS denotes the pass index and the position of the first element in the vector which is to be examined during particular pass.
* MIN\_INDEX denotes the position of the smallest element encountered thus far in a particular pass.
  1. [Loop on pass index]

Repeat thru step 4 for PASS = 1, 2, 3… N-1

* 1. [initialize minimum index]

MIN\_INDEX 🡨PASS

* 1. [Make a pass and obtain element with the smallest value]

Repeat for I = PASS+1 to N

If K [I] < K [MIN\_INDEX] then

MIN\_INDEX🡨I

* 1. [Exchange elements]

If MIN\_INDEX ≠ PASS then

K[PASS]K[MIN\_INDEX]

* 1. [finished]

Return

**Performance:**

* During first pass we required n-1 comparisons. During second pass we required n-2 comparisons and during ith pass we required n-i comparisons. So total number of comparisons is given as:

n-1

∑ (n-i) = n(n-1)/2

i=1

* Thus the order of comparisons is proportional to n2 i.e O(n2)

**Insertion Sort**

**INSERTIONSORT(K, N)**

* L is an array consist of N elements
  1. [Initialize]

K[0]🡨-0

* 1. [Loop on Array]

Repeat thru step 5 for I = 1 to N

* 1. [Set temporary variable and pointer]

TEMP🡨K[I]

J🡨I-1

* 1. [Compare elements]

While (TEMP< K[J])

K[J+ 1] 🡨 K[J]

J 🡨 J -1

* 1. [Insert element to its proper place]

K[J] 🡨 TEMP

* 1. [Finished]

**Shell Sort**

**SHELL\_SORT(K,N)**

* K is an array consist on N elements
  1. [Initialize Gap]

Gap 🡨 N / 2

* 1. Repeat thru step 6 while (Gap = Gap/2)
  2. [Initialize Swap variable]

FLAG🡨0

* 1. [Repeat through step 6 while(Swap)]
  2. [Repeat through step 6 for I = 0 to I < (N - Gap)]
  3. if K[I] > K[I + Gap] then

Temp🡨K [I]

K [I] 🡨K [I + Gap]

K [I + Gap] 🡨Temp

FLAG🡨1

* 1. [Finished]

**Merge Sort**

* Simple merge sort is used to merge two ordered list in a single ordered list.
* Suppose we have to sorted table with name table1 and table 2.
* In simple merge sort we compare the elements of both table and the element which has the smallest value is placed in a new table.
* This process is repeated until all the elements from both the table are placed in a new table.
* Consider the following example:

Table 1: 11 23 42

Table 2: 9 25

* First we compare the elements of both the table and the element which has the smallest value is placed in a new table.
* So the trace is as follow:

Table 1: 11 23 42

Table 2: 25

New Table: 9

Table 1: 23 42

Table 2: 25

New Table: 9 11

Table 1: 42

Table 2: 25

New Table: 9 11 23

Table 1: 42

Table 2:

New Table: 9 11 23 25

Table 1:

Table 2:

New Table: 9 11 23 25 42

* We store the two sorted table in a common vector. Thus two sorted tables in a common vector can be stored as follow:

11 23 42 9 25

First Second Third

* Here First to second-1 represents the first table and second to third represent second table.
* We required temporary vector of the same size to hold the result of sorting
* We can also merge k ordered list into single sorted list. Such a merging process is known as multiple merging or k- way merging.

**SIMPLE\_MERGE(K,FIRST,SECOND,THIRD)**

* This function merges two sorted table into single sorted table.
* Here FIRST to SECOND-1 represents the first table and SECOND to THIRD represent second table
* TEMP is the temporary vector.

**Step 1:** [Initialize]

I🡨FIRST

J🡨SECOND

L🡨0

**Step 2:** [Compare corresponding elements and output the smallest]

Repeat while I<SECOND and J ≤ THIRD

If K [I] ≤ K [J] then

L🡨L+1

TEMP [L] 🡨K [I]

I🡨I+1

Else

L🡨L+1

TEMP [L] 🡨K [J]

J🡨J+1

**Step 3:** [Copy the remaining unprocessed elements in output area]

If I ≥ SECOND then

Repeat while J ≤ THIRD

L🡨L+1

TEMP [L] 🡨K [J]

J🡨J+1

Else

Repeat while I < SECOND

L🡨L+1

TEMP [L] 🡨K [J]

I🡨I+1

**Step 4:** [Copy elements in temporary vector into original vector]

Repeat for I = 1 to L

K [FIRST-1+I] 🡨TEMP [I]

**Step 5:** [Finished]

Return

**Radix Sort**

* In this method we use ten pockets for the digits 0 to 9.
* Consider the following set of data:

42 23 74 11 65 57 94 36 99 87 70 81 61

* During first pass we separate the unit digit of the number and place the number in to appropriate pocket according to its unit digit.
* For example the first number is 42 so we separate the unit digit of the number 42 which is 2 so we place the number in pocket 2. Same procedure is repeated for remaining numbers.
* After first pass the numbers in each pockets are as follow:

61

81 94 87

70 11 42 23 74 65 36 57 99

**Pocket: 0 1 2 3 4 5 6 7 8 9**

* Now arrange the number according to their pocket. The numbers after first pass are as follows:

70 11 81 61 42 23 74 94 65 36 57 87 99

* During second pass we separate the next higher digit of the number and place the number in to appropriate pocket according to its digit.
* After second pass the numbers in each pockets are as follow:

65 74 87 99

11 23 36 42 57 61 70 81 94

**Pocket: 0 1 2 3 4 5 6 7 8 9**

* Now arrange the number according to their pocket. The numbers after second pass are as follows:

11 23 36 42 57 61 65 70 74 81 87 94 99

* The same process is repeated until all the elements are sorted.

**Sequential Search or Linear Search**

* This method is used to find out element in an unordered list.
* Suppose the list L consist of N elements and we want to find the element from the list.
* In this technique first the value of the element is compared with the value of the first element in the list, if match is found then the search is terminated.
* If match is not found then next element from the list is fetched and compared with the element.
* This process is repeated until match is found or all the element in the list is compared.
* Consider the following example:
* The list L consist of 5 elements as shown below:

|  |  |
| --- | --- |
| **Index** | **Element** |
| 1 | 22 |
| 2 | 33 |
| 3 | 05 |
| 4 | 66 |
| 5 | 44 |

* Suppose we want to find 05 in the list. So 05 is compared with the first element in the list which is 22.

|  |  |
| --- | --- |
| **Index** | **Element** |
| 1 | 22 |
| 2 | 33 |
| 3 | 05 |
| 4 | 66 |
| 5 | 44 |

05 Match not found.

* Since match is not found next element is fetched from the list and compared with 05 and this process is repeated until match found or all elements in the list is compared as shown below.

|  |  |
| --- | --- |
| **Index** | **Element** |
| 1 | 22 |
| 2 | 33 |
| 3 | 05 |
| 4 | 66 |
| 5 | 44 |

05 Match not found

|  |  |
| --- | --- |
| **Index** | **Element** |
| 1 | 22 |
| 2 | 33 |
| 3 | 05 |
| 4 | 66 |
| 5 | 44 |

05 Match Found

**SEQENTIAL\_SEARCH(K, N, X)**

* These function searches the list K consist of N elements for the value of X.
  1. [Initialize search]

I🡨1

K [N+1] 🡨X

* 1. [Search the Vector]

Repeat while K [I] ≠ X

I🡨I+1

* 1. [Successful Search?]

If I = N+1 then

Write “Unsuccessful Search”

Return 0

Else

Write “Successful Search”

Return I

**Binary Search**

* This method is used to find out element in an ordered list.
* It is very efficient method to search element.
* Let low represents lower limit of the list and high represents the higher limit of the list.
* First we calculate the value of middle as:

Mid = (Low + High)/2

Low Middle High

1 2 3 4 5 6 7 8 9 10

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **75** | **151** | **203** | **275** | **318** | **489** | **524** | **591** | **647** | **727** |

* Now we compare the value of the middle element with the element to be searched.
* If the value of the middle element is greater then the element to be searched then the element will exist in lower half of the list. So take high = mid – 1 and find value of the middle in this new interval.
* If the value of the middle element is smaller then the element to be searched then the element will exist in upper half of the list. So take low = mid + 1 and find value of the middle in this new interval.
* This process is repeated until entire list is searched or the element is found.
* Suppose we want to find the element 275 in the above list.
* First, mid = (low + high)/2

= (1+10)/2

= 5

* Here the value of middle element is 318 which is greater then 275 so the element is in the lower half of the list.

Take high = mid – 1 = 5-1=4

* Now mid = (low + high)/2

= (1 + 4)/2

= 2

* Here the value of the middle element is 151 which is smaller then 275 so the element is in the upper half of the list.

Take low = mid + 1 = 2+1 = 3

* Now mid = (low + high)/2

= (3+4)/2

= 3

* Here the value of the middle element is 203 which is smaller then 275 so the element is in the upper half of the list.

Take low = mid + 1 = 3+1 = 4

* Now mid = (low + high)/2

= (4+4)/2

= 4

* Here the value of the middle element is 275 which we want to find.

**BINARY\_SEARCH(K, N, X)**

* These function searches the list K consist of N elements for the value of X.
* LOW, HIGH and MIDDLE denotes the lower, upper and middle limit of the list.
  1. [Initialize]

LOW🡨1

HIGH🡨N

* 1. [Perform Search]

Repeat thru step 4 while LOW ≤ HIGH

* 1. [Obtain Index of midpoint interval]

MIDDLE 🡨[(LOW + HIGH)/2]

* 1. [Compare]

If X < K [MIDDLE] then

HIGH 🡨 MIDDLE – 1

Else if X > K [MIDDLE] then

LOW🡨MIDDLE+1

Else

Write “Successful Search”

Return (MIDDLE)

* 1. [Unsuccessful Search]

Write “Unsuccessful Search”

* 1. Return 0

**Compare Sequential Search with Binary Search**

|  |  |
| --- | --- |
| **Sequential Search** | **Binary Search** |
| (1) Sequential Search is used to find elements from unordered list | (1) Binary Search is used to find elements from ordered list. |
| (2) This technique is less efficient | (2) This technique is more efficient |
| (3) The order of sequential search is O(N). | (3) The order of Binary Search is O(Log2N) |

**Hashing Functions**

* Hashing provides direct access of records from the file no matter where the record is in the file.
* This technique uses the hashing function say H which maps the Key to the particular address of the record.
* Thus hashing function provides key to address transformation.
* A hashing function H maps the Key space K into an address space.
* The hashing function generates an address or location by performing some arithmetic or logical operation on given key.
* Example: suppose there are 60 students in the classroom and we want to maintain their record. We have to store the record in an array. In this case the array index ranges from 0-59. We may use this index as the registration number of student. For example if a student has a registration number S15280531 then the record is stored at address 31.
* Hashing function is of two types:
  + 1. Distribution dependent hashing function
    2. Distribution independent hashing function
* Some of the most widely used hashing functions are described below:
  + 1. Division Method
    2. The Mid square Method
    3. The Folding Method
    4. Digit Analysis
    5. The Length Dependent Method
    6. Algebraic Coding
    7. Multiplicative Hashing
* **(1) Division Method:** one of the hashing function uses the division method, which is defined as:

H(x) = x mod m + 1

For example: H (35) = 35 mod 11 + 1 = 2 + 1 = 3

The Division method generates a key value which belongs to the set {1, 2… m}

* **(2) Mid square Method:** In this method a key is multiplied by itself and the address is obtained by selecting an appropriate number of digits from the middle of the square

For example: consider a six digit key 123456. Squaring this key result in the value 15241383936. if a three digit address is required then position 5 to 7 could be chosen which gives 138.

* **(3) Folding Method:** in this method a key is partitioned into parts. The length of each part is similar to the length of the address required. These parts are added together and final carry is ignored to produce the address.

For example if a key 35678943 is transformed into 2 digit address then we have:

35 + 67 + 89 + 43 = 234 = 34.

This method is known as **fold shifting** method.

Another method is **fold boundary** method in which if a key 35678943 is transformed into 2 digit address then we have:

53 + 76 + 98 + 34 = 261 = 61.

**Collision resolution techniques.**

* **Collision Resolution:** When a hashing function maps several keys to same address space then it is known as collision.
* **For Example:** Suppose we are storing the records in an array which ranges from 0 to 99.if a hashing function is H(k) = k % 100 then this function will produces same address for the keys 15433 and 26733. in this case collision is encountered.
* The techniques used to resolve this collision is known as collision resolution technique.
* Some popular collision resolution techniques are as follows:
  1. Linier Probing
  2. Rehashing
  3. Quadratic Probing
  4. Chaining
* **(1) Linier Probing:** If a record with key x is mapped to address location d and that location is already occupied by another key then other locations in the table are examined until a free memory location is found.
* **For Example:**

Suppose we have to insert the following key values with hashing function H(k) = k % 100

50904, 78907, 68403, 86704, 72308

|  |  |
| --- | --- |
| **Index** | **Key** |
| 00 | Empty |
| 01 | Empty |
| 02 | Empty |
| 03 | 68403 |
| 04 | 50904 |
| 05 | 86704 |
| 06 | Empty |
| 07 | 78907 |
| 08 | 72308 |

* The array must be circular in order to search an empty location.
* **(2) Rehashing:** If a hash function results in a collision then we use the secondary hash function to calculate the address. This is known as rehashing and this procedure is repeated until an empty location is found.
* In above example the hash function H (k) = k % 100 produces collision for the key 86704 so we use secondary hash function as: H (k) = (k + constant) % 100. Let constant = 1. Then this secondary function produces the address (86704+1) % 100 = 05. This location is empty so we put the key 86704 at location 05.
* **(3)Quadratic Probing**

If there is a collision at hash address h, this method probes the table at locations h+1, h+4, h+9…., that is h + i2 (mod hash size). That is the increment function is i2. This method substantially reduces clustering.

* **(4)Key dependent increment**

Rather than having the increment depend on the number of probes already made, we can let it

Be some simple function of the key itself. For example, we could truncate the key to a single

Character and use its code as the increment.

A good approach, when the remainder after division is taken as the hash function, is to let the increment depend on the quotient of the same division.

* **(5)Random probing**

A final method is to use a pseudorandom number generator to obtain the increment. The generator used should be one that always generates the same sequence provided it starts with

The same need. This method is excellent in avoiding clustering, but is likely to be slower than

The others.