UNIT V SEARCHING, SORTING AND HASHING TECHNIQUES

Information retrieval in the required format is the central activity in all computer applications. This involves searching, sorting and merging. This block deals with all three, but in this block we will be concerned with searching techniques.

Searching methods are designed to take advantage of the file organisation and optimize the search for a particular record or to establish its absence. The file organisation and searching method chosen can make a substantial difference to an application's performance.

We will now discuss two searching methods and anlyze their performance. These two methods are:

- The sequential search

- The binary search

Sorting technique:

Retrieval of information is made easier when it is stored in some predefined order. Sorting is, therefore, a very important computer application activity. Many sorting algorithms are available. Differing environments require differing sorting methods. Sorting algorithms can be characterized in the following two ways:

1. simple algorithms which require the order of n2 (written as O(n2)comparisons to sort n items.
2. Sophisticated algorithms that require the O(nlog2n) comparisons to sort n items.

The difference lies in the fact that the first method moves data only over small distances in the process of sorting, whereas the second method moves data over large distances, so that items settle into the proper order sooner, thus resulting in fewer comparisons. Performance of a

sorting algorithm can also depend on the degree of order already present in the data.

There are two basic categories of sorting methods: Internal Sorting and External Sorting. Internal soriting are applied when the entire collection of data to sorted is small enough that the sorting can take place within main memory. The time required to read or write is not considered to be significant in evaluating the performance of internal sorting methods.

External sorting methods are applied to larger collection of data which reside on secondary devices read and write access time are major concern in determine sort performances.

In this unit we will study some methods of internal sorting. The next unit will discuss methods of external sorting.

Internal sorting

In internal sorting, all the data to be sorted is available in the high speed main memory the computer. We will study the following methods of internal sorting:

1. Insertion sort

2. Bubble sort

3. Quick sort

4- Way Merge sort

5. Heap sort

Insertion Sort

This is a naturally occuring sorting method exemplified by a card player arranging the cards dealt to him. He picks up the cards as they are dealt and inserts them into the required position. Thus at every step, we insert an item into its proper place in an already ordered list.

We will illustrate insertion sort with an example before presenting the formal algorithm.

Example 1: Sort the following list using the insertion sort method:

Thus to find the correct position search the list till an item just greater than the target is found. Shift all the items from this point one, down the list. Insert the target in the vacated slot.

We now present the algorithm for insertion sort.

ALGORITHM: INSERT SORT

INPUT: LIST[ ] of N items in random order.

OUTPUT: LIST[ ] of N items in sorted order.

1 BEGIN,

2. FOR I = 2 TO N DO

3. BEGIN

4. F LIST[I] LIST[I-1]

5. THEN BEGIN

6. J = I

7. T = LIST[I] /*STORE LIST[I]*/

8. REPEAT /* MOVE OTHER ITEMS DOWN THE LIST*/

9: J = J-1

10. LIST [J + 1] =LIST [J];

11. IFJ = 1THEN

12. FOUND =TRUE

13. UNTIL (FOUND = TRUE)

14. LIST [I] = T

15. END

16. END

17. END

QUICK SORT

This is the most widely used internal sorting algorithm. In its basic form, it was invented by C.A.R. Hoare in 1960. Its popularity lies in the- ease of implementation, moderate use of resources and acceptable behaviour for a variety of sorting cases. The basis of quick sort is the 'divide' and conquer' strategy i.e. Divide the problem [list to be sorted] into sub-problems [sub-lists], until solved sub problems [sorted sub-lists] are found. This is implemented as

Choose one item A[I] from the list A[ ].

Rearrange the list so that this item is in the proper position i.e. all preceding items have a lesser value and all succeeding items have a greater value than this item.

1. A[0], A[1] .. A[I-1] in sub list 1

2. A[I]

3. A[I + 1], A[I + 2] ... A[N] in sublist 2

Repeat steps 1 & 2 for sublist & sublist2 till A[ ] is a sorted list.

As can be seen, this algorithm has a recursive structure.,

Step 2 or the 'divide' procedure is of utmost importance in this algorithm.

This is usually implemented as follows:

1. Choose A[I] the dividing element.

2. From the left end of the list (A[O] onwards) scan till an item A[R] is found whose value is greater than A[I].

3. From the right end of list [A[N] backwards] scan till an item A[L] is found whose Value is less than A[1].

4. Swap A[-R] & A[L].

5. Continue steps 2, 3 & 4 till the scan pointers cross. Stop at this stage.

6. At this point sublist 1 & sublist2 are ready.

7. Now do the same for each of sublist 1 & sublist2.

We will now give the implementation of Quicksort and illustrate it by an example. Quicksort (int A[], int X, int 1)

{

int L, R, V 1.

1. If (IX)

{

2. V = A[1], L = X-1, R = I; 3.

3. For (;;)

4. While (A[+ + L] V);

5. While (A[- -R] V);

6. If (L = R) /* left & right ptrs. have crossed */

7. break;

8. Swap (A, L, R) /* Swap A[L] & A[R] */ }

9. Swap (A, L, I)

10. Quicksort (A, X, L-1)

11. Quicksort (A, L + 1, I) } }

Quick sort is called with A, I, N to sort the whole file.

MERGE SORT

Merge sort is also one of the 'divide and conquer' class of algorithms. The basic idea into this is to divide the list into a number of sub lists, sort each of these sub lists and merge them to get a single sorted list. The recursive implementation of 2- way merge sort divides the fist into 2 sorts the sub lists and then merges them to get the sorted list. The illustrative implementation of 2 way merge sort sees the input initially as n lists of size 1. These are merged to get n/2 lists of size 2. These n/2 lists are merged pair wise and so on till a single list is obtained. This can be better understood by the following example. This is also called CONCATENATE

SORT

We give here the recursive implementation of 2 Way Merge Sort.

```
Mergesort (int List [], int low, int high)

{

    int mid,

1.  Mid = (low + high)/2;

2.  Mergesort (LIST, low, mid);

3.  Mergesort (LIST, mid + 1, high);

4.  Merge (low, mid, high, List, FINAL)

}

Merge (int low, int mid, int high, int LIST[], int FINAL)

{

int a, b, c, d;

a = low, b = low, c = mid + 1

While (a < = mid and c < = high) do

{
```

```
If LIST [a] < = LIST [c] then

{

FINAL [b] =LIST [a]

a = a+1

}

else

{

FINAL [b] = LIST [c]

c = c + l

}

b = b+1

 }

If (a mid) then

For d = c to high do

{

B[b] = LIST [d]

b = b + 1

}

Else

For d = a to mid do

{

B[b] = A[d]
```

```
b = b + l;

}

}
```

To sort the entire list, Mergesort should be called with LIST,1, N.

Mergesort is the best method for sorting linked lists in random order. The total computing time is of the 0(n log2n).

The disadvantage of using mergesort is that it requires two arrays of the same size and type for the merge phase. That is, to sort and list of size n, it needs space for 2n elements.

SEQUENTIAL SEARCH:

This is the most natural searching method. Simply put it means to go through a list or a file till the required record is found. It makes no demands on the ordering of records. The algorithm for a sequential search procedure is now presented.

ALGORITHM: SEQUENTIAL SEARCH

This represents the algorithm to search a list of values of to find the required one.

INPUT: List of size N. Target value T

OUTPUT: Position of T in the list - I

BEGIN

Set FOUND to false

Set I to O

If FOUND is false.

While (I < = N) and (FOUND is false)

T is not present in List.

END

This algorithm can easily be extended for searching for a record with a matching key value.

Linear Search

The most obvious algorithm is to start at the beginning and walk to the end, testing for a match at each item:

```
bool jw_search ( int *list, int size, int key, int*& rec )
{
  // Basic sequential search
  bool found = false;
  int i;

  for ( i = 0; i < size; i++ ) {
    if ( key == list[i] )
      break;
  }
  if ( i < size ) {
    found = true;
    rec = &list[i];
  }

  return found;
}
```

This algorithm has the benefit of simplicity; it is difficult to get wrong, unlike other more sophisticated solutions. The above code follows the convention of this article, they are as follows:

1. All search routines return a true/false boolean value for success or failure.
2. The list will be either an array of integers or a linked list of integers with a key.
3. The found item will be saved in a reference to a pointer for use in client code.

The algorithm itself is simple. A familiar 0 - n-1 loop to walk over every item in the array, with a test to see if the current item in the list matches the search key. The loop can terminate in one of two ways. If i

reaches the end of the list, the loop condition fails. If the current item in the list matches the key, the loop is terminated early with a break statement. Then the algorithm tests the index variable to see if it is less that size (thus the loop was terminated early and the item was found), or not (and the item was not found).

For a linked list defined as:

```
struct node {
  int rec;
  int key;
  node *next;

  node ( int r, int k, node *n )
    : rec ( r )
    , key ( k )
    , next ( n )
  {}
};
```

The algorithm is equally simple:

```
bool jw_search ( node*& list, int key, int*& rec )
{
  // Basic sequential search
  bool found = false;
  node *i;

  for ( i = list; i != 0; i = i->next ) {
    if ( key == i->key )
      break;
  }
  if ( i != 0 ) {
    found = true;
    rec = &i->rec;
  }

  return found;
}
```

Instead of a counting loop, we use an idiom for walking a linked list. The idiom should be familiar to most readers. For those that are not familiar with it, that is how it is done. :-) The loop terminates if i is a null pointer (the algorithm assumes a null pointer terminates the list) or if the item was found.

The basic sequential search algorithm can be improved in a number of ways. One of those ways is to assume that the item being searched for will always be in the list. This way you can avoid the two termination conditions in the loop in favor of only one. Of course, that creates the problem of a failed search. If we assume that the item will always be found, how can we test for failure?

The answer is to use a list that is larger in size than the number of items by one. A list with ten items would be allocated a size of eleven for use by the algorithm. The concept is much like C-style strings and the nul terminator. The nul character has no practical use except as a dummy item delimiting the end of the string. When the algorithm starts, we can simply place the search key in list[size] to ensure that it will always be found:

```
bool jw_search ( int *list, int size, int key, int*& rec )
{
  // Quick sequential search
  bool found = false;
  int i;

  list[size] = key;
  for ( i = 0; key != list[i]; i++ )
    ;
  if ( i < size ) {
    found = true;
    rec = &list[i];
  }

  return found;
}
```

Notice that the only test in the traversal loop is testing for a match. We know that the item is in the list somewhere, so there's no need for a loop body. After the loop the algorithm simply tests if i is less than size. If it is then we have found a real match, otherwise i is equal to size. Because list[size] is where the dummy item was, we can safely say that the item does not exist anywhere else in the list. This algorithm is faster because it reduces two tests in the loop to one test. It isn't a big improvement, but if jw_search is called often on large lists, the optimization may become noticeable.

Another variation of sequential search assumes that the list is ordered (in ascending sorted order for the algorithm we will use):

```
bool jw_search ( int *list, int size, int key, int*& rec )
{
  // Ordered sequential search
  bool found = false;
  int i;

  for ( i = 0; i < size && key > list[i]; i++ )
    ;
  if ( key == list[i] ) {
    found = true;
    rec = &list[i];
  }

  return found;
}
```

Binary Search

All of the sequential search algorithms have the same problem; they walk over the entire list. Some of our improvements work to minimize the cost of traversing the whole data set, but those improvements only cover up what is really a problem with the algorithm. By thinking of the data in a different way, we can make speed improvements that are much better than anything sequential search can guarantee.

Consider a list in ascending sorted order. It would work to search from the beginning until an item is found or the end is reached, but it makes more sense to remove as much of the working data set as possible so that the item is found more quickly. If we started at the middle of the list we could determine which half the item is in (because the list is sorted). This effectively divides the working range in half with a single test. By repeating the procedure, the result is a highly efficient search algorithm called binary search.

The actual algorithm is surprisingly tricky to implement considering the apparent simplicity of the concept. Here is a correct function that implements binary search by marking the current lower and upper bounds for the working range:

```
bool jw_search ( int *list, int size, int key, int*& rec )
{
  // Binary search
  bool found = false;
  int low = 0, high = size - 1;

  while ( high >= low ) {
    int mid = ( low + high ) / 2;
    if ( key < list[mid] )
      high = mid - 1;
    else if ( key > list[mid] )
      low = mid + 1;
    else {
      found = true;
      rec = &list[mid];
      break;
    }
  }

  return found;
}
```

No explanation will be given for this code. Readers are expected to trace its execution on paper and with a test program to fully understand its elegance. Binary search is very efficient, but it can be improved by

writing a variation that searches more like humans do. Consider how you would search for a name in the phonebook. I know of nobody who would start in the middle if they are searching for a name that begins with B. They would begin at the most likely location and then use that location as a gauge for the next most likely location. Such a search is called interpolation search because it estimates the position of the item being searched for based on the upper and lower bounds of the range. The algorithm itself isn't terribly difficult, but it does seem that way with the range calculation:

```
bool jw_search ( int *list, int size, int key, int*& rec )
{
  // Interpolation search
  bool found = false;
  int low = 0, high = size - 1;

  while ( list[high] >= key && key > list[low] ) {
    double low_diff = (double)key - list[low];
    double range_diff = (double)list[high] - list[low];
    double count_diff = (double)high - low;
    int range = (int)( low_diff / range_diff * count_diff + low );
    if ( key > list[range] )
      low = range + 1;
    else if ( key < list[range] )
      high = range - 1;
    else
      low = range;
  }
  if ( key == list[low] ) {
    found = true;
    rec = &list[low];
  }

  return found;
}
```