1

SORTING AND SEARCHING TECHNIQUES

Unit Structure:

1.1 Sorting
1.2 Searching
1.3 Analysis of Algorithm
1.4 Complexity of Algorithm
1.5 Asymptotic Notations for Complexity Algorithms

1.1 SORTING

Sorting and searching are the most important techniques used in the computation. When the history of computing might be defined ‘searching‘ and ‘sorting’ would have been at top. They are the most common ingredients of programming systems. The sorting techniques are classified as internal and external sorting. Here we are going to discuss the different sorting techniques such as bubble sort, selection sort, Insertion, Shell sorts and Sequential, Binary, Indexed Sequential Searches, Interpolation, Binary Search ,Tree Sort, Heap sort, Radix sort.

1.1.1 Insertion Sort

It is one of the simplest sorting algorithms. It consists of N-1 passes. For pass P= 1 through N – 1, insertion sort ensures that the elements in positions 0 through P are sorted order. Insertion sort makes use of the fact that elements in positions 0 through P – 1 are already known to be in sorted order as shown in following table 1.1

Original	**34**	**8**	**64**	**51**	**32**	**21**	**Positions Moved**
P = 1	8	34	64	51	32	21	1
P = 2	8	34	64	51	32	21	0
P = 3	8	34	51	64	32	21	1
P = 4	8	32	34	51	64	21	3
P = 5	8	21	32	34	51	64	4

Table 1.1 Insertion sort after each pass

```
void insertionSort( int a[], int n ) {

/* Pre-condition: a contains n items to be sorted */

   int  i,j, p;

  /* Initially, the first item is considered 'sorted' */
   /* i divides a into a sorted region, x<i, and an
      unsorted one, x >= i */

   for(i=1;i<n;i++)
   {   /* Select the item at the beginning of the
          as yet unsorted section */

      p = a[i];
       /* Work backwards through the array, finding where v
          should go */
       j = i;
       /* If this element is greater than v,
            move it up one */
       while ( a[j-1] > p )
        {
         a[j] = a[j-1];
         j = j-1;
         if( j <= 0 ) break;
        }
       /* Stopped when a[j-1] <= p, so put p at position j */
       a[j] = p;
       }
   }
```

Figure 1.1 Function for insertion sort

Analysis of Insertion Sort

Due to the nested loops each of which can take N iterations, insertion sort is $O(N^2)$. In case if the input is pre-sorted, the running time is O(N), because the test in the inner loop always fails immediately. Indeed, if the input is almost sorted , insertion sort will run quickly.

1.1.2 Bubble Sort

This is common technique used to sort the elements. The bubble sort makes n – 1 passes through a sequence of n elements. Each pass moves through the array from left to right, comparing adjacent elements and swapping each pair that is out of order. This gradually moves the heaviest element to the right. It is called the bubble sort because if the elements are visualized in a vertical column, then each pass appears to 'bubble up' the next heaviest element by bouncing it off to smaller elements.

```
/* Bubble sort for integers */
#define SWAP(a,b)
 { int t; t=a; a=b; b=t; }

void bubble( int a[], int n )
/* Pre-condition: a contains n items to be sorted */
   {    int i, j;
   /* Make n passes through the array */
   for(i=0;i<n;i++)
     {        /* From the first element to the end of the unsorted
section */

      for(j=1;j<(n-i);j++)

        {  /* If adjacent items are out of order, swap them */
           if( a[j-1]>a[j] ) SWAP(a[j-1],a[j]);
         }
      }
   }
```

Figure 1.2 Bubble Sort

Analysis

Each of these algorithms requires ***n***-1 passes: each pass places one item in its correct place. (The n^{th} is then in the correct place also.) The i^{th} pass makes either ***i*** or ***n - i*** comparisons and moves.
The nth item also moves in its correct place. So:

$$T(n) = 1 + 2 + 3 + ... + (n-1)$$
$$= \sum_{i=1}^{n-1} i$$
$$= \frac{n}{2}(n-1)$$

or **O(n^2)** - but we already know we can use heaps to get an **O(*n* log*n*)** algorithm. Thus these algorithms are only suitable for small problems where their simple code makes them faster compared to the more complex code of the **O(*n* log*n*)** algorithm.

1.1.3 The Selection Sort

The selection sort is similar to the bubble sort. It makes the **n -1** passes through a sequence of n elements, each time moving the largest of the remaining unsorted elements into its correct position. This is more efficient than the Bubble sort because it doesn't move any element in the process of finding the largest element. It makes only one swap on each pass after it has found the largest. It is called the selection sort because on each pass it selects the largest of the remaining unsorted elements and puts it in its correct position.

```
void sort(int a[])
{  //Post condition: a[0] <= a[1] <= ... <= a[a.length -1] ;

   for(int i = n; i >0; i--)
{ int m=0;
   for(int j=0;j<=i;j++)
{   if(a[j] > a[m])
 {  m = j;}}

// Invariant: a[m] >= a[j] for all j<=i;
Swap(a,i,m)
// Invariants: a[j] <= a[i] for all j<= i;
                a[i] <= a[i+1] <= .... <= n -1;
}}
```

Figure 1.3 Selection sort

Analysis

The Selection sort runs in O(n^2) time complexity. Though the Selection sort and Bubble sort have the same time complexity, selection sort is comparatively faster.

1.1.4 Shell Sort

This technique is invented by Donald shell. Compare to the Insertion sort, Shell sort uses the increment sequence, h1,h2,.....,ht. Any increment sequence will do as long as h1=1. After a phase,

using some increment hk, for every i, we have A[i] ≤ A[i + hk]. The file is then said to as hk-sorted.

Original	**81**	**94**	**11**	**96**	**12**	**35**	**17**	**95**	**28**	**58**	**41**	**75**	**15**
After5- sort	35	17	11	28	12	41	75	15	96	58	81	94	95
After 3-sort	28	12	11	35	15	41	58	17	94	75	81	96	95
After 1-sort	11	12	15	17	28	35	41	58	75	81	94	95	96

Table 1.2 : Shell Sort flow

An important property of Shell Sort is that an hk-sorted file that is then hk-1 sorted remains hk-sorted. If this was not the case, the algorithm would be of little value, since work done by early phases would be undone by the later phases.

The general strategy to hk-sort is for each position, i, in hk, hk+1,.....,N – 1, place the element in the correct spot among i, i – hk , i - 2hk, etc. Although it doesnot affect the implementation, a careful examination shows that the action of an hk-sort is to perform an insertion sort on hk independent subarrays. A popular choice for increment sequence is to use the sequence suggested by shell:

h1= [N/2] and hk = [hk+1 / 2].

```
void Shellsort(int a[] , int N)
{
   int i,j, increment;
    int temp;
  for(increment = N/2 ; increment > 0; increment / = 2)
     for(i = increment; i < N ; i++)
   {
         temp = a[i];
          for(j = i ; jb>= increment ; j -= increment)
           if(temp < a[j - increment ])
             a[j] = a[j – increment];
          else
             break;
         a[j] = temp;
   }
}
```

Figure 1.4 Shell sort

Analysis
The Shell Sort runs in $O(n^{1.5})$ time.

1.1.5 Radix Sort

Bin Sort : The Bin sorting approach can be generalised in a technique that is known as *Radix sorting*.

An example

Assume that we have **n** integers in the range (**0,n^2**) to be sorted. (For a bin sort, **m** = **n**^{2}, and we would have an **O(n+m)** = **O(n^2)** algorithm.) Sort them in two phases:

1. Using **n** bins, place $\mathbf{a_i}$ into bin $\mathbf{a_i}$ mod **n**,
2. Repeat the process using **n** bins, placing $\mathbf{a_i}$ into bin floor($\mathbf{a_i}$/**n**), *being careful to append to the end of each bin*.

This results in a sorted list.

As an example, consider the list of integers:
36 9 0 25 1 49 64 16 81 4
n is 10 and the numbers all lie in (0,99). After the first phase, we will have:

Bin	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Content	0	181	-	-	644	25	3616	-	-	949

Table 1.3: Bins and contents

Note that in this phase, we placed each item in a bin indexed by the least significant decimal digit.

Repeating the process, will produce:

Bin	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Content	0149	16	25	36	49	-	64	-	81	-

Table 1.4: After reprocess

In this second phase, we used the leading decimal digit to allocate items to bins, being careful to add each item to the end of the bin.

We can apply this process to numbers of any size expressed to any suitable base or *radix*.

Generalised Radix Sorting

We can further observe that it's not necessary to use the same radix in each phase, suppose that the sorting key is a sequence of fields, each with bounded ranges, *eg* the key is a date using the structure:

```
typedef struct t_date {
   int day;
   int month;
   int year;
} date;
```

If the ranges for ***day*** and ***month*** are limited in the obvious way, and the range for ***year*** is suitably constrained, *eg* 1900 < **year** <= 2000, then we can apply the same procedure except that we'll employ a different number of bins in each phase. In all cases, we'll sort first using the least significant "digit" (where "digit" here means a field with a limited range), then using the next significant "digit", placing each item after all the items already in the bin, and so on.

Assume that the key of the item to be sorted has **k** fields, $\mathbf{f_i}|\mathbf{i}=0..\mathbf{k-1}$, and that each $\mathbf{f_i}$ has $\mathbf{s_i}$ discrete values, then a generalised radix sort procedure can be written:

radixsort(A, n) {
  for(i=0;i<k;i++) {
  for(j=0;j<s_i;j++) bin[j] = EMPTY;
	O(s_i)

for(j=0;j<n;j++) {
  move A_i
  to the end of bin[A_i->f_i]
  }
	O(n)

for(j=0;j<s_i;j++)
  concatenate bin[j] onto the end of A;
  }
}
	O(s_i)

Total	$\sum_{i=1}^{k} O(s_i + n) = O\left(kn + \sum_{i=1}^{k} s_i\right)$ $= O\left(n + \sum_{i=1}^{k} s_i\right)$

Figure 1.5 Radix sort

Now if, for example, the keys are integers in (0,$\mathbf{b^k}$-1), for some constant **k**, then the keys can be viewed as **k**-digit base-**b** integers. Thus, $\mathbf{s_i = b}$ for all **i** and the time complexity becomes **O(n+kb)** or **O(n)**. *This result depends on **k** being constant.*

If **k** is allowed to increase with **n**, then we have a different picture. For example, it takes $\log_2\mathbf{n}$ binary digits to represent an integer <**n**. If the key length were allowed to increase with **n**, so that **k** = log**n**, then we would have:

$$\sum_{i=1}^{k} O(s_i + n) = O\left(n \log n + \sum_{i=1}^{\log n} 2 \right)$$
$$= O(n \log n + 2\log n)$$
$$= O(n \log n)$$

Another way of looking at this is to note that if the range of the key is restricted to (0,$\mathbf{b^k}$-1), then we will be able to use the radixsort approach effectively if we allow duplicate keys when $\mathbf{n}>\mathbf{b^k}$. However, if we need to have unique keys, then **k** must increase to at least $\log_b\mathbf{n}$. Thus, as **n** increases, we need to have log**n** phases, each taking **O(n)** time, and the radix sort is the same as quick sort!

1.1.6 The Heap Sort

A heap is by definition partially sorted, because each linear string from root to leaf is sorted. This leads to an efficient general sorting algorithm called the heap sort. Heap sort uses the auxillary function ***Sort()***. I has the complexity function ***O(n log n)*** which is same as merge and quick sort. The heap sort essentially loads n elements into a heap and then unloads them.

```
void sort()
{ // Post condition: a[0] <= a[1] <= ... < n-1;
   for(int i=n/2 - 1; i>= 0 ; i-- )
{  heapify(a,i,n);  }
for(int i = n – 1; i > 0; i--)
{ swap(a,0,i);
  heapify(a,0,i);
}
}
```

Figure 1.6 sort method for heap

```
void heapify(int a[], int i, int j)
{
     int a1 = a[i];
     while(2*i+1 < j)
 {    int k = 2*i + 1;
    if(k+1 < j && a[k+1] > a[k])
  {     ++k;  // a[k] is larger child
   }
 if(ai >= a[k])
{ break; }
a[i] = a[k];
i = k;
}  a[i] = ai;
 }
```

Figure 1.7 heapify method

The ***sort()*** function first converts the array so that its underlying complete binary tree is transformed into a heap. This is done by applying the heapify() function to each nontrivial subtree. The nontrivial subtrees are the subtrees that are rooted above the leaf level. In the array, the leaves are stored at **e** positions **a[n/2]** through **a[n].** So the first for loop in the sort() function applies the heapify() function to elements **a[n/2 - 1]** back through **a[0].** The result is an array whose corresponding tree has the heap property.
The main for loop progresses through **n-1** iterations, Each iteration does two things: it swaps the root element with element **a[i]** and then it applies the heapify() function to the subtree of elements **a[0..i].** That subtree consists of the part of the array that is still unsorted. Before swap executes on each iteration, the subarray **a[0..i]** has the heap property, so a[i] is the largest element in that subarray. That means that the swap() puts element a[i] in its correct position.

0	1	2	3	4	5	6	7	8
99	66	88	44	33	55	77	22	4

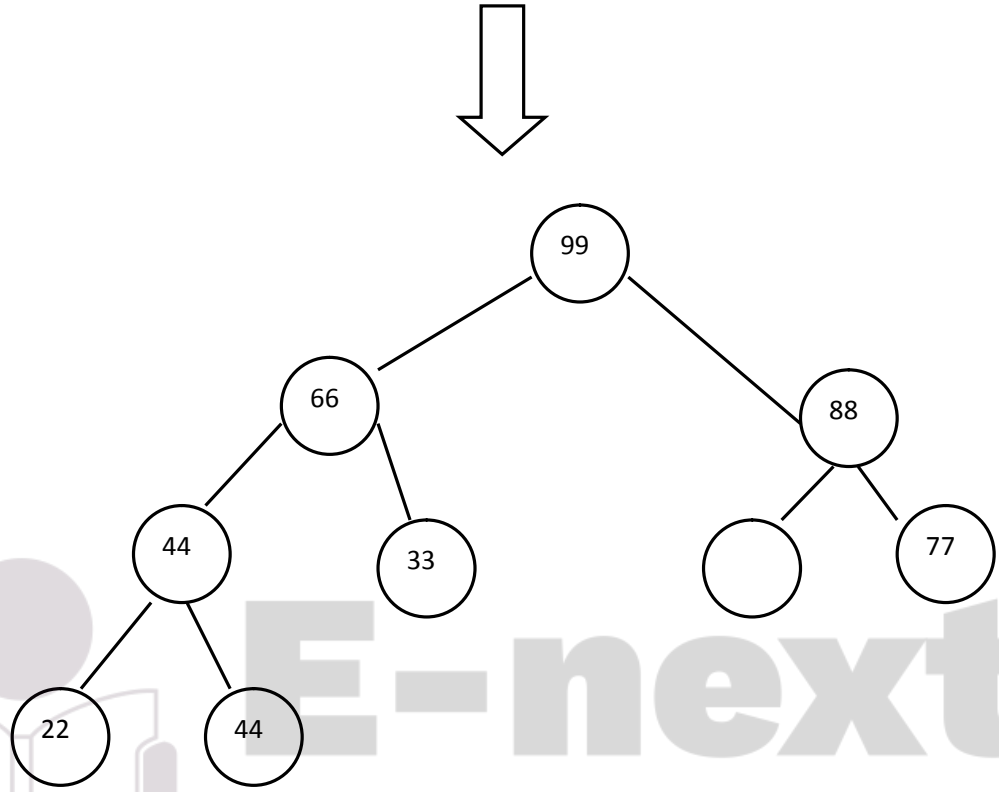

Figure 1.8 Implementation of heapify

1.2 SEARCHING

Searching technique is used to find a record from the set of records. The algorithm used for searching can return the entire record or it may return a pointer to that record. And it may also possible that the record may not exist in the given data set.

1.2.1Sequential Searching

It is the simplest form of search. This search is applicable to a table organized as an array or as a linked list. Let us assume that k is an array of n keys, k(0) through k(n-1), and r an array of records, r(0) through r(n-1), such that k(i) is the key of r(i). Also assume that key is a search argument. And we wish to return the smallest integer i such that k(i) equals key if such an i exists and -1 otherwise the algorithm for doing this is as follows

```
for(i = 0 ; i < n; i++)
    if(key == k(i))
      return(i);
return(-1);
```

As shown above the algorithm examines each key ; upon finding one that matches the search argument, its index is returned. If no match is found, -1 is returned.

The above algorithm can be modified easily to add record *rec* with the key *key* to the table if *key* is not already there.

```
k(n) = key;  /* insert the new key and record */
r(n) = rec;
n++;            /* increase the table size */
return(n - 1);
```

Compare to this more efficient search method involves inserting the argument key at the end of the array before beginning the search, thus guaranteeing that the key will be found.

```
k(n) = key;
for(i=0;key != k(i); i++)
if(i<n)
 return(i);
else
 return(-1);
```

for search and insertion, the entire if statement is replaced by

```
 if(i == n)
  r(n++) = rec;
return(i);
```

The extra key inserted at the end of the array is called a sentinel.

Efficiency of Sequential Searching

Through the example let us examine the number of comparisons made by a sequential search in searching for a given key. Let's assume no insertion or deletions exist, so that we are searching through a table of constant size n. The number of comparisons depends on where the record with the argument key appears in the table. If the record is the first one in the table, only one comparison is required; if the record is the last one in the table, n comparisons are required. If it is equally likely for the argument to

appear at any given table position, a successful search will take (on the average) (n+1) / 2 comparisons, and an unsuccessful search will take n comparisons. In any case, the number of comparisons is ***o(n)***.

1.2.2The Indexed Sequential Search

This is another technique which improves efficiency for a sorted file, but simultaneously it consumes a large amount of large space. As shown in the given figure an auxiliary table is called an ***index***, which is a set aside in addition to the sorted file itself. Each element in the index consists of a key ***kindex*** and a pointer to the record in the file that corresponds to *kindex*. The elements in the index, as well as the elements in the file, must be sorted on the key. If the index is **1/8th** the size of the file, every **8th** record of the file is represented in the index.

The execution of the algorithm is simple. Let r, k and key be defined as before, let *kindex* be an array of the keys in the index, and let *pindex* be the array of pointers within the index to the actual records in the file. Assume that the file is stored as an array **n** is the size of the file, and that indxsze is the size of the index.


```
for(i = 0; i < indxsize && kindex(i) <= key; i++)
lowlim = (i = 0) ? 0 : pindex(i-1);
hlim = (i == indxsize) ? n – 1:pindex(i) – 1;
for(j=lowlim ; j <= hilim && k(j) != key; j++)
return((j > hilim) ? -1 : j);
```

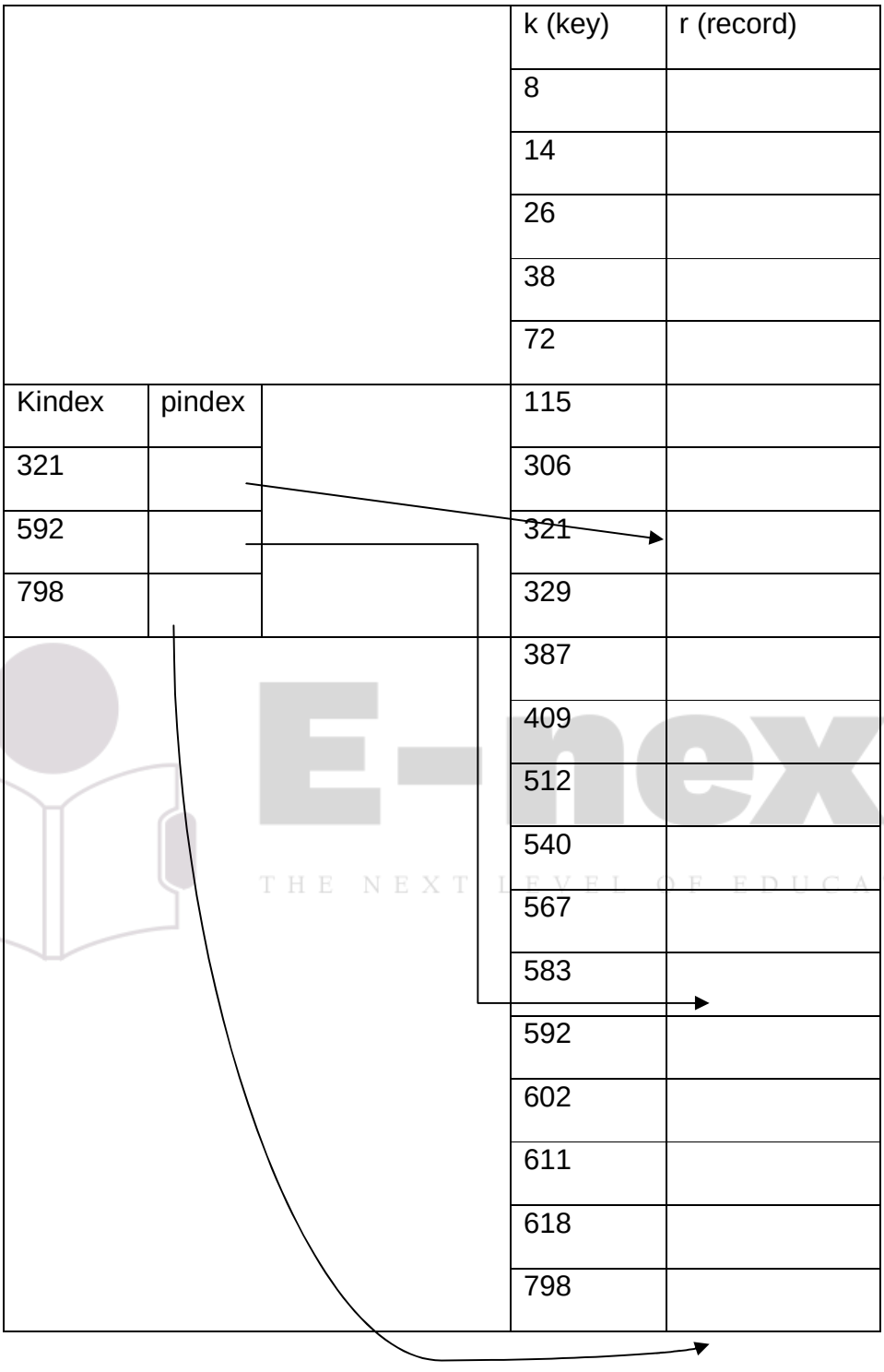

Figure 1.9 Indexing

The main advantage of indexed sequential method is that the items in the table can be examined sequentially if all the records in the file must be accessed, yet the search time for a particular item is sharply reduced. A sequential search is performed on the smaller index rather than on the larger table. And once the

correct index position has been found a second sequential search is performed on a small portion of the record table itself. Index is used for the stored array. If the table i so large that even the use of an index does not achieve sufficient efficiency, a secondary index can be used. The secondary index acts as an index to the primary index, which points to entries in the sequential table

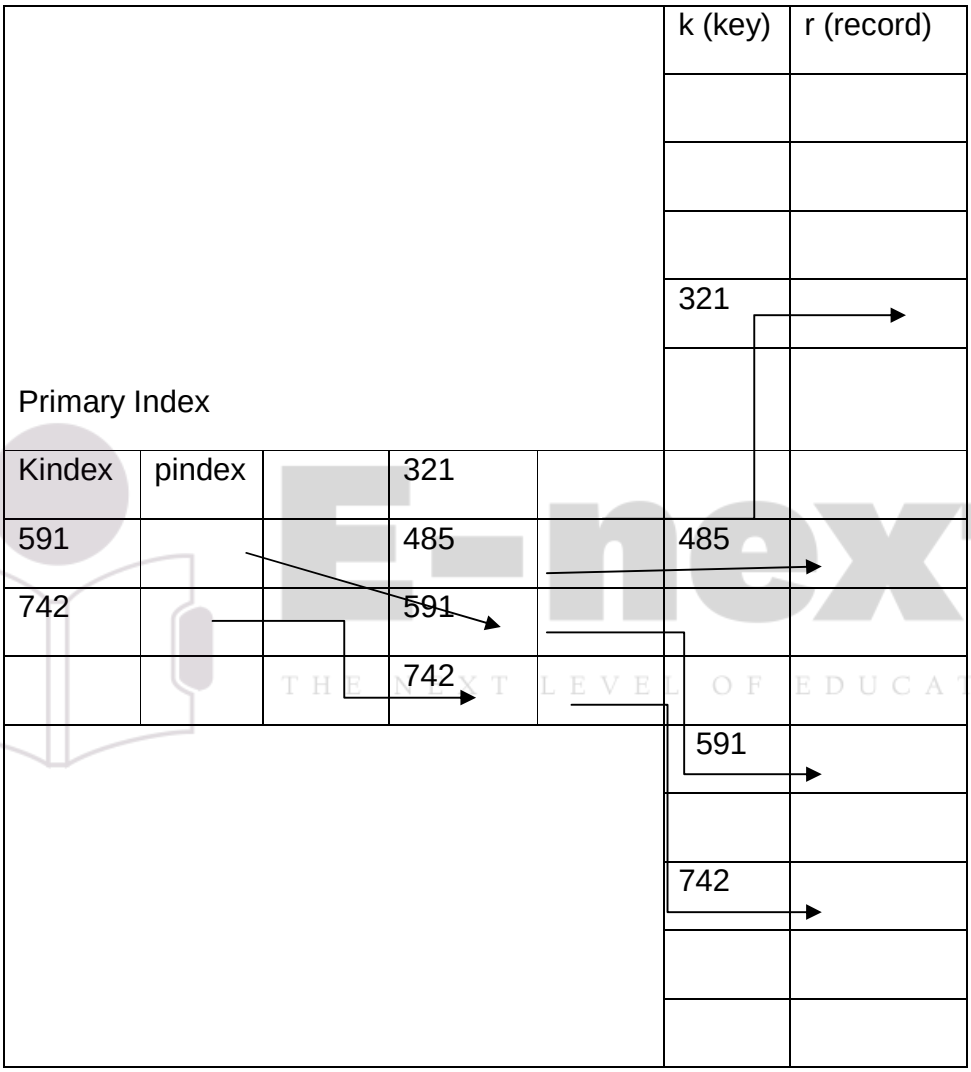

Figure 1.10 Indexed sequential

Deletions from an indexed sequential table can be made most easily by flagging deleted entries. In sequential searching through the table, deleted entries are ignored. Also note that if an element is deleted, even if its key is in the index, nothing need be done to the index, only the original table entry is flagged. Insertion into an sequential table is more difficult, since there may not be room between two already existing table entries.

1.2.3 Binary Search

This is the fastest approach of search. The concept is to look at the element in the middle. If the key is equal to that element, the search is finished. If the key is less than the middle element, do a binary search on the first half. If it's greater, do a binary search of the second half.

Working

The advantage of a binary search over a linear search is astounding for large numbers. For an array of a million elements, binary search, **O(log N),** will find the target element with a worst case of only 20 comparisons. Linear search, O(N), on average will take 500,000 comparisons to find the element. The only requirement for searching is that the array elements should be sorted.

```
int binarySearch(int sortedArray[], int first, int last, int key) {
  // function:
  //   Searches sortedArray[first]..sortedArray[last] for key.
  // returns: index of the matching element if it finds key,
  //         otherwise  -(index where it could be inserted)-1.
  // parameters:
  //   sortedArray in  array of sorted (ascending) values.
  //   first, last in  lower and upper subscript bounds
  //   key         in  value to search for.
  // returns:
  //    index of key, or -insertion_position -1 if key is not in the
array.       //  This value can easily be transformed into the
position to insert it.

  while (first <= last) {
     int mid = (first + last) / 2;  // compute mid point.
     if (key > sortedArray[mid])
        first = mid + 1;  // repeat search in top half.
     else if (key < sortedArray[mid])
        last = mid - 1; // repeat search in bottom half.
     else        return mid;     // found it. return position /////
  }
  return -(first + 1);    // failed to find key }
```

Figure 1.11 binary search flow

This is the most efficient method of searching a sequential table without the use of auxiliary indices or tables. This technique is further elaborated in next chapter.

1.3 ANALYSIS OF ALGORITHM

Algorithm : An algorithm is a systematic list of steps for solving a particular problem.

Analysis of algorithm is a major task in computer science. In order to compare algorithms, we must have some criteria to measure the efficiency of our algorithms. Suppose M is an algorithm and *n* is the size of the input data. The time and space used by the algorithm M are two main measures for the efficiency of M. The time is measured by counting the number of key operations in sorting and searching algorithms, i.e the number of comparisons.

1.4 COMPLEXITY OF ALGORITHMS

The *complexity* of an algorithm M is the function f(n) which gives the running time and / or storage space requirement of the algorithm in terms of the size n of the input data. Frequently ,the storage space required by an algorithm is simply a multiple of the data size n.

1.5 ASYMPTOTIC NOTATIONS FOR COMPLEXITY ALGORITHMS

The "big O" notation defines an upper bound function g(n) for f(n) which represents the time / space complexity of the algorithm on an input characteristic n. There are other asymptotic notations such as Ω, Θ, *o* which also serve to provide bounds for the function f(n).

1.5.1 Omega Notation (Ω)

The omega notation (Ω) is used when the function g(n) defines the lower bound for the function f(n).

Definition: f(n) = Ω(g(n)), iff there exists a positive integer n0 and a positive number M such that $|f(n)| \geq M|g(n)|$, for all $n \geq n0$.

For f(n) = 18n + 9, f(n) > 18n for all n, hence f(n) = Ω(n). Also, for $f(n) = 90n^2 + 18n + 6$, $f(n) > 90n^2$ for $n \geq 0$ and therefore $f(n) = \Omega(n^2)$.

For f(n) = Ω(g(n)), g(n) is a lower bound function and there may be several such functions, but it is appropriate that is the function which is almost as large a function of n is possible such that the definition of Ω is satisfied and is chosen as g(n).

1.5.2 Theta Notation (Θ)

The theta notation is used when the function f(n) is bounded both from above and below by the function g(n).

Definition: f(n) = **Θ** (g(n)), iff there exists a positive constants c1 and c2, and a positive integer n0 such that c1|g(n)| $\leq$ |f(n)| $\leq$ c2|g(n)|, for all n$\geq$n0.

From the definition it implies that the function g(n) is both an upper bound and a lower bound for the function f(n) for all values of n, n$\geq$n0. In other words,f(n) is such that, f(n)=O(g(n)) and f(n) = Ω(g(n))

For f(n) = 18n + 9, since f(n) > 18n and f(n) $\leq$ 27n for n $\geq$ 1, we have f(n) = Ω(n) and therefore f(n) = O(n) respectively, for n $\geq$ 1. Hence f(n) = **Θ** (n).

1.5.3 Small Oh Notation(*o*)

Definition: f(n)=o(g(n)) iff f(n)=O(g(n)) and f(n)≠ Ω(g(n)). For f(n)=18n+9, we have f(n)=O(n^2) but f(n)≠ Ω(n^2).Hence f(n) = o(n^2). However f(n) ≠ o(n).

Exercise:

1. Explain the insertion sort. Discuss its time and space complexity.

2. Sort the sequence 3,1,4,1,5,9,2,6,5 using insertion sort.

3. Explain Bubble sort with its analysis.

4. Explain the algorithm for shell sort.

5. Show the result of running Shellsort on the input 9,8,7,6,5,4,3,2,1 using increments {1,3,7}.

6. Write a program to implement selection algorithm.

7.Show how heapsort processes the input 142,543,123,65,453,879,572,434,111,242,811,102.

8.Using the radix sort, determine the running of radix sort.

9. Explain the sequential searching technique.

10. Explain the indexed sequential searching technique.

11. Give the comparison between sequential and indexed sequential.

12. Explain how binary searching is implemented. And also state its limitations.

13. What is time complexity?

14. Calculate the time complexity for binary searching.

15. Explain the following notations: a) big ‘Oh’ b) small ‘oh’ c) Theta (Θ) d) Omega Notation (Ω).

❖❖❖❖