Unit 5
SORTING, SEARCHING AND HASH TECHNIQUES

SORTING

Ordering the data in an increasing or decreasing fashion according to some relationship among the data item is called sorting.

two main classifications of sorting

a. Internal sorting
b. External sorting

external sorting

External sorting is a process of sorting in which large blocks of data stored in storage Devices are moved to the main memory and then sorted.

internal sorting?

Internal sorting is a process of sorting the data in the main memory.

the various factors to be considered in deciding a sorting algorithm

a. Programming time
b. Execution time of the program
c. Memory needed for program environment

Insertion sort

One of the simplest sorting algorithms is the insertion sort. Insertion sort consists of n - 1 passes. For pass p = 2 through n, insertion sort ensures that the elements in positions 1 through p are in sorted order. Insertion sort makes use of the fact that elements in positions 1 through p - 1 are already known to be in sorted order. Figure shows a sample file after each pass of insertion sort. Figure shows the general strategy. In pass p, we move the pth element left until its correct place is found among the first p elements. The code in Figure implements this strategy. The sentinel in a[0] terminates the while loop in the event that in some pass an element is moved all the way to the front. Lines 3 through 6 implement that data movement without the explicit use of swaps. The element in position p is saved in tmp, and all larger elements (prior to position p) are moved one spot to the right. Then tmp is placed in the correct spot.

```
void
insertion_sort( input_type a[ ], unsigned int n )
{
unsigned int j, p;
input_type tmp;
a[0] = MIN_DATA; /* sentinel */
 for( p=2; p <= n; p++ )
{
 tmp = a[p];
 for( j = p; tmp < a[j-1]; j-- )
a[j] = a[j-1];

a[j] = tmp;
}
```

}

Original 34 8 64 51 32 21 Positions Moved

After p = 2 8 34 64 51 32 21 1
After p = 3 8 34 64 51 32 21 0
After p = 4 8 34 51 64 32 21 1
After p = 5 8 32 34 51 64 21 3
After p = 6 8 21 32 34 51 64 4

Implementation of insertion sort:

```
#include <stdio.h>
 int main()
{
 int n, array[1000], c, d, t;

 printf("Enter number of elements\n");
 scanf("%d", &n);

 printf("Enter %d integers\n", n);

 for (c = 0; c < n; c++) {
  scanf("%d", &array[c]);
 }

 for (c = 1 ; c <= n - 1; c++) {
  d = c;

  while ( d > 0 && array[d] < array[d-1]) {
    t          = array[d];
    array[d]   = array[d-1];
    array[d-1] = t;

    d--;
  }
 }

 printf("Sorted list in ascending order:\n");

 for (c = 0; c <= n - 1; c++) {
```

```
   printf("%d\n", array[c]);
  }

  return 0;
}
```

shell sort.

Routine for Shell sort

```
shellsort( input_type a[ ], unsigned int n )
{
unsigned int i, j, increment;
input_type tmp;
for( increment = n/2; increment > 0; increment /= 2 )
 for( i = increment+1; i<=n; i++ )
{
 tmp = a[i];
 for( j = i; j > increment; j -= increment )
 if( tmp < a[j-increment] )
a[j] = a[j-increment];
else
break;
 a[j] = tmp;
}
}
```

Example:

Original 81 94 11 93 12 35 17 95 28 58 41 75 15

After 5-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

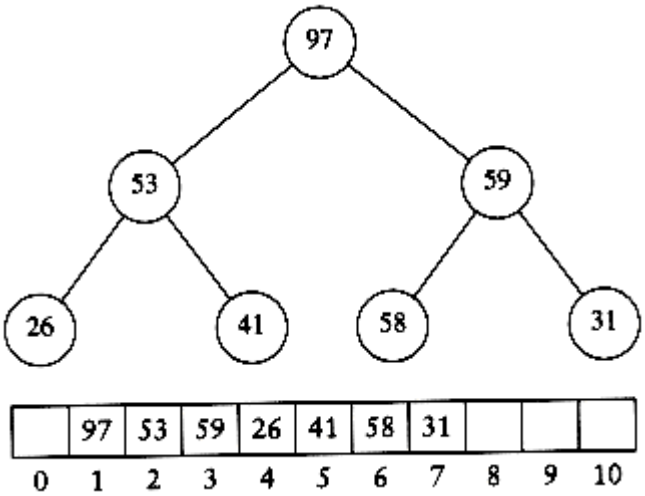
heap sort

Routine for Heap sort

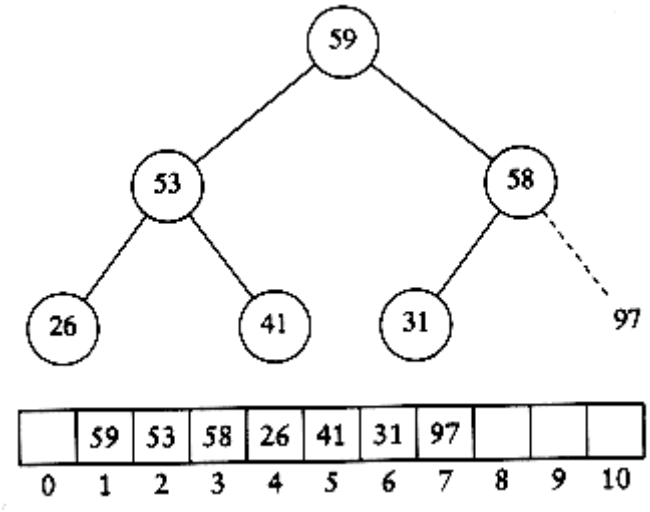
```
Void  heapsort( input_type a[], unsigned int n )
{
int i;
 for( i=n/2; i>0; i-- ) /* build_heap */
perc_down (a, i, n );
for( i=n; i>=2; i-- )
{
swap( &a[1], &a[i] ); /* delete_max */
```

```
perc_down( a, 1, i-1 );
}
}
void
perc_down( input_type a[], unsigned int i, unsigned int n )
{
unsigned int child;
input_type tmp;
for( tmp=a[i]; i*2<=n; i=child )
{
 child = i*2;
if( ( child != n ) && ( a[child+1] > a[child] ) )
child++;
if( tmp < a[child] )
a[i] = a[child];


else
break;
}
 a[i] = tmp;
}
```



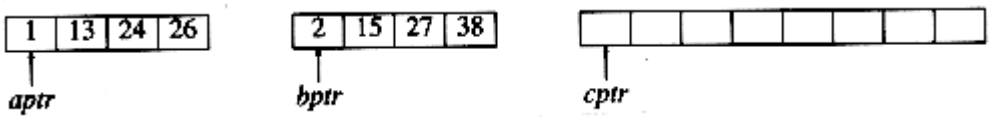
(Max) heap after build_heap phase



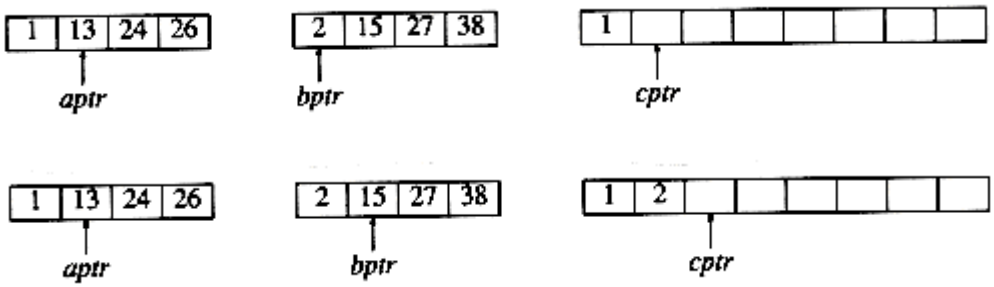
Heap after first delete_max

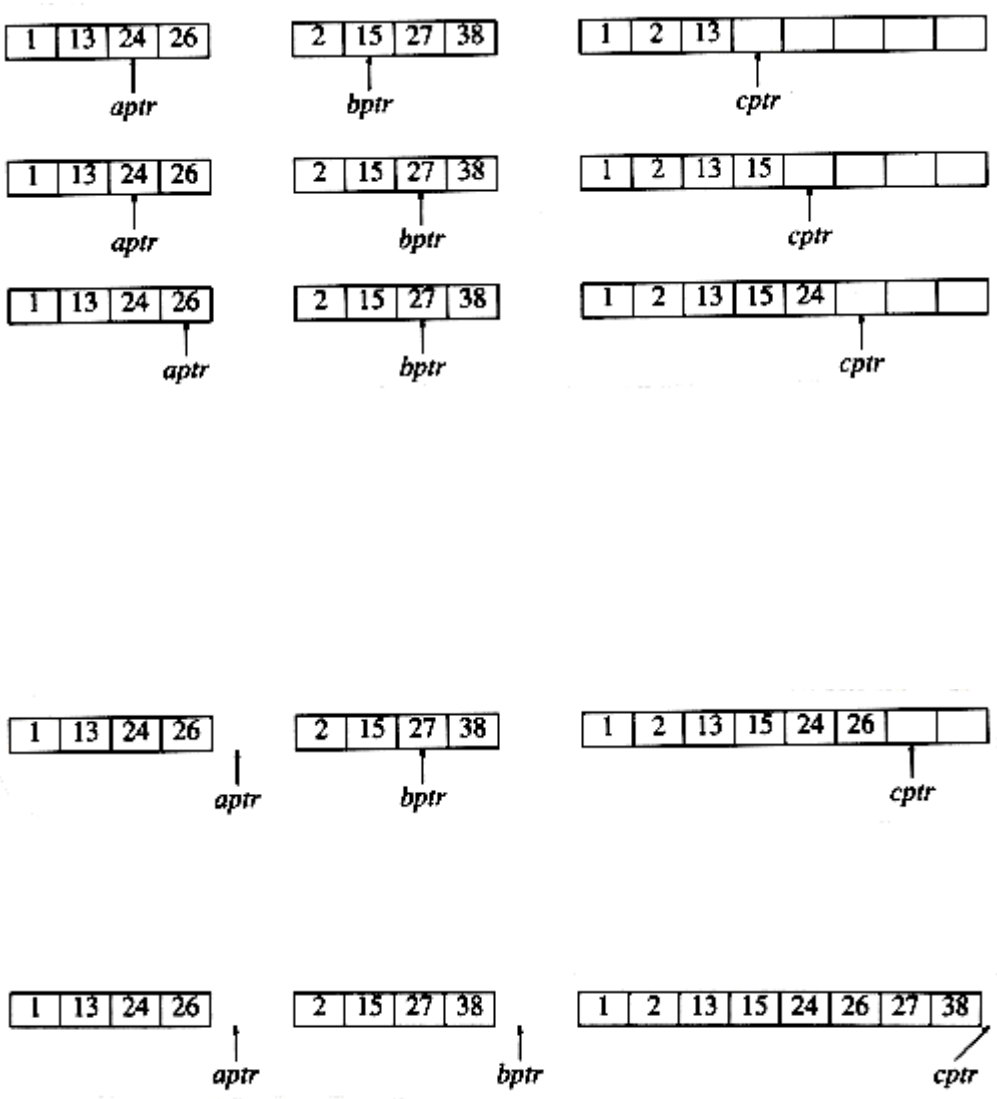
Merge sort.

It is a fine example of a recursive algorithm. The fundamental operation in this algorithm is merging two sorted lists. Because the lists are sorted, this can be done in one pass through the input, if the output is put in a third list. The basic merging algorithm takes two input arrays a and b, an output array c, and three counters, aptr, bptr, and cptr, which are initially set to the beginning of their respective arrays. The smaller of a[aptr] and b[bptr] is copied to the next entry in c, and the appropriate counters are advanced. When either input list is exhausted, the remainder of the other list is copied to c. An example of how the merge routine works is provided for the following input.



If the array a contains 1, 13, 24, 26, and b contains 2, 15, 27, 38, then the algorithm proceeds as follows: First, a comparison is done between 1 and 2. 1 is added to c, and then 13 and 2 are compared.





Routine for Merge sort Algorithm:

```
Void  mergesort( input_type a[], unsigned int n )
{
input_type *tmp_array;
tmp_array = (input_type *) malloc
( (n+1) * sizeof (input_type) );
if( tmp_array != NULL )
{
m_sort( a, tmp_array, 1, n );
free( tmp_array );
}
else
fatal_error("No space for tmp array!!!");
}
void
m_sort( input_type a[], input_type tmp_array[ ],
int left, int right )
{
```

```
int center;
if( left < right )
{
center = (left + right) / 2;
m_sort( a, tmp_array, left, center );
m_sort( a, tmp_array, center+1, right );
merge( a, tmp_array, left, center+1, right );
}
}
```

Implementation of Merge sort:

```
#include<stdio.h>
#define MAX 50
void mergeSort(int arr[],int low,int mid,int high);
void partition(int arr[],int low,int high);
int main()
{
     int merge[MAX],i,n;
     printf("Enter the total number of elements: ");
     scanf("%d",&n);
     printf("Enter the elements which to be sort: ");
          for(i=0;i<n;i++)
          {
           scanf("%d",&merge[i]);
          }
     partition(merge,0,n-1);
     printf("After merge sorting elements are: ");
          for(i=0;i<n;i++)
          {
          printf("%d ",merge[i]);
          }
  return 0;
}
void partition(int arr[],int low,int high)
{
 int mid;

          if(low<high)
          {
          mid=(low+high)/2;
          partition(arr,low,mid);
          partition(arr,mid+1,high);
          mergeSort(arr,low,mid,high);
          }
}
void mergeSort(int arr[],int low,int mid,int high)
{
   int i,m,k,l,temp[MAX];
```

```
    l=low;
    i=low;
    m=mid+1;
                while((l<=mid)&&(m<=high))
                {
                        if(arr[l]<=arr[m])
                        {
                        temp[i]=arr[l];
                        l++;
                        }
                        else
                        {
                        temp[i]=arr[m];
                        m++;
                        }
                 i++;
                }

                if(l>mid)
                {
                        for(k=m;k<=high;k++)
                        {
                        temp[i]=arr[k];
                         i++;
                        }
                }
                else
                {
                        for(k=l;k<=mid;k++)
                        {
                         temp[i]=arr[k];
                         i++;
                        }
                }
                for(k=low;k<=high;k++)
                {
                arr[k]=temp[k];
                }
}
```

Quick sort

As its name implies, quicksort is the fastest known sorting algorithm in practice. Its average running time is O(n log n). It is very fast, mainly due to a very tight and highly optimized inner loop. It has O(n2) worst-case performance, but this can be made exponentially unlikely with a little effort. The quicksort algorithm is simple to understand and prove correct, although for many years it had the reputation of being an algorithm that could in theory be highly optimized but in practice was impossible to code correctly (no doubt

because of FORTRAN). Like mergesort, quicksort is a divide-and-conquer recursive algorithm.

Routine for Quick sort

```
Void  q_sort( input_type a[], int left, int right )
{
int i, j;
input_type pivot;
if( left + CUTOFF <= right )
{
pivot = median3( a, left, right );
i=left; j=right-1;

{
 while( a[++i] < pivot );
 while( a[--j] > pivot );
 if( i < j )
swap( &a[i], &a[j] );
else
 break;
}
swap( &a[i], &a[right-1] ); /*restore pivot*/
q_sort( a, left, i-1 );
q_sort( a, i+1, right );
}
}
```

Figure 7.14 Main quicksort routine

```
i=left+1; j=right-2;

{
while( a[i] < pivot ) i++;
 while( a[j] > pivot ) j--;
 if( i < j )
swap( &a[i], &a[j] );
else
 break;
}
```

Implementation of Quick Sort:

```
#include<conio.h>
#include<stdio.h>
int qsort[25];
void sort(int low,int high);

void sort(int low,int high)
{
int temp;
        if(low<high)
        {
```

```
                int i=low+1;
                int j=high;
                  int t=qsort[low];
                          while(1)
                          {
                                  while(qsort[i]<t) i++;
                                   while(qsort[j]>t) j--;
                                           if(i<j)
                                           {
                                                   temp=qsort[i];
                                                   qsort[i]=qsort[j];
                                                    qsort[j]=temp;
                                                    i++;j--;
                                           }
                                  else break;
                                    }
          qsort[low]=qsort[j];
           qsort[j]=t;
          sort(low,j-1);
           sort(j+1,high);
          }
else return;
}
void main()
{
int n;
clrscr();
printf("\nEnter no of elements:");
scanf("%d",&n);
printf("\nEnter elements to be sorted");
for(int i=1;i<=n;i++)
scanf("%d",&qsort[i]);
sort(1,n);
printf("\nSORTED ELEMENT :");
for(i=1;i<=n;i++)
printf("%d\t",qsort[i]);
getch();
}
```

Example

8 1 4 9 0 3 5 2 7 6

i j

We then swap the elements pointed to by i and j and repeat the process until i and j cross.

After First Swap

---

2 1 4 9 0 3 5 8 7 6

i j

Before Second Swap

---

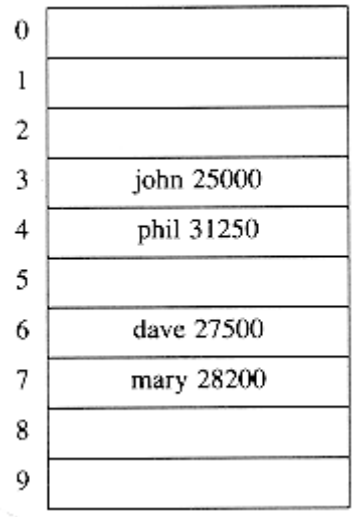
2 1 4 9 0 3 5 8 7 6
i j
After Second Swap
---
2 1 4 5 0 3 9 8 7 6
i j
Before Third Swap
---
2 1 4 5 0 3 9 8 7 6
j i

At this stage, i and j have crossed, so no swap is performed. The final part of the partitioning is to swap the pivot element with the element pointed to by i.
After Swap with Pivot
---
2 1 4 5 0 3 6 8 7 9
i pivot

Hash function.

The implementation of hash tables is frequently called hashing. Hashing is a technique used for performing insertions, deletions and finds in constant average time.

The 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 instance, salary information). We will refer to the table size as H_SIZE, with the understanding that this is part of a hash data structure and not merely some variable floating around globally. The common convention is to have the table run from 0 to H_SIZE-1; we will see why shortly. Each key is mapped into some number in the range 0 to H_SIZE - 1 and placed in the appropriate cell. The mapping is called a hash function, which ideally should be simple to compute and should ensure that any two distinct keys get different cells. Figure 5.1 is typical of a perfect situation. In this example, john hashes to 3, phil hashes to 4, dave hashes to 6, and mary hashes to 7.



Open Hashing (Separate Chaining)

The first strategy, commonly known as either open hashing, or separate chaining,

is to keep a list of all elements that hash to the same value. For convenience, our lists have headers. This makes the list implementation the same as in Chapter 3. If space is tight, it might be preferable to avoid their use. We assume for this section that the keys are the first 10 perfect squares and that the hashing function is simply hash(x) = x mod 10. (The table size is not prime, but is used here for simplicity.) Figure 5.6 should make this clear. To perform a find, we use the hash function to determine which list to traverse. We then traverse this list in the normal manner, returning the position where the item is found. To perform an insert, we traverse down the appropriate list to check whether the element is already in place (if duplicates are expected, an extra field is usually kept, and this field would be incremented in the event of a match). If the element turns out to be new, it is inserted either at the front of the list or at the end of the list, whichever is easiest. This is an issue most easily addressed while the code is being written. Sometimes new elements are inserted at the front of the list, since it is convenient and also because frequently it happens that recently inserted elements are the most likely to be accessed in the near future

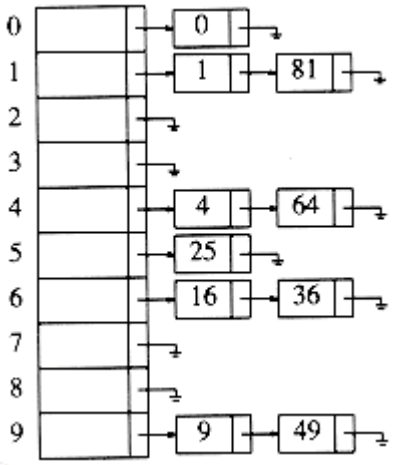


Figure 5.6 An open hash table

Type declaration for open hash table

```
typedef struct list_node *node_ptr;
struct list_node
{
element_type element;
node_ptr next;
};
typedef node_ptr LIST;
typedef node_ptr position;

/* LIST *the_list will be an array of lists, allocated later */
/* The lists will use headers, allocated later */

struct hash_tbl
{

unsigned int table_size;
LIST *the_lists;
};
```

Initialization routine for open hash table

```
HASH_TABLE
initialize_table( unsigned int table_size )
{
HASH_TABLE H;
int i;
if( table size < MIN_TABLE_SIZE )
{
error("Table size too small");
 return NULL;
}
/* Allocate table */
H = (HASH_TABLE) malloc ( sizeof (struct hash_tbl) );
if( H == NULL )
fatal_error("Out of space!!!");
H->table_size = next_prime( table_size );
/* Allocate list pointers */
H->the_lists = (position *)
malloc( sizeof (LIST) * H->table_size );
 if( H->the_lists == NULL )
 fatal_error("Out of space!!!");
/* Allocate list headers */
 for(i=0; i<H->table_size; i++ )
{
```

```
H->the_lists[i] = (LIST) malloc
( sizeof (struct list_node) );
if( H->the_lists[i] == NULL )
fatal_error("Out of space!!!");
else
H->the_lists[i]->next = NULL;
}
return H;
```

Find routine for open hash table

```
position
find( element_type key, HASH_TABLE H )
{
position p;
LIST L;
 L = H->the_lists[ hash( key, H->table_size) ];
p = L->next;
while( (p != NULL) && (p->element != key) )



/* Probably need strcmp!! */
 p = p->next;
return p;
```

Insert routine for open hash table

```
Void insert( element_type key, HASH_TABLE H )
{
position pos, new_cell;
LIST L;
pos = find( key, H );
if( pos == NULL )
{
new_cell = (position) malloc(sizeof(struct list_node));
if( new_cell == NULL )
fatal_error("Out of space!!!");
else
{
L = H->the_lists[ hash( key, H->table size ) ];
new_cell->next = L->next;
new_cell->element = key; /* Probably need strcpy!! */
L->next = new_cell;
}
```

Linear search

The following code implements linear search (Searching algorithm) which is used to find whether a given number is present in an array and if it is present then at what location it occurs. It is also known as sequential search. It is very simple and works as

follows: We keep on comparing each element with the element to search until the desired element is found or list ends. Linear search in c language for multiple occurrences and using function.

```
#include <stdio.h>
 int main()
{
  int array[100], search, c, n;
  printf("Enter the number of elements \n");
  scanf("%d",&n);
  printf("/nEnter the elements”);
  for (c = 0; c < n; c++)
    scanf("%d", &array[c]);
  printf("Enter the number to search\n");
  scanf("%d", &search);
  for (c = 0; c < n; c++)
  {
    if (array[c] == search)
    {
      printf("%d is present at location %d.\n", search, c+1);
      break;
    }
  }
  if (c == n)
    printf("%d is not present in array.\n", search);
    return 0;
}
```

Figure %: The array we're searching

Lets search for the number 3. We start at the beginning and check the first element in the array. Is it 3?

Figure %: Is the first value 3?

No, not it. Is it the next element?

Figure %: Is the second value 3?

Not there either. The next element?

Figure %: Is the third value 3?

Not there either. Next?

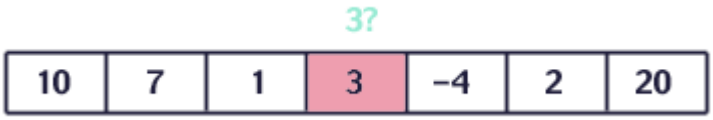


Figure %: Is the fourth value 3?

.
Hence the value 3 is found

binary search

This code implements binary search in c language. It can only be used for sorted arrays, but it's fast as compared to linear search. If you wish to use binary search on an array which is not sorted then you must sort it using some sorting technique say merge sort and then use binary search algorithm to find the desired element in the list. If the element to be searched is found then its position is printed.

```
#include<stdio.h>
int BinarySearch(int *array, int n, int key)
{
	int low = 0, high = n-1, mid;
	while(low <= high)
	{
		mid = (low + high)/2;
		 if(array[mid] < key)
		 {
				low = mid + 1;
		 }
		 else if(array[mid] == key)
		 {
				return mid;
		 }
		 else if(array[mid] > key)
		 {
				high = mid-1;
		 }
		 }
		  return -1;
}
int main()
{
	int n;
	int array[n];
	int i,key,index;
	printf("\nEnter the number of elements");
	scanf("%d",&n);
	printf("\nEnter the elements");
		 for(i = 0;i < n;i++)
		 {
		scanf("%d",&array[i]);
		 }
		  for(i = 1;i < n;i++)
		 {
				if(array[i] < array[i - 1])
```

```
            {
                        printf("Given input is not sorted\n");
                        return 0;
            }
            }
printf("Enter the key to be searched");
scanf("%d",&key);
      index = BinarySearch(array,n,key);
      if(index==-1)
      {
            printf("Element not found\n");




   }


 else
      {
            printf("Element is at index %d\n",index);
      }
      return 0;
}
```

Example

The list to be searched: L = 1 3 4 6 8 9 11. The value to be found: X = 4.
Compare X to 6. It's smaller. Repeat with L = 1 3 4.
Compare X to 3. It's bigger. Repeat with L = 4.
Compare X to 4. It's equal. We're done, we found X.

Radix sort

Radix Sort puts the elements in order by comparing the **digits of the numbers** Radix sort is one of the linear sorting algorithms for integers. It functions by sorting the input numbers on each digit, for each of the digits in the numbers. However, the process adopted by this sort method is somewhat counterintuitive, in the sense that the numbers are sorted on the least-significant digit first, followed by the second-least significant digit and so on till the most significant digit.

Consider the following 9 numbers:
493 812 715 710 195 437 582 340 385

We should start sorting by comparing and ordering the **one's** digits

Digit	Sublist
0	340 710
1	
2	812 582

3	493
4	
5	715 195 385
6	
7	437
8	
9	

Notice that the numbers were added onto the list in the order that they were found, which is why the numbers appear to be unsorted in each of the sublists above. Now, we gather the sublists (in order from the 0 sublist to the 9 sublist) into the main list again: 340 710 812 582 493 715 195 385 43

Note: The **order** in which we divide and reassemble the list is **extremely important** as this is one of the foundations of this algorithm.Now, the sublists are created again, this time based on the **ten's** digit:

Digit	Sublist
0	
1	710 812 715
2	
3	437
4	340
5	
6	
7	
8	582 385
9	493 195

Now the sublists are gathered in order from 0 to 9:

710 812 715 437 340 582 385 493 195

Finally, the sublists are created according to the **hundred's** digit:

Digit	Sublist
0	
1	195
2	
3	340 385
4	437 493
5	582
6	
7	710 715
8	812
9	

At last, the list is gathered up again:

195 340 385 437 493 582 710 715 812

Implementation of Radix Sort:

```
#include<stdio.h>
#include<conio.h>
Int getMax(int arr[],int n)
{
Int mx=arr[0];
For(int i=1;i<n;i++)
if(arr[i]>mx)
mx=arr[i];
return mx;
}
Int countSort(int arr[],int n,int exp)
{
Int output[100];
Int I,count[10]= {0}
For(i=0;i<n;i++)
Count[(arr[i]/exp)%10]++;
For(i=1;i<10;i++)
For(i=n-1;i>=0;i--)
{
Output[count[(arr[i]/exp)%10]-1]=arr[i];

Count[(arr[i/exp)%10]--;
}
For(i=0;i<n;i++)
Arr[i]=output[i];
Void radix sort(int arr[],int n)
{
Int m=getMax(arr,n);
For(int exp=1;m/exp>0;exp*=10)
countSort(arr,n,exp);
}
Void print(int arr[],int n)
{
For(int i=0;i<n;i++)
Printf("%d\t",arr[i]);
}
```