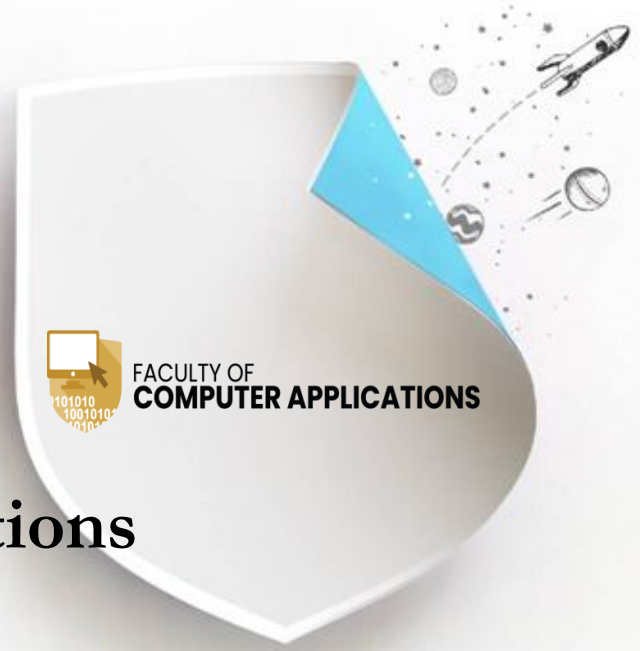
Marwadi University
Marwadi Chandarana Group

NAAC A+



Master of Computer Applications

(MCA Sem : 1)

(05MC0101) (Data structure using C)

UNIT – 5

Searching & Sorting

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.

Searching

There are two popular methods for searching the array elements: *linear search and binary search.*

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};

Linear Search

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 from 0).

Linear Search


```
#include<stdio.h>
#include<conio.h>

void main()
{
   int a[20],i,x,n;
   clrscr();
   printf("How many elements?");
   scanf("%d",&n);

   printf("Enter array elements:\n");
   for(i=0;i<n;++i)
   {
                scanf("%d",&a[i]);
   }
   printf("\nEnter element to search:");
   scanf("%d",&x);
```

Linear Search

```
for(i=0;i<n;++i)
  {
        if(a[i]==x)
          break;
  }
  if(i<n)
        printf("Element found at index %d",i);
  else
        printf("Element not found");

  getch();
}
```

Binary Search

Search a sorted array by repeatedly dividing the search interval in half is called Binary Search.

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

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.

Binary Search

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.

Binary Search


```
#include<stdio.h>
#include<conio.h>

void main()
{
  int c, first, last, middle, n, search, array[100];

  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]);

  printf("Enter value to find\n");
  scanf("%d", &search);

  first = 0;
  last = n - 1;
  middle = (first+last)/2;
```

Binary Search

```
while (first <= last)
 {
   if (array[middle] < search)
   {
     first = middle + 1;
   }
   else if (array[middle] == search)
   {
     printf("%d found at location %d.\n", search, middle+1);
     break;
   }
   else
   {
     last = middle - 1;
   }
   middle = (first + last)/2;
 }
 if (first > last)
   printf("Not found! %d isn't present in the list.\n", search);

 getch();
}
```

Difference Between Linear Search and Binary Search

LINEAR SEARCH	BINARY SEARCH
An algorithm to find an element in a list by sequentially checking the elements of the list until finding the matching element	An algorithm that finds the position of a target value within a sorted array
Also called sequential search	Also called half-interval search and logarithmic search
Best case is to find the element in the first position	Best case is to find the element in the middle position
It is not required to sort the array before searching the element	It is necessary to sort the array before searching the element
Less efficient	More efficient
Less complex	More complex

Hashing

Hashing is a fundamental concept in computer science used for efficient data storage and retrieval. It involves mapping data of arbitrary size to fixed-size values, typically integers, known as hash codes or hash values. The primary purpose of hashing is to quickly locate a data record (or bucket) regardless of the size of the dataset.

In hashing, a hash function is used to generate the hash code from the input data. The hash function takes the input data and produces a fixed-size hash code as output. This hash code is used as an index or key to store or retrieve the data in a data structure called a hash table.

Hashing

Hashing offers several advantages:

Fast Access: Hashing allows for constant-time access to data in the best-case scenario, making it highly efficient for storing and retrieving information.

Dynamic Data Size: Hashing can handle datasets of varying sizes efficiently, as the size of the hash table can dynamically adjust based on the number of elements stored.

Collision Handling: Hashing provides mechanisms for handling collisions, which occur when two different data elements produce the same hash code. Techniques such as chaining or open addressing can resolve collisions efficiently.

Common applications of hashing include:

Hashing

Data Storage: Hashing is used in databases and file systems to store and retrieve records quickly.

Data Retrieval: Hashing is used in data structures such as hash tables, hash maps, and hash sets for fast lookup operations.

Password Storage: Hashing is used to store passwords securely by converting them into hash codes, making it difficult for attackers to reverse-engineer the original passwords.

Overall, hashing is a powerful technique used extensively in computer science and software engineering for efficient data storage, retrieval, and security purposes.

Hashing

A hash function is a mathematical function that takes an input (or "key") and returns a fixed-size string of bytes, which typically represents a hash code or hash value. The output generated by a hash function is commonly used to index data in hash tables, for cryptographic applications, or for data integrity verification.

A good hash function should have the following properties:

Deterministic: For the same input, a hash function should always produce the same output.

Fast: The computation of the hash value should be efficient and quick.

Uniform Distribution: The hash values should be uniformly distributed across the range of possible hash codes, reducing the likelihood of collisions.

Hashing

Minimal Collisions: A collision occurs when two different inputs produce the same hash value. A good hash function should minimize the number of collisions.

Avalanche Effect: A small change in the input should result in a significantly different hash value.

Collision Resolution Techniques

Collision resolution techniques are methods used to handle collisions that occur when two different keys hash to the same index in a hash table. There are several collision resolution techniques, two of the most common ones being open addressing and chaining.

Open Addressing:

In open addressing, also known as closed hashing, when a collision occurs, the algorithm probes the table for another empty slot to place the collided element.

There are several strategies for probing:

Linear Probing: In linear probing, if a collision occurs at index i, the algorithm checks the next index (i+1) and continues linearly until an empty slot is found. It wraps around to the beginning of the table if necessary.

Quadratic Probing: In quadratic probing, the algorithm probes slots by adding increasing quadratic values to the original hash index until an empty slot is found.

Double Hashing: In double hashing, the algorithm uses a second hash function to determine the step size for probing. It calculates the step size as the hash of the key modulo a prime number, ensuring that the step size is relatively prime to the table size.

Open addressing techniques typically require a load factor threshold to ensure efficient performance. If the load factor exceeds a certain threshold, the hash table needs to be resized to maintain a low collision rate.

Chaining:

Collision Resolution Techniques

In chaining, each slot in the hash table contains a linked list of elements that hash to the same index. When a collision occurs, the collided element is inserted into the linked list at the corresponding index.

Chaining is flexible and can handle any number of collisions without affecting the performance of other elements in the table.

However, chaining may incur additional memory overhead due to the storage of linked lists.

Each collision resolution technique has its advantages and disadvantages:

Collision Resolution Techniques

Open Addressing:

Advantages: Simple implementation, space-efficient (no additional memory overhead for pointers), cache-friendly.

Disadvantages: Increased likelihood of clustering, more sensitive to hash function quality, and requires careful handling of deletions.

Chaining:

Advantages: Can handle any number of collisions, unaffected by the quality of the hash function, easy to implement.

Disadvantages: Requires additional memory for pointers, cache-unfriendly due to pointer traversal, and less space-efficient than open addressing for small data.

The choice between open addressing and chaining depends on factors such as the expected number of elements, memory constraints, and performance requirements of the specific application.

Collision Resolution Techniques

```
#include <stdio.h>

#define TABLE_SIZE 10

int hashFunction(int key) {
   return key % TABLE_SIZE;
}

void insert(int hashTable[], int key) {
   int index = hashFunction(key);

   while (hashTable[index] != -1) {
      index = (index + 1) % TABLE_SIZE;
   }

   hashTable[index] = key;
}

void display(int hashTable[]) {
   for (int i = 0; i < TABLE_SIZE; i++) {
      if (hashTable[i] != -1) {
         printf("Index %d: %d\n", i, hashTable[i]);
      } else {
         printf("Index %d: Empty\n", i);
      }
   }
}
```

Collision Resolution Techniques

```
int main() {
    int hashTable[TABLE_SIZE];

    for (int i = 0; i < TABLE_SIZE; i++) {
        hashTable[i] = -1;
    }

    int keys[] = {23, 43, 13, 27, 56, 72, 91};
    int n = sizeof(keys) / sizeof(keys[0]);

    for (int i = 0; i < n; i++) {
        insert(hashTable, keys[i]);
    }

    display(hashTable);

    return 0;
}
```

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.

For example, if we have an array that is declared and initialized as

int A[] = {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};

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.

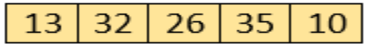
BUBBLE SORT

Algorithm

```
begin BubbleSort(arr)
   for all array elements
      if arr[i] > arr[i+1]
         swap(arr[i], arr[i+1])
      end if
   end for
   return arr
end BubbleSort
```

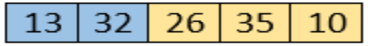
BUBBLE SORT

Let the elements of array are -

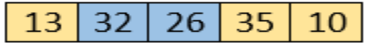


First Pass

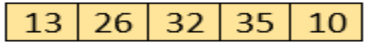
Sorting will start from the initial two elements. Let compare them to check which is greater.



Here, 32 is greater than 13 (32 > 13), so it is already sorted. Now, compare 32 with 26.



Here, 26 is smaller than 36. So, swapping is required. After swapping new array will look like -



Now, compare 32 and 35.

BUBBLE SORT

13	26	32	35	10

Here, 35 is greater than 32. So, there is no swapping required as they are already sorted.

Now, the comparison will be in between 35 and 10.

13	26	32	35	10

Here, 10 is smaller than 35 that are not sorted. So, swapping is required. Now, we reach at the end of the array. After first pass, the array will be -

13	26	32	10	35

Now, move to the second iteration.

BUBBLE SORT

Second Pass

The same process will be followed for second iteration.

13	26	32	10	35

13	26	32	10	35

13	26	32	10	35

Here, 10 is smaller than 32. So, swapping is required. After swapping, the array will be -

13	26	10	32	35

13	26	10	32	35

Now, move to the third iteration.

BUBBLE SORT

Third Pass

The same process will be followed for third iteration.

13	26	10	32	35

13	26	10	32	35

Here, 10 is smaller than 26. So, swapping is required. After swapping, the array will be -

13	10	26	32	35

13	10	26	32	35

13	10	26	32	35

Now, move to the fourth iteration.

BUBBLE SORT

Fourth pass

Similarly, after the fourth iteration, the array will be -

10	13	26	32	35

Hence, there is no swapping required, so the array is completely sorted.

BUBBLE SORT


```
#include<stdio.h>
#include<conio.h>
void main()
{
  int array[100], n,i,j, swap;
  clrscr();

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

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

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

BUBBLE SORT

```
for (i = 0 ; i < n - 1; i++)
{
  for (j = 0 ; j < n - i - 1; j++)
  {
    if (array[j] > array[j+1])
    {
                swap      = array[j];
                array[j]   = array[j+1];
                array[j+1] = swap;
    }
  }
}
```

BUBBLE SORT


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

  for (i = 0; i < n; i++)
     printf("%d\n", array[i]);

  getch();
}
```

INSERTION SORT

Insertion Sort in C is a simple and efficient sorting algorithm, that creates the final sorted array one element at a time.

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. If the given input array is sorted or nearly sorted, then it gives best performance.

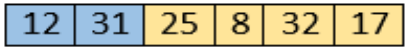
INSERTION SORT

Insertion sort works similar to the sorting of playing cards in hands. It is assumed that the first card is already sorted in the card game, and then we select an unsorted card. If the selected unsorted card is greater than the first card, it will be placed at the right side; otherwise, it will be placed at the left side. Similarly, all unsorted cards are taken and put in their exact place.

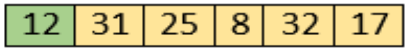
INSERTION SORT

Let the elements of array are -

Initially, the first two elements are compared in insertion sort.



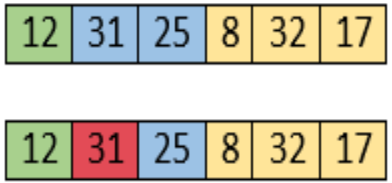
Here, 31 is greater than 12. That means both elements are already in ascending order. So, for now, 12 is stored in a sorted sub-array.



INSERTION SORT

Marwadi University
Marwadi Chandarana Group
NAAC A+

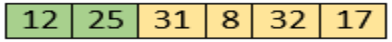
Now, move to the next two elements and compare them.



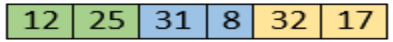
Here, 25 is smaller than 31. So, 31 is not at correct position. Now, swap 31 with 25. Along with swapping, insertion sort will also check it with all elements in the sorted array.

For now, the sorted array has only one element, i.e. 12. So, 25 is greater than 12. Hence, the sorted array remains sorted after swapping.

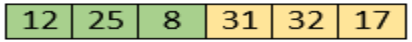
INSERTION SORT



Now, two elements in the sorted array are 12 and 25. Move forward to the next elements that are 31 and 8.



Both 31 and 8 are not sorted. So, swap them.



After swapping, elements 25 and 8 are unsorted.

INSERTION SORT

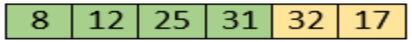
So, swap them.

Now, elements 12 and 8 are unsorted.

So, swap them too.

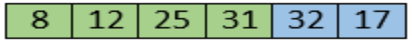
Now, the sorted array has three items that are 8, 12 and 25. Move to the next items that are 31 and 32.

Hence, they are already sorted. Now, the sorted array includes 8, 12, 25 and 31.

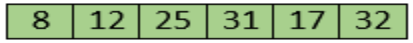


Move to the next elements that are 32 and 17.

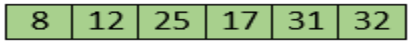
INSERTION SORT



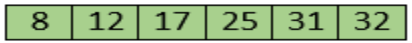
17 is smaller than 32. So, swap them.



Swapping makes 31 and 17 unsorted. So, swap them too.



Now, swapping makes 25 and 17 unsorted. So, perform swapping again.



Now, the array is completely sorted.

INSERTION SORT


```
#include<stdio.h>
#include<conio.h>
void main()
{
 int n, array[100], c, d, t, flag = 0;
 clrscr();

 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++)
 {
  t = array[c];

  for (d = c - 1 ; d >= 0; d--)
  {
```

INSERTION SORT

```
if (array[d] > t)
     {
       array[d+1] = array[d];
       flag = 1;
     }
     else
       break;
   }
   if (flag)
     array[d+1] = t;
 }

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

 for (c = 0; c <= n - 1; c++)
 {
   printf("%d\n", array[c]);
 }

 getch();
}
```

Differentiate Bubble Sort and Insertion Sort

BUBBLE SORT	INSERTION SORT
A simple sorting algorithm that repeatedly goes through the list, comparing adjacent pairs and swapping them if they are in the wrong order	A simple sorting algorithm that builds the final sorted list by transferring one element at a time
Checks the neighboring elements and swaps them accordingly	Transfers an element at a time to the partially sorted array
More number of swaps	Less number of swaps
Bubble sort is slower than insertion sort	Insertion sort is twice as fast as bubble sort
Simple	Complex than bubble sort

SELECTION SORT

Selection Sort algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end.

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 small keys.

SELECTION SORT

Marwadi University
Marwadi Chandarana Group
NAAC A+

Selection sort works as follows:

First find the smallest value in the array and place it in the first position. Then, find the second smallest value in the array and place it in the second position. Repeat this procedure until the entire array is sorted.

Selection sort is not a stable sorting algorithm.

Consider the situation in which assignment operation is very costly. so that the number of assignment operations is minimized in general.

SELECTION SORT

Marwadi University
Marwadi Chandarana Group
NAAC A+

Now, let's see the working of the Selection sort Algorithm.

To understand the working of the Selection sort algorithm, let's take an unsorted array. It will be easier to understand the Selection sort via an example.

Let the elements of array are -

12	29	25	8	32	17	40

Now, for the first position in the sorted array, the entire array is to be scanned sequentially.

At present, **12** is stored at the first position, after searching the entire array, it is found that **8** is the smallest value.

12	29	25	8	32	17	40

So, swap 12 with 8. After the first iteration, 8 will appear at the first position in the sorted array.

SELECTION SORT

8	29	25	12	32	17	40

For the second position, where 29 is stored presently, we again sequentially scan the rest of the items of unsorted array. After scanning, we find that 12 is the second lowest element in the array that should be appeared at second position.

8	29	25	12	32	17	40

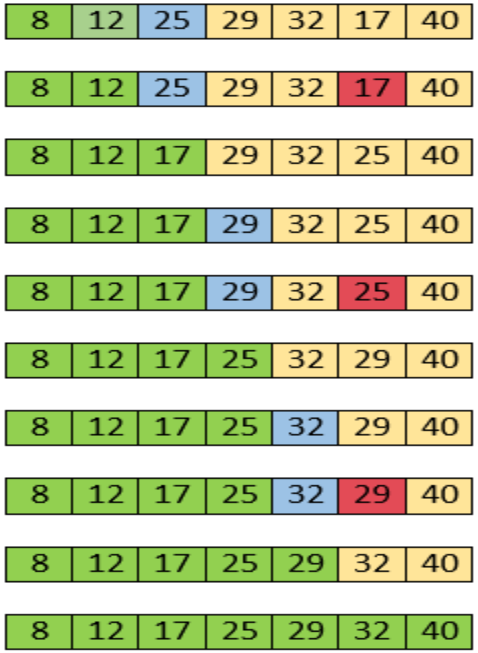
Now, swap 29 with 12. After the second iteration, 12 will appear at the second position in the sorted array. So, after two iterations, the two smallest values are placed at the beginning in a sorted way.

8	12	25	29	32	17	40

The same process is applied to the rest of the array elements. Now, we are showing a pictorial representation of the entire sorting process.

SELECTION SORT

The same process is applied to the rest of the array elements. Now, we are showing a pictorial representation of the entire sorting process.



Now, the array is completely sorted.

SELECTION SORT

```
#include <stdio.h>
#include <conio.h>
void main()
{
        int a[100], n, i, j, position, swap;
        clrscr();
        printf("Enter number of elements\n");
        scanf("%d", &n);
        printf("Enter %d Numbers\n", n);
        for (i = 0; i < n; i++)
        {
                scanf("%d", &a[i]);
        }
        for(i = 0; i < n - 1; i++)
        {
                position=i;
                for(j = i + 1; j < n; j++)
                {
                        if(a[position] > a[j])
                        position=j;
                }
```

SELECTION SORT

```
        if(position != i)
                {
                        swap=a[i];
                        a[i]=a[position];
                        a[position]=swap;
                }
        }

        printf("Sorted Array:\n");
        for(i = 0; i < n; i++)
        {
                printf("%d\n", a[i]);
        }
        getch();
}
```

DIFFERENTIATE INSERTION AND SELECTION SORT

INSERTION SORT	SELECTION SORT
A simple sorting algorithm that builds the final sorted list by transferring one element at a time	A simple sorting algorithm that repeatedly searches remaining items to find the smallest element and moves it to the correct location
Transfers an element at a time to the partially sorted array	Finds the least element and moving it accordingly
More efficient than selection sort	Less efficient than insertion sort
Complex than selection sort	Simpler than insertion sort

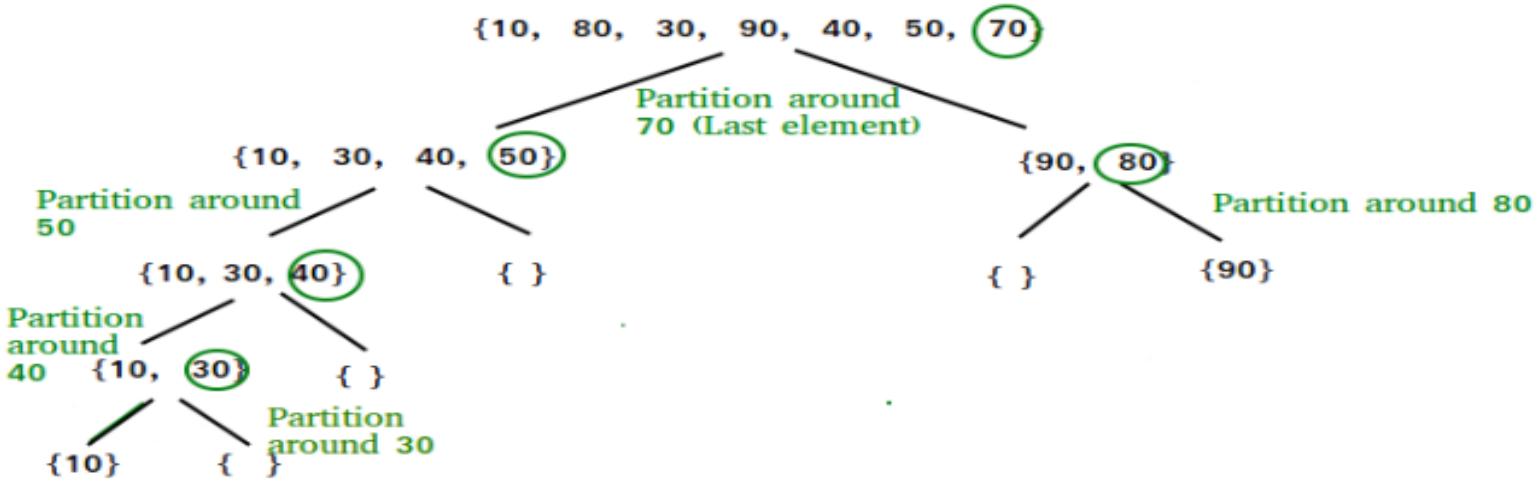
QUICK SORT

Quicksort is a divide and conquer algorithm.

The steps are: 1) Pick an element from the array, this element is called as pivot element. 2) Divide the unsorted array of elements in two arrays with values less than the pivot come in the first sub array, while all elements with values greater than the pivot come in the second sub-array (equal values can go either way). This step is called the partition operation. 3) Recursively repeat the step 2(until the sub-arrays are sorted) to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

The same logic we have implemented in the following C program.

QUICK SORT


{10, 80, 30, 90, 40, 50, 70}
Partition around
70 (Last element)
{10, 30, 40, 50}
{90, 80}
Partition around
50
Partition around 80
{10, 30, 40}
{ }
{ }
{90}
Partition
around
40
{10, 30}
{ }
Partition
around 30
{10}
{ }

QUICK SORT

```
#include<stdio.h>
#include<conio.h>

void quicksort(int number[25],int first,int last)
{
  int i, j, pivot, temp;

  if(first<last)
  {
    pivot=last;
    i=first;
    j=last;

    while(i<j)
    {
      while(number[i]<=number[pivot]&&i<last)
        i++;
      while(number[j]>number[pivot])
        j--;
      if(i<j)
      {
        temp=number[i];
        number[i]=number[j];
        number[j]=temp;
      }
    }
```

QUICK SORT

```
 temp=number[pivot];
    number[pivot]=number[j];
    number[j]=temp;
    quicksort(number,first,j-1);
    quicksort(number,j+1,last);

  }
}

void main()
{
  int i, count, number[25];
  clrscr();

  printf("How many elements are u going to enter?: ");
  scanf("%d",&count);

  printf("Enter %d elements: ", count);
  for(i=0;i<count;i++)
    scanf("%d",&number[i]);

  quicksort(number,0,count-1);

  printf("Order of Sorted elements: ");
  for(i=0;i<count;i++)
    printf(" %d",number[i]);

  getch();
}
```

HEAP SORT

Heap sort is a comparison based sorting technique based on Binary Heap data structure.

It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.

HEAP SORT

Now, let's see the working of the Heapsort Algorithm.

In heap sort, basically, there are two phases involved in the sorting of elements. By using the heap sort algorithm, they are as follows –

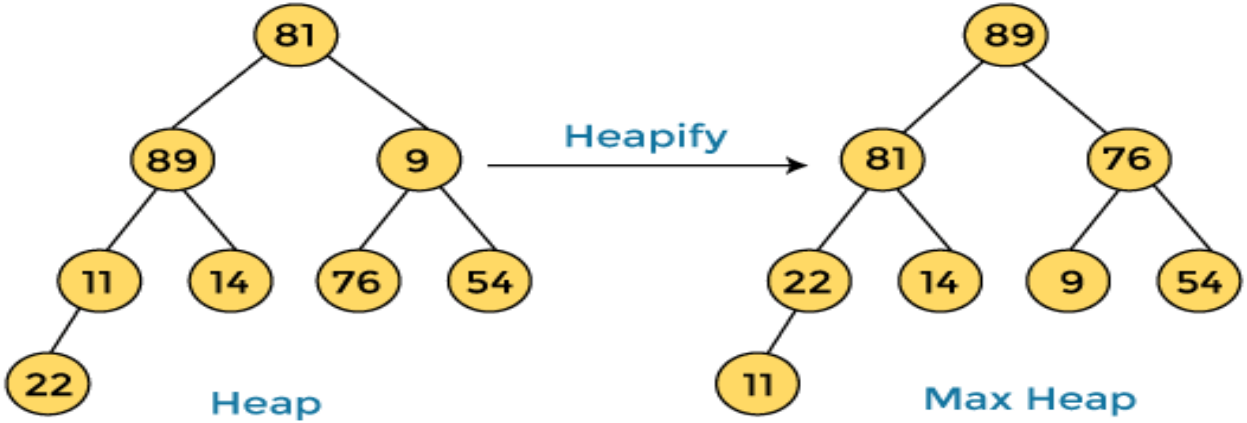
- The first step includes the creation of a heap by adjusting the elements of the array.

- After the creation of heap, now remove the root element of the heap repeatedly by shifting it to the end of the array, and then store the heap structure with the remaining elements.

HEAP SORT

81	89	9	11	14	76	54	22

First, we have to construct a heap from the given array and convert it into max heap.

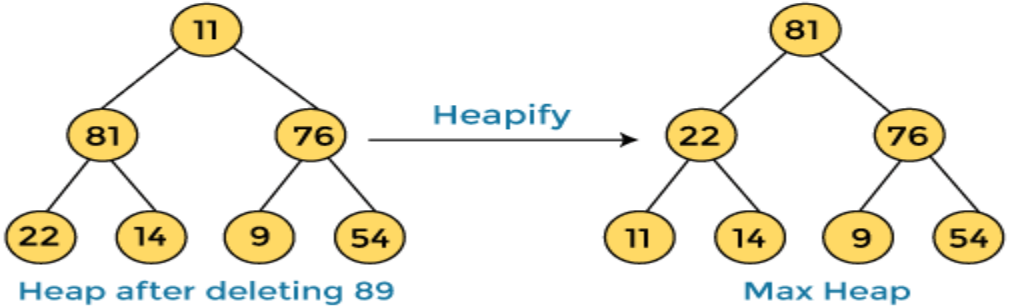


HEAP SORT

After converting the given heap into max heap, the array elements are -

89	81	76	22	14	9	54	11

Next, we have to delete the root element **(89)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(11).** After deleting the root element, we again have to heapify it to convert it into max heap.

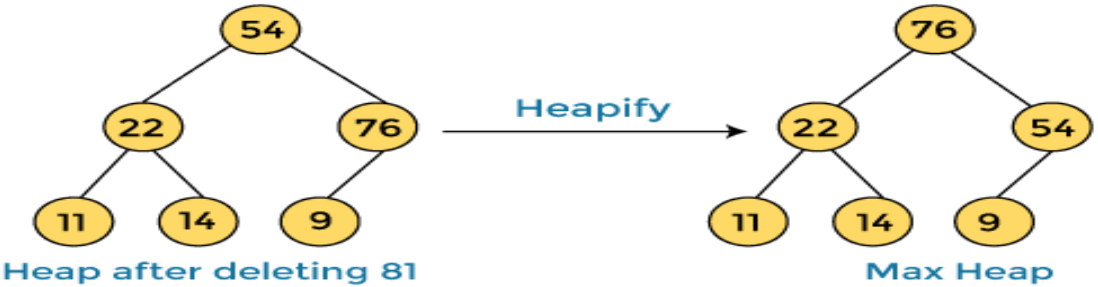


HEAP SORT

After swapping the array element **89** with **11,** and converting the heap into max-heap, the elements of array are -

81	22	76	11	14	9	54	89

In the next step, again, we have to delete the root element **(81)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(54).** After deleting the root element, we again have to heapify it to convert it into max heap.

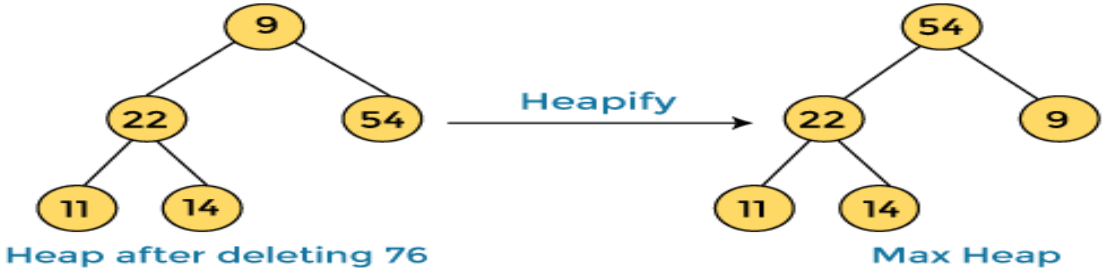


After swapping the array element **81** with **54** and converting the heap into max-heap, the elements of array are -

76	22	54	11	14	9	81	89

HEAP SORT

In the next step, we have to delete the root element **(76)** from the max heap again. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.

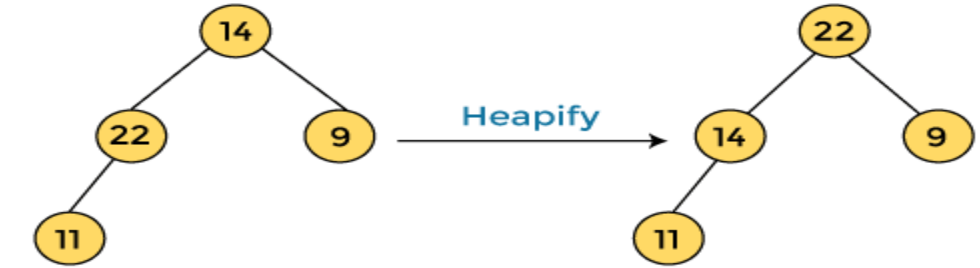


After swapping the array element **76** with **9** and converting the heap into max-heap, the elements of array are -

54	22	9	11	14	76	81	89

HEAP SORT

In the next step, again we have to delete the root element **(54)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(14).** After deleting the root element, we again have to heapify it to convert it into max heap.



Heap after deleting 54 Max Heap

After swapping the array element **54** with **14** and converting the heap into max-heap, the elements of array are -

22	14	9	11	54	76	81	89

HEAP SORT

In the next step, again we have to delete the root element **(22)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(11).** After deleting the root element, we again have to heapify it to convert it into max heap.

Heap after deleting 22 Max Heap

After swapping the array element **22** with **11** and converting the heap into max-heap, the elements of array are -

14	11	9	22	54	76	81	89

HEAP SORT

In the next step, again we have to delete the root element **(14)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.

Heap after deleting 14 Max Heap

After swapping the array element **14** with **9** and converting the heap into max-heap, the elements of array are -

11	9	14	22	54	76	81	89

HEAP SORT

After swapping the array element **14** with **9** and converting the heap into max-heap, the elements of array are -

11	9	14	22	54	76	81	89

In the next step, again we have to delete the root element **(11)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.

After swapping the array element **11** with **9,** the elements of array are -

9	11	14	22	54	76	81	89

HEAP SORT

Now, heap has only one element left. After deleting it, heap will be empty.

After completion of sorting, the array elements are -

9	11	14	22	54	76	81	89

Now, the array is completely sorted.

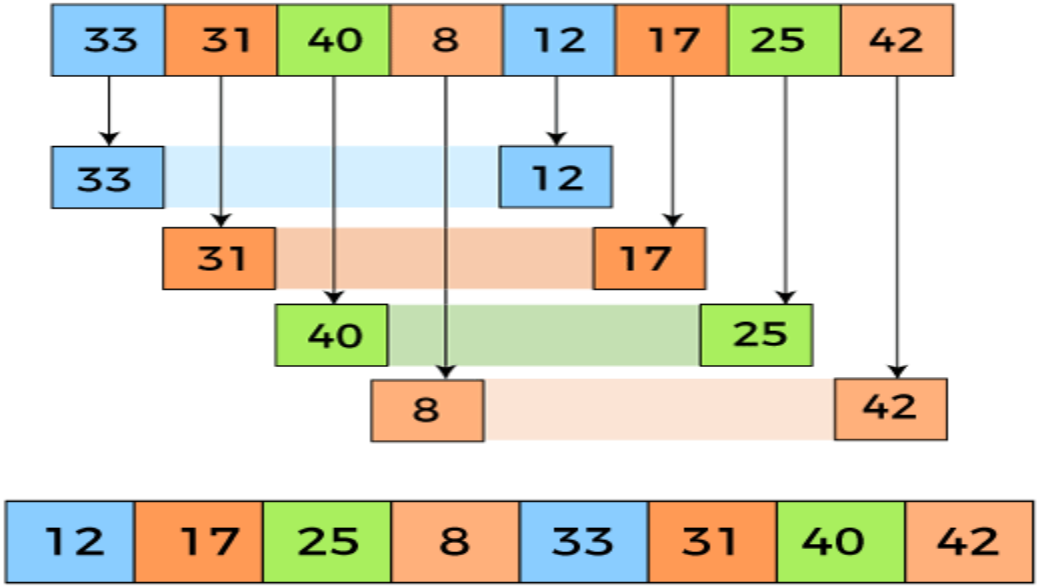
SHELL SORT

Shell sort is an algorithm that first sorts the elements far apart from each other and successively reduces the interval between the elements to be sorted. It is a generalized version of insertion sort.

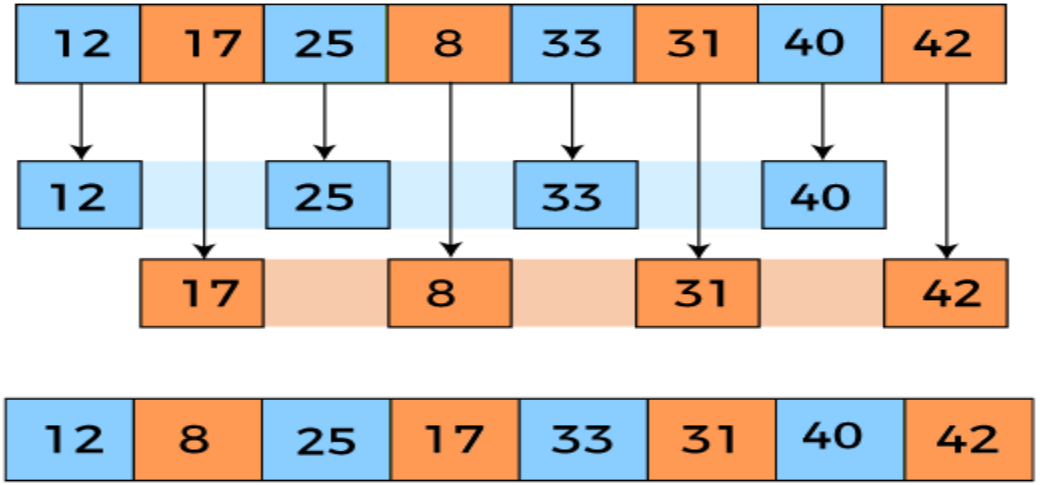
In Shell sort, elements are sorted in multiple passes and in each pass, data are taken with smaller and smaller gap sizes. However, the last step of shell sort is a plain insertion sort.

But by the time we reach the last step, the elements are already ‘almost sorted’, and hence it provides good performance.

SHELL SORT



SHELL SORT



SHELL SORT

12	8	25	17	33	31	40	42
12	8	25	17	33	31	40	42
8	12	25	17	33	31	40	42
8	12	25	17	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	31	33	40	42
8	12	17	25	31	33	40	42

SHELL SORT


```
#include <stdio.h>
#include <conio.h>
void shellsort(int arr[], int num)
{
   int i, j, k, tmp,gap;
   for (gap = num / 2; gap > 0; gap = gap / 2)
   {
      for (j = gap; j < num; j++)
      {
         for(i = j - gap; i >= 0; i = i - gap)
         {
            if (arr[i+gap] >= arr[i])
               break;
            else
            {
               tmp = arr[i];
               arr[i] = arr[i+gap];
               arr[i+gap] = tmp;
            }
         }
      }
   }
}
```

SHELL SORT


```
void main()
{
   int arr[30];
   int k,  num;
   clrscr();
   printf("Enter total no. of elements : ");
   scanf("%d", &num);
   printf("\nEnter %d numbers: ", num);

   for (k =  0 ; k < num; k++)
   {
      scanf("%d", &arr[k]);
   }
   shellsort(arr, num);
   printf("\n Sorted array is: ");
   for (k = 0; k < num; k++)
      printf("%d ", arr[k]);
   getch();
}
```

RADIX SORT

Radix sort is a sorting technique that sorts the elements by first grouping the individual digits of same place value. Then, sort the elements according to their increasing/decreasing order.

A **sorting algorithm** is an algorithm that puts components of a listing in a certain order. The most-used orders are numerical order and lexicographic order.

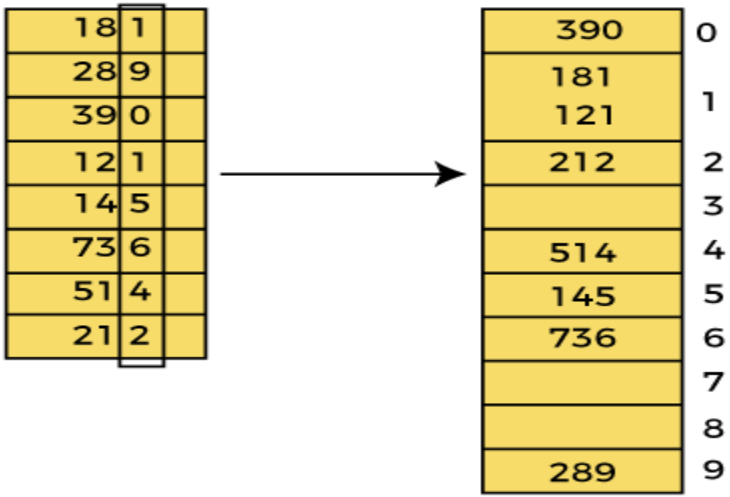
The **Radix** sort is a non-comparative sorting algorithm. The Radix sort algorithm is the most preferred algorithm for the unsorted list.

RADIX SORT

181	289	390	121	145	736	514	212

Pass 1:

In the first pass, the list is sorted on the basis of the digits at 0's place.



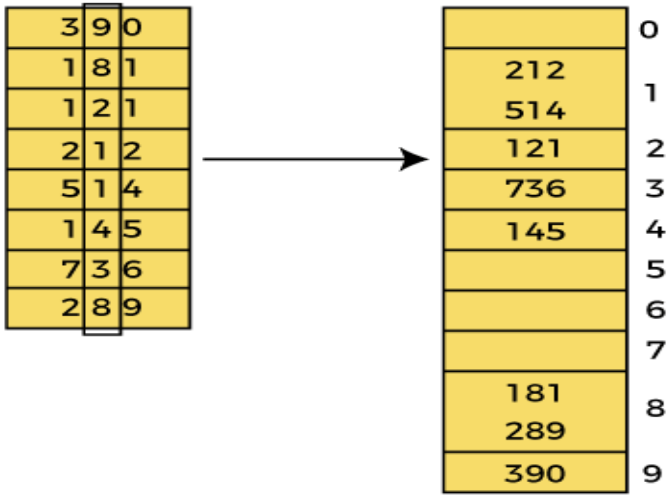
RADIX SORT

After the first pass, the array elements are -

390	181	121	212	514	145	736	289

Pass 2:

In this pass, the list is sorted on the basis of the next significant digits (i.e., digits at 10^{th} place).

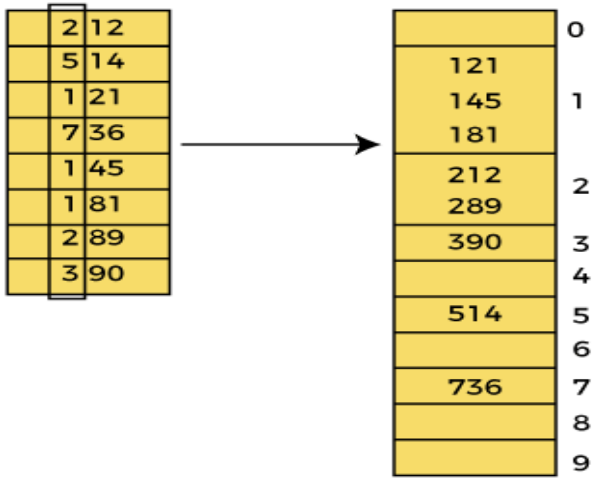


RADIX SORT

212	514	121	736	145	181	289	390

Pass 3:

In this pass, the list is sorted on the basis of the next significant digits (i.e., digits at 100th place).



RADIX SORT

After the third pass, the array elements are -

121	145	181	212	289	390	514	736

Now, the array is sorted in ascending order.

MERGE SORT

Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves.

It is very efficient sorting algorithm with near optimal number of comparison. Recursive Algorithm used for merge sort comes under the category of divide and conquer technique. It is stable sorting algorithm.

You can sort lots of amount of data using small available main memory using merge sort.

An array of n elements is split around its center producing two smaller arrays. After these two arrays are sorted independently, they can be merged to produce the final sorted array.

MERGE SORT

Working of Merge sort Algorithm

Now, let's see the working of merge sort Algorithm.

To understand the working of the merge sort algorithm, let's take an unsorted array. It will be easier to understand the merge sort via an example.

Let the elements of array are –

According to the merge sort, first divide the given array into two equal halves. Merge sort keeps dividing the list into equal parts until it cannot be further divided.

As there are eight elements in the given array, so it is divided into two arrays of size 4.

MERGE SORT

Now, again divide these two arrays into halves. As they are of size 4, so divide them into new arrays of size 2.

Now, again divide these arrays to get the atomic value that cannot be further divided.

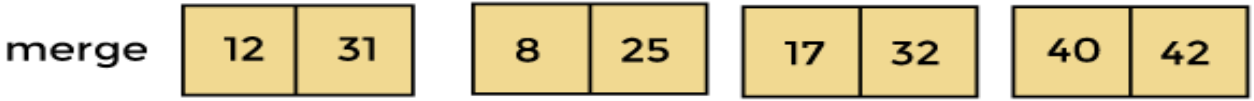
MERGE SORT

Now, combine them in the same manner they were broken.

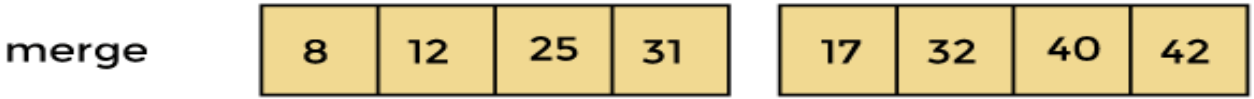
In combining, first compare the element of each array and then combine them into another array in sorted order.

So, first compare 12 and 31, both are in sorted positions. Then compare 25 and 8, and in the list of two values, put 8 first followed by 25. Then compare 32 and 17, sort them and put 17 first followed by 32. After that, compare 40 and 42, and place them sequentially.

MERGE SORT



In the next iteration of combining, now compare the arrays with two data values and merge them into an array of found values in sorted order.



Now, there is a final merging of the arrays. After the final merging of above arrays, the array will look like -

Now, the array is completely sorted.

MERGE SORT


```
#include <stdio.h>

void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;
    int L[n1], R[n2];

    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int i = 0; i < n2; i++)
        R[i] = arr[mid + 1 + i];

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
```

MERGE SORT

```
while (i < n1) {
      arr[k] = L[i];
      i++;
      k++;
   }

   while (j < n2) {
      arr[k] = R[j];
      j++;
      k++;
   }
}
void mergeSort(int arr[], int left, int right) {
   if (left < right) {
      int mid = left + (right - left) / 2;
      mergeSort(arr, left, mid);
      mergeSort(arr, mid + 1, right);
      merge(arr, left, mid, right);
   }
}
```

MERGE SORT


```
int main() {
   int arr[] = {38, 27, 43, 3, 9, 82, 10};
   int n = sizeof(arr) / sizeof(arr[0]);

   mergeSort(arr, 0, n - 1);

   for (int i = 0; i < n; i++)
      printf("%d ", arr[i]);

   return 0;
}
```

THANK YOU