Unit 6

Sorting

Sorting is a technique to arrange the elements in either ascending or descending order, which can be numerical lexicographical, or any user defined order . Sorting is a process through which the data is arranged in ascending or descending order.

Sorting can be classified into two types.

A) INTERNAL SORTING

If the data to be sorted remains in main memory and also the sorting is carried out in main memory it is called internal sorting .internal sorting takes place in the main memory of a computer .the internal sorting methods are applied to small collections of data .it means that, the entire collection of data to be sorted in small enough that the sorting can take place within main memory.

The following are some internal sorting techniques:

1. Insertion sort

2. Selection sort

3. Merge sort

4. Radix sort

5. Quick sort

6. Heap sort

7. Bubble sort

B) EXTERNAL SORTING

If the data resides in secondary memory and is brought into main memory in blocks for sorting and then result is returned back to secondary memory is called external sorting.

External sorting is required when the data been sorted do not fit into the main memory (usually ram) and instead they must reside in the slower external memory (usually a hard drive)

The following are some external sorting techniques

1) Two way external merge sort

2) K –way external merge sort

INSERTION SORT

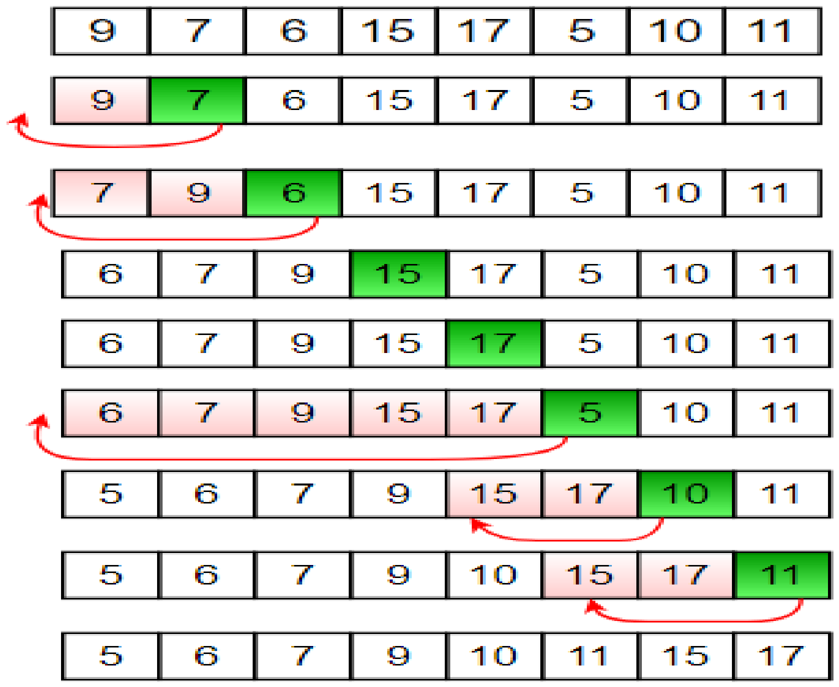
Insertion sort iterates through a list of data items .each data item is inserted at the correct position within a sorted list composed of those data items already processed.

This is an in-place comparison –based sorting algorithm. Here, a sub-list is maintained which is always sorted .for example, the lower part of an array is has to find its appropriate place and then it has to be inserted there .hence the name, insertion sort.

Insertion sort is a faster and more improved sorting algorithm than selection sort. In selection sort the algorithm iterates through all of the data through every pass whether it is already sorted or not. However, insertion sort works differently, instead of iterating through all of the data after every pass the algorithm only traverses the data it needs to until the segment that is being sorted is sorted.

Insertion Sort Algorithm
Step1−If it is the first element, it is already sorted. Return 1;
Step2−Pick next element
Step3−Compare with all elements in the sorted sub-list
Step4−Shift all the elements in the sorted sub-list that is greater than the value to be sorted
Step5−Insert the value
Step6−Repeat until list is sorted

Insertion sort example



}

Time complexity of Insertion Sort

Time complexity,T(n) =1+2+3+45+......+n
=O(n2).

Implementation of insertion sort

```
/* Insertion sort ascending order */

#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-1] > array[d]) {

  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;

  }
```

Merge sort

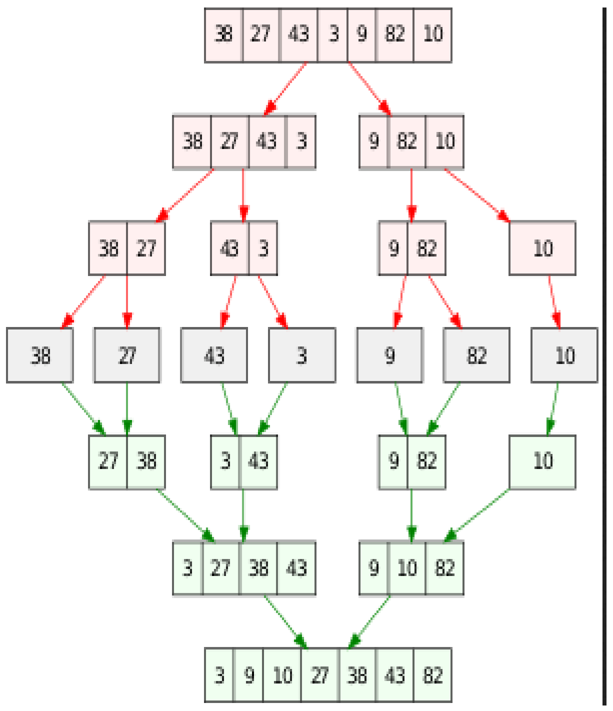
Merge sort is a sorting technique based on divide and conquer technique.With worst-case time complexity being O(nlogn),it is one of the most respected algorithms.

Merge sort divide the list in to two half and finally combine them.

Conceptually, a merge sort works as follows:

1. Divide the unsorted list into *n*sublists, each containing one element (a list of one element is considered sorted).
2. mergesublists to produce new sorted sublists until there is only one sublist remaining. This will be the sorted list.

Working procedure of merge sort



TimecomplexityofInsertionSort

Timecomplexity,T(n) =1+2+3+45+......+n
=O(nlogn).

Implementation of merge sort

```
#include<stdio.h>

#define max 10

int a[11]={10,14,19,26,27,31,33,35,42,44,0};
int b[10];

void merging(int  low,int  mid,int  high){
int l1, l2,i;

for(l1 = low, l2 = mid +1,i= low; l1 <= mid && l2 <= high;i++){
if(a[l1]<= a[l2])
b[i]= a[l1++];
else
b[i]= a[l2++];
}
while(l1 <= mid)
b[i++]= a[l1++];

while(l2 <= high)
b[i++]= a[l2++];

for(i= low;i<= high;i++)
a[i]= b[i];
}

void sort(int low,int high){
int mid;

if(low < high){
mid=(low + high)/2;
sort(low, mid);
sort(mid+1, high);
merging(low, mid, high);
}else{
return;
}
}

int main(){
int i;

printf("List before sorting\n");

for(i=0;i<= max;i++)
printf("%d ", a[i]);

sort(0, max);

printf("\nList after sorting\n");

for(i=0;i<= max;i++)
printf("%d ", a[i]);
}
```

Quick sort

In Quick sort, an element called pivot is identified and that element is fixed in its place by moving all the elements less than that to its left and all the elements greater than that to its right.

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data in to smaller arrays. A large array is partitioned in to two arrays one of which holds values smaller than the specified value , say pivot , based on which the partition is made and another array holds values greater than the pivot value.

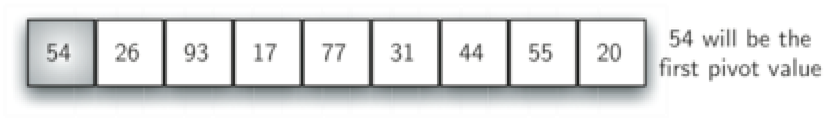
Quick sort partitions an array and then calls itself recursively twice to sort the two resulting sub arrays. This algorithm is quite efficient for large-sized data sets as its average and worst case complexity is O(nlogn),where n is the number of items.

QuickSort Partition Algorithm

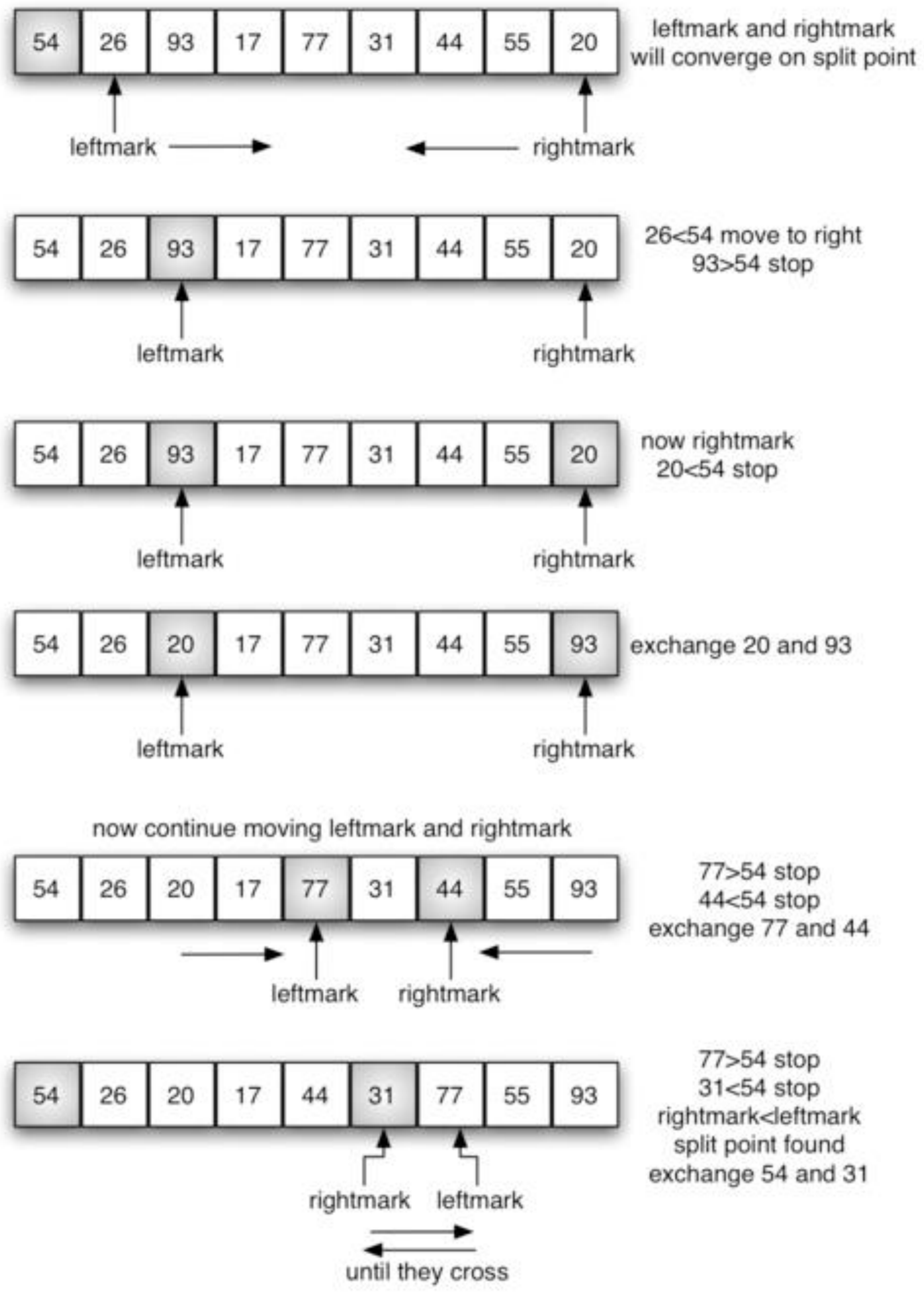
Step 1−Choose the lowest index value has pivot
Step 2−Take two variables I and j to point left and right to the list respectively
Step 3–'i' points to the low index
Step 4–'j' points to the high index
Step 5−while value at a[i] is less than pivot move 'i' right
Step 6−while value at a[j] is greater than pivot move 'j' left
Step 7−if both step5 and step6 does not match swap a[i] and a[j]
Step 8− if left ≥ right , swap pivot and a[j] , where partition of the list occurs in such away that all the elements in the left partition are less than pivot and all the elements of in the right partition are greater than pivot.

Example:

54 will serve as our first pivot value. Since we have looked at this example a few times already, we know that 54 will eventually end up in the position currently holding 31. The **partition** process will happen next. It will find the split point and at the same time move other items to the appropriate side of the list, either less than or greater than the pivot value.



Partitioning begins by locating two position markers—let's call them leftmark and rightmark—at the beginning and end of the remaining items in the list (positions 1 and 8 in. The goal of the partition process is to move items that are on the wrong side with respect to the pivot value while also converging on the split point. shows this process as we locate the position of 54.



Implementation program:

```
/* C Program for Quick Sort */
#include <stdio.h>
void Swap(int *x, int *y) {
int Temp;
   Temp = *x;
```

```
    *x = *y;
    *y = Temp;
}
voidquickSort(int a[], int first, int last) {
int pivot, i, j;
if(first < last) {
pivot = first;
i = first;
        j = last;
while (i< j) {
while(a[i] <= a[pivot] &&i< last)
i++;
while(a[j] > a[pivot])
j--;
if(i< j) {
Swap(&a[i], &a[j]);
            }
        }
Swap(&a[pivot], &a[j]);
quickSort(a, first, j - 1);
quickSort(a, j + 1, last);
    }
}
int main() {
int a[100], number, i;
printf("\n Please Enter the total Number of Elements  :  ");
scanf("%d", &number);
printf("\n Please Enter the Array Elements  :  ");
for(i = 0; i< number; i++)
scanf("%d", &a[i]);

quickSort(a, 0, number - 1);
printf("\n Selection Sort Result : ");
for(i = 0; i< number; i++)  {
printf(" %d \t", a[i]);
    }
printf("\n");
return 0;
}
```