University of Kelaniya Sri Lanka

COSC 21063

Merge Sort and QuickSort

Lecturer : Ms.D.M.L.Maheshika Dissanayake

Structure of Lesson

- Divide and Conquer Sorting
- Merge Sort and QuickSort
  - Definition
  - Method
  - Implementation
  - Analysis

Learning Outcomes

By the end of this lesson you should be able to

- Understand the concept of divide and conquer sorting methods
- Implement the merge sort and quick sort algorithms in java language
- Analyze the algorithms

Divide and Conquer Sorting

- A sorting algorithm is an algorithm whose input is *any* list and whose output is a list (or a change in the given list) that is sorted and has the same elements as the given list.
- Most well-known sorting algorithms are based on this general strategy:
- Given a list L, if L has zero or one element, then it is already sorted, so nothing needs to be done;
- Otherwise
  - divide L into two smaller lists, L1 and L2.
  - recursively sort each of the smaller lists. Result: L1 and L2 are now sorted.
  - Combine L1 and L2.
- The result is a sorted version of the original list. How do we combine L1 and L2? As just discussed there's only one way to do so and produce a sorted list. We MERGE them.

Divide and Conquer Sorting

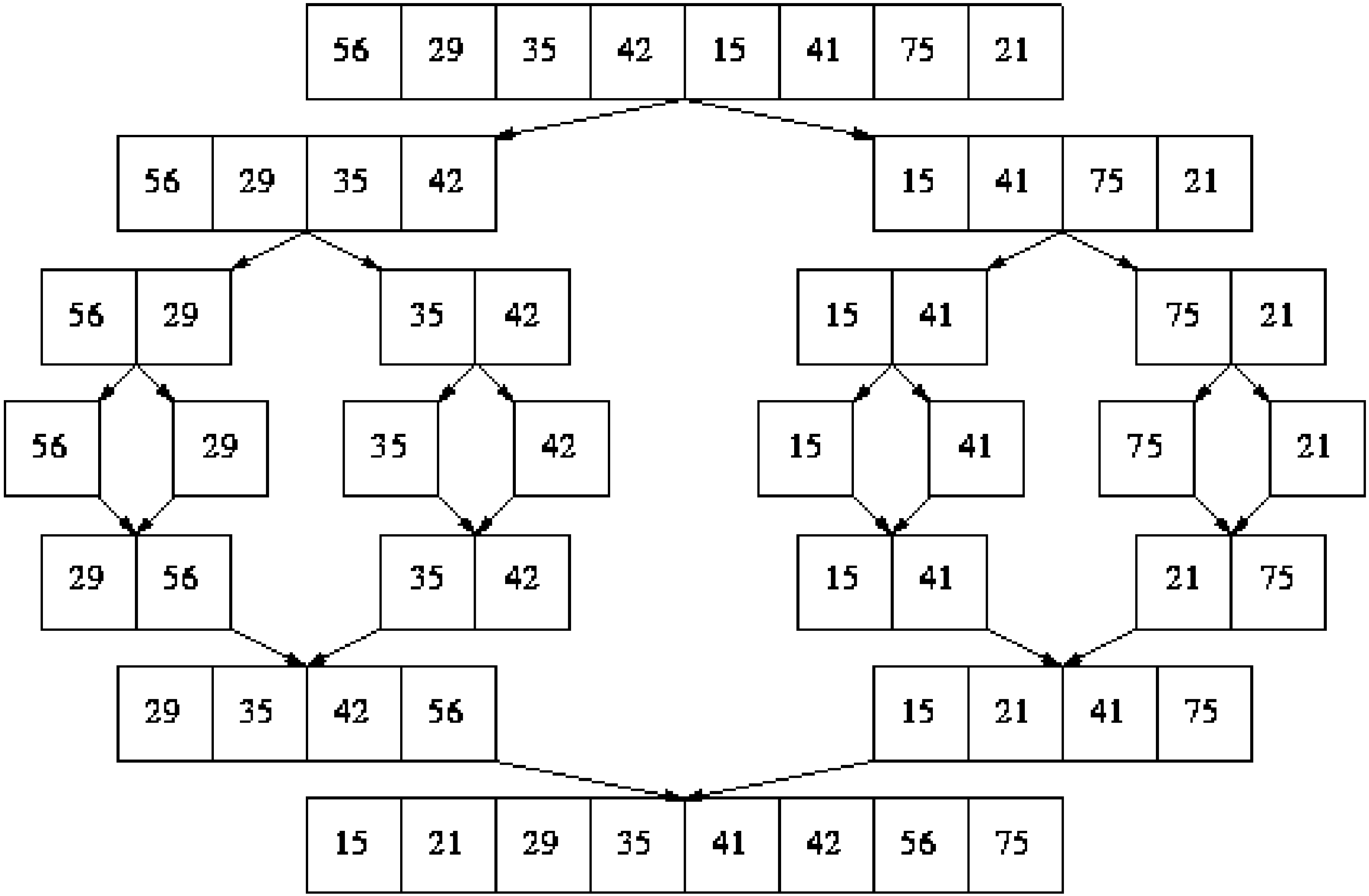
- This strategy works no matter how we do step (1).
- The pieces can be of any size; they could be the same size, or they could be very different sizes; and it does not matter which elements of L we put together or in what order we put them into the pieces.
- We can divide up and scramble up the elements any way we like, and we will still sort L.
- In fact, we could even divide L into more than two pieces, we could divide it into 3, or 7, or whatever.
- The one thing we must avoid is having one of the pieces equal to L; Here again we have an application of the idea of dividing a problem into smaller sub problems, That is *divide and conquer*.

Divide and Conquer Sorting

```
Sort(list)
        {
          if the list has length greater than 1 then
                    {
                Partition the list into L1, L2
                Sort(L1);
                Sort(L2);
                Combine(L1,L2);
           }
        }
```

Merge Sort

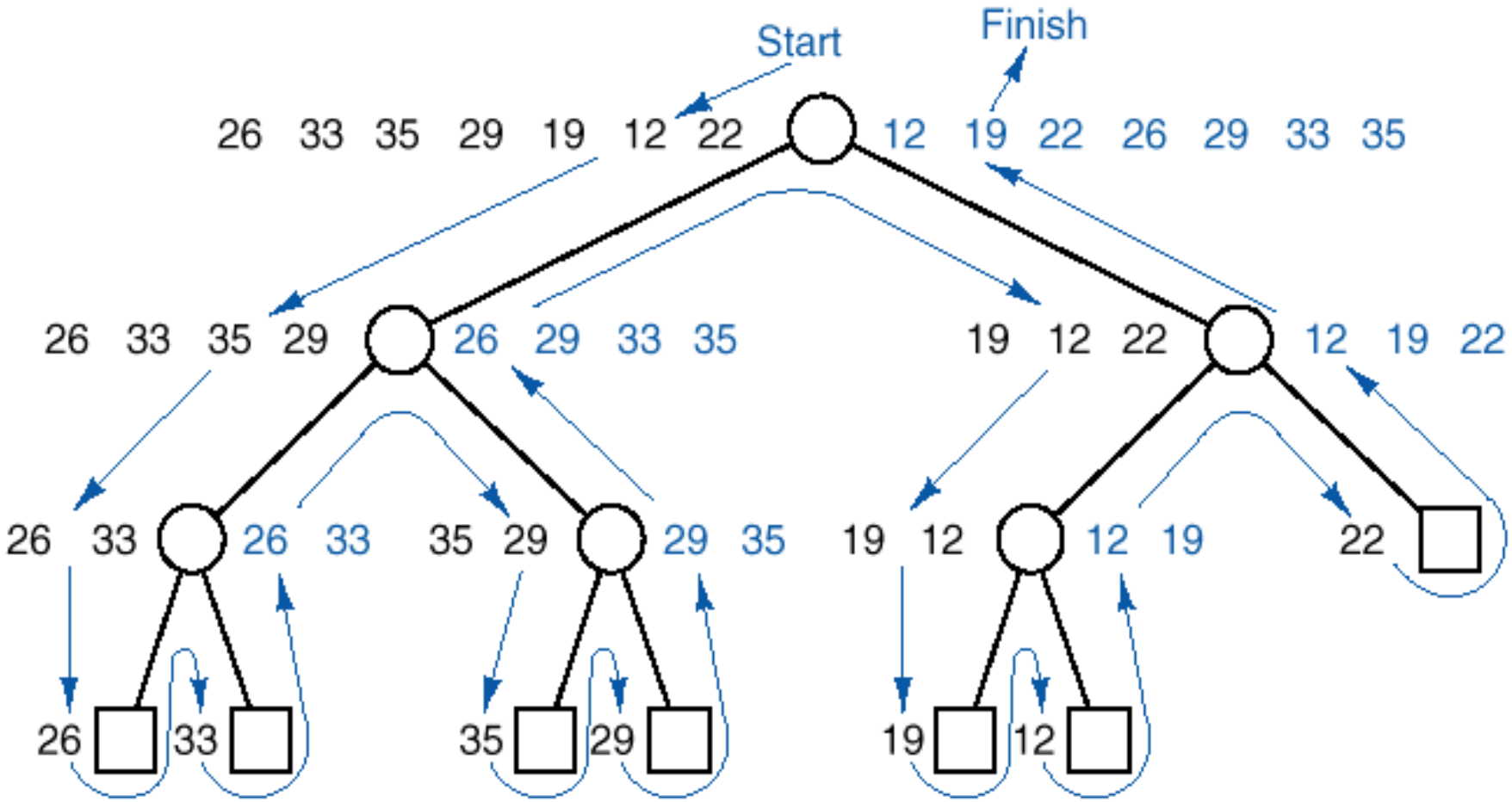
- There are various ways of dividing L into 2 equal pieces
- We will illustrate the algorithm by putting the first half of L in one piece and the second half of L in the other piece.
- We proceed until the sublists are of length one, which are always sorted, then we work our way back up the recursion by MERGING together the short lists into larger ones (which replace the list that we started with at that level).


56 29 35 42 15 41 75 21
56 29 35 42
15 41 75 21
56 29
35 42
15 41
75 21
56
29
35
42
15
41
75
21
29 56
35 42
15 41
21 75
29 35 42 56
15 21 41 75
15 21 29 35 41 42 56 75

As an example Consider the following list:

L = 26 33 35 29 19 12 22

- Chop the list into two, L1 = 26 33 35 29 and L2 = 19 12 22
- Consider the first left list
- Chop the list L1, Now L = 26 33 35 29
  L1 = 26 33 and L2 = 35 29
- Consider the left list,
- Chop the list L1: Now L = 26 33
  L1 = 26 and L2 = 33
- Since the length is one L1 and L2 are sorted, merge the sublists to get the sorted list: L = 26 33
- Do the same for the remaining lists obtained by the chopping, and merge each sublist.


Start
Finish
26 33 35 29 19 12 22
12 19 22 26 29 33 35
26 33 35 29
26 29 33 35
19 12 22
12 19 22
26 33
26 33
35 29
29 35
19 12
12 19
22
26
33
35
29
19
12

Merge Sort – Contiguous Implementation

```
void mergeSort( int start, int end)
  {
     if (start < end)
     {
        int mid = (start + end) / 2;
        mergeSort(start, mid);
        mergeSort(mid + 1, end);
        merge(start, mid, end);
     }
  }

 void merge( int start, int mid, int end)
   {
      int i, j, k;
      int n1 = mid - start + 1;
      int n2 = end - mid;

      int LeftArray[] = new int[n1];
      int RightArray[] = new int[n2];
```

Merge Sort – Contiguous Implementation

```
for (i = 0; i < n1; i++)
    LeftArray[i] = ListEntry[start+ i];
for (j = 0; j < n2; j++)
    RightArray[j] = ListEntry[mid + 1 + j];

i = 0;
j = 0;
k = start;
while (i < n1 && j < n2)
{
    if(LeftArray[i] <= RightArray[j])
    {
        ListEntry[k] = LeftArray[i];
        i++;
    }
    else
    {
        ListEntry[k] = RightArray[j];
        j++; }
    k++;
}
```

```
while (i<n1)
    {
        ListEntry[k] = LeftArray[i];
        i++;
        k++;
    }

    while (j<n2)
    {
        ListEntry[k] = RightArray[j];
        j++;
        k++;
    }
}
```

Analysis

- Consider the number of comparisons
- Comparison is done with the main loop of the merge function.
- After each comparison one of the two nodes is sent to the output list.
- Hence the number of comparisons certainly cannot exceed the number of entries being merged.
- Consider the recursion tree, for the case $n = 2^m$ a power of 2.

Analysis

- The total lengths of the lists on each level = n, the total number of entries.
- Hence the total number of comparisons done on each level cannot exceed n.
- The number of levels, excluding the leaves = lg(n) rounded up to the next smallest integer.
- The total number of merges: n/2 + n/4 + n/8 +……..+ 1 = n-1
- we can show that the total number of comparisons done by mergesort is therefore, nlg(n) –n +1.

Divide and Conquer Sorting

```
Sort(list)
        {
          if the list has length greater than 1 then
                  {
                  Partition the list into L1, L2
                  Sort(L1);
                  Sort(L2);
                  Combine(L1,L2);
           }
        }
```

QuickSort

- The idea behind *QuickSort* is to replace the MERGE operation in step 3 with the much faster JOIN operation.
- If this could be done, while still splitting the list more or less in half in step (1), we would have an algorithm that is clearly faster than *Merge Sort*.
- How can we change the way we do step (1) - SPLIT L into pieces - so that we can use JOIN instead of MERGE?
- This is the secret to *QuickSort*.

QuickSort

- When is it safe to JOIN two sorted lists L1 and L2?
- When everything in L1 is smaller than everything in L2.
- So that's how we cut our list in two.
- Pick a number, Pivot (Cutoff), and put all the elements less than Pivot into L1 and all the ones bigger than Pivot into L2.

e.g. Consider the list L = 26 33 35 29 19 12 22

Method

- Consider the list L = 26 33 35 29 19 12 22
- Choose the first number in the list as the Pivot and partition the list L, Pivot = 26
- The two sub lists L1 = 19 12 22 -less than Pivot and L2 = 33 35 29 greater than Pivot
- L1 and L2 are sorted recursively
  - **Sort L1**
- L = 19 12 22
  - Pivot 19
  - L1 = 12 and L2 = 22
  - L1 and L2 has one entry so not need to sort
  - Now combine the sublists L1 and L2 with the Pivot between them: L1 = 12 19 22
- Do the same thing for list L2: So you get L2 = 29 33 35
- Now combine the two lists L1 and L2 you get 12 19 22 26 29 33 35

Partition into (19,12,22) and (33, 35,29); pivot = 26

Sort(19,12,22)

Partition into (12)and (22) pivot = 19

Sort(12)

Sort(22)

Combine(12,19,22)

Sort(33,35,29)

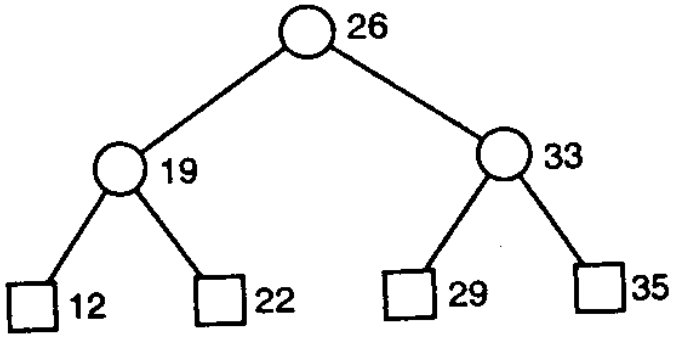
Partition into (29)and (35) pivot = 33

Sort(29)

Sort(35)

Combine(29,33,35)

Combine (12,19,22,26,29,33,35)



The recursion tree

The two calls to Sort at each level are shown as the children of the vertex.

Consider these names:

Tim, Dot, Eva, Roy, Tom, Kim, Guy, Amy, Jon, Ann, Jim, Kay, Ron, Jan

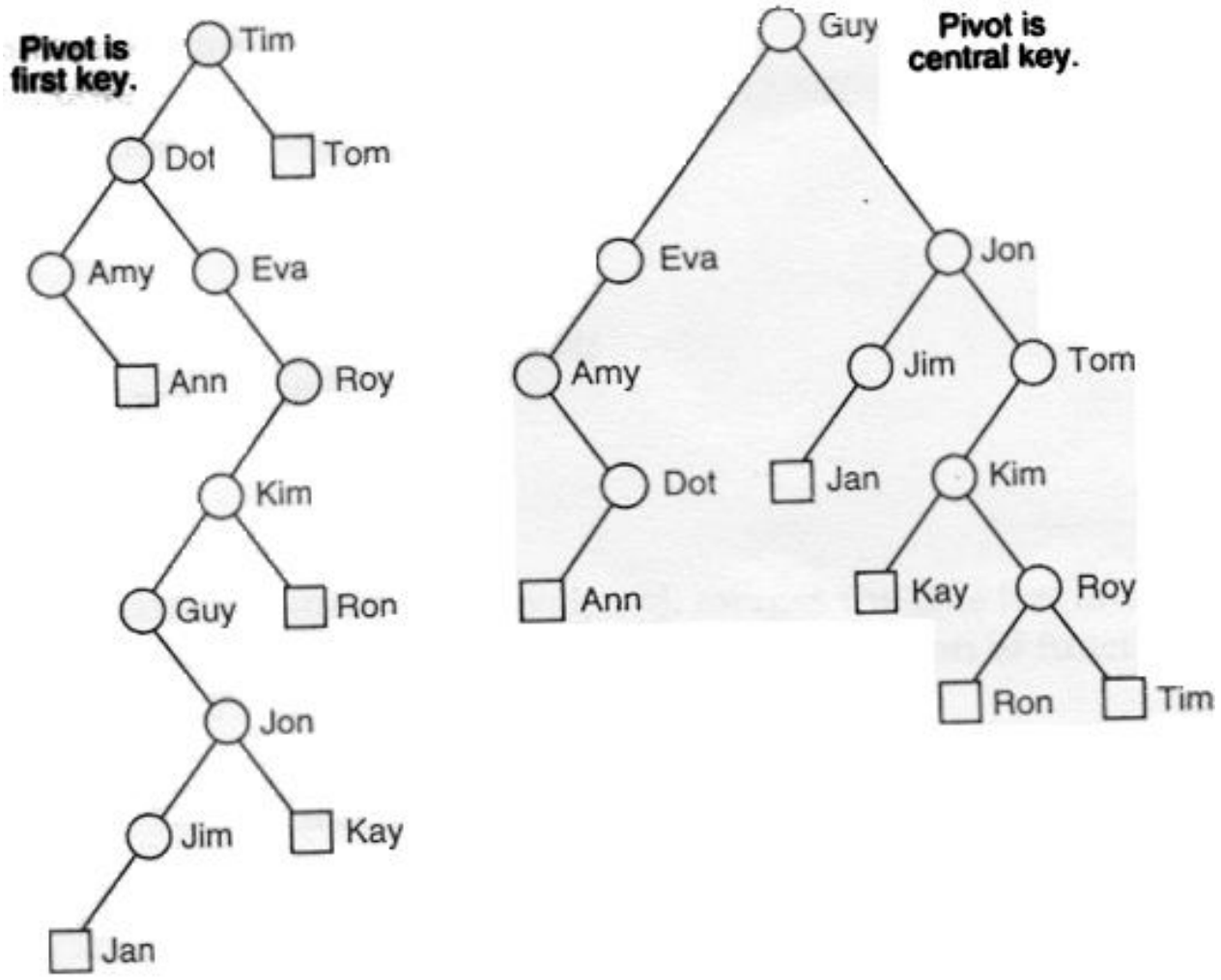


Figure 7.11. Recursion trees, quicksort of 14 name

Quick Sort for Contiguous Lists

- Must write an algorithm for partitioning entries in a list by use of a pivot key, swapping the entries, within the list so that all those with keys before the pivot come first, then the entry with the pivot key, and the entries with the larger keys.
- Let *pivotpos* – the position of the pivot in the partitioned list.
- Partitioned sublists are kept in the same array, in the proper relative position.

Algorithm Development

Partitioning the list

- There are several methods that could be used.
- The algorithm that is discussed is much simpler and easier to understand, and is not slow, it does the smallest possible number of key comparisons.
- Given a pivot value p, we must rearrange the entries of the list and compute pivotpos so that the pivot is at pivotpos all entries to its left have keys less than p, and all entries to its right have larger keys.

Quick Sort – Contiguous Implementation

```
void quick( int start, int end)
  {
     if (start < end)
     {
        int p = partition(start, end);  //p is partitioning index
        quick(start, p - 1);
        quick( p + 1, end);
     }
  }
```

Quick Sort – Contiguous Implementation

```
int partition ( int start, int end)
  {
     int pivot = ListEntry[end]; // pivot element
     int i = (start - 1);
     for (int j = start; j <= end - 1; j++)
     {
     if ( ListEntry[j] < pivot){
           i++; // increment index of smaller element
           int t =  ListEntry[i];
           ListEntry[i] =  ListEntry[j];
           ListEntry[j] = t;
        }
     }
     int t =  ListEntry[i+1];
     ListEntry[i+1] =  ListEntry[end];
     ListEntry[end] = t;
     return (i + 1);
  }
```

Summary of main points

- Divide and Conquer sorting methods
- Definitions of Mergesort and Quicksort
- Implementation of Mergesort and Quicksort
- Analysis of Mergesort and Quicksort algorithms