Assignment A-2

Aim:

To understand Divide and Conquer strategy for Quick Sort.

Problem Statement:

Using Divide and Conquer Strategies design a class for Concurrent Quick Sort using C++.

Theory:

Divide and Conquer Strategies:

The name "divide and conquer" is sometimes applied to algorithms that reduce each problem to only one sub-problem, such as the binary search algorithm for finding a record in a sorted list. These algorithms can be implemented more efficiently than general divide-and-conquer algorithms; in particular, if they use tail recursion, they can be converted into simple loops. Under this broad definition, however, every algorithm that uses recursion or loops could be regarded as a "divide and conquer algorithm". Therefore, some authors consider that the name "divide and conquer" should be used only when each problem may generate two or more subproblems. The name decrease and conquer has been proposed instead for the single-subproblem class.

In computer science, divide and conquer is an algorithm design paradigm based on multi-branched recursion. A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem. An important application of decrease and conquer is in optimization, where if the search space is reduced by a constant factor at each step, the overall algorithm has the same asymptotic complexity as the pruning step, with the constant depending on the pruning factor; this is known as prune and search.

Quick Sort Algorithm:

Quicksort is an efficient sorting algorithm, serving as a systematic method for placing the elements of an array in order. Developed by Tony Hoare in 1959, with his work published in 1961, it is still a commonly used algorithm for sorting. When implemented well, it can be about two or three times faster than its main competitors, merge sort and heapsort. Quicksort is a comparison sort, meaning that it can sort items of any type for which a "less-than" relation is defined. In efficient implementations it is not a stable sort, meaning that the relative order of equal sort items is not preserved. Quicksort can operate in-place on an array, requiring small additional amounts of memory to perform the sorting.

Mathematical analysis of quicksort shows that, on average, the algorithm takes O(n log n) comparisons to sort n items. In the worst case, it makes

$O(n^2)$ comparisons, though this behavior is rare.

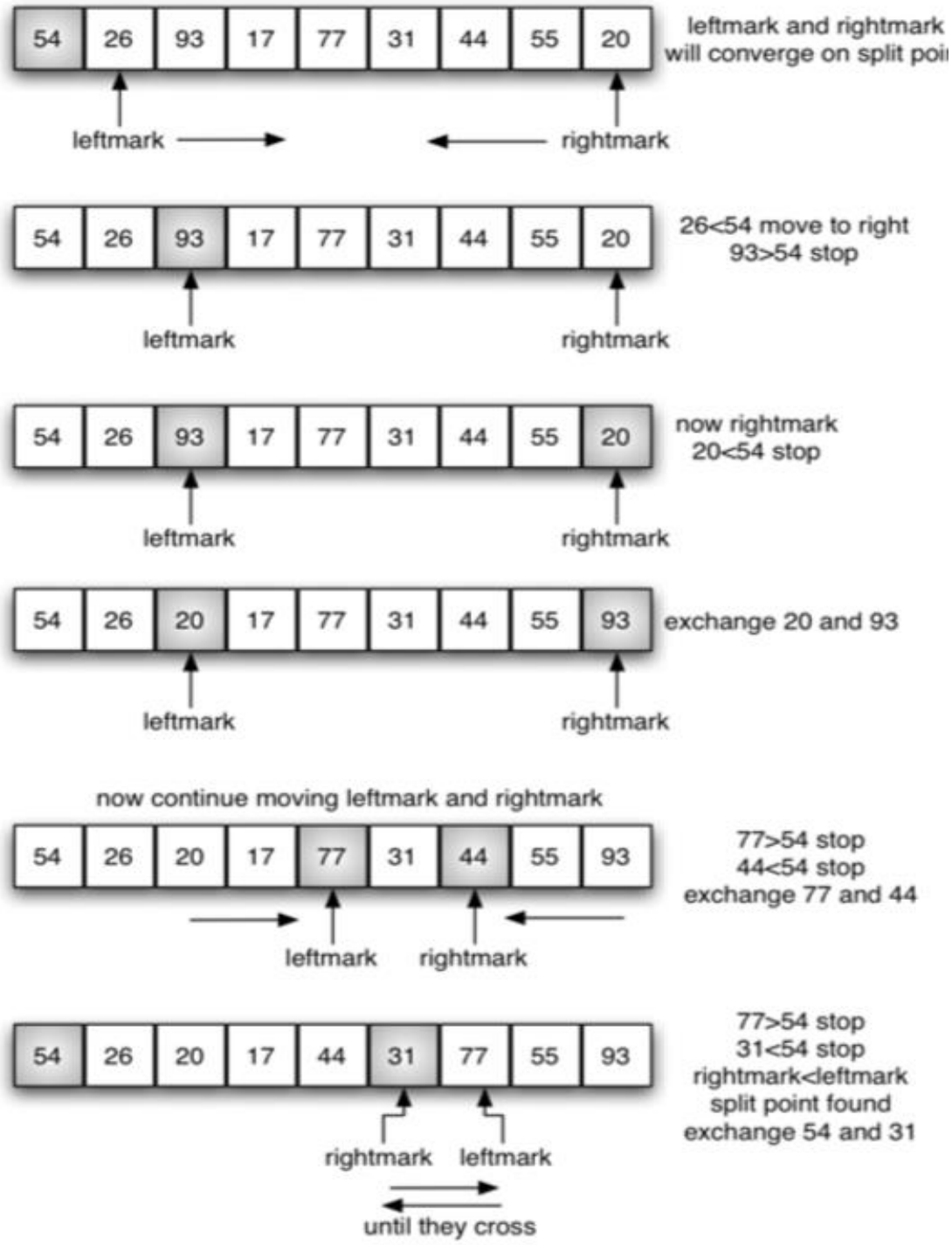


Figure 1: Quick Sort

Algorithm:

```
quicksort(A, lo, hi)
if lo < hi
    p = partition(A, lo, hi)
    quicksort(A, lo, p)
    quicksort(A, p + 1, hi)
partition(A, lo, hi)
pivot = A[lo]
i = lo - 1
j = hi + 1
while True
do
   j = j - 1
   while A[j] > pivot
   do
     i = i + 1
     while A[i] < pivot
     if i < j
     swap A[i] with A[j]
     else
     return j
```

Space Complexity:

1. Quicksort with in-place and unstable partitioning uses only constant additional space before making any recursive call. Quicksort must store a constant amount of information for each nested recursive call. Since the best case makes at most O(log n) nested recursive calls, it uses O(log n) space. However, in the worst case quicksort could make O(n) nested recursive calls and need O(n) auxiliary space.
2. From a bit complexity viewpoint, variables such as lo and hi do not use constant space; it takes O(log n) bits to index into a list of n items.
3. Another, less common, not-in-place, version of quicksort uses O(n) space for working storage and can implement a stable sort.

Parallel Sort:

1. We randomly choose a pivot from one of the processes and broadcast it to every process.
2. Each process divides its unsorted list into two lists: those smaller than (or equal) the pivot, those greater than the pivot.
3. Each process in the upper half of the process list sends its low list to a partner process in the lower half of the process list and receives a high list in return.

4. Now, the upper-half processes have only values greater than the pivot, and the lower-half processes have only values smaller than the pivot.
5. Thereafter, the processes divide themselves into two groups and the algorithm recurses.
6. After logP recursions, every process has an unsorted list of values completely disjoint from the values held by the other processes.
7. The largest value on process i will be smaller than the smallest value held by process i + 1. Each process can sort its list using sequential quicksort.

Conclusion:

Program of Concurrent Quick Sort Using Divide And Conquer Strategies in C++ is implemented successfully.

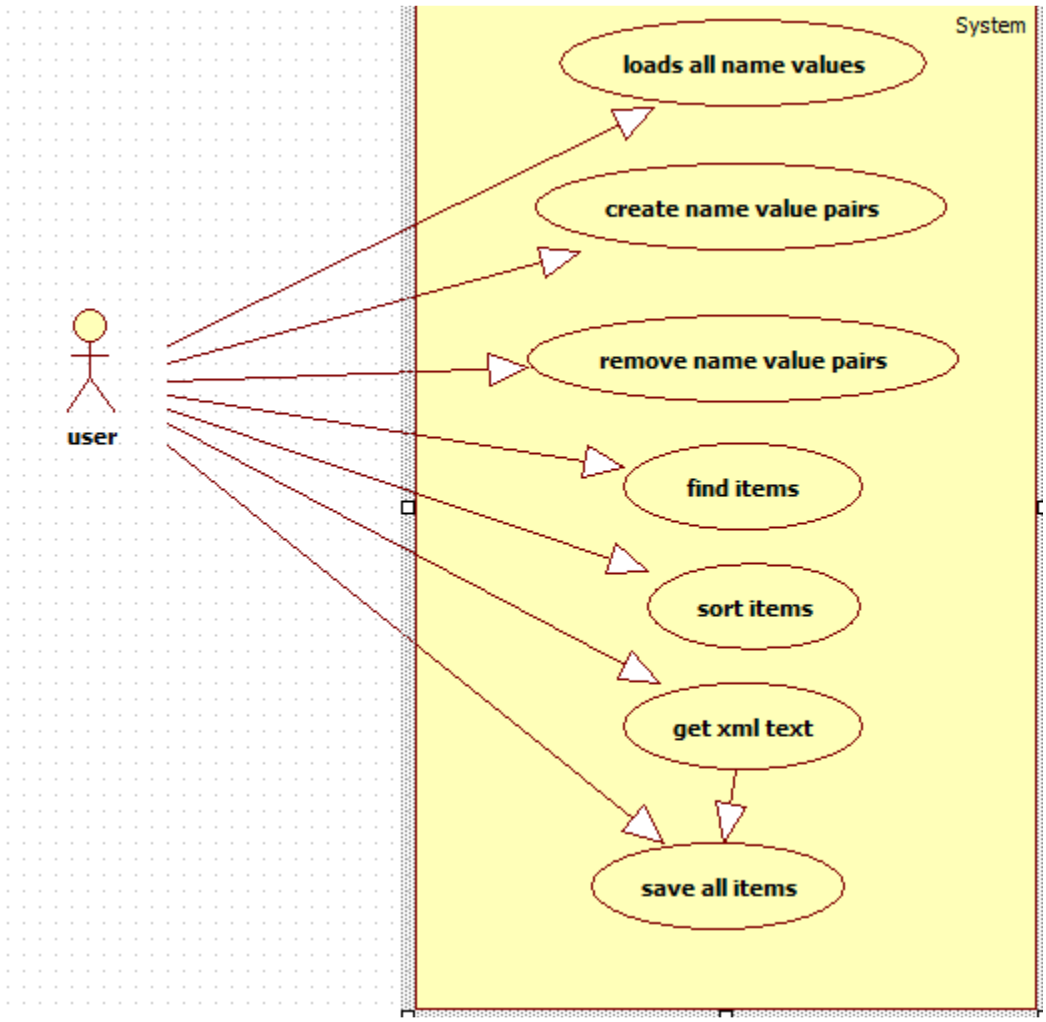
Mathematical Modeling:

Let S be the system that represents the Quick Sort program.
Initially,
S ={ ∅ }
Let,
S ={ I, O, F }
Where,
I – Represents Input set
O – Represents Output set
F – Represents Function set
Input set – I:
Size of input array.
Output set – O:
Sorted input array.
Function set – F:
F = { F1, F2, F3 }
F1 – Represents the main quicksort function.
F1(E1, E2 ..... En) –> { E5, E1 ..... En−3 }
Where,
· Ei: ith element of the input array.
F2 – Represents the partition function.
F2(E1, E2 ..... En) –> { L1, L2 ..... Ln } & { G1, G2 ..... Gn }
Where,
· Li: elements less than pivot.
· Gi: elements grater than pivot.
F3 – Represents the swap function.
F3(E1, E2) –> { S1, S2 }

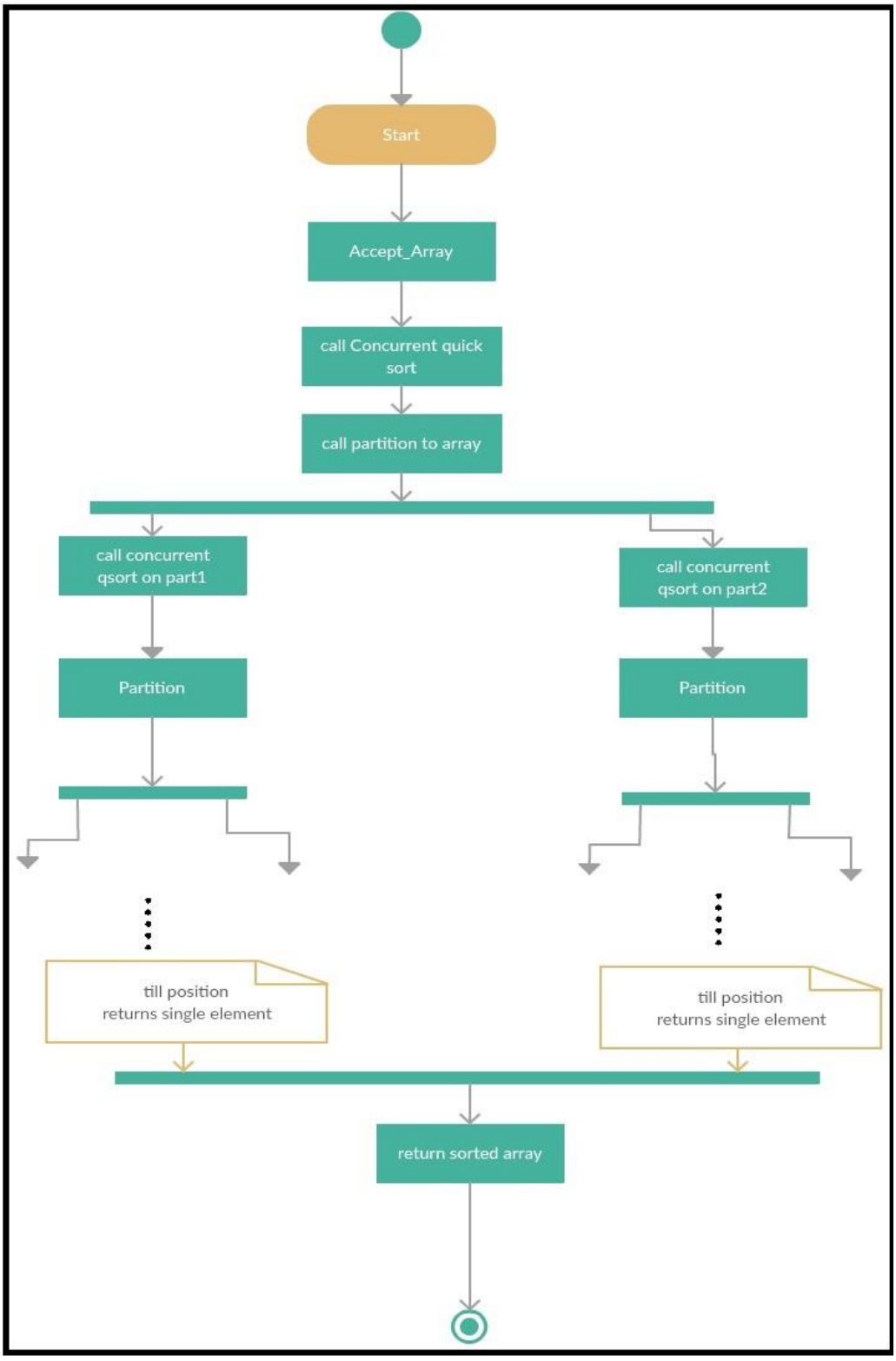
Where,
· Ei: elements to be swapped.
· Si: Swapped elements.
Finally,
S ={ I, O, F }

UMLS:

Use Case Diagram:



Activity Diagram:



Relative efficiency:

Class	Search algorithm
Worst case performance	O(N2)
Best case performance	O(NlogN)
Average case performance	O(NlogN)