Quick Sort

Laasyasree Desu

Fall 2023

1 Introduction

This assignment involved the implementation of another sorting algorithm called quick sort. The algorithm was implemented in two different ways - firstly through an array and then through a linked list. The two algorithms were then bench-marked and compared to see the run time for each of them.

2 Quick Sort

Quick Sort is an efficient sorting algorithm that can be thought of being based on the concept of "divide and conquer". The algorithm functions relatively similar o merge sort where it starts by partitioning the sequence of elements in a particular way. This is based on selected element called "pivot", which can be any element on the sequence but is often chosen as first, last or middle. The sequence is divide into a section of elements smaller than the pivot element and section with elements that are larger. The sections are then sorted through recursion (continuous division into smaller and smaller sections) and the sorted sections are the joined together to for the entire sorted sequence.

3 Array Implementation

The array implementation of the quick sort algorithm is comprised of two main methods. The first method, partition handles the rearrangement of the elements depending on the pivot while the sort method handle the recursive sorting of the array. Apart from this an additional method "swap" was added which handled the swapping of elements and was utilised in the partition method.

3.1 Partition

As mentioned above, the partition method is responsible for rearranging elements within a specific range. The method started off by choosing the pivot element which in this case is the first element. Then two pointers were created, "i" and "j", where "i" traverses the sequence from the left (beginning) and "j"

from the right (end). Afterwards a while loop was created with the condition of looping as long as "i" is less than "j". Within that two more while loops were created, one to increment "i" during traversal as long as it is less than the ending index and the element at "i" is less than the pivot element. The other loop was responsible to decrement "j" while the element was greater than the pivot. Lastly the swap was called if "i" was less than "j" (since they are in wrong places) and the pivot would be updated accordingly and returned.

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

3.2 Sort

This method is responsible for the sorting the array through recursion. The method starts by check that the array has two or more elements through an if-statement that checks that the first index of the array in question is less than the last index. The partition method is the called with the array, first index and last index as parameters. The returned value (the pivot index) is then taken into a new variable. Using this pivot index, the sort method is called recursively for the elements less than or equal to the pivot and elements greater than the pivot as parameters for two different calls of the method. The sub-arrays are then sorted independently until they have one or zero elements left.

```
 if (first < last) {
        int indx = partition(arr, first, last);
        sort(arr, first, indx - 1);
        sort(arr, indx + 1, last);
}
```

4 Linked List Implementation

The linked list implementation of quick sort was relatively similar to the array implementation, it uses the same principles of a partition and sort as well as an append method to join the lists. The method started of by initialising an empty list with a constructor setting head and tail to null. A method "append" was then responsible for adding nodes to the list as well as joining two lists. This method first checks if the list is empty with an if statement. In the case that it it is, the "first" and "last" node references were updated accordingly. In the case that it is not empty the new node would be added to the end of the list and the "last" reference would be updated accordingly, therefore the tail reference removed the need to iterate through the entire list.

```
if (first == null) {
   first = new;
   last = new;}
else {
   last.next = new;
   last = new;
   }
```

4.1 Sort

The sort method similar to the previous version, was responsible for sorting the list through recursion. The again started by verifying that there are two or more elements in the list, otherwise the list would already be sorted. This was done by checking if the first node or the next node were equal to null. In the case that it was not then it moved onto the partitioning section where the first node was set as the pivot and two new empty lists were created to be able be able to make the sub-lists - one for elements greater than the pivot and one for smaller. The node after the pivot was set to "compare" and with a while loop the list was traversed while the "compare" node would, after each increment, compare its value to the pivot to determine which of the sub-lists the node would be put in. The respective sub-lists were also updated to keep track of the new nodes. Once the list was partitioned the method would be called recursively to sort the sub-lists. The lists would then be joined together with the pivot element in its place.

```
Node pivotNode = partition(first, last);
       quickSort(first, pivot.prev);
       quickSort(pivot.next, last);
   }
```

5 Benchmark

Array size	Time taken (µs)
100	6.5
200	10.1
400	22.3
800	56.2
1600	101.8

Table 1: Average time taken to sort an array using quick sort

List size	Time taken (µs)
100	9.7
200	19.3
400	37.2
800	68.4
1600	126.8

Table 2: Average time taken to sort linked list using quick sort.

The implementation of quick sort through arrays and linked list were both bench-marked to measure their respective run-times. The tables above (ref. table 1 and table 2) show relatively similar results, with some slight differences, such as the linked list in general having a slightly longer run-time. The general or average time complexity for both implementations is O(n log(n)) and worst case time complexity is O(n^2). This is due to a number or of reasons, which could also explain the slight differences in run-time for both comparatively. One such reason is how the pivot element is chose, consistently choosing the first or last element can lead to unbalanced partitions (ex. one sub-array/list is significantly larger than the other) that result in a worse run-times. Other factors include access to the elements where arrays provide faster random access or memory overhead where arrays can have lower memory overhead.