Quick Sort

Ayda Ghalkhanbaz

Fall 2023

Introduction

This assignment involves implementing a further sorting method, namely quick sort, and analysing its execution time for sorting datasets with an increasing size. This comparative analysis is done on sorting data in arrays and linked lists. In the later section, the results of benchmarking will be discussed.

Quick Sort

The quick sort is a fast sorting algorithm which has $\mathcal{O}(n \cdot \log(n))$ time complexity in its average and best case scenario. This algorithm utilises a divide-and-conquer approach and includes three main steps: 1.Choosing pivot, 2.Partitioning, 3.Sort recursively.

Choosing a pivot is an important step since it can impact negatively on the time complexity and make it reach $\mathcal{O}(n^2)$ in a worst case scenario. Each of the mentioned steps above is explained in upcoming sections.

Array Implementation

In this task, the QuickSort algorithm is implemented to sort datasets stored in arrays. The implementation includes two static methods *sort()* and *partition()*, where the recursive calls occur in the *sort()* methods while *partition()* method handles selection of the pivot and swapping items.

The *sort()* method takes in three arguments: the array to be sorted, the low index and the high index. If the low and high are equal, it means that the array has only one element and it is assumed as a sorted array. But if it is not the case, then the *partition()* method is called recursively until the whole array is sorted.

The *partition()* method starts by choosing the first element in the array as pivot, a left pointer starting from the low index, and a right pointer starting from the high index. The idea is to move the left and right pointers to search for elements that are in wrong places and should swap their positions.

In other words, the left pointer moves to the right (increments) until it finds an element that is larger than the pivot or it hits the right pointer. In the same way, the right pointer moves to the left (decrements) till it finds an element smaller than pivot or it hits the left pointer. In the case of finding a smaller/larger element than pivot, the elements that left pointer and right pointer are pointing to should change their places. Once this process is done, an if-statement is checked in order to find the right position of the pivot element. The code snippet below demonstrates the *partition()* method implementation.

```
public static int partition(int[] array, int low, int high){
    // first element as pivot
    int pivot = array[low];
    int leftp = low;
    int rightp = high;
    // while the two pointers don't point to the same element
    while(leftp != rightp){
    // while leftp points to smthg smaller than pivot
    // and it hasnt hit the rightp yet
        while(array[leftp] <= pivot && leftp < rightp){
            leftp++;
        }
        while(array[rightp] > pivot && rightp > leftp){
            rightp--;
        }
        // swap the elements that are in the wrong place
        swap(array,leftp,rightp);
    }
    // pivot is in its right place?
    if(array[rightp] < array[low])
        swap(array,low,rightp);

    return rightp;
}
```

Linked List Implementation

In this part of the task, the QuickSort algorithm is implemented for the linked list data structure. Similar to the previous part, the linked list implementation utilises two methods, *sort()* and *partition()*. However the approach in this task is slightly different.

In the *sort()* method, two new lists are created which later in the *partition()* method, will be filled with smaller/larger nodes than pivot. Then, by recursively calling the sort method, each of these lists will be sorted, and at the end, the two small and great lists together with the pivot node will be appended and create a whole sorted linked list. A basic illustration of appending these lists is shown in figure 1.

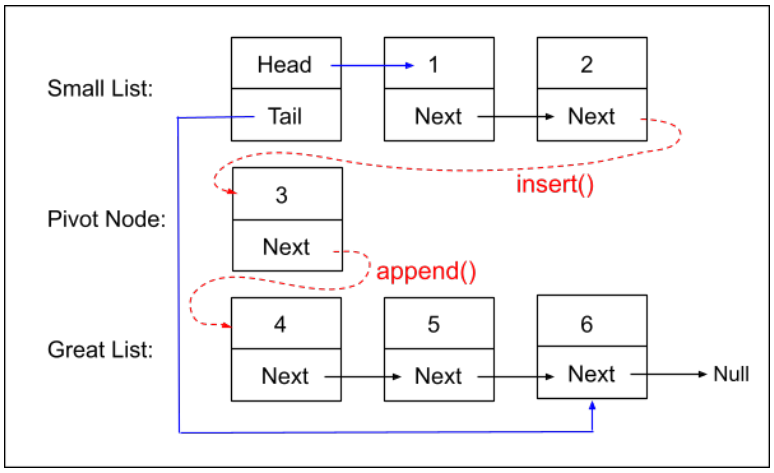

Figure 1: *A basic illustration of appending lists after sorting [PS! insert() method adds a node at the end of a list. append() method adds a list at the end of another]*

The idea of the *partition()* method is similar to the previous task. The first node in the list is selected as the pivot. While traversing the list, each node that has a value smaller than the pivot will be unlinked from the list and inserted to the *smaller* list, and if the node has a greater value than pivot it will be added to the *greater* list. The provided code snippet below illustrates the implementation details of the *partition()* method.

```
public static Node partition(Node first, Node last,
    QuickSortLL smaller, QuickSortLL greater) {

        Node pivot = first;
        Node current = first.next;
        // Ensure the pivot is disconnected from the list
        pivot.next = null;
        while (current != null) {
            Node nextNode = current.next;
            // Disconnect the current node
            current.next = null;
```

```
            if (current.value <= pivot.value) {
                smaller.insert(current);
            } else {
                greater.insert(current);
            }
            current = nextNode;
        }
        return pivot;
    }
```

Result

To benchmark the QuickSort on arrays and linked lists, some datasets with increasing sizes and random values were created. Each of the implementations sorted the same random values 20 times and the best execution time of each has been picked up and presented in table 1.

size	Array	$\frac{array}{nlog(n)}$	Linked List	$\frac{LL}{nlog(n)}$
1000	20	0.002	40	0.005
2000	50	0.003	120	0.008
4000	170	0.005	300	0.009
8000	400	0.005	700	0.009
16000	900	0.005	1800	0.011
32000	1800	0.005	4200	0.012
64000	4100	0.005	12000	0.017
128000	10500	0.007	37000	0.024
256000	23500	0.007	124500	0.040

Table 1: Benchmark results of sorting arrays and linked lists in μs

Discussion

As we can observe from the results presented in table 1, both arrays and linked lists have similar logarithmic behaviour. It is also noteworthy that the constant factor $n \cdot \log(n)$ increases for each doubling of the data size.

However QuickSort on arrays are approximately 2 times faster than QuickSort on linked lists. Although both share the same time complexity, there is an undeniable difference between the execution time of the same operation on arrays vs linked lists. One main reason for this significant difference, is the memory access pattern. While arrays take advantage of CPU

cache and have access to its contiguous memory, each node in the linked lists points to the next node in the memory, this means more cache misses and slower performance.

In summary, sorting arrays and linked lists may have the same time complexity in theory, but in the real world, arrays often outperform linked lists. Utilising a more organised memory structure and the CPU cache, makes it easier and faster for arrays to find and retrieve data. In contrast, linked lists resemble a scattered collection of data, where each one points to the next item, leading to a slower performance.