Quick sort an array and a linked list

Fatima Mohammad Ali

Fall 7/10/2023

Introduction

In this assignment, the sorting algorithm *quick sort* was implemented for arrays and singly linked lists. The main approach of the algorithm is to sort a sequence by dividing it into two parts, *smaller* and *larger*, in regards to a *pivot* element. After sorting each sub-part recursively, they are to be combined to one sorted entity and so, a sequence was sorted. Since the implementation differs for arrays respective linked list, the methods and their benchmark results were analysed in this report to form a deeper understanding of the data structure.

Array quick sort

Quick sort for arrays was done in a `class ArrayQuickSort` that included the static methods `void sort(int[] arr, int min, int max)`, `void swap(int[] array, int i, int j)`, and `int partition(int[] arr, int min, int max)`. The main role of the `sort` method was to use recursion to divide and sort subarrays by using the `partition` method, which was implemented as follows:

```java
public static int partition(int[] arr, int min, int max){
   int i = min;
   int j = max;
   int pivot = arr[min];   //pivot el. is the first el. in the array

   while(i < j){   //as long as i is smaller than j
       while(arr[j] > pivot)  //check the larger items
           j--;

       while(arr[i] <= pivot && i < j) //check the smaller items
           i++;

       if(i < j)   //check if i < j does not overlap
```

```
            swap(arr, i, j);    //swap misplaced items at i and j
        }
        swap(arr, min, j);  //swap pivot el. at index min with the el. at index j

        return j;
    }
```

The `swap` method simply swapped elements at the given indices of the array. After implementing `partition`, the sorting in `sort` was done by calling `partition(arr, min, max)` and saving the return value (index j) in a variable *mid*. `sort` would then recursively call itself to sort the smaller and larger sub-arrays. It would sort items from *min* to *mid-1*, and *mid+1* to *max*. This recursion would carry on as long as *min* was smaller than *max*. At last, a sorted sequence was achieved.

Linked list quick sort

Quick sort was implemented for singly linked lists with a slightly different approach than arrays. Per usual, a node structure was created in a `class ListQuickSort` where each node held a *next* pointer and a *value*. The class kept track of the *first* and *last* pointers and several side-methods to build the list. For instance, the `add(int item)` method added an integer value at the end of the list and `insert(Node el)` added a node. Both methods kept track of the lastly added node by making the `last` pointer point to it. A `void printList()` method was also used to constantly check if the implemented code worked as it should. Furthermore, a `void appendList(ListQuickSort b)` method that appends a given list to another was also used.

The sorting procedure was done recursively in a `void sort(ListQuickSort list)` method that called a `Node partition(ListQuickSort smaller, ListQuickSort larger, Node min)` method for the partition procedure. Instead of swapping elements as done with arrays, the partitioning in this task was done by re-linking nodes from the main list to two empty lists, *smaller* and *larger*. These empty lists were created in the `sort` method and sent as arguments to the `partition` method (`Node pivot = partition(smaller, larger, list.first)`). In the `partition` method, the pivot element was chosen as the first node in the list, and the rest of the nodes were compared to that pivot element.

The `partition` method was done as follows:

```
public static Node partition(ListQuickSort smaller,
                                           ListQuickSort larger, Node min){

    Node pivot = min;   //pivot element is the first Node in the list
    Node cur = min.next;   //start at the next element to the pivot

    while(cur != null){ //traverse the list
        Node nxt = cur.next; //unlink current node
        cur.next = null;    //unlink -||-

        //insert to the smaller or larger list
        //depending on if the current element is <= or > than pivot
        if(cur.value <= pivot.value)
            smaller.insert(cur);
        else
            larger.insert(cur);

        cur = nxt;  //unlink -||-
    }
    return pivot;
}
```

When the partitioning was done, *smaller* and *larger* would respectively contain all the smaller respective larger values compared to the pivot element. Each sorting of the sub-list was done recursively since `sort` called itself (`sort(smaller)` and `sort(larger)`). The recursion was done as long as the list's `first` pointer (and its `first.next`) was not null. Lastly, the *smaller* list was appended to the `list`, the pivot node was inserted at the end of the `list`, and then the *larger* list was appended to the `list`. Therefore, what was left was now a sorted linked list.

Benchmark

n	arr QS [µs]	arr QS/n*log(n)	list QS [µs]	list QS/n*log(n)
1000	14	2.0	61	8.8
2000	31	2.0	140	9.0
4000	150	4.4	320	9.8
8000	380	5.2	610	8.4
16000	850	5.5	1400	8.9
32000	1800	5.5	3200	9.7
64000	3900	5.5	7600	11
128000	8300	5.5	17000	12

Table 1: The same random elements/keys were given as a sequence to the arrays and lists. Each timestamp is the minimum value of 100 tries after running a warmup. The values are rounded to two sign. figures.

Discussion

At worst case, the pivot element is either the smallest or largest value in the sequence. This would lead to one of the sub-sequences being empty and the other being full of all elements. If this repeatedly occurs in every partition, then each recursive call would sort a one-less-element sequence. This case of sorting an already sorted sequence has the big-O notation $O(n^2))$.

In more balanced cases where the partition produces almost equally as big sub-sequences, the recursion would constantly divide them in half. This would require only $log(n)$ recursion calls, and the partition work during each call is done in $O(n)$. This leads to the time complexity being $O(n * log(n))$ for average and best cases.

The timestamps shown in table 1 confirm the expected average-case time complexity. As seen in columns 3 and 5, dividing the array and list timestamps with $n * log(n)$ shows logarithmic behaviour since there is a slight increase in the factor for each doubling of size n. Each row in columns 2 and 4 also *more than doubles* in size for each doubling of n. This only further proves that the time complexity is $n*log(n)$ since the timestamps should be proportional to n but also *a little more* since they are multiplied by $log(n)$.

It could also be noted that quick sorting an array is done almost double as fast as quick sorting a list (notice the ratio between values in column 2 and 4). One reason could be that arrays are favoured by the cache since array elements are stored closer in memory and are accessed speedily. Nodes in lists however, are scattered all over memory and are accessed by following pointers, which is time-consuming.

One question that could be discussed is how to improve the execution time by selecting the right pivot element. Choosing the right pivot element could be detrimental to achieve the best execution time since it directly affects how a sequence is partitioned. In the best case, reaching two equal halves as sub-sequences would result in the best case time complexity. The question is, how should the ideal pivot element be chosen? It could be the median, median of three, a randomized element, or the first or last values of a sequence. Theoretically, a median would be a perfect choice, only if it appeared out of thin air since sorting a sequence only to find the middle value is costly (not worth it). If the sequence is sorted or nearly sorted, choosing the first or last elements as pivot could also prove costly. Perhaps choosing a randomized element could be ideal depending on how it is produced. Yet, as long as not an already sorted sequence is to be sorted, perhaps the best and easiest choice is to use the first/last element as pivot, which was done is this assignment.