Assignment C: Quick sort

Mahad Ahmed

Fall 2022

Introduction

In this assignment we implemented a new algorithm called *quick sort.* Quick sort is a sorting algorithm. By now we now how important that sorting data can be and how different sorting algorithms differ in their run time. By it's name we already now that it is fast but how fast is it? That's a question I will answer in this report. Just to make things interesting and maybe a bit more difficult we have implemented quick sort on both an array and for a linked list. What is the run time difference in these two implementation of this algorithm?

Quick sort algorithm

To explain the quick sort algorithm is rather trivial. But it's implementation has a few tricks up it's sleeve to make our life a bit more difficult. The quick sort algorithm works like this; beginning with choosing a pivot element from a given array. The the rest of the elements in the array is compared to the pivot element. We compare it as such that to the left of the pivot there are elements that are smaller and to the right of the pivot elements that are smaller. When this is done we do the same thing by choosing a new pivot from the two "stacks" of values then we redo the same sorting. This is why this algorithm is called a divide and conquer algorithm. Because how it breaks down the given data into smaller data points and then restores it.

Implementation of quick sort on an array

Quick sort is a recursive algorithm which means it calls itself in inside the method. I have two methods, one called *sort* which sorts are array and keep calling itself. The second method is called *partition.* This is the key process in our quick sort. I will explain each method more closely down below.

Partition

What this partition does is divide and conquer. This method takes in an array, a first pointer and a last pointer. What I did first was to choose a pivot. A pivot can be chosen in variety of ways. I chose the first element in the array as a pivot. Then set a counter to after the last element in the array called *i*. Then also another counter in the for loop starting at the last position called *j*. What the for loop does is check if the pivot is smaller than the element j is pointing at, if it is then we are going decrement i and then to change the element i is pointing to with the element j i pointing to. So first check if the pivot element is smaller or bigger, then if it is smaller do a swap. *i* is only decremented before a swap. But if it isn't we just continue.

```
public static int partition (int[] array, int first, int last){
   int i = last + 1;
   int pivot = array[first];

   for (int j = last; j >= 0; j--){
     if (array[j] > pivot){
       i--;
       int temp = array[i];
       array[i] = array[j];
       array[j] = temp;
     }
   }
   i--;
   int temp = array[i];
   array[i] = array[first];
   array[first] = temp;

   return i;
 }
```

Sort method

In this method what we are doing is really just calling function over and over until we get a sorted array. What is important is to not fall out of the array so our first if statement checks if we are in bounds. Then we just call the partition method and divide and conquer. This is called recursive programming.

public static void sort (int[] array, int first, int last) if (first ¿= last) return;

int middle = partition(array, first, last);

sort(array, first, middle - 1); sort(array, middle + 1, last);

Linked list implementation

Quick sort on linked list works in the same way. But the difference is instead of swapping data you instead change references/pointers. In the same way you choose a pivot node and then call partition. The partition method traverses in the current list and moves them around like before. The nodes value that are greater than the pivot node value goes to after the tail but smaller values keep their positions. Then we repeat the process for the left side and right side of the linked list.

This was also done recursively using a sort method. Works the same as for the array.

Code, linked list implementation

Here is the code for this task. I explained it in the paragraph above.

Partition

```java
public static LinkedList.Node partitionLinked (LinkedList.Node
    first, LinkedList.Node last){
  if (first == null || last == null || first == last)
    return null;

  LinkedList.Node prev = first;
  LinkedList.Node current = first;
  int pivot = last.value;

  while (first != last){
    if (first.value < pivot){
      prev = current;
      int temp = current.value;
      current.value = first.value;
      first.value = temp;
      current = current.next;
    }
    first = first.next;
  }
  int temp = current.value;
  current.value = pivot;
  last.value = temp;

  return prev;
}
```

Sort

```
public static void sortLinked (LinkedList.Node first,
    LinkedList.Node last){
  if (first == null || last == null || first == last)
    return;

  LinkedList.Node middle = partitionLinked(first, last);

  sortLinked(first, middle);
  sortLinked(middle.next, last);
}
```

Benchmarks

Elements n	Array	Linked list
100	5300	6500
200	12700	14100
400	25320	32100
800	71200	70100
1600	121900	145100

Table 1: Benchmark, Quick sort: Array vs Linked list

Discussion on benchmarks and time complexity

In the benchmarks we can see that our time is consistent with O(nlogn) If you plot the graph for both of our data structures when you apply the quick sort algorithm. This is in the best case and average case. We choose the first element as the pivot for our array implementation and the last node for the linked list implementation. Problem with this is if we already have a sorted array then if we choose the last/the first element as the pivot, we will then only have one group of elements after the first partition, the left or the right group. Then if we partition again the same thing. will happen again. We will partition n-1 times. This is worst case scenario and will give us a complexity with O(n^2). But if you choose a random as a pivot or the median of three elements then you will ensure a O(nlogn) time complexity.