Quick sort an array and a linked list

Isak Wilkens TIDAB

Fall 2022

Introduction

This is a report on the second higher grade assignment "Quick sort an array and a linked list" on the course "ID2021" TIDAB, KTH. The assignment is a continuation on the topic of sorting where the concept is expanded on through implementing the quick sort algorithm on both an array and a linked list. Quick sort works similarly to merge sort and also make use of recursive properties in the code. Both are what is called "divide and conquer" algorithm where a problem is broken down in to smaller and smaller parts to where it is easily solved. The algorithm chooses a pivot point in the algorithm and then sorts all the elements in the list smaller than the pivot to its left and all the algorithm larger than the pivot to its right. Since this is done recursively the elements that are to be sorted are divided into smaller and smaller lists of elements. A list with only one item is already sorted but two elements can be sorted this way up to an arbitrary number of elements.

Quick sort with an array

To be able to use this algorithm on an array we need access to the array itself as well as the index of the first and last item. We need these two indexes to be able to use the method recursively since when we make a recursive call the indexes will be on parts of the array that we intend to divide and conquer. To get an average best spread the pivot point is chosen at random within the range of the elements between our input indexes. The next step is to partition the elements in the list so that the elements larger than the pivot is to the right and the elements smaller than the pivot to the left. Two pointers are created that start from the left and the right side respectively of the part of the array that is to the left of the pivot. The left pointer will be moved rightwards until we find a value larger than the pivot. The right pointer will be moved leftwards until we find an element smaller than the pivot. When these are found the position of these two elements is swapped, this process is repeated until the pointers are pointing to the same element.

When this happens the element that is being pointed to is swapped with the pivot thus creating a list where the smaller elements are to the left of the pivot and the bigger ones to the right. The sub arrays the are to the left and right of the pivot can now be recursively sorted. The code for this partitioning operation is given below:

```
private static int partition(int[] array, int low, int high, int pivot) {
    int lPointer = low;
    int rPointer = high;

    while (lPointer < rPointer){
        while (array[lPointer] <= pivot && lPointer < rPointer){
            lPointer++;
        }
        while (array[rPointer] >= pivot && lPointer < rPointer){
            rPointer--;
        }
        swap(array, lPointer, rPointer);
    }

    if (array[lPointer] > array[high]){
        swap(array, lPointer, high);
    }
    else{
        lPointer = high;
    }
    return lPointer;
}
```

Table 1 shows bench marking of the quick sort function. Since the array is continuously divided the length of the "binary tree" that is formed will be log(n) and the number of operations per branch is equal to n so the time complexity of quick sort is $n * log(n)$. to get more accuracy more tests could have been done but after an array size of 10 000 elements the run time gets very long. Figure 1 shows the results put into a graph.

Repetitions(n)	**Run-time(ns)**
5	$1 * 10^3$
10	$1,7 * 10^3$
50	$6,5 * 10^3$
100	$1 * 10^4$
500	$6 * 10^4$
1000	$1 * 10^5$
10000	$1,3 * 10^6$

Table 1: Mesurements of the run-time as a function of array size for quick sort. Run-time in nano seconds.

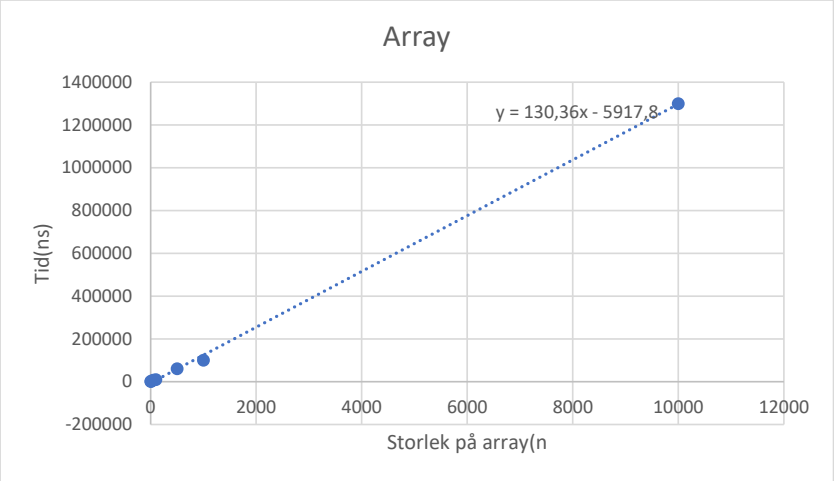

Figure 1: Measurements of quick sort with an array. Run-time in ns.

Quick sort with a linked list

For the second part of the assignment the quick sort algorithm will be implemented using a single linked list. Similarly to the queue assignment we have a nested class for nodes whilst the list class only has one node that we call head. The add function finds the last element in the list through

iterating through the nodes and then checking for when a nodes tail is null. The code for the add operation is given below:

```
  void add(int item){
        if (head == null){
            head = new Node(item);
            return;
        }

        Node curr = head;
        while (curr.tail != null){
            curr = curr.tail;
        }

        Node temp = new Node (item);
        curr.tail = temp;
    }
```

The partition operation is called with a reference to the first and the last nodes of the list and does not break any links in the original list to make a new one. The partition code then picks the last element as the pivot element to then sort all element from the first one in the list until the last. This is done through swapping their item values, the last element that was used as in a swap will then be put as the new pivot. We then return "recursive pivot" which is the element before this pivot. Through calling this function recursively we will and divide and conquer the elements in the linked list. If our pivot is equal to the first element we instead will take the element after this as the pivot as a special case. If our pivot will end between the divided lists we will instead start on the element that follows, since we only have the value for "recursive pivot" which is one element before pivot we will here jump two steps. The code for the partition operation is given below:

```
public Node partition(Node first, Node last){
    if (first == null || last == null || first==last){
        return first;
    }

    Node recursive_pivot = first;
    Node curr = first;
    int pivot = last.item;
    while (first != last){
        if (first.item < pivot){
            //curr, vilken nod vi ändrade senast
            recursive_pivot = curr;
```

```
            int temp = curr.item;
            curr.item = first.item;
            first.item = temp;
            curr = curr.tail;
        }
        first = first.tail;
    }
    int temp = curr.item;
    curr.item = pivot;
    last.item = temp;

    // Vi skickar tillbaks pekare till noden som är ett steg
    // innan pivot
    return recursive_pivot;
}
```

Table 2 shows the results from benchmarking the linked list implementaion from quick sort. Figure 2 shows these values put into a graph.

Repetitions(n)	**Run-time(ns)**
5	$2 * 10^2$
10	$4,5 * 10^2$
50	$2,5 * 10^3$
100	$4,5 * 10^3$
500	$3,5 * 10^4$
1000	$7,5 * 10^4$
10000	$8 * 10^5$

Table 2: Mesurements of the run-time as a function of array size for quick sort. Run-time in nano seconds.

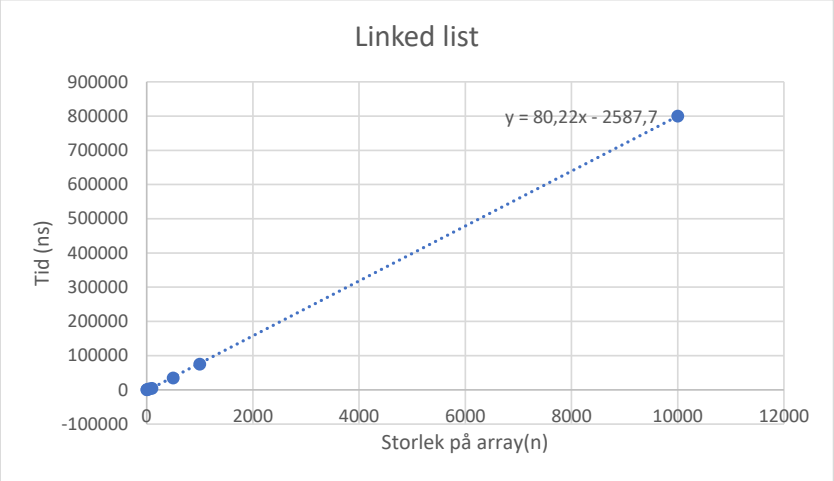

Figure 2: Measurements of quick sort with linked list. Run-time in ns.

Conclusion

In conclusion quick sort is a fast sorting algorithm in comparison to the bench marks done on merge sort it was on average 10 times faster. The implementation with an array was easier to grasp and implement than the linked list one. The linked list implementation was very hard to grasp and took much discussion with other students and trial and error to finally get working. The results from our graphs above show something that looks fairly linear that confirms the time complexity of $n * log(n)$. For the array implementation we get our worst case scenario when the list is already sorted. Since the algorithm will pick the first element as the pivot it will partition the array in a way so that every iteration of one partition will have no elements in it. This will lead to the worst case scenario of $(O(n^2)$. The linked list implementation that has two recursive calls the worst case will occur when we always pick the smallest or greatest element as pivot. The partitioning function will then repeatedly create an empty list with $n - 1$ elements which will also lead to $(O(n^2)$. On average the linked list implementation performed better, this could be since we can make use of recursive properties better for the remainder in the list whereas in the divide

and conquer algorithm for the array we had to handle both halves separately.