Quick Sort: Array vs Linked List

Dhruba Das

Spring Fall 2023

Introduction

This assignment was quite fun and interesting to do, as I had never implemented or used the quick sort algorithm before. There are mainly three problems in this assignment: implementing quick sort for an array, for a linked list, and running benchmarks too compare the costs of each. The Quick sort algorithm picks a pivot element, and partitions an array into two parts: one containing all elements smaller than the pivot, and one containing all those which are larger; this is done recursively multiple times until the entire array is sorted in an ascending order.

QuickList

Although I knew I would be working with the Integer datatype, I was told it would be better to implement a generic Linked List which would be able to handle all primitive datatypes. Below, I will go through all the methods which were implemented and used while creating the QuickList class; I haven't included the node structure this time as it's the same as that of the linked list implemented during Queues assignment.

QuickList constructor

This constructor creates a linked list from a given array. It has properties first and last, which are pointers pointing to the first and the last nodes of the linked list. Below is the code snippet:

```
public QuickList(T[] arr){
        if(arr.length == 0){
            this.first = null;
            this.last = null;
        }
        else{
            this.first = new Node(arr[0], null);
```

```
            Node cur = this.first;
            for(int i = 1; i < arr.length; i++){
                cur.next = new Node(arr[i], null);
                cur = cur.next;
            }
            this.last = cur;
        }
    }
```

Append method

This method appends another linked list to the end of the current linked list we are looking at. Below is the code snippet:

```
public void append(QuickList<T> tail){
        if(tail != null){
            if(this.last != null)
              this.last.next = tail.first;
            else
              this.first = tail.first;
            if(tail.last != null)
              this.last = tail.last;
        }
    }
```

Prepend method

This method appends another linked list to the beginning of the current linked list we are looking at. Below is the code snippet:

```
public void prepend(QuickList<T> front){
        if(front != null){
            if(front.last != null)
              front.last.next = this.first;
            if(this.last == null)
              this.last = front.last;
            if(front.first != null)
              this.first = front.first;

        }
    }
```

Cons method

This method adds a node, NOT AN ITEM, to the beginning of the linked list. Below is the code snippet, including comments:

```
public void cons(Node nd){
        if(this.last == null) // Check if list is empty
          this.last = nd;
        nd.next = this.first; //If not, make nd the first node in list
        this.first = nd;
    }
```

Sort method

Lastly, is the sort method, which sorts a given linked list according to the quick sort algorithm. Below is the code snippet:

```
public void sort(){
        if(this.first == null || this.first.next == null)
          return;

        QuickList<T> smaller = new QuickList<T>();
        QuickList<T> larger = new QuickList<T>();

        Node pivot = this.first;
        Node cur = pivot.next;
        pivot.next = null;

        //pivot is now the only element in the linked list
        this.last = pivot;

        T p = pivot.item;

        while(cur != null){
            Node nxt = cur.next;
            if(p.compareTo(cur.item) > 0) {
                smaller.cons(cur);
            } else {
                larger.cons(cur);
            }
            cur = nxt;
        }
        smaller.sort();
        larger.sort();
        //this holds pivot
        this.append(larger);
        this.prepend(smaller);
    }
```

QuickArray

This section outlines the array implementation of the quick sort algorithm, and goes through the methods which have been implemented in the QuickArray class.

Partition method

This method picks a pivot element in an array (for me, it's the last element) and partitions the array into two parts: one with all elements smaller than the pivot (to the left) , and one with all elements larger than or equal to the pivot element. It then, swaps the end element with the element which is in the index where the pivot element should be. It then returns the position of the pivot element. Below is the code snippet:

```
public static int partition(Integer[] array, int start, int end){
        int pivot = array[end];
        int i = start - 1;

        for(int j = start; j <= end - 1; j++){
            if(array[j] < pivot){
                i++;
                swap(array, i, j);
            }
        }
        i++;
        swap(array, i, end);
        return i;
    }
```

Sort method

This method recursively sorts all the elements for an array, within the given indices. Below is the code snippet:

```
public static void sort(Integer[] array, int start, int end){
        if(end <= start)
          return;

        int pivot = partition(array, start, end);
        sort(array, start, pivot - 1);
        sort(array, pivot + 1, end);

    }
```

Benchmarks

Below is the difference in performance (i.e costs) between implementing quick sort for an array and quick sort for a linked list:

Size(n)	**Array (ms)**	**Linked List (ms)**
100	3	4
200	4	7
400	8	10
800	16	24
1600	52	73
3200	189	194
6400	422	477
12800	880	1124
25600	1825	2806

Table 1: (For each size, a thousand tries are done; min. time taken; time in microseconds)

As we can see, the linked list becomes significantly slower than the array implementation, as the size of the array grows. Although both have a time complexity of O($nlog(n)$), it's noted than they are far from ideal. The linked list is significantly slower as, in sort, two new linked lists have to be created and then merged.