Problem Set 13 - Solution

Sorting

---

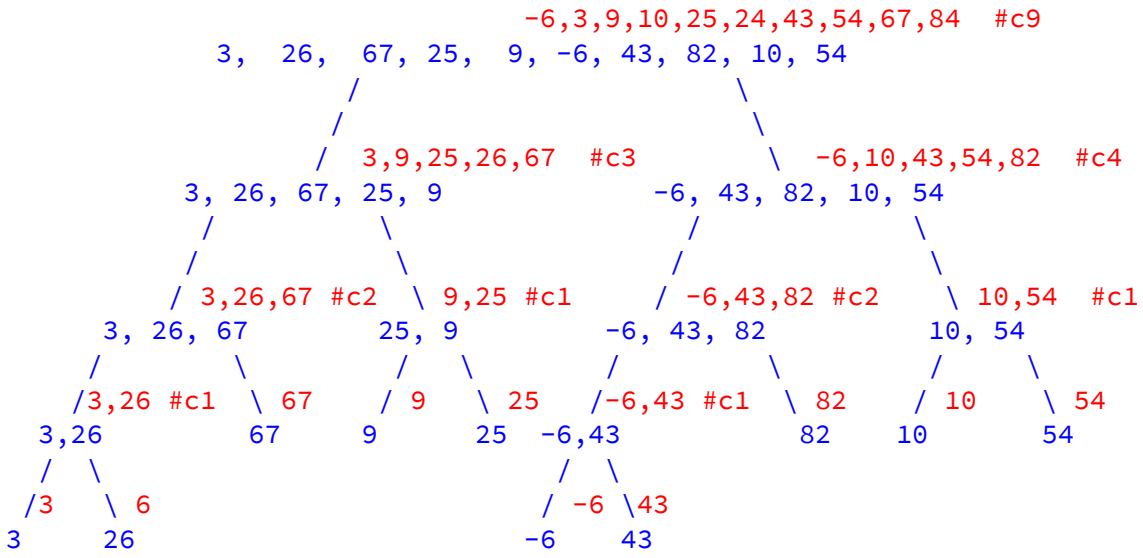
1. Trace the mergesort algorithm on the following list:

```
3, 26, 67, 25, 9, -6, 43, 82, 10, 54
```

Show the resulting recursion tree, with the to-be-sorted original and sub-lists at each node, and the number comparions for each merge. (Assume that if there is an odd number of entries in an array, the left part has one more entry than the right after the split.)

SOLUTION

The original and sorted result indicated above and below:



---

2. Mergesort works well on linked lists since it doesn't need any extra space. Given the following linked list node class:

```
public class LLNode<T extends Comparable<T>>  {
   public T info;
   public LLNode<T> next;
   ...
}
```

complete the following method to "split" the linked list in half:

```
// splits the given list in half such that the return value is
// a reference to the first node of the second half. Also, the
// "next" field of the last node in the first half is set to null.
static <T extends Comparable<T> LLNode<T> split(LLNode<T> list) {
   // COMPLETE THIS METHOD
}
```

SOLUTION

```
static <T extends Comparable<T> LLNode<T> split(LLNode<T> list) {
   if (list == null) return null;

   int size=0;
   LLNode<T> ptr;
   for (ptr=list; ptr != null; ptr = ptr.next) {
       size++;
   }
   size = (size+1)/2; ptr=list;
   while (size > 1) {
      ptr = ptr.next;
      size--;
   }
   LLNode<T> second = ptr.next;
   ptr.next = null;
   return second;
}
```

---

3. In Problem Set 3, #6, you saw how to merge two sorted linked lists of integers, into a single sorted list without duplicates. Here is the header of a modified version of that method, that merges lists of `Comparable` objects (not just `int`s), while preserving duplicates, if any. Complete this method using recursion. Your method should recycle the nodes in the original lists (no new nodes should be created).

```
// merge the lists l1 and l2 into a single linked list, whose
// first node is referenced by the return value - no additional
// linked list nodes are used
static <T extends Comparable<T>> LLNode<T> merge(LLNode<T> l1, LLNode<T> l2) {
   // COMPLETE METHOD USING RECURSION, NO NEW NODES TO BE CREATED
}
```

Using this merge solution, and the solution to the split in the previous problem, complete the mergesort implementation:

```
// Sorts the input linked list using mergesort, and returns the front of
// the sorted linked list. DOES NOT CREATE ANY ADDITIONAL NODES.
public static <T extends Comparable<T>
LLNode<T> mergesort(LLNode<T> list) {
   // COMPLETE THIS METHOD
}
```

SOLUTION

```
// merge the lists l1 and l2 into a single linked list, whose
// first node is referenced by the return value - no additional
// linked list nodes are used
static <T extends Comparable<T>> LLNode<T> merge(LLNode<T> l1, LLNode<T> l2) {
   // COMPLETE METHOD USING RECURSION, NO NEW NODES TO BE CREATED
   if (l1 == null) { return l2; }
   if (l2 == null) { return l1; }
   if (l1.info.compareTo(l2.info) < 0) {
      l1.next = merge(l1.next, l2);
      return l1;
   } else {
      l2.next = merge(l1, l2.next);
      return l2;
   }
}
```

```
public static <T extends Comparable<T>
LLNode<T> mergesort(LLNode<T> list) {
    // empty or single node list
    if (list == null || list.next == null) {
       return list;
    }
    // split
    LLNode<T> second = split(list);

    // recursive calls
    list = mergesort(list);
    second = mergesort(second);

    // merge
    return merge(first, second);
}
```

---

4. Trace the quicksort algorithm on the following array:

```
3, 26, 67, 25, 9, -6, 43, 82, 10, 54
```

Use the median of the first, middle, and last entries as the pivot to split a subarray. (If a subarray has fewer than 3 entries, use the first as the pivot.) Show the quicksort tree and the number of comparisons at each split.
SOLUTION

```
      [3  26  67  25  9  -6  43  82  10  54]   median(3,9,54)=9, swap(9,3)
                    split
                     | pivot=9, # of comp = 9
                     V
      [3  -6]         9        [25  67  26  43  82  10  54]
        split                              split
          | piv=3,#c = 1                     | piv=43,#c = 6
          V                                  V
  [-6]  3                  [25  10  26]    43     [82  67  54]
                               split                 split
                                 | piv=25,#c=2         | piv=67,#c=2
                                 V                     V
                           [10]  25   [26]       [54]  67  [82]
```

Total number of comparisons = 20

---

5. A *stable* sorting algorithm is one which preserves the order of duplicate elements when sorted. For instance, if the following (key,value) pairs are sorted on the keys:

```
(3,sun)  (2,mars)  (4,moon)  (3,venus)
```

then the output of a stable sorting algorithm would be:

```
(2,mars)  (3,sun)  (3,venus)  (4,moon)
```

Notice that `(3,sun)` comes before `(3,venus)`, preserving the order of the input for elements that have the same key (i.e. 3), hence *stable*.

However, if the output is this:

```
(2,mars)  (3,venus)  (3,sun)  (4,moon)
```

then the sorting algorithm is not stable since it does not preserve the input order of `(3,sun)` before `(3,venus)`.

For each of insertion sort, mergesort, and quicksort, tell whether the algorithm is stable or not.

SOLUTION

- Insertion sort is a stable sort.
- Mergesort is a stable sort.
- Quicksort is not a stable sort.

---

6. Given the following input array:

```
3, 26, 67, 25, 9, -6, 43, 82, 10, 54
```

1. Trace the linear time build-heap algorithm on this array, to build a max-heap. How many comparisons did it take to build the heap?
2. Starting with this max-heap, show how the array is then sorted by repeatedly moving the maximum entry to the end and applying sift-down to restore the (remaining) heap. How many comparisons did it take to sort the heap?

SOLUTION

1.

Array	Sift Down	Comparisons
3 26 67 25 9 -6 43 82 10 54	9	1
3 26 67 25 54 -6 43 82 10 9	25	2
3 26 67 82 54 -6 43 25 10 9	67	2
3 26 67 82 54 -6 43 25 10 9	26	4
3 82 67 26 54 -6 43 25 10 9	3	5
82 54 67 26 9 -6 43 25 10 3	done	

2.

Array	Sift Down	Comparisons
82 54 67 26 9 -6 43 25 10 3		
swap(82,3)		
3 54 67 26 9 -6 43 25 10 82	3	4
67 54 43 26 9 -6 3 25 10 82		
swap(67,10)		
10 54 43 26 9 -6 3 25 67 82	10	6
54 26 43 25 9 -6 3 10 67 82		
swap(54,10)		
10 26 43 25 9 -6 3 54 67 82	10	4
43 26 10 25 9 -6 3 54 67 82		
swap(43,3)		
3 26 10 25 9 -6 43 54 67 82	3	4

```
26   25   10   3   9   -6   43   54   67   82
                swap(26,-6)
-6   25   10   3   9   26   43   54   67   82          -6            4


25   9   10   3   -6   26   43   54   67   82
                swap(25,-6)
-6   9   10   3   25   26   43   54   67   82          -6            2


10   9   -6   3   25   26   43   54   67   82
                swap(10,3)
3   9   -6   10   25   26   43   54   67   82           3            2


9   3   -6   10   25   26   43   54   67   82
                swap(9,-6)
-6   3   9   10   25   26   43   54   67   82          -6            1


3   -6   9   10   25   26   43   54   67   82
                swap(3,-6)
-6   3   9   10   25   26   43   54   67   82         done
```