Project 3: Sorting Algorithms and their "Big-Oh"

100 points

Posted: 11/14/2014

Due: 12/12/2014

Last day of Project advising: 12/08/2014

Objective

This project offers experience of implementing
--- sorting algorithms for arrays
--- sorting algorithms for linked lists
--- measuring and comparing algorithm performance for large data structures

ABET Program Learning Outcomes:

- The ability to recognize the need for data structures and choose the appropriate data structure (b,c,i,j)
- The ability to evaluate the performance of an algorithm (b,c,i)
- The ability to implement and use linked lists (a,b,c,i,j)
- The ability to implement binary trees, and heaps (a,b,c,i,j)
- The ability to design and implement algorithms for various sorting and searching problems (a,b,c,k,i,j)

The Problem

The programming problem in this project is to implement and test several sorting algorithms applied for arrays and linked lists of various sizes. In each case the structure stores integer numbers, using generic types is not necessary.
The goal of these applications is to collect experimental data on the performance of the applied algorithms. The observed data are to be compared to each other and to the theoretical Big-Oh estimates.
The required algorithms are

- insertion sort of arrays
- insertion sort of linked lists
- merge sort of arrays
- merge sort of linked lists
- heap sort of arrays

Analysis and Design

Input:

- Integer values randomly selected from the range 0 – 12,000,000 (this same range is used for all structure when they compared to each other)
- The size of the structure (varies between 20 and 10,240,000)
- An unsorted structure of given size such that it stores the randomly selected integer values

Output:

- The sorted structure. For large structures of size >200 only fifty elements evenly distributed over the structure shall be printed to the console
- An estimate of running time of the applied algorithms.

For this project you have to define three classes

1. Class **`Sortings`**

This class is a utility class that contains the implementations of the required algorithms. Each method is static. The class has no fields.
The required method headings are as follows:

```
public static void insertionSort( int[]data )

public static Node insertionSort( Node head )

public static void mergeSort( int[] data, int first, int n )

public static Node mergeSort( Node head )

public static void heapSort( int[] data )
```

An array to be sorted is passed as parameter **data** to the relevant methods. The recursive **mergeSort** method needs additional parameters to specify the range to be sorted:
`data[first] ... data[first+n-1 ]`.

Sorting methods for linked lists receive the head reference of the unsorted list as parameter and return the head reference of the sorted list.
Note that array sorters are void, they do not instantiate another array.
Private helper methods can be added to the **`Sortings`** class as necessary. They can be particularly useful for the merge sort algorithm, but also recommendable for the linked insertion sort and the heap sort as well.

2. Class **Node**

You are familiar with this class used in several previous assignments to represent the nodes of a linked list. Sorting applications do not use the tail reference, lists are represented by their head. You may re-use one of your previous implementations, but for the current applications you may conveniently re-write the class, since beside the constructor you only need the toString( ) method for displaying test results. However, you cannot print millions of data to the console, therefore you must limit the number of recursive calls in toString. The suggested limit is 20. To implement such limitation you have to declare a counter variable as a field in the Node class. Update the counter in the method, and make the recursive call conditional in an if statement controlled by the counter. The counter must be reset to zero before every new display application.

3. Class **Test**

This class shall exercise all the methods from Sortings and organizes the performance analysis. You may include the main method in this class, if so, all members of the class must be static. If you create another Application class for main, none of the members in Test are static.

Fields:
`int[ ] data:` : to store the numbers in an array

`Node head:` to store the reference to the head node of a linked list that stores the numbers

`int TOP = 12000000` named constant

Methods:

- **randomData( ):** void; takes an int parameter for **data** length; instantiates the data array, and instantiates the head field for a linked list. Fills the array with integers randomly selected from the range 0 <= x < TOP. Using the same for loop, the method generates the nodes of the list with the same random numbers as used for the array entries.

- **testSort( ):** void; parameters as suitable; this method calls the sorting methods and measures the running time for each, independent of each other. Calls display method(s) to print the time results and to demonstrate the sorting execution.

- **displayArray( ) and displayList ( )** print arrays and linked lists to the consol. Create an attractive display format for each case. For arrays larger than 200, print only 50 array entries evenly distributed over the total array. For instance, Figure 1 below illustrates the display of an array of length 300000.

- **main( )**

```
The array data[0..299999] was sorted in 16.39 seconds.

     The resulting array is:

          data[ 0]       21
          data[6000]    238904
          data[12000]    481443
          data[18000]    721207
          data[24000]    961016
          data[30000]    1200480
          data[36000]    1438963
          data[42000]    1677970
          data[48000]    1916379
          data[54000]    2155558
          data[60000]    2394058
          data[66000]    2629376
          data[72000]    2862326
          data[78000]    3103038
          data[84000]    3340798
          data[90000]    3584410
          data[96000]    3822442
          data[102000]    4067575
          data[108000]    4309194
          data[114000]    4552299
          data[120000]    4790055
          data[126000]    5029607
          data[132000]    5267807
          data[138000]    5510326
          data[144000]    5751761
          data[150000]    5989749
          data[156000]    6232703
          data[162000]    6469388
          data[168000]    6713051
          data[174000]    6948927
          data[180000]    7189037
          data[186000]    7430532
          data[192000]    7673084
          data[198000]    7920143
          data[204000]    8159646
          data[210000]    8396322
```

```
data[216000]    8636451
data[222000]    8877219
data[228000]    9122181
data[234000]    9358565
data[240000]    9604353
data[246000]    9844392
data[252000]    10085564
data[258000]    10325063
data[264000]    10563061
data[270000]    10799306
data[276000]    11042243
data[282000]    11281520
data[288000]    11517798
data[294000]    11757536
```

Figure 1

Testing requirements

Testing must be carried out in two phases

Phase 1. This phase is supposed to eliminate bugs, and must demonstrate that all sorting methods in the program work correct. In this phase do not use random selections and do not use large structures. Keep the size under 10 and test the boundary cases (empty structure, single element structure, two element structure). Empty and singleton structure do not need sorting. Document your test results by attaching short console printouts to your code.

Phase 2. In this phase the program is applied to test and compare the **performance** of working algorithms. Apply the method that randomly fills the structures and let the structures grow by taking for size input values the numbers 20, 5000, 10000, 20000, 40000, 80000, 160000, 320000, 640000, 1280000, 2560000, 5120000, and 10240000.
You may get an OutOfMemoryError on your computer when using the largest data set(s) and also StackOverflowError in case of recursive methods. In these cases restrict the input and/or comment out the method calls that cause trouble. Comment all your experience and use the results you are able to execute. Include a statement in your report specifying the value which could not be handled.

Report: interpretation of the results

Using Word or your favorite editor, create a table that lists the run time results as a function of input size for all five sorting algorithms. The table should show the growing factor, that is the

ratio of the current run time to the previous (smaller) run time.

Write a brief report containing your conclusions based a on the table and possible break-down events. Your report should correlate the **theoretical "big-O"** performance of each algorithm with the **actual run times** observed. In order to do this, **state how the run time is expected to change with the array size** as the array size gets doubled, match this against **the change you actually observe,** and finally **state whether or not these changes agree**.

In your report demonstrate the best case versus the worst case performance of the insertion sort algorithms. For this purpose do not use random data. Simply load the array indices to the array as values, first sorted in increasing order and for second, in reversed order.
Note that exact correspondence of observed results to theoretical performance is not expected, particularly for shorter run times. Because there is a certain small amount of error inherent in each of the time measurements ("noise"), the results from the smallest data sets may get buried in the noise. In fact, all run times will vary somewhat from one run to the next. **The largest 3 or 4 data sets are the most significant** for each method.

Your report should be clearly and professionally written with correct grammar and punctuation.

Hints

1. You must complete some designing details on your own. For instance the local variables will be needed in the testSort( ) method, and you may or may not apply here loops. Also, the display methods may print a cluster of output at one call, or just one result.
2. The linked list implementations require local variables for Node references. A pseudo code is provided below for insertion sort, and a description of the method design for merge sort.
3. The implementation of the mergeSort method follows the array ideas:
    - divide
    - mergeSort each of two halves
    - merge

    It is suggested that the divide (split) operation as well as the merge operation be implemented in private helper methods.
    Splitting a linked list cannot be analogous to splitting an array since the number of nodes in the list is not known, we cannot use it for the halving procedure.
    Instead, the split algorithm declares two Node references, say **middle** and **fast** initialized to **head** and **head.link** respectively, then it runs a while loop such that middle is forwarded one step, fast is forwarded two steps at each turn as long as fast reaches the end of the list. Now **head** is the head of the first half of the list and the **link** of middle is the head if the second half. Create a second head reference, then set middle.link to null (middle is now the tail of the first half). To test your split method, temporarily declare it public. If the length of a list is an even number, the split must produce two lists of equal length. For the merge operation the head reference of each list is given, say head1 and

head2. Declare the Nodes **head** and **tail** to be used in the merged list. Variable head is assigned head1 or head 2, depending which one stores smaller data and tail is initialized to head. Use head1 and head2 as cursors on the corresponding lists, that is, every time a cursor is merged, it moves up one step in the list. The merge iteration stops when one of the cursor falls off the list. The remaining other list must be joined to **tail.**

4. Pseudo-code for insertion sort
   The method takes the **head** reference of a linked list as parameter, and returns the head reference of the sorted list. Note that this pseudo-code does not specify any helper method, though using one is recommendable. The inOrder reference below is a cursor up to which the list is supposed to be sorted. Its link named checkOrder is the next node to be investigated for insertion.

- if head or head.link null, send message and return null
- declare Node references inOrder and checkOrder //these work for the outer while loop;
  precursor, cursor // these work for the inner while loop-- -
  - initialize inOrder to head
- while inOrder link not null
  - checkOrder assigned inOrder.link
  - store checkorder data in a temp variable
  - if temp < head.data
    - rearrange links to insert checkOrder before head
    - update head
  - else
    - declare Node references
      - precursor, initialize it as head
      - cursor, initialize to head.link
  - while cursor.data < temp
    - update precursor and cursor (moving one step forward)
  - if cursor is not checkOrder
    - rearrange links to insert checkOrder between precursor and cursor
  - else
    - update inOrder (moving one step forward)
- return head

Documentation and style requirements

As usual, conform to the document "Java documentation and style requirements." Document the methods you create (including any "helper" methods).

Evaluation (Total 100 points)

1. **Correctness (50 points).** Includes all aspects of your Java implementation meeting established requirements.
2. **Documentation and style (10 points).** Includes having the required banner and internal comments, choosing descriptive variable names, indentation, and overall professional appearance. Comments should be clearly written with correct grammar and spelling. Apply proper indentation.
3. **Design (20 points).** Includes the quality of method design. Complicated methods should be

modularized using private helper methods. Variables should be grouped at the beginning of methods, and methods should use local variables whenever needed. All activity should be organized well.

4. **Testing (10 points).** Includes ability to execute the program and proper submission of all materials.
5. **Written report (10 points).** Graded on both quality and content.

Submit: Upload the zipped project folder including the report file at Blackboard.