Ben Durao

CS 253

Homework #4

November 20, 2016

**Problem 1 –**

Heap sort using the bottom-up insertion method builds a heap containing elements to be sorted then removed in order by their values. It does so by first constructing the heap by calling its insertItem() method for each element to be sorted. Heap sort then recursively removes the root item and places it into its final location by swapping the smallest element in the heap to the root node, and the removed root item into the memory location to start the collection of sorted elements. The heap property must now by restored by moving the smallest element now in the root node to its proper place by moving it down heap appropriately comparing it to the values of its current children and swapping places if need be recursively until the property is satisfied. This method effectively shrinks the heap by one after each step and adds the removed element to the sorted sequence in order. It was an array based implementation because a heap is always nearly complete or is a complete tree. With an efficiency of O(n log n) for all cases, one can expect this method of sorting to be quite efficient in comparison to other advanced sorting methods that may take more time in their worst can than heap sort does in its worst case.

Findings:

The main.java program creates 3 identical arrays for each case. Then creates 3 objects that contain their respective advanced sorting method to compare with each other in this project, called in order of Heap Sort, Quick Sort, Shell Sort.

O Notation

Heap –

* Best Case (Reverse Sorted) – O(n log n) or O(a \* n log n)
* Worst Case (Already Sorted) – O(n log n) or O(b \* n log n)
* where a < b

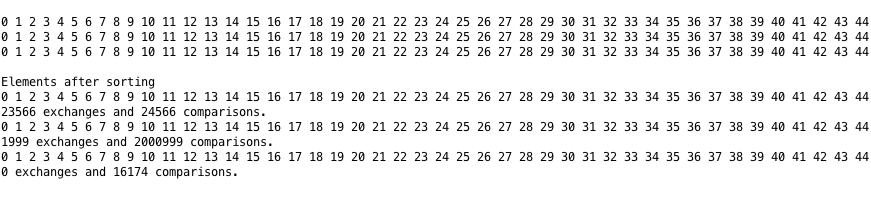
Quick –

* Best Case – O (n log n)
* Worst Case – O(n ^ 2)

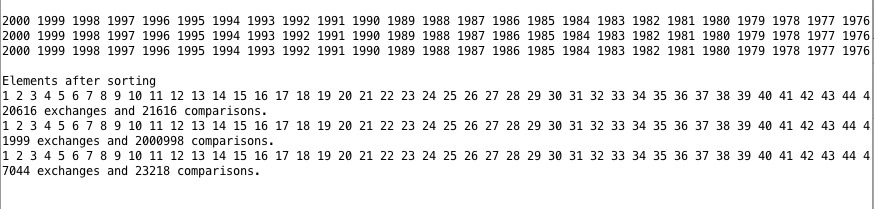
Shell –

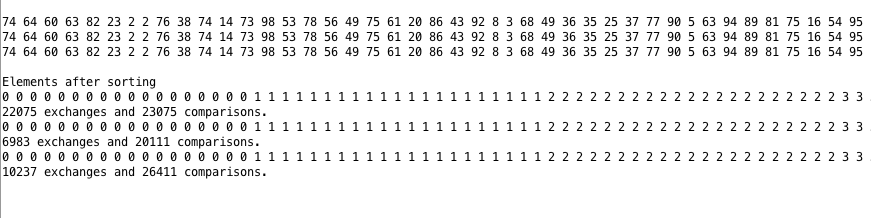
* Best Case – O(n log n)
* Worst Case – O(n (log n)^2)

The first case is an already sorted array of size 2000 where the elements range from 0 to 1999. After the program ran and the three sorting objects sorted their respective identical arrays it can be concluded that in the case that the array is already sorted, shell sort was the most efficient as it finished with the least number of comparisons and 0 exchanges, with heap sort being the next efficient algorithm and quick sort being the least appropriate method for sorting in this case. This makes sense because this is shell sort’s best case scenario where it does not have to do any sorting itself and just checks the data it needs to in order to determine the array is already sorted before terminating. [SHELL > HEAP > QUICK] (REMINDER: Results Top to Bottom: Shows initial arrays, calls Heap sort and shows sorted array its exchanges and comparisons, then the same for Quick sort then the same for Shell sort)



The second case is the reverse sorted array of size 2000 where the elements range from 2000 to 1. After the program ran and the three sorting algorithms worked their magic on their respective identical arrays, it can be concluded that in this case of the reverse sorted array, heap sort was the king with the least amount of comparisons needed to complete the sort. Shell sort was the next efficient method, and quick sort was the worst. This makes sense because this is the worst case scenario for the other two algorithms and it is indeed heap sort’s best case. Heap sort is able to utilize its maxheap property to maintain the least amount of comparisons needed to complete the sort. [HEAP > SHELL > QUICK] ((REMINDER: Results Top to Bottom: Shows initial arrays, calls Heap sort and shows sorted array its exchanges and comparisons, then the same for Quick sort then the same for Shell sort)



The third case was the randomly sorted array of size 2000 where the each of the elements were randomly generated numbers from 1 to 100. After the program ran and the three sorting algorithms had a chance to sort their respective identical arrays, it was found that quick sort was the most efficient to do so in this case with respect to its comparisons used to put the array in order. Heap sort was the next efficient, and shell short was the worst method to use in this case. This makes sense because quick sort is able to establish its pivots in such a way in randomly sorted arrays that it efficiently sorts the average case in n log n time in comparison to heap and shell taking a little more extra time with respect to their constant coefficients that affects their overall efficiency. [QUICK > HEAP > SHELL] (REMINDER: Results Top to Bottom: Shows initial arrays, calls Heap sort and shows sorted array its exchanges and comparisons, then the same for Quick sort then the same for Shell sort)

Final Conclusion for sorting method to use per data set style:

* Ordered Data Set: Shell
* Reversed Data Set: Heap
* Average Data Set: Quick

Code For Heap:

|  |
| --- |
| public class Heap  {  private static int N; // number of elements in tree  public static int comparisons, exchanges;  public static void sort(int arr[]) // sort the array  {  buildHeap(arr);  for (int i = N; i > 0; i--)  {  swap(arr,0, i);  N = N-1;  maxheap(arr, 0);  }  }  public static void buildHeap(int arr[]) // insert new element and then fix it  {  N = arr.length-1;  for (int i = N/2; i >= 0; i--)  maxheap(arr, i);  }  public static void maxheap(int arr[], int i) // fixes heap property  {  int leftChild = 2\*i;  int rightChild = 2\*i + 1;  int max = i;  if (leftChild <= N && arr[leftChild] > arr[i])  max = leftChild;  if (rightChild <= N && arr[rightChild] > arr[max])  max = rightChild;  comparisons++;  if (max != i)  {  swap(arr, i, max);  maxheap(arr, max);  }  }  public static void swap(int arr[], int i, int j) // swaps elem in arr[i] with element in arr[j]  {  int temp = arr[i];  arr[i] = arr[j];  arr[j] = temp;  exchanges++;  } } |

**Problem 2 –**

10 Elements Inserted into empty 2-3 Tree

tree[] = 100 90 80 70 60 50 40 30 20 10

This creates 8 nodes, where each node can contain at most 2 elements = 16 available spots.

There are 6 empty flags left over at the end of the insertion calls.

6 / 16 = 0.375 \* 100 = 37.5 % wasted storage of empty flags

10 / 16 = 0.625 \* 100 = 62.5 % filled

