**Lab 4 Report**

**CS303L Algorithms and Data Structures**

**Sam Lazrak**

**Spring Semester 2018**

**In-class Assignment:**

**Objectives:**

* Implement heap sort using a max-heap
* Compare the performance of insertion sort, merge sort, and heap sort

**In-class Assignments:**

1. Implement a method to sort a given array using the heap sort algorithm. Use the algorithm from the textbook (see below). You should finish implementing Build Max Heap and Max Heapify by the end of lab. You can finish implementing the full algorithm for homework.

**Code for in class lab assignment:**

|  |
| --- |
| package lab4;  import java.io.File; import java.io.FileNotFoundException; import java.util.Arrays; import java.util.Scanner;  public class lab4 {   public static void main(String[] args) throws FileNotFoundException {  File file = new File("input\_100.txt");  Scanner scanner = new Scanner(file);   int[] array = new int[100];   int i = 0;  while(scanner.hasNextInt()) {  array[i] = scanner.nextInt();  i++;  }   System.out.println("Unsorted Array: " + Arrays.toString(array));  heapSort(array, array.length-1);  System.out.println("Sorted Array: " + Arrays.toString(array) + "\n");   }   public static int left(int i){  return 2\*i;  }   public static int right(int i){  return 2\*i +1;  }  public static void maxHeapify(int[] A, int i, int n) {  int left = left(i);  int right = right(i);  int largest = 0;  if( left <= n && A[left] > A[i]){  largest = l;  }  else  largest = i;  if(right <= n && A[right] > A[largest]){  largest = right;  }  if(largest != i){  int temp = A[i];  A[i] = A[largest];  A[largest] = temp;  maxHeapify(A, largest, n);  }  }  public static void buildMaxHeap(int[] A, int n){  for(int i=(int) Math.floor(n/2); i >= 0; i--) {  maxHeapify(A, i, n);  }  }  public static void heapSort(int[] A, int n){  buildMaxHeap(A, n);  for(int i= n; i >= 1; i--){  int temp = A[0];  A[0] = A[i];  A[i] = temp;  maxHeapify(A,0,i-1);  }   }  }  **The output of this code:**  Unsorted Array: [4, 50, 34, 40, 22, 54, 94, 3, 94, 38, 8, 95, 0, 36, 54, 54, 81, 30, 24, 98, 12, 25, 43, 0, 52, 52, 88, 22, 83, 70, 96, 57, 89, 53, 13, 64, 74, 18, 37, 86, 73, 76, 15, 1, 93, 69, 77, 81, 29, 78, 14, 45, 67, 1, 0, 41, 60, 63, 74, 16, 75, 75, 36, 49, 68, 5, 67, 29, 15, 84, 47, 77, 40, 80, 24, 61, 25, 7, 85, 83, 81, 47, 10, 39, 22, 72, 87, 64, 92, 27, 50, 69, 12, 54, 23, 85, 38, 75, 73, 94]  Sorted Array: [0, 0, 0, 1, 1, 3, 4, 5, 7, 8, 10, 12, 12, 13, 14, 15, 15, 16, 18, 22, 22, 22, 23, 24, 24, 25, 25, 27, 29, 29, 30, 34, 36, 36, 37, 38, 38, 39, 40, 40, 41, 43, 45, 47, 47, 49, 50, 50, 52, 52, 53, 54, 54, 54, 54, 57, 60, 61, 63, 64, 64, 67, 67, 68, 69, 69, 70, 72, 73, 73, 74, 74, 75, 75, 75, 76, 77, 77, 78, 80, 81, 81, 81, 83, 83, 84, 85, 85, 86, 87, 88, 89, 92, 93, 94, 94, 94, 95, 96, 98] |

**Homework Assignment:**

1. Compare the execution time of heap sort with insertion sort implemented in Lab 2 and merge sort implemented in Lab 3. Make sure you use the same array to compare the performance. Use a table or plot to summarize the results and document your observations and analysis in the report.
2. Test the program for the same array sizes and values. Compare the performance with the original merge sort implementation, plot the execution times, and document the analysis in your lab report.

|  |
| --- |
| package lab4;  import lab4.InsertionSort;  import lab4.MergeSort;  import java.io.File;  import java.io.FileNotFoundException;  import java.util.Arrays;  import java.util.Scanner;  public class HeapSort {  public static void main(String[] args) throws FileNotFoundException {  MergeSort MergeSort = new MergeSort();  InsertionSort InserstionSort = new InsertionSort();  File[] files = new File[7];  files[0] = new File("input\_100.txt");  files[1] = new File("input\_1000.txt");  files[2] = new File("input\_5000.txt");  files[3] = new File("input\_10000.txt");  files[4] = new File("input\_50000.txt");  files[5] = new File("input\_100000.txt");  files[6] = new File("input\_500000.txt");  int[] fileSize = {100, 1000, 5000, 10000, 50000, 100000, 500000};  for (int i = 0; i <= 6; i++) {  int length = fileSize[i];  int[] array = new int[length];  Scanner scanner = new Scanner(files[i]);  int j = 0;  while (scanner.hasNextInt()) {  array[j] = scanner.nextInt();  j++;  }  int[] heapSort = Arrays.copyOf(array, array.length);  long HeapSortTime = System.nanoTime();  heapSort(heapSort, heapSort.length - 1);  HeapSortTime = System.nanoTime() - HeapSortTime;  int[] insertionSort = Arrays.copyOf(array, array.length);  long InsertionSortTime = System.nanoTime();  InsertionSort.insertionSort(insertionSort);  InsertionSortTime = System.nanoTime() - InsertionSortTime;  int[] mergeSort = Arrays.copyOf(array, array.length);  long MergeSortTime = System.nanoTime();  MergeSort.mergeSort(mergeSort);  MergeSortTime = System.nanoTime() - MergeSortTime;  int[] insertionSortmergeSort = Arrays.copyOf(array, array.length);  long InsertionSortMergeSortTime = System.nanoTime();  MergeSort.mergeInsertionSort(insertionSortmergeSort, fileSize[i] / 10);  InsertionSortMergeSortTime = System.nanoTime() - InsertionSortMergeSortTime;  System.out.println("Array Size: " + fileSize[i] + "\n" + "Insertion Sort Time: " + InsertionSortTime + "\n" + "Heap Sort Time: " + HeapSortTime + "\n" + "Merge Sort Time: " + MergeSortTime + "\n" +  "Merge Sort with Insertion Time: " + InsertionSortMergeSortTime + "\n");  }  }  public static int left(int i) {  return 2 \* i;  }  public static int right(int i) {  return 2 \* i + 1;  }  public static void maxHeapify(int[] A, int i, int n) {  int l = left(i);  int r = right(i);  int largest = 0;  if (l <= n && A[l] > A[i]) {  largest = l;  } else  largest = i;  if (r <= n && A[r] > A[largest]) {  largest = r;  }  if (largest != i) {  int temp = A[i];  A[i] = A[largest];  A[largest] = temp;  maxHeapify(A, largest, n);  }  }  public static void buildMaxHeap(int[] A, int n) {  for (int i = (int) Math.floor(n / 2); i >= 0; i--) {  maxHeapify(A, i, n);  }  }  public static void heapSort(int[] A, int n) {  buildMaxHeap(A, n);  for (int i = n; i >= 1; i--) {  int temp = A[0];  A[0] = A[i];  A[i] = temp;  maxHeapify(A, 0, i - 1);  }  }  } |

**Output for homework assignment:**

Array Size: 100

Insertion Sort Time: 73353

Heap Sort Time: 791718

Merge Sort Time: 102457

Merge Sort with Insertion Time: 33971

Array Size: 1000

Insertion Sort Time: 3805174

Heap Sort Time: 285566

Merge Sort Time: 1562188

Merge Sort with Insertion Time: 67046

Array Size: 5000

Insertion Sort Time: 17568019

Heap Sort Time: 840455

Merge Sort Time: 1520938

Merge Sort with Insertion Time: 308811

Array Size: 10000

Insertion Sort Time: 79551112

Heap Sort Time: 3990845

Merge Sort Time: 1202736

Merge Sort with Insertion Time: 128758

Array Size: 50000

Insertion Sort Time: 642811795

Heap Sort Time: 10982531

Merge Sort Time: 4825958

Merge Sort with Insertion Time: 440872

Array Size: 100000

Insertion Sort Time: 1605541476

Heap Sort Time: 13993595

Merge Sort Time: 9640689

Merge Sort with Insertion Time: 727191

Array Size: 500000

Insertion Sort Time: 39712590488

Heap Sort Time: 76300778

Merge Sort Time: 54823836

Merge Sort with Insertion Time: 5098615

**Analysis of results:**

Insertion sort has a running time of Merge sort has a running time of Heap sort has a running time of From the data one can see that insertion sort takes the longest time and heap sort takes slightly longer than merge sort but merge sort with insertion sort is the fastest.