College of Engineering, Software Department
SE 2221 Algorithms

Laboratory 3
Tree Sort and Heap Sort

Name: Rafael III Prudente Year: II
Name: Paul Vincent Leradao Year: II

General Instructions: For every item below, please submit one zip file named "<surname1>_<surname2>_<item number>.zip". For example, if Kaedahara and Sangonomiya were submitting for section number 3, they would name their zip file "Kaedahara_Sangonomiya_3.zip". Please also submit a filled out version of this laboratory named "Kaedahara_Sangonomiya_Lab3.doc" or with any file extension you have .

Objectives:

- To grasp the concepts of how tree sort and heap sort would perform their sorting algorithm.
- To be able to measure the worst case and best case of the tree sort algorithm.

Section 1.
Perform a simple experiment measuring the execution time of the Tree Sort algorithm on differing sizes of data. Include two general kinds of arrays: a random array and an array that will produce an imbalanced tree.

Make sure to try out at least five different sizes of array, with each size having a random version and an imbalanced version. Therefore, I expect there should be at least 10 trials. For each trial, you should perform multiple measurements and get the average to get a more viable measurement. The sizes you use for your arrays should differ by a factor of around 100 (i.e. 100 for one array then 10000 for the other).

Present your results using a table or a graphical chart. Once again, chart.js might help you in compiling your results. You must submit code for this along with this laboratory doc.

Results. (Table or Chart)

Execution Time		
Test Trials	Random	Imbalanced
100	3.416ms	2.449ms
200	3.072ms	2.563ms
300	3.628ms	3.363ms
400	5.06ms	4.753ms
500	5.614ms	5.196ms

Questions:

1. Were you able to confirm that Tree Sort performs slower on an array that produces an imbalanced tree? If not, what factors contributed to the results not following their theoretical Big O notation counterpart? If so, how were you able to confirm this?

- No, because by basing on our trials, it do pertains that the array with random numbers is much slower than the imbalanced array. We do think that it depends also on the random numbers being inserted into the random array. It is expected that when the randomly sorted array, it do have a chance that it can make a balance array. We think that the factors that affects this result are the specific random numbers inputted in the array because it might create also an imbalanced array and it can be also the computer hardware or our laptop it was being executed.
  But, imbalanced array such as we made that it is already a sorted array is the worst case of tree sort which has the time complexity of O(n^2). In the books, the imbalanced array is much slower than the random array but for our case, it is the random array that is much slower. We will stick to it as for the reason that a random array can also have the chance to create an imbalanced array.

Section 2.
This week we have discussed the Heap Sort algorithm. The heap that we used in the discussion was called a “max-heap”, meaning the root node would always contain the maximum value in the whole data structure. Now, it is your turn to implement your own heap sort algorithm. However, you shall use a “min-heap” instead. Plus, the sorted array should be in descending order.

A Heap Sort algorithm that uses “min-heap” is still fairly similar to the “max-heap” version. Please implement your own needed classes or functions especially the bubbleDown and heapify functions.

Questions:

1. How did you convert the Tree Sort’s “max-heap” variant into the “min-heap” variant?

As for our example last time, we made a max-heap variant and for this case, we converted it to a min heap variant. But how ? As discussed in our previous class, in a max heap variant, the largest element is at the root node of the binary tree and it’s continuously swapped down its correct position in the heap. So to modify this to min heap variant, we do need to change the comparison operators used to find the smallest element in the heap. Which makes us use “>’ instead of “<” comparison operator. The resulting array is in descending order.

2. Please give me a rundown of your code, and explain how each section works. You can put screenshots or directly paste the code here in the document.

```
function heapSort(arr: number[]): number[] {
  // this function takes an array of numbers and returns a sorted
array
  buildMinHeap(arr);
  for (let i = arr.length - 1; i > 0; i--) {
      [arr[0], arr[i]] = [arr[i], arr[0]];
      minHeapify(arr, 0, i);
      //this part of the code loops through the array of numbers
from the
      //end to the start or from right to left of the array
      //it swaps the first element or the current lowest element
with
      //the last element of the input array.
      //after each of the swaps, the minHeapify function is called
to maintain the min-hip property.
```

```
  }
  return arr; // returns the sorted array
}
//function to convert the input array of numbers into a binary
min-heap
function buildMinHeap(arr: number[]): void {
  const n = arr.length;
  for (let i = Math.floor(n / 2) - 1; i >= 0; i--) {
      minHeapify(arr, i, n);
  }
}
//it calculates first the length of the input array,
//then it iterates over the elements from the middle to the
beginning
//and class the minHeapify function to  maintain the min-hip
property


function minHeapify(arr: number[], i: number, heapSize: number):
void {
  const left = 2 * i + 1;
  const right = 2 * i + 2;
  let smallest = i;
  //checks if the index of the left child is within the heap (less
then the heap size)
  // and if the value of the left child index is less than the value
of the smallest index
  // then the smallest if set to the lest
  if (left < heapSize && arr[left] < arr[smallest]) {
      smallest = left;
  }
  //checks if the index of the right child is within the heap
  //and if the value of the right child index is less than the value
of the smallest index
  // then the snallest if set to the right
  if (right < heapSize && arr[right] < arr[smallest]) {
      smallest = right;
  }
  // checks if the smallest index if not equal to i
  //if true, then the values at then smallest index and i index are
```

```
swapped/
  // and the minHeapify function if called recursively on the
smallest index
  // to ensure that the subtree is rooted at the smallest index.
  if (smallest !== i) {
      [arr[i], arr[smallest]] = [arr[smallest], arr[i]];
      minHeapify(arr, smallest, heapSize);
  }
}

let sampleRandomArray: number[] = [];
for (let i = 1; i <= 100; i++) {
  let randomNum = Math.ceil(Math.random() * i);
  sampleRandomArray.push(randomNum);
}



console.log(sampleRandomArray);
console.log('Original array:', sampleRandomArray);

console.time("Execution Time   ")
const sortedArr = heapSort(sampleRandomArray);
console.log('Sorted array:', sortedArr);
console.timeEnd("Execution Time   ")
```

Conclusions:

For our conclusion, we just made a tress sort and heap sort algorithms based on the lessons we've discussed these past two weeks. For the tree sort, we are tasked to compare the random array of numbers an the imbalanced array for their execution time. Based on our results, the one with random array is much slower than the one in imbalanced or already sorted that we've made. but , basing for the books, the worst case of a tree sort is when it nearly or already sorted array.
For the heap sort, we are tasked to convert the max heap into a min heap which pertains to get the minimum element element in the array and the resulting array is in descending order.. We've successfully made one and have trials on it by basing in array that have 100, 200, 300, 400, and 500 elements in it.