**Question 1: (Chapter 23)**

Write a program to obtain the execution time of selection sort, bubble sort, merge sort, quick sort, heap sort and radix sort for input size 5000. Your program will create the data randomly and use the same data for all sort algorithms.

*BigOSort (Main Method)*

import java.util.Arrays;

public class BigOSort {

public static void main(String[] args) {

int [] value = creatingArray(5000);

//printUnsorted(value);

QuickSort quick = new QuickSort(Arrays.copyOf(value, value.length));

BubbleSort bub = new BubbleSort (Arrays.copyOf(value, value.length));

SelectionSort sel = new SelectionSort (Arrays.copyOf(value, value.length));

MergeSort meg = new MergeSort (Arrays.copyOf(value, value.length));

RadixSort rad = new RadixSort (Arrays.copyOf(value, value.length));

HeapSort hep = new HeapSort (Arrays.copyOf(value, value.length));

meg.result();

//meg.printArray();

quick.result();

//quick.printArray();

rad.result();

//rad.printArray();

hep.result();

//hep.printArray();

sel.result();

//sel.printArray();

bub.result();

//bub.printArray();

}

private static void printUnsorted(int[] value) {

System.out.println("UnSorted Array ");

for (int e: value) {

System.out.print(value[e]+ " ");

}

System.out.println("\n");

}

private static int[] creatingArray(int size) {

int[] arr = new int [size];

for (int i=0; i<size; i++) {

arr[i] = (int) (Math.random()\* size);

}

return arr;

}

}

*Selection Sort*

// based on introduction to Java Programming and Data Structure Listing 7.8

public class SelectionSort {

private int [] arr;

public SelectionSort(int [] newArr) {

arr = newArr;

}

private void sort () {

for (int i = 0;i<arr.length-1; i++){

int currentMin = arr [i];

int currentMinIndex = i;

for (int j = i +1 ; j <arr.length; j++){

if (currentMin > arr[j]){

currentMin = arr[j];

currentMinIndex = j;

}

}

if (currentMinIndex !=i){

arr[currentMinIndex] = arr [i];

arr[i] =currentMin;

}

}

}

public void result() {

long startTime = System.currentTimeMillis();

sort ();

long endTime = System.currentTimeMillis();

System.out.println("Execution Time for Selection Sort: " + (endTime -startTime) + " milliseconds");

}

public void printArray () {

System.out.println("Sorted Array from Selection sort");

for (int i= 0; i< arr.length; i++) {

System.out.print(arr[i]+ " ");

}

System.out.println("\n");

}

}

*Bubble Sort*

// based on Lecture Side on Chapter 23

public class BubbleSort {

private int [] list;

public BubbleSort(int [] newArr){

list = newArr;

}

private void sort () {

boolean needNextPass = true;

for (int k = 1; k < list.length && needNextPass; k++) {

needNextPass = false;

for (int i = 0; i < list.length - k; i++) {

if (list[i] > list[i + 1]) {

int temp = list[i];

list[i] = list[i + 1];

list[i + 1] = temp;

needNextPass = true;

}

}

}

}

public void result() {

long startTime = System.currentTimeMillis();

sort();

long endTime = System.currentTimeMillis();

System.out.println("Execution Time for Bubble Sort: " + (endTime -startTime) + " milliseconds" );

}

public void printArray() {

System.out.println("Sorted Array from Bubble sort");

for (int e : list) {

System.out.print(e + " ");

}

System.out.println("\n");

}

}

*MergeSort*

//based on lecture slides on chapter 23

public class MergeSort {

private int [] arr;

MergeSort(int [] newArr){

arr = newArr;

}

private void sort() {

if (arr.length > 1) {

int [] firstHalf = new int[arr.length/2];

System.arraycopy(arr, 0, firstHalf, 0, arr.length/2);

sort(firstHalf);

int secondHalfLength = arr.length - arr.length / 2;

int[] secondHalf = new int[secondHalfLength];

System.arraycopy(arr, arr.length / 2,

secondHalf, 0, secondHalfLength);

sort(secondHalf);

merge(firstHalf, secondHalf, arr);

}

}

private void sort(int[] arr) {

if (arr.length > 1) {

int [] firstHalf = new int[arr.length/2];

System.arraycopy(arr, 0, firstHalf, 0, arr.length/2);

sort(firstHalf);

int secondHalfLength = arr.length - arr.length / 2;

int[] secondHalf = new int[secondHalfLength];

System.arraycopy(arr, arr.length / 2,

secondHalf, 0, secondHalfLength);

sort(secondHalf);

merge(firstHalf, secondHalf, arr);

}

}

private void merge(int[] list1, int[] list2, int[] temp) {

int current1 = 0;

int current2 = 0;

int current3 = 0;

while (current1 < list1.length && current2 < list2.length) {

if (list1[current1] < list2[current2])

temp[current3++] = list1[current1++];

else

temp[current3++] = list2[current2++];

}

while (current1 < list1.length)

temp[current3++] = list1[current1++];

while (current2 < list2.length)

temp[current3++] = list2[current2++];

}

public void result() {

long startTime = System.currentTimeMillis();

sort ();

long endTime = System.currentTimeMillis();

System.out.println("Execution Time for Merge Sort: " + (endTime -startTime) + " milliseconds");

}

public void printArray() {

System.out.println("Sorted Array from Merge sort");

for (int i= 0; i< arr.length; i++) {

System.out.print(arr[i]+ " ");

}

System.out.println("\n");

}

}

*QuickSort*

//based on lecture slides on chapter 23

public class QuickSort {

private int [] list;

public QuickSort(int [] newArr){

list = newArr;

}

private void sort () {

sort (list, 0 , list.length -1);

}

private void sort (int [] arr, int first, int last) {

if (last > first ) {

int pivotIndex = partition(arr, first, last);

sort (arr, first, pivotIndex - 1);

sort(arr, pivotIndex + 1, last);

}

}

private int partition(int[] arr, int first, int last) {

int pivot = arr[first];

int low = first + 1;

int high = last;

while (high > low) {

while (low <= high && arr[low] <= pivot)

low++;

while (low <= high && arr[high] > pivot)

high--;

if (high > low) {

int temp = arr[high];

arr[high] = arr[low];

arr[low] = temp;

}

}

while (high > first && arr[high] >= pivot)

high--;

if (pivot > arr[high]) {

arr[first] = arr[high];

arr[high] = pivot;

return high;

}

else {

return first;

}

}

public void result() {

long startTime = System.currentTimeMillis();

sort ();

long endTime = System.currentTimeMillis();

System.out.println("Execution Time for Quick Sort: " + (endTime -startTime) + " milliseconds" );

}

public void printArray() {

System.out.println("Sorted Array from Quick sort");

for (int e : list) {

System.out.print(e + " ");

}

System.out.println("\n");

}

}

*HeapSort*

// based on lecture side 23

import java.util.Arrays;

import java.util.Comparator;

public class HeapSort {

private int [] arr;

private Integer [] arr2;

public HeapSort(int [ ] newArr) {

arr = newArr;

}

private <E> void heapSort(E[] list) {

heapSort(list, (e1, e2) -> ((Comparable<E>)e1).compareTo(e2));

}

private <E> void heapSort(E[] list, Comparator<E> c) {

Heap<E> heap = new Heap<>(c);

for (int i = 0; i < list.length; i++)

heap.add(list[i]);

for (int i = list.length - 1; i >= 0; i--)

list[i] = heap.remove();

}

public void result () {

arr2 = Arrays.stream( arr ).boxed().toArray( Integer[]::new );

long startTime = System.currentTimeMillis();

heapSort (arr2);

long endTime = System.currentTimeMillis();

System.out.println("Execution Time for Heap Sort: " + (endTime -startTime) + " milliseconds");

}

public void printArray() {

System.out.println("Sorted Array from Heap sort");

for (int i= 0; i< arr2.length; i++) {

System.out.print(arr2[i]+ " ");

}

System.out.println("\n");

}

}

*Heap Class*

//based on lecture sides chapter 23

public class Heap<E> {

private java.util.ArrayList<E> list = new java.util.ArrayList<>();

private java.util.Comparator<? super E> c;

public Heap() {

this.c = (e1, e2) -> ((Comparable<E>)e1).compareTo(e2);

}

public Heap(java.util.Comparator<E> c) {

this.c = c;

}

public Heap(E[] objects) {

this.c = (e1, e2) -> ((Comparable<E>)e1).compareTo(e2);

for (int i = 0; i < objects.length; i++)

add(objects[i]);

}

public void add(E newObject) {

list.add(newObject);

int currentIndex = list.size() - 1;

while (currentIndex > 0) {

int parentIndex = (currentIndex - 1) / 2;

if (c.compare(list.get(currentIndex),

list.get(parentIndex)) > 0) {

E temp = list.get(currentIndex);

list.set(currentIndex, list.get(parentIndex));

list.set(parentIndex, temp);

}

else

break;

currentIndex = parentIndex;

}

}

public E remove() {

if (list.size() == 0) return null;

E removedObject = list.get(0);

list.set(0, list.get(list.size() - 1));

list.remove(list.size() - 1);

int currentIndex = 0;

while (currentIndex < list.size()) {

int leftChildIndex = 2 \* currentIndex + 1;

int rightChildIndex = 2 \* currentIndex + 2;

if (leftChildIndex >= list.size()) break;

int maxIndex = leftChildIndex;

if (rightChildIndex < list.size()) {

if (c.compare(list.get(maxIndex),

list.get(rightChildIndex)) < 0) {

maxIndex = rightChildIndex;

}

}

if (c.compare(list.get(currentIndex),

list.get(maxIndex)) < 0) {

E temp = list.get(maxIndex);

list.set(maxIndex, list.get(currentIndex));

list.set(currentIndex, temp);

currentIndex = maxIndex;

}

else

break;

}

return removedObject;

}

public int getSize() {

return list.size();

}

public boolean isEmpty() {

return list.size() == 0;

}

}

*Radix Sort*

// based on https://www.geeksforgeeks.org/radix-sort/

import java.util.Arrays;

public class RadixSort {

private int [] arr;

RadixSort (int [] newArr){

arr = newArr;

}

private void sort() {

int mx = getMax(arr);

for (int exp = 1; mx/exp >0;exp\*=10) {

countSort (arr, exp);

}

}

private void countSort(int[] arr2, int exp) {

int output [] = new int [arr2.length];

int count [] = new int [10];

Arrays.fill(count, 0);

for (int i =0; i< arr.length;i++) {

count[ (arr2[i]/exp)%10 ]++;

}

for (int i=1;i<10; i++ ) {

count[i] += count[i - 1];

}

for (int i = arr.length-1;i>=0; i--) {

output[count[ (arr2[i]/exp)%10 ] - 1] = arr2[i];

count[ (arr2[i]/exp)%10 ]--;

}

for (int i = 0 ; i< arr.length; i++) {

arr2[i] = output[i];

}

}

private int getMax(int[] arr2) {

int max = arr2[0];

for (int i : arr2) {

if (arr2[i]> max) {

max = arr2[i];

}

}

return max;

}

public void result() {

long startTime = System.currentTimeMillis();

sort ();

long endTime = System.currentTimeMillis();

System.out.println("Execution Time for Radiox Sort: " + (endTime -startTime) + " milliseconds");

}

public void printArray() {

System.out.println("Sorted Array from Radiox sort");

for (int i= 0; i< arr.length; i++) {

System.out.print(arr[i]+ " ");

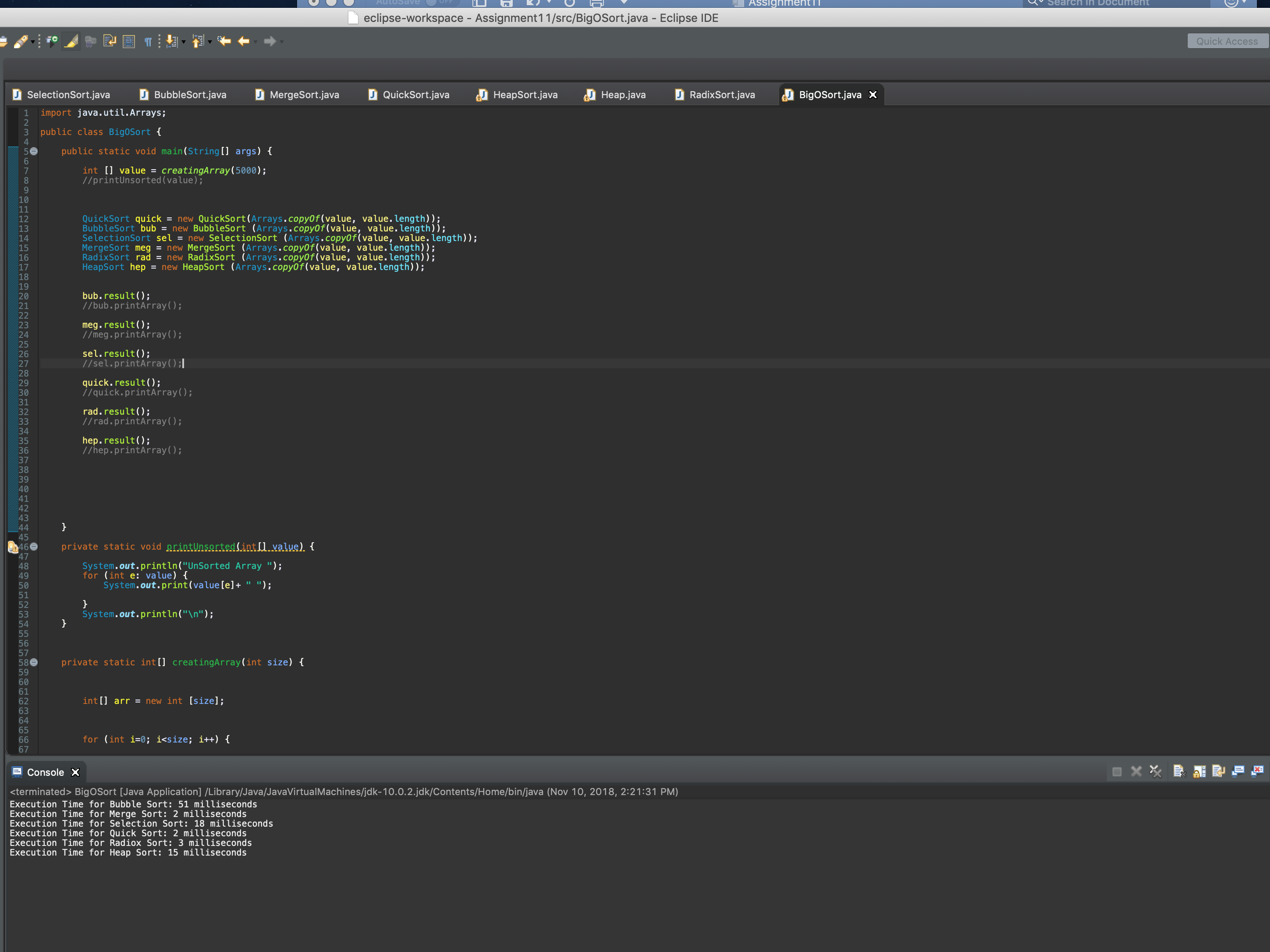
}

System.out.println("\n");

}

}

*ScreenShot*

**

**Question 2: (Chapter 24)**

Write a program to implement the min heap. Your insert method will insert the value in the correct place and display the vale of parent node, value of left child and value of right child. The remove method will remove the root and display its value which is the min value.

Test your program on the following:

        minHeap.insert(5);

        minHeap.insert(3);

        minHeap.insert(17);

        minHeap.insert(10);

        minHeap.insert(84);

        minHeap.insert(19);

        minHeap.insert(6);

        minHeap.insert(22);

        minHeap.insert(9);

        minHeap.remove());

*MinHeap*

//modified the code from https://www.cs.cmu.edu/~adamchik/15-121/lectures/Binary%20Heaps/code/Heap.java

public class MinHeap <E extends Comparable <E>> {

private int len;

private E [] heap;

public MinHeap () {

len = 0;

heap = (E []) new Comparable [2];

}

public MinHeap (E [] arr) {

len = arr.length;

heap = (E []) new Comparable [arr.length+1];

}

private void add() {

for (int i = len/2 ; i >0 ; i--) {

siftDown(i);

}

}

private void siftDown(int i) {

// TODO Auto-generated method stub

E tmp = heap[i];

int child = 0;

for (;2\*i <=len ; i = child) {

child = 2\*i;

if (child != len && heap [child].compareTo(heap[child + 1])> 0) {

child++;

}

if (tmp.compareTo(heap[child])> 0) {

heap [i] = heap [child];

}

else {

break;

}

}

heap[i] = tmp;

}

public void remove() {

if (len == 0) {

System.out.println("Empty heap");

}

E min = heap [1];

heap [1] = heap [len--];

siftDown(1);

System.out.println("Removed root value is " + min);

}

public void insert (E n) {

if(len == heap.length-1) {

doubleSize();

}

int pos = ++len;

for(; pos> 1 && n.compareTo(heap[pos/2]) <0; pos = pos/2) {

heap [pos] = heap [pos/2];

}

heap [pos] = n;

printHeap();

}

private void printHeap() {

// TODO Auto-generated method stub

for (int i = 1 ; i<=len /2; i++) {

System.out.print("Parent Node: " + heap [i]+ " Left Child: " + heap[2\*i]

+ " Right Child: " + heap[2\*i +1]);

System.out.println();

}

System.out.println("\n");

}

private void doubleSize() {

// TODO Auto-generated method stub

E [] old = heap;

heap = (E[] ) new Comparable [heap.length \*2];

System.arraycopy(old, 1, heap, 1, len);

}

public void printArray () {

for (int i = 1 ; i <=len; i++) {

System.out.print(heap [i] + " ");

}

System.out.println("\n");

}

public static void main (String [] args) {

MinHeap <Integer> minHeap = new MinHeap <Integer> ();

minHeap.insert(5);

minHeap.insert(3);

minHeap.insert(17);

minHeap.insert(10);

minHeap.insert(84);

minHeap.insert(19);

minHeap.insert(6);

minHeap.insert(22);

minHeap.insert(9);

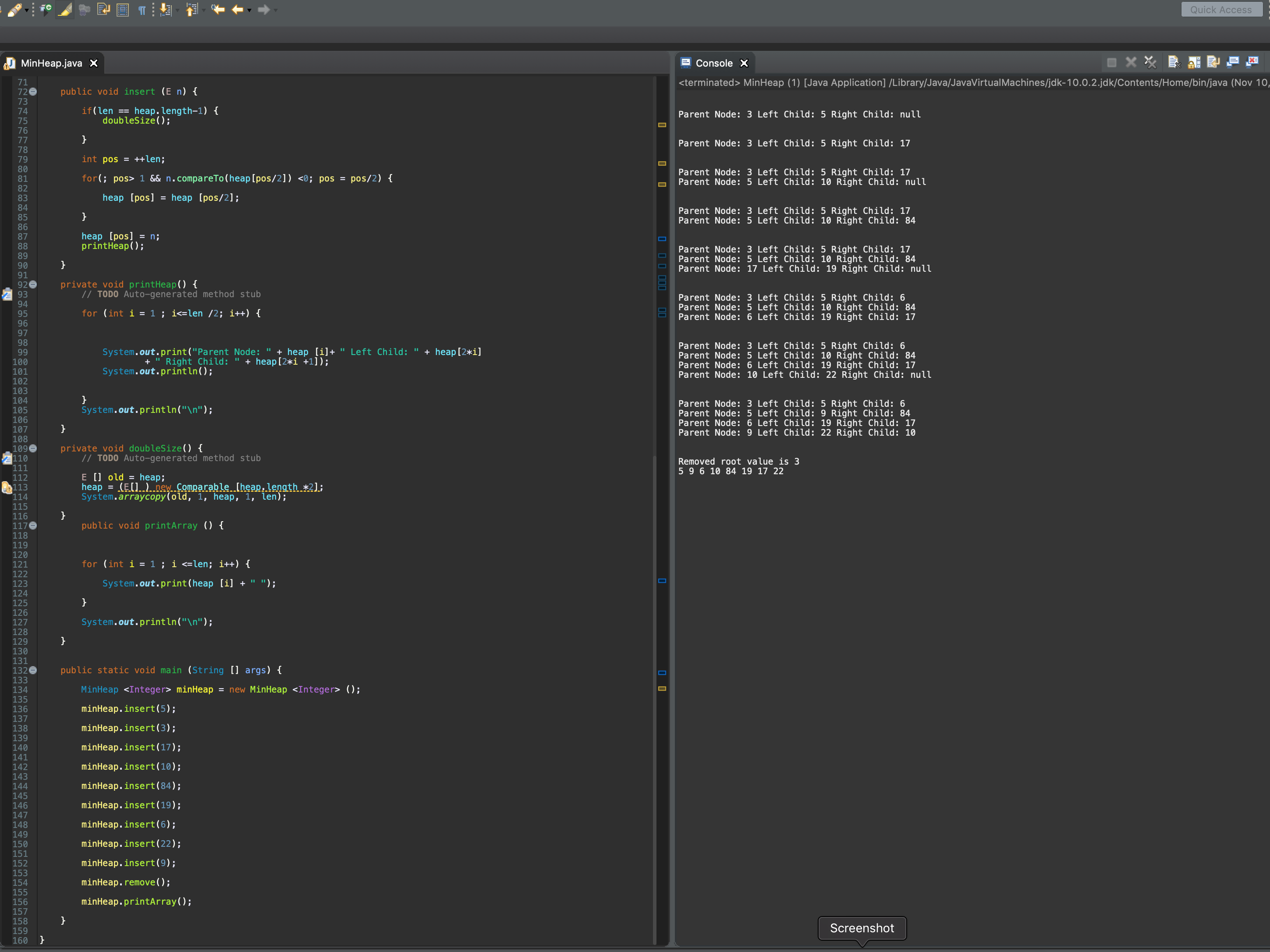
minHeap.remove();

minHeap.printArray();

}

}

*Screenshots*

**