**Assignment Cover Sheet**

Assignment No.: \_\_2\_\_\_

|  |  |  |
| --- | --- | --- |
| **Student Information (For group assignment, please state names of all members)** | | **Grade/Marks** |
| **Name** | **ID** |  |
|  | **B1901898** |  |
|  |  |  |
|  |  |  |
|  |  |  |

|  |  |  |
| --- | --- | --- |
| **Module/Subject Information** | | **Office Acknowledgement** |
| **Module/Subject Code** | BIT208 |  |
| **Module/Subject Name** | Data Structures and Algorithms |  |
| **Lecturer/Tutor/Facilitator** | Dr. Abdul Qayoom |  |
| **Due Date** | 21 November, 2021 |  |
| **Assignment Title/Topic** | Assignment 2 |  |
| **Intake (where applicable)** | Semester 3, 2021 |  |
| **Word Count** | N/A | **Date/Time** |

**Declaration**

* I/We have read and understood the Programme Handbook that explains on **plagiarism**, and I/we testify that, unless otherwise acknowledged, the work submitted herein is entirely my/our own.
* I/We declare that no part of this assignment has been written for me/us by any other person(s) except where such collaboration has been authorized by the lecturer concerned.
* I/We authorize the University to test any work submitted by me/us, using text comparison software, for instances of plagiarism. I/We understand this will involve the University or its contractors copying my/our work and storing it on a database to be used in future to test work submitted by others.

Note: 1) The attachment of this statement on any electronically submitted assignments will be deemed to have the same authority as a signed statement.

2) The Group Leader signs the declaration on behalf of all members.

|  |  |
| --- | --- |
| Signature: | Date: |
| E-mail: |  |

ACA-F-020(010611:01)

Page **1** of **2**

|  |
| --- |
| **Feedback/Comments\*** |
| **Main Strengths** |
|  |
|  |
|  |
|  |
|  |
|  |
| **Main Weaknesses** |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| **Suggestions for improvement** |
|  |
|  |
|  |

|  |  |
| --- | --- |
|  | **Student acknowledge feedback/comments** |
|  |
| Grader’s signature | Student’s signature: |
| Date: | Date: |

Note:

1. A soft and hard copy of the assignment shall be submitted.
2. The signed copy of the assignment cover sheet shall be retained by the marker.
3. If the Turnitin report is required, students have to submit it with the assignment. However, departments may allow students up to **THREE** (3) working days after submission of the assignment to submit the Turnitin report. The assignment shall only be marked upon the submission of the Turnitin report.

\*Use additional sheets if required.

Name: StudentID: B1901898

ACA-F-020(010611:01)

Page **2** of **2**

Question 1

**package** Assignment2;

**public** **class** Heap {

**int**[] Heap;//the array for insertion

**private** **int** heapSize;//to keep track of the elements

**public** Heap(**int** size) {

Heap = **new** **int**[size];

heapSize = 0;

}

**boolean** isEmpty() {

**return** heapSize==0;

}

**public** **void** insert(**int** ele) {

**if**(heapSize==Heap.length) {

System.***out***.println("Cannot insert");

}

**else** **if**(heapSize<Heap.length){

Heap[heapSize] = ele;

heapSize++;//increment heap size

heapifyUp();

}

}

**public** **int** max() {

**int** max = Heap[heapSize];

**return** max;

}

**private** **void** heapifyUp() {

//for max-heap

//to place my last-inserted ele in position

**int** eleIndex = heapSize-1;//last position of heap

**int** ele = Heap[eleIndex];//stored to avoid data loss

**int** parentIndex = parent(eleIndex);

**while**(eleIndex>0 && ele > Heap[parentIndex]) {

**int** temp = Heap[parentIndex];//ele in parentIndex placed in temp

Heap[eleIndex] = temp;//temp is placed in eleIndex position

eleIndex = parentIndex;//eleIndex is now parentIndex

Heap[eleIndex] = ele;

}

}

**int** parent(**int** index) {

//index of child

**int** parent = (index - 1)/2;

**return** parent;

}

**int** leftChild(**int** index) {

//index is that of parent

**int** leftChild = 2\*index + 1;

**return** leftChild;

}

**int** rightChild(**int** index) {

**int** rightChild = 2\*index + 2;

**return** rightChild;

}

**public** String print() {

String print = "";

**for**(**int** i = 0; i<heapSize; i++) {

print += Heap[i]+" ";

}

**return** print;

}

}

**package** Assignment2;

**import** java.io.Serializable;

**public** **class** Customer {

**private** String name;

**private** **double** mileage;

**private** **int** years;

**private** **int** sequence;

**public** Customer(String name, **double** M, **int** Y) {

setName(name);

setMileage(M);

setYears(Y);

}

**public** **boolean** equals(Customer c2) {

**return** (**this**.getName().equalsIgnoreCase(c2.getName()));

}

/\*\*

\* **@return** the name

\*/

**public** String getName() {

**return** name;

}

/\*\*

\* **@param** name the name to set

\*/

**public** **void** setName(String name) {

**this**.name = name;

}

/\*\*

\* **@return** the mileage

\*/

**public** **double** getMileage() {

**return** mileage;

}

/\*\*

\* **@param** mileage the mileage to set

\*/

**public** **void** setMileage(**double** mileage) {

**this**.mileage = mileage;

}

/\*\*

\* **@return** the years

\*/

**public** **int** getYears() {

**return** years;

}

/\*\*

\* **@param** years the years to set

\*/

**public** **void** setYears(**int** years) {

**this**.years = years;

}

/\*\*

\* **@return** the sequence

\*/

**public** **int** getSequence() {

**return** sequence;

}

/\*\*

\* **@param** sequence the sequence to set

\*/

**public** **void** setSequence(**int** sequence) {

**this**.sequence = sequence;

}

/\*\*

\* **@return** the priority

\*/

/\*\*

\* **@param** priority the priority to set

\*/

**public** **double** calculatePriority(**int** Sequence) {

setSequence(Sequence);//the passed parameter now the customer's sequence

**double** p = getMileage()/1000 +getYears()-getSequence();

**return** p;

//calculating priority

}

**public** String toString() {

**return** getName()+" (years: "+getYears()+

" and mileage: "+getMileage();

}

}

**package** Assignment2;

**import** java.io.BufferedReader;

**import** java.io.FileNotFoundException;

**import** java.io.FileReader;

**import** java.io.IOException;

**import** java.util.HashMap;

**import** java.util.Scanner;

**import** java.util.Set;

**public** **class** FlightBooked {

**static** HashMap<Integer, Customer> *M* = **new** HashMap<>();

**static** Scanner *kbd* = **new** Scanner(System.***in***);

**static** Heap *priorityQueueM*;

**public** **static** **void** main(String[] args) {

*readFromTextFile*();

/\*storing all the keys in the hashMap inside a set

\* then converting the set to an int array

\*/

Set<Integer> keys = *M*.keySet();

**int** keyArray[] = keys.stream().

mapToInt(Integer::intValue).toArray();

**int** size = keyArray.length;

*priorityQueueM* = **new** Heap(size);

//storing all the values in the array inside the priorityQueue

**for**(**int** key: keyArray) {

*priorityQueueM*.insert(key);

}

//Printing all the passengers according to priority

System.***out***.println("Priority number of all passengers: ");

System.***out***.println(*priorityQueueM*.print());

**do**{

System.***out***.println("Enter a number to view passenger details: ");

**int** priority = *kbd*.nextInt();

Customer c = **null**;

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

**if**(*M*.containsKey(priority)) {

c = *M*.get(priority);

}

**else** {

c = **null**;

}

}

**if**(c==**null**) {

System.***out***.println("Invalid priority number");

}

**else** {

System.***out***.println(c);

}

size--;

}**while**(size!=-1);

}

**public** **static** **void** readFromTextFile() {

System.***out***.println("Reading from file...");

System.***out***.print("File name? ");

String fileName = *kbd*.nextLine();

**try** (FileReader fr = **new** FileReader(fileName);

BufferedReader br = **new** BufferedReader(fr);)

{

String aLine = br.readLine();

**while** (aLine != **null**) {

String[] tokens = aLine.split(",");

//creating a customer obj

Customer c = **new** Customer(tokens[0],

Double.*parseDouble*(tokens[1]),

Integer.*parseInt*(tokens[2]));

**int** s = Integer.*parseInt*(tokens[3]);

**int** p = (**int**)c.calculatePriority(s);

*M*.put(p, c); //now the hashMap has the key-value pair as entry

aLine = br.readLine();

}

br.close();

fr.close();

} **catch** (FileNotFoundException fnfe) {

System.***out***.println("'" + fileName + "' does not exist.");

//fnfe.printStackTrace();

} **catch** (IOException ioe) {

System.***out***.println("Error in reading....");

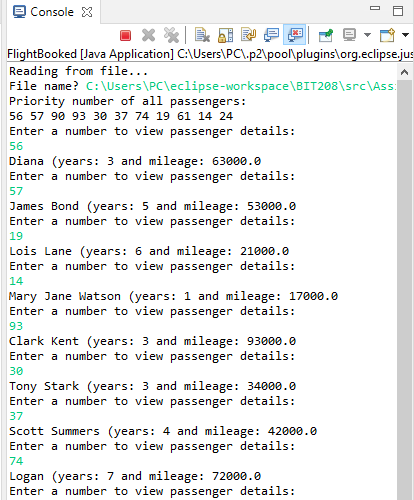
}

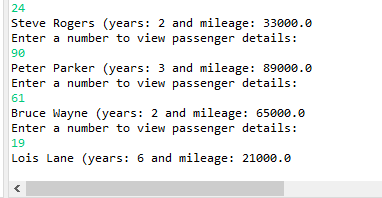
}

Code Description: Code efficiently calculates priority number, also returns correct passenger based on priority number. But the algorithm does not print the list of priority number in ascending order.

Code limitations: Code is designed to automatically terminate once all priority passengers are searched and displayed. Though the code terminates, it take one more element to finally terminate. Code cannot permanently dequeue once an element is already shown. That is why to terminate, one priority number needs to be inserted twice.

Sample output:





Question 2

//node of binary search tree

**public** **class** BSTNode {

**int** key;

BSTNode left, right;

BSTNode(**int** key){

**this**.key = key;

left = right = **null**;

}

**public** String toString() {

**return** "node: "+**this**.key;

}

}

**package** Assignment2;

**public** **class** BinarySearchTree {

BSTNode root; //the very first requirement of a tree

/\*upon creating the BST, the node/root is null\*/

**public** BinarySearchTree() {

root = **null**;

}

**boolean** isEmpty(){

**return** (root==**null**);

}

/\*if there is no node in the tree, create

\* a node with key and declare it root

\* if there is a node, recursively call the insert() function

\*/

**public** **void** insert(**int** key) {

**if**(isEmpty()) {

root = **new** BSTNode(key);//for root

}

**else** {

insert(key, root);

}

}

/\* if key is less than node's key, create left child with key

\* if there is a left child - check in respect of the left child

\* if key is less than left child - create its left child with key

\* if key is greater than left child, create its right child. \*/

/\*if key is greater than node's key, create right child

\* if key has a right child - than check in respect of that child

\* if key is less than right child, create its left child. if

\* key is greater than left child, create its right child.

\*/

**private** **void** insert(**int** key, BSTNode node) {

**if**(key<node.key) {

/\*

\* if left of node empty, create a new node with the

\* key and insert it as left child of node

\*/

**if**(node.left == **null**) {

node.left = **new** BSTNode(key);//base case

}

**else** {

insert(key, node.left);//recursive call

//keep on going until there's an empty

//left child

}

}

**else** **if**(key>node.key) {

**if**(node.right == **null**) {

node.right = **new** BSTNode(key);//base case

}

**else** {

insert(key, node.right);//recursive call

}

}

**else** {

node = **new** BSTNode(key);//overwrite the existing node

}

}

/\*

\* searches a node in respect to root/node

\* if key is less than node's key, searches left

\* if key is more than node's key, searches right

\* if key is equal node's key, searchNode is node

\* if key not found searchNode is null

\*/

**private** BSTNode search(BSTNode node, **int** key) {

BSTNode searchNode = **null**;

**if**(key<node.key) {

searchNode = search(node.left, key);

}

**else** **if**(key>node.key) {

searchNode = search(node.right, key);

}

**else** **if**(key==node.key) {

searchNode = node;

}

**else** {

searchNode = **null**;

}

**return** searchNode;

}

**public** BSTNode search(**int** key) {

**if**(isEmpty()) {

**return** **null**;

}

**else** {

**return** search(root, key);

}

}

/\*BST algorithm follows inOrder() for insertion, search and deletion\*/

**public** **void** inOrder() {

**if**(isEmpty()) {

System.***out***.println("Empty tree");

}

**else** {

inOrder(root);

}

}

**private** **void** inOrder(BSTNode node) {

//<left><root><right>

**if**(node!=**null**) {

inOrder(node.left);

System.***out***.println(node.key+" ");

inOrder(node.right);

}

}

/\*deleted for three scenarios

\* 1)key is less than root

\* 2)key is greater than root

\* 3)either left child or right child is null or there is no child

\*/

**public** BSTNode delete(BSTNode node, **int** key) {

//create a temp BSTNode

BSTNode temp = **null**;

**if**(node==**null**) {

**return** **null**;

}

//1

**if**(key<node.key) {

node.left = delete(node.left, key);

}

//2

**else** **if**(key>node.key) {

node.right = delete(node.right, key);

}

//3

**else** {

**if**(node.left==**null** || node.right==**null**) {

//if node.left null, then temp is node.right and vice versa

temp = node.left == **null**? node.right : node.left;

**if**(temp == **null**) {

**return** node;

}

**else** {

**return** temp;

}

} **else** {

//fetches its child, then makes its key its node-key

BSTNode inOrderChild = getChild(node);

node.key = inOrderChild.key;

}

}

**return** node;

}

**public** BSTNode getChild(BSTNode node) {

**if**(node==**null**) {

**return** node;

}

BSTNode temp = node.right;//changge

**while**(temp.left != **null**) {

temp = temp.left;

}

**return** temp;

}

}

**package** Assignment2;

**public** **class** Student {

**private** **int** id;

**private** String name;

**private** **double** cgpa;

**public** Student(**int** id, String name, **double** cgpa) {

setId(id);

setName(name);

setCgpa(cgpa);

}

/\*\*

\* **@return** the id

\*/

**public** **int** getId() {

**return** id;

}

/\*\*

\* **@param** id the id to set

\*/

**public** **void** setId(**int** id) {

**this**.id = id;

}

/\*\*

\* **@return** the name

\*/

**public** String getName() {

**return** name;

}

/\*\*

\* **@param** name the name to set

\*/

**public** **void** setName(String name) {

**this**.name = name;

}

/\*\*

\* **@return** the cgpa

\*/

**public** **double** getCgpa() {

**return** cgpa;

}

/\*\*

\* **@param** cgpa the cgpa to set

\*/

**public** **void** setCgpa(**double** cgpa) {

**this**.cgpa = cgpa;

}

**public** String toString() {

**return** getId()+" "+getName()+" "+getCgpa();

}

}

**package** Assignment2;

**import** java.io.BufferedReader;

**import** java.io.FileNotFoundException;

**import** java.io.FileReader;

**import** java.io.IOException;

**import** java.util.ArrayList;

**import** java.util.Scanner;

**public** **class** StudentBST {

**static** Scanner *scan* = **new** Scanner(System.***in***);

**static** ArrayList<Student> *students*= **new** ArrayList<>(); //storing all the students read

**static** ArrayList<Student> *studentsStored*= **new** ArrayList<>();

**static** BSTNode *root*;

**static** BinarySearchTree *studentBST*;

**static** ArrayList<Integer> *addChecker*;

**public** **static** **void** main(String[] args) {

*readFromFile*();

*studentBST* = **new** BinarySearchTree(); //new BST

*addChecker* = **new** ArrayList<>();//to keep track of every studentID entered in BST

**int** choice;

**do** {

choice = *mainMenu*();

**switch**(choice) {

**case** 1:

*addAStudent*();

**break**;

**case** 2:

*removeStudent*();

**break**;

**case** 3:

*searchStudent*();

**break**;

**case** 4:

*displayAllStudents*();

**break**;

}

}**while**(choice!=0);

}

**public** **static** **int** mainMenu() {

System.***out***.println("Welcome to Student system!");

System.***out***.println("--------------------------");

System.***out***.println("1. Add a student to BST");

System.***out***.println("2. Remove a student from BST");

System.***out***.println("3. Search a student from BST");

System.***out***.println("4. Display all students from BST");

System.***out***.println();

**int** choice = *scan*.nextInt();

**return** choice;

}

/\*

\* all student IDs display. choose an ID to add

\*/

**public** **static** **void** addAStudent() {

System.***out***.println("All student IDs: ");

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

**int** id = *students*.get(i).getId();

System.***out***.print(id+" ");

}

System.***out***.println();

System.***out***.println("Select id to continue");

**int** id\_input = *scan*.nextInt();

**int** count = 0;//to keep track how many student was found

Student temp = **null**;

**for**(Student st: *students*) {

**if**(id\_input == st.getId()) {

System.***out***.println("Student found: "+st);

temp = st; //storing the latest student for ID

count++;

}

}

**if**(count>=1) {

*studentBST*.insert(temp.getId());

*addChecker*.add(temp.getId());//stores the newly added id to arraylist

*studentsStored*.add(temp);//only adding after it is added to BST

System.***out***.println("Student "+temp.getName()+" added to BST successfully");

**if**(*studentBST*.isEmpty()) {

*root* = **new** BSTNode(temp.getId());//first node is root

}

}

**else** {

System.***out***.println("Invalid id. Try again");

}

}

//can only remove ID that belong in BST

//addChecker keeps track of all Nodes added to BST

**public** **static** **void** removeStudent() {

**try** {

System.***out***.print("Student in BST: ");

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

**int** id = *studentsStored*.get(i).getId();

System.***out***.println(id+" ");

}

System.***out***.println();

System.***out***.println("Select id to continue");

**int** id\_input = *scan*.nextInt();

**for**(Student st: *studentsStored*) {

**if**(id\_input == st.getId()) {

*studentBST*.delete(*root*, id\_input);

*studentsStored*.remove(st);//student removed from arrayList

*addChecker*.remove(id\_input);//id removed from arraylist

}

}

}**catch**(Exception e) {

System.***out***.println("Student removed. Select any integer to continue"); }

}

**public** **static** **void** searchStudent() {

System.***out***.println("All ids: ");

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

System.***out***.println(*addChecker*.get(i)+" ");

}

System.***out***.println("Enter an ID to check: ");

**int** id\_input = *scan*.nextInt();

//using user input to search BST

BSTNode studentID = *studentBST*.search(id\_input);

**if**(studentID == **null**) {

System.***out***.println("id not found");

}

**else** {

**for**(Student st: *studentsStored*) {

**if**(id\_input == st.getId()) {

System.***out***.println("Student found: "+st);

}

}

}

}

/\*displays all the student ID in-order,

\* displays details of particular student, when ID entered

\*/

**public** **static** **void** displayAllStudents() {

System.***out***.println("All student IDs in-order: ");

*studentBST*.inOrder();

**int** id\_input;

**do**{

System.***out***.println("Enter id to view details(-1 to exit): ");

id\_input = *scan*.nextInt();

**for**(Student st : *studentsStored*) {

**if**(id\_input == st.getId()) {

System.***out***.println(st);

}

}

}**while**(id\_input!= -1);

}

//students read and stored in an arrayList

**public** **static** **void** readFromFile() {

System.***out***.println("Filename? ");

String filename = *scan*.nextLine();

**try**(FileReader fr = **new** FileReader(filename);

BufferedReader br = **new** BufferedReader(fr);)

{

String aLine = br.readLine();

**while** (aLine != **null**) {

String[] tokens = aLine.split(" ");

//creating a student obj

Student st = **new** Student(Integer.*parseInt*(tokens[0]),

tokens[1], Double.*parseDouble*(tokens[2]));

//storing the read student into arraylist

*students*.add(st);

aLine = br.readLine();

}

br.close();

fr.close();

}**catch**(FileNotFoundException fnfe) {

System.***out***.println(filename+" does not exist");

}**catch** (IOException ioe) {

System.***out***.println("Error in reading....");

}

}

}

Code Description:

Binary Search Tree only stored student ID, not the whole student data. So, for the algorithmic accuracy – some additional datasets such as Arraylists were implemented. Three separate arrayLists – students, studentsStored and addCheck were created for various purpose.

students = to store the read student Objects from records.txt (only required for adding all students)

studentsStored = to store the student Objects after they were added to BST

addCheck = to store the student IDs inserted in BST

some variables :

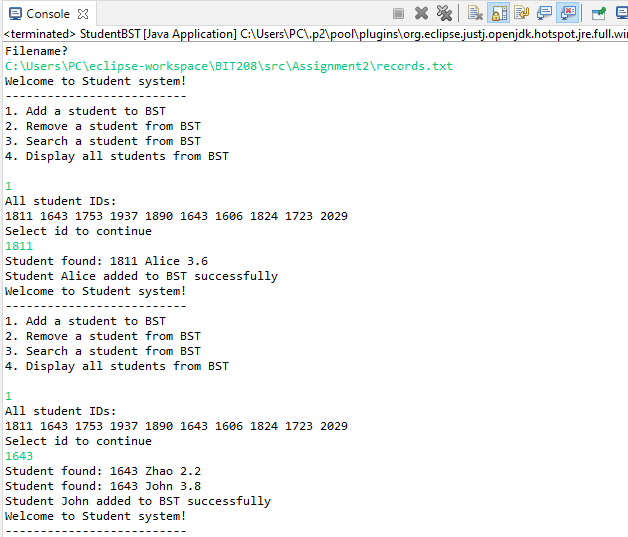
count = to count the student objects found for a specific id.

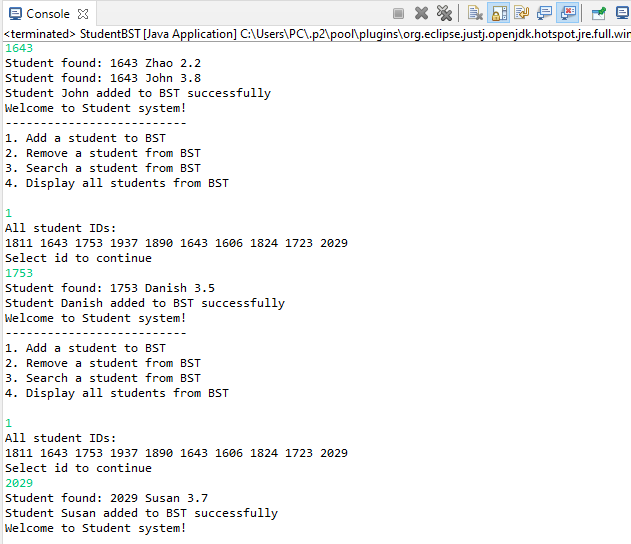
Student temp = to store the last Student object found for a student ID

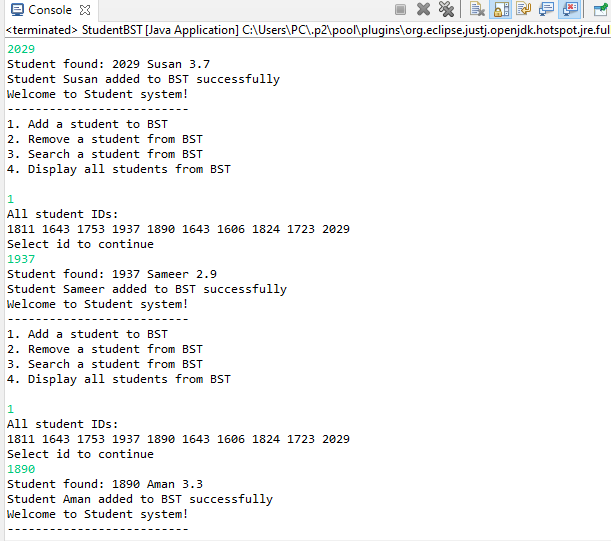
Justification for temp: when a node is inserted while it already exists in the BST, the node is not rejected rather overridden. So, to maintain the algorithmic integrity, only temp was inserted into BST and for later operations(remove, search and display) – only studentsStored and addChecker was used as they stored temp Student objects.

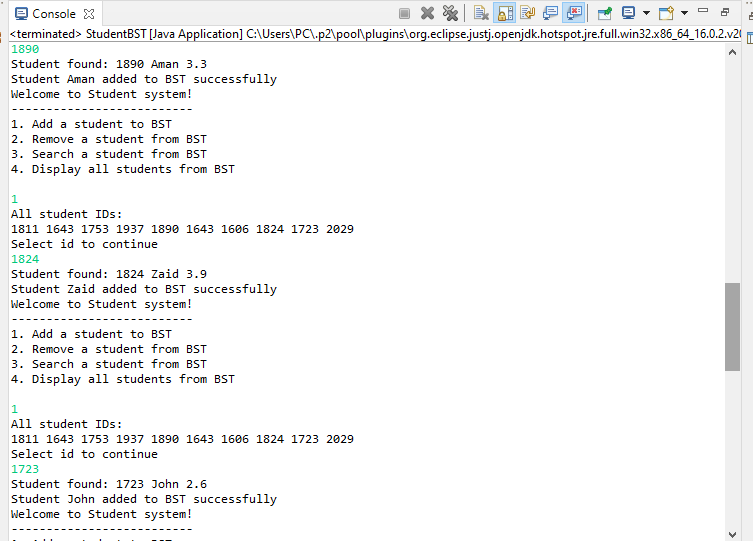
Code limitations: it needs to update value in multiple collections – such as BST, arraylists etc. It can insert and display in-order. Also can search elements. But delete algorithm might not be efficient.

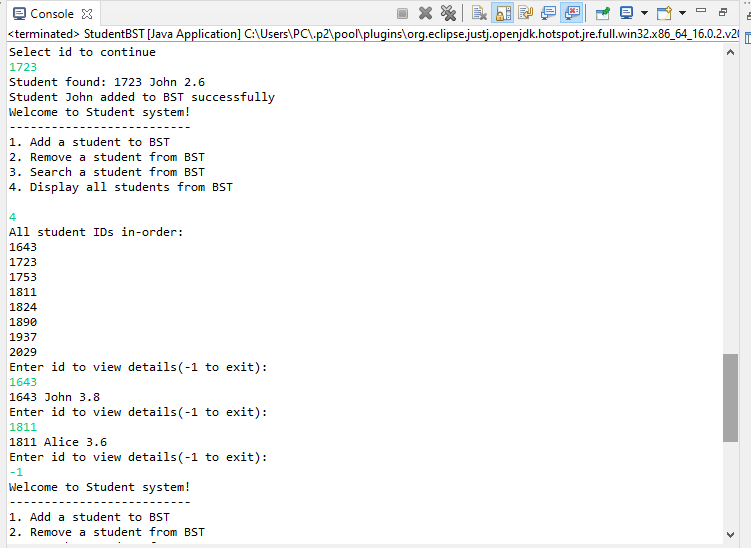
Sample output:

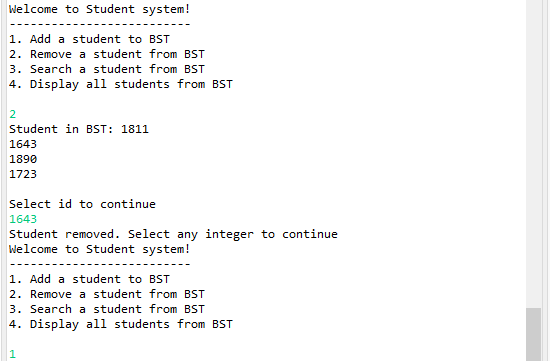


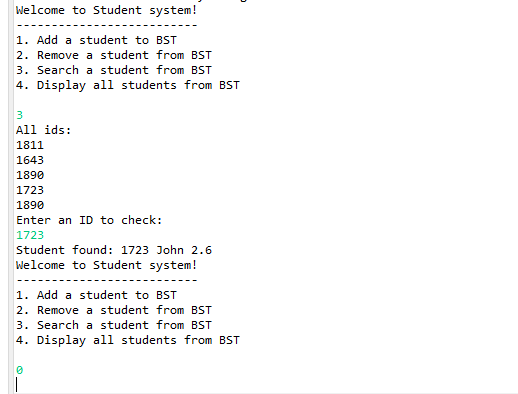












Question 3

Three separate programs for three sorts. Each program has two separate sorting – one for already sorted list and another for randomly-built list. To calculate time for each list, one list had to be commented out.

a)

**package** Assignment2;

**import** java.util.Random;

**public** **class** MergeSort {

**static** Random *rand* = **new** Random();

**static** **int**[] *array* = **new** **int**[1000];

**static** **int**[] *arrayRandom* = **new** **int**[1000];

**public** **static** **void** main(String[] args) {

/\*to create an array from 0 - 1000

\* where elements are already sorted

\*/

**for**(**int** i = 0; i<1000; i++) {

*array*[i] = i;

}

//to create random array

**for**(**int** i = 0; i<1000; i++) {

**int** randomNum = *rand*.nextInt(1000);

*arrayRandom*[i] = randomNum;

}

/\*//printed before sorting

System.out.println("An array of integers(before sorting:) ");

for(int num: array) {

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

}

System.out.println();

mergeSort(array);//method is called to sort array

//printed after sorting

System.out.println("An array of integers(after sorting:) ");

for(int numSorted: array) {

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

}\*/

//printed before sorting

System.***out***.println("An array of random integers(before sorting:) ");

**for**(**int** num: *arrayRandom*) {

System.***out***.print(num+" ");

}

System.***out***.println();

*mergeSort*(*arrayRandom*);//method to sort array

//printed after sorting

System.***out***.println("An array of random integers(after sorting:) ");

**for**(**int** numSorted: *arrayRandom*) {

System.***out***.print(numSorted+" ");

}

}

**public** **static** **void** mergeSort(**int**[] array) {

*mergeSort*(array, 0, array.length-1);

}

**public** **static** **void** mergeSort(**int**[] arr, **int** lowIndex, **int** highIndex) {

**if**(lowIndex<highIndex) {

**int** midIndex = (lowIndex+highIndex)/2;

//recursive call for lower half

*mergeSort*(arr, lowIndex, midIndex);

//recurcive call for upper half

*mergeSort*(arr, midIndex+1, highIndex);

//helper method to merge the sorted halves

*merge*(arr, lowIndex, midIndex, highIndex);

}

}

//helper method that combines two temporary arrays

//with their sorted elements

**public** **static** **void** merge(**int**[] arr, **int** lowIndex, **int** midIndex, **int** highIndex) {

**int** leftSize = midIndex - lowIndex+1;

**int** rightSize = highIndex - midIndex;

//temporary arrays with size

**int**[] leftArr = **new** **int**[leftSize];

**int**[] rightArr = **new** **int**[rightSize];

**for**(**int** i = 0; i<leftSize; i++) {

leftArr[i] = arr[lowIndex+i];

}

**for**(**int** j = 0; j<rightSize; j++) {

rightArr[j] = arr[midIndex+1+j];//cannot take mid as it is a part of leftArray

}

/\*after building the arrays, we sort them

\* i for left and j for right arrays, and k is to

\* compare with main array

\*/

**int** i = 0; **int** j = 0; **int** k = lowIndex;

**while**(i<leftSize && j<rightSize) {

/\*compare the first element of each array,

\* whichever is the smallest, place it into parametric array

\* then increment i&k or j&k

\* only applicable if both arrays are not empty

\*/

**if**(leftArr[i]<rightArr[j]) {

arr[k] = leftArr[i];

i++;

k++;

}

**else** {

arr[k] = rightArr[j];

j++;

k++;

}

}

/\*if rightArray has no more elements, but left-array has

\* unvisited elements

\*/

**while**(i<leftSize) {

arr[k] = leftArr[i];

i++;

k++;

}

/\*if leftArray has no more elements, but right-array has

\* unvisited elements

\*/

**while**(j<rightSize) {

arr[k] = rightArr[j];

j++;

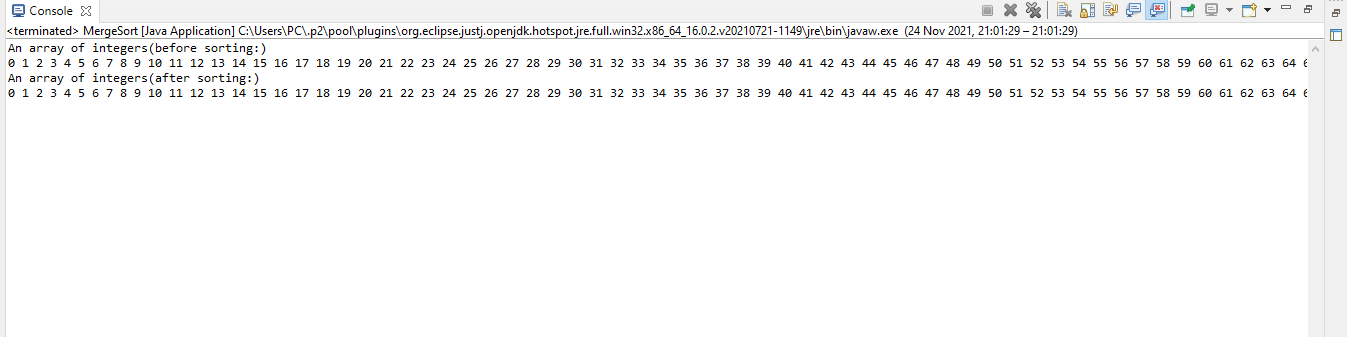
k++;

}

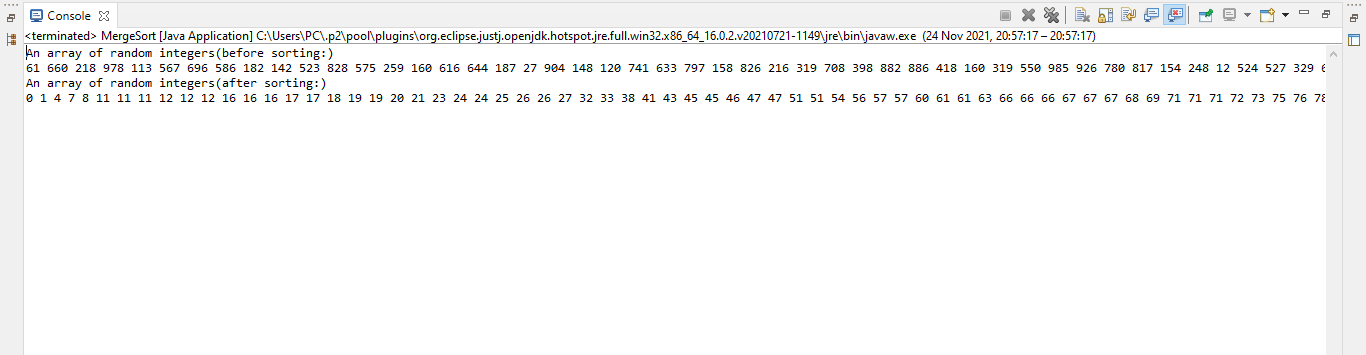
}

}

Sample output for sorted integers:



Sample output for random integers:



b)

**package** Assignment2;

**import** java.util.Random;

**public** **class** QuickSort {

**static** Random *rand* = **new** Random();

**static** **int**[] *array* = **new** **int**[1000];

**static** **int**[] *arrayRandom* = **new** **int**[1000];

**public** **static** **void** main(String[] args) {

/\*to create an array from 0 - 1000

\* where elements are already sorted

\*/

**for**(**int** i = 0; i<1000; i++) {

*array*[i] = i;

}

//to create random array

**for**(**int** i = 0; i<1000; i++) {

**int** randomNum = *rand*.nextInt(1000);

*arrayRandom*[i] = randomNum;

}

/\* //array is printed before sorting

System.out.println("An array of integers(before sorting:) ");

for(int num: array) {

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

}

System.out.println();

quickSort(array); //sorting function is called to sort the array

//sorted array is printed

System.out.println("An array of integers(after sorting:) ");

for(int numSorted: array) {

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

}\*/

//array is printed before sorting

System.***out***.println("An Array of random integers(before sorting:) ");

**for**(**int** num: *arrayRandom*) {

System.***out***.print(num+" ");

}

System.***out***.println();

//function is called to sort array, then sorted array is printed

System.***out***.println("An array of integers(after sorting:) ");

*quickSort*(*arrayRandom*);

**for**(**int** numSorted: *arrayRandom*) {

System.***out***.print(numSorted+" ");

}

}

//method to call

**public** **static** **void** quickSort(**int**[] arr) {

*quickSort*(arr, 0, *array*.length-1);

}

**private** **static** **void** quickSort(**int** array[], **int** low, **int** high) {

/\*

\* pivot is a crucial number for quickSort. i and j are

\* made to store temporary start/low and end/high value at each partition

\* temp will be used for swapping value

\*/

/\*

\* here partition is being performed without creating another

\* collection list - the algorithm of partition being performed in

\* one one collection(array) though recursion by changing i and j

\* value with each loop

\*/

**int** pivot, i, j, temp;

**if**(low<high) {

pivot = (low+high)/2;//mid-value is stored as pivot

i = low;

j = high;

/\*

\* for sort to excecute, low has to be less than high

\* i=temp-low and j=temp-high

\*/

/\*

\* value at i has to be less or equal value at pivot position

\*

\*/

**while**(i < j) {

**while**(array[i] <= array[pivot] && i < high) {

i++;

}

/\*as low is lowest position, next recursion of low has

\*to be higher than low, sow i is incremented

\*/

**while**(array[j] > array[pivot]) {

j--;

}

/\*as high is highest position, next recursion of high has

\*to be lower than high, so j is decremented

\*/

**if**(i < j) {

temp = array[i];

array[i] = array[j];

array[j] = temp;

}

/\* elements between i and j position are swapped

\* thus two separate sorted list of

\* low and high elements created, in respect to pivot

\*/

}

/\*

\* now element between j and pivot position is swapped

\* as upon exiting, j is less than i

\*/

temp = array[pivot];

array[pivot] = array[j];

array[j] = temp;

/\*recursively calling quicksort

\* for two partition - L and G

\*/

*quickSort*(array, low, j-1);

*quickSort*(array, j+1, high);

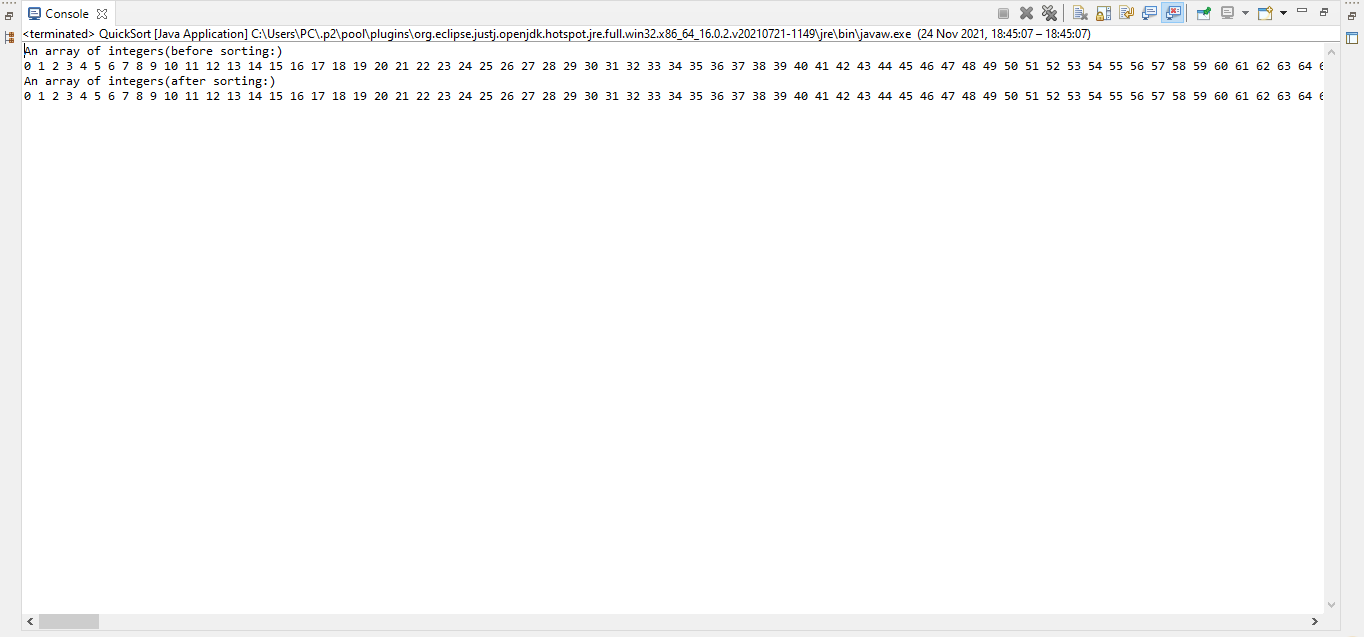
}

}

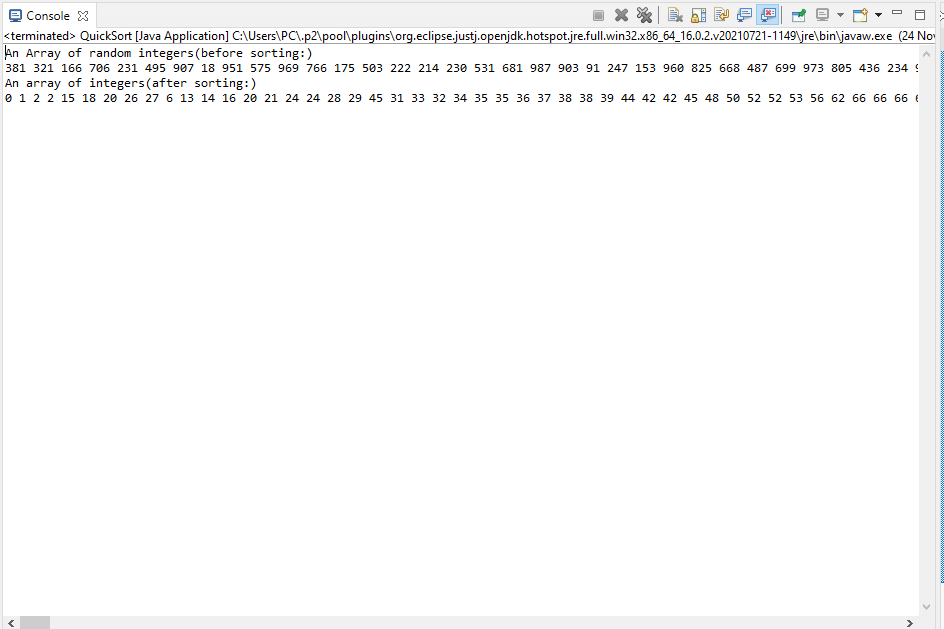
}

sample output:

for already sorted array



For randomly generated array(including repeating numbers)



c)

**package** Assignment2;

**import** java.util.Random;

**public** **class** sortArrays {

**static** Random *rand* = **new** Random();

**static** **int**[] *array* = **new** **int**[1000];

**static** **int**[] *arrayRandom* = **new** **int**[1000];

**public** **static** **void** main(String[] args) {

/\*to create an array from 0 - 1000

\* where elements are already sorted

\*/

**for**(**int** i = 0; i<1000; i++) {

*array*[i] = i;

}

//to create random array

**for**(**int** i = 0; i<1000; i++) {

**int** randomNum = *rand*.nextInt(1000);

*arrayRandom*[i] = randomNum;

}

//printed before sorting

/\* System.out.println("An array of integers(before sorting:) ");

for(int num: array) {

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

}

System.out.println();

insertionSort(array);//method is called to sort array

//printed after sorting

System.out.println("An array of integers(after sorting:) ");

for(int numSorted: array) {

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

}\*/

//printed before sorting

System.***out***.println("An array of random integers(before sorting:) ");

**for**(**int** num: *arrayRandom*) {

System.***out***.print(num+" ");

}

System.***out***.println();

*insertionSort*(*arrayRandom*);//method to sort array

//printed after sorting

System.***out***.println("An array of random integers(after sorting:) ");

**for**(**int** numSorted: *arrayRandom*) {

System.***out***.print(numSorted+" ");

}

}

/\*insertion sort has two arrays - a sorted array and an unsorted array

\* initially the sorted array is the first element a[0], and key is the

\* element right next to it. key is compared with the sorted element,

\* if key is smaller than a[0], swap

\* and the rest of the array is in unsorted array

\*/

**public** **static** **void** insertionSort(**int**[] array) {

**int** i, j, key, temp;

**for**(i = 1; i<array.length; i++) {

key = array[i];

j = i - 1; //j is the element of sorted list

**while**(j >= 0 && key < array[j]) {

temp = array[j];

array[j] = array[j+1];

array[j+1] = temp;

j--;

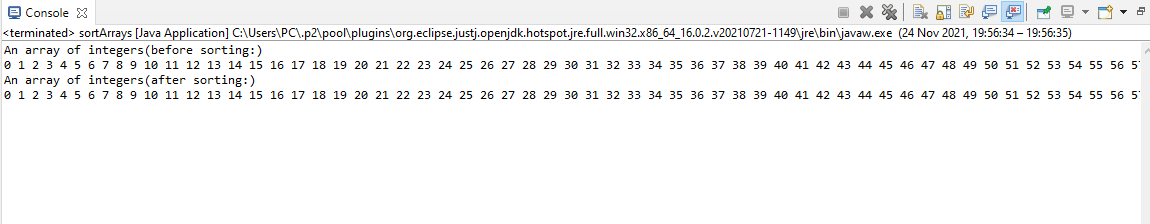
}

}

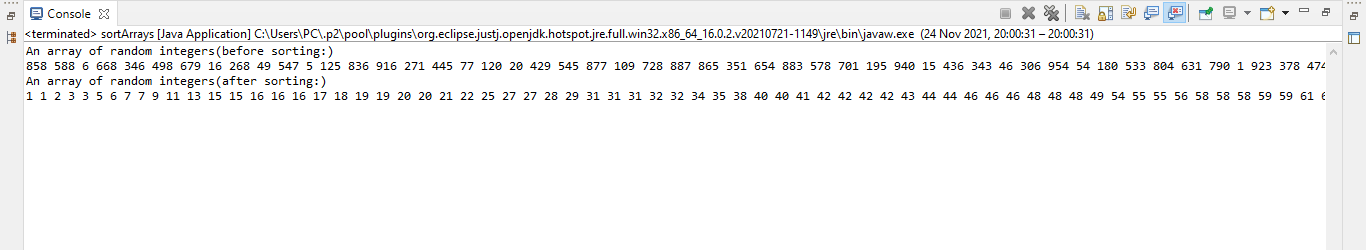
}

}

Sample output for already sorted integers



Sample output for random integers



d)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Algorithm** | **Worst-Case** | **Average-Case** | **Best-Case** | **Actual time on Sorted list** | **Actual time on Random list** |
| Mergesort | O(n log n) | O(n log n) | O(n log n) | 1.78 sec | 1.05 sec |
| Insertion Sort | O(n^2) | O(n^2) | O(n^2) | 2.10 sec | 3.05 sec |
| Quick Sort | O(n2) | O(n2) | O(n log n) | 2.43 sec | 2.43 sec |

e)

**MergeSort:** In merge sort, in each recursive call the sequence is divided in n/2 – n is the size of the sequence to be divided. Swapping only starts once each sequence has either 0 or 1 item. As merge-sort structure is a binary tree, the depth of merge-sort is O(log n). At each level the amount of work taken is O(n).

So, total time complexity for merge sort is O(n log n).

**Insertion sort:** Insertion sort is not an efficient algorithm as to sort, it uses nested loops(as shown in the code above – it used a while-loop inside a for-loop). It is useful for small data sets. But not useful for large sets. Worst case and best best case for Insertion sort is O(n^2) .(for loop runs n-time, while loop runs n-time so total time n^2).

Average case is also O(n^2)

**QuickSort:** Worst-case for pivot occurs when pivot = low or pivot = high;

Because then one of L or G will have n-1 element and the other will have 0 element

So the worst-case running time of quickSort is

= depth\*sum of running time = (n-1)(n+1)/2 = (n^2 – 1)/2 = O(n^2) [as in time-complexity, constant is omitted]

Best-case for quick sort is when pivot = 3/4n. In this case, every step will take O(n) time.

Depth will be O(log n) ].

So, best-case running time for quick sort=O(log n)\*O(n) = O(n log n)

Eve though, the best case is not always guaranteed as pivot is chosen arbitrarily and the sorting depends on the pivot.

Average-case: the probability of the best case happening is ½ . Also, for the worst-case also ½. So, average case running time

= O(n^2)+O(n log n)/2 = (n^2 + nlog n)/2 = O(n ^2)(as low-order terms are omitted)

Comparing all three algorithms, based on time complexity and found time – the conclusion that can be drawn is - mergeSort is better than insertion sort and quick sort. As for any given set of integers/data – it will always have the same time complexity. Also, it is very efficient for large set of data so, merge-sort is a better algorithm for sorting.