Dylan T Carlson

1161653

Lab 111

11/18/18

**Comparator Interface:**

/\*\*

\* Data Structures & Algorithms 6th Edition

\* Goodrick, Tamassia, Goldwasser

\* Page 363

\*

\* An implementation of the Comparator interface.

\*/

public interface Comparator<A> {

int compare(A a, A b);

}

**Other Comparators:**

/\*\*

\* This is a comparator that implements the Comparator interface and

\* compares employees based on their hire date.

\* @author dylca

\*/

public class HiredComparator implements Comparator<Employee>{

@Override

public int compare(Employee e1, Employee e2){

if( e1.getHired() > e2.getHired() )

return 1;

if (e1.getHired() == e2.getHired())

return 0;

return -1;

}

}

/\*\*

\* This is a comparator that implements the Comparator interface and

\* compares employees based on their department.

\* @author dylca

\*/

public class DeptComparator implements Comparator<Employee>{

@Override

public int compare(Employee e1, Employee e2){

if( e1.getDept() > e2.getDept() )

return 1;

if (e1.getDept() == e2.getDept())

return 0;

return -1;

}

}

/\*\*

\* This is a comparator that implements the Comparator interface and

\* compares employees based on their id.

\* @author dylca

\*/

public class IdComparator implements Comparator<Employee> {

@Override

public int compare(Employee e1, Employee e2){

if( e1.getId() > e2.getId() )

return 1;

if (e1.getId() == e2.getId())

return 0;

return -1;

}

}

/\*\*

\* This is a comparator that implements the Comparator interface and

\* compares employees based on their name.

\* @author dylca

\*/

public class NameComparator implements Comparator<Employee> {

public int compare(Employee e1, Employee e2){

return e1.getName().compareTo(e2.getName());

}

}

**Sort Class:**

/\*\*

\* Multiple code fragments were used for some of these methods, the headers

\* above the methods say where they were from. Anyways, the Sort class includes

\* multiple generic sort methods.

\* @author dylca

\*/

public class Sort {

/\*\*

\* This simple bubble sort algorithm is a generic implementation

\* of the bubble sort algorithm.

\* @param <E>

\* @param data

\* @param comp

\*/

public static <E> void simpleBubbleSort(E[] data, Comparator<E> comp){

boolean isSorted = false;

E temp = null;

for (int k = 0; k < data.length; k++){

isSorted = true;

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

if( (comp.compare(data[i], data[i+1])) > 0){

temp = data[i];

data[i] = data[i+1];

data[i+1] = temp;

isSorted = false;

}

}

if(isSorted)

break;

}

}

//Taken from Java Illuminated and made Generic

public static <E> void insertionSort(E[] data, Comparator<E> comp){

int j;

E temp;

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

j = i;

temp = data[i];

while(j != 0 && (comp.compare(data[j-1], temp)) > 0){

data[j] = data[j-1];

j--;

}

data[j] = temp;

}

}

//Taken from Java Illuminated and made Generic

public static <E> void selectionSort(E[] data, Comparator<E> comp){

E temp;

int max;

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

max = indexOfLargestElement(data, data.length - i, comp);

temp = data[max];

data[max] = data[data.length - i - 1];

data[data.length - i - 1] = temp;

}

}

//part of selectionsort

private static <E> int indexOfLargestElement(E[] data, int size, Comparator<E> comp){

int index = 0;

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

if(comp.compare(data[i], data[index]) > 0)

index = i;

}

return index;

}

/\*\*

\* Data Structures & Algorithms 6th Edition

\* Goodrick, Tamassia, Goldwasser

\* Code Fragment 12.1 and 12.2

\*

\* An implementation of the merge and mergeSort methods

\*/

/\*\*Merge contents of arrays S1 and S2 into properly sized array S. \*/

public static <K> void merge(K[] S1, K[] S2, K[]S, Comparator<K> comp){

int i = 0, j = 0;

while(i + j < S.length){

if(j == S2.length || (i < S1.length && comp.compare(S1[i], S2[j]) < 0))

S[i+j] = S1[i++]; //copy ith element of S1 and increment i

else

S[i+j] = S2[j++]; //copy jth element of S2 and increment j

}

}

/\*\* Merge-sort contents of array S. \*/

public static <K> void mergeSort(K[] S, Comparator<K> comp){

int n = S.length;

if(n < 2) return; //array is trivially sorted

//divide

int mid = n/2;

K[] S1 = copyOfRange(S, 0, mid); //copy of first half

K[] S2 = copyOfRange2(S, mid, n); //copy of second half

//conquer (with recursion)

mergeSort(S1, comp); //sort copy of first half

mergeSort(S2, comp); //sort copy of second half

//merge results

merge(S1, S2, S, comp); //merge sorted halves back into original

}

/\*\*

\* The following two methods were created since Arrays could not be imported.

\* @param <E>

\* @param S

\* @param low

\* @param mid

\* @return

\*/

private static <E> E[] copyOfRange(E[] S, int low, int mid){

E[] newS = (E[]) new Object[mid - low];

while(low < mid){

newS[low] = S[low];

low++;

}

return newS;

}

private static <E> E[] copyOfRange2(E[] S, int mid, int n){

E[] newS = (E[]) new Object[n-mid];

int i = 0;

while(mid < n){

newS[i] = S[mid];

i++;

mid++;

}

return newS;

}

/\*\*

\* Data Structures & Algorithms 6th Edition

\* Goodrick, Tamassia, Goldwasser

\* Code Fragment 12.5

\*

\* An implementation of the quickSort method

\*/

/\*\* Quick-sort contents of a queue. \*/

public static <K> void quickSort(Queue<K> S, Comparator<K> comp){

int n = S.size();

if( n < 2) return; // queue is trivially sorted

//divide

K pivot = S.first(); // using first as arbitrary pivot

Queue<K> L = new LinkedQueue<>();

Queue<K> E = new LinkedQueue<>();

Queue<K> G = new LinkedQueue<>();

while(!S.isEmpty()){ // divide original into L, E, G

K element = S.dequeue();

int c = comp.compare(element, pivot);

if (c < 0) // element is less than pivot

L.enqueue(element);

else if ( c ==0 ) // element is equal to pivot

E.enqueue(element);

else // elelemtn is greater than pivot

G.enqueue(element);

}

//conquer

quickSort(L, comp); // sort elements less than pivot

quickSort(G, comp); // sort elements greater than pivot

//concatenate results

while(!L.isEmpty())

S.enqueue(L.dequeue());

while(!E.isEmpty())

S.enqueue(E.dequeue());

while(!G.isEmpty())

S.enqueue(G.dequeue());

}

/\*\*

\* Since quickSort needs a Queue passed to it, this method converts the

\* array to a queue, and the queue back to an array.

\* @param <E>

\* @param S

\* @param comp

\*/

public static <E> void quickSort(E[] S, Comparator<E> comp){

Queue<E> converted = new LinkedQueue<>();

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

converted.enqueue(S[i]);

}

quickSort(converted, comp);

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

S[i] = converted.dequeue();

}

}

/\*\*

\* The following three methods are radix sorts with different amounts of comparators.

\* The order of most value is from left to right.

\* @param <E>

\* @param data

\* @param comp1

\* @param comp2

\*/

public static <E> void radixSort(E[] data, Comparator<E> comp1, Comparator<E> comp2){

mergeSort(data, comp2);

mergeSort(data, comp1);

}

public static <E> void radixSort(E[] data, Comparator<E> comp1, Comparator<E> comp2, Comparator<E> comp3){

mergeSort(data, comp3);

mergeSort(data, comp2);

mergeSort(data, comp1);

}

public static <E> void radixSort(E[] data, Comparator<E> comp1, Comparator<E> comp2, Comparator<E> comp3, Comparator<E> comp4){

mergeSort(data, comp4);

mergeSort(data, comp3);

mergeSort(data, comp2);

mergeSort(data, comp1);

}

}

**Client Class:**

/\*\*

\* This Client class creates an employee list of 100000 employees

\* and uses it to test multiple sorting algorithms.

\* @author dylca

\*/

public class Client {

public static void main(String[] args) {

Employee employeeList[], temp1[], temp2[], temp3[], temp4[], temp5[];

employeeList = new Employee[100000];

/\*\* Creating X amount of Employees\*\*/

for(int i = 0; i < 100000; i++){

employeeList[i] = new Employee();

}

long startTime = 0;

long endTime = 0;

long elasped = 0;

temp1 = employeeList;

temp2 = employeeList;

temp3 = employeeList;

temp4 = employeeList;

temp5 = employeeList;

/\*\* Comparators \*\*/

IdComparator idComp = new IdComparator();

DeptComparator deptComp = new DeptComparator();

HiredComparator hiredComp = new HiredComparator();

NameComparator nameComp = new NameComparator();

startTime = System.currentTimeMillis();

Sort.mergeSort(temp1, nameComp);

endTime = System.currentTimeMillis();

elasped = endTime - startTime;

System.out.println("Time for a Merge Sort on name: "+ elasped);

startTime = System.currentTimeMillis();

Sort.quickSort(temp2, deptComp);

endTime = System.currentTimeMillis();

elasped = endTime - startTime;

System.out.println("Time for a Quick Sort on department: "+ elasped);

startTime = System.currentTimeMillis();

Sort.simpleBubbleSort(temp3, idComp);

endTime = System.currentTimeMillis();

elasped = endTime - startTime;

System.out.println("Time for a Bubble Sort on id: "+ elasped);

startTime = System.currentTimeMillis();

Sort.insertionSort(temp4, nameComp);

endTime = System.currentTimeMillis();

elasped = endTime - startTime;

System.out.println("Time for a Insertion Sort on name: "+ elasped);

startTime = System.currentTimeMillis();

Sort.selectionSort(temp5, idComp);

endTime = System.currentTimeMillis();

elasped = endTime - startTime;

System.out.println("Time for a Selection Sort on name: "+ elasped);

startTime = System.currentTimeMillis();

Sort.radixSort(employeeList, deptComp, hiredComp, nameComp );

endTime = System.currentTimeMillis();

elasped = endTime - startTime;

System.out.println("Time for a Radix Sort sorted by department, hire date, and then name: "+ elasped);

}

}

**Employee Class:**

import java.util.Random;

/\*\*

\* This Employee Class is for the Lab111 assignment and the Employee

\* is completely random.

\* @author dylca

\*/

public class Employee {

private int id;

private int dept;

private int hired;

private String name;

private int length;

Random rand = new Random();

public Employee(){

id = rand.nextInt(100000000);

dept = rand.nextInt(5) + 1;

hired = rand.nextInt(11) + 2008;

length = rand.nextInt(6) + 5;

//97 and 122

name = "";

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

char a = ' ';

a = (char)(rand.nextInt(26) + 97);

name += a;

}

}

public int getId(){

return id;

}

public int getDept(){

return dept;

}

public int getHired(){

return hired;

}

public String getName(){

return name;

}

public String toString(){

return getClass().getName() + id + dept + hired + name;

}

public boolean equals(Object o){

if(!(o instanceof Employee))

return false;

Employee e = (Employee) o;

return e.id == id

&& e.dept == dept

&& e.hired == hired

&& e.name.equals(name);

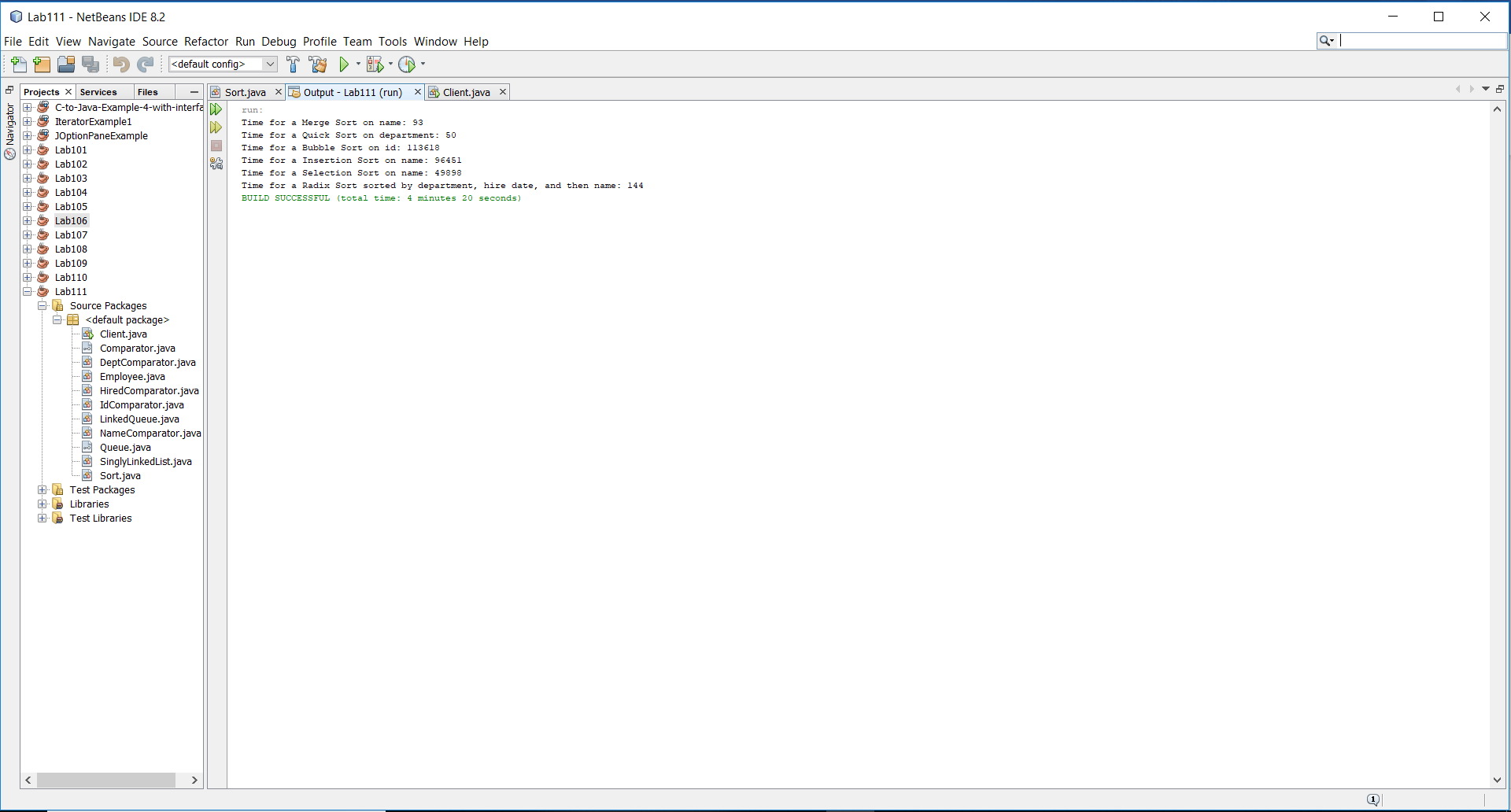
}

}

**Output Screenshot:**

In Milliseconds.

It says selection sort on name, but that was a mistake, it was actually on ID like the assignment says. Also, it is correct in the code.

****