Lab110

Steven Glasford

April 25, 2019

1 Client.java

```
/**
 * A main class controlling everything rekcufrehtom bitch.
 * @author Steven Glasford
 * @version 1
 */
public class Client {

    /**
     * @param args No command line shit,
     */
    public static void main(String[] args) {
         //make the max time for the shitfuckers to run, in this case the
        //maximum amount of time that will pass for any of the sorting
        //algorithms is 60000 milliseconds, which is equafilant to ten minutes
        final int MAX_TIME = 60000;

        //an array to store all of the information that I came to see on the
        //whatever, I am hungry.
        String[][] data = new String[8][6];

        //make fucking comparators for each potential
        //sort employees by name
        Comparator<Employee> employeeName, employeeID, employeeHired,
                employeeDept;

        //this lambda expression will be used to facilitate
        //by name comparisons of employees
        employeeName = Employee::compareNameTo;

        //this lambda expression will be used to compare by ID number
        employeeID = Employee::compareIDTo;

        //this lambda expression will be used to compare the hire dates of
        //employees
        employeeHired = Employee::compareHiredTo;

        //this lambda expression will be used to compare the departments of
        //the employees
        employeeDept = Employee::compareDeptTo;

        //the system can support from 1e3 to 1e9
        for (int i = 0; i < data[0].length; i++){
            //the number in question for N
            double N = Math.pow(10,i + 3);
            //save the number for N
            data[0][i] = Integer.toString((int) N);

            //create the fucker that will be sorted use a try block incase
```

```
//of memory errors
//start a timer
long start = System.currentTimeMillis();

Employee[] cockTwist = new Employee[(int) N];
try {
    //fill in the array with stupid fucking employees
    for (int j = 0; j < N; j++){
        cockTwist[j] = new Employee();
        if ((System.currentTimeMillis() - start) > MAX_TIME){
            data[1][i] = "Timed";
        }
    }
}
//save the memory error if you ran out of memory.
catch (OutOfMemoryError e){
    data[1][i] = "memed";
}

//if there was not a memory error finding space, then save the time
if (data[1][i] == null){
    data[1][i] = Integer.toString((int) (System.currentTimeMillis()
            - start));
}



if (data[1][i].equals("Timed")){
    data[2][i] = "N/A";
}
else{
    try{
        //fill in the array with stupid fucking employees
    for (int j = 0; j < N; j++){
        cockTwist[j] = new Employee();
        if ((System.currentTimeMillis() - start) > MAX_TIME){
            data[1][i] = "Timed";
        }
    }
        //store the time it took for the thing to work
        data[2][i] = Sort.simpleBubbleSort(cockTwist,
                employeeID, MAX_TIME);
    }
    //produce an out of memory error if there is not enough memory
    catch (OutOfMemoryError e){
        data[2][i] = "memed";
    }
}

if (data[1][i].equals("Timed")){
    data[3][i] = "N/A";
}
else{
    //create another temporary array to copy the piece of shit into
    try{
        //fill in the array with stupid fucking employees
    for (int j = 0; j < N; j++){
        cockTwist[j] = new Employee();
        if ((System.currentTimeMillis() - start) > MAX_TIME){
            data[1][i] = "Timed";
        }
```

```
    }
        data[3][i] = Sort.enhancedBubbleSort(cockTwist,
                employeeHired, MAX_TIME);
    }
    //catch a memory error
    catch (OutOfMemoryError e){
        data[3][i] = "memed";
    }
}

if (data[1][i].equals("Timed")){
    data[4][i] = "N/A";
}
else{
    //try the selection sort
    try{
        //fill in the array with stupid fucking employees
    for (int j = 0; j < N; j++){
        cockTwist[j] = new Employee();
        if ((System.currentTimeMillis() - start) > MAX_TIME){
            data[1][i] = "Timed";
        }
    }
        data[4][i] = Sort.selectionSort(cockTwist,
                employeeID, MAX_TIME);
    }
    //catch a memory error
    catch (OutOfMemoryError e){
        data[4][i] = "memed";
    }
}

if (data[1][i].equals("Timed")){
    data[5][i] = "N/A";
}
else{
    //try the merge sort
    try{
        //fill in the array with stupid fucking employees
    for (int j = 0; j < N; j++){
        cockTwist[j] = new Employee();
        if ((System.currentTimeMillis() - start) > MAX_TIME){
            data[1][i] = "Timed";
        }
    }
        data[5][i] = Sort.mergeSort(cockTwist, employeeID,
                System.currentTimeMillis(), MAX_TIME);
    }
    //catch a memory error
    catch (OutOfMemoryError e){
        data[5][i] = "memed";
    }
}

if (data[1][i].equals("Timed")){
    data[6][i] = "N/A";
}
else{
    //try the radix sort
    try{
        //fill in the array with stupid fucking employees
```

```
        for (int j = 0; j < N; j++){
            cockTwist[j] = new Employee();
            if ((System.currentTimeMillis() - start) > MAX_TIME){
                data[1][i] = "Timed";
            }
        }
            data[6][i] = Sort.radixSort(cockTwist,employeeDept,
                    employeeHired,employeeName,employeeID,MAX_TIME);
        }
        //catch a memory error
        catch (OutOfMemoryError e){
            data[6][i] = "memed";
        }
    }

    if (data[1][i].equals("Timed")){
        data[7][i] = "N/A";
    }
    else{
        //insertion sort
        try{
            //fill in the array with stupid fucking employees
        for (int j = 0; j < N; j++){
            cockTwist[j] = new Employee();
            if ((System.currentTimeMillis() - start) > MAX_TIME){
                data[1][i] = "Timed";
            }
        }
            data[7][i] = Sort.insertionSort(cockTwist,
                    employeeName,MAX_TIME);
        }
        //catch a memory error
        catch (OutOfMemoryError e){
            data[7][i] = "memed";
        }
    }
}

StringBuilder table = new StringBuilder();
//print out all the data

table.append("N/A means the table was not able to be made for");
table.append(" whatever reason \n");
table.append("'memed' is indicative of a memory error, such as it ");
table.append("takes too much memory\n");
table.append("'Timed' means the particular function timed out\n");
table.append("Creat\t\t");
for (String item : data[0]) {
    table.append(item);
    table.append("\t\t");
}
table.append("\n");

table.append("sBub\t\t");
for (String item : data[1]) {
    table.append(item);
    table.append("\t\t");
}
table.append("\n");

table.append("eBub\t\t");
```

```
        for (String item : data[2]) {
            table.append(item);
            table.append("\t\t");
        }
        table.append("\n");

        table.append("sBub\t");
        for (String item : data[3]) {
            table.append(item);
            table.append("\t\t");
        }
        table.append("\n");

        table.append("slct\t");
        for (String item : data[4]) {
            table.append(item);
            table.append("\t\t");
        }
        table.append("\n");

        table.append("merge\t");
        for (String item : data[5]) {
            table.append(item);
            table.append("\t\t");
        }
        table.append("\n");

        table.append("radix\t");
        for (String item : data[6]) {
            table.append(item);
            table.append("\t\t");
        }
        table.append("\n");

        table.append("insrt\t");
        for (String item : data[7]) {
            table.append(item);
            table.append("\t\t");
        }
        table.append("\n");

        System.out.println(table.toString());
    }
}
```

2 Sort.java

```java
//This piece of udder shit (aka cowshit) is for the books merge sort
//or something
import java.util.Arrays;

/**
 * This is the fucking place for the sorting algorithms bitch face fucker.
 * @author Steven Glasford
 */
public class Sort {
    //using the simple bubble sort algorithm sort that fucking list
    //vaginal cum pancakes
    //funny story about those, I once had a girl friend at South Dakota School
    //of Mines and Technology, I think her name was Sydney, anyways she was
    //really into kinky stuff, especially BDSM, but it was her "fancy" device
    //that she invented to collect her vagina fluid (she was wet as fuck)
    //then she would put the substance into different foods, luckily she
    //was not a stank, and the pancakes she made were actually pretty decent.

    /**
     * A fucking method to sort something using the new and enhanced bubble
     * sort method.
     * @param <K>           A generic placeholder, will probably be an
     *                      employee.
     * @param dildoEater    A generic array for the array you want to sort.
     * @param comp          The comparison lambda function.
     * @param maxTime       The maximum time you want the piece of shit to run.
     * @return              A string of how long it took for it to run.
     */
    public static <K> String enhancedBubbleSort(K[] dildoEater, Comparator<K>
            comp, int maxTime){
        //use this to start a "timer" of sorts
        long start = System.currentTimeMillis();
        for (int j = 0; j < dildoEater.length; j++){
            //the j in the comparison part of the for loop makes the flavor
            //enhanced, like those little knobs on fancy condoms
            for (int i = 0; i < dildoEater.length - 1 - j/*knobber*/; i++){
                if (comp.compare(dildoEater[i],dildoEater[i+1]) > 0){
                    K shitForLube;
                    shitForLube = dildoEater[i];
                    dildoEater[i] = dildoEater[i + 1];
                    dildoEater[i+1] = shitForLube;
                }
            }
            //print out a time out, if it times out
            if ((System.currentTimeMillis() - start) > maxTime){
                return "Timed";
            }
        }
        //return the elapsed time
        return Long.toString(System.currentTimeMillis() - start);
    }

    /**
     * A simpler bubble sort without the knobs on the condom.
     * @param <K>       A generic placeholder name.
     * @param scrum     A generic array that you want to be sorted, and the
     *                  action of cumming and shitting at the same time.
     * @param comp      The comparison lambda function, that you want to
     *                  to use the comparison for.
```

```
     * @param maxTime   The maximum amount of time that you wish to have this
     *                  bitch run
     * @return          A string of how long it took, or a message indictating
     *                  it timed out.
     */
    public static <K> String simpleBubbleSort(K[] scrum, Comparator<K> comp,
            int maxTime){
        long start;
        start = System.currentTimeMillis();
        for (K scrum1 : scrum) {
            for (int i = 0; i < scrum.length - 1; i++){
                if (comp.compare(scrum[i],scrum[i+1]) > 0){
                    //This variable will be used as a temporary variable of
                    //genericland, free and open trading exists between
                    //genericland and the rest of javaland, however due to a
                    //recent referendum by the generics, the peacefull
                    //existance and economic stability of both javaland
                    //and genericland are in jeopardy.
                    //also look up what schmegma is, its sort of funny,
                    K schmegma;
                    //temporarily use schmegma
                    schmegma = scrum[i];
                    //exchange the scrums (scrum is when you ejaculate and shit
                    //at the same time.
                    scrum[i] = scrum[i + 1];
                    //use the tempoary variable to finish the switch.
                    scrum[i+1] = schmegma;
                }
            }
            //print out a time out, if it times out
            if ((System.currentTimeMillis() - start) > maxTime){
                return "Timed";
            }
        }
        return Long.toString(System.currentTimeMillis() - start);
    }

    /**
     * An insertion sorting algorithm that knows how to fuck.
     * @param <K>       A generic placeholder name that is just a fucking
     *                  twat if you get to know them.
     * @param sybian    A generic array that you want to sort, and a type of
     *                  vaginal self-stimulation device that is commonplace
     *                  in BDSM.
     * @param comp      The comparison lambda expression that you want to sort.
     * @param maxTime   The maximum amount of time you want this thing to run.
     * @return          A string of how long it took, and the amount of pain
     *                  inflicted from the the Discipline, or a string
     *                  containing an error message.
     */
    ////////////////////////fuck me harder...daddy//////////////////////////
    public static <K> String insertionSort(K[] sybian, Comparator<K> comp,
            int maxTime){
        long start = System.currentTimeMillis();
        for (int i = 1; i< sybian.length; ++i){
            //make a temporary variable, BDSM should be more culturally
            //acceptable
            K urethralPlay = sybian[i];
            //A sybian is a sort of electric masturbation device for women
            //typically used in BDSM.
```

```
        //use a while loop, because they look better
        int j = i - 1;

        //compare the things
        while (j >= 0 && comp.compare(sybian[j], urethralPlay) < 0){
            sybian[j + 1] = sybian[j];
            j -= 1;
        }

        sybian[j+1] = urethralPlay;

        //use this to kill the shit if the time out is reached
        if ((System.currentTimeMillis()-start) > maxTime){
            return "Timed";
        }

    }
    //return the total time ellapsed in milliseconds
    return Long.toString(System.currentTimeMillis() - start);
}

/**
 * A selection sorting algorithm, pretty fucking self-explain.
 * @param <K>       A generic place holder name.
 * @param frogtie   A position in BDSM, where the femme ties their legs and
 *                  arms into a position that resembles a tied frog.
 * @param comp      The comparison lambda fucntion.
 * @param maxTime   The maximum amount of time that you want the piece of
 *                  shit to  run
 * @return          The amount of time it took to run, or an error message.
 */
public static <K> String selectionSort(K[] frogtie, Comparator<K> comp,
        int maxTime){
    //start a timer
    long start = System.currentTimeMillis();

    for (int i = 0; i < frogtie.length -1; i++){
        //this will be the smallest found piece of shit
        int midgetsFucking = i;
        for (int j = i+1; j < frogtie.length; j++){
            if (comp.compare(frogtie[j],frogtie[midgetsFucking]) < 0){
                midgetsFucking = j;
            }
        }

        //check the amount of time that has passed
        if ((System.currentTimeMillis() - start) > maxTime){
            return "Timed";
        }

        //swap the found minimum element, breastBondage is a temporary
        //variable
        K breastBondage = frogtie[midgetsFucking];
        frogtie[midgetsFucking] = frogtie[i];
        frogtie[i] = breastBondage;
    }

    //return the amount of time that has passed
    return Long.toString(System.currentTimeMillis() - start);
}
```

```java
//Merge contents of arrays S1 and S2 into properly sized array S.
//The lack of vulgarity means it was copied from the book
/**
 * Merges two different arrays, I got this from the book.
 * @param <K>   A generic placement name.
 * @param S1    The first array you want to merge.
 * @param S2    The second array you want to merge.
 * @param S     The final array that you want to be merged into.
 * @param comp  The comparison lambda expression.
 */
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)){
            //copy the ith element of S1 and increment i.
            S[i+j] = S1[i++];
        }
        else{
            //copy the jth element of S2 and increment j.
            S[i+j] = S2[j++];
        }
    }
}

//merge-sort contents of array S.
/**
 * The actual merge sorting algorithm, this is stable.
 * @param <K>       A generic type thingy.
 * @param S         The array you want to be sorted
 * @param comp      The Comparison lambda expression.
 * @param startTime The start of the algorithm.
 * @param maxTime   The maximum amount of time you want the thing to run.
 * @return          A string containing the time it took to run, or an
 *                  error message.
 */
public static <K> String mergeSort(K[] S, Comparator<K> comp, long startTime,
        int maxTime){
    int n = S.length;
    //array is tivially sorted
    if (n < 2) {
        return Long.toString(System.currentTimeMillis() - startTime);
    }

    if ((System.currentTimeMillis() - startTime) > maxTime){
        return "Timed";
    }

    //divide (What a weirdly vague statement from the fucking book)
    int mid = n/2;
    //copy of first half
    K[] S1 = Arrays.copyOfRange(S, 0, mid);
    //copy of second half
    K[] S2 = Arrays.copyOfRange(S,mid,n);
    //conquer (with recursion)
    //sort copy of first half
    if (mergeSort(S1, comp, startTime, maxTime).equals("Timed")){
        return "Timed";
    }
    //sort copy of second half
    if (mergeSort(S2, comp, startTime, maxTime).equals("Timed")){
```

```
            return "Timed";
        }

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

        //return the total amount of time that has passed
        return Long.toString(System.currentTimeMillis() - startTime);
    }

    //Quick-sort contents of a queue.
    /**
     * A quick sort algorithm, that uses queues
     * @param <K>       A generic placement name.
     * @param S         A generic queue of the shit you want to sort.
     * @param comp      The comparison lambda expression.
     * @param startTime The start of the function as it is recursive.
     * @param maxTime   The maximum time you want to run the shit.
     * @return          The amount of time it took to run, or an error mess.
     */
    public static <K> String quickSort(Queue<K> S, Comparator<K> comp,
            long startTime, int maxTime){
        int n = S.size();
        //queue is trivially sorted
        if (n < 2){
            return Long.toString(System.currentTimeMillis() - startTime);
        }

        //check to see if anything has exceeded its stay
        if ((System.currentTimeMillis() - startTime) > maxTime){
            return "Timed";
        }

        //divide
        //using first as arbitrary pivot
        K pivot = S.first();
        Queue<K> P = new LinkedQueue<>();
        Queue<K> E = new LinkedQueue<>();
        Queue<K> G = new LinkedQueue<>();

        //divide original into P, E, and G
        while (!S.isEmpty()){
            K element = S.dequeue();
            int c = comp.compare(element, pivot);
            //element is less than pivot
            if (c < 0){
                P.enqueue(element);
            }
            //element is equal to pivot
            else if (c == 0){
                E.enqueue(element);
            }
            //element is greater than pivot
            else{
                G.enqueue(element);
            }
        }

        //conquer
        //sort elements less than pivot
```

```
        if (quickSort(P, comp, startTime, maxTime).equals("Timed")){
            return "Timed";
        }
        //sort elements greater than pivot
        if (quickSort(G, comp, startTime, maxTime).equals("Timed")){
            return "Timed";
        }

        //concatenate results
        while(!P.isEmpty()){
            S.enqueue(P.dequeue());
        }
        while(!E.isEmpty()){
            S.enqueue(E.dequeue());
        }
        while(!G.isEmpty()){
            S.enqueue(G.dequeue());
        }

        //return how much time has elapsed
        return Long.toString(System.currentTimeMillis() - startTime);
    }

    /**
     * A sort of radix sort that uses subroutines of of the merge sort
     * @param <K>       A generic type name.
     * @param bukkake   A bunch of men in a circle jacking off on a femme,
     *                  part of the Sadism and Masochism parts of BDSM.
     *                  Also the array you want to sort.
     * @param cock1     The first comparator (Also the least important).
     * @param cock2     The second comparator.
     * @param cock3     The third comparator.
     * @param cock4     The fourth comparator.
     * @param maxTime   The maximum amount of time you want to run the shit.
     * @return          A string containing the time it took to run, or
     *                  an error message.
     */
    public static <K> String radixSort(K[] bukkake, Comparator<K> cock1,
            Comparator<K> cock2, Comparator<K> cock3, Comparator<K> cock4,
            int maxTime){
        //get the max value from the first cock (which is the least significant
        //variable, so its the smallest cock
        //first check to see if the first cock is in place

        //start the watch
        long start = System.currentTimeMillis();
        if (cock1 != null){
            //check to see if the piece of shit timed out
            if (mergeSort(bukkake, cock1, start,maxTime).equals("Timed")){
                return "Timed";
            }
        }

        if (cock2 != null){
            //check to see if the piece of shit has timed out
            if (mergeSort(bukkake, cock2, start, maxTime).equals("Timed")){
                return "Timed";
            }
        }

        if (cock3 != null){
```

```
            //check to see if the piece of shit has timed out
            if (mergeSort(bukkake, cock3, start, maxTime).equals("Timed")){
                return "Timed";
            }
        }

        if (cock4 != null){
            //check to see if the bitch has timed out, if it has then kill it
            if (mergeSort(bukkake,cock4, start, maxTime).equals("Timed")){
                return "Timed";
            }
        }
        //suckatoof

        //return the elaspsed time
        return Long.toString(System.currentTimeMillis() - start);
    }


}
```

3 Employee.java

```
/**
 * An class for storing information about stupid employees, I guess I should
 * not call them stupid, the                        is a very important part of the
 * society, sorry         .
 *
 * @author Steven Glasford
 * @version 15 April 2019
 */
public class Employee implements Comparable<Employee>{
    //this will be a random number, which is super weird, like does that mean
    //all employees are random, and that corporations hate their employees
    //my god, NDSU hates Unions,             .
    //Anyways, facts aside, this id number is random, and between 0
    //and 99999999.
    //We also ignore the fact that we may get duplicate id numbers
    private final int id;
    //Oh Shit, these people we are making don't even have real names, they are
    //fucking slaves, they only got random letters for names, this is getting
    // more and more fucked up, it be like seeing your mate on the street and
    //instead of saying "Ahoy Oliver, what a marvelous bird carcass on the
    //ground", it be like "Howdy xcjsmkmw, it is a marvelous high tide," that
    //is fucking crazy
    //I might change this in the future to at least be a random English word,
    // like shoeFuck, or something, at least that is a name and not hvnerfew
    private final StringBuilder name = new StringBuilder();
    //o my fucking god, these people are nothing to this corporation,
    //its fucking worse than the chattle loan industry.
    //This variable is also random, and between 1 to 5
    private int dept = 0;
    //this is the date when the person was hired, at least this one is sort of
    //realistic
    private int hired = 0;

    //public utilites, like gas, electricty, and power, and garbage,
    //maybe if you live somewhere nice you get sewer and water, OOOOOhhhh.
    //initialize this fucking class, with a bunch of random shit, because
    //we at NDSU hate Unions.
    /**
     * A main instatiator class
     */
    Employee (){
        //fuck the police bitchhole
        id = (int) (Math.random() * 99999999);
        //decide how big you want that fucking name to be inside (sex)
        //add five to the number to keep it between 5 and 10
        int sizeOfTheDick = (int) (Math.random() * 5) + 5;
        for (int i = 0; i < sizeOfTheDick; i++){
            //the name will only have lowercase letters for ease of use
            name.append((char) ((int) (Math.random() * (122 - 97)) + 97));
        }
        //put these motherfuckers into a random department
        dept = (int) (Math.random() * (5-1)) + 1;
        //give them a random year they started to work
        hired = (int) (Math.random() * 10) + 2008;
    }

    /**
     * Get all of this bitching data out of the fucking class, like a fart
     * in the elevator.
```

```
     * @return  The Employees damn id number
     */
    public int getId(){
        return id;
    }

    /**
     * Get the fucking name out of here.
     * @return The cunt blasters name
     */
    public String getName(){
        return name.toString();
    }

    /**
     * Get the fucking department out of the fuck here.
     * @return The department number.
     */
    public int getDept(){
        return dept;
    }

    /**
     * One Flew over the Cookoos nest
     * @return The year the motherfucker was hired
     */
    public int getHired(){
        return hired;
    }

    /**
     * Change the year the bitch was hired.
     * @param hired the hired to set
     */
    public void setHired(int hired) {
        this.hired = hired;
    }

    /**
     * Compare two fucking employees based on their ID number
     * @param anotherFuckingEmployee A different employee you are comparing to.
     * @return  How much different the employee is to the first one.
     */
    public int compareIDTo(Employee anotherFuckingEmployee){
        return this.getId() - anotherFuckingEmployee.getId();
    }

    /**
     * This is to compare two different employees to see which is better.
     * @param paperPusher   A regular Joe, a stupid ass Employee
     * @return              An int depending on how far away the dept are.
     */
    public int compareDeptTo(Employee paperPusher){
        return this.getDept() - paperPusher.getDept();
    }

    /**
     * Compare two employees based on the year they were hired
     * @param cockTease A generic employee from genericland
     * @return          A positive or negative number, or zero if the
     *                  the two are equal.
```

```
 */
public int compareHiredTo(Employee cockTease){
    return this.getHired() - cockTease.getHired();
}

/**
 * Compare the names of two people alphabetically, if the output is
 * negative then the first variable is greater, and elsewise.
 * @param managementSucks   The variable you are comparing to bitch
 * @return  A positive, negative or a zero number, zero means the two are
 *          the same, a positive number is greater than, and a negative
 *          is a less than.
 */
public int compareNameTo(Employee managementSucks){
    //temp variables
    int i = 0, nipple = 0;

    //find the string with the least letters
    if (this.getName().length() < managementSucks.getName().length()){
        //save the smallest value into the nipple
        nipple = this.getName().length();
    }
    else if (this.getName().length() > managementSucks.getName().length()){
        //save the lower bound (which happens to be the managementSucks
        //variable
        nipple = managementSucks.getName().length();
    }
    else {
        //the fucking strings are the same length
        nipple = this.getName().length();
    }

    //this is a temporary variable and is used to measure the difference
    //between each letter in the string, a negative number corresponds to a
    //string that comes before the comparator, and vice verse
    int difference = 0;
    while ((i < nipple) && (difference == 0)){
        difference = this.getName().charAt(i) -
                managementSucks.getName().charAt(i);

        //increase the doolly bop, which is a another word for the tensions
        //of Israel and Gaza.
        i++;
    }

    //this should be used to determine which value is bigger,
    //without the use of a ruler.
    return difference;
}

//this function will be a place holder, as we will use lambda expressions
//for the compare function
/**
 * A standard compareTo function without very much purpose, so it should
 * be a great tool, it compares the id
 * @param anotherGreatEmployee  Another fucking employee.
 * @return  An int of how far away one employee is to another.
 */
@Override
public int compareTo(Employee anotherGreatEmployee){
    return this.getId() - anotherGreatEmployee.getId();
```

```
    }
}
```

4 LinkedQueue.java

```java
/**
 * Realization of a FIFO queue as an implementation of a SinglyLinkedSet.
 *
 * @author Michael T. Goodrich
 * @author Roberto Tamassia
 * @author Michael H. Goldwater
 * @author Steven Glasford
 * @version 2-21-2019
 * @param <E>
 */


public class LinkedQueue<E> implements Queue<E> {
    //an empty list
    private final SinglyLinkedList<E> list = new SinglyLinkedList<>();
    //new queue relies on the initially empty list
    public LinkedQueue() {}

    @Override
    public int size() {return list.size();}

    @Override
    public boolean isEmpty() {return list.isEmpty();}

    @Override
    public void enqueue(E element) {list.addLast(element);}

    @Override
    public E first() {return list.first();}

    @Override
    public E dequeue() {return list.removeFirst();}

}
```

5 SinglyLinkedList.java

```java
/**
 *
 * SinglyLinkedList Class
 * Code Fragments 3.14, 3.15
 * from
 * Data Structures & Algorithms, 6th edition
 * by Michael T. Goodrich, Roberto Tamassia & Michael H. Goldwasser
 * Wiley 2014
 * Transcribed by
 * @author Steven Glasford
 * @version January 31, 2019
 * @param <E> a generic placeholder name
 */
public class SinglyLinkedList<E> {
    /**
     *
     * @param <E> a generic placeholder name
     *
     * A subclass creating the Node
     */
    private static class Node<E>{
        //reference to the element stored at this node
        private final E element;
        //reference to the subsequent node in the list
        private Node<E> next;
        public Node(E e, Node<E> n){
            element = e;
            next = n;
        }

        /**
         *
         * @return Return the current element
         */
        public E getElement(){return element;}

        /**
         *
         * @return return the address of the next item in the linked list
         */
        public Node<E> getNext() {return next;}

        /**
         *
         * @param n the next item in the list
         */
        public void setNext(Node<E> n) {next = n;}
    }

    //head node of the list (or null if empty)
    private Node<E> head = null;
    //last node of the list (or null if empty)
    private Node<E> tail = null;
    //number of nodes in the list
    private int count = 0;

    /**
     * constructs an initially empty list
     */
```

```
public SinglyLinkedList(){}

//access methods
/**
 *
 * @return Return the size of the linked list
 */
public int size() {return count;}

/**
 *
 * @return Determine if the linked list is empty
 */
public boolean isEmpty() {return count == 0;}

/**
 *
 * @return return the first element in the list
 *
 * returns (but does not remove) the first element
 */
public E first(){
    if (isEmpty()) return null;
    return head.getElement();
}

/**
 *
 * @return the last element in the linked list
 *
 * returns (but does not remove the last element
 */
public E last(){
    if (isEmpty()) return null;
    return tail.getElement();
}

//update methods

/**
 *
 * @param e A generic element
 *
 * adds element e to the front of the list
 */
public void addFirst(E e){
    //create and link a new node
    head = new Node<>(e, head);
    //special case: new node becomes tail also
    if (count == 0)
        tail = head;
    count++;
}

/**
 *
 * @param e A generic item
 *
 * adds element e to the end of the list
 */
public void addLast(E e) {
```

```
        //node will eventually be the tail
        Node<E> newest = new Node<>(e,null);
        //special case: previously empty list
        if (isEmpty())
            head = newest;
        else
            tail.setNext(newest);
        tail = newest;
        count++;
    }

    /**
     *
     * @return return the item that was removed
     *
     * removes and returns the first element
     */
    public E removeFirst(){
        //nothing to remove
        if (isEmpty()) return null;
        E answer = head.getElement();
        //will become null if list had only one node
        head = head.getNext();
        count--;
        //special case as list is now empty
        if(count == 0)
            tail = null;
        return answer;
    }
}
```

6 Queue.java

```java
/**
 * @author Michael T. Goodrich
 * @author Roberto Tamassia
 * @author Michael H. Goldwater
 * @author Steven Glasford
 * @version 2-21-2019
 * @param <E>
 */

public interface Queue<E> {
    /**
     * Returns the number of elements in the queue
     * @return
     */
    int size();

    /**
     * Tests whether the queue is empty
     * @return
     */
    boolean isEmpty();

    /**
     * Inserts an element at the rear of the queue
     * @param e
     * @todo    modify so that this is required to throw a queue Full Exception
     *          if called on a full queue
     */
    void enqueue(E e);

    /**
     * returns, but does not remove, the first element of the queue
     * (null if empty).
     * @return
     */
    E first();

    /**
     * Removes and returns the first element of the queue (null if empty)
     * @return
     */
    E dequeue();
}
```

7 output.txt

```
/*********this file was edited to ensure proper latex presentation******/

run:
N/A means the table was not able to be made for whatever reason
'memed' is indicative of a memory error, such as it takes too much memory
'Timed' means the particular function timed out
Creat   1000    10000   100000  1000000 10000000        100000000
sBub    3       4       Timed   Timed   Timed           Timed
eBub    34      512     Timed   Timed   Timed           memed
sBub    26      371     Timed   Timed   Timed           N/A
slct    13      221     N/A     N/A     N/A             N/A
merge   4       23      N/A     N/A     N/A             N/A
radix   14      56      N/A     N/A     N/A             N/A
insrt   100     2531    N/A     N/A     N/A             N/A

BUILD SUCCESSFUL (total time: 9 minutes 13 seconds)
```