**GIT Department of Computer Engineering**

**CSE 222/505 - Spring 2021**

**Homework #3 Report**

**Sena Erdoğan**

**1901042680**

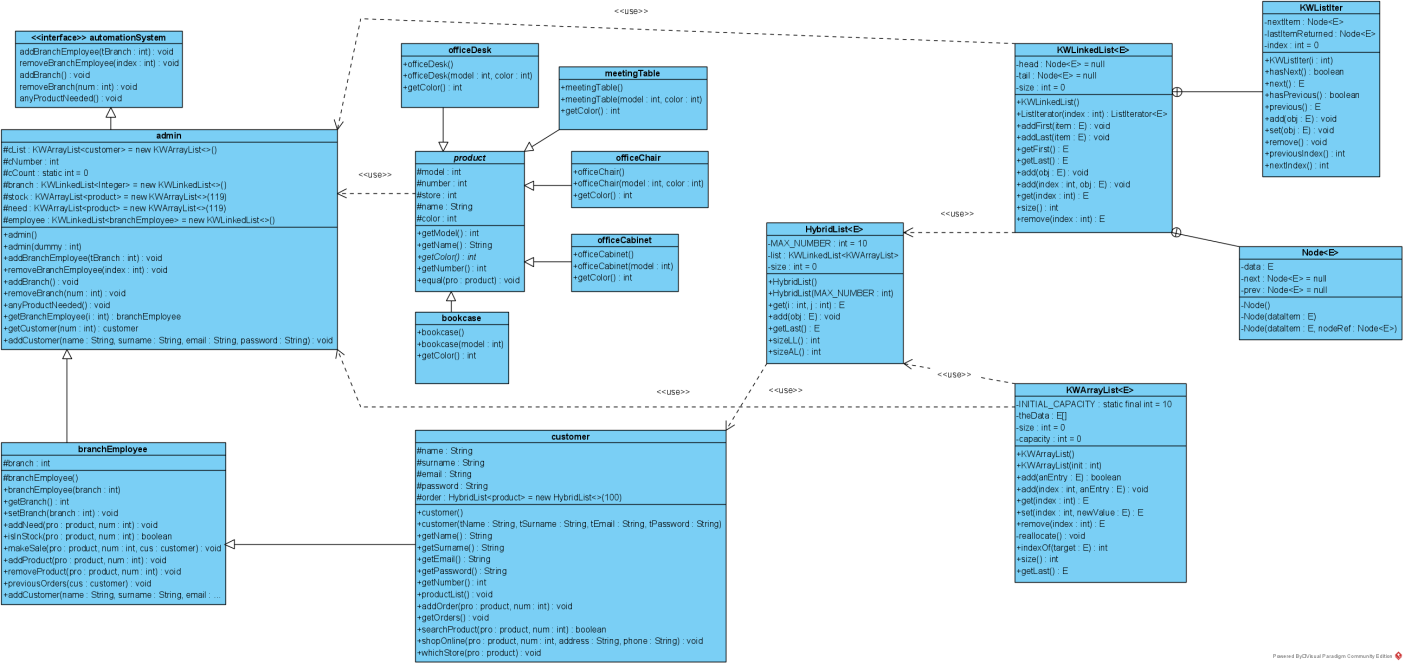
1. **SYSTEM REQUIREMENTS**

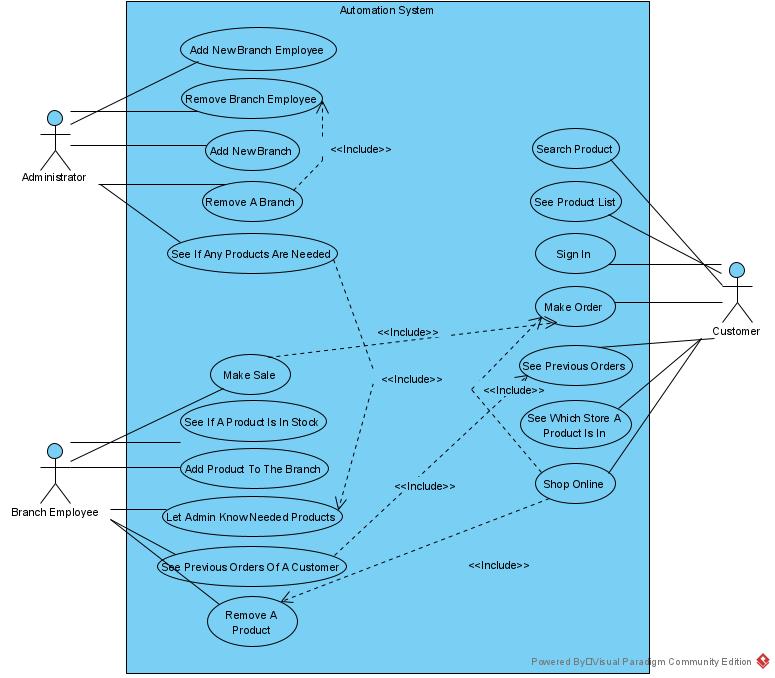
Automation system is an interface that declares some of the methods that is used. Admin implements automation system’s methods. Admin class can add branch employees, remove branch employees, add branches, remove branches, see if there are any products needed. Branch employee class extends admin class and can add needs, see if a product is in stock, make sales, add products to branches, remove products from branches and access a customer’s previous orders. Customer class extends branch employee class and can see the which products are in stock, make an order, see their own previous orders, see if a product is in stock, shop online and see which stores have the product they want.

Customers are logged into the system by entering their names, surnames, emails and passwords. These information should be valid and it is checked during this process. Branch employees manipulate customers’ information and operations. Admin can manipulate customers’ and branch employees’ information and operations. Abstract product class is used in place of every furniture in methods.

There is a different class hierarchy for the products. There are 5 categories of products and they all extend the abstract class “product”. Product field is abstract as the user cannot make an object of it but all the methods using an uncertain product have product as parameter. All furnitures have integers as fields representing their models and colors. Product class implements all the functions these 5 class needs (getters for furniture name, model and number) except for getColor() method. Bookcases and office cabinets don’t have a color field therefore it will cause an error if color of these furnitures tried to be used. So instead, these classes returns -1 as an indicator of not having a color field and this property is used frequently in automation system functions.

1. **USE CASE AND CLASS DIAGRAMS**





1. **PROBLEM SOLUTION APPROACH**

There are three main users of the automation system, therefore three classes. Automation system should be the interface they all will use. In an automation system, administrator should have access to everything that is happening in the system, to customers information and branch employee information. Likewise, branch employees should have access to customer information. That states the hierarchy should go from automation system to customer. “Customer” should be the subclass of “branch employee” and “branch employee” should be the subclass of “admin” while “admin” implements the interface “automation system”.

Most of the fields belong to the admin so while branch employees and customers are using them, admin should be able to manipulate all the information. All fields are protected because of this reason. Every class uses its own methods for its different tasks, but because of the hierarchy, they use each other’s methods while performing the tasks.

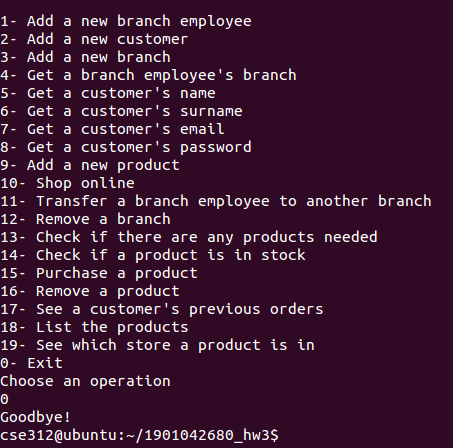
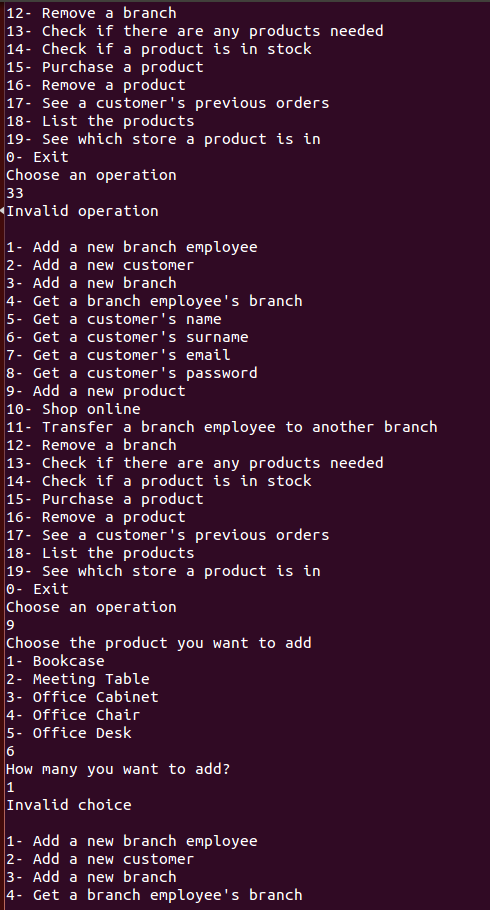
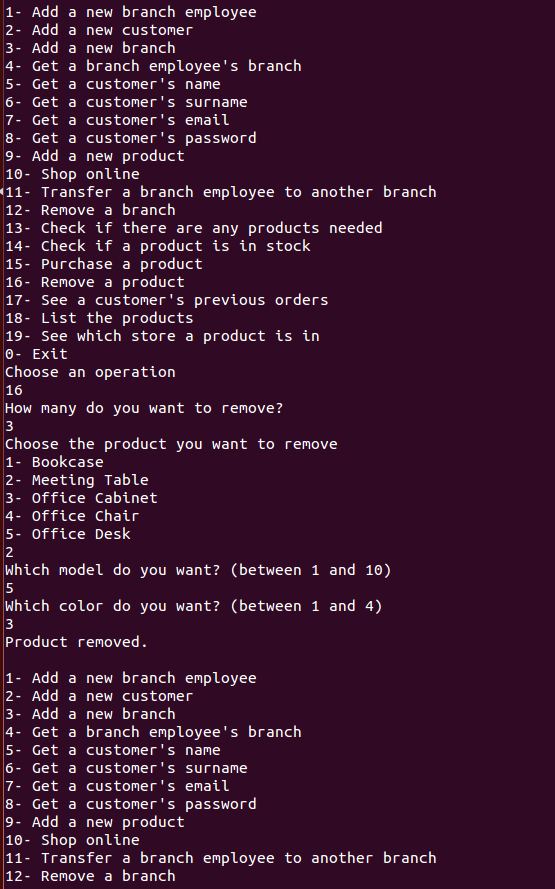
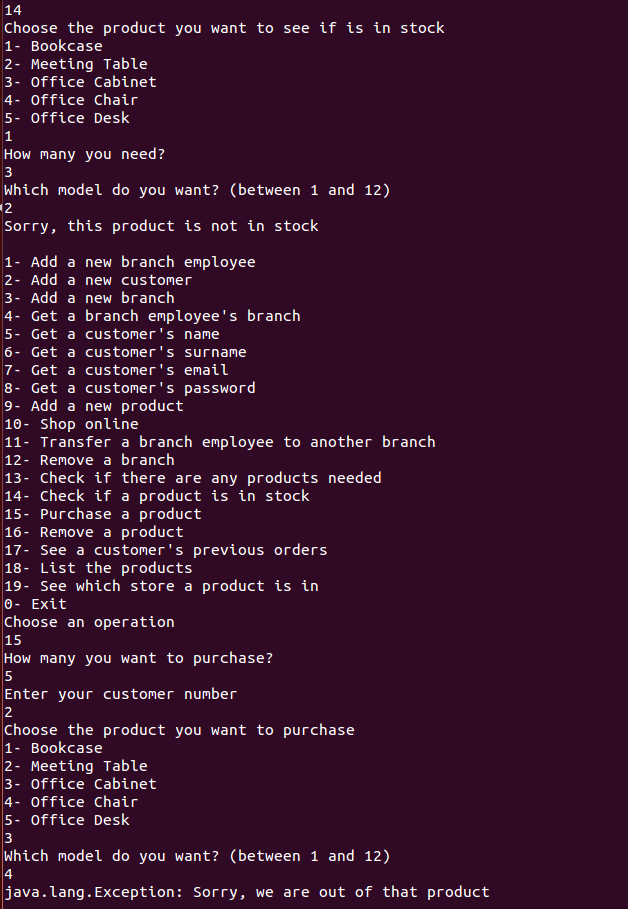
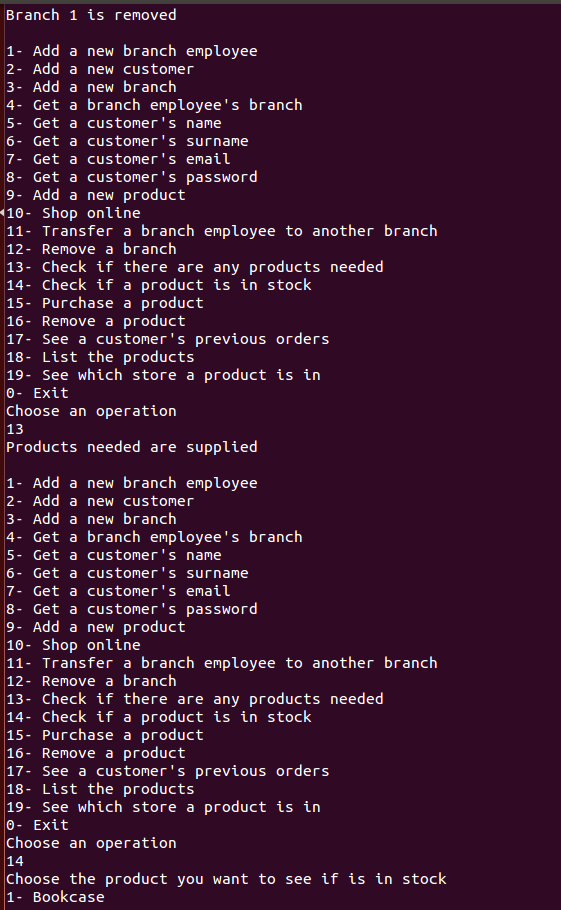
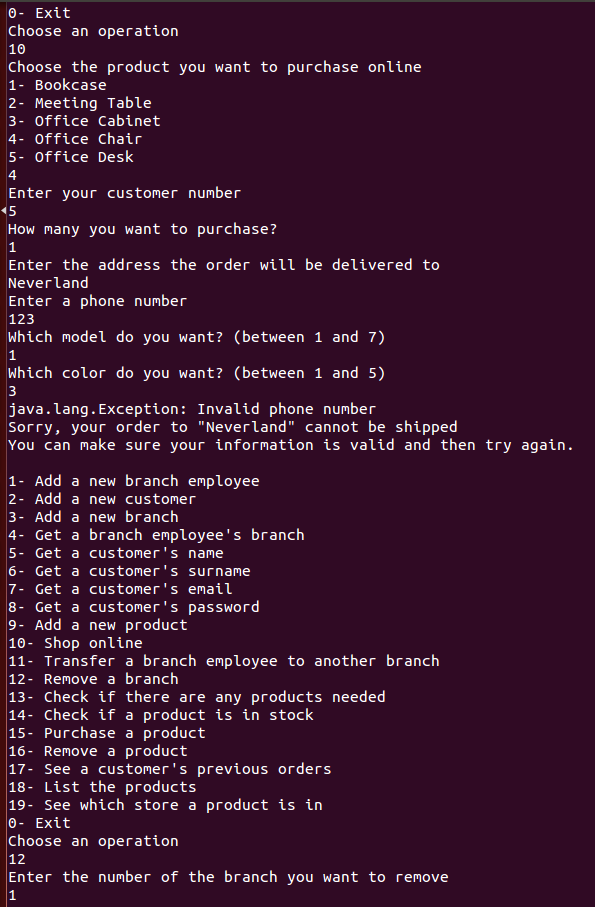
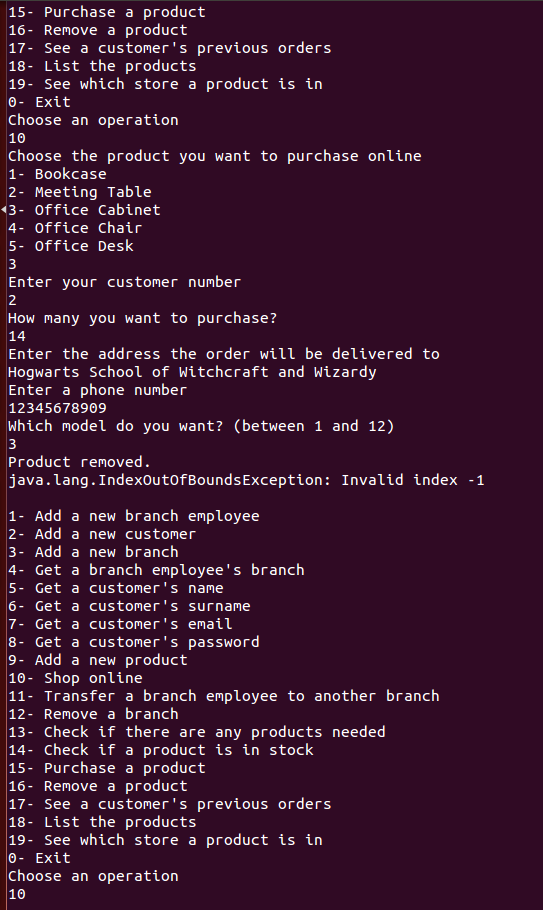
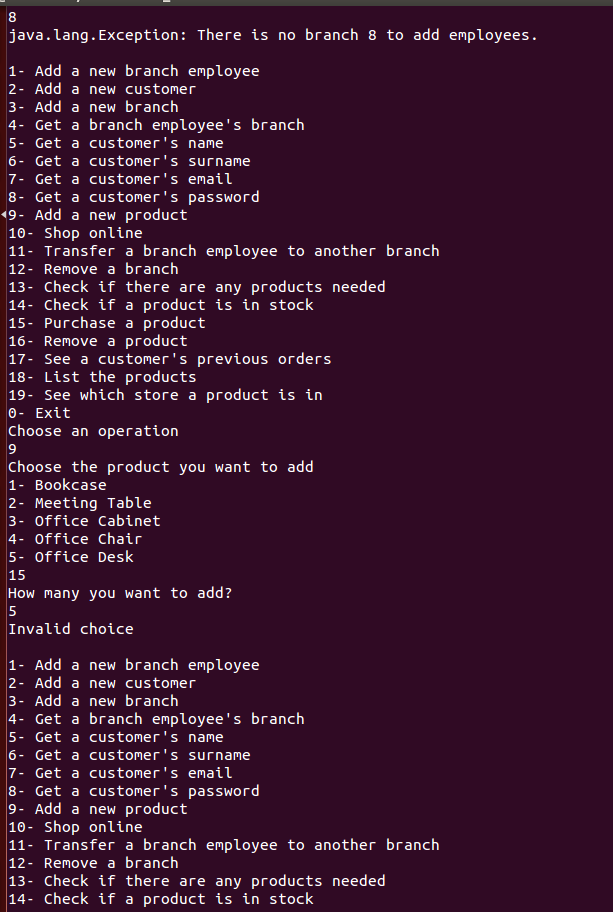
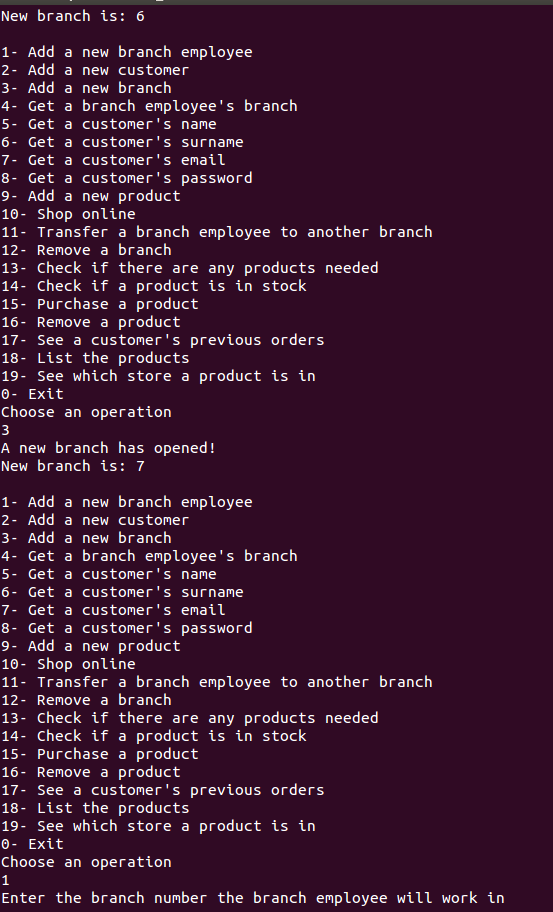
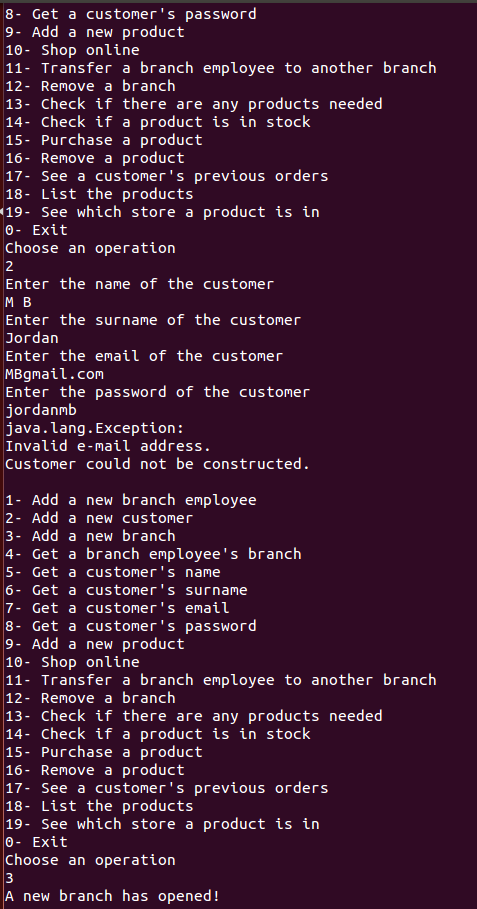
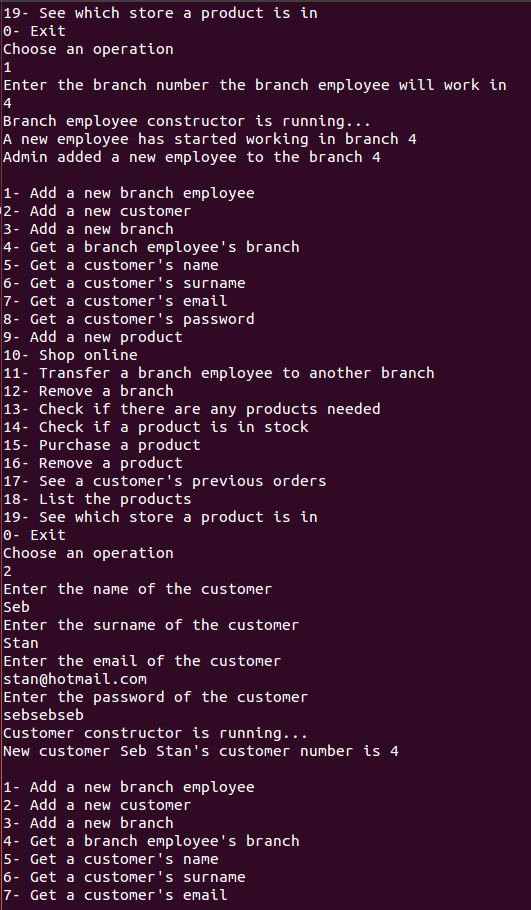
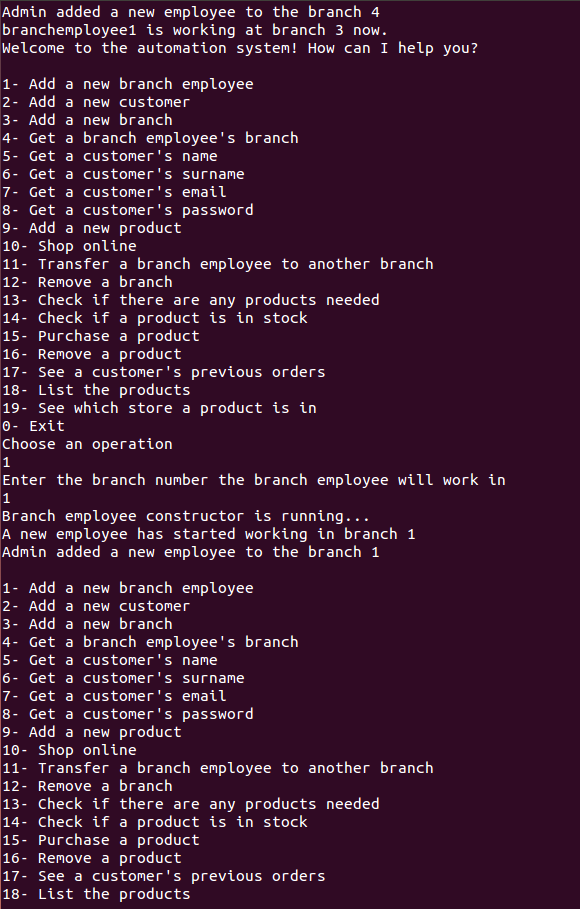
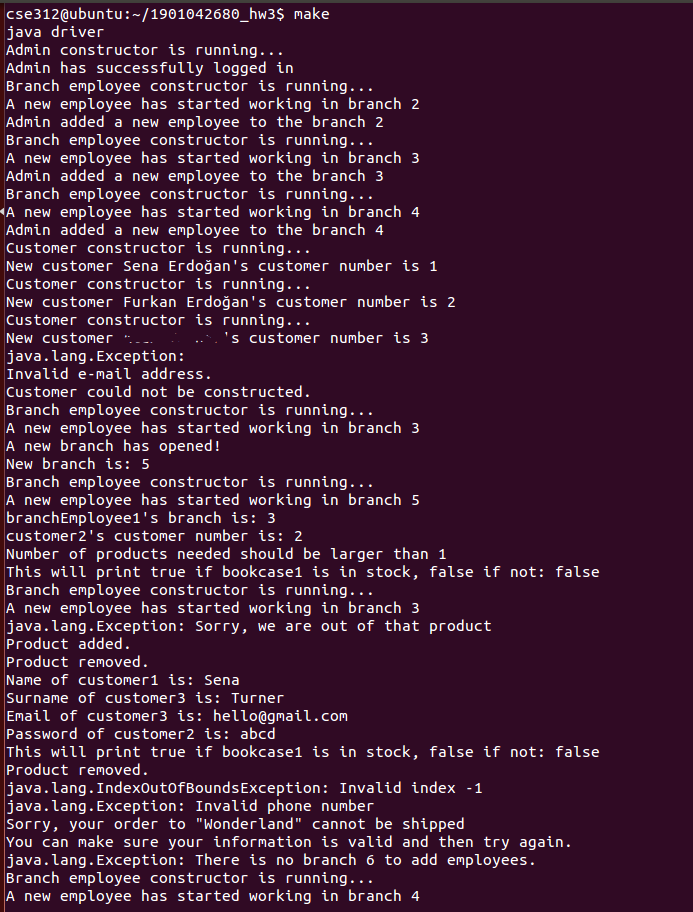
Products have model, color, number (how many of that product), store (which branch that product is in) fields. Name of a product just indicates its kind. Name field of an office cabinet is simply “Office Cabinet”. Stock and needs are stored in product KWArrayList. Customers are stored in a customer KWArrayList. Branches and branch employees are stored in an integer KWLinkedList. Customer orders are stored in HybridList of products.

1. **TEST CASES**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **TEST CASE ID** | **TEST SCENARIO** | **TEST STEPS** | **TEST DATA** | **EXPECTED RESULTS** | **ACTUAL RESULTS** | **PASS/FAIL** |
| TU01 | Add a new branch employee | Call branch employee constructor through the admin | Branch=2 | A new branch employee for branch 2 should be generated | As expected | Pass |
| TU02 | Check customer login with valid data | Call customer constructor | User’s name = Sena  User’s surname = Erdoğan  User’s e-mail = [s@gmail.com](mailto:s@gmail.com)  User’s password = 1234 | New customer should be generated and user should be informed of the new customer’s customer number | As expected | Pass |
| TU03 | Check customer login with invalid data | Call customer constructor | User’s name = myname  User’s surname = mysurname  User’s e-mail = NotAnEmail  User’s password = something | An e-mail should be in different forms of …@...com  This email doesn’t satisfy that, new customer shouldn’t be generated and an error message should be printed on the screen | As expected | Pass |
| TU04 | Add a new branch | Admin adds a new branch | There is no data needed.  The new branch’s number should be one more than the last branch. | There are 4 branches in total.  The new branch’s number should be 5. | As expected | Pass |
| TU05 | Add a new need with valid data | Branch employee adds a need | Product = Office Chair  Number = 13 | Product should be added to need array and user should be informed. | As expected | Pass |
| TU06 | Add a new need with invalid data | Branch employee adds a need | Product = Office Chair  Number = -2 | Product number cannot be less than 1.  Product cannot be added and the user should be informed. | As expected | Pass |
| TU07 | See if a product is in stock when it is not | Branch employee checks if the product is in stock | Product = Bookcase  Number = 2 | The user should be informed that the product is not in the stock. | As expected | Pass |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| TU08 | Try making sales with products that are not in the stock | Branch employee checks if the product is in stock | Product = Bookcase  Number = 4  Customer = customer2 | The user should be informed that the product is not in the stock. | As expected | Pass |
| TU09 | Add a product to the branch | Branch employee adds a product | Product = Office Chair  Number = 4 | The user should be informed that the product is added to the branch. | As expected | Pass |
| TU10 | Remove a product | Branch employee removes a product from the branch | Product = Office Cabinet  Number = 30 | The user should be informed that the product is removed from the branch. | As expected | Pass |
| TU11 | Shop online with valid data | Customer shops online | Product =  Office Cabinet  Number = 5  Address = A Galaxy Far Far Away  Phone = 12345678901 | The user should be informed that their order is successfully taken. | As expected | Pass |
| TU12 | Shop online with invalid data | Customer shops online | Product =  Office Cabinet  Number = 10  Address = Wonderland  Phone = 1234 | Phone number should consist of 11 digits. The user should be informed that and their order cannot be taken. | As expected | Pass |
| TU13 | Add a branch employee to a branch that does not exist | Admin adds a branch employee | Branch = 6 | The user should be informed that such branch does not exist | As expected | Pass |
| TU14 | Add a branch employee to a branch that does exist | Admin adds a branch employee | Branch = 4 | The user should be informed that the branch employee was successfully generated to work in specified branch | As expected | Pass |
| TU15 | Search a product that is not in the stock | Customer searches for the product | Product = Bookcase  Number = 2 | The user should be informed that the specified product is not in the stock | As expected | Pass |

1. **RUNNING AND RESULTS**



public void addBranchEmployee(int tBranch){ **ϴ(1)**

try{

if(tBranch>branch.size()) throw new Exception("There is no branch " + tBranch + " to add employees."); **ϴ(1)**

branchEmployee employee1 = new branchEmployee(tBranch); **ϴ(1)**

System.out.println("Admin added a new employee to the branch " + tBranch); **ϴ(1)**

}catch(Exception e){ System.out.println(e);} **ϴ(1)**

public admin(int dummy){ **ϴ(n2)**

System.out.println("Admin constructor is running..."); **ϴ(1)**

branch.add(1); **ϴ(1)**

branch.add(2); **ϴ(1)**

branch.add(3); **ϴ(1)**

branch.add(4); **ϴ(1)**

int k=0; **ϴ(1)**

for(int i=1; i<8; i++){ **ϴ(n)**

for(int j=1; j<6; j++){ **ϴ(n)**

officeChair officeChair1 = new officeChair(i,j); **ϴ(1)**

officeChair1.number = 50; **ϴ(1)**

stock.add(officeChair1); **ϴ(1)**

officeChair1.number = 0; **ϴ(1)**

need.add(officeChair1); **ϴ(1)**

k++; **ϴ(1)**

}

}

for(int i=1; i<6; i++){ **ϴ(n)**

for(int j=1; j<5; j++){ **ϴ(n)**

officeDesk officeDesk1 = new officeDesk(i,j); **ϴ(1)**

officeDesk1.number = 50; **ϴ(1)**

stock.add(officeDesk1); **ϴ(1)**

officeDesk1.number = 0; **ϴ(1)**

need.add(officeDesk1); **ϴ(1)**

k++; **ϴ(1)**

}

}

for(int i=1; i<11; i++){ **ϴ(n)**

for(int j=1; j<5; j++){ **ϴ(n)**

meetingTable meetingTable1 = new meetingTable(i,j); **ϴ(1)**

meetingTable1.number = 50; **ϴ(1)**

stock.add(meetingTable1); **ϴ(1)**

meetingTable1.number = 0; **ϴ(1)**

need.add(meetingTable1); **ϴ(1)**

k++; **ϴ(1)**

}

}

for(int i=1; i<13; i++){ **ϴ(n)**

bookcase bookcase1 = new bookcase(i); **ϴ(1)**

bookcase1.number = 50; **ϴ(1)**

stock.add(bookcase1); **ϴ(1)**

bookcase1.number = 0; **ϴ(1)**

need.add(bookcase1); **ϴ(1)**

if(i==2) stock.get(k).number = 0; // for test case **ϴ(1)**

}

public void removeBranchEmployee(int index){ **ϴ(n)**

try{

if(index>employee.size()) throw new Exception("There isn't a branch employee of the specified index."); **ϴ(1)**

employee.remove(index); **ϴ(n)**

System.out.println("Branch employee is successfully removed"); **ϴ(1)**

}catch(Exception e){ System.out.println(e);} **ϴ(1)**

}

public void addBranch(){**ϴ(1)**

branch.add(branch.size());**ϴ(1)**

System.out.println("A new branch has opened!"); **ϴ(1)**

System.out.println("New branch is: " + branch.size());**ϴ(1)**

}

public void removeBranch(int num){ **ϴ(n3)**

branch.remove(num-1); **ϴ(n)**

for(int i=0; i<employee.size(); i++){ **ϴ(n)**

if(employee.get(i).branch == num){ **ϴ(n)**

removeBranchEmployee(i); **ϴ(n)**

--i; **ϴ(1)**

}

}

}

public void anyProductNeeded(){**ϴ(n)**

for(int i=0; i<need.size(); i++){ **ϴ(n)**

if(need.get(i).number>0){ **ϴ(1)**

stock.get(i).number += need.get(i).number + 30; **ϴ(1)**

need.get(i).number = 0; **ϴ(1)**

System.out.println("Products needed are supplied"); **ϴ(1)**

}

}

}

public bookcase(int model){ **ϴ(1)**

try{

if(model>12 && model<1) throw new Exception("A bookcase of the specified model does not exist."); **ϴ(1)**

this.model = model; **ϴ(1)**

this.name = "Bookcase"; **ϴ(1)**

}

catch(Exception e){ System.out.println(e);} **ϴ(1)**

}

public branchEmployee(int branch){ **ϴ(1)**

System.out.println("Branch employee constructor is running..."); **ϴ(1)**

this.branch = branch; **ϴ(1)**

employee.add(this); **ϴ(1)**

System.out.println("A new employee has started working in branch " + branch); **ϴ(1)**

}

public void addNeed(product pro, int num){ **ϴ(n)**

if(num<1) System.out.println("Number of products needed should be larger than 1"); **ϴ(1)**

else{

for(int i=0; i<need.size(); i++){ **ϴ(n)**

if(need.get(i) == pro) need.get(i).number += num; **ϴ(1)**

}

}

}

public boolean isInStock(product pro, int num){ **ϴ(n)**

for(int i=0; i<stock.size(); i++){ **ϴ(n)**

if(stock.get(i) == pro){ **ϴ(1)**

if(stock.get(i).getNumber()>=num) return true; **ϴ(1)**

}

}

return false; **ϴ(1)**

}

public void makeSale(product pro, int num, customer cus){ **ϴ(n)**

try{

if(isInStock(pro, num) == false) throw new Exception("Sorry, we are out of that product"); **ϴ(1)**

cus.addOrder(pro, num); **ϴ(1)**

for(int i=0; i<stock.size(); i++){ **ϴ(n)**

if(stock.get(i) == pro) stock.get(i).number -= num; **ϴ(1)**

}

}

catch(Exception e){ System.out.println(e);} **ϴ(1)**

}

public void addProduct(product pro, int num){ **ϴ(n)**

for(int i=0; i<stock.size(); i++){ **ϴ(n)**

if(stock.get(i) == pro){ **ϴ(1)**

stock.get(i).number += num; **ϴ(1)**

stock.get(i).store = branch; **ϴ(1)**

}

}

System.out.println("Product added."); **ϴ(1)**

}

public void removeProduct(product pro, int num){ **ϴ(n)**

for(int i=0; i<stock.size(); i++){ **ϴ(n)**

if(stock.get(i) == pro && stock.get(i).number>=num) stock.get(i).number -= num; **ϴ(1)**

else if(stock.get(i) == pro && stock.get(i).number<num){ **ϴ(1)**

System.out.println("Sorry, we don't have enough of that product, we have informed the manager."); **ϴ(1)**

addNeed(pro, num); **ϴ(n)**

}

}

System.out.println("Product removed."); **ϴ(1)**

}

public void previousOrders(customer cus){ **Tb = ϴ(n3) Tw = ϴ(n4)**

cus.getOrders();**Tb = ϴ(n3) Tw = ϴ(n4)**

}

public customer(String tName, String tSurname, String tEmail, String tPassword){ **ϴ(1)**

try{

if(tEmail.contains("@")==false) throw new Exception("Invalid e-mail address."); **ϴ(1)**

System.out.println("Customer constructor is running..."); **ϴ(1)**

name = tName; **ϴ(1)**

surname = tSurname; **ϴ(1)**

email = tEmail; **ϴ(1)**

password = tPassword; **ϴ(1)**

++cCount; **ϴ(1)**

cNumber = cCount; **ϴ(1)**

System.out.println("New customer " + name + ' ' + surname + "'s customer number is " + cNumber); **ϴ(1)**

cList.add(this); **ϴ(1)**

}

catch(Exception e){ System.out.println(e);} **ϴ(1)**

}

public void productList(){**ϴ(n)**

for(int i=0; i<35; i++){ **ϴ(n)**

System.out.println("Office chair of model" + stock.get(i).getModel() + " in the color " + stock.get(i).getColor());**ϴ(1)**

}

for(int i=35; i<55; i++){ **ϴ(n)**

System.out.println("Office desk of model" + stock.get(i).getModel() + " in the color " + stock.get(i).getColor());**ϴ(1)**

}

for(int i=55; i<95; i++){ **ϴ(n)**

System.out.println("Meeting table of model" + stock.get(i).getModel() + " in the color " + stock.get(i).getColor());**ϴ(1)**

}

for(int i=95; i<107; i++){ **ϴ(n)**

System.out.println("Bookcase of model" + stock.get(i).getModel());**ϴ(1)**

}

for(int i=107; i<119; i++){ **ϴ(n)**

System.out.println("Office cabinet of model " + stock.get(i).getModel());**ϴ(1)**

}

}

protected void addOrder(product pro, int num){ **ϴ(1)**

order.add(pro); **ϴ(1)**

order.getLast().number = num; **ϴ(1)**

}

public void getOrders(){**Tb = ϴ(n3) Tw = ϴ(n4)**

for(int i=0; i<order.sizeLL(); i++){ **ϴ(n)**

for(int j=0; j<order.sizeAL(); j++){ **ϴ(n)**

System.out.println(order.get(i,j).getName() + " of model" + order.get(i,j).getModel());**ϴ(n)**

if(order.get(i,j).getColor() != -1) System.out.println(" int the color" + order.get(i,j).getColor());**ϴ(n)**

}

}

}

public boolean searchProduct(product pro, int num){ **ϴ(n)**

for(int i=0; i<stock.size(); i++){ **ϴ(n)**

if(stock.get(i) == pro && stock.get(i).number>=num){ **ϴ(1)**

System.out.println("This product is in stock"); **ϴ(1)**

return true; **ϴ(1)**

}

}

return false; **ϴ(1)**

}

public void shopOnline(product pro, int num, String address, String phone){ **ϴ(n)**

try{

if(phone.length() != 11) throw new Exception("Invalid phone number"); **ϴ(1)**

removeProduct(pro, num); **ϴ(n)**

addOrder(pro, num); **ϴ(1)**

System.out.println("Your order of " + num + ' ' + pro.getName() + "s has been shipped to the address \"" + address + "\" and your phone number is " + phone); **ϴ(1)**

}

catch(Exception e){ **ϴ(1)**

System.out.println(e); **ϴ(1)**

System.out.println("Sorry, your order to \"" + address + "\" cannot be shipped"); **ϴ(1)**

System.out.println("You can make sure your information is valid and then try again."); **ϴ(1)**

}

}

public void whichStore(product pro){ **ϴ(n)**

for(int i=0; i<stock.size(); i++){ **ϴ(n)**

if(stock.get(i).getName() == pro.getName() && stock.get(i).getModel() == pro.getModel() &&

stock.get(i).getColor() == pro.getColor())**ϴ(1)** System.out.println("This product is in branch" + stock.get(i).store); **ϴ(1)**

}

}

public void equal(product pro){ **ϴ(1)**

this.model = pro.getModel();**ϴ(1)**

this.number = pro.getNumber();**ϴ(1)**

this.name = pro.getName();**ϴ(1)**

this.color = pro.getColor(); **ϴ(1)**

}

public KWArrayList(){**ϴ(1)**

capacity = INITIAL\_CAPACITY; **ϴ(1)**

theData = (E[]) new Object[capacity]; **ϴ(1)**

}

public KWArrayList(int init){ **ϴ(1)**

capacity = init; **ϴ(1)**

theData = (E[]) new Object[capacity]; **ϴ(1)**

}

public boolean add(E anEntry){ **ϴ(1)(amortized)**

if(size == capacity){ **ϴ(1)**

reallocate();**ϴ(1)(amortized)**

}

theData[size] = anEntry; **ϴ(1)**

size++; **ϴ(1)**

return true; **ϴ(1)**

}

public void add(int index, E anEntry){ **ϴ(n)**

if(index < 0 || index > size){ **ϴ(1)**

throw new ArrayIndexOutOfBoundsException(index); **ϴ(1)**

}

if(size == capacity){ **ϴ(1)**

reallocate();**ϴ(n)**

}

// Shift data in elements from index to size - 1

for(int i = size; i > index; i--){ **ϴ(n)**

theData[i] = theData[i - 1]; **ϴ(1)**

}

// Insert the new item.

theData[index] = anEntry; **ϴ(1)**

size++; **ϴ(1)**

}

public E get(int index) { **ϴ(1)**

if(index < 0 || index >= size){ **ϴ(1)**

throw new ArrayIndexOutOfBoundsException(index); **ϴ(1)**

}

return theData[index]; **ϴ(1)**

}

public E set(int index, E newValue){ **ϴ(1)**

if(index < 0 || index >= size){ **ϴ(1)**

throw new ArrayIndexOutOfBoundsException(index); **ϴ(1)**

}

E oldValue = theData[index]; **ϴ(1)**

theData[index] = newValue; **ϴ(1)**

return oldValue; **ϴ(1)**

}

public E remove(int index){ **ϴ(n)**

if(index < 0 || index >= size){ **ϴ(1)**

throw new ArrayIndexOutOfBoundsException(index); **ϴ(1)**

}

E returnValue = theData[index]; **ϴ(1)**

for(int i = index + 1; i < size; i++){ **ϴ(n)**

theData[i - 1] = theData[i]; **ϴ(1)**

}

size--; **ϴ(1)**

return returnValue; **ϴ(1)**

}

private void reallocate(){**ϴ(n)**

capacity = 2 \* capacity; **ϴ(1)**

theData = Arrays.copyOf(theData, capacity); **ϴ(n)**

}

public int indexOf(E target){ **ϴ(n)**

for(int i=0; i<size; i++){ **ϴ(n)**

if(theData[i] == target) return i; **ϴ(1)**

}

return -1; **ϴ(1)**

}

public int size(){**ϴ(1)**

return size; **ϴ(1)**

}

public E getLast(){**ϴ(n)**

return get(size()-1); **ϴ(n)**

}

private Node(E dataItem) { **ϴ(1)**

data = dataItem; **ϴ(1)**

}

private Node(E dataItem, Node<E> nodeRef) { **ϴ(1)**

data = dataItem; **ϴ(1)**

next = nodeRef; **ϴ(1)**

}

public KWListIter(int i) { **ϴ(n)**

// Validate i parameter.

if (i < 0 || i > size) { **ϴ(1)**

throw new IndexOutOfBoundsException("Invalid index " + i);

}

lastItemReturned = null; // No item returned yet. **ϴ(1)**

// Special case of last item.

if (i == size) { **ϴ(1)**

index = size; **ϴ(1)**

nextItem = null; **ϴ(1)**

} else { // Start at the beginning

nextItem = head; **ϴ(1)**

for (index = 0; index < i; index++) { **ϴ(n)**

nextItem = nextItem.next; **ϴ(1)**

}

}

}

public boolean hasNext() { **ϴ(1)**

return nextItem != null; **ϴ(1)**

}

public E next() { **ϴ(1)**

if (!hasNext()) { **ϴ(1)**

throw new NoSuchElementException();**ϴ(1)**

}

lastItemReturned = nextItem; **ϴ(1)**

nextItem = nextItem.next; **ϴ(1)**

index++; **ϴ(1)**

return lastItemReturned.data; **ϴ(1)**

}

public boolean hasPrevious() { **ϴ(1)**

return (nextItem == null && size != 0)

|| nextItem.prev != null; **ϴ(1)**

}

public E previous() { **ϴ(1)**

if (!hasPrevious()) { **ϴ(1)**

throw new NoSuchElementException();**ϴ(1)**

}

if (nextItem == null) { // Iterator is past the last element **ϴ(1)**

nextItem = tail; **ϴ(1)**

} else {

nextItem = nextItem.prev; **ϴ(1)**

}

lastItemReturned = nextItem; **ϴ(1)**

index--; **ϴ(1)**

return lastItemReturned.data; **ϴ(1)**

}

public void add(E obj) { **ϴ(1)**

if (head == null) { // Add to an empty list. **ϴ(1)**

head = new Node<>(obj); **ϴ(1)**

tail = head;**ϴ(1)**

} else if (nextItem == head) { // Insert at head. **ϴ(1)**

Node<E> newNode = new Node<>(obj); **ϴ(1)**

newNode.next = nextItem; // Step 1 **ϴ(1)**

nextItem.prev = newNode; // Step 2 **ϴ(1)**

head = newNode; // Step 3 **ϴ(1)**

} else if (nextItem == null) { // Insert at tail. **ϴ(1)**

Node<E> newNode = new Node<>(obj); **ϴ(1)**

tail.next = newNode; // Step 1 **ϴ(1)**

newNode.prev = tail; // Step 2 **ϴ(1)**

tail = newNode; // Step 3 **ϴ(1)**

} else { // Insert into the middle.

Node<E> newNode = new Node<>(obj); **ϴ(1)**

newNode.prev = nextItem.prev; // Step 1 **ϴ(1)**

nextItem.prev.next = newNode; // Step **ϴ(1)**

newNode.next = nextItem; // Step 3 **ϴ(1)**

nextItem.prev = newNode; // Step 4 **ϴ(1)**

}

size++; **ϴ(1)**

index++; **ϴ(1)**

lastItemReturned = null; **ϴ(1)**

} // End of method add.

public void set(E obj){ **ϴ(1)**

if(lastItemReturned == null) throw new IllegalStateException();**ϴ(1)**

else lastItemReturned.data = obj; **ϴ(1)**

}

public void remove(){**ϴ(1)**

if(lastItemReturned == null) throw new IllegalStateException();**ϴ(1)**

else lastItemReturned.prev.next = null; **ϴ(1)**

}

public int previousIndex(){**ϴ(1)**

if(index == 0) return -1; **ϴ(1)**

lastItemReturned = lastItemReturned.prev; **ϴ(1)**

nextItem = lastItemReturned.next; **ϴ(1)**

index--; **ϴ(1)**

return index; **ϴ(1)**

}

public int nextIndex(){**ϴ(1)**

if(size == index+1) return size; **ϴ(1)**

lastItemReturned = lastItemReturned.next; **ϴ(1)**

nextItem = lastItemReturned.next; **ϴ(1)**

index++; **ϴ(1)**

return index; **ϴ(1)**

}

public ListIterator<E> listIterator(){**ϴ(1)**

return listIterator(0); **ϴ(1)**

}

public ListIterator<E> listIterator(int index){ **ϴ(n)**

KWListIter listIterator = new KWListIter(index); **ϴ(n)**

return listIterator; **ϴ(1)**

}

public void addFirst(E item) { **ϴ(1)**

head = new Node<>(item, head); **ϴ(1)**

size++; **ϴ(1)**

}

public void addLast(E item) { **ϴ(1)**

tail = new Node<>(item, tail); **ϴ(1)**

size++; **ϴ(1)**

}

public E getFirst() { **ϴ(1)**

if(size == 0) throw new NoSuchElementException("No such element exception"); **ϴ(1)**

else return head.data; **ϴ(1)**

}

public E getLast() { **ϴ(1)**

if(size == 0) throw new NoSuchElementException("No such element exception"); **ϴ(1)**

else return tail.data; **ϴ(1)**

}

public void add(E obj) { **ϴ(1)**

addLast(obj); **ϴ(1)**

}

public void add(int index, E obj) { **ϴ(n)**

listIterator(index).add(obj); **ϴ(n) + ϴ(1)**

}

public E get(int index) { **ϴ(n)**

return listIterator(index).next();**ϴ(n)**

}

public E remove(int index){ **ϴ(n)**

Node<E> t = new Node<E>(); **ϴ(1)**

if(index == size-1){ **ϴ(1)**

t = tail; **ϴ(1)**

tail = tail.prev; **ϴ(1)**

}else if(size == 0) throw new IllegalStateException();**ϴ(1)**

else if(size == 1){ **ϴ(1)**

head = null; **ϴ(1)**

tail = null; **ϴ(1)**

}else{

Node<E> temp = new Node<E>(); **ϴ(1)**

temp = head; **ϴ(1)**

for(int i=0; i<index; i++){ **ϴ(n)**

if(i == index){ **ϴ(1)**

t=temp; **ϴ(1)**

temp.prev.next = temp.next; **ϴ(1)**

}

temp = temp.next; **ϴ(1)**

}

}

return t.data; **ϴ(1)**

}

public HybridList(){**ϴ(1)**

list = new KWLinkedList<>(); **ϴ(1)**

}

public HybridList(int MAX\_NUMBER){ **ϴ(1)**

this.MAX\_NUMBER = MAX\_NUMBER; **ϴ(1)**

list = new KWLinkedList<>(); **ϴ(1)**

}

public E get(int i, int j){ **ϴ(n)**

KWArrayList<E> temp = new KWArrayList<>(); **ϴ(1)**

temp = list.get(i); **ϴ(1)**

return temp.get(j); **ϴ(n)**

}

public void add(E obj){ **Tw = ϴ(n) Tb = ϴ(n)**

if(list.get(list.size() - 1).size() == MAX\_NUMBER){ **ϴ(n)**

KWArrayList<E> temp = new KWArrayList();**ϴ(1)**

temp.add(obj); **ϴ(1)**

list.add(temp); **ϴ(1)**

}else list.get(list.size()-1).add(obj); **ϴ(n)**

size++; **ϴ(1)**

}

public E getLast(){**ϴ(1)**

KWArrayList<E> temp = new KWArrayList<>(); **ϴ(1)**

temp = list.getLast();**ϴ(1)**

return temp.getLast();**ϴ(1)**

}

public int sizeLL(){**ϴ(1)**

return list.size(); **ϴ(1)**

}

public int sizeAL(){**ϴ(1)**

return list.getLast().size();**ϴ(1)**

}