21.1 Introduction to memory management

An ArrayList stores a list of items in contiguous memory locations, which enables immediate access to any element at index i of ArrayList v by using the get() and set() methods — the program just adds i to the starting address of the first element in v to arrive at the element. The methods `add(objRef)` and `add(i, objRef)` append and insert items into an ArrayList, respectively. Now recall that inserting an item at locations other than the end of the ArrayList requires making room by shifting higher-indexed items. Similarly, removing (via the `remove(i)` method) an item requires shifting higher-indexed items to fill the gap. Each shift of an item from one element to another element requires a few processor instructions. This issue exposes the ***ArrayList add/remove performance problem***.

For ArrayLists with thousands of elements, a single call to add() or remove() can require thousands of instructions, so if a program does many insert or remove operations on large ArrayLists, the program may run very slowly. The following animation illustrates shifting during an insertion operation.

PARTICIPATION ACTIVITY 21.1.1: ArrayList add() performance problem.

Animation captions:

1. Inserting an item at a specific location in an ArrayList requires make room for the item by shifting higher-indexed items.

The shifting of elements done by add() and remove() requires several processor instructions per element. Doing many insertions/removes on large ArrayLists can take a significantly long time.

The following program can be used to demonstrate the issue. The user inputs an ArrayList size, and a number of elements to insert. The program then carries out several tasks. The program creates an ArrayList of size numElem, writes an arbitrary value to all elements, performs numOps appends, numOps inserts, and numOps removes.

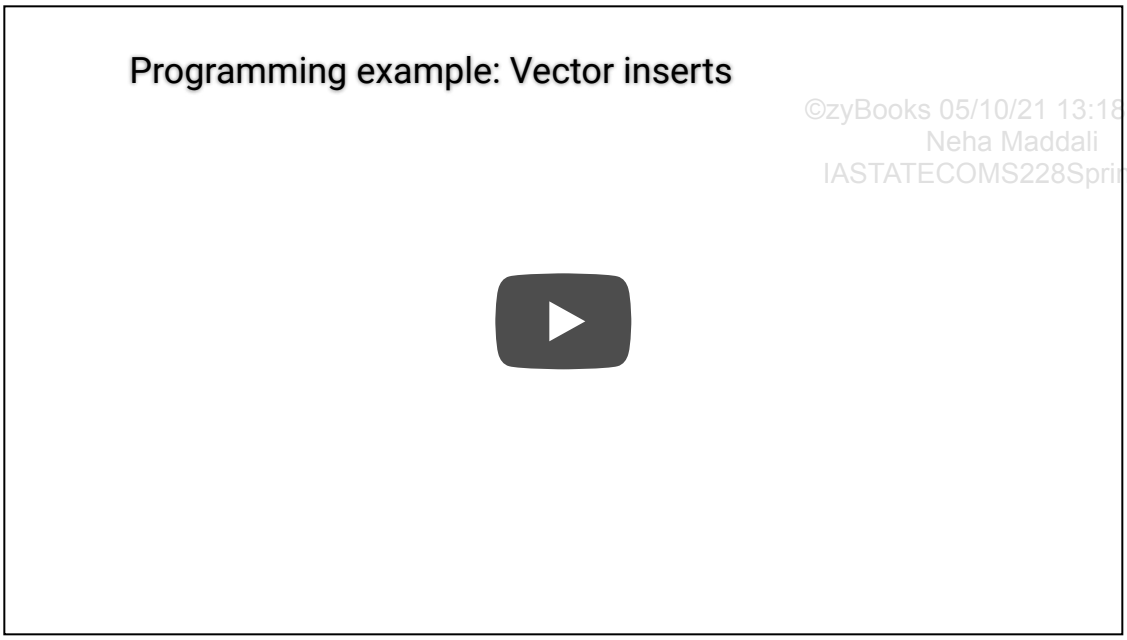
Figure 21.1.1: Program illustrating how slow ArrayList add() and remove() operations can be.

```
import java.util.ArrayList;
import java.util.Scanner;

public class ArrayListAddRemove {
   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);
      ArrayList<Integer> myInts = new ArrayList<Integer>(); // Dummy array list to demo ops
      int numElem;                                          // User defined array size
      int numOps;                                           // User defined number of inserts
      int i;                                                // Loop index

      System.out.print("\nEnter initial ArrayList size: ");
      numElem = scnr.nextInt();

      System.out.print("Enter number of ArrayList adds: ");
      numOps = scnr.nextInt();

      System.out.print("  Adding elements to ArrayList...");

      myInts.clear();
      for (i = 0; i < numElem; ++i) {
         myInts.add(new Integer(0));
      }

      System.out.println("done.");
      System.out.print("  Writing to each element...");

      for (i = 0; i < numElem; ++i) {
         myInts.set(i, new Integer(777)); // Any value
      }

      System.out.println("done.");
      System.out.print("  Doing " + numOps + " additions at the end...");

      for (i = 0; i < numOps; ++i) {
         myInts.add(new Integer(888)); // Any value
      }

      System.out.println("done.");
      System.out.print("  Doing " + numOps + " additions at index 0...");

      for (i = 0; i < numOps; ++i) {
         myInts.add(0, new Integer(444));
      }
      System.out.println("done.");
      System.out.print("  Doing " + numOps + " removes...");

      for (i = 0; i < numOps; ++i) {
         myInts.remove(0);
      }

      System.out.println("done.");
   }
}
```



```
Enter initial ArrayList size: 100000
Enter number of ArrayList adds: 40000
  Adding elements to ArrayList...done.       (fast)
  Writing to each element...done.            (fast)
  Doing 40000 additions at the end...done.   (fast)
  Doing 40000 additions at index 0...done.   (SLOW)
  Doing 40000 removes...done.                (SLOW)
```

Video 21.1.1: ArrayList inserts.



The appends are fast because they do not involve any shifting of elements, whereas each insert requires 500,000 elements to be shifted — one at a time. 7,500 inserts thus requires 3,750,000,000 (over 3 billion) shifts.

One way to make inserts or removes faster is to use a different approach for storing a list of items. The approach does not use contiguous memory locations. Instead, each item contains a "pointer" to the next item's location in memory, as well as, the data being stored. Thus, inserting a new item B between existing items A and C just requires changing A to refer to B's memory location, and B to refer to C's location, as shown in the following animation.

PARTICIPATION ACTIVITY 21.1.2: A list avoids the shifting problem.

Animation captions:

1. List's first two items initially: (A, C, ...). Item A points to the next item at location 88. Item C points to next item at location 113 (not shown).
2. To insert new item B after item A, the new item B is first added to memory at location 90.
3. Item A is updated to point to location 90. Item B is set to point to location 88. New list is (A, B, C, ...). No shifting of items after C was required.

The animation begins with a list having some number of items, with the first two items being A and C. The first item has data A and a next reference storing the address 88, which refers to the next item's

location in memory. That second item has data C, and a next reference storing address 113, which refers to the next item (not shown). The animation shows a new item being created at memory location 90, having data B. To keep the list in sorted order, item B should go between A and C in the list. So item A's next reference is changed to point to B's location of 90, and B's next reference is set to address 88.

A ***linked list*** is a list wherein each item contains not just data but also a reference — a *link* — to the next item in the list. Comparing ArrayLists and linked lists:

- *ArrayList*: Stores items in contiguous memory locations. Supports quick access to i'th element via the set() and get() methods, but may be slow for inserts or removes on large ArrayLists due to necessary shifting of elements.
- *Linked list*: Stores each item anywhere in memory, with each item referring to the next item in the list. Supports fast inserts or removes, but access to i'th element may be slow as the list must be traversed from the first item to the i'th item. Also uses more memory due to storing a link for each item.

PARTICIPATION ACTIVITY 21.1.3: ArrayList performance.

1) Appending a new item to the end of a 1000 element ArrayList requires how many elements to be shifted?

Check Show answer

2) Inserting a new item at the beginning of a 1000 element ArrayList requires how many elements to be shifted?

Check Show answer

21.2 A first linked list

A common use of objects and references is to create a list of items such that an item can be efficiently inserted somewhere in the middle of the list, without the shifting of later items as required for an ArrayList. The following program illustrates how such a list can be created. A class is defined to represent each list item, known as a ***list node***. A node is comprised of the data to be stored in each list

item, in this case just one int, and a reference to the next node in the list. A special node named head is created to represent the front of the list, after which regular items can be inserted.

Figure 21.2.1: A basic example to introduce linked lists.

IntNode.java

```
public class IntNode {
   private int dataVal;         // Node data
   private IntNode nextNodePtr; // Reference to the next node

   public IntNode() {
      dataVal = 0;
      nextNodePtr = null;
   }

   // Constructor
   public IntNode(int dataInit) {
      this.dataVal = dataInit;
      this.nextNodePtr = null;
   }

   // Constructor
   public IntNode(int dataInit, IntNode nextLoc) {
      this.dataVal = dataInit;
      this.nextNodePtr = nextLoc;
   }

   /* Insert node after this node.
    Before: this -- next
    After:  this -- node -- next
    */
   public void insertAfter(IntNode nodeLoc) {
      IntNode tmpNext;

      tmpNext = this.nextNodePtr;
      this.nextNodePtr = nodeLoc;
      nodeLoc.nextNodePtr = tmpNext;
   }

   // Get location pointed by nextNodePtr
   public IntNode getNext() {
      return this.nextNodePtr;
   }

   public void printNodeData() {
      System.out.println(this.dataVal);
   }
}
```

CustomLinkedList.java

```
public class CustomLinkedList {
   public static void main(String[] args) {
      IntNode headObj;  // Create IntNode reference variables
      IntNode nodeObj1;
      IntNode nodeObj2;
      IntNode nodeObj3;
      IntNode currObj;

      // Front of nodes list
      headObj = new IntNode(-1);

      // Insert more nodes
      nodeObj1 = new IntNode(555);
      headObj.insertAfter(nodeObj1);

      nodeObj2 = new IntNode(999);
      nodeObj1.insertAfter(nodeObj2);

      nodeObj3 = new IntNode(777);
      nodeObj1.insertAfter(nodeObj3);

      // Print linked list
      currObj = headObj;
      while (currObj != null) {
         currObj.printNodeData();
         currObj = currObj.getNext();
      }
   }
}
```

```
-1
555
777
999
```


PARTICIPATION ACTIVITY 21.2.1: Inserting nodes into a basic linked list.

Animation captions:

1. The headObj pointer points to a special node that represents the front of the list. When the list is first created, no list items exists, so the head node's nextNodePtr pointer is null.
2. To insert a node in the list, the new node nodeObj1 is first created with the value 555.
3. To insert the new node, tmpNext is pointed to the head node's next node, the head node's nextNodePtr is pointed to the new node, and the new node's nextNodePtr is pointed to tmpNext.
4. A second node nodeObj2 with the value 999 is inserted at the end of the list, and a third node nodeObj3 with the value 777 is created.
5. To insert nodeObj3 after nodeObj1, tmpNext is pointed to the nodeObj1's next node, the nodeObj1's nextNodePtr is pointed to the nodeObj3, and nodeObj3's nextNodePtr is pointed to tmpNext.

PARTICIPATION ACTIVITY 21.2.2: A first linked list.

Some questions refer to the above linked list code and animation.

1) A linked list has what key advantage over a sequential storage approach like an array or ArrayList?

   - An item can be inserted somewhere in the middle of the list without having to shift all subsequent items.
   - Uses less memory overall.
   - Can store items other than int variables.

2) What is the purpose of a list's head node?

   - Stores the first item in the list.
   - Provides a reference to the first item's node in the list, if such an item exists.
   - Stores all the data of the list.

3) After the above list is done having items inserted, at what memory address is the last list item's node located?

   - 80
   - 82
   - 84
   - 86

4) After the above list has items inserted as above, if a fourth item was inserted at the front of the list, what would happen to the location of node1?

   - Changes from 84 to 86.
   - Changes from 84 to 82.
   - Stays at 84.

In contrast to the above program that declares one reference variable for each item allocated by the new operator, a program commonly declares just one or a few variables to manage a large number of items allocated using the new operator. The following example replaces the above main() method, showing how just two reference variables, currObj and lastObj, can manage 20 allocated items in the list.

Figure 21.2.2: Managing many new items using just a few reference variables.

IntNode.java

```
public class IntNode {
   private int dataVal;         // Node data
   private IntNode nextNodePtr; // Reference to the next node

   public IntNode() {
      dataVal = 0;
      nextNodePtr = null;
   }

   // Constructor
   public IntNode(int dataInit) {
      this.dataVal = dataInit;
      this.nextNodePtr = null;
   }

   // Constructor
   public IntNode(int dataInit, IntNode nextLoc) {
      this.dataVal = dataInit;
      this.nextNodePtr = nextLoc;
   }

   /* Insert node after this node.
    Before: this -- next
    After:  this -- node -- next
    */
   public void insertAfter(IntNode nodeLoc) {
      IntNode tmpNext;

      tmpNext = this.nextNodePtr;
      this.nextNodePtr = nodeLoc;
      nodeLoc.nextNodePtr = tmpNext;
   }

   // Get location pointed by nextNodePtr
   public IntNode getNext() {
      return this.nextNodePtr;
   }

   public void printNodeData() {
      System.out.println(this.dataVal);
   }
}
```


CustomLinkedList.java

```
public class CustomLinkedList {
   public static void main(String[] args) {
      IntNode headObj; // Create IntNode reference variables
      IntNode currObj;
      IntNode lastObj;
      int i;           // Loop index

      headObj = new IntNode(-1); // Front of nodes list
      lastObj = headObj;

      for (i = 0; i < 20; ++i) { // Append 20 rand nums
         int rand = (int)(Math.random() * 100000); // random int (0-99999)
         currObj = new IntNode(rand);

         lastObj.insertAfter(currObj); // Append curr
         lastObj = currObj;
      }

      currObj = headObj; // Print the list
      while (currObj != null) {
         currObj.printNodeData();
         currObj = currObj.getNext();
      }
   }
}
```

```
-1
40271
6951
29273
86846
64952
65650
98162
51229
30690
61008
17489
87486
24318
44035
32368
10906
75441
88659
65688
18443
```


zyDE 21.2.1: Managing a linked list.

Finish the program so that it finds and prints the smallest value in the linked list.

Current file: **IntNode.java** ▾ ault template...

Run

```
1
2  public class IntNode {
3     private int dataVal;         // Node dat
4     private IntNode nextNodePtr; // Referenc
5
6     public IntNode() {
7        dataVal = 0;
8        nextNodePtr = null;
9     }
10
11    // Constructor
12    public IntNode(int dataInit) {
13       this.dataVal = dataInit;
14       this.nextNodePtr = null;
15    }
16
17    // Constructor
18    public IntNode(int dataInit, IntNode nex
19       this.dataVal = dataInit;
20       this.nextNodePtr = nextLoc;
21
```

Normally, a linked list would be maintained by member methods of another class, such as IntList. Private fields of that class might include the list head (a list node allocated by the list class constructor), the list size, and the list tail (the last node in the list). Public member methods might include insertAfter (insert a new node after the given node), pushBack (insert a new node after the last node), pushFront (insert a new node at the front of the list, just after the head), deleteNode (deletes the node from the list), etc.

Exploring further:

- More on linked lists from Oracle's Java tutorials

CHALLENGE ACTIVITY 21.2.1: Enter the output of the program using the linked list.

Start

Type the program's output

CallPlaylistSong.java | PlaylistSong.java

```
public class CallPlaylistSong {
   public static void main(String[] args) {
      PlaylistSong headObj = null;
      PlaylistSong firstSong = null;
      PlaylistSong secondSong = null;
      PlaylistSong thirdSong = null;
      PlaylistSong currObj = null;

      headObj = new PlaylistSong("head");

      firstSong = new PlaylistSong("Lacrimosa");
      headObj.InsertAfter(firstSong);

      secondSong = new PlaylistSong("Vocalise");
      firstSong.InsertAfter(secondSong);

      thirdSong = new PlaylistSong("Canon");
      secondSong.InsertAfter(thirdSong);

      currObj = headObj;

      while (currObj != null) {
         currObj.PrintNodeData();
         currObj = currObj.GetNext();
      }
   }
}
```


1 2

Check Next

CHALLENGE ACTIVITY 21.2.2: Linked list negative values counting.

Assign negativeCntr with the number of negative values in the linked list.

```
1  // ===== Code from file IntNode.java =====
2  public class IntNode {
3     private int dataVal;
4     private IntNode nextNodePtr;
5
6     public IntNode(int dataInit, IntNode nextLoc) {
7        this.dataVal = dataInit;
8        this.nextNodePtr = nextLoc;
9     }
10
11    public IntNode(int dataInit) {
12       this.dataVal = dataInit;
13       this.nextNodePtr = null;
14    }
15
16    /* Insert node after this node.
17    * Before: this -- next
18    * After:  this -- node -- next
19    */
20    public void insertAfter(IntNode nodePtr) {
21       IntNode tmpNext;
```

Run

21.3 Memory regions: Heap/Stack

A program's memory usage typically includes four different regions:

- ***Code*** — The region where the program instructions are stored.
- ***Static memory*** — The region where static fields are allocated. The name "static" comes from these variables not changing (static means not changing); they are allocated once and last for the duration of a program's execution, their addresses staying the same.
- ***The stack*** — The region where a method's local variables are allocated during a method call. A method call adds local variables to the stack, and a return removes them, like adding and removing dishes from a pile; hence the term "stack." Because this memory is automatically allocated and deallocated, it is also called ***automatic memory***.

- ***The heap*** — The region where the "new" operator allocates memory for objects. The region is also called ***free store***.

In Java, the code and static memory regions are actually integrated into a region of memory called the ***method area*** , which also stores information for every class type used in the program.

PARTICIPATION ACTIVITY 21.3.1: Use of the four memory regions.

Animation captions:

1. The code regions store program instructions. myStaticField is a static field and is stored in the static memory region. Code and static regions last for the entire program execution.
2. Method calls push local variables on the program stack. When main() is called, the variables myInt and myInteger are added on the stack.
3. new allocates memory on the heap for an Integer object and returns the address of the allocated memory, which is assigned to myInteger.
4. Calling myMethod() grows the stack, pushing the method's local variables on the stack. Those local variables are removed from the stack when the method returns.
5. When main() completes, main's local variables are removed from the stack.

PARTICIPATION ACTIVITY 21.3.2: Stack and heap definitions.

The heap | **Free store** | **The stack** | **Static memory** | **Automatic memory** | **Code**

	A method's local variables are allocated in this region while a method is called.
	The memory allocation operator (new) affects this region.
	Static class fields are allocated in this region once for the duration of the program.
	Another name for "The heap" because the programmer has explicit control of this memory.
	Instructions are stored in this region.

Another name for "The stack" because the programmer does not explicitly control this memory.

Reset

21.4 Basic garbage collection

Because the amount of memory available to a program is finite, objects allocated to the heap must eventually be deallocated when no longer needed by the program. The Java programming language uses a mechanism called ***garbage collection*** wherein a program's executable includes automatic behavior that at various intervals finds all unreachable — i.e., unused — allocated memory locations, and automatically frees such memory locations in order to enable memory reuse. Garbage collection can present the programmer with the illusion of a nearly unlimited memory supply at the expense of runtime overhead.

In order to determine which allocated objects the program is currently using at runtime, the Java virtual machine keeps a count, known as a ***reference count***, of all reference variables that are currently referring to an object. If the reference count is zero, then the object is considered an ***unreachable object*** and is eligible for garbage collection, as no variables in the program refer to the object. The Java virtual machine marks unreachable objects, and deallocation occurs the next time the Java virtual machine invokes the garbage collector. The following animation illustrates.

PARTICIPATION ACTIVITY 21.4.1: Marking unused objects for deallocation.

Animation captions:

1. myInt and myOtherInt are reference type variables, which are initialized with null.
2. The program allocates memory for an Integer object and assigns myInt with a reference to the object's memory location. That Integer object's reference count is incremented, which indicates one reference variable currently refers to the object.
3. myOtherInt is assigned with a reference to the same Integer object's memory location. That Integer object's reference count is incremented, which indicates two reference variables currently refers to the object.
4. myInt is assigned with null, indicating that the reference variable no longer refers to the Integer object. That Integer object's reference count is decremented to one, which indicates one reference variable currently refers to the object.
5. myOtherInt is assigned with null, so Integer object's reference count is decremented to zero, which indicates no reference variables currently refers to the object. When an object's reference becomes zero, the Java virtual machine marks that object for deallocation.

The program initially allocates memory for an Integer object and assigns a reference to the object's memory location to variables myInt and myOtherInt. Thus, the object's reference count is displayed as two at that point in the program's execution. After the object is no longer needed, the reference variables are assigned a value of *null*, indicating that the reference variables no longer refer to an object. Consequently, the object's reference count decrements to zero, and the Java virtual machine marks that object for deallocation.

PARTICIPATION ACTIVITY 21.4.2: Garbage collection.

Garbage collection **Unreachable object** **Object reference count**

	Object that is not referenced by any valid reference variables in the program.
	Value updated by the Java virtual machine in order to keep track of the number of variables referencing an object.
	Automatic process of finding unused allocated memory locations and deallocating that unreachable memory.

Reset

21.5 Garbage collection and variable scope

A programmer does not explicitly have to set a reference variable to null in order to indicate that the variable no longer refers to an object. The Java virtual machine can automatically infer a null reference once the variable goes out of scope — i.e., the reference variable is no longer visible to the program. For example, local reference variables that are declared within a method go out of scope as soon as the method returns. The Java virtual machine decrements the reference counts associated with the objects referred to by any local variables within the method.

PARTICIPATION 21.5.1: Marking unused objects in methods.

ACTIVITY

Animation captions:

1. binaryStr is a local reference variable that refers to a String object used to store the binary representation of the integer inNum. The assignment of binaryStr increments the object's reference count.
2. When the method returns, the reference variable binaryStr goes out of scope. So, the string object's reference count becomes zero, and the Java virtual machine marks the object for deallocation.

Every time CountBits() is invoked, the method declares a local reference variable called binaryStr, which refers to a newly allocated String object used to store the binary representation of the integer num. The assignment of binaryStr increments the object's reference count. When the method returns, the reference variable binaryStr goes out of scope, and the Java virtual machine will decrement the reference count for the String object. The reference count for that String object becomes zero and the object is marked for deallocation, which occurs whenever the Java virtual machine invokes the garbage collector.

Although CountBits() happens to allocate binaryStr in the same memory location whenever CountBits() is called, note that Java makes no such guarantee. Also, recall that main() is itself a method. Thus, the Java virtual machine will decrement the reference count of any objects associated with reference variables declared in main() upon returning from main().

PARTICIPATION ACTIVITY 21.5.2: Garbage collection and variable scope.

1) A method's local reference variables automatically go out of scope when the method returns.
   - ○ True
   - ○ False

2) A programmer must explicitly set all reference variables to null in order to indicate that the objects to which the variables referred are no longer in use.
   - ○ True
   - ○ False

21.6 Java example: Employee list using

ArrayLists

zyDE 21.6.1: Managing an employee list using an ArrayList.

The following program allows a user to add to and list entries from an ArrayList, which maintains a list of employees.

1. Run the program, and provide input to add three employees' names and related da
   Then use the list option to display the list.
2. Modify the program to implement the deleteEmployee method.
3. Run the program again and add, list, delete, and list again various entries.

Load default templ

```
1  import java.util.ArrayList;
2  import java.util.Scanner;
3
4  public class EmployeeManager {
5     static Scanner scnr = new Scanner(System.in);
6
7     public static void main(String[] args) {
8        final int MAX_ELEMENTS = 10;
9        final char EXIT_CODE = 'X';
10       final String PROMPT_ACTION = "Add, Delete, List or eXit (a,d,l,x): ";
11       ArrayList<String> name       = new ArrayList<String>(MAX_ELEMENTS);
12       ArrayList<String> department = new ArrayList<String>(MAX_ELEMENTS);
13       ArrayList<String> title      = new ArrayList<String>(MAX_ELEMENTS);
14       char userAction;
15
16       // Loop until the user enters the exit code.
17       userAction = getAction(PROMPT_ACTION);
18
19       while (userAction != EXIT_CODE) {
20          if(userAction == 'A') {
21             addEmployee(name, department, title);
```

```
a
Rajeev Gupta
Sales
```

Run

Below is a solution to the above problem.

zyDE 21.6.2: Managing an employee list using an ArrayList (solution).

Load default templ

```
1  import java.util.ArrayList;
2  import java.util.Scanner;
3
4  public class MemoryManagement {
5     static Scanner scnr = new Scanner(System.in);
6
7     public static void main(String[] args) {
8        final int MAX_ELEMENTS = 10;
9        final char EXIT_CODE = 'X';
10       final String PROMPT_ACTION = "Add, Delete, List, or eXit (a,d,l,x): ";
11       ArrayList<String> name       = new ArrayList<String>(MAX_ELEMENTS);
12       ArrayList<String> department = new ArrayList<String>(MAX_ELEMENTS);
13       ArrayList<String> title      = new ArrayList<String>(MAX_ELEMENTS);
14       char userAction;
15
16       // Loop until the user enters the exit code
17       userAction = getAction(PROMPT_ACTION);
18
19       while (userAction != EXIT_CODE) {
20          if(userAction == 'A') {
21             addEmployee(name, department, title);
```



```
a
Rajeev Gupta
Sales
```

Run

21.7 LAB: Library book sorting

This section's content is not available for print.

21.8 LAB: Mileage tracker for a runner

This section's content is not available for print.

21.9 LAB: Inventory (linked lists: insert at the front of a list)

This section's content is not available for print.

21.10 LAB: Playlist (output linked list)

This section's content is not available for print.

21.11 LAB: Grocery shopping list (linked list: inserting at the end of a list)

This section's content is not available for print.

©zyBooks 05/10/21 13:18 728163
Neha Maddali
IASTATECOMS228Spring2021