**Lesson 8 – Linked Lists II**

**Learning Objectives:**

* Use a node class to build a reference-base list (linked list).
* Write methods to insert and remove items within a linked list.
* Implement a reference-based linked-list in Java for storing integers.
* Explain alternative variations of the linked list.

**Reading:**

* Chapter 5, Section 3-4 of the text.

**The Linked-List:**

* A close examination of the array-based implementation of the ADT list reveals that an array is not always the best data structure to use to maintain a collection of data. An array has a fixed-size—at least in most commonly used programming languages—but the ADT List can have an arbitrary length.
* Also, an array orders its items physically and, you must shift data when you insert or delete an item at a specified position. Shifting data can be a time-consuming process that should be avoided, if possible.
* Like you implemented in the set, we can modify the array implementation so that, when the array is full, we create a new, larger array and copy the contents over. This process is time consuming, as well.
* We want a list implementation that does not involve shifting and copying data to a new array.

**Object References:**

* Note that an object of a given class does not come into existence until you apply the **new** operator.
* When you declare a variable that refers to the object, you are creating a **reference** to the object. A **reference variable**, or simply a **reference**, contains the location, or **address** in memory, of an object.
* By using a reference to a particular object, you can locate the object and, for example, access the object’s public members.
* For example:

Graphical user interface, application

Description automatically generated

Shape

Description automatically generated with low confidence

* Key Concepts:
  + The declaration

Integer intRef;

statically allocates a reference variable intRef whose value is null. When a reference variable contains null, it does not reference anything.

* + intRef can reference an Integer object. The statement

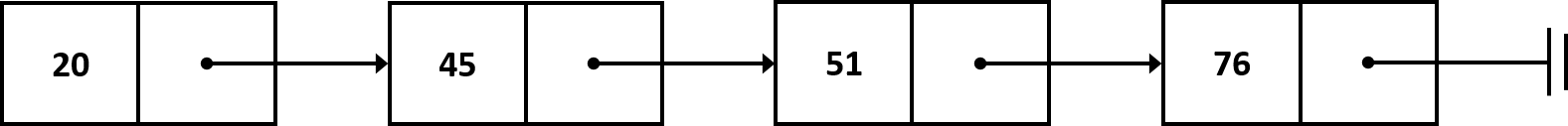
intRef = **new** Integer(5);

dynamically allocates an Integer object referenced by intRef. (However, see item 3 on the list.)

* + If, for some reason, new cannot instantiate an object of the class represented, it may throw a java.lang.InstantiationException or a java.lang.IllegalAccessException. Thus, you can place the following statement within a try block to test whether memory was successfully allocated:

intRef = new Integer(5);

* + When the last reference to an object is removed, the object is marked for garbage collection.
* To get a conceptual notion of a list implementation that does not involve shifting, consider the following figure:

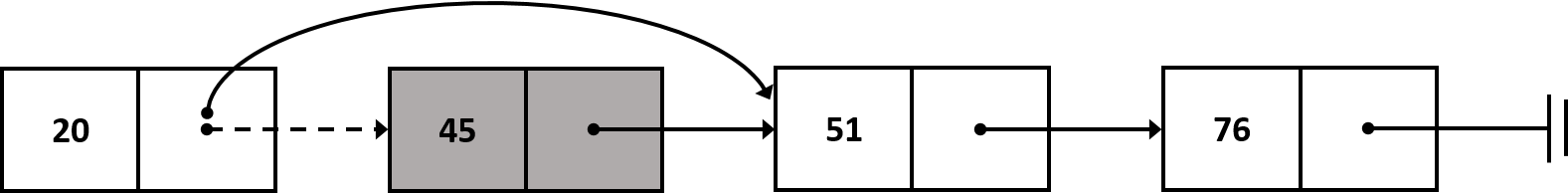


* In the diagram, each item of the list is actually linked to the next item. Thus, if you know where an item is, you can determine its successor, which can be physically located anywhere in memory.
* This flexibility not only allows you to insert and delete data items without shifting data, but it also allows you to increase the size of the list easily.
  + If you need to insert a new item, you simply find its place in the list and set two **links**.

Shape

Description automatically generated with medium confidence

* + Similarly, to delete an item, you find the item and change a link to bypass the item.



* Because the items in this data structure are linked to one another, it is called a **linked list**. Using this kind of structure, a linked list is able to grow as needed. In many applications, this flexibility gives a linked list a significant advantage.

**Reference-Based Linked Lists:**

* A linked list contains components that are linked to one another. Each component—usually called a node—contains both data and a “link” to the next item.

**The Node Class:**

public class Node<E> {

    private E item;

    private Node<E> nextNode;

    public Node(E nodeItem) {

        this.item = nodeItem;

        this.nextNode = null;

    }

    public E getItem() {

        return this.item;

    }

    public void setNext(Node<E> nextNode){

        this.nextNode = nextNode;

    }

    public Node<E> getNext() {

        return this.nextNode;

    }

    @Override  
// Wait until the end of class to implement this

// method in order to demonstrate recursion within

// the Node class.

    public String toString() {

        if (this.nextNode == null)

            return "(" + item.toString() + ", null";

        else

            return "(" + item.toString() + ", " + nextNode.toString() + ")";

    }

}

public class Friend {

    private String username;

    private boolean status = false; // false - offline; true online

    public Friend(String username, boolean status) {

        this.username = username;

        this.status = status;

    }

    public Friend(String username) {

        this(username, false);

    }

    public void setStatus(boolean online) {

        this.status = online;

    }

    @Override

    public String toString() {

        String s = this.status ?  "Online" : "Offline";

        return username + " " + s;

    }

}

**The Linked-List:**

public class FriendsList extends ListADT<Friend> {

    private Node<Friend> head;

    private int numFriends;

    public FriendsList() {

        this.head = null;

        this.numFriends = 0;

    }

    public FriendsList(Friend firstFriend) {

        this.head = new Node<Friend>(firstFriend);

        this.numFriends = 1;

    }

    @Override

    public boolean isEmpty() {

        return this.numFriends == 0;

    }

    @Override

    public int size() {

        return this.numFriends;

    }

    @Override

    public void removeAll() {

        // makes all nodes unreachable

        // and thus garbage collected

        this.head = null;

        this.numFriends = 0;

    }

    // Pre : 0 <= i < numItems

    private Node<Friend> getNodeAt(int i) {

        Node<Friend> n = this.head;

        for (int k = 1; k <= i; k++)

            n = n.getNext();

        return n;

    }

    @Override

    public void add(int index, Friend item) throws ListException {

        if (index < 0 || index > this.size())

            throw new ListException("Index " + index + " is invalid for a list of size " + this.size());

        Node<Friend> newFriend = new Node<Friend>(item);

        if (index == 0) {

            newFriend.setNext(this.head);

            this.head = newFriend;

        } else {

            Node<Friend> previous = this.getNodeAt(index - 1);

            Node<Friend> current = previous.getNext();

            // previous should not go to the new Node

            previous.setNext(newFriend);

            newFriend.setNext(current);

        }

        this.numFriends++;

    }

    @Override

    public Friend get(int index) throws ListException {

        if ((index < 0) || (index >= this.size()))

            throw new ListException("Index " + index + " is invalid for a list of size " + this.size());

        Node<Friend> node = getNodeAt(index);

        return node.getItem();

    }

    @Override

    public void remove(int index) throws ListException {

        if ((index < 0) || (index >= this.size()))

            throw new ListException("Index " + index + " is invalid for a list of size " + this.size());

        if (index == 0)

            this.head = this.head.getNext();

        else {

            Node<Friend> previous = getNodeAt(index - 1);

            Node<Friend> nodeToRemove = previous.getNext();

            Node<Friend> nodeAfter = nodeToRemove.getNext();

            previous.setNext(nodeAfter);

        }

        this.numFriends--;

    }

    @Override

    public String toString() {

        String s = "[";

        if (this.size() != 0) {

            Node<Friend> current = this.head;

            for (int i = 0; i < this.size() - 1; i++) {

                s = s + current.getItem().toString() + ",";

                current = current.getNext();

            }

            s = s + current.getItem().toString();

        }

        return s + "]";

    }

}

**The Main Method:**

public class TestFriendList {

    public static void main(String[] args) throws ListException {

        FriendsList friendList = new FriendsList();

        System.out.println("Testing add: ");

        friendList.add(0, new Friend("magicschoolbusdropout", false));

        friendList.add(1, new Friend("Lezduit", true));

        friendList.add(2, new Friend("HoosierDaddy", true));

        System.out.println(friendList.toString());

        System.out.println();

        System.out.println("Testing remove: ");

        friendList.remove(1);

        System.out.println(friendList);

        System.out.println();

        System.out.println("Testing remove all: ");

        friendList.removeAll();

        System.out.println(friendList);

    }

}

**Announcements:**

* Exam 01 is in one week.
  + True/False; Multiple Choice; Short Answer; Coding