CSCI 132 Basic Data Structures

James Goudy

Abstract

Keywords: .

1 Basic Algorithms

1.1 Java Language

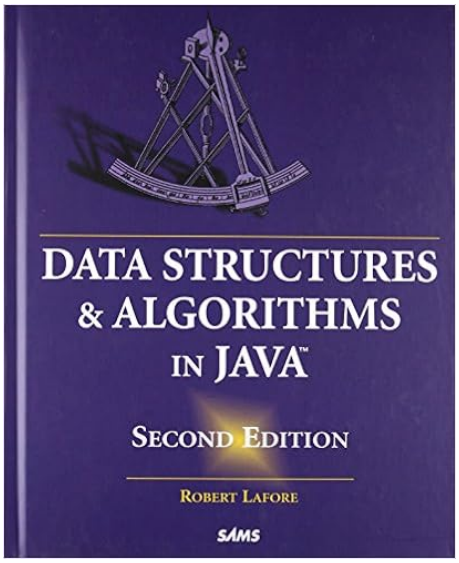
An introduction to the Java Language

> Java is C++ without the guns, clubs and knives. *James Gosling*

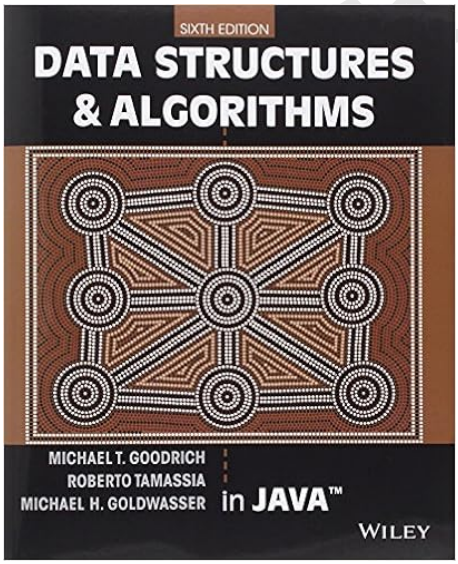
> C makes it easy to shoot yourself in the foot. C++ makes it harder. But when you do, it blows your whole leg off. *Bjarne Stroustrup*

1.2 Data Structures and Algorithms

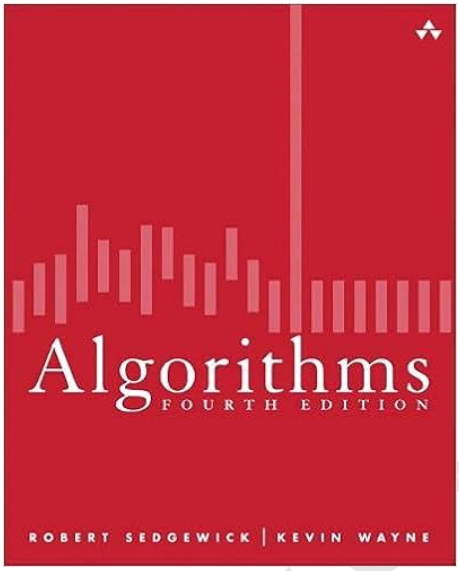
1.2.1 *Textbooks and Online Sources*



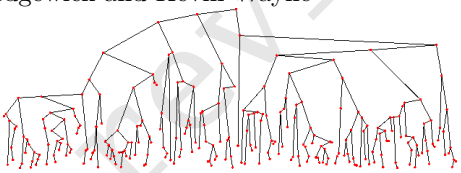
Data Structures and Algorithms in Java Second Edition Robert Lafore



Data Structures and Algorithms in Java 6th Edition Michael T. Goodrich



Algorithms (4th Edition) 4th Edition by Robert Sedgewick and Kevin Wayne



Open Data Structures |

End Of Topic

1.3 Array Techniques

- Array Wrap Around
- Array Add and Delete Data Continuously

End of Topic

1.3.1 *Arrays Core Functionality*

`sort(int[] SampleData, boolean printData)` This method implements an **Insertion Sort** algorithm. Unlike a traditional in-place Insertion Sort, this version copies elements from the input `SampleData` array into a *new* array, `SortedData`, inserting each element into its correct, sorted position within the `SortedData` array as it iterates. The sorted array is returned. If the `printData` flag is true, it prints the partially sorted array after each insertion.

`reverse(int[] arr)` This function reverses the order of elements in the given array **in-place**. It uses a two-pointer approach, with one pointer starting at the beginning (`start`) and the other at the end (`end`) of the array, iteratively swapping the elements they point to until the pointers meet or cross. The array itself is modified, and nothing is returned.

`deleteByValue(int[] arr, int valueToDelete)` This method finds the **first occurrence** of a specified `valueToDelete` in the array. If found, it delegates the actual deletion to the `deleteByPosition` function. If the value is not found, it prints a message and returns the original array unchanged. Because Java arrays have a fixed size, this function (via `deleteByPosition`) returns a **new array** that is one element shorter than the original.

`deleteByPosition(int[] theArray, int pos)` This is a utility function for deleting an element at a specific index (`pos`). It creates a **new array** that is one element smaller and copies all elements from the original array into the new one, skipping the element at the specified position and shifting subsequent elements left to fill the gap. The new, smaller array is returned.

`printArray(int[] theArray, int dataLength)` A simple **utility function** used to print the elements of an array from the beginning up to the specified `dataLength`.

Program Execution in `main` The `main` method orchestrates a demonstration of these functions using an initial data array: {9,1,8,2,7,3,6,5,3}.

1. **Insertion Sort:** It first sorts the initial data, printing the array after each insertion, resulting in a sorted array: {1,2,3,3,5,6,7,8,9}.
2. **Reverse:** It then reverses the resulting sorted array **in-place**, yielding: {9,8,7,6,5,3,3,2,1}.
3. **Delete by Value:** It deletes the first occurrence of the value **3** from the reversed array, returning a new array: {9,8,7,6,5,3,2,1}.
4. **Delete by Position:** Finally, it deletes the element at **index 2** (which is the value 7) from the array resulting from the previous step, producing the final array: {9,8,6,5,3,2,1}.

```
package instinsertinorder;

public class InstInsertInOrder {

    // Insertion -sort into a NEW array; optionally print aft
    static int[] sort(int[] SampleData, boolean printData) {
        int[] SortedData = new int[SampleData.length];
        int elementsSorted = 0; // size of the sorted prefix

        // For each element in SampleData
        for (int i = 0; i < SampleData.length; i++) {
            int key = SampleData[i];
            int j = elementsSorted - 1;

            // While items > key, shift them right
            while (j >= 0 && SortedData[j] > key) {
                SortedData[j + 1] = SortedData[j];
                j - -;
            }

            // Insert key at correct position
            SortedData[j + 1] = key;
            elementsSorted++;

            if (printData) // If enabled, show current sorted
                printArray(SortedData, elementsSorted);
        }
        return SortedData;
    }

    /** Reverse array in place using two pointers. */
    static void reverse(int[] arr) {
        int start = 0, end = arr.length - 1;
        while (start < end) { // While pointers haven't cross
            int tmp = arr[start];
            arr[start] = arr[end];
            arr[end] = tmp;
            start++; end - -;
        }
    }

    /**
     * Delete first occurrence of value; returns NEW array (l
     * If not found, return original.
     */
    static int[] deleteByValue(int[] arr, int valueToDelete)
        int indexToDelete = -1;

        // For each element, check for match
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] == valueToDelete) { // Found target
                indexToDelete = i;
                break;
            }
```

```
    }                                                   }
                                                                10/2025
    if (indexToDelete == -1) { // If value not found
        System.out.println("Value " + valueToDelete + " not found.");
        return arr;
    }

    return deleteByPosition(arr, indexToDelete);
}

/**
 * Delete element at index 'pos'; returns NEW array (length -1).
 */
static int[] deleteByPosition(int[] theArray, int pos) {
    int[] temp = new int[theArray.length - 1];

    // Copy elements, skipping the deleted position
    for (int i = 0; i < temp.length; i++) {
        if (i >= pos) // If past delete index, shift left
            temp[i] = theArray[i + 1];
        else          // Otherwise copy directly
            temp[i] = theArray[i];
    }
    return temp;
}

// Print first dataLength elements on one line.
static void printArray(int[] theArray, int dataLength) {
    System.out.print("Array: ");
    for (int i = 0; i < dataLength; i++) // Print up to given length
        System.out.print(theArray[i] + " ");
    System.out.println();
}

public static void main(String[] args) {
    int[] SampleData = {9, 1, 8, 2, 7, 3, 6, 5, 3};

    System.out.println(" - - - 1. Insertion Sort  - - -");
    int[] sortedArray = sort(SampleData, true);

    System.out.println("\n - - - 2. Reverse Function  - - -");
    printArray(sortedArray, sortedArray.length);
    reverse(sortedArray);
    System.out.print("Array after reverse: ");
    printArray(sortedArray, sortedArray.length);

    System.out.println("\n - - - 3. Delete by Value  (Value 3) - - -");
    int[] arrayAfterValueDelete = deleteByValue(sortedArray, 3);
    System.out.print("Array after deleting first '3': ");
    printArray(arrayAfterValueDelete, arrayAfterValueDelete.length);

    System.out.println("\n - - - 4. Delete by Position  (Index 2) - - -");
    int[] finalArray = deleteByPosition(arrayAfterValueDelete, 2);
    System.out.print("Array after deleting element \nat index 2 (Value 7): ");
    printArray(finalArray, finalArray.length);

    System.out.println("\nbye\n");
}
```

1.3.2 *Array Warp Around*

Key Ideas

- Place data in an array.
  - If the data element is “full/occupied”, look to the right for the next empty element and wrap around to the beginning element if at the end.

Lecture Code

```
/*
 * DS132SU_WrapAround
 *
 * Programmer: Jim Goudy
 * Project: Wrap Around Array
 This shows how to wrap around in an around.
 Meaning, the program looks to see if an array
 element is open. If it is not, then look to
 the right. Continue looking to the right,
 till the next available element is open.

 *
 */


import java.util.Random;
import java.util.Scanner;

public class DS_WrapAround {

    static int ArrayLength = 8;
    static String[] myArray = new String[ArrayLength];
    static int maxVal = ArrayLength;

    static Scanner myScan = new Scanner(System.in);
 static Random RNG = new Random();

    static int myRNG() {
        // Generate a random number
        // within the number of array elements
        return RNG.nextInt(maxVal);
    }

    static void printArray() {
        // print the array

        for (int c = 0; c < myArray.length; c++) {
            if (myArray[c] == null) {
                System.out.print(” - |”);
            } else {
                System.out.print(myArray[c] + ” |”);
            }
        }
    }

    public static void main(String[] args) {

        String run = ”y”;
        int aIndex = -1;
        boolean check = true;
        int boxCntr = 0;

        String quit = ”y”;

        // assign values to the array - go to right if occupi
        // this loops continues till the array is full
        while (run.equals(”y”)) {
            aIndex = myRNG();
            System.out.print(”\nComputer chooses ” + aIndex +

            // fill the array element
            while (check) {
                if (myArray[aIndex] == null) {
                    //array index(box) is empty
                    myArray[aIndex] = ” X”;

                    // this variable keeps track of
                    // the total number of elments that
                    // are occupied/filled
                    boxCntr++;

                    // sets variable to exit
                    // the inner while loop
                    check = false;

                    // check if all the array elements are fi
                    if (boxCntr == myArray.length) {
                        run = ”n”;
                    }
                } else {
                    // array index (box) is not empty
                    aIndex++;
                    System.out.println(”move right ” + aIndex

                    // if the index is at the end of the arra
                    // wrap around to the first element 0
                    if (aIndex == myArray.length) {
                        aIndex = 0;
                        System.out.println(”move right ” + aI
                    }
                }
            }

            check = true;
            printArray();
        }

    }
}
/*
Note: Since the computer use a random generator to pick numbe
Example Output:
Computer chooses 5
 - | - | - | - | - | X | - | - |
Computer chooses 6
 - | - | - | - | - | X | X | - |
```

```
Computer chooses 7
 - | - | - | - | - | X | X | X |
Computer chooses 7
move right 8
move right 0
 X | - | - | - | - | X | X | X |
Computer chooses 6
move right 7
move right 8
move right 0
move right 1
 X | X | - | - | - | X | X | X |
Computer chooses 3
 X | X | - | X | - | X | X | X |
Computer chooses 0
move right 1
move right 2
 X | X | X | X | - | X | X | X |
Computer chooses 5
move right 6
move right 7
move right 8
move right 0
move right 1
move right 2
move right 3
move right 4
 X | X | X | X | X | X | X | X |
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
*/
```

End Of Topic

1.3.3 *Array Add and Delete Data*

Key Ideas

- In working with arrays, keeping data continuous is a good practice the majority of the time. If data in an element is deleted, the data in the elements to the right should be shifted left to remove the empty space.
- Also, there may be times when data has to be inserted into an array that has continuous data elements. In this case, data is shifted to the right to create a space where the new data can be inserted.

> **Tip**
> In many cases, arrays are usually set to have more elements than is needed. Therefore, it is important to create a variable that will alway track the number of items in the array.

Lecture Code

```
/*
 *
 * Project: Add Delete In an Array
 * Programmer: J Goudy

 */


public class DS_ArraysAddDelete {

    static String[] arrString;
    static int arrStrDataCount = 0;

    static void loadStringArray() {
    // this is a helper function to setup our example array

        String[] names = {"Adam", "Bobby", "Howard", "Mary", "Zuzu"};

        //load names
        for (int c = 0; c < names.length; c++) {
            arrString[c] = names[c];
        }

  //set our number of data items
        arrStrDataCount = names.length;

        // print number of items
        System.out.println("DataCount = " + arrStrDataCount);

        printArray(arrString, arrStrDataCount);

    }

    static void printArray(String[] theArray, int dataCount) {
    // This function prints the array.
    // Note that an array is being passed to it.
    // for spacing
    System.out.println();

    // iterate through the array and print the data
    for (int i = 0; i < dataCount; i++) {
        System.out.print(theArray[i] + " ");
    }

    System.out.println("\n - - - - - - - - - - - -\n");

}

static void insertStringByPos(int pos, String aName) {
    // - - - - - - - - - - Do some checks - - - - - - - -

    // check if position is withing array bounds
    if (pos > arrStrDataCount) {
        System.out.println("Error out array bounds");

        // exit the function
        return;
    }

    // check if the array is full
    if (arrStrDataCount >= arrString.length) {
        System.out.println("Array is full");
        return;
    }

    // - - - - - - - - - - - - - - Insert Code - - - - -
    // shift to right
    // note that the loop is starting at the end and
    // working backwards to the insert spot (pos)
    for (int i = arrStrDataCount; i > pos; i - -) {

        arrString[i] = arrString[i - 1];
    }

    // insert the new name
    arrString[pos] = aName;

    // increment our data Count
    arrStrDataCount++;

    // print the array to show data was inserted
    printArray(arrString, arrStrDataCount);

}

static void deleteByPos(int pos) {
    // check if there is contents
    if (arrStrDataCount <= 0) {
        System.out.println("Array is empty\n");
        return;
    }

    // check if the position to delete is out of bounds
```

```
        if (pos >= arrStrDataCount) {
            System.out.println("Pos is out of bounds\n");
            return;
        }

        // - - - - - - - - - - - -  Delete Code - - - - - - - - - - - - - - - - - - - - -
        // shift loop
        // note that the loop starts at the element location
        // that is being deleted
        for (int i = pos; i < arrStrDataCount; i++) {
            arrString[i] = arrString[i + 1];
        }

        // decrease the item count by 1
        arrStrDataCount - -;

        printArray(arrString, arrStrDataCount);

    }

    public static void main(String[] args) {

        // instantiate the data array
        arrString = new String[8];

        try {

            // setup the demo array
            loadStringArray();

            // insert "Bubba" in the third positon of the array
            insertStringByPos(2, "Bubba");

            // delete the data in the second
            // element/position of the array
            deleteByPos(1);

        } catch (Exception e) {
            System.out.println(e.getMessage());
        }

        System.out.println("\n\nbye\n");
    }
}
```

End Of Topic

1.4 Encapsulation vs Non-Encapsulation

This program is a Java demonstration designed to illustrate the core Object-Oriented Programming (OOP) concept of Encapsulation using the contrast between private and public access modifiers. It primarily shows how data hiding and controlled access are used to maintain an object's integrity.

1.4.1 *The Encapsulated Person Class (Good Practice)*

The Person class exemplifies proper encapsulation, also known as data hiding.

Private Data Fields: The class's data (firstName and lastName) are declared as private. This prevents any code outside the class from directly reading or modifying them, effectively protecting the data.

Public Accessors (Getters and Setters): The class provides public methods—getters (like getFirstName()) for safely reading the data, and setters (like setFirstName()) for safely writing the data.

Controlled Integrity: The key benefit is that the setFirstName() method includes validation logic. If an external caller attempts to set a first name that is null or shorter than two characters, the setter detects the invalid input, prints an error message, and prevents the private field from being changed. This ensures the Person object remains in a valid, consistent state.

1.4.2 *The Non-Encapsulated Dog Class (Risky Practice)*

The Dog class demonstrates the pitfalls of breaking encapsulation by using public fields.

Public Data Fields: The class's data (dogName and dogBreed) are declared as public. Any external code can directly access and modify these variables without any mediation.

No Control or Validation: Because external code bypasses any methods to change the data, the Dog class has no opportunity to validate the new values. As demonstrated in the main method, an external caller can directly set the dogName to a potentially disastrous value like null. This makes the object's internal state vulnerable and potentially invalid, and the class cannot prevent it.

1.4.3 *The main Demonstration*

The main method creates instances of both classes to showcase the difference:

It attempts to set an invalid, short first name ("S") on the encapsulated Person object. The setter rejects the change, and the object's name remains the previous valid value ("Jane Doe").

It directly sets an invalid, null name on the non-encapsulated Dog object. The public field is directly overwritten, allowing the object to enter an inconsistent, invalid state ("null Mutt"), demonstrating the lack of control.

1.4.4 *Demo Code*

```
/*
 * Developer: James Goudy
 *
 * This file demonstrates the key Object -Oriented Programmin
 * concept of ENCAPSULATION using access modifiers (private v
 */
package classpublicprivateelements;

// The Person class demonstrates ENCAPSULATION (Data Hiding)
class Person {

    // 1. Data Hiding: The fields are private, protecting ther
    // direct external modification. This forces users to go
    // our controlled public interface.
    private String firstName;
    private String lastName;

    public Person() {
    }

    public Person(String firstName, String lastName) {
        // Use setters in the constructor to ensure validatio
        setFirstName(firstName);
        setLastName(lastName);
    }

    // Getters: Public methods to safely READ the private dat
    public String getFirstName() {
        return firstName;
    }

    // 2. Controlled Access (Setters): Public methods to safe
    // the private data. Setters give you the opportunity to
    // data and maintain the object's integrity. If the input
    // invalid, the field is not changed.
    public void setFirstName(String firstName) {
        // ENHANCEMENT: Simple validation logic
        if (firstName != null && firstName.length() >= 2) {
            this.firstName = firstName;
        } else {
            // Note: In real production code, you might throw
            // IllegalArgumentException.
            System.out.println("Error: First name '" + firstN
                               "' is invalid. Value ignored."
        }
    }

    public String getLastName() {
        return lastName;
    }
```

```
    public void setLastName(String lastName) {

        // The same checks as setFirstName could also be applied here.
        this.lastName = lastName;
    }

    public String allInfo() {
        return (this.firstName + " " + this.lastName);
    }
}

// The Dog class demonstrates the dangers of PUBLIC ACCESS
class Dog {

    // 1. Public Fields: The elements are PUBLIC - they can be accessed
    // and changed directly by any external code. This breaks
    // encapsulation.
    public String dogName;
    public String dogBreed;
}

    // THERE IS NO OPPORTUNITY TO VALIDATE! A user can set these fields
    // to any value (like null or an empty string), potentially making
    // the object's state INVALID without the class having any control.

    public Dog() {
    }

    public Dog(String dogName, String dogBreed) {
        // If we wanted validation here, we would have to duplicate the
        // logic in every constructor, instead of just using a Setter.
        this.dogName = dogName;
        this.dogBreed = dogBreed;
    }

    public String allInfo() {
        return (this.dogName + " " + this.dogBreed);
    }
}

public class ClassPublicPrivateElements {

    public static void main(String[] args) {

        // - - - Demonstration of Encapsulation (Person) - - -
        Person p1 = new Person();

        // Use the public interface (Setters) to change the private fields.
        p1.setFirstName("Jane");
        p1.setLastName("Doe");
        System.out.println("Person 1 (Initial Set): " + p1.allInfo());

        // Attempting to set an invalid value - The Setter PROTECTS the data!
        System.out.println("\nAttempting to set an INVALID first name ('S'):");
        p1.setFirstName("S");

        // The field remains unchanged because of the validation in the setter.
        System.out.println("Person 1 (After Invalid Set): " + p1.allInfo());

        // - - - Demonstration of Public Access (Dog) - - -
        Dog dog = new Dog();

        // Directly setting the public elements - NO validati
        dog.dogName = "Fido";
        dog.dogBreed = "Mutt";
        System.out.println("\nDog (Initial Set): " + dog.allI

        // Directly setting an invalid value - The public fie
        System.out.println("Directly setting an INVALID name

        // This is legal but potentially disastrous!
        dog.dogName = null;
        // The object's state is now inconsistent, and the Do
        // could not prevent it.
        System.out.println("Dog (After Invalid Set): " + dog.
```

1.5 Big O Notation

1.5.1 *Key Ideas*

- Big O Notation

> **Note**
>
> Big O notation is used in Computer Science to describe the performance or complexity of an algorithm. Big O specifically describes the worst-case scenario and can be used to describe the execution time required or the space used (e.g. in memory or on disk) by an algorithm. - Rob Bell

1.5.2 *Reading*

ON_Visualizing Big O Notation

1.5.3 *Videos*

https://youtu.be/v4cd1O4zkGw
https://youtu.be/Q_1M2JaijjQ

End Of Topic

1.5.4 *Visualizing Big O notation*

From '' mellowd co uk/ccie/?p=6122"

Author: Darren O'Connor

I'm currently learning as much computer science as I can on the side. I've come across Big O notation a few times already, and while I understand it, I'm much more of a visual guy.

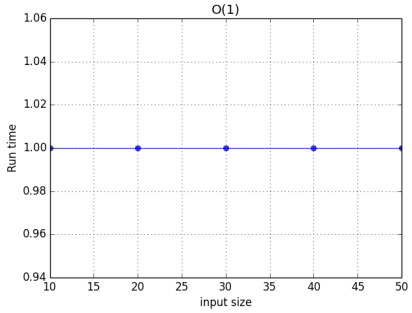
It's rather easy to use Python and matplotlib to graph out how a function's execution time grows as the size of the input grows. The important thing to note is not total execution time, but rather how the runtime of that function grows in relation to the input size. This can be plotted onto a graph which should give us a nice representation of Big O notations.

Note too that Big O notations always show the worst case. For this reason, I'll ensure to use values which the function will have to do the most work for.

O(1) O(1) means constant time. No matter what size the input, the runtime will always be the same. A simple example is finding the middle number in a list. I'll ensure that all code return the amount of time a command was run in the function. This may make the code look just a bit bloated, but for a good reason.To find the center of a list we simply divide the length of the list in two, and return that number. It does not matter if a list has 10 elements or 100 elements, the same amount of steps is performed:

```
def O1(input):
    count = 0
    result = input[len(input) / 2]
    count += 1
    return count
```

I have created 5 lists. The first is length 10, the second the length 20, and so on. I'll get the returned values and plot them.

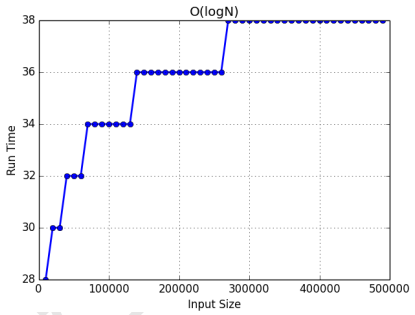


]O(1) plot

As can be seen, it doesn't matter the size of the input. It will always run at the same constant time.

O(logN) O(logN) increases as the input size goes up. However, it goes up as a log of the input size. This means that you can exponentially increase your input size, without linearly increasing the processing time to match.

```
def OlogN(input):
    def search(length, count):
        count += 1
        length /= 2
        if length == 1 or length == 0:
            return 1 + count
        else:
            return 1 + search(length, count)
    return 1 + search(len(input), 1)
```

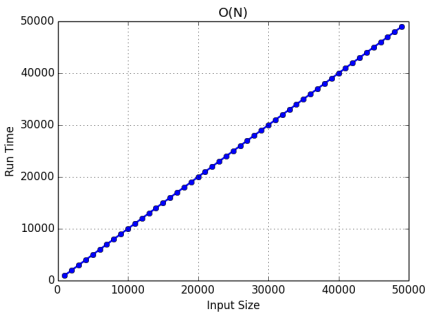


O(logN) plot

The run time is going up but look at the size of the inputs at the bottom. I start with 10,000 and move up to 500,000. The number of steps has increased, but not significantly.

O(N) O(N) is linear. This means that the run time is linearly matched to the input size. They should increase at exactly the same rate.

```
def ON(input, check):
    count = 0
    for number in input:
        count += 1
        if number == check:
            return 1 + count
```



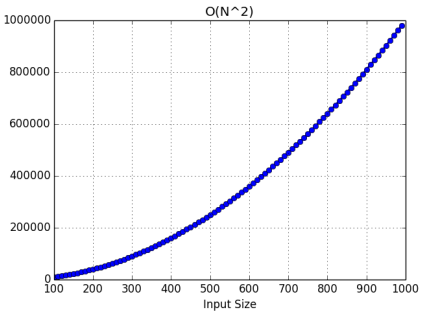
O(N) plot

There is a 1:1 correlation between input size and run time. As expected this produces a linear graph.

O(N2) O(N2)'s runtime will go up as a square of the input size. The runtime goes up faster than your input sizes, so processing time increases rapidly. This is usually when you iterate through multiple loops at the same time like so:

```
def ON2(input):
    count = 0
    for i in input:
```

```
        count += 1
        for j in input:
            count += 1
    return 1 + count
```

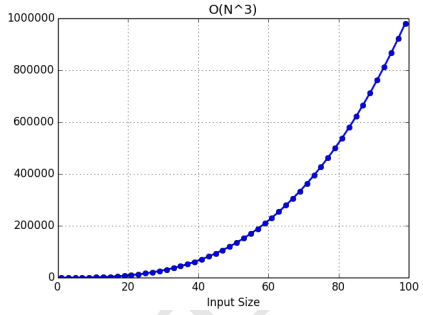


O(N2) plot

O(N3) O(N3)is merely O(N2) with another exponent. I wanted to show the difference by simply changing the exponent.

```
def ON3(input):
    count = 0
    for i in input:
        count += 1
        for j in input:
            count += 1
            for k in input:
                count +=1
    return 1 + count
```

O(N3) plot Graphs increase rapidly as the exponent increases.



Conclusions I've not shown every single type of algorithm, as I just wanted to show the ones I have the most experience with. It's nice to have a visual representation of these things as it really drills down just how fast your runtime can increase with larger inputs.

You can find my code used over here. https://github.com/mellowdrifter/Blog_Code/tree/master/Big_O

End Of Topic

1.6 Linked Lists

Definition
Linked List is a set of nodes where each node contains a data field(s) and a reference(link) to the next node in the list.

1.6.1 *Benefits of a linked list*

- Not limited to a specific data space amount.
- Easy to add and delete data

1.6.2 *Disadvantages of a linked list*

- The retrieval time of stored data is dependent on the size of the list and the position/node of the data in the list. O(n)

1.6.3 *Types of linked lists*

- Singly linked lists
- Doubly linked lists
- Doubly linked lists with links as sub class
- Circular linked lists

End of Topic

1.6.4 *Singly Linked List*

> **Note**
>
> **Singly linked list** is a type of linked list that is *unidirectional*. It can be traversed in only one direction from the head to the last node (tail). The last node always points to null

Ternary Operator

Ternary Operator is an instruction that consists of three parts. *Condition Part* - test if something is true or false *True Part* - what is returned if the condition is true *False Part* - what is returned if the condition is false

x = **(**Conditional Part**)** ** **?** True Part **:** False Part;

```
// Example
y = 3;
x = (y < 4) ? "y is less than four": "y is greater than four";
System.out.write(x);

// Output
// y is less than four
```

Lecture Code

Singly Linked List Code The *Link* and the controlling *Linked List* are written as two separate classes

```
/*
 * Singly Linked List
 *
 * singlylinkedlists_rev3
 * Programmer: James Goudy
 *
 */
package singlylinkedlists_rev3;

class Link
{

    // Data goes here
    public String city = "";
    public int population = 0;

    // link to next node
    public Link next;

    // constructor
    public Link(String city, int population)
    {
        this.city = city;
        this.population = population;
    }

    // display the link
    public void displayLink()
    {
        System.out.print("{" + city + ", " + population + "} ");
    }

}

class LinkedList
{

    // first is a reference / "address" of the first link
    private Link first;

    // constructor
    public LinkedList()
    {
        first = null;
    }

    //Check if the list is empty
    public boolean isEmpty()
    {
        return (first == null);
    }

    // insert at the front
    // of the list (front[left] - - to - -> back[right])
    public void insertFirst(String city, int population)
    {

        // create a new link
        Link newLink = new Link(city, population);

        // the new link is to the left or in front of first
        // the new link will make first in the second spot
        // so newLink.next has to point to the address first
        newLink.next = first;

        // now that the newLink.next is looking at the seond
        // or the next spot, first can have the address of th
        first = newLink;

    }

    //This function displays the linked list
    public void displayList()
    {
        System.out.print("\nList\n(first - -> last): ");

        //temp variable to hold first
        Link current = first;

        while (current != null)
        {
            current.displayLink();
```

```
            // move to the next  link
            current = current.next;
        }

        System.out.println();
    } //end of display list

    public boolean findCity(String city)
    {
        boolean found = false;

        Link current = first;
        // iterate through the loop
        while (current != null)
        {
            // check if current city matches search city
            // set found to true and break out of the loop
            if (current.city.equals(city))
            {
                found = true;
                break;
            }

            current = current.next;

        }

        return found;
    }

    // delete first
    public Link deleteFirst()
    {

        // temp variable to hold first address
        // temp is pointing to an address
        Link temp = first;

        if (!isEmpty())
        {
            // set variable to second spot
            first = first.next;
        }

        // return a link to object
        // in case calling program wants
        // to retreive the deleted data
        return temp;
    }

    public void deleteCity(String city)
    {
        // assumes the data does not have duplicates

        //need to check if the city was found
        boolean found = false;

        Link current = first;
        Link prev = first;
        Link temp;

        // check if city is in the first node
        if (first.city.equals(city))
        {
            temp = first;
            first = first.next;
            temp = null;
            System.out.println("City was deleted");
            return;
        }

        // start at the beginning of list
        while (current != null)
        {
            // check if the current city matches the search c
            if (current.city.equals(city))
            {
                found = true;

                // break out of the while statement
                break;
            }

            // set the prev variable
            prev = current;

            // move to the next node
            current = current.next;
        }

        // check if the while statment made it to
        // the end of the loop
        if (found == false)
        {
            System.out.println("\n*** City not found - Nothing
            return;
        }

        // delete process
        prev.next = current.next;

        current = null;
        System.out.println("\n*** City was successfully delete
    }

    public void deleteList()
    {
        while (first != null)
        {
            deleteFirst();
        }

    }
```

```
    // city is the location of where the new data
    // will be inserted (after)
    // newCity and newPop are the new city and new population
    public void insertAfter(String city, String newCity, int newPop)
    {
        boolean found = false;
        Link current = first;

        //create a new link
        Link NewLink = new Link(newCity, newPop);

        try
        {
            // iterate through loop
            while (current != null)
            {
                // break out of the loop if found
                // stops the loop at the found city
                // sets found to true
                if (current.city.equals(city))
                {
                    found = true;
                    break;
                }

                // move to next node
                current = current.next;
            }

            // insert process
            NewLink.next = current.next;
            current.next = NewLink;

        } catch (Exception e)
        {
            System.out.println("**Error**\n" + e.getMessage() + "\n***\n");
        }

    }

} // end of class

public class SinglyLinkedLists_Rev3
{

    public static void main(String[] args)
    {
        LinkedList theList = new LinkedList();


        theList.insertFirst("Kali", 32000);
        theList.insertFirst("Whitefish", 7700);
        theList.insertFirst("Polson", 20000);
        theList.insertFirst("Chicago", 13000000);
        theList.insertFirst("Convoy", 500);

        theList.displayList();

        System.out.println("Find Kali : " + theList.findCity("

        // ternary operator
        String output = (theList.findCity("Polson") == true)
                ? "City Found" : "City Not Found";
        System.out.println("Polson: " + output);

        output = (theList.findCity("Somers") == true)
                ? "City Found" : "City Not Found";
        System.out.println("Somers: " + output);

        theList.deleteFirst();
        theList.displayList();

        theList.deleteCity("Chicago");
        theList.displayList();

        theList.insertAfter("Polson", "New York", 10000000);
        theList.displayList();

        theList.deleteList();
        theList.displayList();
    }

}
```

Singly Linked List - Nested Link Class

```
/*
 * Single Link List
 *
 * SingleLinkList_nested_Rev3
 * Programmer: James Goudy
 *
 */
package singlylinkedlists_nested_rev3;

class LinkedList
{

    class Link
    {

        // Data goes here
        public String city = "";
        public int population = 0;

        // link to next node
        public Link next;

        // constructor
        public Link(String city, int population)
        {
            this.city = city;
            this.population = population;
        }

        // display the link
        public void displayLink()
```

```
        {
            System.out.print("{" + city + ", " + population + "}");
        }

    }

    // first is a reference / "address" of the first link
    private Link first;

    // constructor
    public LinkedList()
    {
        first = null;
    }

    //Check if the list is empty
    public boolean isEmpty()
    {
        return (first == null);
    }

    // insert at the front
    // of the list (front[left] - - to - -> back[right])
    public void insertFirst(String city, int population)
    {

        // create a new link
        Link newLink = new Link(city, population);

        // the new link is to the left or in front of first
        // the new link will make first in the second spot
        // so newLink.next has to point to the address first
        newLink.next = first;

        // now that the newLink.next is looking at the seond spot
        // or the next spot, first can have the address of the newLink
        first = newLink;

    }

    //This function displays the linked list
    public void displayList()
    {
        System.out.print("\nList\n(first - -> last): ");

        //temp variable to hold first
        Link current = first;

        while (current != null)
        {
            current.displayLink();

            // move to the next  link
            current = current.next;
        }

        System.out.println();
    } //end of display list
    public boolean findCity(String city)
    {
        boolean found = false;

        Link current = first;
        // iterate through the loop
        while (current != null)
        {
            // check if current city matches search city
            // set found to true and break out of the loop
            if (current.city.equals(city))
            {
                found = true;
                break;
            }

            current = current.next;

        }

        return found;
    }
    // delete first
    public Link deleteFirst()
    {

        // temp variable to hold first address
        // temp is pointing to an address
        Link temp = first;

        if (!isEmpty())
        {
            // set variable to second spot
            first = first.next;
        }

        // return a link to object
        // in case calling program wants
        // to retreive the deleted data
        return temp;
    }
    public void deleteCity(String city)
    {
        // assumes the data does not have duplicates

        //need to check if the city was found
        boolean found = false;

        Link current = first;
        Link prev = first;
        Link temp;

        // check if city is in the first node
        if (first.city.equals(city))
        {
```

```
            temp = first;
            first = first.next;
            temp = null;
            System.out.println("City was deleted");
            return;
        }

        // start at the beginning of list
        while (current != null)
        {
            // check if the current city matches the search city
            if (current.city.equals(city))
            {
                found = true;

                // break out of the while statement
                break;
            }

            // set the prev variable
            prev = current;

            // move to the next node
            current = current.next;
        }

        // check if the while statment made it to
        // the end of the loop
        if (found == false)
        {
            System.out.println("\n*** City not found - Nothing deleted");
            return;
        }

        // delete process
        prev.next = current.next;

        current = null;
        System.out.println("\n*** City was successfully deleted");
    }

    public void deleteList()
    {
        while (first != null)
        {
            deleteFirst();
        }

    }

    // city is the location of where the new data
    // will be inserted (after)
    // newCity and newPop are the new city and new population
    public void insertAfter(String city, String newCity, int newPop)
    {
        boolean found = false;
        Link current = first;

        //create a new link
        Link NewLink = new Link(newCity, newPop);

        try
        {
            // iterate through loop
            while (current != null)
            {
                // break out of the loop if found
                // stops the loop at the found city
                // sets found to true
                if (current.city.equals(city))
                {
                    found = true;
                    break;
                }

                // move to next node
                current = current.next;
            }

            // insert process
            NewLink.next = current.next;
            current.next = NewLink;

        } catch (Exception e)
        {
            System.out.println("**Error**\n" + e.getMessage()
        }
            }

    } // e

    public class Singlylinkedlists_nested_rev3
    {

        public static void main(String[] args)
        {

            LinkedList theList = new LinkedList();

            // how to create a Link from a nested class
            // note that the new link has to be created from the
            // outer link ('theList')
            LinkedList.Link aLink =  theList.new Link("Detroit",

            // display the created link
            System.out.println("Created Inner Link");
            aLink.displayLink();
            System.out.println("\n\n");

            theList.insertFirst("Kali", 32000);
            theList.insertFirst("Whitefish", 7700);
            theList.insertFirst("Polson", 20000);
```

```
        theList.insertFirst("Chicago", 13000000);
        theList.insertFirst("Convoy", 500);

        theList.displayList();

        System.out.println("Find Kali : " + theList.findCity("Kali"));

        // ternary operator
        String output = (theList.findCity("Polson") == true)
                ? "City Found" : "City Not Found";
        System.out.println("Polson: " + output);

        output = (theList.findCity("Somers") == true)
                ? "City Found" : "City Not Found";
        System.out.println("Somers: " + output);

        theList.deleteFirst();
        theList.displayList();

        theList.deleteCity("Chicago");
        theList.displayList();

        theList.insertAfter("Polson", "New York", 10000000);
        theList.displayList();

        theList.deleteList();
        theList.displayList();

    }

}
```

End Of Topic

1.6.5 *Doubly Linked List*

Key Ideas

- The doubly linked list allows the list to be traversed in both directions; forwards and backwards

A **doubly linked list** is a type of linked list in which each node contains three parts:

1. **Data**: The value stored in the node.
2. **Pointer to the next node**: A reference to the next node in the list.
3. **Pointer to the previous node**: A reference to the previous node in the list.

Structure:

- In a doubly linked list, each node is connected to both its next and previous nodes, creating a two-way linkage.
- The first node (head) has its previous pointer set to `null`, and the last node (tail) has its next pointer set to `null`.

Benefits of a Doubly Linked List:

- **Bidirectional Traversal**: You can easily traverse the list in both forward and backward directions.
- **Efficient Deletion**: Insertion and deletion of nodes can be more efficient since you can easily access the previous node.

Lecture Code

```
/*
 * Programmer: James Goudy
 * Project: Doubly Linked List
 */

// Define a class representing each link/node in the doubly linked list
class Link {

    // Pointers for the first and last nodes in the list
    Link first = null;
    Link last = null;

    // Data stored in the node
    String city = null;

    // Pointers for the next and previous nodes
    Link next = null;
    Link prev = null;

    // Constructor to initialize a node with the city name
    Link(String city) {
        this.city = city;
        this.next = null;
        this.prev = null;
    }

    // Display the data of the current node
    public void displayNode() {
        System.out.print(city + " ");
    }
} // End of Link class

// Define a class representing the doubly linked list itself
class Doubly {

    // References to the first and last nodes in the list
    Link first = null;
    Link last = null;

    // Constructor to initialize an empty doubly linked list
    public Doubly() {
        first = null;
        last = null;
    }

    // Method to add a node at the beginning of the list
    public boolean addFirst(String city) {
        Link newLink = new Link(city);

        // If list is empty, set first and last to the new no
        if (first == null) {
            first = newLink;
            last = newLink;
        } else {
            // Otherwise, update pointers for the new node an
            newLink.next = first;
            first.prev = newLink;
            first = newLink;
        }

        return true;
    }

    // Method to add a node at the end of the list
    public boolean addLast(String city) {
        Link newLink = new Link(city);

        // If list is empty, set first and last to the new no
        if (first == null) {
            first = newLink;
            last = newLink;
        } else {
            // Otherwise, update pointers for the new node an
            newLink.prev = last;
            last.next = newLink;
            last = newLink;
        }

        return true;
    }
```

```
    // Method to find a city in the list
    public boolean findCity(String citySearch) {

        // If list is empty, return false
        if (first == null) {
            return false;
        } else {
            Link current = first;

            // Traverse the list to search for the city
            while (current != null) {
                if (current.city.equals(citySearch)) {
                    return true; // City found
                }
                current = current.next;
            }

            return false; // City not found
        }
    }

    // Method to insert a new node after a given city
    public boolean insertAfter(String citySearch, String insertCity) {
        Link newLink = new Link(insertCity);

        // If list is empty, add the new node as the only node
        if (first == null) {
            first = newLink;
            last = newLink;
        } else {
            Link current = first;

            // Traverse the list to find the specified city
            while (current != null) {
                if (current.city.equals(citySearch)) {

                    // If the city is the last node, update last
                    if (current.next == null) {
                        current.next = newLink;
                        newLink.prev = current;
                        last = newLink;
                    } else {
                        // Update pointers for inserting in the middle
                        newLink.next = current.next;
                        newLink.prev = current;
                        current.next.prev = newLink;
                        current.next = newLink;
                    }

                    return true; // City inserted
                }
                current = current.next;
            }
        }

        return false; // City not found
    }

    // Method to insert a new node before a given city
    public boolean insertBefore(String citySearch, String insertCity) {
        Link newLink = new Link(insertCity);

        // If list is empty, add the new node as the only node
        if (first == null) {
            first = newLink;
            last = newLink;
        } else {
            Link current = first;

            // Traverse the list to find the specified city
            while (current != null) {
                if (current.city.equals(citySearch)) {

                    // If the city is the first node, update first
                    if (current.prev == null) {
                        current.prev = newLink;
                        newLink.next = current;
                        first = newLink;
                    } else {
                        // Update pointers for inserting in the middle
                        newLink.next = current;
                        newLink.prev = current.prev;
                        current.prev.next = newLink;
                        current.prev = newLink;
                    }

                    return true; // City inserted
                }
                current = current.next;
            }
        }

        return false; // City not found
    }

    // Method to delete a node with a specified city
    public boolean deleteCity(String citySearch) {

        // If list is empty, return false
        if (first == null) {
            return false;
        } else {
            Link current = first;

            // Traverse the list to find the specified city
            while (current != null) {
                if (current.city.equals(citySearch)) {

                    // If the city is the first node
                    if (current.prev == null) {
                        current.next.prev = null;
                        first = current.next;
                        current = null;
                        return true; // City deleted
                    } else if (current.next == null) {
                        // If the city is the last node
```

```
                    current.prev.next = null;
                    last = current.prev;
                    current = null;
                    return true; // City deleted
                } else {
                    // If the city is in the middle
                    current.prev.next = current.next;
                    current.next.prev = current.prev;
                    current = null;
                    return true; // City deleted
                }
            }
            current = current.next;
        }

        return false; // City not found
    }
}

// Main class to demonstrate doubly linked list functionality
public class DS_DoublyLinkedList {

    // Create an instance of the Doubly linked list
    static Doubly dl = new Doubly();

    // Method to search for a city
    public static void citySearch(String searchCity) {
        if (dl.findCity(searchCity)) {
            System.out.println("\n" + searchCity + " is in the list");
        } else {
            System.out.println("\n" + searchCity + " not found");
        }
    }

    // Method to delete a city
    public static void deleteCity(String searchCity) {
        if (dl.deleteCity(searchCity)) {
            System.out.println(searchCity + " was deleted");
        } else {
            System.out.println(searchCity + " was NOT deleted");
        }
    }

    // Main method to execute the program
    public static void main(String[] args) {
        String searchCity = "";
        String insertCity = "";

        // Insert data at the front of the list
        dl.addFirst("Kali");
        dl.addFirst("Polson");
        dl.addFirst("Missoula");
        dl.addFirst("Whitefish");

        // Insert data at the end of the list
        dl.addLast("Chicago");
        dl.addLast("Denver");
        dl.addLast("Sandiego");

        dl.displayList(); // Display the list
        System.out.println("\n - - - - - Find Examples - - - -

        searchCity = "Chicago";
        citySearch(searchCity);

        searchCity = "Bozeman";
        citySearch(searchCity);

        System.out.println("\n - - - - - Delete Examples - - -

        searchCity = "Polson";
        deleteCity(searchCity);

        searchCity = "Bozeman";
        deleteCity(searchCity);

        searchCity = "Sandiego";
        deleteCity(searchCity);

        searchCity = "Whitefish";
        deleteCity(searchCity);

        dl.displayList(); // Display the list

        System.out.println("\n - - - - - Insert Examples - - -

        searchCity = "Missoula";
        insertCity = "Dayton";
        dl.insertAfter(searchCity, insertCity);

        searchCity = "Denver";
        insertCity = "Boulder";
        dl.insertAfter(searchCity, insertCity);

        dl.displayList(); // Display the list

        searchCity = "Chicago";
        insertCity = "Springfield";
        dl.insertBefore(searchCity, insertCity);

        searchCity = "Missoula";
```

```
        insertCity = "Libby";
        dl.insertBefore(searchCity, insertCity);

        dl.displayList(); // Display the list

        System.out.println("\nbye");
    }
}

/*
 * OUTPUT  *
 * Whitefish Missoula Polson Kali Chicago Denver Sandiego  *
 * - - - - - Find Examples - - - - - -
 *
 *
 * Chicago is in list
 *
 * Bozeman not found
 *
 * - - - - - Delete Examples - - - - - -
 *
 * Polson was deleted
 * Bozeman was NOT deleted
 * Sandiego was deleted
 * Whitefish was deleted
 *
 * Missoula Kali Chicago Denver  *
 * - - - - - Insert Examples - - - - - -
 *
 *
 * Missoula Dayton Kali Chicago Denver Boulder  *
 * Libby Missoula Dayton Kali Springfield Chicago Denver Boulder  *
 * bye
 *
 */
```

End Of Topic

1.6.6 *Doubly Linked List - Links as Sub Class*

Key Ideas

- The doubly linked list allows the list to be traversed in both directions; forwards and backwards

> **Note**
> Not all languages support subclasses.

Lecture Code

```
/*
 * Programmer: James Goudy
 * Project: Doubly Linked List written
 * with  Links / Nodes as SubClass
 */

class Doubly {

    Link first = null;
    Link last = null;

    // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    // sub class - Link / Nodes
    // NOTE: this can be a separate class as well

    class Link {

        Link first = null;
        Link last = null;

        // data
        String city = null;

        // link navigation
        Link next = null;
        Link prev = null;

        // constructor
        Link(String city) {
            this.city = city;
            this.next = null;
            this.prev = null;
        }

        public void displayNode() {
            System.out.print(city + " ");
        }
    } // end of link
    // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

    // constructor
    public Doubly() {

        first = null;
        last = null;
    }

    // add link at the beginning of the list
    public boolean addFirst(String city) {
        Link newLink = new Link(city);

        if (first == null) {
            // if list is empty
            first = newLink;
            last = newLink;
        } else {
            newLink.next = first;
            first.prev = newLink;
            first = newLink;
        }

        return true;
    }

    // add link to the end of the list
    public boolean addLast(String city) {

        Link newLink = new Link(city);

        if (first == null) {
            // if list is empty
            first = newLink;
            last = newLink;
        } else {
            newLink.prev = last;
            last.next = newLink;
            last = newLink;
        }

        return true;
    }

    public boolean findCity(String citySearch) {

        if (first == null) {

            // if list is empty
            return false;
        } else {
            Link current = first;

            while (current != null) {
                if (current.city.equals(citySearch)) {
                    return true;
                }
                current = current.next;
            }

            return false;
        }
    }

    public boolean insertAfter(String citySearch, String inse
```

```
        Link newLink = new Link(insertCity);

        if (first == null) {

            // list is empty - add the link
            first = newLink;
            last = newLink;

            // NOTE: there is an option not to insert
            // a link then the code above would be replaced
            // with return false

        } else {
            Link current = first;

            while (current != null) {
                if (current.city.equals(citySearch)) {

                    // check if last link
                    if (current.next == null) {
                        // check if last link
                        current.next = newLink;
                        newLink.prev = current;

                        last = newLink;

                    } else {
                        newLink.next = current.next;
                        newLink.prev = current;

                        current.next.prev = newLink;
                        current.next = newLink;
                    }

                    return true;
                }
                current = current.next;

            }
        }

        return false;

    }

    public boolean insertBefore(String citySearch, String insertCity) {

        Link newLink = new Link(insertCity);

        if (first == null) {

            // list is empty - add the link
            first = newLink;
            last = newLink;

            // NOTE: there is an option not to insert
            // a link then the code above would be replaced
            // with return false

        } else {
            Link current = first;

            while (current != null) {
                if (current.city.equals(citySearch)) {

                    // check for first link
                    if (current.prev == null) {
                        // check if last link
                        current.prev = newLink;
                        newLink.next = current;

                        first = newLink;

                    } else {
                        newLink.next = current;
                        newLink.prev = current.prev;

                        current.prev.next = newLink;
                        current.prev = newLink;
                    }

                    return true;
                }
                current = current.next;

            }
        }

        return false;
    }

    public boolean deleteCity(String citySearch) {

        if (first == null) {
            return false;
        } else {
            Link current = first;

            while (current != null) {
                if (current.city.equals(citySearch)) {

                    if (current.prev == null) {
                        // first node
                        current.next.prev = null;
                        first = current.next;
                        current = null;
                        return true;
                    } else if (current.next == null) {
                        // last node
                        current.prev.next = null;
                        last = current;
                        current = null;
                        return true;
                    } else {
                        // a center node
                        current.prev.next = current.next;
```

```
                        current.next.prev = current.prev;
                        current = null;

                        return true;
                    }
                }
                current = current.next;
            }

            return false;
        }
    } //end of function

    public void displayList() {
        Link current = first;

        System.out.println("");
        while (current != null) {
            current.displayNode();
            current = current.next;
        }
        System.out.println("");
    }

    public void displayListBackwards() {
        Link current = last;

        System.out.println("");
        while (current != null) {
            current.displayNode();
            current = current.prev;
        }
        System.out.println("");
    }

}

public class J2_SubClass {

    static Doubly dl = new Doubly();

    public static void citySearch(String searchCity) {

        // note that findCity returns a boolean
        // so it can be used in an "if" statement
        if (dl.findCity(searchCity)) {
            System.out.println("\n" + searchCity + " is in list");
        } else {
            System.out.println("\n" + searchCity + " not found");
        }

    }

    public static void deleteCity(String searchCity) {

        // note that findCity returns a boolean
        // so it can be used in an "if" statement
        if (dl.deleteCity(searchCity)) {
            System.out.println(searchCity + " was deleted");
        } else {
            System.out.println(searchCity + " was NOT deleted
        }
    }

    public static void main(String[] args) {

        String searchCity = "";
        String insertCity = "";

        // insert data at front of list
        dl.addFirst("Kali");
        dl.addFirst("Polson");
        dl.addFirst("Missoula");
        dl.addFirst("Whitefish");

        // insert data at end of list
        dl.addLast("Chicago");
        dl.addLast("Denver");
        dl.addLast("Sandiego");

        dl.displayList();

        System.out.println("\n - - - - - Find Examples - - -

        searchCity = "Chicago";
        citySearch(searchCity);

        searchCity = "Bozeman";
        citySearch(searchCity);

        System.out.println("\n - - - - - Delete Examples - -

        searchCity = "Polson";
        deleteCity(searchCity);

        searchCity = "Bozeman";
        deleteCity(searchCity);

        searchCity = "Sandiego";
        deleteCity(searchCity);

        searchCity = "Whitefish";
        deleteCity(searchCity);
        dl.displayList();
        System.out.println("\n - - - - - Insert Examples - -

        searchCity = "Missoula";
        insertCity = "Dayton";
        dl.insertAfter(searchCity, insertCity);

        searchCity = "Denver";
        insertCity = "Boulder";
        dl.insertAfter(searchCity, insertCity);
```

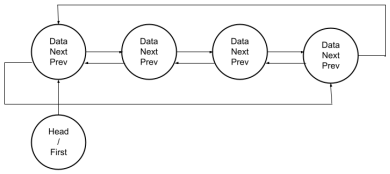
```
        dl.displayList();

        searchCity = "Chicago";
        insertCity = "Springfield";
        dl.insertBefore(searchCity, insertCity);

        searchCity = "Missoula";
        insertCity = "Libby";
        dl.insertBefore(searchCity, insertCity);

        dl.displayList();

        System.out.println("\nbye");
    }
}

/*
 OUTPUT

 Whitefish Missoula Polson Kali Chicago Denver Sandiego

 - - - - - Find Examples - - - - - -


Chicago is in list

Bozeman not found

 - - - - - Delete Examples - - - - - -

Polson was deleted
Bozeman was NOT deleted
Sandiego was deleted
Whitefish was deleted

Missoula Kali Chicago Denver

 - - - - - Insert Examples - - - - - -


Missoula Dayton Kali Chicago Denver Boulder

Libby Missoula Dayton Kali Springfield Chicago Denver Boulder

bye

 */
```

End Of Topic

1.6.7 *Circular Linked List - Links as Sub Class*

Key Ideas

- The doubly linked list allows the list to be traversed in both directions; forwards and backwards
- In a circular doubly-linked list the last node (next) points to the first. The first node (previous) points to the last node.
- The doubly linked list allows the list to be traversed in both directions; forwards and backwards



> **Note**
> Not all languages support subclasses.

Lecture Code

```
package com.mycompany.linkedlistcircular;

/*
 * Programmer: James Goudy
 * Project Circular LinkedList
 *
 * NOTE: last link is referenced as first.prev
 */
class CircularLinkedList {

    Link first = new Link("");

    // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    // sub class - Link / Nodes
    // NOTE: this can be a separate class as well
    class Link {

        Link first = null;

        // data
        String city = null;

        // link navigation
        Link next = null;
        Link prev = null;

        // constructor
        Link(String city) {
            this.city = city;
            this.next = null;
            this.prev = null;
        }

        public void displayNode() {
            System.out.print(city + " ");
        }
    } // end of link
    // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

    // constructor
    public CircularLinkedList() {

        first = null;
    }

    // add link at the beginning of the list
    public boolean addFirst(String city) {

        Link newLink = new Link(city);

        if (first == null) {
            // empty list
            newLink.next = newLink;
            newLink.prev = newLink;

            first = newLink;

        } else {

            // connect the newLink references
            newLink.next = first;

            newLink.prev = first.prev;

            first.prev = newLink;

            // move first to the new link
            first = newLink;

            // point the last link to the new first
            first.prev.next = first;

        }

        return true;
    }

    // add link to the end of the list
    public boolean addLast(String city) {

        Link newLink = new Link(city);

        if (first == null) {
            //list is empty
            first = newLink;
        } else {
            // set new link references
            newLink.next = first;

            newLink.prev = first.prev;
```

```
            // last link is (first.prev)
            first.prev.next = newLink;

            first.prev = newLink;

        }

        return true;
    }

    public boolean findCity(String citySearch) {

        if (first == null) {

            // if list is empty
            return false;
        } else {
            Link current = first;

            do {
                if (current.city.equals(citySearch)) {
                    return true;
                }
                current = current.next;

            } while (current != first);

            return false;
        }
    }

    public boolean insertAfter(String citySearch, String insertCity) {

        Link newLink = new Link(insertCity);

        if (first == null) {

            // list is empty - add the link
            first = newLink;

            // NOTE: there is an option not to insert
            // a link then the code above would be replaced
            // with return false
        } else {
            Link current = first;

            while (current != null) {
                if (current.city.equals(citySearch)) {

                    // check if last link
                    if (current.next == first.prev) {
                        // check if last link
                        current.next = newLink;
                        newLink.prev = current;

                        first.prev = newLink;
                    } else {
                        newLink.next = current.next;
                        newLink.prev = current;

                        current.next.prev = newLink;
                        current.next = newLink;
                    }

                    return true;
                }
                current = current.next;

            }
        }

        return false;

    }

    public boolean insertBefore(String citySearch, String ins

        Link newLink = new Link(insertCity);

        if (first == null) {

            // list is empty - add the link
            first = newLink;

            // NOTE: there is an option not to insert
            // a link then the code above would be replaced
            // with return false
        } else {
            Link current = first;

            while (current != null) {
                if (current.city.equals(citySearch)) {

                    // check for first link
                    if (current.prev == null) {
                        // check if last link
                        current.prev = newLink;
                        newLink.next = current;

                        first = newLink;

                    } else {
                        newLink.next = current;
                        newLink.prev = current.prev;

                        current.prev.next = newLink;
                        current.prev = newLink;
                    }

                    return true;
                }
                current = current.next;

            }
```

```
        }

        return false;
    }

    public boolean deleteCity(String citySearch) {

        if (first == null) {
            return false;
        } else {
            Link current = first;

            do {
                if (current.city.equals(citySearch)) {

                    if (current.prev == null) {
                        // first node
                        current.next.prev = null;
                        first = current.next;
                        current = null;
                        return true;
                    } else if (current.next == null) {
                        // last node
                        current.prev.next = null;
                        first.prev = current;
                        current = null;
                        return true;
                    } else {
                        // a center node
                        current.prev.next = current.next;
                        current.next.prev = current.prev;
                        current = null;

                        return true;
                    }
                }

                current = current.next;
            } while (current != first);

            return false;
        }
    } //end of function

    public void displayList() {
        Link current = first;

        System.out.println("\n** Display List Forward To Back **");

        do {
            current.displayNode();
            current = current.next;
            if (current == first) {
                System.out.println("\n - - - - - - - - - - -\n");
                return;
            }

        } while (current.next != null);

    } // end of method

    public void displayList(String startCity) {

        Link current = first;
        Link start = null;

        System.out.println("\n ** Display List Forward To Bac
                + " starting at " + startCity + "**");

        // find the city in the list
        do {
            if (current.city.equals(startCity)) {
                break;
            }

            current = current.next;

            if (current == first) {
                System.out.println("City Not Found");
                return;
            }
        } while (current.next != null);

        System.out.println("");

        start = current;

        do {
            current.displayNode();
            current = current.next;
            if (current == start) {
                System.out.println("\n - - - - - - - - - - -\
                return;
            }

        } while (current.next != null);

    } // end of method

    public void displayListReverse() {

        Link current = first.prev;

        System.out.println("\n** Display List in Reverse **\n
        do {
            current.displayNode();
            current = current.prev;
            if (current == first.prev) {
                System.out.println("\n - - - - - - - - - - -\
                return;
            }

        } while (current.prev != null);
    } // end of method
```

```
    public void displayListReverse(String startCity) {

        Link current = first;
        Link start = null;

        System.out.println("\n ** Display list in reverse"
                + " starting at " + startCity + "**");

        // find the city in the list
        do {
            if (current.city.equals(startCity)) {
                break;
            }

            current = current.next;

            if (current == first) {
                System.out.println("City Not Found");
                return;
            }

        } while (current.next != null);

        System.out.println("");

        start = current;

        do {
            current.displayNode();
            current = current.prev;
            if (current == start) {
                System.out.println("\n - - - - - - - - - - \n");
                return;
            }

        } while (current.prev != null);
    } // end of method

} // end of class

public class DS_LinkedListCircular {

    static CircularLinkedList dl = new CircularLinkedList();

    public static void citySearch(String searchCity) {

        // note that findCity returns a boolean
        // so it can be used in an "if" statement
        if (dl.findCity(searchCity)) {
            System.out.println("\n" + searchCity + " is in list");
        } else {
            System.out.println("\n" + searchCity + " not found");
        }

    }

    public static void deleteCity(String searchCity) {

        // note that findCity returns a boolean
        // so it can be used in an "if" statement
        if (dl.deleteCity(searchCity)) {
            System.out.println(searchCity + " was deleted");
        } else {
            System.out.println(searchCity + " was NOT deleted
        }

    }

    public static void main(String[] args) {

        String searchCity = "";
        String insertCity = "";

        // insert data at front of list
        dl.addFirst("Kali");
        dl.addFirst("Polson");
        dl.addFirst("Missoula");
        dl.addFirst("Whitefish");
        dl.addFirst("Plains");

        // insert data at end of list
        dl.addLast("Chicago");
        dl.addLast("Denver");
        dl.addLast("Sandiego");

        dl.displayList();
        dl.displayListReverse();

        dl.displayList("Missoula");
        dl.displayList("Wolfcreek");

        dl.displayListReverse();
        dl.displayListReverse("Missoula");

        System.out.println("\n - - - - - Find Examples - - -

        searchCity = "Missoula";
        citySearch(searchCity);

        searchCity = "Bozeman";
        citySearch(searchCity);

        System.out.println("\n - - - - - Delete Examples - -


        searchCity = "Polson";
        deleteCity(searchCity);

        searchCity = "Bozeman";
        deleteCity(searchCity);

        searchCity = "Sandiego";
        deleteCity(searchCity);

        searchCity = "Whitefish";
        deleteCity(searchCity);
```

```
        dl.displayList();

        System.out.println("\n - - - - - Insert After Examples - - - - - - - - - \n");

        searchCity = "Missoula";
        insertCity = "Dayton";
        dl.insertAfter(searchCity, insertCity);

        searchCity = "Denver";
        insertCity = "Boulder";
        dl.insertAfter(searchCity, insertCity);

        dl.displayList();

        System.out.println("\n - - - - - Insert After Examples - - - - - - - - - \n");

        searchCity = "Chicago";
        insertCity = "Springfield";
        dl.insertBefore(searchCity, insertCity);

        searchCity = "Missoula";
        insertCity = "Libby";
        dl.insertBefore(searchCity, insertCity);

        dl.displayList();
        System.out.println("\nbye");
    }
}
/*
 * output
 *
 ** Display List Forward To Back **
 * Plains Whitefish Missoula Polson Kali Chicago Denver Sandiego
 * - - - - - - - - - - - -
 *
 ** Display List in Reverse **
 *
 * Sandiego Denver Chicago Kali Polson Missoula Whitefish Plains
 * - - - - - - - - - - - -
 *
 ** Display List Forward To Back starting at Missoula**
 *
 * Missoula Polson Kali Chicago Denver Sandiego Plains Whitefish
 * - - - - - - - - - - - -
 *
 ** Display List Forward To Back starting at Wolfcreek**
 * City Not Found
 *
 ** Display List in Reverse **
 *
 * Sandiego Denver Chicago Kali Polson Missoula Whitefish Plains
 * - - - - - - - - - - - -
 *
 *
 ** Display list in reverse starting at Missoula**
 *
 * Missoula Whitefish Plains Sandiego Denver Chicago Kali Pols
 *
 *
 * - - - - - Find Examples - - - - - -
 *
 *
 * Missoula is in list
 *
 * Bozeman not found
 *
 * - - - - - Delete Examples - - - - - -
 *
 * Dayton was deleted
 * Bozeman was NOT deleted
 * Sandiego was deleted
 * Whitefish was deleted
 *
 ** Display List Forward To Back **
 * Plains Missoula Kali Chicago Denver
 * - - - - - - - - - - -
 *
 *
 * - - - - - Insert After Examples - - - - - -
 *
 *
 ** Display List Forward To Back **
 * Plains Missoula Dayton Kali Chicago Denver Boulder
 * - - - - - - - - - - - -
 *
 *
 * - - - - - Insert After Examples - - - - - -
 *
 *
 ** Display List Forward To Back **
 * Plains Libby Missoula Dayton Kali Springfield Chicago Denve
 * - - - - - - - - - - -
 *
 * bye
 *
```

End Of Topic

1.6.8 *Reading Datafile Into Linked List*

Lecture Code

```
/*
Programmer: James Goudy
Project: Reading a csv file into a linked list
 */



import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.util.logging.Level;
import java.util.logging.Logger;

class Link {

    Link first = null;
    Link last = null;

    // data
    String id = null;
    String firstname;
    String lastname;
    String pet;


    // link navigation
    Link next = null;
    Link prev = null;

    // constructor
    public Link(String id, String firstname, String lastname, String pet) {
        this.id = id;
        this.firstname = firstname;
        this.lastname = lastname;
        this.pet = pet;
    }


    public void displayNode() {
        System.out.println("{"+ firstname + " " + lastname + " "
                           +pet+ " " + id +" } " );
    }
} // end of link



class Doubly {

    Link first = null;
    Link last = null;

    // constructor
    public Doubly() {

        first = null;
        last = null;
    }

// add link at the beginning of the list
public boolean addFirst(String id,String firstname,
        String lastname, String pet) {
    Link newLink = new Link( id,firstname,  lastname,  pet

    if (first == null) {
        // if list is empty
        first = newLink;
        last = newLink;
    } else {
        newLink.next = first;
        first.prev = newLink;
        first = newLink;
    }

    return true;
}

// add link to the end of the list
public boolean addLast(String id,String firstname,
        String lastname, String pet) {
    Link newLink = new Link( id,firstname,  lastname,  pet

    if (first == null) {
        // if list is empty
        first = newLink;
        last = newLink;
    } else {
        newLink.prev = last;
        last.next = newLink;
        last = newLink;
    }

    return true;
}

public boolean findId(String searchID) {

    if (first == null) {
        // if list is empty
        return false;
    } else {
        Link current = first;

        while (current != null) {
            if (current.id.equals(searchID)) {
                return true;
            }
            current = current.next;
        }

        return false;
    }
}
```

```
public boolean insertAfter(String searchId, String id,String firstname,
        String lastname, String pet) {
    Link newLink = new Link( id,firstname,  lastname,  pet);

    if (first == null) {

        // list is empty - add the link
        first = newLink;
        last = newLink;

        // NOTE: there is an option not to insert
        // a link then the code above would be replaced
        // with return false
    } else {
        Link current = first;

        while (current != null) {
            if (current.id.equals(searchId)) {

                // check if last link
                if (current.next == null) {
                    // check if last link
                    current.next = newLink;
                    newLink.prev = current;

                    last = newLink;

                } else {
                    newLink.next = current.next;
                    newLink.prev = current;

                    current.next.prev = newLink;
                    current.next = newLink;
                }

                return true;
            }
            current = current.next;

        }
    }

    return false;

}

public boolean insertBefore(String searchId, String id,String firstname,
        String lastname, String pet) {

    Link newLink = new Link( id,firstname,  lastname,  pet);

    if (first == null) {

        // list is empty - add the link
        first = newLink;
        last = newLink;

        // NOTE: there is an option not to insert
        // a link then the code above would be replaced
        // with return false
    } else {
        Link current = first;

        while (current != null) {
            if (current.id.equals(searchId)) {

                // check for first link
                if (current.prev == null) {
                    // check if last link
                    current.prev = newLink;
                    newLink.next = current;

                    first = newLink;

                } else {
                    newLink.next = current;
                    newLink.prev = current.prev;

                    current.prev.next = newLink;
                    current.prev = newLink;
                }

                return true;
            }
            current = current.next;

        }
    }

    return false;
}

public boolean deleteId(String searchID) {

    if (first == null) {
        return false;
    } else {
        Link current = first;

        while (current != null) {
            if (current.id.equals(searchID)) {

                if (current.prev == null) {
                    // first node
                    current.next.prev = null;
                    first = current.next;
                    current = null;
                    return true;
                } else if (current.next == null) {
                    // last node
                    current.prev.next = null;
                    last = current;
                    current = null;
                    return true;
                } else {
```

```
                        // a center node                {
                        current.prev.next = current.next;
                        current.next.prev = current.prev;
                        current = null;

                        return true;
                    }
                }
                current = current.next;
            }

            return false;
        }
    } //end of function

    public void displayList() {
        Link current = first;

        System.out.println("");
        while (current != null) {
            current.displayNode();
            current = current.next;
        }
        System.out.println("");

    }
}

public class DS_ReadDataIntoLinkedList {

    static Doubly dl = new Doubly();

    public static void idSearch(String searchId) {

        // note that findCity returns a boolean
        // so it can be used in an "if" statement
        if (dl.findId(searchId)) {
            System.out.println("\n" + searchId + " is in list");
        } else {
            System.out.println("\n" + searchId + " not found");
        }

    }

    public static void deleteID(String searchId) {

        // note that findCity returns a boolean
        // so it can be used in an "if" statement
        if (dl.deleteId(searchId)) {
            System.out.println(searchId + " was deleted");
        } else {
            System.out.println(searchId + " was NOT deleted");
        }

    }

    public static boolean addData(String filePath)
    {
        BufferedReader br;
        FileReader fr;
        String inputLine;

        try {
            // create buffered reader
            br = new BufferedReader(new FileReader(filePath))

            // skip the header line
            br.readLine();

            while((inputLine = br.readLine()) != null)
            {
                String[] inputArray = inputLine.split(",");
                dl.addLast(inputArray[0],
                           inputArray[1],
                           inputArray[2],
                           inputArray[3]);

            }

            br.close();

        } catch (FileNotFoundException ex) {
            System.out.println(ex.getMessage());
            return false;
        }catch (Exception e)
        {
            System.out.println(e.getMessage());
            return false;
        }

        return true;
    }
    public static void main(String[] args) {
        String searchID = "";
        String insertCity = "";

        String filePath = "c:\\z\\peoplePets.csv";

        addData(filePath);

        dl.displayList();

        System.out.println("\n - - - - - Find Examples - - -
        searchID = "10";
        idSearch(searchID);

        searchID = "AAA";
        idSearch(searchID);

        System.out.println("\n - - - - - Delete Examples - -
```

```
        searchID = "10";
        deleteID(searchID);

        searchID = "AA";
        deleteID(searchID);

        System.out.println("\n - - - - - Insert Example - - - - - ");

        searchID = "140";
        dl.insertAfter(searchID, "1440","Duke","Ellingham","Phoenix");

        searchID = "145";
        dl.insertBefore(searchID, "1455","John","Coleraine","Tyrant");

        dl.displayList();

        System.out.println("\nbye");
    }
}

/* peoplePet.csv

id,firstname,lastname,pet
1,Wini,Gwinnett,Crowned hawk -eagle
2,Birgit,Tabb,Silver gull
3,Alfonso,Moyle,Ring -tailed possum
4,Ode,Buckwell,Leopard
5,Francene,Zanazzi,Wambenger
6,Willie,Hakking,Butterfly
7,Carlie,Pizey,Northern fur seal
8,Cobby,Chittock,Great white pelican
9,Ingamar,Cardenoza,Puma
10,Brady,Vowles,Red meerkat
11,Nonna,Betser,Lesser mouse lemur
12,Antonina,Dovey,Insect
13,Erastus,Crackett,Cow
14,Lucien,Cardenosa,Yellow baboon
15,Leon,Storm,Blue catfish
16,Reinaldos,Welberry,Common grenadier
17,Cherice,Coleson,Shrike
18,Elyn,Antill,Echidna
19,Dicky,Guppie,Puna ibis
20,Erasmus,Pauncefort,White -necked stork
21,Stewart,Pettifer,Starling
22,Brand,Tytcomb,Raccoon
23,Edwina,Cosens,Radiated tortoise
24,Martelle,Barkus,Cormorant
25,Blithe,Prevett,Kafue flats lechwe
26,Abbie,Ferber,Common wombat
27,Antonin,Sayes,Small -clawed otter
28,Vaughan,Barzen,Hawk -eagle
29,Elwira,Braemer,Monkey
30,Sibelle,Vennings,Javanese cormorant
31,Mavra,Bulter,Lemur
32,Darrelle,Sanford,Beisa oryx
33,Rodney,Whapples,Gull
34,Horace,Gerwood,Tropical buckeye butterfly
35,Garvin,Pestell,Bleu
36,Lauralee,Crowdace,Dama wallaby
37,Farica,Juara,Penguin
38,Bucky,Taylo,Crab
39,Carlynne,Pleasaunce,Common genet
40,Dana,Percy,White -throated kingfisher
41,Esma,McKerley,Southern ground hornbill
42,Hube,Grills,Flamingo
43,Dmitri,Schmidt,Kangaroo
44,Hildy,Matfin,Gull
45,Lelia,Donaghy,American marten
46,Eric,Tydd,Bahama pintail
47,Carlin,Teysha,Swan -winged goose
48,Berke,Brotherwood,Little cormorant
49,Steve,Bride,Turtle
50,Christiane,Stoppe,Fox
51,Riley,Badgers,Swallow
52,Leonidas,Pughsley,Roseate cockatoo
53,Richard,Baudon,White -browed owl
54,Marline,Tousey,Indian mynah
55,Margarita,Breche,Phalarope
56,Gerek,Aspinwall,Great horned owl
57,Mabelle,Aronin,Grouse
58,Curtice,Provost,Indian mynah
59,Nikolaos,Cass,Desert tortoise
60,Edmund,Pogosian,Grenadier
61,Cindi,Vell,Catfish
62,Davis,Roberts,Three -banded plover
63,Clayborne,Jennrich,Roe deer
64,Malory,Iwanicki,Snowy owl
65,Toddy,Vannuchi,Arboral spiny rat
66,Terri,Dudson,Malachite kingfisher
67,Gabriel,Prine,Possum
68,Jeannine,Westwick,Gecko
69,Conney,Mattke,Stork
70,Eulalie,Wapplington,Flamingo
71,Darrick,Porcas,Rat
72,Matty,Marchment,Deer
73,Fedora,Semper,Nighthawk
74,Barnebas,Wychard,Flying fox
75,Edmund,Whitton,Squirrel
76,Oliver,Dragoe,Parrot
77,Carly,Royden,Lourie
78,Cairistiona,Brothwell,Chestnut weaver
79,Barnabas,Eastby,Lion
80,Clementina,McCoish,European stork
81,Rupert,Goosnell,Boa
82,Jerrylee,Keir,Lizard
83,Cariotta,Strettell,Little blue penguin
84,Onida,Wysome,White -rumped vulture
85,Reggis,Thursby,Cape wild cat
86,Sharleen,Yele,Gazer
87,Obadias,Rosedale,Vine snake
88,Jodie,Harmond,Boat -billed heron
89,Jonie,Goodricke,Brush -tailed phascogale
```

```
90,Carissa,Clorley,Elk
91,Jacki,Belhome,Northern elephant seal
92,Virgina,Jarrette,House sparrow
93,Hasheem,Cordeiro,Oystercatcher
94,Roseann,Hussy,Fairy penguin
95,Emmeline,Saurat,Bee -eater
96,Ashlin,Ollerhead,Grey lourie
97,Frannie,Sailes,Jungle kangaroo
98,Shelden,Imason,Woodpecker
99,Nicole,Cattle,Starling
100,Bennie,Selliman,Ant
101,Monroe,Sturzaker,Wild turkey
102,Lou,Drew -Clifton,Ferruginous hawk
103,Gerladina,Broadbere,Yellow -billed stork
104,Laney,Scartifield,Common zebra
105,Rolfe,Dressel,Cape Barren goose
106,Clarita,Zylbermann,Red -legged pademelon
107,Ev,Buckston,Kite
108,Vidovic,Lawson,Buffalo
109,Daloris,Grzesiak,Southern brown bandicoot
110,Pier,Sproson,Brocket
111,Edwina,Barlace,Giant girdled lizard
112,Elvin,Birchwood,Deer
113,Jedediah,Lazonby,Wallaby
114,Ambur,Lochead,Gila monster
115,Mirabella,Ferron,Possum
116,Dougy,Gianinotti,Openbill
117,Cher,Sivill,Dog
118,Durante,Wissby,Genet
119,Rora,Shord,Dove
120,Dame,Jennison,Red -billed toucan
121,Ira,Karolowski,Eurasian red squirrel
122,Juliet,Hobson,Greylag goose
123,Natale,Tattersdill,Starling
124,Caitlin,Leggett,Ring -necked pheasant
125,Hamnet,Danelut,South African hedgehog
126,Christye,Stores,Porcupine
127,Vince,Paolo,Kangaroo
128,Cristy,Fesby,Starling
129,Britney,Standfield,Burrowing owl
130,Kaleena,Volkers,Langur
131,Averil,Kimbell,Rhea
132,Carmelita,Gehrels,Spotted -tailed quoll
133,Cory,Sreenan,Hyrax
134,Sybille,Filippone,Dove
135,Delia,Forkan,Agama lizard
136,Sigfried,Cattlemull,Eastern dwarf mongoose
137,Rowena,James,Fox
138,Shane,Naisey,Jaguarundi
139,Janine,Ielden,Eastern fox squirrel
140,Beverlie,Biggerstaff,Fox
141,Menard,Archbould,Roe deer
142,Darline,Keating,Caribou
143,Heall,Chritchley,Beaver
144,Corenda,Bunnell,Glossy starling
145,Town,Mandal,White -winged tern
146,Pietrek,Primmer,Superb starling
147,Guillemette,Jasper,Gemsbok
148,Diane,Mewes,Civet
149,Hatty,Liddle,Grouse
150,Ange,Beardwell,Deer
*/
```

End Of Topic

1.6.9 *C# Linked List Class*

Here's the revised version, adding information about `ArrayList` in Section 3:

1. What is a Linked List? A linked list is a data structure used to store a sequence of elements, where each element is called a node. In the case of a doubly linked list:

- **Data**: Each node contains the data or value.
- **Pointers**: Each node has two pointers—one pointing to the next node and one pointing to the previous node.

C# has a built-in generic `LinkedList<T>` class that represents a doubly linked list, supporting dynamic memory allocation and efficient insertions and deletions.

2. Real-Life Applications of Linked Lists

- **Music Playlists**: Linked lists can manage playlists, where songs can be accessed or rearranged using the next and previous pointers.
- **Undo Functionality in Applications**: Applications like text editors can use linked lists to maintain a history of changes.
- **Web Browser History**: Browsers can use linked lists to manage navigation history, allowing forward and backward traversal.
- **Dynamic Memory Allocation**: Operating systems can use linked lists to manage memory blocks, making memory allocation and deallocation efficient.
- **Adjacency Lists in Graphs**: In graphs, adjacency lists are often implemented using linked lists for efficient traversal of vertices and edges.

3. ArrayList vs. LinkedList C# provides two main collections that can be used to manage dynamic sequences of elements: `ArrayList` and `LinkedList<T>`. Here's a comparison:

Feature	ArrayList	LinkedList (C#)
Underlying Data Structure	Resizable array	Doubly linked list
Memory Allocation	Contiguous (requires resizing)	Non-contiguous (nodes can be scattered)
Access Time	O(1) for index-based access	O(n) (must traverse nodes)
Insertion/Deletion	O(n) for shifting elements	O(1) at ends; O(n) for middle
Memory Overhead	Less (just array)	More (extra pointers for each node)
Search Complexity	O(n) (linear search)	O(n) (linear search)
Thread Safety	Not synchronized (not thread-safe)	Not synchronized (not thread-safe)

Key Differences:

- **Access Speed**:
  - `ArrayList` provides faster access due to contiguous memory allocation, allowing elements to be accessed directly via index.
  - `LinkedList<T>` requires traversal from the head to access an element by index, making it slower for random access.
- **Insertion and Deletion**:
  - `ArrayList` has slower insertion and deletion times (O(n) in the worst case) because elements need to be shifted.
  - `LinkedList<T>` excels in insertions and deletions at both ends (O(1)), making it ideal for scenarios where these operations are frequent.
- **Memory Usage**:
  - `ArrayList` typically uses less memory since it only stores the elements.
  - `LinkedList<T>` uses more memory due to the extra pointers needed for each node.

Use Cases:

- **ArrayList**: Better suited for scenarios where fast access to elements is needed and the size of the data structure is stable or grows infrequently.
- **LinkedList**: Ideal for applications that require frequent insertions and deletions, such as queues, stacks, and dynamic data management.

4. Use Cases

- **ArrayList**: Best for fast access and stable size.
- **LinkedList**: Ideal for scenarios involving frequent insertions and deletions, particularly at the ends.

5. C# Linked List Methods Here's a breakdown of common methods in the C# `LinkedList<T>` class:

C# Method	Description
`AddLast(T value)`	Adds an element to the end of the list.
`AddFirst(T value)`	Adds an element at the start of the list.
`Remove(T value)`	Removes the first occurrence of the specified element.
`RemoveFirst()`	Removes and returns the first element.
`RemoveLast()`	Removes and returns the last element.
`Count`	Returns the number of elements in the list.

6. Demo Code Here's a C# code example that performs basic operations with `LinkedList<T>`:

```
using System;
using System.Collections.Generic;

public class LinkedListDemo
{
    public static void Main(string[] args)
    {
        // Create a LinkedList
        LinkedList<string> list = new LinkedList<string>();

        // Add elements to the LinkedList
        list.AddLast("Apple");
        list.AddLast("Banana");
        list.AddLast("Cherry");
        list.AddFirst("Mango");
        list.AddLast("Orange");

        // Display the LinkedList
        Console.WriteLine("Initial LinkedList: " + string.Join(", ", list));

        // Access elements in the LinkedList
        Console.WriteLine("First element: " + list.First.Value);
        Console.WriteLine("Last element: " + list.Last.Value);

        // Check the size of the LinkedList
        Console.WriteLine("Size of LinkedList: " + list.Count);

        // Remove elements from the LinkedList
        list.RemoveFirst();
        list.RemoveLast();
        list.Remove("Banana");

        // Display the LinkedList after removals
        Console.WriteLine("LinkedList after removals: " + str

        // Check if LinkedList contains a specific element
        if (list.Contains("Cherry"))
        {
            Console.WriteLine("LinkedList contains Cherry");
        }
        else
        {
            Console.WriteLine("LinkedList does not contain Ch
        }

        // Clear the LinkedList
        list.Clear();
        Console.WriteLine("LinkedList after clearing: " + (li
    }
}
```

7. Iterator with LinkedList in C# Iterators allow sequential traversal through a linked list in C#, facilitating the processing of each element.

Example Code Using Iterator with LinkedList:

```
using System;
using System.Collections.Generic;

public class LinkedListIteratorDemo
{
    public static void Main(string[] args)
    {
        LinkedList<string> list = new LinkedList<string>(new[

        // Get an enumerator for the LinkedList
        var enumerator = list.GetEnumerator();

        Console.WriteLine("Traversing the LinkedList:");
        while (enumerator.MoveNext())
        {
            string element = enumerator.Current;
            Console.WriteLine(element);

            // Remove "Banana" during iteration
            if (element == "Banana")
            {
                list.Remove(element);
                Console.WriteLine("\"Banana\" has been remove
            }
        }

        // Display the LinkedList after iteration
        Console.WriteLine("\nLinkedList after iteration: " +
    }
}
```

8. Why Use an Iterator with LinkedList in C#?

- **Efficient Traversal**: Iterators (or enumerators) provide an efficient way to traverse through linked list nodes.
- **Modification During Traversal**: Enumerators allow safe modification of elements during traversal, making them useful for dynamic data handling.

1.6.10 *Java Linked List Class*

What is a Linked List? A **linked list** is a data structure used to store a collection of elements (nodes). Each node contains two parts:

1. **Data**: The actual value or information.
2. **Pointer (or Reference)**: A link to the next node in the sequence (and optionally, a link to the previous node in the case of doubly linked lists).

Linked lists allow for dynamic memory allocation and efficient insertions and deletions since nodes can be added or removed without reorganizing the entire structure. Unlike arrays, linked lists do not require a contiguous block of memory, making them more flexible.

Real-Life Applications of Linked Lists

1. **Music Playlists**:
   - A music player can use a linked list to manage a playlist. Each song can be a node, and the player can easily move forward or backward through the playlist (using next and previous pointers), allowing for easy manipulation of the song order.
2. **Undo Functionality in Applications**:
   - Many applications, like text editors, implement an undo feature using a linked list. Each action can be stored as a node in the list, allowing users to navigate backward through their actions and revert to previous states.
3. **Web Browser History**:
   - Browsers use linked lists to maintain the history of visited pages. Each page can be a node, and users can navigate forward and backward through their browsing history easily.
4. **Dynamic Memory Allocation**:
   - Operating systems often use linked lists for managing memory. For instance, free memory blocks can be stored as nodes, allowing the OS to allocate and deallocate memory efficiently as processes start and finish.
5. **Adjacency Lists in Graphs**:
   - In graph data structures, linked lists are often used to represent adjacency lists. Each node in the list represents a vertex, and its linked nodes represent the edges connecting to other vertices, allowing for efficient traversal of graph data.

These applications highlight the versatility and efficiency of linked lists in various scenarios, particularly where dynamic data management is essential.

ArrayList vs. LinkedList

Feature	ArrayList	LinkedList
Underlying Data Structure	Resizable array	Doubly linked list
Memory Allocation	Contiguous memory (requires resizing when full)	Non-contiguous memory (nodes can be scattered)
Access Time	O(1) for index-based access	O(n) for index-based access (must traverse nodes)
Insertion/Deletion	O(n) in the worst case (shifting elements needed)	O(1) for adding/removing at ends; O(n) for middle (traversal needed)
Memory Overhead	Less overhead (just the array)	More overhead (extra pointers for each node)
Performance for Large Data	Better for large datasets with frequent access	Better for large datasets with frequent insertions/deletions
Search Complexity	O(n) (linear search)	O(n) (linear search)
Thread Safety	Not synchronized (not thread-safe by default)	Not synchronized (not thread-safe by default)
Use Cases	Best for frequent access, fewer insertions/deletions	Best for frequent insertions/deletions, less access
Iteration	Faster for sequential access due to locality of reference	Slower for sequential access (due to pointers)

Summary of Key Differences

1. **Access Speed**:
   - **ArrayList** allows for faster access due to its contiguous memory allocation. You can directly access any element via its index.
   - **LinkedList** requires traversal from the head to access an element by index, making it slower for random access.

2. **Insertion and Deletion**:

   - **ArrayList** has slower insertion and deletion times (O(n) in the worst case) because elements may need to be shifted.
   - **LinkedList** excels in insertions and deletions at both ends (O(1)), making it ideal for scenarios where these operations are frequent.

3. **Memory Usage**:

   - **ArrayList** typically uses less memory per element since it only stores the elements.
   - **LinkedList** uses more memory due to the extra pointers needed for each node.

Use Cases

- **ArrayList**: Ideal for scenarios where you need fast access to elements and the size of the data structure is stable or grows infrequently.
- **LinkedList**: Better suited for applications that involve frequent insertions and deletions, such as queues or stacks.

Conclusion Choosing between an `ArrayList` and a `LinkedList` depends on the specific use case and performance needs. If you need frequent access and stable size, go for `ArrayList`. If your application requires a lot of insertions and deletions, opt for `LinkedList`.

Java Linked List Methods

Method	Description
`add(E e)`	Appends the specified element to the end of the list.
`add(int index, E element)`	Inserts the specified element at the specified position.
`addFirst(E e)`	Inserts the specified element at the beginning of the list.
`addLast(E e)`	Appends the specified element to the end of the list.
`remove(Object o)`	Removes the first occurrence of the specified element.
`remove(int index)`	Removes the element at the specified position.
`removeFirst()`	Removes and returns the first element of the list.
`removeLast()`	Removes and returns the last element of the list.
`get(int index)`	Returns the element at the specified position.
`set(int index, E element)`	Replaces the element at the specified position with the given element.
`size()`	Returns the number of elements in the list.
`isEmpty()`	Returns `true` if the list is empty, `false` otherwise.
`clear()`	Removes all elements from the list.
`contains(Object o)`	Returns `true` if the list contains the specified element.
`indexOf(Object o)`	Returns the index of the first occurrence of the specified element, or -1 if not found.
`lastIndexOf(Object o)`	Returns the index of the last occurrence of the specified element, or -1 if not found.
`listIterator()`	Returns a list iterator over the elements in the list.
`iterator()`	Returns an iterator over the elements in the list.
`toArray()`	Returns an array containing all elements in the list.
`toArray(T[] a)`	Returns an array containing all elements in the list; the runtime type of the returned array is that of the specified array.
`addAll(Collection<? extends E>c)`	Appends all elements in the specified collection to the end of the list.
`addAll(int index, Collection<? extends`	Inserts all elements in the specified collection at the

Demo Code Here is a Java program that demonstrates the most frequently used methods in the `LinkedList` class. This program showcases common operations such as adding, removing, accessing elements, and checking the size of the list:

```
import java.util.LinkedList;

public class LinkedListDemo {
    public static void main(String[] args) {
        // Create a LinkedList
        LinkedList<String> list = new LinkedList<>();

        // Add elements to the LinkedList
        list.add("Apple");
        list.add("Banana");
        list.add("Cherry");
        list.addFirst("Mango"); // Adds element at the start
        list.addLast("Orange"); // Adds element at the end

        // Display the LinkedList
        System.out.println("Initial LinkedList: " + list);

        // Access elements in the LinkedList
        System.out.println("First element: " + list.getFirst());
        System.out.println("Last element: " + list.getLast());
        System.out.println("Element at index 2: " + list.get(2));

        // Check the size of the LinkedList
        System.out.println("Size of LinkedList: " + list.size());

        // Remove elements from the LinkedList
        list.removeFirst(); // Removes the first element
        list.removeLast();  // Removes the last element
        list.remove(1);     // Removes element at index 1

        // Display the LinkedList after removals
        System.out.println("LinkedList after removals: " + list);

        // Check if LinkedList contains a specific element
        if (list.contains("Banana")) {
            System.out.println("LinkedList contains Banana");
        } else {
            System.out.println("LinkedList does not contain Banana");
        }

        // Clear the LinkedList
        list.clear();
        System.out.println("LinkedList after clearing: " + list);
    }
}
```

Explanation of the Methods Used:

1. **`add(element)`** - Adds an element to the end of the list.
2. **`addFirst(element)`** - Adds an element at the beginning of the list.

3. **addLast(element)** - Adds an element at the end of the list.
4. **get(index)** - Returns the element at the specified position.
5. **getFirst()** / **getLast()** - Returns the first or last element, respectively.
6. **size()** - Returns the number of elements in the list.
7. **remove(index)** - Removes the element at the specified position.
8. **removeFirst()** / **removeLast()** - Removes the first or last element, respectively.
9. **contains(element)** - Checks if the list contains the specified element.
10. **clear()** - Removes all the elements from the list.

LinkedList Iterator The **LinkedList Iterator** in Java is an object that allows you to traverse the elements of a `LinkedList` sequentially. The `Iterator` is part of the Java Collections Framework and provides methods to iterate through the list, remove elements during iteration, and check for more elements.

Key Methods of the Iterator:

1. **hasNext()**: Returns `true` if there are more elements to iterate.
2. **next()**: Returns the next element in the iteration.
3. **remove()**: Removes the last element returned by the iterator (optional operation).

Example Code Using `Iterator` with `LinkedList`: Here's a sample Java program that demonstrates how to use the `Iterator` to traverse a `LinkedList`:

```
import java.util.LinkedList;
import java.util.Iterator;

public class LinkedListIteratorDemo {
    public static void main(String[] args) {
        // Create a LinkedList
        LinkedList<String> list = new LinkedList<>();
        list.add("Apple");
        list.add("Banana");
        list.add("Cherry");
        list.add("Mango");

        // Get the iterator from the LinkedList
        Iterator<String> iterator = list.iterator();

        // Iterate through the LinkedList
        System.out.println("Traversing the LinkedList:");
        while (iterator.hasNext()) {
            String element = iterator.next();
            System.out.println(element);

            // Remove "Banana" during iteration
            if (element.equals("Banana")) {
                iterator.remove();
                System.out.println("\"Banana\" has been remove
            }
        }

        // Display the LinkedList after iteration
        System.out.println("\nLinkedList after iteration: " +
    }
}
```

Output:

```
Traversing the LinkedList:
Apple
Banana
"Banana" has been removed
Cherry
Mango

LinkedList after iteration: [Apple, Cherry, Mango]
```

Explanation:

1. **Create a `LinkedList`**:
   - We create a `LinkedList` of strings and add elements to it.
2. **Get an iterator**:
   - The `iterator()` method of `LinkedList` is used to obtain an `Iterator` for the list.
3. **Use the iterator to traverse**:
   - We loop through the list using `hasNext()` and `next()`, which ensure that each element is processed one by one.
4. **Remove an element during iteration**:
   - The `remove()` method removes the last element returned by the iterator.
   - In this case, if the current element is `"Banana"`, it is removed from the list.

Why Use an Iterator with LinkedList?

- **Efficient Traversal**: Iterators are efficient for traversing and manipulating lists because they maintain the state of traversal.

- **Modification During Traversal**: Unlike using a `for` loop or enhanced `for` loop, an `Iterator` allows safe removal of elements during traversal, avoiding `ConcurrentModificationException`.

The iterator is a flexible and powerful way to navigate through and manipulate `LinkedList` elements in Java.

1.7 Stacks and Queues

1.7.1 *Key Ideas*

- Stacks
- Queues

1.7.2 *Discussion*

Stacks and queues are a way of organizing data and consuming data. The data can be stored in an array or linked list.

> **Note**
>
> A **Stack** is a way of organizing and consuming data where the Last In is the First Out (LIFO). Or it can be thought of as First In is the Last Out (FILO).

Stack Most people think of a stack as a stack of plates, where each plate represents data. Data is ***pushed*** onto the stack. Meaning. it is added to the array or linked list. Data is consumed by ***popping*** it off of the stack. Meaning, that the last piece of data added to the stack is removed from the array or linked list. This data is usually used by the part of the program that is popping/removing it from the stack. ***Peeking*** is looking at the last/next piece of data, but *not* removing it from the stack. An example is Stack Array

Methods usually associated with a **stack** are as follows:

- **Push** - adds data to the stack
- **Pop** - removes data from the stack
- **Peek** - retrieves the next piece/top data from the stack, but does not remove it.
- **isFull** - this is used when making a stack with an array since an array has a limited number of elements.
- **isEmpty** - this is used to determine if the stack is empty.

In computer science, a stack is a LIFO (Last In, First Out) data structure that stores elements in a linear order. This means that the last element added to the stack is the first one to be removed. Stacks are often used to manage a sequence of operations, such as function calls or undo/redo functionality.

Here are some of the key properties of stacks:

- **LIFO order**: Elements are removed in the reverse order they were added.
- **Bounded capacity**: Stacks have a limited size, and adding more elements than the stack can hold will result in an error.
- **Efficient access**: Pushing and popping elements from the top of the stack is a very efficient operation, typically taking constant time.

Stacks are used in a variety of applications in computer science, including:

- **Function calls**: When a function is called, its arguments and local variables are pushed onto a stack. When the function returns, its stack frame is popped off the stack.
- **Expression evaluation**: Stacks are used to evaluate expressions in many programming languages. For example, when evaluating arithmetic expressions, operands are pushed onto the stack, and then operators are popped off the stack and applied to the operands.
- **Backtracking algorithms**: Stacks are used to store the history of decisions made in backtracking algorithms, such as depth-first search. This allows the algorithm to backtrack to previous states if it reaches a dead end.
- **Undo/redo functionality**: Stacks are used to implement undo/redo functionality in many applications. When an action is performed, the state of the application is pushed onto the undo stack. To undo the action, the state is popped off the undo stack and restored.

Here are some examples of how stacks are used in real-world applications:

- **Compilers**: Compilers use stacks to keep track of the current state of the program being compiled.
- **Interpreters**: Interpreters use stacks to evaluate expressions and statements in the program being interpreted.
- **Web browsers**: Web browsers use stacks to keep track of the history of visited web pages.
- **Operating systems**: Operating systems use stacks to manage memory allocation and process scheduling.

Stacks are a versatile and powerful data structure that is used in a wide variety of applications in computer science. Their LIFO order and efficient access make them well-suited for managing sequences of operations and storing the history of decisions made in algorithms.

> **Note**
>
> A **Queue** is a way of organizing and consuming data where the First In is the First Out (FIFO).

Queue Most people think of a queue as people standing in a line, where people represent data. Data is consumed in the order in which it was added.

Methods usually associated with a **queue** are as follows:

- **Enqueue** - add data to the queue.
- **Dequeue** - remove data from the queue
- **Peek** - retrieves the next piece/"head" data from the queue, but does not remove it.
- **isFull** - this is used when making a queue with an array since an array has a limited number of elements.
- **isEmpty** - this is used to determine if the queue is empty.

In computer science, a queue is a LIFO (First In, First Out) data structure that stores elements in a linear order. This means that the first element added to the queue is the first one to be removed. Queues are often used to manage a sequence of tasks or events, such as printing jobs or network traffic.

Here are some of the key properties of queues:

- **FIFO order**: Elements are removed in the order they were added.
- **Bounded capacity**: Queues have a limited size, and adding more elements than the queue can hold will result in an error.
- **Efficient access**: Adding and removing elements from the front and back of the queue are efficient operations, typically taking constant time.

Queues are used in a variety of applications in computer science, including:

- **Task scheduling**: Queues are used to schedule tasks in operating systems and other systems. For example, a print queue is used to manage the order in which print jobs are processed.
- **Network traffic**: Queues are used to manage the flow of data in networks. For example, a network buffer is used to store data packets that are waiting to be sent over a network.
- **Buffering**: Queues are used to buffer data between different components of a system. For example, a keyboard buffer is used to store characters typed on a keyboard until they can be processed by the operating system.
- **Message passing**: Queues are used to pass messages between different processes or threads. For example, a message queue is used to store messages that are waiting to be processed by a consumer process.

Here are some examples of how queues are used in real-world applications:

- **Operating systems**: Operating systems use queues to manage task scheduling, memory allocation, and device drivers.
- **Networking**: Networking protocols use queues to manage the flow of data packets.
- **Multimedia applications**: Multimedia applications use queues to buffer audio and video data.
- **Messaging applications**: Messaging applications use queues to store messages that are waiting to be delivered to recipients.

Queues are a versatile and powerful data structure that is used in a wide variety of applications in computer science. Their FIFO order and efficient access make them well-suited for managing sequences of tasks or events.

Definition
A **Priority Queue** is a queue where there is a mechanism to place data at the start or infront of other data.

Priority Queue In computer science, a priority queue is a specialized type of queue where elements are prioritized based on their associated priority values. Elements with higher priority values are served before elements with lower priority values. Priority queues are often implemented using heap data structures, which allow for efficient insertion and extraction operations.

Here are some of the key properties of priority queues:

- **Priority-based ordering**: Elements are served based on their priority values, with higher priority elements being served first.
- **Efficient insertion and extraction**: Priority queues can efficiently insert and extract elements, typically taking logarithmic time.
- **Dynamic priority updates**: Priority values can be updated dynamically, allowing for reprioritization of elements within the queue.

Priority queues are used in a variety of applications where prioritization is important, including:

- **Task scheduling**: Priority queues are used to schedule tasks based on their urgency or importance. For example, a critical job processing system might use a priority queue to prioritize high-priority tasks over less urgent ones.

- **Network traffic management**: Priority queues can be used to prioritize network traffic based on its importance or quality of service requirements. For instance, real-time voice or video traffic might be prioritized over non-real-time data transfers.
- **Event handling**: Priority queues can be used to manage a sequence of events, ensuring that high-priority events are handled first. For example, an event-driven system might use a priority queue to prioritize system alerts or critical user interactions.
- **Algorithm optimization**: Priority queues are used in various algorithms, such as Dijkstra's algorithm for shortest path finding and A* search for pathfinding in graph-based problems.

Here are some real-world examples of how priority queues are used:

- **Operating systems**: Operating systems use priority queues to manage task scheduling, ensuring that high-priority tasks, such as system processes, are executed before less urgent user tasks.
- **Network routers**: Network routers use priority queues to manage network traffic, prioritizing real-time audio or video traffic over non-real-time data transfers.
- **Emergency response systems**: Emergency response systems might use priority queues to prioritize dispatching emergency responders based on the severity of incidents.
- **Hospital patient care**: Critical care units in hospitals might use priority queues to manage patient care, ensuring that patients with the most urgent medical needs are seen first.

Priority queues are a valuable data structure that plays a crucial role in various applications that require efficient prioritization and management of tasks, events, or data. Their ability to handle dynamic priority updates and their efficient insertion and extraction operations make them a well-suited choice for prioritizing elements in a variety of contexts.

End Of Topic

1.7.3 *Stack - Using Array of Objects*

Stack Methods Methods usually associated with a **stack** are as follows:

- **Push** - adds data to the stack
- **Pop** - removes data from the stack
- **Peek** - retrieves the next piece/top data from the stack, but does not remove it.
- **isFull** - this is used when making a stack with an array since an array has a limited number of elements.
- **isEmpty** - this is used to determine if the stack is empty.

> **Tip**
> isFull is not needed if a Stack is created using a Linked List

Details discussed here

Lecture Code

```
/*
 * Project: Stack Using Array of Objects
 * Programmer: James Goudy
 * DS132SU_StackArray
 *
 *
 * Stack
 * push
 * pop
 * peek
 * isEmpty
 * isFull
 *
 *
 */

class Town {

    public String city;
    public int population;

    // constructor
    public Town(String city, int population) {
        this.city = city;
        this.population = population;
    }

    public void displayCity() {
        System.out.print("{" + city + ", " + population + "} ");
    }

}

class Stack {

    private int maxSize;
    private Town[] stackArray;
    private int top = -1;

    //constructor
    public Stack(int maxSize) {
        this.maxSize = maxSize;
        stackArray = new Town[maxSize];
        top = -1;
    }

    public boolean push(String city, int population) {

        // add data to the stack
        if (isFull()) {
            return false;
        } else {
            Town theTown = new Town(city, population);
            stackArray[++top] = theTown;
            return true;
        }

    }

    public Town pop() {
        // remove data from the stack
        return stackArray[top - -];
    }

    public Town peek() {
        // look/peek at the top of the stack
        return stackArray[top];
    }

    public boolean isEmpty() {
        //check if the array is empty
        return (top == -1);
    }

    public boolean isFull() {
        // check if the array is full
        return (top == maxSize - 1);
    }

}

public class DS132SU_StackArray {

    public static void main(String[] args) {

        // create a stack
        Stack myStack = new Stack(10);
        Town tempTown = null;

        // add data to the stack
        myStack.push("Kali", 300000);
```

```
        myStack.push("Bozeman", 100000);
        myStack.push("Whitefish", 40000);
        myStack.push("Columbia Falls", 30000);

        // peek at the top data
        System.out.print("Peek - ");
        myStack.peek().displayCity();
        System.out.println("");

        // pop one data object from the stack and store it in an object
        tempTown = myStack.pop();
        tempTown.displayCity();

        // empty the list
        while (!myStack.isEmpty()) {
            myStack.pop().displayCity();
        }

        System.out.println("");

        //ternary operator
        boolean flag;
        flag = myStack.push("Plains", 15000) ? true : false;

        if (flag) {
            System.out.println("Item added");
        } else {
            System.out.println("Item NOT added");
        }

        System.out.print("\nbye");
    }
}
```

End of Topic

1.7.4 *Queue - Using Array Of Objects*

Queue Methods Methods usually associated with a **queue** are as follows:

- **Enqueue** - add data to the queue.
- **Dequeue** - remove data from the queue
- **Peek** - retrieves the next piece/"head" data from the queue, but does not remove it.
- **isFull** - this is used when making a queue with an array since an array has a limited number of elements.
- **isEmpty** - this is used to determine if the queue is empty.

> **Tip**
> isFull is not needed if a Stack is created using a Linked List

```
// make a new queue object

 Queue theQue = new Queue(5);

 /*
  NOTICE THE CODE BELOW -
  make an object from the sub class
 */

Queue.Town theTown = theQue.new Town("", 0);
```

Details discussed

Lecture Code

```
/*
 *Programmer: James Goudy
 *Project: Queue of Objects
 *
 */
package com.mycompany.ds132su_queue;

/**
 *
 * @author jgoudy
 */
class Queue {

    class Town {
        // subclass
        public String city;
        public int population;

        public Town(String city, int population) {
            this.city = city;
            this.population = population;
        }

        public void displayTown() {
            System.out.print("{" + city + " - " + population +
        }

}// end of town

private int maxSize;
private Town[] queArray;
private int numItems;
private int head;
private int tail;

// Queue Constructor
public Queue(int maxSize) {
    this.maxSize = maxSize;
    queArray = new Town[maxSize];
    head = 0;
    tail = -1;
    numItems = 0;
}

public boolean isFull() {
    return (numItems == maxSize);
}

public boolean isEmpty() {
    return (numItems == 0);
}

public boolean enqueue(String city, int population) {

    // insert at tail
    // "enqueue"
    if (isFull()) {
        return false;
    }

    // create town
    Town newTown = new Town(city, population);

    // wrap to front if neccessary
    if (tail == maxSize - 1) {
        tail = -1;
    }

    // add object to array
    queArray[++tail] = newTown;

    // increment the count of objects in the array
    numItems++;

    return true;
}

public Town dequeue() {
    // remove from the front - the head
    // "dequeue"
```

```
        if (isEmpty()) {
            System.out.print("Queue is empty");
            return null;
        }

        // retreive the next in queue and move the head to
        // the next data item.
        Town temp = queArray[head++];

        if (head == maxSize) {
            head = 0;
        }

        // decrease the count of objects in the array
        numItems - -;

        return temp;
    }

    public Town peek() {
        // retreive the head information but do not remove
        return queArray[head];
    }

    public void displayQueue() {
        int cntr = 0;
        int pos = head;
        while (cntr < numItems) {
            queArray[pos].displayTown();
            cntr++;

            pos++;
            // loop to the start of the array if necessary
            if (pos == maxSize) {
                pos = 0;
            }

        }
        System.out.println("\n");
    }

} // end of queue

public class DS132SU_Queue {

    public static void main(String[] args) {

        // make a new queue object
        Queue theQue = new Queue(5);

        // make an object from the sub class
        Queue.Town theTown = theQue.new Town("", 0);

        String acity;

        theQue.enqueue("Bozeman", 100000);
        theQue.enqueue("Culver", 100001);
        theQue.enqueue("Dover", 100002);
        theQue.enqueue("Edger", 100003);

        System.out.println("\n - - - - - - Queue - - - - - - - - -");

        theQue.displayQueue();

        System.out.println("\nDequeue the first two items in

        // return next item from the queue
        theTown = theQue.dequeue();
        System.out.println("\n" + theTown.city + " - " + theT

        theTown = theQue.dequeue();
        System.out.println("\n" + theTown.city + " - " + theT

        System.out.println("\n - - - - - - - - - - - - - -\n");

        System.out.println("\nDequeue and retreive only the c

        // another option to dequeue and retrieve city only
        acity = theQue.dequeue().city;
        System.out.println(acity);

        System.out.println("\n - - - - - - Queue - - - - - - -

        theQue.displayQueue();

        theQue.enqueue("Franklin", 104);
        theQue.enqueue("Georgetown", 105);
        theQue.enqueue("Highland", 106);

        System.out.println("\n - - - - - - - Queue - - - - - -
        theQue.displayQueue();

        System.out.println("\n - - - - - Peek just city - - -

        acity = theQue.peek().city;
        System.out.println(acity);

        System.out.println("\n - - - - - Peek city population

        theTown = theQue.peek();
        System.out.println("\n" + theTown.city + " - " + theT

        System.out.println("\n - - - - - - Queue - - - - - - -
        // notice that peek did not remove any towns
        theQue.displayQueue();

    }
}
/*
 Output

 - - - - - - Queue - - - - - - - -

{Bozeman - 100000} {Culver - 100001} {Dover - 100002} {Edger
```

```
Dequeue the first two items in the queue

Bozeman - 100000
Culver - 100001

 - - - - - - - - - - - - - -

Dequeue and retreive only the city
Dover

 - - - - - - Queue - - - - - - - -
{Edger - 100003}


 - - - - - - - Queue - - - - - - -
{Edger - 100003} {Franklin - 104} {Georgetown - 105} {Highland - 106}


 - - - - - Peek just city - - - - - - - - -
Edger

 - - - - - Peek city population - - - - - - - - -

Edger - 100003

 - - - - - - Queue - - - - - - - -

{Edger - 100003} {Franklin - 104} {Georgetown - 105} {Highland - 106}


 */
```

End Of Topic

1.7.5 *Priority Queue*

Key Ideas

- Priority Queue

> **Definition**
> A **Priority Queue** is a queue where there is a mechanism to place data at the start or in front of other data.

Lecture Code

```
// priorityQ.java
// demonstrates priority queue

/**
 *
 * @author jgoudy
 */
class PriorityQ {

    private int maxSize;
    private int[] queArr;
    private int nItems;

    // constructor
    public PriorityQ(int maxSize) {
        this.maxSize = maxSize;
        queArr = new int[maxSize];
        nItems = 0;
    }

    public boolean isFull() {
        return (nItems == maxSize);
    }

    public boolean isEmpty() {
        return (nItems == 0);
    }

    // enqueue - add to queue
    // lower numbers take a higher place in the queue

    public void enqueue(int key) {
        int c;

        if (isEmpty()) {
            queArr[nItems++] = key;
        } else {
            for (c = nItems - 1; c >= 0; c - -) {
                if (key > queArr[c]) {
                    queArr[c + 1] = queArr[c];
                } else {
                    break;
                }
            }// end of for
            queArr[c + 1] = key;

            nItems++;
        }
    }

    // retreive the next item from the queue
    // and remove it from the queue
    public int dequeue() {
        return queArr[ - -nItems];
    }

    // reteive the data from the next item in the queue
    // but does NOT remove it
    public int peek() {
        return queArr[nItems - 1];
    }

    // print the queue
    public void printPriorityQ() {
        System.out.println();
        for (int c = 0; c < nItems; c++) {
            System.out.print(queArr[c] + " ");
        }
        System.out.println();

    }

} // end of class

public class Ds_priorityQueue {

    public static void main(String[] args) {

// assumption lower the number - higher the priority
        PriorityQ thePQ = new PriorityQ(10);

        // add data items to the queue
        thePQ.enqueue(30);
        thePQ.enqueue(50);
        thePQ.enqueue(10);
        thePQ.enqueue(40);
        thePQ.enqueue(20);
        thePQ.enqueue(60);
        thePQ.enqueue(5);

        // print the queue
        thePQ.printPriorityQ();

        System.out.println("\n - - - - - - - - - - -\n");

        // peek at the next data item
        System.out.println("Peek " + thePQ.peek());

        System.out.println("\n - - - - - - - - - - -\n");

        // remove all the items from the queue
        while (!thePQ.isEmpty()) {
```

```
            int x = thePQ.dequeue();
            System.out.print(x + " ");
        }

        System.out.println("\n - - - - - - - - - - -\n");

    }
}

/* Output

 60 50 40 30 20 10 5

 - - - - - - - - - - -

Peek 5

 - - - - - - - - - - -

5 10 20 30 40 50 60
 - - - - - - - - - - -

 */
```

End Of Topic

1.8 Sorting Algorithms

- Bubble Sort

1.8.1 *Visualizations*

Visual Aglo
Toptal
Comparison Sorting Algorithms
Sort Visualizer

End Of Topic

1.8.2 *Bubble Sort*

Key Ideas

- Bubble Sort

> **Definition**
> **Bubble Sort** is a simple sorting algorithm that repeatedly iterates through the list or array. It compares adjacent elements and swaps them if they are in the wrong order. The algorithm repeatedly passes through the list until the list is sorted. It is a comparison sort and is named for the way smaller or larger elements "bubble" to the top of the list.

Performance Bubble sort has a worst-case and average complexity of $O(n^2)$.

Visualizations Visual Aglo
Toptal
Comparison Sorting Algorithms
Sort Visualizer

Videos —
—

Lecture Code

```
/*
 * Programmer: James Goudy
 * Project: Bubble Sort
 */
package com.mycompany.bubblesort_lecturecode;

import java.util.Random;

class BubbleSort {

    // arrInt is an array of integers
    // numDataElements is the actual count
    // of elements of data in the array
    // algorithm assumes the data is contiguous
    int arrInt[];
    int numDataElments;

    public BubbleSort(int[] arrInt, int numDataElments) {
        this.arrInt = arrInt;
        this.numDataElments = numDataElments;
    }

    public void Sort() {
        //
        int n = numDataElments;

        for (int c = 0; c < n; c++) {
            for (int j = 1; j < (n); j++) {

                // check if the left element is
                // greater to the one on the right
                // "Bubble" the lowest to the left

                if (arrInt[j - 1] > arrInt[j]) {
                    // swap left element arr[j -1]
                    // with the one on the right and arr[j]

                    // store left one in temp
                    int temp = arrInt[j - 1];
                    //copy the right into the left
                    arrInt[j - 1] = arrInt[j];
                    //copy the left into the right
                    arrInt[j] = temp;
                }
            }
        }
    }

}

public class BubbleSort_LectureCode {

    static int arrSize = 8;
    //static int theArray[] = new int[arrSize];
    static int theArray[] ={3,60,35,2,45,320,5,1};

    static void fillTheArray()
    {
        Random RNG  = new Random();
        for(int c = 0; c < arrSize; c++)
        {
            theArray[c] = RNG.nextInt(0,(arrSize*10));
        }
    }

    static void printArray(int anArray[], int numOfDataElemen
    {
        System.out.println("");
        for (int i = 0; i < numOfDataElements; i++) {
            System.out.print(anArray[i] + " ");
        }
        System.out.println("\n - - - - - - - - - - - - - -\n"

    }

    public static void main(String[] args) {

        // option to randomly fill the array
        //fillTheArray();

        printArray(theArray, arrSize);

        BubbleSort bs = new BubbleSort(theArray, arrSize);
        bs.Sort();

        printArray(theArray, arrSize);
```

```
    }
}

/*
3 60 35 2 45 320 5 1
 - - - - - - - - - - - - - - -


1 2 3 5 35 45 60 320
 - - - - - - - - - - - - - - -
*/
```

End Of Topic

1.8.3 *Merge Sort*

Here's a summary of the pros and cons of merge sort presented in bullet points:

Pros of Merge Sort:

- **Efficient for large datasets:** Merge sort's average and worst-case time complexities are O(n log n), making it highly efficient for sorting large datasets.
- **Stable sorting algorithm:** Merge sort preserves the original order of equal elements, making it a stable sorting algorithm. This is particularly useful when sorting data that contains multiple instances of the same value.
- **Adaptable to external sorting:** Merge sort can be adapted for external sorting, where the data to be sorted is too large to fit into main memory.
- **Highly parallelizable:** Merge sort can be effectively parallelized, making it suitable for multithreaded and distributed computing environments.

Cons of Merge Sort:

- **Additional memory usage:** Merge sort requires additional memory to store the temporary sublists created during the sorting process.
- **Overhead for small datasets:** For small datasets, the overhead of recursion and merging can make merge sort less efficient than simpler algorithms like insertion sort.
- **Not in-place sorting:** Merge sort is not an in-place sorting algorithm, meaning it requires additional memory to store the sorted result.

Overall, merge sort is a versatile and efficient sorting algorithm that is particularly well-suited for large datasets. However, its additional memory requirements and overhead for small datasets make it less suitable for certain applications.

1.8.4 *Merge Sort - Demo Code*

How Merge Sort Works Merge sort is a **divide-and-conquer** sorting algorithm. It recursively divides the array into two halves until each subarray has only one element (which is trivially sorted). Then it **merges** these sorted halves back together, comparing elements and building a larger sorted array step by step. This merging process continues until the entire array is reassembled in sorted order.

The algorithm has a consistent time complexity of **O(n log n)** — it divides the array (log n times) and merges n elements at each level. Merge sort is stable and efficient but requires extra space proportional to the array size due to its temporary workspace array.

Arrays

Demo Code

```
/*
Engineer: James Goduy
 */
package mergesort_rev;

import java.util.Random;



/*
 * The Sorter interface defines a contract that all sorting classes must follow.
 * Any class that implements Sorter must provide concrete versions of insert(),
 * sort(), and displayArray(). These are method signatures only—
no code
 * is written here.
 *
 * Using an interface allows us to treat different sorting algorithms (e.g.,
 * recursive and iterative merge sorts) as the same "type." This is an example
 * of polymorphism: we can write one block of code that works with any object
 * implementing Sorter, regardless of how it performs the sorting internally.
 *
 * In short, the interface provides a common structure (what must be done),
 * while each class defines its own behavior (how it is done).
 */

interface Sorter {
    void insert(int value);
    void sort();
    void displayArray();
}


class MergeSort implements Sorter {

    private int[] theArray;  // Main array to be sorted
    private int nElems;      // Current number of elements in the array
    private int max;         // Maximum capacity of the array

    // Constructor: initializes the array and tracking variables
    public MergeSort(int max) {
        theArray = new int[max];
        nElems = 0;
        this.max = max;
    }

    // Check if array is full
    private boolean isFull() {
        return nElems == max;
    }

    // Insert a value into the array (if space allows)
    public void insert(int value) {
        if (isFull()) {
            System.out.println("Array is full");
            return;
        }
        theArray[nElems++] = value; // Place value and increme
    }

    // Public method to start the merge sort process
    public void sort() {
        int[] workspace = new int[nElems]; // Temporary works
        recMergeSort(workspace, 0, nElems - 1); // Begin recu
    }

    // Recursive function that splits and sorts subarrays
    private void recMergeSort(int[] workspace, int lowerBound
        if (lowerBound >= upperBound) return; // Base case: o
        int mid = (lowerBound + upperBound) / 2; // Find midpo

        // Recursively sort left and right halves
        recMergeSort(workspace, lowerBound, mid);
        recMergeSort(workspace, mid + 1, upperBound);

        // Merge the two sorted halves
        merge(workspace, lowerBound, mid + 1, upperBound);
    }

    // Merge two sorted halves into a single sorted ran
    private void merge(int[] workspace, int lowPtr, int highP
        int j = 0;                  // Index for workspace
        int lowerBound = lowPtr;    // Start index of left h
        int mid = highPtr - 1;      // End index of left hal
        int n = upperBound - lowerBound + 1; // Total element

        // Compare elements from both halves and copy the sma
        while (lowPtr <= mid && highPtr <= upperBound) {
            workspace[j++] = (theArray[lowPtr] <= theArray[hi
                    ? theArray[lowPtr++]
                    : theArray[highPtr++];
        }

        // Copy any remaining elements from the left half
        if (lowPtr <= mid) {
            System.arraycopy(theArray, lowPtr, workspace, j, n
            j += (mid - lowPtr + 1);
        }
```

```
        // Copy any remaining elements from the right half
        if (highPtr <= upperBound) {
            System.arraycopy(theArray, highPtr, workspace, j, upperBound - highPtr + 1);
        }

        // Copy merged elements back into the original array
        System.arraycopy(workspace, 0, theArray, lowerBound, n);
    }

    // Utility method to display the array contents
    public void displayArray() {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < nElems; i++) sb.append(theArray[i]).append(' ');
        System.out.println(sb.toString());
    }
} // End Of Class

class MergeSort_Iterative implements Sorter {

    // Instance Variables
    private int[] theArray;  // The array to be sorted
    private int nElems;      // Current number of elements in the array
    private int max;         // Maximum array capacity (limit)

    // Constructor
    public MergeSort_Iterative (int max) {
        // Initialize the array and size counters
        theArray = new int[max];
        nElems = 0;
        this.max = max;
    }

    // Utility Method
    // Checks whether the array is already full
    private boolean isFull() {
        return nElems == max;
    }

    // Insert Method
    // Adds a value to the array if space is available
    public void insert(int value) {
        if (isFull()) {
            System.out.println("Array is full");
            return;
        }
        // Store value and increase the element count
        theArray[nElems++] = value;
    }

    // Iterative Merge Sort
    public void sort() {
        int[] workspace = new int[nElems]; // Temporary array used during merging

        // subSize represents the size of subarrays being merged each pass
        // Start with subarrays of size 1 and double the size each iteration
        for (int subSize = 1; subSize < nElems; subSize *= 2) {

            // Process and merge all pairs of subarrays of the
            for (int leftStart = 0; leftStart < nElems - subSi

                // leftStart: beginning index of left subarra
                int mid = leftStart + subSize - 1; // Last in

                // Ensure the right subarray does not exceed
                int rightEnd = Math.min(leftStart + 2 * subSi

                // Merge the two adjacent sorted subarrays in
                merge(workspace, leftStart, mid + 1, rightEnd
            }
        }
    }

    // Merge Method
    // Merges two sorted halves into a single sorted section
    private void merge(int[] workspace, int lowPtr, int highP
        int j = 0;                  // Index for workspace
        int lowerBound = lowPtr;    // Start index of the l
        int mid = highPtr - 1;      // End index of the lef
        int n = upperBound - lowerBound + 1; // Total number

        // Compare elements from both halves and copy the sma
        while (lowPtr <= mid && highPtr <= upperBound) {
            if (theArray[lowPtr] <= theArray[highPtr])
                workspace[j++] = theArray[lowPtr++]; // Copy
            else
                workspace[j++] = theArray[highPtr++]; // Copy
        }

        // Copy any remaining elements from the left half
        while (lowPtr <= mid) {
            workspace[j++] = theArray[lowPtr++];
        }

        // Copy any remaining elements from the right half
        while (highPtr <= upperBound) {
            workspace[j++] = theArray[highPtr++];
        }

        // Copy merged elements back into the original array
        for (int k = 0; k < n; k++) {
            theArray[lowerBound + k] = workspace[k];
        }
    }

    // Utility method to display the array contents
    public void displayArray() {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < nElems; i++) sb.append(theArray[i
        System.out.println(sb.toString());
    }
}
```

```
public class MergeSort_Rev {


    public static void main(String[] args)
    {

        int maxSize = 15    ;
        int sampleSize =  (int)(maxSize * .9);
        int choice = 2;

        Sorter ms;

        // create the object
        if(choice == 1)
        {
            // recursive
         ms = new MergeSort(maxSize);
        }
        else
        {
            // iterative - loops
            ms = new MergeSort_Iterative (maxSize);
        }

        // Create RNG
        Random RNG = new Random();

        // insert random numbers
        for(int c = 0 ; c < sampleSize; c++)
        {
            ms.insert(RNG.nextInt(0,maxSize));
        }

        // display the orignal array
        ms.displayArray();


        // sort the list
        ms.sort();

        // spacing
        System.out.println();

        // display the sorted list
        ms.displayArray();

        System.out.println("\nbye\n");
    }

}
```

Linked Lists

Demo Code

```
/*
 * Example program demonstrating two implementations of Merge Sort
 * for a linked list of people (first name, last name, city).
 *
 * Both recursive (top-down) and iterative (bottom-up) versi
 * implement a common interface (Sorter) for interchangeable
 *
 * Author: [Your Name]
 * Course: [Your Class Name or Section]
 */

package mergesortlinkedlist;

// - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Interface Definition
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

/**
 * The Sorter interface defines a simple contract for
 * inserting, sorting, and displaying a collection.
 *
 * Both recursive and iterative merge sort classes
 * will implement this interface to ensure consistent usage.
 */
interface Sorter {
    void insert(String fn, String ln, String cty);
    void sort();
    void displayList();
}

// - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Recursive Merge Sort Implementation
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

/**
 * MergeSortLinkedList_Recursive
 *
 * Uses a recursive (top-down) merge sort approach.
 * The list is divided into halves until single nodes remain,
 * then merged back together in sorted order by last name.
 */
class MergeSortLinkedList_Recursive implements Sorter {

    /**
     * Inner class Node -
represents a single record in the linked list.
     * Each node stores a person's first name, last name, and
     */
    class Node {
        String firstName;
        String lastName;
        String city;
        Node next;

        Node(String firstName, String lastName, String city)
            this.firstName = firstName;
            this.lastName = lastName;
            this.city = city;
            this.next = null;
        }
```

```
    }

    private Node head;  // Head pointer for the linked list

    /**
     * Inserts a new node at the end of the list.
     */
    public void insert(String firstName, String lastName, String city) {
        Node newNode = new Node(firstName, lastName, city);
        if (head == null) {
            head = newNode;
            return;
        }

        Node current = head;
        while (current.next != null)
            current = current.next;
        current.next = newNode;
    }

    /**
     * Public sort method - initiates recursive merge sort.
     */
    public void sort() {
        head = mergeSort(head);
    }

    /**
     * Recursive merge sort: splits the list, sorts each half, and merges them.
     */
    private Node mergeSort(Node h) {
        // Base case: 0 or 1 element
        if (h == null || h.next == null) return h;

        // Split the list into two halves
        Node middle = getMiddle(h);
        Node nextOfMiddle = middle.next;
        middle.next = null; // Split into two sublists

        // Recursively sort both halves
        Node left = mergeSort(h);
        Node right = mergeSort(nextOfMiddle);

        // Merge the two sorted halves
        return sortedMerge(left, right);
    }

    /**
     * Merges two sorted linked lists into one (sorted by last name).
     */
    private Node sortedMerge(Node a, Node b) {
        if (a == null) return b;
        if (b == null) return a;

        Node result;

        // Compare by last name (case-insensitive)
        if (a.lastName.compareToIgnoreCase(b.lastName) <= 0) {
            result = a;
            result.next = sortedMerge(a.next, b);
        } else {
            result = b;
            result.next = sortedMerge(a, b.next);
        }
        return result;
    }

    /**
     * Finds the middle node of a linked list using the fast/
     */
    private Node getMiddle(Node h) {
        if (h == null) return h;

        Node slow = h, fast = h;
        while (fast.next != null && fast.next.next != null) {
            slow = slow.next;
            fast = fast.next.next;
        }
        return slow;
    }

    /**
     * Displays the contents of the linked list.
     */
    public void displayList() {
        Node current = head;
        while (current != null) {
            System.out.printf("% -10s % -10s % -10s%n",
                    current.firstName, current.lastName, curr
            current = current.next;
        }
        System.out.println();
    }
}

// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Iterative Merge Sort Implementation
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

/**
 * MergeSortLinkedList_Iterative
 *
 * Uses an iterative (bottom-up) merge sort approach.
 * Starts by merging small sorted sublists of size 1, then do
 * the size of the sublists (1, 2, 4, 8...) until the full li
 */
class MergeSortLinkedList_Iterative implements Sorter {

    /**
     * Inner class Node - represents a single record in the linked list.
     */
    class Node {
        String firstName;
```

```
    String lastName;
    String city;
    Node next;

    Node(String firstName, String lastName, String city) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.city = city;
        this.next = null;
    }
}

private Node head; // Head of the linked list

/**
 * Inserts a new node at the end of the list.
 */
public void insert(String firstName, String lastName, String city) {
    Node newNode = new Node(firstName, lastName, city);
    if (head == null) {
        head = newNode;
        return;
    }
    Node current = head;
    while (current.next != null)
        current = current.next;
    current.next = newNode;
}

/**
 * Public method to start iterative merge sort.
 */
public void sort() {
    head = mergeSortIterative(head);
}

/**
 * Iterative (loop-based) merge sort implementation.
 * Uses sublist merging of increasing size to avoid recursion.
 */
private Node mergeSortIterative(Node head) {
    if (head == null || head.next == null) return head;

    int n = getSize(head);

    // Dummy node simplifies pointer management during merging
    Node dummy = new Node("", "", "");
    dummy.next = head;

    // Merge sublists of size 1, 2, 4, 8, etc.
    for (int step = 1; step < n; step *= 2) {
        Node prev = dummy;
        Node current = dummy.next;

        while (current != null) {
            Node left = current;
            Node right = split(left, step);
            current = split(right, step);
            Node merged = sortedMerge(left, right);
            prev.next = merged;

            // Move 'prev' to the end of the merged sublis
            while (prev.next != null)
                prev = prev.next;
        }
    }

    return dummy.next;
}

/**
 * Splits the list after 'size' nodes and returns the nex
 */
private Node split(Node head, int size) {
    if (head == null) return null;
    for (int i = 1; head.next != null && i < size; i++)
        head = head.next;

    Node second = head.next;
    head.next = null;
    return second;
}

/**
 * Merges two sorted linked lists (by last name).
 */
private Node sortedMerge(Node a, Node b) {
    Node dummy = new Node("", "", "");
    Node tail = dummy;

    while (a != null && b != null) {
        if (a.lastName.compareToIgnoreCase(b.lastName) <=
            tail.next = a;
            a = a.next;
        } else {
            tail.next = b;
            b = b.next;
        }
        tail = tail.next;
    }

    tail.next = (a != null) ? a : b;
    return dummy.next;
}

/**
 * Counts the number of nodes in the list.
 */
private int getSize(Node head) {
    int count = 0;
    while (head != null) {
        count++;
        head = head.next;
    }
    return count;
```

```
    }

    /**
     * Displays the contents of the linked list.
     */
    public void displayList() {
        Node current = head;
        while (current != null) {
            System.out.printf("% -10s % -10s % -10s%n",
                    current.firstName, current.lastName, current.city);
            current = current.next;
        }
        System.out.println();
    }
}

// ------------------------------------------------------------------
// Driver Class
// ------------------------------------------------------------------

/**
 * Main driver class.
 *
 * Demonstrates the use of both recursive and iterative
 * linked list merge sort implementations via the Sorter interface.
 */
public class MergeSortLinkedList {

    public static void main(String[] args) {

        Sorter list;

        int choice = 1;  // 1 = Recursive, 2 = Iterative

        if (choice == 1)
            list = new MergeSortLinkedList_Recursive();
        else
            list = new MergeSortLinkedList_Iterative();

        // Insert sample data
        list.insert("Alice", "Zimmer", "Chicago");
        list.insert("Bob", "Anderson", "Kalispell");
        list.insert("Cathy", "Johnson", "Seattle");
        list.insert("Daniel", "Brown", "Denver");

        // Display before sorting
        System.out.println("Before Sorting:");
        list.displayList();

        // Sort and display results
        list.sort();
        System.out.println("After Sorting by Last Name:");
        list.displayList();
    }
}
```

When To Use

Quick summary

- **Both** are `O(n log n)` time, **stable**, and `O(1)` extra space on the list (recursive adds `O(log n)` call-stack space).
- **Recursive** = simpler to read/teach; **Iterative** = no recursion, often better constants and safer for huge inputs.

Strengths vs. weaknesses

Recursive (top-down) **How it works:** repeatedly split with fast/slow pointers (`getMiddle`), then merge.

Strengths

- **Clarity & pedagogy:** mirrors the textbook definition; very readable for students.
- **Natural structure:** the "divide → *conquer* → *combine″ flowmatchesthementalmodelofmergesort.*"
- **Easy to parallelize:** left and right halves can be sorted concurrently if you go multi-threaded later.
- **Stable by construction:** merge step preserves order of equals.

Weaknesses

- **Call-stack use:** `O(log n)` stack frames. Usually fine, but it's still extra memory and can matter on very tight stacks or unusual environments.
- **Function-call overhead:** many small recursive calls; minor but measurable on some JVMs.
- **Middle finding every level:** each split uses fast/slow pointers; total cost stays `O(n log n)` but the constant factor isn't free.

Iterative (bottom-up) **How it works:** repeatedly merge runs of size `1, 2, 4, 8, …` using loops; uses pointer "splicing" (`split`, `merge`) and a dummy head.

Strengths

- **No recursion:** avoids stack growth entirely; safer for **very large lists** or constrained runtimes.
- **Good constants:** one linear pass per run size; practical throughput often edges out recursive on linked lists.
- **Predictable control flow:** all in loops; easy to bound and instrument.
- **Still stable:** merge loop preserves order of equals.

Weaknesses

- **More pointer surgery:** more places to make off-by-one / null-next mistakes; trickier to get right first time.
- **Slightly less intuitive to newcomers:** bottom-up run doubling is less obvious than "split in half".
- **Needs size or tail walking:** typical pattern computes `n` up front or advances pointers carefully; again, more book-keeping.

Performance & resource notes

- **Time complexity:** both `O(n log n)` on singly linked lists.
- **Space:** both in-place on nodes; **recursive adds `O(log n)` call stack**, iterative doesn't.
- **Cache locality:** neither is great (linked lists aren't cache-friendly), but iterative's fewer function calls can help a bit.
- **Stability:** both remain stable as long as your merge uses `<=` (or equivalent) and never reorders equals.

When to pick which

- **Teaching / readability / quick correctness: Recursive**.
- **Production on huge lists / tight memory / maximum robustness: Iterative**.
- **Parallel sort of very large lists: Recursive** lends itself to parallelizing the two halves.

Practical checklist

- Need simple code? → **Recursive**.
- Worried about stack or sorting millions of nodes? → **Iterative**.

1.8.5 *Quick Sort*

Quicksort is a highly efficient sorting algorithm that utilizes the divide-and-conquer strategy to recursively partition an array into smaller subarrays until the entire array is sorted. It's a versatile algorithm applicable to various data types and input conditions, making it a popular choice for both educational and practical purposes.

Here's a step-by-step breakdown of how quicksort works:

1. **Pivot Selection:** Choose an element from the array as the pivot. This pivot will serve as a reference point for partitioning the array.
2. **Partitioning:** Partition the array around the pivot, placing elements smaller than the pivot to its left and elements larger than the pivot to its right. This process involves moving elements from one end of the array to the other until the pivot is at its correct position in the sorted array.
3. **Recursion:** Recursively apply the partitioning process to the two subarrays created in the previous step. This means treating each subarray as a new unsorted array and repeating the pivot selection and partitioning steps until all subarrays are sorted.
4. **Base Case:** The recursion stops when a subarray contains only one element, as it is already sorted.

The efficiency of quicksort stems from its ability to divide the array into smaller and smaller subarrays, reducing the overall number of comparisons required for sorting. However, its worst-case performance is O(n2), which occurs when the pivot selection consistently leads to unbalanced subarrays.

In summary, quicksort is a fast and effective sorting algorithm that works by recursively partitioning an array around a pivot element until the entire array is sorted. Its efficiency and simplicity make it a popular choice for various sorting tasks.

Here is a list of the pros and cons of quicksort in bullet points:

Pros:

- **Efficiency:** Quicksort has an average time complexity of O(n log n), which is one of the best among sorting algorithms.
- **Simplicity:** Quicksort is relatively easy to understand and implement, making it a popular choice for teaching and practical applications.
- **In-place:** Quicksort is an in-place sorting algorithm, meaning it does not require additional memory to store intermediate results.
- **Adaptability:** Quicksort can be adapted to handle various data types and input conditions.

Cons:

- **Worst-case performance:** Quicksort has a worst-case time complexity of O(n2), which occurs when the pivot selection is consistently poor.
- **Instability:** Quicksort is unstable, meaning it does not preserve the original order of equal elements.
- **Memory usage:** Quicksort may require a significant amount of stack space for recursion, especially for large datasets.
- **Sensitivity to pivot selection:** The performance of quicksort is heavily dependent on the choice of pivot, making it susceptible to variations in input data.

Overall, quicksort is a versatile and efficient sorting algorithm with a proven track record in various applications. However, it is essential to consider its potential drawbacks, such as worst-case performance and instability, when selecting an appropriate sorting algorithm for a specific task.

1.8.6 *End Of Section*

Algorithms and Data Structures

End Of Section

Copyright notice