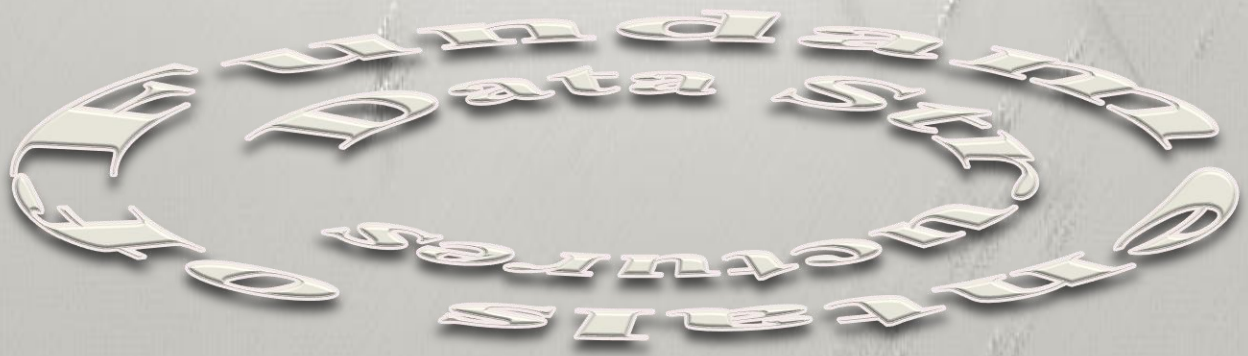
Fundamentals of Data Structures

Using Java

Manual
Second
Edition



Mohd Nizam bin Osman

Preface

This manual is designed for students who take a course of the Fundamentals of Data Structure (CSC248) using Java Programming Language. This manual is a guide to help a student in understanding the basic concepts of information organization in data structure such as the list, stack, queue, recursion and tree. It contains an example and rich in information about the concepts of data structure. Besides, it easy-to-follow by the example given for each topic discussed, which is designed to get you started. Furthermore, at the end of each topic, student will be given past year sample exam questions with the suggested answer.

To be excellent in this course, student must understand the concepts for each topic. Hence, this manual provides the details explanation by given an analogy to the real application in our life with the concept of data structure. Then, students are taught on how to transfer the concept that you have understood into coding using Java Programming Language.

This course is the continuation from the Object-Oriented Programming subject. At this level, student must be familiar or excelled with Java's syntax. Please to be reminded, that this manual just a guideline for your study, you must attend to the classroom in order to solid understanding by a detail explanation from your lecturer. So, I wish to all my students let enjoy the subjects and good luck.

To be excellence in a programming language you have to practice, practice and practice……..and never give up."

Mr. Mohd Nizam bin Osman
Department of Computer Science
College of Computing, Informatics and Mathematics

Fundamentals of Data Structures
By Mr. Mohd Nizam bin Osman
e-mail: mohdnizam@uitm.edu.my

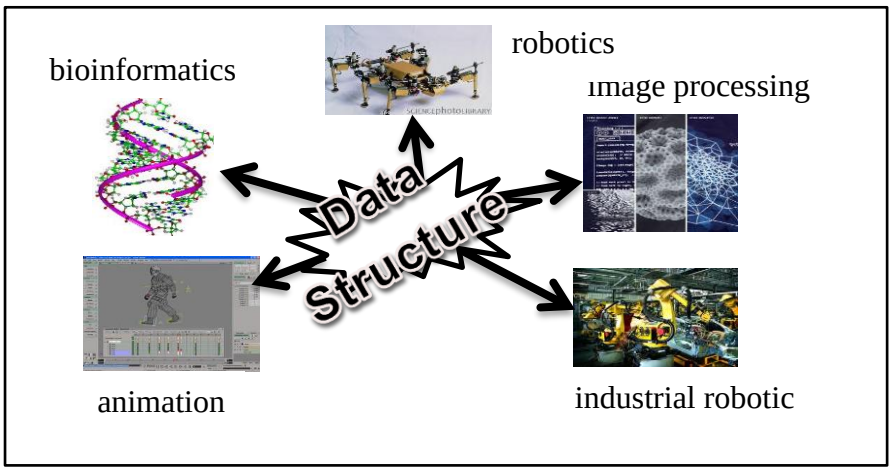
CONTENTS

Introduction to Data Structure and Abstract Data Type

CHAPTER 1

1.1 INTRODUCTION

- Representing information is a fundamental to the computer science field. The primary purpose of most computer programs is not to perform calculations, but to store and retrieve information — usually as fast as possible.
- For this reason, the study of data structures and the algorithms that manipulate them is at the heart of computer science.
- The data structures are an important way of organizing information in a computer. There are many different data structures that programmers use to organize data in computers such as the list, stack and queue.
- The following figure shows the uses of data structure to organize the data (normally, it will involve huge/large data) in many fields such as bioinformatics, robotics, animation and many other fields.

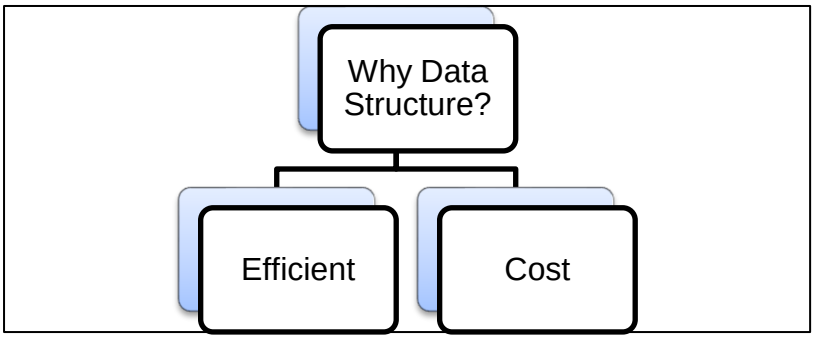


The uses of data structure

- Each data structure has their own unique properties that make it well suited to give a certain view of the data.

1.2 A PHILOSOPHY OF DATA STRUCTURE

- You might think that with ever more powerful computers, program efficiency is becoming less important.
- After all, processor speed and memory size still seem to double every couple of years. Won't any efficiency problem we might have today to be solved by tomorrow's hardware?
- Therefore, we need to organize the data in a proper way to make its effective and efficient by implementing a suitable data structure approach.
- The following figure shows the need of data structure in order to manipulate the data.



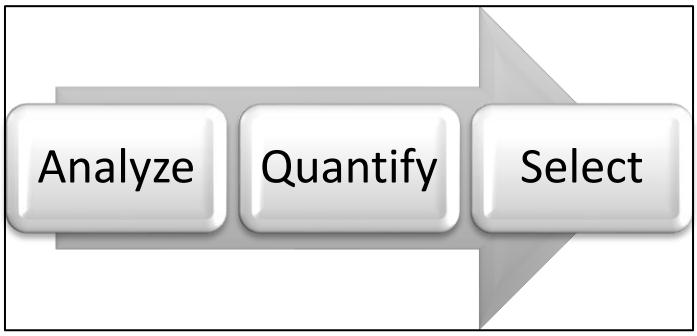
The need of data structure

1.3 THE NEED FOR DATA STRUCTURE

- A solution is said to be efficient if it solves the problem within the required resource constraints.
- Examples of resource constraints include the total space available to store the data:
  - Possibly divided into separate main memory and disk space constraints.
  - The time allowed to perform each subtask.

- The cost of a solution is the amount of resources that the solution consumes.
- It should go without saying that people write programs to solve problems. However, it is crucial to keep this truism in mind when selecting a data structure to solve a particular problem.

- Only by first analyzing the problem to determine the performance goals that must be achieved can there be any hope of selecting the right data structure for the job.
- Poor program designers ignore this analysis step and apply a data structure that they are familiar with, but which is inappropriate to the problem. The result is typically a slow program.
- When selecting a data structure to solve a problem, you should follow these steps as shown in the diagram.



Steps to select a data structure

1. Analyze your problem to determine the basic operations that must be supported. Examples of basic operations include inserting a data item into the data structure, deleting a data item from the data structure, and finding a specified data item.
2. Quantify the resource constraints for each operation.
3. Select the data structure that best meets these requirements.

- Each data structure has associated costs and benefits.
- A data structure requires a certain amount of space for each data item it stores, a certain amount of time to perform a single basic operation, and a certain amount of programming effort. Each problem has constraints on available space and time.
- Each solution to a problem makes use of the basic operations in some relative proportion, and the data structure selection process must account for this.
- Only after a careful analysis of your problem's characteristics, then you can determine the best data structure for the task.

1.4 TERMINOLOGY

- This section presents terminology and motivates the design process embodied in the three-step approach to selecting a data structure. This motivation stems from the need to manage the tremendous complexity of computer programs.
- The following table represents the terminology use in this course in order to discuss the data structure.

Terminology	Description	Example
Type	a collection of values	**Simple type:** Boolean, Integer **Aggregate/Composite:** A bank account record – contain several pieces information such as name, address, account number and account balance.
Data item	a piece of information or a record whose value is drawn from a type. A data item is said to be a member of a type.	
Data type	Is a type together with a collection of operations to manipulate the type.	An integer variable is a member of the integer data type. Addition is an example of an operation on the integer data type

Abstract Data Type (ADT)	the realization of a data type as a software component.	`class List{`
		`  public Object data;`
		`  void insert(Object o);`
		`  Object remove();`
		`}`

Member function	each operation associated with the ADT is implemented by a member function or method	`void insert(Object o);`
Data member	The variables that define the space required by a data item.	`Object data;`
Class	used to represent ADT	
Object	object is an instance of a class, which defines the type of the object, as well as the kinds of operations that it performs.	
Data Structure	Is the implementation of an ADT	

Terminology use in data structure

1.5 ABSTRACT DATA TYPE CONCEPT

- One of the goals of software engineering is to write reusable code, which is code that can be reused in many different applications, preferably without having to be recompiled.
- One way to make code reusable is to encapsulate or combine data elements together with methods that operate on that data in separate program module (a class).
- A new program can use the methods to manipulate the data without being concerned about the details of the data representation and method implementations.
- In this way, class can be used as a building block to construct a new application program.
- The combination of data together with its methods is called an Abstract Data Type (ADT).
- An Abstract Data Type (ADT) consists of:
  - A mathematical model of the data.
  - Methods for accessing and modifying the data.

Abstract Data Type (ADT)
Is a well-specified collection of data and a group of operation that can be performed upon the data.

- The ADT's specification describes what data can be stored (the characteristics of the ADT), and how it can be used (the operations).
- The interface of the ADT is defined in terms of a type and a set of operations on that type.
- The behavior of each operation is determined by its inputs and outputs.
- An ADT does not specify how the data type is implemented. These implementation details are hidden from the user of the ADT and protected from outside access, a concept referred to as encapsulation.

- The following figure shows a diagram of an abstract type.

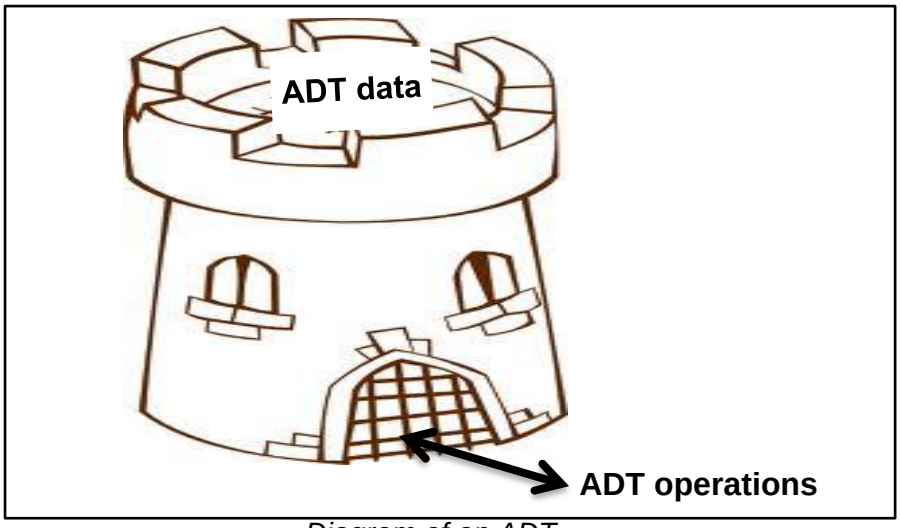


Diagram of an ADT

  - The data values stored in the ADT are hidden inside the circular wall.
  - The bricks around this wall are used to indicate that these data values cannot be accessed except by going through the ADT's methods.

- A data structure is the implementation of an ADT. In an object-oriented language such as Java, an ADT is implemented by a member function or method. The variables that define the space required by a data item are referred to as data members.
- An object is an instance of a class, that is, something that is created and takes up storage during the execution of a computer program.
- The term "data structure" often refers to data stored in computer's main memory.
- The ADT concept is a useful aid in the software design process.
- For example, If you need to store data,
  - Start by considering the operations that need to be performed on that data.
    - Do you need access to the last item inserted? The first one? An item with a specified key? An item in a certain position?
  - Answering such questions leads to the definition of an ADT.

- Only after the ADT is completely defined should you worry about the details of how to represent the data and how to code the methods that access the data.
- By decoupling the specification of the ADT from the implementation details, you can simplify the design process.
- You also make it easier to change the implementation at some future time.
- If a user relates only to the ADT interface, you should be able to change the implementation without “breaking” the user’s code.
- Of course, once the ADT has been designed, the underlying data structure must be carefully chosen to make the specified operations as efficient as possible.
- If you need random access to element N, for example, the linked-list representation isn’t so good because random access isn’t an efficient operation for a linked list. You’d be better off with an array.

EXAMPLE OF PROGRAM 1 - ADT

The ADT: To calculate area and perimeter for object 2D.

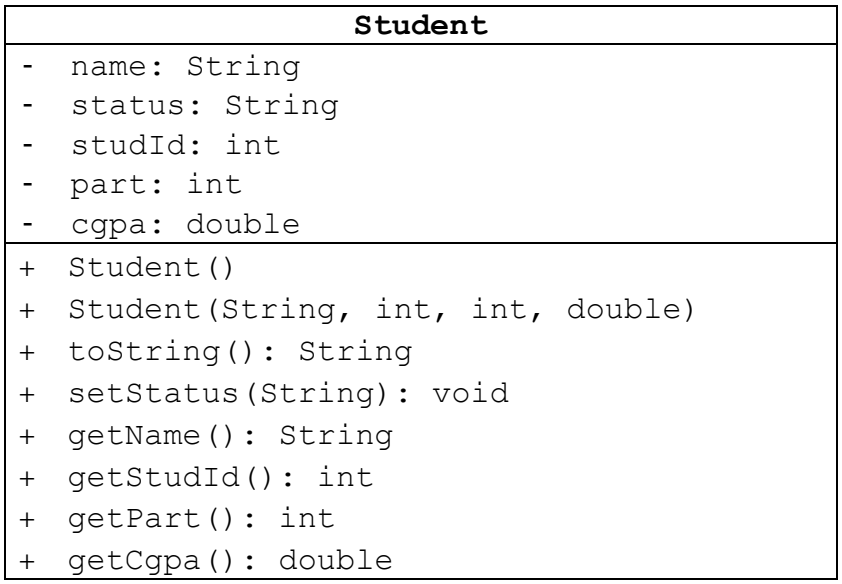
TwoD
- area: double - perimeter: double
+ getArea(): double + getPerimeter(): double + calculateArea(): abstract double + calculatePerimeter(): abstract double + toString(): abstract String

UML diagram of the the ADT for class TwoD

```
//Class declaration and definition for ADT
public abstract class TwoD
{
    protected double area;
    protected double perimeter;

    //Constructor
    public TwoD()
    {
       area=0;
       perimeter=0;
    }

    //Getter methods
    public double getArea(){return area;}
    public double getPerimeter(){return perimeter;}

    //Processor methods
    abstract public double calculateArea();
    abstract public double calculatePerimeter();

   //Printer methods
    abstract public String toString();
}
```

EXAMPLE OF PROGRAM 2 - ADT

The ADT: to determine the highest cgpa among students and to calculate the average of cgpa.



UML diagram of class Student

```
public class Student extends Object {
      private String name,status;
      private int studId;
      private int part;
      private double cgpa;

      public Student ()
      {
            name = "";
            studId = -1;
            part = -1;
            cgpa = -1;
            status = "";
      }

      public Student (String nm, int sid, int pt, double cg)
      {
           name = nm;
           studId = sid;
           part = pt;
           cgpa = cg;
      }
```

```
      public String toString ()
      {
            return "\nName : " + name + "\nStudent ID :  "
              + studId + "\nPart : " + part + "\nCGPA : " + cgpa
              + "\nStatus : " + status;
      }

      public void setStatus(String stat)    { status = stat; }
      public String getName()     { return name; }
      public int getStudId()      { return studId; }
      public int getPart()        { return part; }
      public double getCgpa()     { return cgpa; }
}

public class StudentApp{
      public static void main (String[] args)
      {
         Scanner scan = new Scanner(System.in);
         Scanner scan1 = new Scanner(System.in);

         //Declare array of object
         Student[] DCS = new Student[5];

         //Input process
         for (int i=0;i<5;i++)
         {
             System.out.print("Please enter name: ");
             String name = scan.nextLine();
             System.out.print("Please enter student id: ");
             int sid = scan1.nextInt()
             System.out.print("Please enter part: ");
             int part = scan1.nextInt();
             System.out.print("Please enter cgpa: ");
             double cgpa = scan1.nextDouble();

             //Store onto object
             DCS[i] = new Student (name,sid,part,cgpa);
         }

         //Manipulation: set status, find the highest cgpa and
         //calculate average cgpa
         double highcgpa = -1;
         int total = 0;
         Student highStud = new Student();

         for (int i=0;i<5;i++)
         {
             //set status
             if (DCS[i].getCgpa() > 3.5 )
                  DCS[i].setStatus ("Dean's List");
             else if (DCS[i].getCgpa() >= 2.0)
                  DCS[i].setStatus ("Pass");
             else
                  DCS[i].setStatus ("Fail");
```

```
            //identify the highest cgpa
            if (DCS[i].getCgpa() > highcgpa)
            {    highcgpa = DCS[i].getCgpa();
                 highStud = DCS[i];
            }
            total += DCS[i].getCgpa();
         }//end for

         System.out.println ("Student with
                highest CGPA"+ highStud.toString());
        System.out.println("Average CGPA
                among students: " +total/5);
       }//end main
}//end class
```

1.6 DATA STRUCTURE

- A data structure is an arrangement of data in a computer's memory (or sometimes on a disk).
- Data may be organized in many different ways: the logical or mathematical model of a particular organization of data is called a data structure.
- Data structures include arrays, linked list, stack, queue, tree and graph as follows:

Data Structure	Description
Array	The simplest type of data structure is a linear (or one dimensional) array. By a linear array, we mean a list of a finite number n of similar data elements referenced respectively by a set of n consecutive numbers.
Stack	A stack, also called a last-in-first-out (LIFO) system, is a linear list in which items may be inserted or removed only at one end called the top of the stack
Queue	A queue, also called a first-in-first-out (FIFO) system, is a linear list in which deletions can take place only at one end of the list, the "front" of the list, and insertions can take place only at the other end of the list, the "rear" of the list.
Tree	A tree is an acyclic, connected graph.
Graph	Graphs are classified in the non-linear category of data structures.

Category of data structure

- Algorithms manipulate the data in these structures in various ways, such as searching for a particular data item and sorting the data.
- A data structure realizing an ADT consists of:
  - Collections of variables for storing the data described by the mathematical model underlying the ADT.
  - Algorithms for the methods of the ADT.

Data Structure
Consists of a base storage method (e.g., an array) and one or more algorithms that are used to access or modify that data.

1.7 THE WAY TO SPECIFY DATA

- The following criteria should be considered when we are intense to choose a data:

(i) Abstract.

It should abstract the crucial features of the data without forcing the programmer to focus on implementation details, thus making the code (and design) easier to understand and maintain.

(ii) Safe.

It should allow control over the manipulation of the data representation so that errors can be prevented.

(iii) Modifiable.

It should make it relatively easy for modifications in the representation to be made.

(iv) Reusable.

It ought to be possible to reuse the representation and its implementation in other code.

1.8 APPLICATION OF DATA STRUCTURE

- Several types of applications of data structure that you will have learned in this course as follows:
  (i) Linked list – are collections of data items “line up in row” :- insertion and deletion are made at anywhere.
  (ii) Stack – First In Last Out :- insertion and deletion made only at one end of a stack.
  (iii) Queue – First In First Out :- insertion are at the back and deletion are made from the front.
  (iv) Binary Trees – facilitate high-speed searching and sorting of data, efficient elimination of duplicate data items, representing file system directories.

1.9 ADVANTAGES AND DISADVANTAGES OF DATA STRUCTURE

- Another way to look at data structure is to focus on their strengths and weaknesses.
- The following table shows the advantages and disadvantages of the various data structures.

Data Structure	Advantages	Disadvantages
Array	Quick insertion, very fast access if index known.	Slow search, slow deletion, fixed size.
Ordered array	Quicker search than unsorted array.	Slow insertion and deletion, fixed size.
Stack	Provides last-in, first out access.	Slow access to other items.
Queue	Provides first-in, first-out access.	Slow access to other items.
Linked list	Quick insertion, quick deletion	Slow search.
Binary tree	Quick search, insertion, deletion (if tree remains balanced).	Deletion algorithm is complex.
Red-black tree	Quick search, insertion, deletion. Tree always balanced.	Complex.

2-3-4 tree	Quick search, insertion, deletion. Tree always balanced. Similar trees good for disk storage.	Complex.
Hash table	Very fast access if key known. Fast insertion	Slow deletion, access slow if key not known, the inefficient memory usage.
Heap	Fast insertion, deletion, access to largest item.	Slow access to other items.
Graph	Models real-world situations.	Some algorithms are slow and complex.

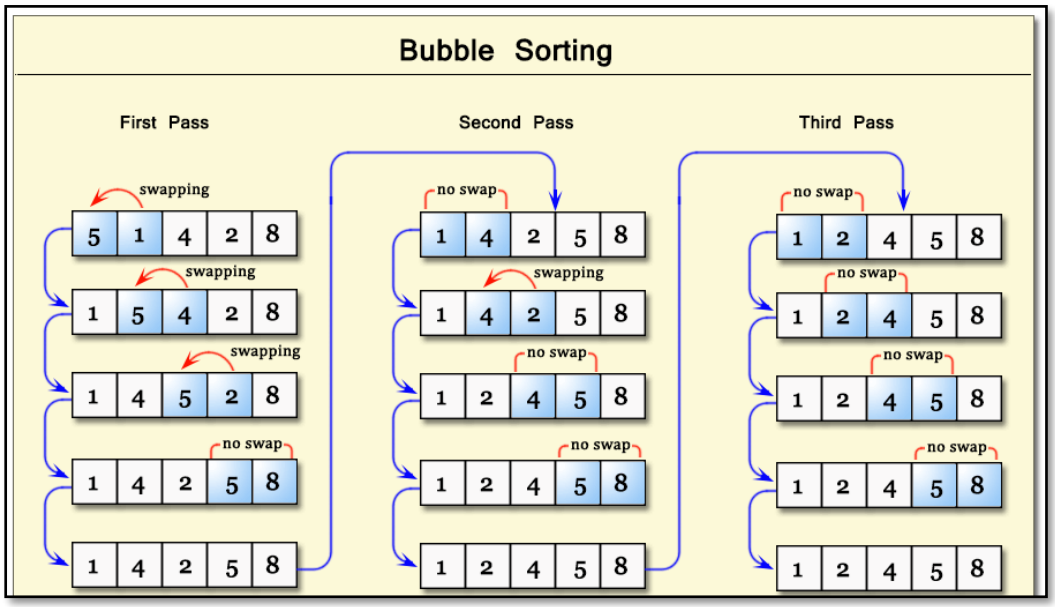
Advantages and disadvantages of data structure

1.10 BASIC ALGORITHMS – SORTING AND SEARCHING

- In this section we will study a simple algorithm for sorting and searching. There are many type of sorting and searching algorithm. For instance, sorting algorithm:- selection sort, merge sort, buble sort, insertion sort and searching algorithm:- linear seacr, binary search, jump search.
- Sorting is a process of arranging the data in acending or descending order. Meanwhile, searching is a process of finding a particular item in a collection of items.
- There are so many things in our real life that we need to search for, like a particular record in database, roll numbers in merit list, a particular telephone number in telephone directory, a particular page in a book, etc. All this would have been a mess if data was kept unordered and unsorted, but fortunately the concept of sorting come into existence, making it easier for everyone to arrange data in an order, hence making it easier to search.
- Sorting arranges data in a sequence which makes searching easier.
- This section discusses on the following sorting and searching algorithm:

i) **Bubble Sort**

- Bibble sort is the simplest sorting algorithm that works by repeteadly swapping the adjacent elements if they in wrong order.
- The bubble sort makes multiple passes through a list.
- It compares adjacent items and exchanges those that are out of order.
- Each pass through the list places the next largest value in ots proper place.
- For example:

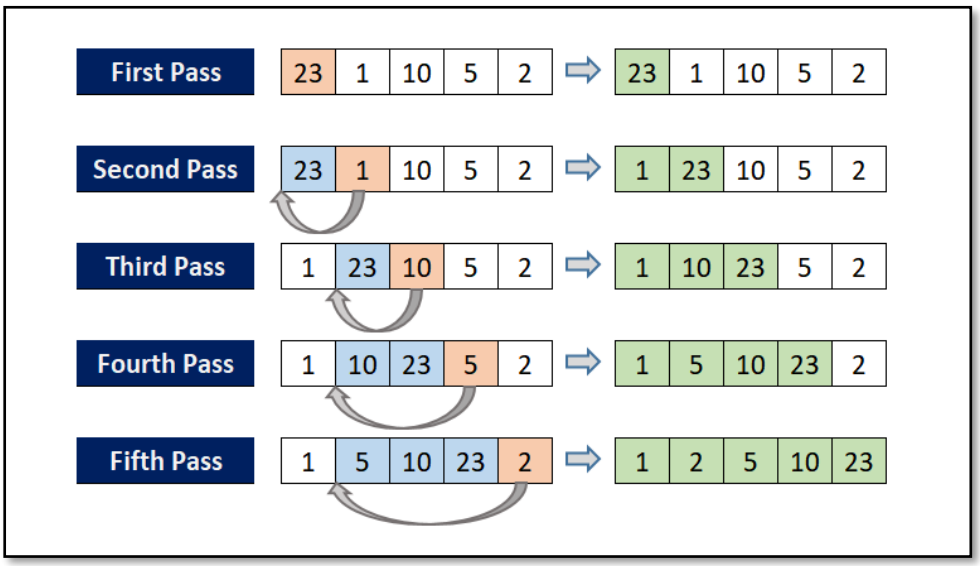


EXAMPLE OF PROGRAM - BUBBLE SORT

```
// Sort an array of integers in ascending order
void bubble(int data[], int size) {//an array and size of array
    int temp;       // for swap
    for(int outer=size-1;  outer > 0; outer--){
       for (int inner=0; inner < outer; inner++) {
            // traverse the nested loops
            if ( data[inner] > data[inner+1] ) {
                 // swap current element with next
                 // if the current element is greater
                 temp = data[inner];
                 data[inner] = data[inner+1];
                 data[inner+1] = temp;
            }
       }  // inner for loop
    }  // outer for loop
}
```

ii) **Insertion Sort**

- This method is very similar to what one does in preparing to play a game of cards. One receives cards one at a time and orders them in the hand.As each new card arrives, the player scans his hand, generally left-to-right, searching for the correct place for the new arrival, then inserts the arrival in that place.
- Insertion sort is based on the idea that one element from the input elements is consumed in each iteration to find its correct position (the position to which it belongs in a sorted array)
- For example:

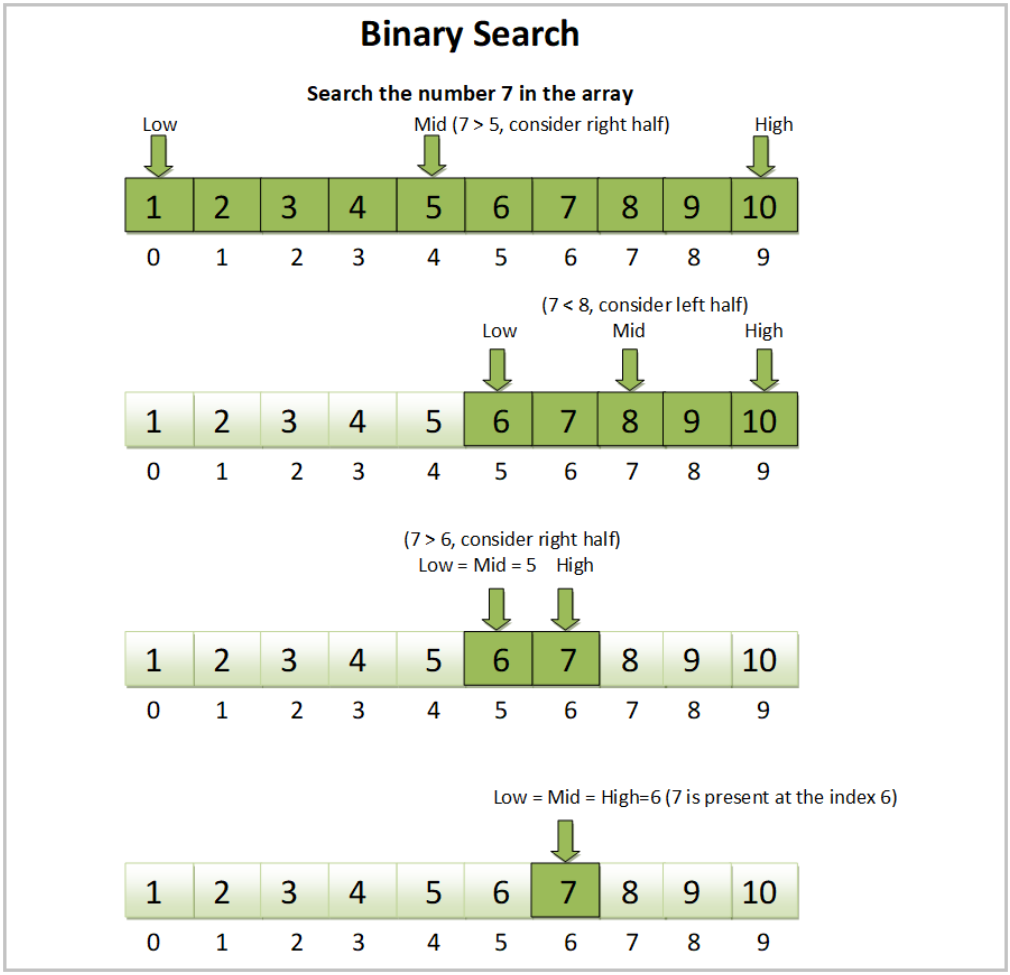


EXAMPLE OF PROGRAM

```
void sort(int arr[])
 {
    int n = arr.length;
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;

        /* Move elements of arr[0..i-1], that are
           greater than key, to one position ahead
           of their current position */
         while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];
                j = j - 1;
         }
         arr[j + 1] = key;
     }
 }
```

iii) **Binary Search**

- In the binary search method, the collection is repeatedly divided into half and the key element is searched in the left or right half of the collection depending on whether the key is less than or greater than the mid element of the collection.
- For example:



EXAMPLE OF PROGRAM

```
import java.util.*;
public class BinarySearch{
 public static void main(String args[]){
    int numArray[] = {5,10,15,20,25,30,35};
    System.out.println("The input array:"
        + Arrays.toString(numArray));
    //key to be searched
    int key = 20;
    System.out.println("\nKey to be searched=" + key);
    //set first to first index
    int first = 0;
    //set last to last elements in array
    int last=numArray.length-1;
    //calculate mid of the array
    int mid = (first + last)/2;
    //while first and last do not overlap
    while( first <= last ){
      //if the mid < key, then key to be searched is in the
      //first half of array
      if ( numArray[mid] < key ){
          first = mid + 1;
      }
      else if ( numArray[mid] == key ){
          //if key = element at mid, then print the location
          System.out.println("Element is found at index: "
             + mid);
          break;
      }
      else{
          //the key is to be searched in the second half
          //of the array
            last = mid - 1;
      }
      mid = (first + last)/2;
   }
   //if first and last overlap, then key is not present in
   //the array
   if ( first > last ){
      System.out.println("Element is not found!");
   }
 }//end main
} //end class
```

1.11 IMPLEMENTATION OF GENERIC CLASSES

- So far, students have learned and created several variations of a list ADTs. For example, a “standalone” unsorted string list or a sorted string list method which is only string value can be considered.
- It would be nice if we could write a single sort method that could sort the elements in an Integer array, a String array or an array of any type that supports ordering.
- Java Generic methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods or, with a single class declaration, a set of related types, respectively.
- Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.
- Using Java Generic concept we might write a generic method for sorting an array of objects, then invoke the generic method with Integer arrays, Double arrays, String arrays and so on, to sort the array elements.

GENERIC METHODS

- You can write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately.
- The following are the rules to define Generic Methods:
    1. All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type (< E > in the next example).
    2. Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.
    3. The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.
    4. A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types not primitive types (like int, double and char).

- Following example illustrate how we can print array of different type using a single Generic method:

```
public class GenericMethodTest
{
   // generic method printArray
   public static < E > void printArray( E[] inputArray )
   {
      // Display array elements
         for ( E element : inputArray ){
            System.out.printf( "%s ", element );
         }
         System.out.println();
    }

    public static void main( String args[] )
    {
        // Create arrays of Integer, Double and Character
        Integer[] intArray = { 1, 2, 3, 4, 5 };
        Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
        Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

        System.out.println( "Array integerArray contains:" );
        printArray( intArray  ); // pass an Integer array

        System.out.println( "\nArray doubleArray contains:" );
        printArray( doubleArray ); // pass a Double array

        System.out.println("Array characterArray contains:" );
        printArray( charArray ); // pass a Character array
    }
}
```

Generic method

GENERIC CLASS

- A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.
- As with generic methods, the type parameter section of a generic class can have one or more type parameters separated by commas. These classes are known as parameterized classes or parameterized types because they accept one or more parameters.
- Following example illustrate how we can define a generic class:

```
public class Box<T> {

  private T t;

  public void add(T t) {
    this.t = t;
  }

  public T get() {
    return t;
  }

  public static void main(String[] args) {
     Box<Integer> integerBox = new Box<Integer>();
     Box<String> stringBox = new Box<String>();

     integerBox.add(new Integer(10));
     stringBox.add(new String("Hello World"));

     System.out.printf("Integer Value :%d\n\n",
                       integerBox.get());
     System.out.printf("String Value :%s\n", stringBox.get());
  }
}
```

Generic class

CHAPTER 2

List

2.1 INTRODUCTION

- If your program needs to store a few things – numbers, payroll records, or job descriptions for example – the simplest and most effective approach might be to put them in a list.
- A list has the property that elements can be inserted or removed anywhere in the list, not just at the beginning or at the end.
- Some lists are indexed, which means their elements can be accessed in arbitrary order (called random access) using subscript to select an element.
- For other lists you must start at the beginning and process the element in the sequence.
- Only when you have to organize or search through a large number of things do more sophisticated data structures usually become necessary.

List
A collection of finite elements of the same type.

- Each list element has the same data type (homogenous).
- The basic operation on the list is based on the application of the problem.
- It normally consists of:
  - (i) Create a new list. The list is initialized to an empty state.
  - (ii) Determine whether the list is empty.
  - (iii) Determine whether the list is full.
  - (iv) Find the size of the list.
  - (v) Clear the list.
  - (vi) Determine whether an item is the same as given in the list.
  - (vii) Insert an item in the list at the specified location.
  - (viii) Remove an item from the list at the specified location.

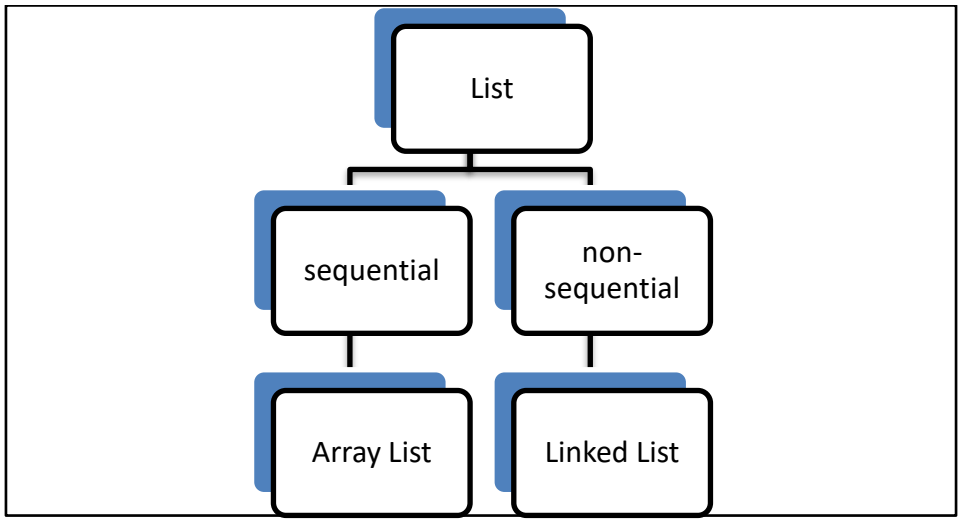
(ix) Replace an item at the specified location with another item

(x) Retrieve an item from the list from the specified location.

(xi) Search the list

(xii) Insert new element into the list for a given item.

- The beginning of the list is called the **head/front** and the end of the list is called the **tail/rear.**
- The list can be categorized into two types; sequential and non-sequential as shown in the following figure:



Types of List

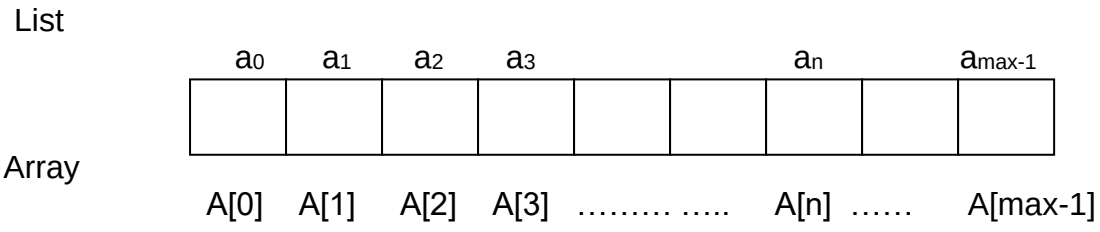
2.2 LIST IMPLEMENTATION

- There are two standard approaches to implementing list, the array-based list, and the linked list.
- When we define for the sequential list, the using of an array is suitable to implement the list. Meanwhile, for the non-sequential list, the use of linked list is preferable.
- An array is an indexed data structure, which means you can select its elements in arbitrary order as determined by the subscript value.
  - ➢ You can also access the elemets in sequence using a loop that increments the subscripts.

- However, you can't do the following with an array object:
  - Increase or decrease its length, which is fixed.
  - Add an element at a specified position without shifting the other elements to make room.
  - Remove an element at a specified position without shifting the other element to fill in the resulting gap.

- Meanwhile, the linked list uses dynamic memory allocation, that is, it allocates memory for new list elements as needed.
  - You are allowed to increase and decrease its length at any time.
  - You are allowed to add and remove an element at specified position without shifting the other element.

- Element in the list is **homogenous** and the size of an element has always changed – depend on insertion and deletion element in the list.
- The first element in the list – head/front.
- The last element in the list – tail/rear.
- The following figure shows the diagram to represent sequential list using an array:



- To create a new list, determine empty list and traverse the list, it is quite easy:

(i) Create a new list

- We need to define an array variable type.
- We need variables to count the element in the list.

(ii) Empty List

- We need to test the variables that count the number of elements in the list. If the value of count is zero – empty list.

(iii) Traversal list

- It can be done by using iteration statement such as for/while statement.

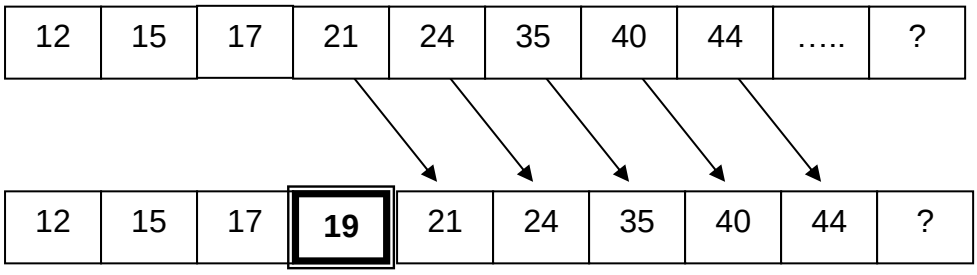
- For insertion and deletion operation, it is quite difficult because it depends on the types or characteristics of list elements such as when we have to consider a list in ascending or descending order. This is an important factor when we need to insert new element or remove any element from the list in order to maintain its characteristic.

- **Example:**

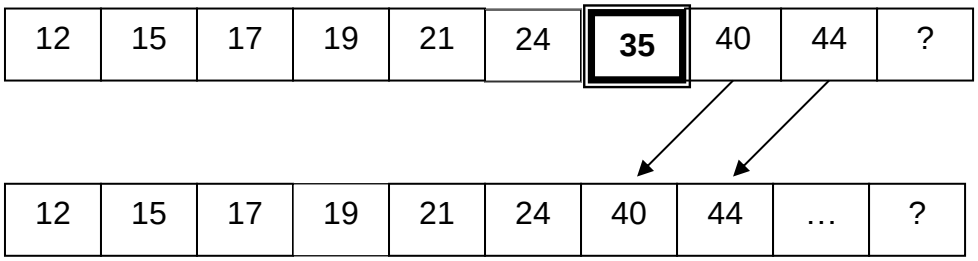
  Consider a list with a sorting element (ascending/descending) implemented using an array:

  - Insertion and deletion operation must maintain the feature/characteristic after completing the process.
  - To insert new element - Need to shift on right every element in the list in order to give a place to a new element.
  - To delete element – Need to shift on left every element from the list in order to replace the element that we want to delete.
  - Based on the following diagram:
    - Insert 19 into sorting list (A)
    - Delete 35 from sorting list (B)

Insertion operation into sorting list



Deletion operation from sorting list

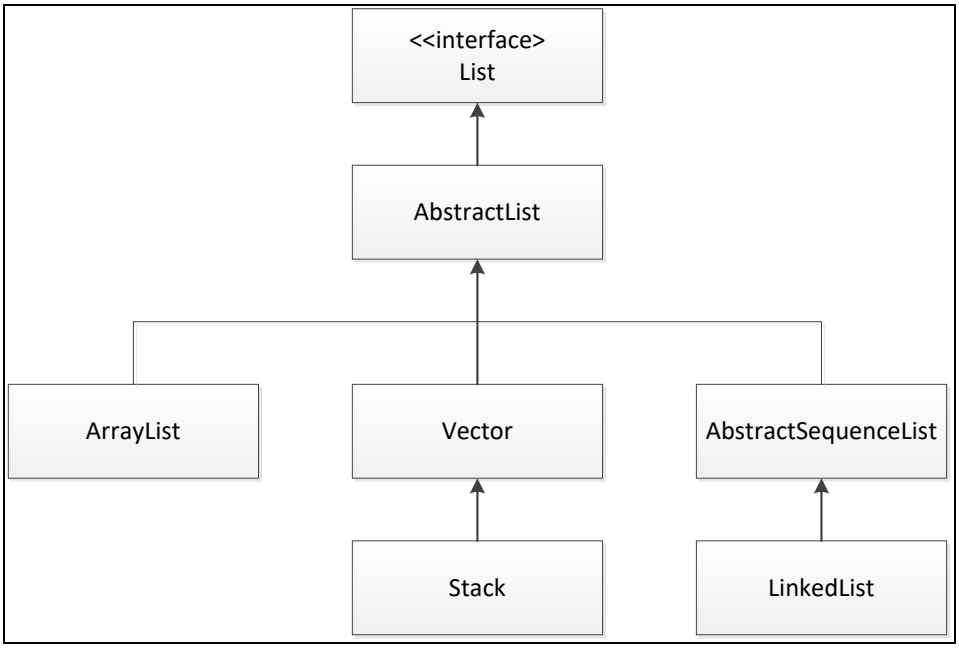


- The efficiency of insertion and deletion operation of sorting a list is depending on the number of shifting elements based on the number of elements in the list.

2.3 THE LIST INTERFACE

- The classes that implement the Java `List` interface (part of Java API java.util) all provide methods to do these operations and more. Here is some operations that can be performed:
  - Find a specified target value.
  - Add an element at either end
  - Remove an element from either end.
  - Traverse the list structure without having to manage subscripts.

- One feature that the array data structure provides that these classes don't is the ability to store primitive type values.
- The `List` classes all store references to `Objects`, so all primitive type values must be wrapped in `Objects`.
- The following figure shows an overview of the `List` interface and the four actual classes that implement it.



The `java.util.List` interface and its implementation

- We will study the `ArrayList` and `LinkedList` classes in this course.
- The following table shows the `List` ADT:-

```
/** List ADT */
public interface List<E> {
      /** Remove all contents from the list, so it is once
      again empty. Client is responsible for reclaiming
      storage used by the list elements. */
      public void clear();

      /** Insert an element at the current location. The
      client is responsible for ensuring that the list's
      capacity is not exceeded. @param item The element to be
      inserted. */
      public void insert(E item);

      /** Append an element at the end of the list. The client
      is responsible for ensuring that the list's capacity is
      not exceeded. @param item The element to be appended. */
      public void append(E item);

      /** Remove and return the current element. @return The
      element that was removed. */
      public E remove();

      /** Set the current position to the start of the list */
      public void moveToStart();

      /** Set the current position to the end of the list */
      public void moveToEnd();

      /** Move the current position one step left. No change
      if already at beginning. */
      public void prev();

      /** Move the current position one step right. No change
      if already at end. */
      public void next();

      /** @return The number of elements in the list. */
      public int length();

      /** @return The position of the current element. */
      public int currPos();

      /** Set current position. @param pos The position to
      make current. */
      public void moveToPos(int pos);

      /** @return The current element. */
      public E getValue();
}
```

The Java interface for a list

EXAMPLE OF LIST PROGRAM IMPLEMENTS USING AN ARRAY

```
import java.util.ArrayList;

public class ArrayListExample{
   public void doExample()
   {
      // set initial array capacity to 5
      ArrayList myList = new ArrayList(5);
      //load the list with integers 0 - 4
      for(int i=0; i<5; i++){
         myList.add(new Integer(i));
      }

      System.out.println("List contains " +myList.size()
         + " Element");

      //locate a specific object in the list
      Integer int2 = new Integer(2);
      System.out.println("List contains Integer(2): "
         +myList.contains(int2));
      System.out.println("Integer(2) is at index "
         +myList.indexOf(int2));

      //replace an object and then locate it by index
      myList.set(2, new Integer(99));
      System.out.println("Get element at index 2:
         "+myList.get(2));

      //add 5 more elements - capacity will grow automatically
      for(int i=5; i<10;i++){
         //add by specifying the index
         myList.add(i, new Integer(i));
      }

      //add 5 more elements, but increase the capacity first
      myList.ensureCapacity(15);
      for(int i=10; i<15; i++){
        myList.add(new Integer(i));
      }

      //take the last 5 elements back out and reduce
      //the capacity
      myList.subList(10,15).clear();
      myList.trimToSize();

      //create another list and copy it into the original one
      ArrayList otherList = new ArrayList();
      otherList.add(new String("otherList 1"));
      otherList.add(new String("otherList 2"));
      myList.add(7,otherList);
```

```
      //display the list elements
      System.out.println(myList);
   }
}

public class ArrayListExampleApp{
   public static void main(String args[])
   {
      ArrayListExample aryList = new ArrayListExample();
      aryList.doExample();
   }
}
```

2.4 OVERVIEW TO CLASS `java.util`

- **`java.util`** package – contains a number of useful classes that support important functionality as part of the Java API (Application Programming Interface)
  - Focuses mostly on collection objects – that is, objects contain or hold other objects.
  - Also adds support for property files, dates, bit manipulation, random number generation and a carryover from Smalltalk called observables.

2.5 ARRAY-BASED LISTS IMPLEMENTATION

- This section discusses the array-based approach. As we know, a sequence that supports access to its elements by their indices is called an array list.
- This index concept is a simple yet powerful notion, since it can be used to specify where to insert a new element into a list or where to remove an old element
- The Java collection framework's `ArrayList` class implements the `List` interface with underlying array that allows constant-time access of any element from the index.

2.5.1 THE `LIST` INTERFACE

- The `List` interface extends the `Collection` interface with methods that have an index as either parameter or a return type.
- The following table shows five of the methods in the `List` interface (ADT). For complete `List` interface, please refer to any Java Data Structure textbook.

Method	Description
`Object get(int index);`	**Postcondition**: the element at position index has been returned
`Object set(int index, Object elemet);`	**Postcondition**: element has replaced the element that was at index before this call, and that previous occupant has been returned.
`int indexOf(Object elem);`	**Postcondition**: if elem does occur in this List, the index of the first occurrence of elem has been returned. Otherwise, -1 has been returned.
`void add(int index, Object element);`	**Postcondition**: element has been inserted at position index and every element that was at a position >=index before this call is now at the next higher position.
`Object remove(int index);`	**Postcondition**: the element that was at position index in this List before this call has been removed, every element that was in a position > index before this call is now at the next lower position, and the removed element has been returned.

Some methods in the `List` interface ADT

2.5.2 THE *ArrayList* CLASS

- An `ArrayList` can be thought of as an improved version of a one-dimensional array.
- The `ArrayList` object supports random access of its elements; that is, any element can be accessed in constant time, given only the index of the element
- An `ArrayList` object's size is automatically maintained during the execution of a program.
- The method heading for each public method in the `ArrayList` class.

	Methods
1.	`public ArrayList(int initialCapacity)`
2.	`public ArrayList()`
3.	`public ArrayList(Collection c)`
4.	`public boolean add(Object o) //insert at back`
5.	`public void add(int index, Object element)`
6.	`public boolean addAll(Collection c)`
7.	`public boolean addAll(int index, Collection c)`
8.	`public void clear() //worstTime(n) is O(n)`
9.	`public Object clone()`
10.	`public boolean contains(Object elem)`
11.	`public boolean containsAll(Collection c)`
12.	`public void ensureCapacity(int minCapacity)`
13.	`public boolean equals (Object o)`
14.	`public Object get(int index) //worstTime(n) is constant`
15.	`public int hashCode()`
16.	`public int indexOf(Object elem)`
17.	`public boolean isEmpty()`
18.	`public Iterator Iterator()`
19.	`public int lastIndexOf(Object elem)`
20.	`public ListIterator listIterator()`
21.	`public ListIterator listIterator(final int index)`
22.	`public boolean remove(Object o)`
23.	`public Object remove(int index)`
24.	`public boolean removeAll(Collection c)`
25.	`public boolean retainAll(Collection c)`
26.	`public Object set(int index, Object element)`

```
27.  public int size()
28.  public List subList(int fromIndex, int toIndex)
29.  public Object[] toArray()
30.  public Object[] toArray(Object[] a)
31.  public String toString()
32.  public void trimToSize()
```

Method header for `ArrayList` Class

2.5.2.1 METHOD DESCRIPTION FOR THE `ArrayList` CLASS

- The following method descriptions give a user's perspective of the `ArrayList` class.

Method	Description	Example
`public ArrayList (int initialCapacity );`	**Precondition:** initialCapacity >=0. Otherwise,NegativeArraySizeException will be thrown **Postcondition:** this ArrayList object is empty, with an initial capacity of initialCapacity elements	`/*creates an empty ArrayList object called fruits, with initial capacity of 100*/` `ArrayList fruits = new ArrayList(100);`
`public ArrayList ();`		`/*construct an empty ArrayList without specifying an initial capacity*/` `ArrayList fruits = new ArrayList();`

`public boolean add(Object o);`	**Postcondition:** the Object o has been inserted at the back of this ArrayList object, and true has been returned. The averageTime(n) is constants. The worstTime(n) is still O(n)	`fruits.add("kumquats");`
		`fruits.add("apples");`
		`fruits.add("durian");`

public int size();	**Postcondition:** The number of elements in this ArrayList object has been returned	fruits.size();
public Object get(int index);	**Precondition:** 0 <= index <size(). Otherwise,an IndexOutOfBoundException will be thrown **Postcondition:** The element that is index elements from the beginning of this ArrayList object has been returned.	fruits.get(1);
public Object set(int index, Object element);	**Precondition:** 0<=index <size(). Otherwise,an IndexOutOfBoundsException will be thrown **Postcondition:** element has replaced the element that was at index before this call, and that previous occupant has been returned.	fruits.set(2,"kiwi ");

public void add(int index, Object element);	**Precondition:** 0<=index <=size(). Otherwise, an OutOfBoundsException will be thrown.
	Postcondition: element has been inserted at position index and all elements that were at positions >= index before this call have been moved to the next higher position. The worstTime(n) is O(n).
		fruits.add(1, "bananas");
		for(int i=0; i< fruits.size();i++)
		System.out.println (fruits.get(i));

public Object remove(int index);	**Postcondition:** 0 <= index < size(). Otherwise,anIndexOutOf BoundsException will be thrown **Postcondition:** The element that was at position index in this ArrayList object, every element that was in a position > index before this call has	fruits.remove(1);

	been moved to the next lower position, and the removed element has been returned. The worstTime(n) is O(n)	
public int indexOf(Object elem);	**Postcondition:** The index of the first occurrence of elem in this ArrayList object has been returned, if elem does not occur in this ArrayList object. Otherwise, -1 has been returned. The worstTime(n) is O(n).	fruits.indexOf("kiwi");
public void clear();	Postcondition: All of the elements have been removed from this ArrayList object. The worstTime(n) is O(n)	fruits.clear();

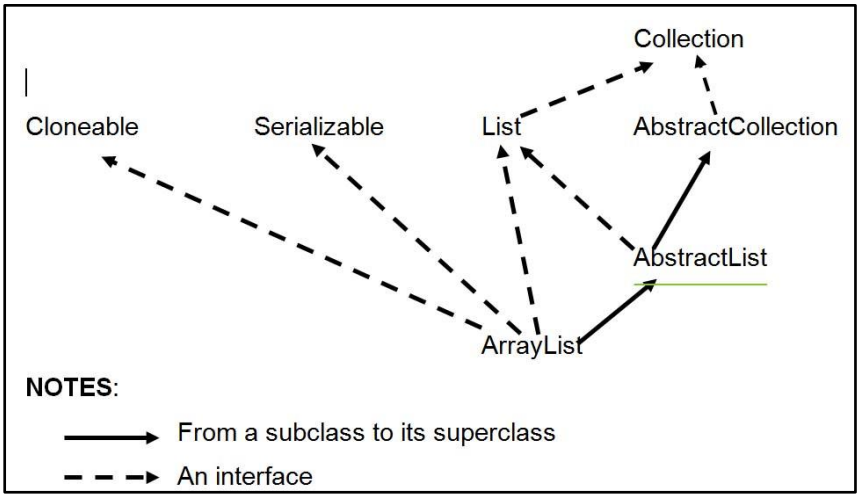
The `ArrayList` Class

2.5.2.2 `ArrayList` Class Heading

- The heading of the `ArrayList` class:

```
public class ArrayList extends AbstractList implements
     List, Cloneable, java.io.Serializable
```

The `ArrayList` class is a subclass of the class `AbstractList`, and implements three interfaces: `List`, `Cloneable`, and `Serializable`



The `ArrayList` class heading hierarchy

2.5.2.3 `ArrayList` Object Are Serializable

- To save the `ArrayList` object, on file, so that we can later resume the execution of the project without having to reconstruct the `ArrayList` object.
- Implements the `Serializable` interface that is in java.io

- **Example:** Suppose we have created an `ArrayList` object named fruits.

```
try{
     ObjectOutputStream oos = new ObjectOutputStream(
     new FileOutputStream("fruits.data"));
     oos.writeObject(fruits);
}//try

catch(IOException e){
      gui.println(e);
}//catch
```

`ArrayList` object serializable

  - The `ArrayList` object fruits has been serialized: - has been saved as a stream of bytes
  - First the size of the `ArrayList` object is stored, and the length of the elementData array, finally the elements in the ArrayList object are saved

- In another program, or in a later execution of the same program, we deserialize the fruits object:

- **Example:**

```
try{
    ObjectInputStream ois = new ObjectInputStream( new
                            FileInputStream("fruits.data"));
    fruits = (ArrayList).ois.readObject();
}
catch(Exception e){
      gui.println(e);
}//catch
```

`ArayList` object deserialize

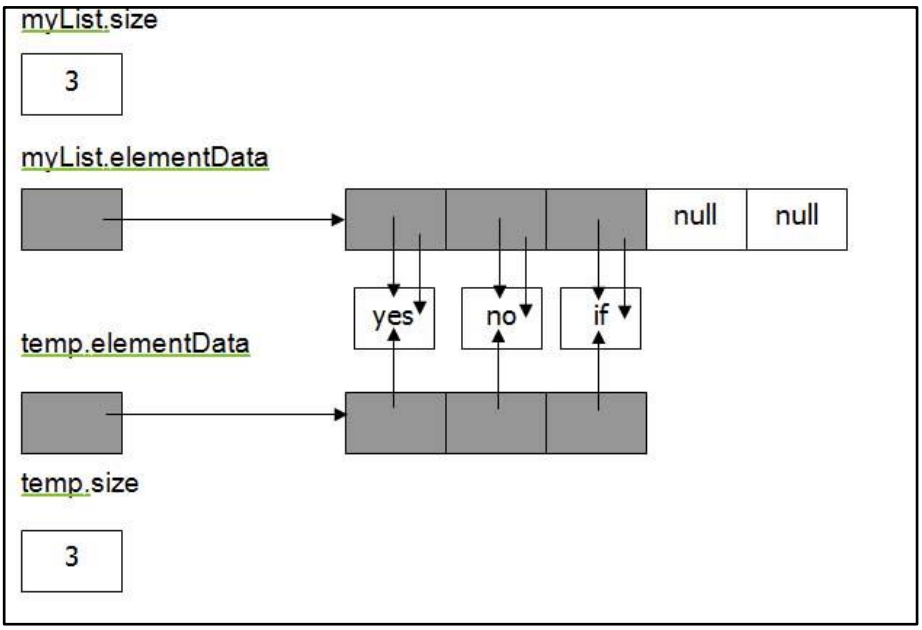
  - First the size of `ArrayList` object is read in, and then the length of the underlying array, and finally, the individual elements

2.5.2.4 ArrayList Object Are Cloneable

- The `Cloneable` interfaces has no method.
- **Example:**

```
// Postcondition: a distinct copy of this ArrayList object has
// been returned
public Object clone();

ArrayList myList = new ArrayList(5);
myList.add("yes");
myList.add("no");
myList.add("if");
ArrayList temp = (ArrayList)myList.clone();
```

  - The figure shows the effect of these statements on the size and element data fields:

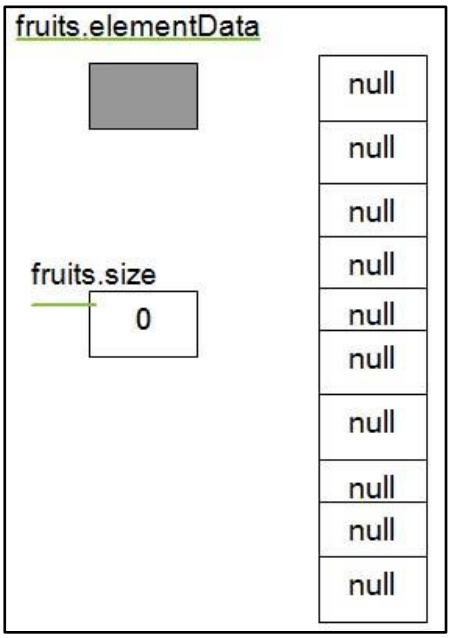


The effect on the size and element data fields when the clone method executed

2.5.2.5 THE `ArrayList` IMPLEMENTATION

- Constructor:

```
public ArrayList(int initialCapacity){
      super();
      this.elementData = new Object[initialCapacity];
}//constructor with int parameter
```



The contents of the elementData and size fields in the ArrayList object fruits after the default constructor is called

- The `ArrayList` constructor automatically initializes the size field to 0
- The default constructor has a simple definition:

```
public ArrayList(){
      this(10);
}
```

- The calling constructor:

```
ArrayList fruits = new ArrayList();
```

2.5.2.6 Definition Of The `add` Method

- Definition of the single-parameter add method:

```
public boolean add(Object o){
      ensureCapacity (size + 1);
      elementData[size++] = o;
      return true;
}
```

- **Example:**

```
fruits.add("kumquats");
```

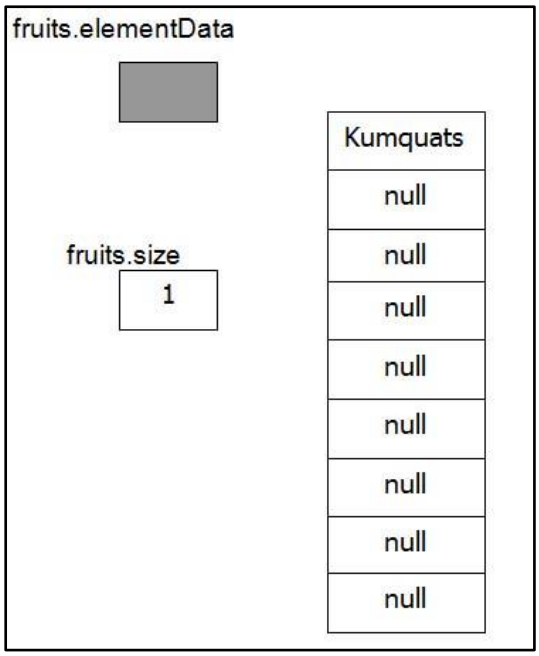


Diagram for add method using arrayList class

- **`ensureCapacity ( )`** :- If the argument, size + 1, is greater than elementData.length, the array's old reference, elementData, is copied to oldData:

```
Object oldData[] = elementData;
```

- This does not make a copy of the array, just a copy of the reference. A new array is constructed:

```
elementData = new Object[newCapacity];
//complete method for ensureCapacity
public void ensureCapacity(int minCapacity)
{
   modCount++; //discussed below
   int oldCapacity = elementData.length;

   if(minCapacity > oldCapacity){
      Object oldData[] = elementData;
      int newCapacity = (oldCapacity * 3)/2+1;
      if(newCapacity < minCapacity)
        newCapacity = minCapacity;
      elementData = new Object[newCapacity];
      system.arraycopy(oldData, 0, elementData,
                       0, size);
   }
}
```

2.5.2.7 The `clone` Method and the Copy Constructors

- The definition of the clone method:

```
public Object clone()
{
      try{
            ArrayList v = (ArrayList)super.clone();
            v.elementData = new Object[size];
            System.arraycopy(elementData,0,v.elementData,
                              0,size);
            v.modCount = 0;
            return v;
      }
      catch(CloneNotSupportedException e){
           //this shouldn't happen, since we are Cloneable
            throw new InternalError();
      }
}
```

- Copy constructor: - whose the effect is similar to that of the clone method when the constructor's argument is an ArrayList object
- The method:

```
//Postcondition: this ArrayList has been initialized to
//a copy of c.
public ArrayList(Collection c){
      this((c.size()*110)/100); //Allow 10% room for growth
      iterator I = c.iterator();
      while(i.hasNext())
            elementData[size++] = i.next();
}
```

EXAMPLE OF PROGRAM 1 - PRIMITIVE DATA TYPE

To store 100 integer values into the array list. Then, calculate the sum of values. This program also manipulates some of the methods belong to `ArrayList` class.

```
import java.util.Scanner;
import java.util.*;
import java.lang.String;

public class PrimitiveSeqList {
  public static void main(String args[])
  {
      Scanner scanner = new Scanner (System.in);
      String lineSeparator= System.getProperty (
                           "line.separator");
      scanner.useDelimiter(lineSeparator);

     int y;
     // create empty list without initial capacity
     ArrayList aList = new ArrayList();
     // create empty list with initial capacity 100
     ArrayList sList = new ArrayList(100);
     if (aList.isEmpty())
        System.out.println("Sequential list is empty\n");

     if (sList.isEmpty())
        System.out.println("Sequential list with capacity is
             empty\n");

     aList.add (0, "10"); // add 1 data
     sList.add (0, "100");
     Object num, num1;

     for (int x=0;x<100;x++)
     {
         y = x*2;

        if (aList.add (""+y))   // add 100 numbers
          System.out.println (y+" is added\n");
     }

     num = aList.remove (0); // remove element at loc 0
     num1 = aList.set(2,"100"); // replace element at location
                                // 2 with 100
     if(aList.contains ("8"))//check if value 8 is in the list
         System.out.println ("Number 8 is in the list\n");
```

```
     int numint,sum=0;

     for (int x=0;x<aList.size();x++)
     {
        num = aList.get (x);
        numint = Integer.parseInt(num.toString());//convert
                                              //element to int
       sum += numint; // calculate sum
      }
      System.out.println("Sum:  "+sum);

      System.out.print("Please enter integer: ");
      String elem = scanner.next();

      if (aList.contains (elem))
         System.out.pritln("element "+elem+ " is in the list";
      else
         System.out.println("element "+elem+" is not in the
                             list";
      aList.clear();// delete all elements in the list

      if (aList.isEmpty())
         System.out.println ("\nList is empty\n");
      else
         System.out.println ("\nList is not empty\n");
      System.exit(0);
   }
}
```

EXAMPLE OF PROGRAM 2 - PRIMITIVE DATA TYPE

To store string of colors into the array list. This program also manipulates some of the methods belong to `ArrayList` class.

```
import java.awt.Color;
import java.util.*;

public class CollectionTest{
  private static final String colors[]= {"red","white","blue"};

   public CollectionTest()
   {
      ArrayList clrList = new ArrayList(6);

      //add objects to list
      clrList.add(Color.magenta); //add a color object

      for(int count=0; count<colors.length;count++)
         clrList.add(colors[count]);

      clrList.add(Color.cyan);  //add a color object


      //Output list contents
      System.out.println("\nArrayList: ");

      for(int count=0; count<clrList.size(); count++)
         System.out.print(clrList.get(count) +"  ");


      //remove all String objects
      removeStrings(clrList);

      //Output list contents
      System.out.println("\n\nArrayList after calling
                          removeStrings: ");
      for(int count=0; count<clrList.size(); count++)
         System.out.print(clrList.get(count) +"  ");

   }//end constructor CollectionTest

   //remove String objects from Collection
   private void removeStrings(Collection collection)
   {
      Iterator iterator = collection.iterator(); //get iterator

      //loop while collection has items
      while(iterator.hasNext())
          if(iterator.next() instanceof String)
               iterator.remove(); //remove String object
   }
```

```
   public static void main(String args[])
   {
      CollectionTest c = new CollectionTest();
   }
}//end class CollectionTest
```

EXAMPLE OF PROGRAM 3 - PRIMITIVE DATA TYPE

To store some integer values into the array list. Then, sorting the list using the insertion sort algorithm. This program shows and manipulates some of the methods belong to `ArrayList` class.

```
import java.util.Scanner;
import java.util.*;
import java.lang.String;

public class PrimSeqSortList {

   // function to display all elements of the list
   public static void display(ArrayList S)
   {
      System.out.println("List after sorting\n");
      for (int x=0;x<S.size();x++)
         System.out.println ("\nData"+x+" "+S.get(x));
   }

   // function to do insertion sort
   public static void insertSort (ArrayList S)
   {
      Object A,B;
      for (int x=1;x<S.size();x++)
      {
         A=S.get(x);
         int y = x - 1;
         B=S.get(y);


         int n = Integer.parseInt(S.get(x).toString());
         int m = Integer.parseInt(S.get(y).toString());

         while (n < m && y >= 0)
         {
            S.set (y+1,S.get(y));
            y--;
         }
         S.set(y+1,A);
      }
    }
```

```
    public static void main(String args[])
    {
        int y;
        // create empty list with initial capacity 100
        ArrayList sList = new ArrayList(100);

        if (sList.isEmpty())
           System.out.println("Sequential list is empty\n");

        sList.add(0,"4");
        sList.add(1,"3");
        sList.add(2,"2");
        sList.add(3,"1");

        if (sList.add("5")
           System.out.pritln("5 is added in sList";

        System.out.println("\nIndex 0:" +sList.get(0));
        System.out.println("\nIndex 1:" +sList.get(1));
        System.out.println("\nIndex 2:"+sList.get(2));
        System.out.println("\nIndex 3:"+sList.get(3);
        System.out.println("\nIndex 4:"+sList.get(4));

        Object num = sList.remove(1);
        System.out.println("\nAfter removed");
        System.out.println("\nIndex 0:"+sList.get(0));
        System.out.println("\nIndex 1 :"+sList.get(1));
        System.out.println("\nIndex 2:"+sList.get(2));
        System.out.println("\nIndex 3:"+sList.get(3));

        // create empty list with initial capacity 100
        ArrayList sortList = new ArrayList(100);
        sortList = sList;
        insertSort (sortList);

        // Collections.sort(sortList); // to sort in order
        // Collections.sort (sList,Collections.reverseOrder());
        // sort reverse order
        display (sortList);
        display (sList);
        System.exit(0);
    }
}
```

EXAMPLE OF PROGRAM 4 - OBJECT DATA TYPE

To store an employee object into the array list. Then, determine the highest salary and display the details of employee information. This program shows and manipulates some of the methods belong to `ArrayList` class.

```
import java.util.Scanner;
import java.util.*;
import java.lang.String;

public class Employee {
   private String firstName;
   private String lastName;
   private double salary;

   public Employee () {  }

   public Employee ( String fName, String lName, double sal )
   {
      firstName = fName;
      lastName = lName;
      salary = sal;
   }

   public String getFirstName() { return firstName; }

   public String getLastName() { return lastName; }

   public double getSalary() { return salary; }
}


public class EmployeeSeqList {
    public static void main(String[] args)
    {
       // create two objects of Employee
       Employee EMP = new Employee("Ahmad","Amin",2000);
       Employee EMP1 = new Employee("Amran","Hadi",5000);//

       // create empty list with initial capacity
       ArrayList aList = new ArrayList(100);

       if (aList.isEmpty())
          System.out.println("Sequential list is empty\n");

       aList.add (0, EMP); // add employee at index 0
       if (aList.add (EMP1)); // appends employee at end of
                              //list
       Object empl = aList.get(0);
       Object empl1 = aList.get(1);
```

```
        Employee staff1 = (Employee)empl; // casting object to
        Employee staff2 = (Employee)empl1;// Employee

        Employee stafhighsalary = new Employee();
        double salary1 = staff1.getSalary();
        double salary2 = staff2.getSalary();

        if (salary1 > salary2)
          stafhighsalary = staff1;
        else
          stafhighsalary = staff2;

        System.out.println("Employee with higher salary: "
                   +stafhighsalary.getFirstName()+ " "
                   +stafhighsalary.getLastName()
                   +"with salary: "+stafhighsalary.getSalary();
        System.exit(0);
    }
}
```

PAST YEAR EXAM QUESTION – Sample

Given the following Student and ArrayList ADTs:

```
public class Student
{
     private String studName;
     private int creditHours;
     private double gradePoints;

     public Student(String sn, int ch, double gp) {…}

     public double computeGPA() {…}
     //Function to compute GPA is computed by dividing
     //gradePoints by creditHours

     public double isSenior() {…}
     //should return true if the given student has at least
     //125 credit hours and has a GPA of at least 2.0;
     //otherwise return false

     public String getStudName()  {…}
     public int getCreditHours ()  {…}
     public double getGradePoint ()  {…}

}


public class ArrayList
{
     public ArrayList (int size)
     //normal constructor

     public boolean add (Object elem)
     ///insert at back

     public Object remove (int index)
     //remove element based on index

     public Object get (int index)
     //return element from the specified location

     public Object set (int index, Object elem)
     //replace with specified element at specified location

     public int size()
     //return size of list

     //definition for other methods

}
```

Write a java application to solve the following problems:

a) Declare two sequential lists named as `listStudent1` and `listStudent2`.
(2 marks)

b) Insert ten (10) students into `listStudent1`.
(3 marks)

c) Count and display the number of students who has a GPA at least 3.0.
(3 marks)

d) Count the number of senior students and also display the information of those students.
(3 marks)

e) Update the record where the `studName is Azrul bin Ahmad`. If the record exists, replace its current value of `creditHours` and `gradePoints` with 150 and 350 respectively, otherwise display an appropriate message.
(3 marks)

f) Remove all records for student who has a GPA less than 2.00 and store them into `listStudent2`.
(3 marks)

Answer Scheme (Suggestion):

```
public class AppStudent
{
      public static void main(String args[])
      {
      a)    ArrayList listStudent1 = new ArrayList(20); [1 marks]
            ArrayList listStudent2 = new ArrayList(20); [1 marks]
                                                              (2 marks)

      b)    Scanner scanner = new Scanner (System.in);
            String lineSeparator = System.getProperty
                                   ("line.separator");
            scanner.useDelimiter (lineSeparator);

            for(int i=0; i<3; i++)
            {
                  System.out.print ("Enter student name: ");
                  String sn = scanner.next();
                  System.out.print ("Enter credit hours: ");
                  int ch = scanner.nextInt();
                  System.out.print ("Enter grade points: ");
                  int gp = scanner.nextInt();           [1 ½ mark]

                  Student S = new Student(sn,ch, gp);   [1/2 marks]
                  if (listStudent1.add(S))              [1 marks]
                        System.out.println("Success to add...");
```

```
            else
                  System.out.println("Fail to add....");
      }
```

(3 marks)

c)
```
      int cnt = 0;          [1/2 marks]

      for(int m = 0; m<listStudent1.size(); m++)  [1/2 marks]
      {
            Object obj = listStudent1.get(m);
            Student S = (Student)obj;             [1/2 marks]

            if(S.computeGPA() >= 3.00)
                  cnt = cnt + 1;                  [1 marks]
      }
     System.out.println("The num.of student:" +cnt);[1/2 marks]
```

(3 marks)

d)
```
      int cnt2 = 0;         [1/2 marks]

      for(int m = 0; m<listStudent1.size(); m++)
      {
            Object obj = listStudent1.get(m);
            Student S = (Student)obj;             [1/2 marks]

            if(S.isSenior())
            {
                  cnt2 = cnt2 + 1;
                  System.out.println("Student name:
                        "+S.getStudName());
                  System.out.println("Credit hours:
                        "+S.getCreditHours());
                  System.out.println("Grade points:
                        "+S.getGradePoint());
            }                                     [1 ½ marks]

      }
      System.out.println("The num.of senior student: " +cnt2);
                                                  [1/2 marks]
```

(3 marks)

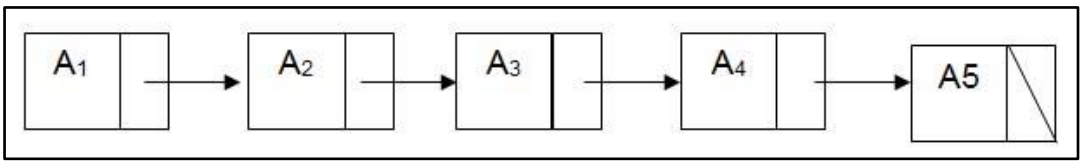
e)
```
      boolean flag = true;                  [1/2 marks]
      for(int z = 0; z<listStudent1.size(); z++)
      {
            Object obj = listStudent1.get(z);
            Student S = (Student)obj;             [1/2 marks]
            if(S.getStudName().equals("Azrul bin Ahmad"))
            {
                  Student stud = new Student ("Azrul bin Ahmad",
                                    150, 350);
                  listStudent1.set(z,stud);
                  flag = false;
            }                                     [1  ½  marks]
      }
```

```
            if(flag)
                  System.out.println("can't find any record");
                                                      [1/2 marks]
                                                            (3 marks)

      f)    for(int y = 0; y<listStudent1.size(); y++)
            {
                  Object obj = listStudent1.get(y);
                  Student S = (Student)obj;        [1/2 marks]

                  if(S.computeGPA() < 2.00)        [1/2 marks]
                  {
                        obj = listStudent1.remove(y);    [1/2 marks]
                        if (listStudent2.add(obj))
                              System.out.println("Success to add....");
                        else
                              System.out.println("Fail to add.....");
                                                      [1 marks]
                        y--;                          [1/2 marks]
                  }
            }
                                                            (3 marks)
      System.exit(0);
      }
}
```

2.6 LINKED-BASED LIST IMPLEMENTATION

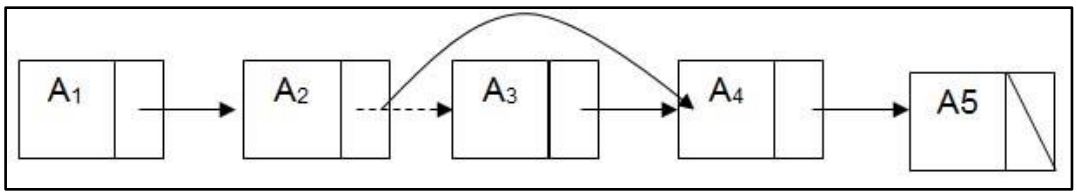
2.6.1 Introduction

- In order to avoid the linear cost of insertion and deletion, we need to ensure that list is not stored contiguously, since otherwise entire parts of the list will need to be moved.
- The linked list consists of a series of objects, called the nodes of the list, which are not necessarily adjacent in memory.
  - Each node contains the **element** and a **link** to a node containing its successor.
  - Called as **next** link.
  - The last cell's **next** link references **null.**



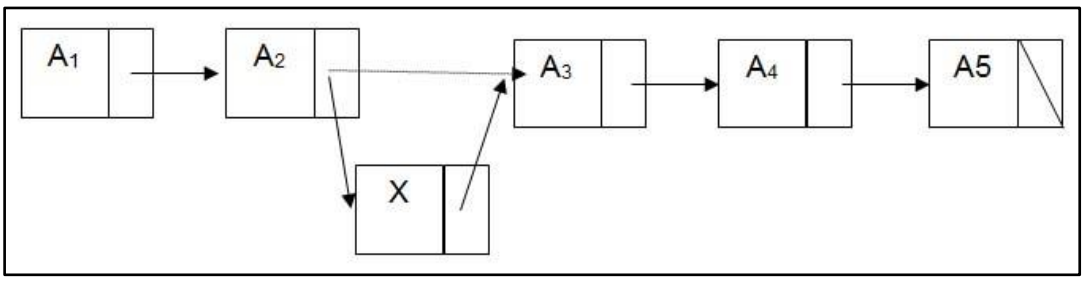
A linked list

- The **remove** method can be executed in one next reference change.



Deletion from a linked list

- The **insert** method requires obtaining a new node from the system by using a `new` call and then executing two reference maneuvers.



Insertion into a linked list

2.6.2 `LinkedList` Class Overview

- **`LinkedList`** class implements the **List** interface.
- Some significant performance differences between **`ArrayList`** and **`LinkedList`** class
  - **`LinkedList`** class lacks the random-access features of the **`ArrayList`** class
  - **`LinkedList`** class allows constant-time insertion and deletions, once the insertion point or deletion point have been accessed.

2.6.3 THE LinkedList CLASS

- The following table shows the method headings for all the public methods in the **`LinkedList`** class:

	Methods
1.	`public LinkedList()`
2.	`public LinkedList(Collection c)`
3.	`//worsTime (n) is constant` `public boolean add(Object o)`
4.	`public void add(int index, Object element)`
5.	`public void addAll(Collection c)`
6.	`public boolean addAll(int index, Collection c)`
7.	`public boolean addFirst(Object element)`
8.	`public boolean addLast(Object element)`
9.	`public void clear() // worstTime(n) is constant`
10.	`public Object clone()`
11.	`public boolean contains(Object elem)`
12.	`public boolean containsAll(Collection c)`
13.	`public boolean equals(Object o)`
14.	`//worstTime(n) is O(n)` `public Object get(int index)`
15.	`public Object getFirst(int index)`
16.	`public Object getLast(int index)`
17.	`public int hashCode()`

18. public int indexOf(Object elem)
19. public boolean isEmpty()
20. public Iterator iterator()
21. public int lastIndexOf(Object elem)
22. public ListIterator listIterator()
23. public ListIterator listIterator(final int index)
24. public boolean remove(Object o)
25. public Object remove(int index)
26. public boolean removeAll(Colection c)
27. //worstTime(n) is constant public Object removeFirst()
28. //worstTime(n) is constant public Object removeLast()
29. public boolean retainAll (Collection c)
30. public Object set (int index, Object element)
31. public int size()
32. public List subList(int fromIndex, int toIndex)
33. public Object[] toArray()
34. public Object[] toArray (Object[] a)
35. public String toString()

The LinkedList Class

2.6.4 The `LinkedList` Class versus the `ArrayList` Class

- The **`LinkedList`** class has **six methods not available** for the `ArrayList` class as shown in the following table:

Method	Descriptions
public boolean addFirst(Object element);	**Postcondition**: element has been inserted at the front of the LinkedList object.
public boolean addLast(Object element);	**Postcondition**: element has been inserted at the back of this LinkedList object.

public Object getFirst();	**Precondition**: this LinkedList object is not empty. Otherwise NoSuchElementException will be thrown. **PostCondition**: the element at index 0 in this LinkedList object has been returned.
public Object getLast();	**Precondition**:this LinkedList object is not empty. Otherwise, NoSuchElementException will be thrown. **Postcondition**:the element at index size() – 1 in this LinkedList object has een returned.
public Object removeFirst();	**Precondition**: this LinkedList object is not empty. Otherwise, NoSuchElementException will be thrown. **Postcondition**:the element at index 0 in this LinkedList object has been removed and returned.
public Object removeLast();	**Precondition**: this LinkedList object is not empty. Otherwise, NoSuchElementException will be thrown **Postcondition**:the element at index size() – 1 in this LinkedList object has been remove and returned.

The LinkedList Class

- Some other's performance differences between **`ArrayList`** object and **`LinkedList`** objects as shown in the following table.

Method	Description	Example
public boolean add(Object o)	**Postcondition**: the Object o has been inserted at the back of this LinkedList object, and true has been returned. **LinkedList -:** worstTime(n) is constant. **ArrayList -:** O(n) worst time LinkedList and ArrayList -: averageTime(n) is constant	LinkedList fruits = new LinkedList(); fruits.add("apples"); fruits.add("apples"); fruits.add("kumquats"); fruits.add("durian"); fruits.add("apples");
public void clear();	**Postcondition**:this LinkedList object is empty **LinkedList -:** worstTime(n) is constant **ArrayList -:** O(n) worst time LinkedList and ArrayList -: averageTime(n) is constant	fruits.clear();
public Object get (int index);	**Precondition**: 0 <=index <size(). Otherwise IndexOutOfBoundsException will be thrown. **Postcondition**: the element that is index elements from the beginning of this LinkedList object has been returned. The worstTime(n) is O(n)	fruits.get(1);
public Object set(int index, Object element);	**Precondition**: 0<=index<size(). Otherwise, an IndexOutOfBoundException will be thrown Postcondition:element has replaced the element that was at position index before this call, and the previous occupant has been returned. The worstTime(n) is O(n)	fruits.set(2, "kiwi");

*The LinkedList Class***LinkedList Iterator**

- The iterators are bidirectional: they can move either forward (to the next element) or backward (to the previous element).
- The name of class that defines the iterator is **`ListItr`**.
- The **`ListItr`** class – which implements the `ListIterator` interface – is embedded as a private class in the `LinkedList`.
- They are two **`LinkedList`** methods that return a (reference to a) `ListIterator` object, that is, an object in a class that implements the `ListIterator` interfaces.
- The following table shows the methods for `ListIterator` interfaces:

Method	Description	Example
`public ListIterator listIterator();`	**Postcondition:** the iterator object returned is positioned at the beginning of this LinkedList object	`ListIterator itr1 = fruits.listIterator();`
`public listIterator listIterator(final int index);`	**Precondition:** 0<=index<=size(). Otherwise, IndexOutOfBoundsException will be thrown **Postcondition:** the ListIterator object returned starts at the element positioned at index in this LinkedList object, or beyond the last element if index = size(). The worstTime(n) is O(n)	`ListIterator itr2 = fruits.listIterator(2);`

The `ListIterator` interfaces

- The following table shows the method heading for all the methods in the **ListItr** class.

Methods
1. `public void add(Object o)`
2. `public boolean hasNext()`
3. `public boolean hasPrevious()`
4. `public Object next()`
5. `public int nextIndex()`

6. public Object previous()
7. public Object previousIndex()
8. public void remove()
9. public void set (Object o)

Method heading for `ListItr`

- The following table shows the method descriptions for `ListItr` ADT:

Method	Description	Example
public boolean hasPrevious();	**Postcondition:** true has been returned if this ListIterator object is not positioned at the first element in this LinkedList object. Otherwise, false has been returned.	ListIterator itr = listiterator(2); gui.println(itr. hasPrevious());

public Object previous();	**Precondition:** this ListIterator object is not positioned at index 0. Otherwise, NoSuchElementException will be thrown
	Postcondition: this ListIterator object has retreated to the previous element, and that element has been returned.
		Example 1:
		ListIterator itr =
		fruits.listIterator
		();
		gui.println(itr.
		next() + "
		"+itr.next()+ " " +
		itr.previous());
		
		Example 2:
		ListIterator itr =
		fruits.listIterator
		(fruits.size());
		
		while(itr.
		hasPrevious()){
		gui.println(
		itr.previous());
		}

public void add(Object o);	**Postcondition:** the element o has been inserted into the LinkedList object in front of the element that would be returned by next() and in back of the element that would be returned by previous(). There is	ListIterator itr =
		fruits.listIterator
		();
		
		while(itr.
		hasNext()){
		itr.next();
		
		itr.add("pears");
		}//while

	now no element in the LinkedList object that is considered the "last returned" element.	

public void remove();	**Precondition**: the next() or previous() method has been called since the most recent (if any) call to the add method by this ListIterator object. Otherwise, NoSuchElementException has been thrown
	Postcondition: the last returned element has been removed from theLinkedList object that this ListIterator object is iterating through. This ListIterator object is now no element in this LinkedList object that is considered the "last returned" element.
		ListIterartor itr = fruits.listIterator (1);
		while(itr.hasNext()){
		itr.next();
		itr.remove();
		if(itr.hasNext())
		itr.next();
		}//while

public void set(Object o);	**Precondition**: The element this ListIterator object is positioned at was last returned (by a call to next() or previous()) by this ListIterator object. Otherwise, NoSuchElementException has been thrown.
	Postcondition: the element this ListIterator object was positioned at before this call has been changed to o.
		String aFruit;
		char first;
		ListIterator itr = fruits.listIterator ();
		while(itr.
		hasNext()){
		AFruit = (String)
		itr.next();
		First = Character.toUpperca se(aFruit.charAt(0));
		Afruit = first + aFruit.substring(1) ;
		Itr.set(aFruit);
		}//while

The details implementation of ListIterator interfaces

- To summarize the above discussion:
  1. If the application entails a lot of accessing and/or modifying of elements at widely varying indices, it will be completed much faster if an **`ArrayList`** object is used instead of a **`LinkedList`** object
  2. If a large part of the application consists of iterating through a list and making insertions and/or removals during the iterations, the application will be complete much faster if a **`LinkedList`** object is used instead of an **`ArrayList`** object

EXAMPLE OF PROGRAM 1– PRIMITIVE DATA TYPE

To store string of colours into the linked list. Then, convert it to uppercase and print in reverse order. This program shows and manipulates some of the methods belong to `LinkedList` class.

```
import java.util.*;

public class ListTest
{
   private static final String colors[] = {"black", "yellow",
                       "green", "blue", "violet","silver"};
   private static final String colors2[] = {"gold", "white",
                       "brown", "blue", "gray","silver"};

   //set up and manipulate LinkedList objects
   public ListTest()
   {
       List link = new LinkedList();
       List link2 = new LinkedList();

       //add elements to each list
       for(int count=0; count<colors.length; count++)
       {
           link.add(colors[count]);
           link2.add(colors2[count]);
       }

       link.addAll(link2);  //concatenate list
       link2 = null;        //release resources

       printList(link);

       uppercaseStrings(link);

       printList(link);
```

```
      System.out.println("Deleting elements 4 to 6...");
      removeItems(link,4,7);

      printList(link);

      printReversedList(link);
   }//end constructor ListedTest

   //output list contents
   public void printList(List list)
   {
      System.out.println("list: ");

      for(int count=0; count<list.size(); count++)
         System.out.print(list.get(count) +" ");
      System.out.println();
   }

   //locate String objects and convert to upperase
   private void uppercaseStrings(List list)
   {
      ListIterator iterator = list.listIterator();

      while(iterator.hasNext())
      {
         Object object = iterator.next(); //get item

         if(object instanceof String)  //check for string
            iterator.set(((String) object).toUpperCase() );
      }
   }

   //obtain sublist and use clear method to delete sublist
   //items
   private void removeItems(List list, int start, int end)
   {
       list.subList(start,end).clear();  //remove items
   }

   //print reversed list
   private void printReversedList(List list)
   {
       ListIterator iterator=list.listIterator(list.size());

       System.out.println("Reversed List: ");

       //print list in reverse order
       while(iterator.hasPrevious())
         System.out.print(iterator.previous() + " ");

   }
```

```
    public static void main(String[] args)
    {
       new ListTest();
    }
}//end class ListTest
```

EXAMPLE OF PROGRAM 2 - PRIMITIVE DATA TYPE

To store integer numbers into the linked list. Then, calculate the sum of numbers. This program shows and manipulates some of the methods belong to `LinkedList` class.

```
import java.util.*;
import java.util.Scanner;

public class LinkedListNumber
{
  public LinkedListNumber()
  {
       Scanner scanner = new Scanner (System.in);
       String lineSeparator = System.getProperty("line.separator");
       scanner.useDelimiter(lineSeparator);

       String strNum, strInd, strNumber;
       int total, ind, intNum;

       LinkedList listNum = new LinkedList();

       for (int i=0; i<4; i++){
         System.out.println("Enter number:");
         strNum = scanner.next();
         listNum.add(i,strNum);
       }

       System.out.println("Enter index to remove:");
       strInd = scanner.next();

       ind = Integer.parseInt(strInd);
       System.out.println("Removed number: "
                         +listNum.remove(ind));

       System.out.println("Enter number to remove:");
       strNumber = scanner.next();

       if(listNum.remove(strNumber))
          System.out.println("The number has been successfully
                              removed");
```

```
      else
         System.out.println("Sorry..The number is not in the
                             list ");

      System.out.println("Content of the list after removed:"
                         +listNum);
      System.out.println("The numbers of elements in the
                         list:" +listNum.size());

      Object tempNum;
      total = 0;

      for (int i=0; i<listNum.size(); i++){
          tempNum = listNum.get(i);
          intNum = Integer.parseInt(tempNum.toString());
          total = total + intNum;
      }

      System.out.println("Content of the list : " +listNum);
      System.out.println("Total : " +total);
      System.exit(0);
  }

  public static void main(String args[])
  {
       LinkedListNumber list = new LinkedListNumber();
  }
}
```

EXAMPLE OF PROGRAM 3 - OBJECT DATA TYPE

To store object of Staff into the linked list. Then, count the number of staff who gets the salary less than RM 1500. Find and display the details of staff information who gets the highest and minimum salary. This program shows and manipulates some of the methods belong to `LinkedList` class.

```
import java.util.*;
import javax.swing.JOptionPane;

public class Staff
{
  private String name;
  private int staffNum;
  private String position;
  private double salary;
```

```
  public Staff()
  {
     name = " ";
     staffNum = 0;
     position = " ";
     salary = 0.0;
  }

  public Staff(String nm, int sn, String ps, double sl)
  {
     name = nm;
     staffNum = sn;
     position = ps;
     salary = sl;
  }

  public String getName()  {  return name;  }

  public int getStaffNum() {  return staffNum;  }

  public String getPosition() {  return position;  }
  public double getSalary()  {  return salary;  }

}

public class StaffLinkedList
{

  public static void main(String[] args)
  {
     String strCount, strName, strPosition, strSalary,
            strStaffNum;
     int count, staffNumber;
     double salary;

     LinkedList staffList = new LinkedList();

     strCount = JOptionPane.showInputDialog("How many data to
                insert: ");
     count = Integer.parseInt(strCount);



     for(int i=0; i<count; i++)
     {
        strName = JOptionPane.showInputDialog("Enter
                  name: ");
        strStaffNum = JOptionPane.showInputDialog("Enter
                  Staff Number: ");
        strPosition = JOptionPane.showInputDialog("Enter
                  position: ");
        strSalary = JOptionPane.showInputDialog("Enter
                  Salary: ");
        staffNumber = Integer.parseInt(strStaffNum);
        salary = Double.parseDouble(strSalary);
```

```
        Staff st = new Staff(strName, staffNumber,
                        strPosition, salary);
        staffList.add(i, st);
     }

     Object obj= staffList.get(0);
     Staff s = (Staff)obj;
     double max = s.getSalary();
     double min = s.getSalary();
     int cnt =0;
     Staff maxStaff = s;
     Staff minStaff = s;

     for(int i=0; i<staffList.size(); i++)
     {
        obj= staffList.get(i);
        s = (Staff)obj;

        if(s.getSalary() < 1500)
           cnt = cnt + 1;

        if(s.getSalary() > max){
           max = s.getSalary();
           maxStaff = s;
        }

        if(s.getSalary()< min){
           min = s.getSalary();
           minStaff = s;
        }
    }

    JOptionPane.showMessageDialog(null,"The number of
       employee salary less RM 1500: " +cnt);
    JOptionPane.showMessageDialog(null,"Employee with higher
       salary\n"+maxStaff.getName()+"\n"
       +maxStaff.getStaffNum() + "\n" +maxStaff.getPosition()
       + "\n"+maxStaff.getSalary());
    JOptionPane.showMessageDialog(null,"Employee with minimum
       salary\n"+minStaff.getName()+"\n"
       +minStaff.getStaffNum()+"\n"+minStaff.getPosition()
       +"\n"+minStaff.getSalary());
  }
}
```

2.7 LINKED LISTS - USER DEFINED TYPE

- Linked list are collections of data items “lined up in a row” – insertions and deletions can be made anywhere in a linked list.

2.7.1 Self-Referential Classes

- A self-referential class contains an instance variable that refers to another object of the same class type.
- **Example :**

```
class Node{
      private int data;
      private Node nextNode;  //reference to next linked node

      public Node(int data)             {/* Constructor body */}
      public void setData(int data)     { /* method body */}
      public int getData()              { /* method body */}
      public void setNext(Node next)    { /* method body */}
      public Node getNext()             { /* method body */}
}
```

  - **Explanation:**
    - Two private instance variables -> Integer `data` and `Node` reference `nextNode` (field `nextNode` references an object class `Node`, an object of the same class being declared here – hence, the term “self-referential class”.
    - Field `nextNode` is a link → it “links” an object of type `Node` to another object of the same type.
    - Class `Node` has five methods:

      (i) **constructor** – That receives an integer to initialize data

      (ii) **setData** → To set the value of data.

      (iii) **getData** → To return the value of data.

      (iv) **setNext** → To set the value of `nextNode`.

      (v) **getNext** → To return a reference to the next node.

- The program can link self-referential objects together to form such useful data structure as list, queues, stacks and trees.

- **Example:**

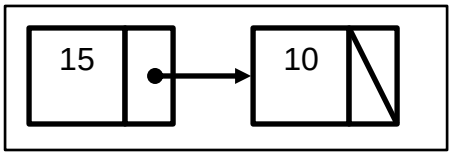


Diagram for linked list node

  - **NOTE:** backslash → representing a *null* reference is placed in the link member of the second self-referential object to indicate that the link does not refer to another object (a *null* indicates the end of a data structure) .

- The **declaration** and class-instance creation expression:

```
Node nodeToAdd = new Node(10); //10 is nodeToAdd's data
```

  - **NOTE:** allocate the memory to store a `Node` object and returns a reference to the object, which is assigned to `nodeToAdd`, if insufficient memory is available, the expression throws an `OutOfMemoryError`.

EXAMPLE OF PROGRAM – PRIMITIVE DATA TYPE

To store primitive data type objects into the user defined linked list. This program shows and manipulates some of the user defined methods in order to implement the linked list.

```
// class to represent one node in a list
public class ListNode {
   // package access members; List can access these directly
   Object data;
   ListNode nextNode;

   // create a ListNode that refers to object
   ListNode( Object object )
   {
      this( object, null );
   }
```

```
   // create ListNode that refers to Object and to next
   // ListNode
   ListNode( Object object, ListNode node )
   {
      data = object;
      nextNode = node;
   }

   // return reference to data in node
   Object getObject()
   {
      return data; // return Object in this node
   }

   // return reference to next node in list
   ListNode getNext()
   {
      return nextNode; // get next node
   }

} // end class ListNode

// class List definition
public class List {
   private ListNode firstNode;
   private ListNode lastNode;
   private String name;  // string like "list" used in printing

   // construct empty List with "list" as the name
   public List()
   {
      this( "list" );
   }

   // construct an empty List with a name
   public List( String listName )
   {
      name = listName;
      firstNode = lastNode = null;
   }

   // insert Object at front of List
   public synchronized void insertAtFront( Object insertItem )
   {
      // firstNode and lastNode refer to same object
      if ( isEmpty() )
         firstNode = lastNode = new ListNode( insertItem );

      else // firstNode refers to new node
         firstNode = new ListNode( insertItem, firstNode );
   }
```

```
   // insert Object at end of List
   public synchronized void insertAtBack( Object insertItem )
   {
      // firstNode and lastNode refer to same Object
      if ( isEmpty() )
         firstNode = lastNode = new ListNode( insertItem );

      else // lastNode's nextNode refers to new node
         lastNode=lastNode.nextNode= new ListNode(insertItem );
   }

   // remove first node from List
   public synchronized Object removeFromFront() throws
   EmptyListException {
      if ( isEmpty() ) // throw exception if List is empty
         throw new EmptyListException( name );

      // retrieve data being removed
      Object removedItem = firstNode.data;

      // update references firstNode and lastNode
      if ( firstNode == lastNode )
         firstNode = lastNode = null;
      else
         firstNode = firstNode.nextNode;

      return removedItem; // return removed node data

   } // end method removeFromFront

   // remove last node from List
   public synchronized Object removeFromBack() throws
   EmptyListException {
      if ( isEmpty() ) // throw exception if List is empty
         throw new EmptyListException( name );

      // retrieve data being removed
      Object removedItem = lastNode.data;

      // update references firstNode and lastNode
      if ( firstNode == lastNode )
         firstNode = lastNode = null;


      else { // locate new last node
         ListNode current = firstNode;

         // loop while current node does not refer to lastNode
         while ( current.nextNode != lastNode )
            current = current.nextNode;

         lastNode = current; // current is new lastNode
         current.nextNode = null;
      }
```

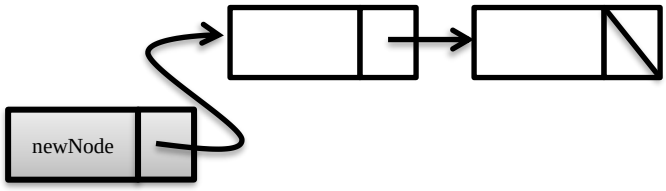
```
      return removedItem; // return removed node data

   } // end method removeFromBack

   // determine whether list is empty
   public synchronized boolean isEmpty()
   {
      return firstNode == null; // return true if List is empty
   }


   // output List contents
   public synchronized void print()
   {
      if ( isEmpty() ) {
         System.out.println( "Empty " + name );
         return;
      }
      System.out.print( "The " + name + " is: " );
      ListNode current = firstNode;

      // while not at end of list, output current node's data
      while ( current != null ) {
         System.out.print( current.data.toString() + " " );
         current = current.nextNode;
      }

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

} // end class List


//*************************************************************

public class EmptyListException extends RuntimeException {
   // no-argument constructor
   public EmptyListException()
   {
      this("List");//call other EmptyListException constructor
   }

   // constructor
   public EmptyListException( String name )
   {
      super( name + " is empty" );//call superclass constructor
   }

} // end class EmptyListException
```

```
//*************************************************************

public class ListTest {
   public static void main( String args[] )
   {
      List list = new List();  // create the List container

      // objects to store in list
      Boolean bool = Boolean.TRUE;
      Character character = new Character( '$' );
      Integer integer = new Integer( 34567 );
      String string = "hello";

      // insert references to objects in list
      list.insertAtFront( bool );
      list.print();
      list.insertAtFront( character );
      list.print();
      list.insertAtBack( integer );
      list.print();
      list.insertAtBack( string );
      list.print();

      // remove objects from list; print after each removal
      try {
         Object removedObject = list.removeFromFront();
         System.out.println( removedObject.toString()
               + " removed" );
         list.print();

         removedObject = list.removeFromFront();
         System.out.println( removedObject.toString()
               + " removed" );
         list.print();

         removedObject = list.removeFromBack();
         System.out.println( removedObject.toString()
               + " removed" );
         list.print();

         removedObject = list.removeFromBack();
         System.out.println( removedObject.toString()
               + " removed" );
         list.print();

      } // end try block

      //catch exception if remove is attempted on an empty List
      catch ( EmptyListException emptyListException ) {
         emptyListException.printStackTrace();
      }
   }
} // end class ListTes
```

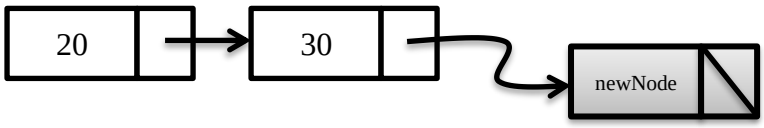
2.7.2 OPERATION FOR A LINKED LIST

- In this section, we will discuss four main operations in the linked list as follows:

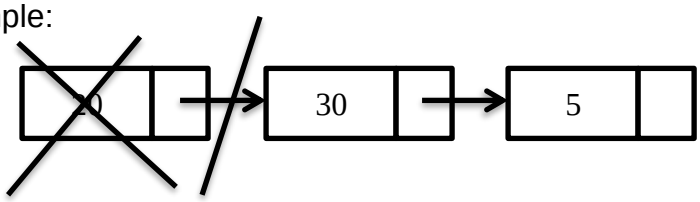
1. Insert a new node at the front of the list.
   - This operation will insert a new node at the front of the list.
   - Example:



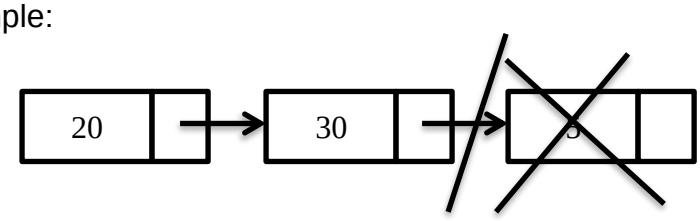
2. Insert a new node at the back of the list.
   - This operation will insert a new node at the back of the list.
   - Example:



3. Delete a node from the front of the list.
   - This operation will delete a node from the front of the list.
   - Example:
   -



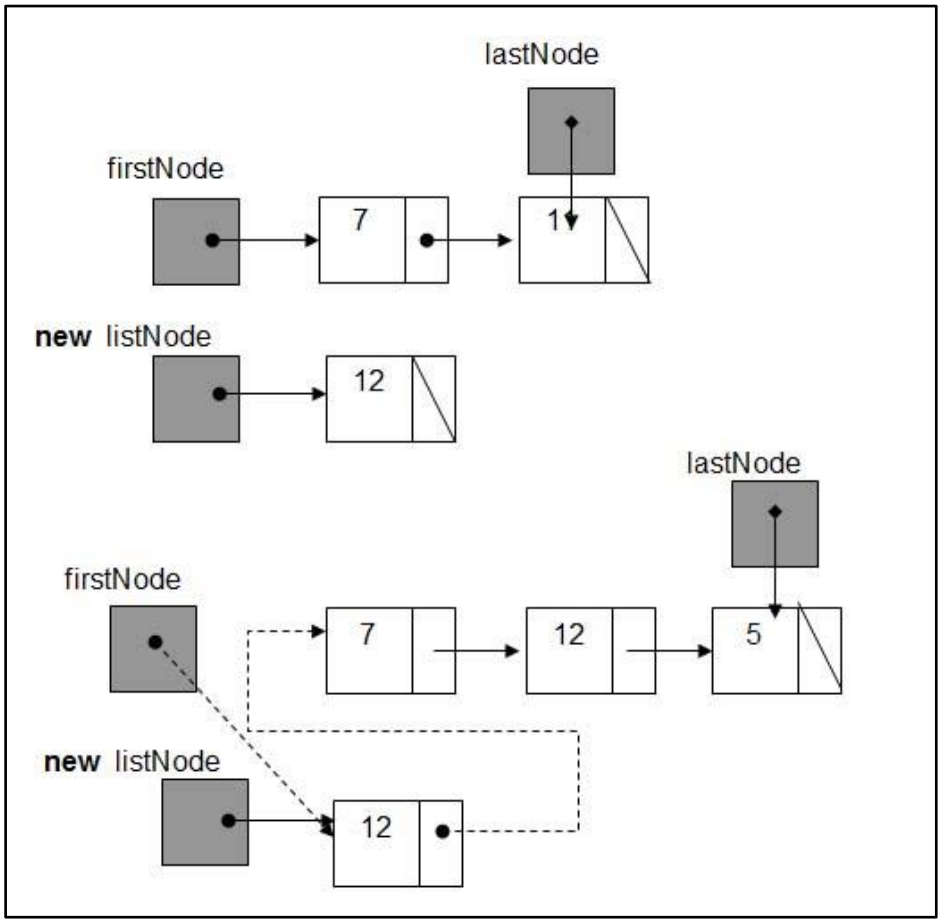
4. Delete a node from the back of the list.
   - This operation will delete a node from the back of the list.
   - Example:



Insert a new node at the front of the list

The Steps:

1. Call `isEmpty`, to determine whether the list is empty.
2. If the list is empty, assign `firstNode` and `lastNode` to the new `ListNode` that was initialized with `insertItem`. The constructor calls the `ListNode` constructor to set the instances variable data to refer to the `insertItem` passed as an argument and to set a reference `nextNode` to `null`, because this is the first and last node in the list.
3. If the list is not empty, the new node is "linked" into the list by setting `firstNode` to a new `ListNode` object and initializing that object with `insertItem` and `firstNode`. When the `ListNode` constructor executes, it sets instance variable data to refer to the `nextNode` reference of the new node to the `ListNode` passed as an argument, which previously was the first node.

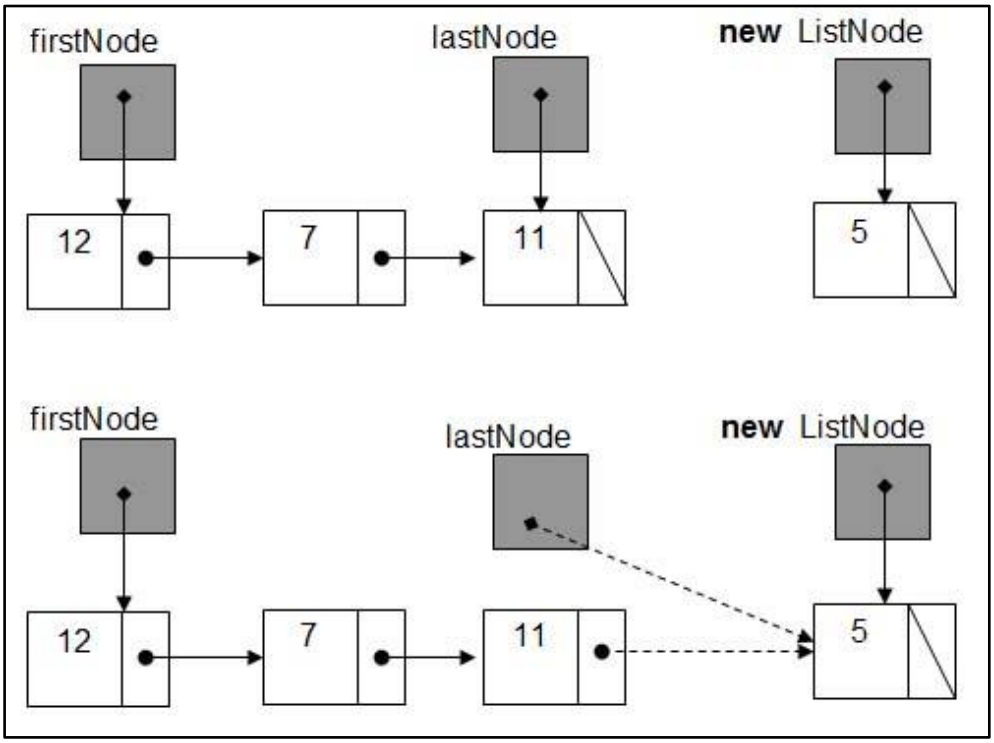


Graphical representation of operation insertAtFront

Insert a new node at the back of the list

The Steps:

1. Call `isEmpty` to determine whether the list is empty.
2. If the list is empty, assign `firstNode` and `lastNode` to the new `ListNode` that was initialized *with* `insertItem`. The `ListNode` constructor calls the constructor to set instance variable data to refer the `insertItem` passed as an argument and to set reference `nextNode` to null.
3. If the list is not empty, links the new node into the list by assigning `lastNode` and `lastNode.nextNode` the reference to the new `ListNode` that was initialized with `insertItem`. `ListNode's` constructors set instance variable data to refer to the `insertItem` passed as an argument and sets reference `nextNode` to *null*, because this is the last node in the list.

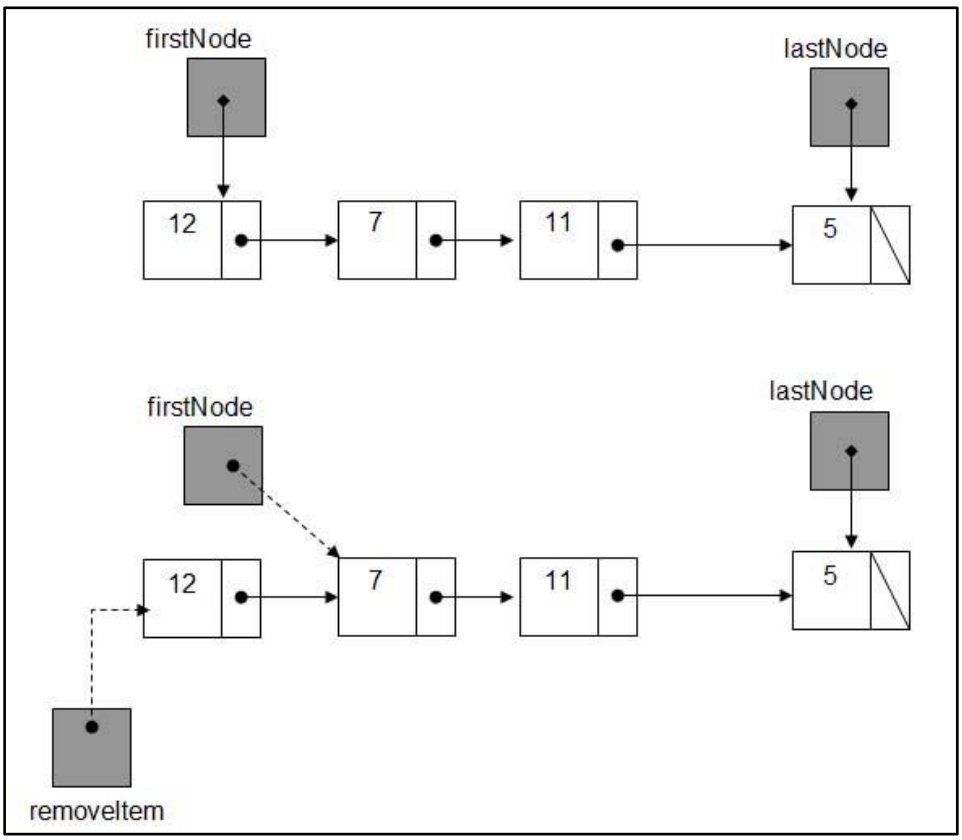


Graphical representation of operation insertAtBack

Delete a node from the front of the list

The Steps:

1. Assign `firstNode.data` (the data being removed from the list) to reference `removedItem`
2. If the `firstNode` and `lastNode` refer to the same object, the list has only one element at this time. So, the method sets `firstNode` and `lastNode` to *null* to remove the node from the list (leaving the list empty)
3. If the list has more than one node, then the method leaves reference `lastNode` as is and assigns the value of `firstNode.nextNode` to `firstNode`. Thus, *firstNode* references the node that was previously the second node in the list
4. Return the `removedItem` reference

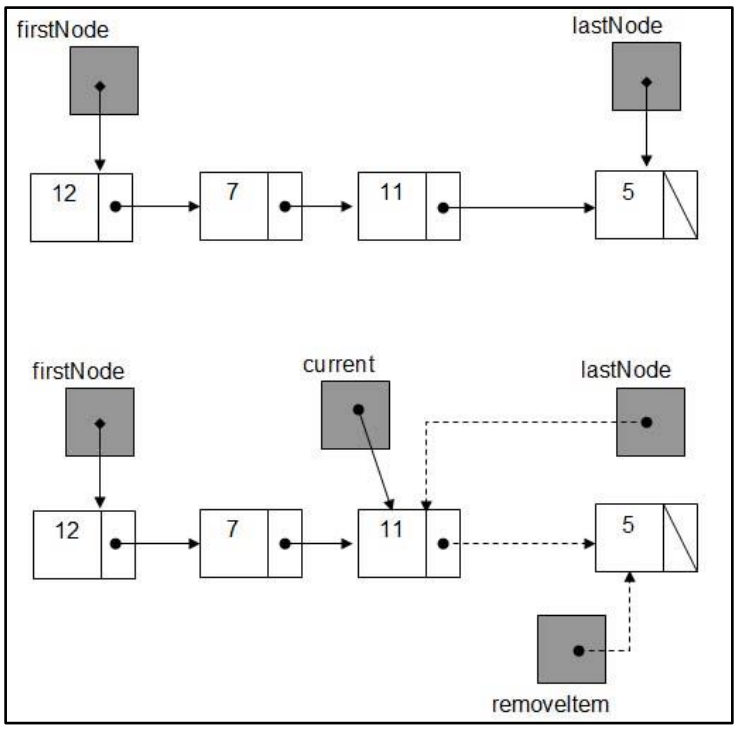


Graphical representation of operation iremoveFromFront

Delete a node from the back of the list

The Steps:

1. Assign `lastNode.data` (the data being removed from the list) to `removeItem`.
2. If the `firstNode` and *lastNode* refer to the same object, the list has only one element at this time. So sets `firstNode` and `lastNode` to *null* to remove that node from the list (leaving the list empty).
3. If the list has more than one node, create the `listNode` reference `current` and assign it `firstNode`.
4. Now "walk the list" with `current` until it references the node before the last node. The while loop assigns `current.nextNode` to current as long as `current.nextNode` (the next node in the list) is not `lastNode`.
5. After locating the second-to-last node, assign *current* to `lastNode` to update which node is the last in the list.
6. Set the `current.nextNode` to *null* to remove the last node from the list and terminate the list at the `current` node
7. Return the `removedItem` reference.



Graphical representation of operation removedFromBack

EXAMPLE OF PROGRAM – OBJECT DATA TYPE

```
/*
Author: Mazidah Puteh
Objective:
  1.  Store Student into Linked List from front
  2.  Calculate sum and average cgpa of all students
  3.  Removed one element from front

  Note:
  Program can be modified to
  1. Insert more functions.
  2. break the classes into packages and use import statement
*************************************************************/
import javax.swing.JOptionPane;

public class Student {
  private String name,status;
  private int studId;
  private int part;
  private double cgpa;


  public Student ()
  {
       name = "";
       studId = -1;
       part = -1;
       cgpa = -1.0;
       status = "";
  }

  public Student ( String nm, int sid, int pt, double cg)
  {
    name = nm;
    studId = sid;
    part = pt;
    cgpa = cg;
  }

  public void setStudent (String nm,int sid,int pt, double cg)
  {
    name = nm;
    studId = sid;
    part = pt;
    cgpa = cg;
  }

  public String toString ()
  {
    return "\nName : " + name + "\nStudent ID :  " + studId +
           "\nPArt : " + part + "\nCGPA : " + cgpa +
           "\nStatus : " + status;
  }
```

```
  public void setStatus(String stat) { status = stat; }
  public String getName()   { return name; }
  public int getStudId()    { return studId; }
  public int getPart()      { return part; }
  public double getCgpa()   { return cgpa; }
}

class EmptyListException extends RuntimeException {
   public EmptyListException( String name )
   {
      super( "The " + name + " is empty" );
   }
}

class ListNode {
   // package access data so class List can access it directly
   Object data;
   ListNode next;

   // Constructor: Create a ListNode that refers to Object o.
   ListNode( Object o ) { this( o, null ); }

   // Constructor: Create a ListNode that refers to Object o
   // and to the next ListNode in the List.
   ListNode( Object o, ListNode nextNode )
   {
      data = o;          // this node refers to Object o
      next = nextNode;   // set next to refer to next
   }

   // Return a reference to the Object in this node
   Object getObject() { return data; }

   // Return the next node
   ListNode getLink() { return next; }
}

// Class List definition
public class LinkedList {
   private ListNode firstNode;
   private ListNode lastNode;
   private ListNode currNode; // use to traverse the list
   private String name; // String like "list" used in printing

   // Constructor: Construct an empty List with s as the name
   public LinkedList( String s )
   {
      name = s;
      firstNode = lastNode = currNode = null;
   }

   // Constructor: Construct an empty List with  "list" as the
   // name
   public LinkedList() { this( "linked list" ); }
```

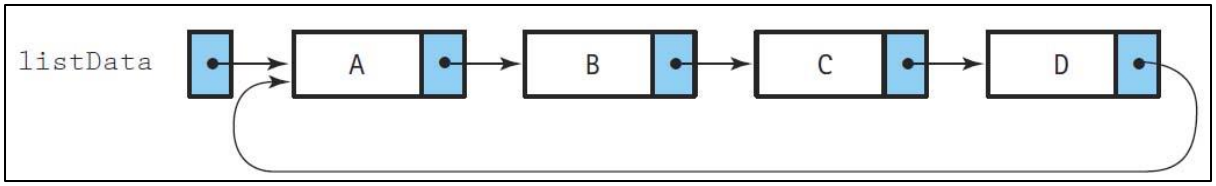
```
   // Insert an Object at the front of the List If List is
   // empty, firstNode and lastNode will refer to the same
   // object. Otherwise, firstNode refers to new node.
   public void insertAtFront( Object insertItem )
   {
      if ( isEmpty() )
         firstNode = lastNode = new ListNode( insertItem );
      else
         firstNode = new ListNode( insertItem, firstNode );
   }

   // Insert an Object at the end of the List If List is
   // empty, firstNode and lastNode will refer to the same
   // Object. Otherwise, lastNode's next instance variable
   // refers to new node.
   public void insertAtBack( Object insertItem )
   {
      if ( isEmpty() )
         firstNode = lastNode = new ListNode( insertItem );
      else
         lastNode = lastNode.next = new ListNode(insertItem);
   }

   // Remove the first node from the List.
   public Object removeFromFront() throws EmptyListException
   {
      Object removeItem = null;

      if ( isEmpty() )
         throw new EmptyListException( name );

      removeItem = firstNode.data;  // retrieve the data

      // reset the firstNode and lastNode references
      if ( firstNode.equals( lastNode ) )
         firstNode = lastNode = null;
      else
         firstNode = firstNode.next;

      return removeItem;
   }

   // Remove the last node from the List.
   public Object removeFromBack() throws EmptyListException
   {
      Object removeItem = null;

      if ( isEmpty() )
         throw new EmptyListException( name );

      removeItem = lastNode.data;  // retrieve the data

      // reset the firstNode and lastNode references
      if ( firstNode.equals( lastNode ) )
         firstNode = lastNode = null;
```

```
      else {
         ListNode current = firstNode;

         while ( current.next != lastNode )  // not last node
            current = current.next;      // move to next node

         lastNode = current;
         current.next = null;
      }

      return removeItem;
   }

   // Return true if the List is empty
   public boolean isEmpty()
   { return firstNode == null; }

   // Return First element
   public Object getFirst()
   {
       if (isEmpty())
           return null;
        else
        {
           currNode = firstNode;
           return currNode.data;
        }
   }

   public Object getNext()
   {
      if (currNode != lastNode)
      {
           currNode = currNode.next;
           return currNode.data;
      }
       else
           return null;
   }
}

public class StudLinkList {
   public static void main( String args[] )
   {
      // create the List container
      LinkedList objList = new LinkedList();

      // Create objects to store into the List
      for (int x=0;x<3;x++)
      {
          String sname = JOptionPane.showInputDialog("Please
                         enter name");
          String sid   = JOptionPane.showInputDialog("Please
                          enter id");
```

```
            String spart = JOptionPane.showInputDialog("Please
                           enter part");
            String scgpa = JOptionPane.showInputDialog("Please
                           enter cgpa");

            int id = Integer.parseInt(sid);
            int part = Integer.parseInt(spart);
            double cgpa = Double.parseDouble (scgpa);

            // must create new student in the loop
            Student stud = new Student (sname,id,part,cgpa);

            if (cgpa >= 3.5)
               stud.setStatus("Dean's List");
            else if (cgpa >= 2.00)
               stud.setStatus("Pass");
            else if (cgpa >= 1.8)
               stud.setStatus("Probation");
            else
               stud.setStatus("Fail");

            objList.insertAtFront(stud);
         }

         // To claculate sum and average all numbers in List
         double sumcgpa = 0,avecgpa,cg;

         // access first element in the list
         Student data = (Student) objList.getFirst();
         if (data!=null)
         {
              sumcgpa += data.getCgpa();

              // access next element in the list
              data = (Student) objList.getNext();
              while (data != null)
              {
                 sumcgpa += data.getCgpa();
                 data = (Student) objList.getNext();
              }
              avecgpa = sumcgpa / 3;
              System.out.println ("Average cgpa is "+ avecgpa);
         }
         else
           System.out.println ("Cannot average, List is empty");

         // Use the List remove methods
         Student removedObj;
         removedObj = (Student)objList.removeFromFront();
         System.out.println("Removed:"+removedObj.toString());

      }
}
```

2.8 CONCEPT IN VARIATION OF LINKED LIST: CIRCULAR LINKED LIST AND DOUBLY LINKED LIST

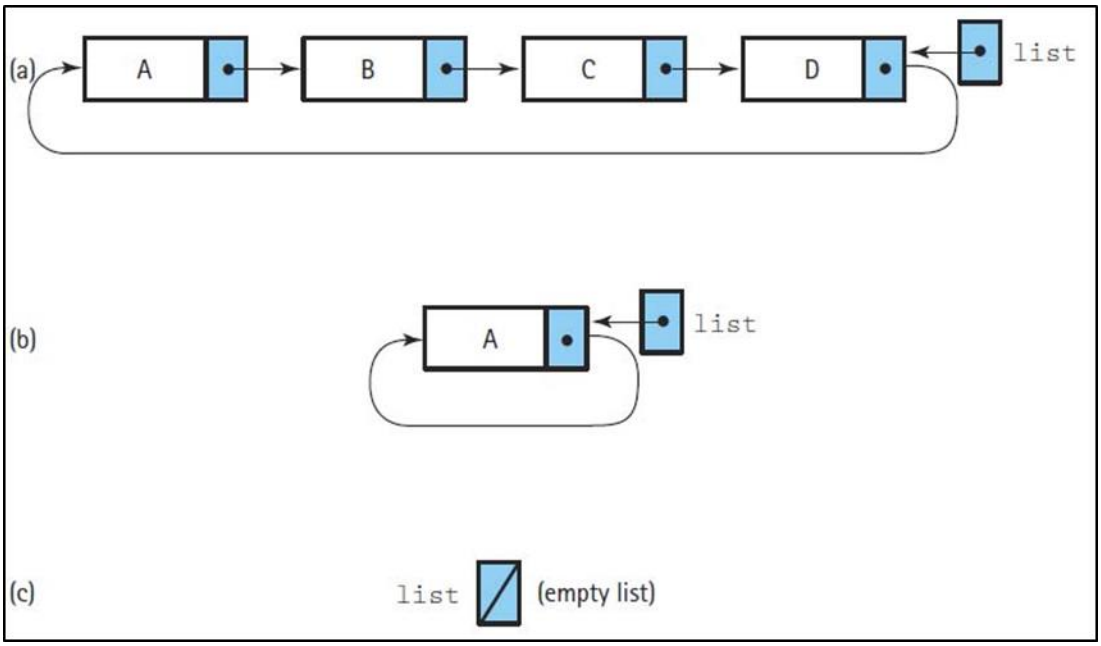
- The linked lists that we implemented in previous discussion are characterized by a linear (linelike) relationship between the elements: Each element (except the first one) has a unique predecessor, and each element (except the last one) has a unique successor.
  - Suppose we change the linear list slightly, making the next reference of the last node point back to the first node instead of containing null as shown in the following figure.



A circular linked list

- Now our list is a circular linked list rather than a linear linked list. We can start at any node in the list and traverse the whole list.
- Of course, we must now ensure that all of our list operations maintain this new property of the list:
  - That after the execution of any list operation, the last node continues to point to the front node.
  - For example, if we try to delete the first element. Our previous delete approach (linear linked list) would simply change the list reference to point to the second element on the list, effectively removing the first element. Now, however, we must also update the reference in the last element on the list, so that it points to the new first element. The only way to do that is to traverse the entire list to obtain access to the last element, and then make the change.
  - A similar problem arises if we insert an item into the front of the list.
- Inserting and deleting elements at the front of a list might be a common operation for some applications.

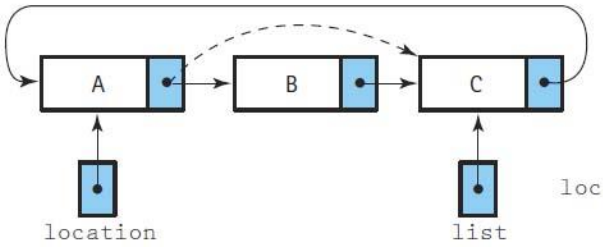
- Our linear linked list approach supported these operations very efficiently, but our circular linked list approach does not.
- We can fix this problem by letting our list reference point to the last element in the list rather than the first; now we have direct access to both the first and the last elements in the list as shown in the following figure.



Circular linked list with the external pointer pointing to the rear element

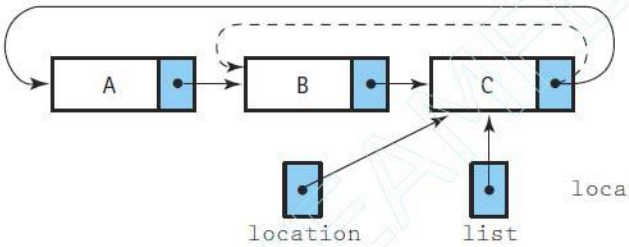
- Based on the above figure, where `list.info` references the information in the last node, and `list.next.info` references the information in the first node.
- We can use the same basic approach to deleting an element from a circular list as we used for a linear list.
  - First, find the element that matches the targeted item and then delete it. To delete it we unlink it from the chain of elements by setting the next reference of the element previous to the identified element to reference the element after the identified element. We have to consider general cases and special cases as shown in the following diagram.

(a) The general case (delete B)



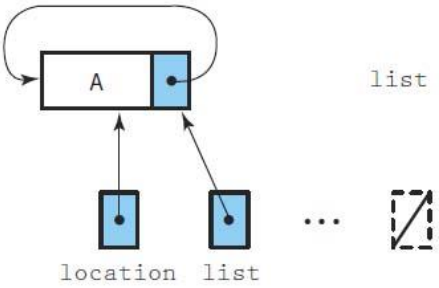

```
location.next = location.next.next;
```

(b) Special case (?): deleting the smallest item (delete A)



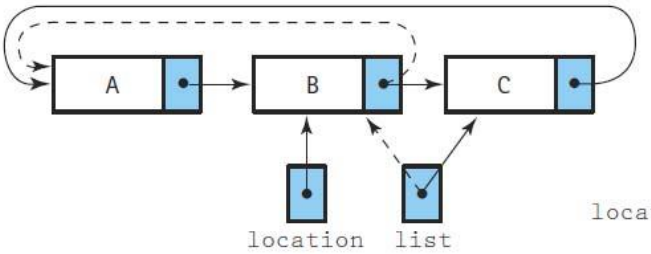

```
location.next = location.next.next;
```

(c) Special case: deleting the only item (delete A)




```
list = null;
```

(d) Special case: deleting the largest item (delete C)

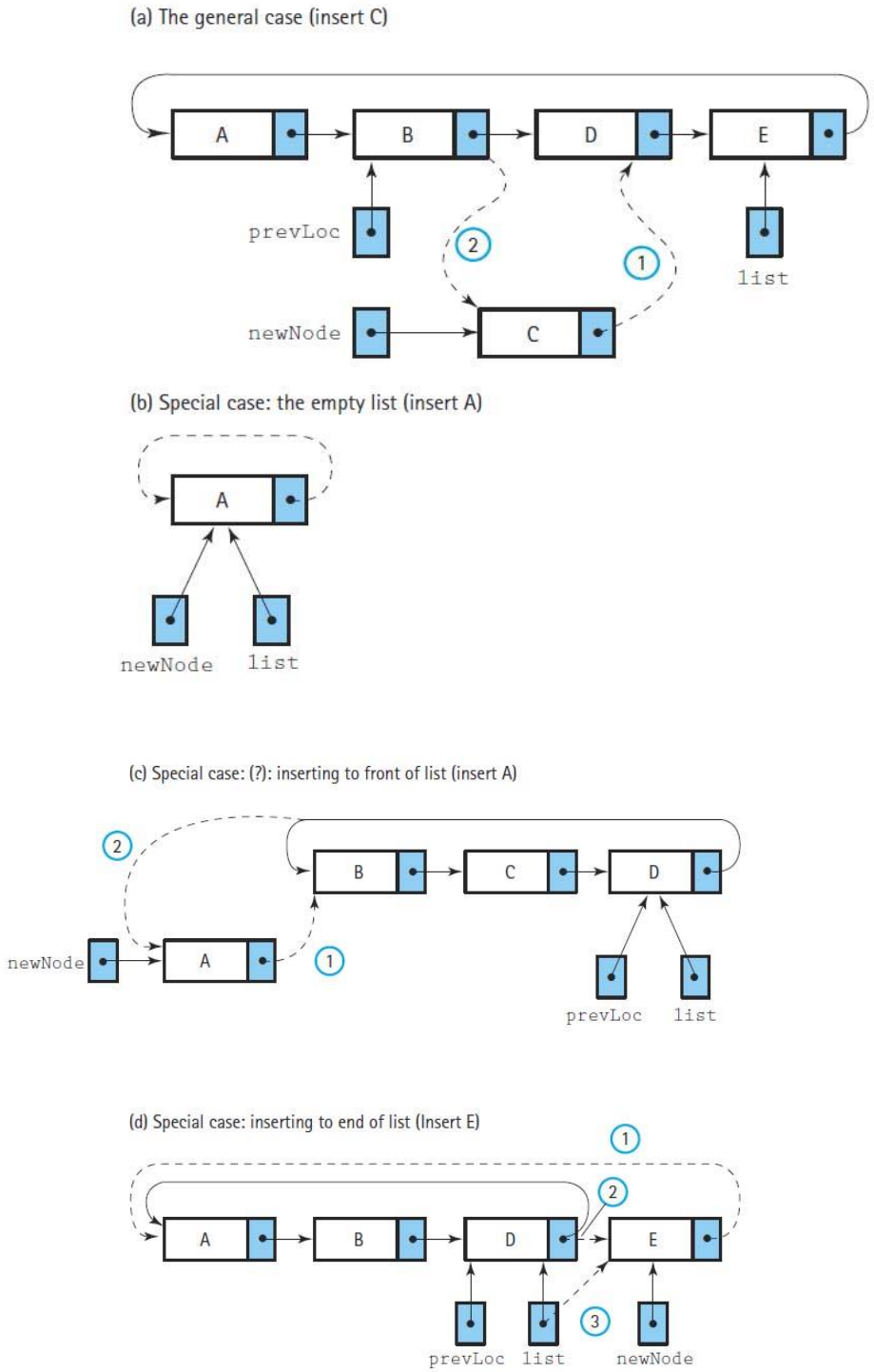



```
list = location;
```
(the general case PLUS:)
```
location.next = location.next.next;
```

Deleting from a circular linked list

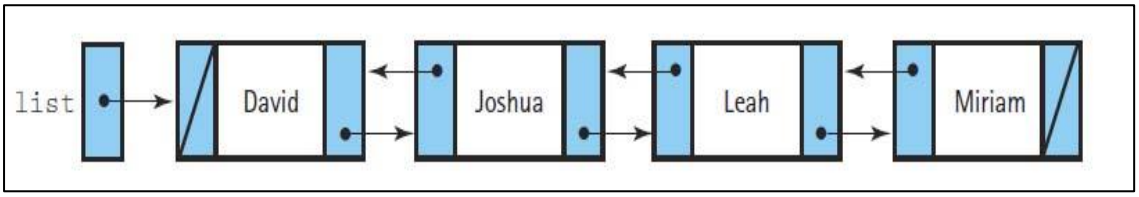
- The algorithm to insert an element into a circular linked list is also similar to its linear list counterpart.

- Essentially, we find the insertion location by performing a search, and insert the new item by rearranging some references.
- The following figure shows the insertion process for general and special cases.



Insertion into circular linked list

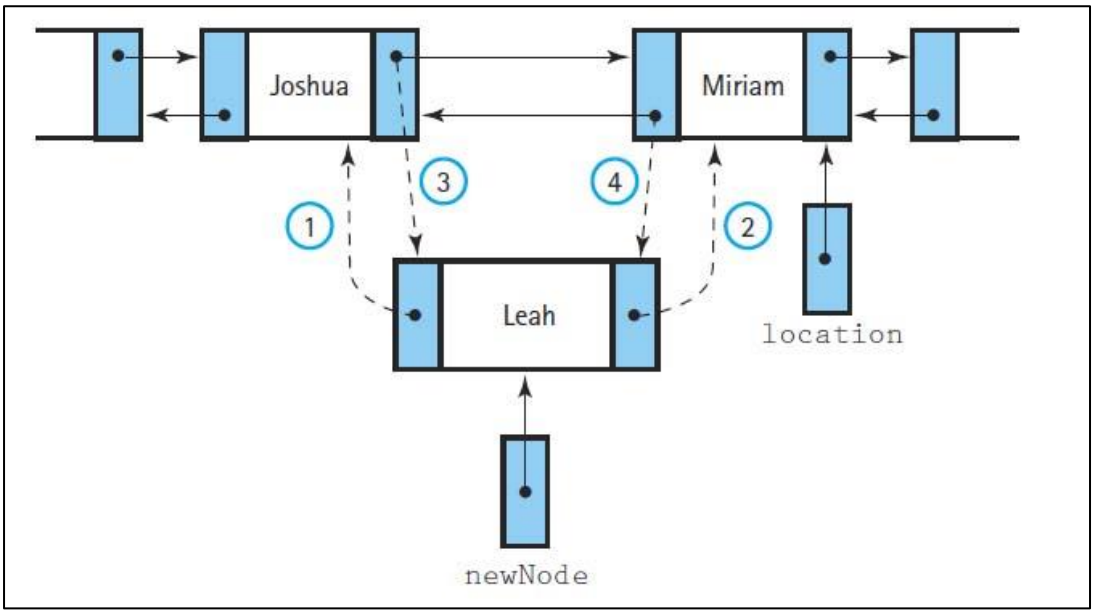
- Another task that is difficult to perform on a linear linked list (or even a circular linked list) is traversing the list in reverse.
- For instance, suppose we have a list of student records, sorted by grade point average (GPA) from lowest to highest. The Dean of Students might want a printout of the students' records, sorted from highest to lowest, to use in preparing the Dean's List.
  - In that application the user can step through a list of student information, viewing the information student by student on the screen, by pressing a "next" button. Suppose the user requests an enhancement to the interface— the idea is to include a "previous" button so that the user can browse through the students in either direction.
  - In cases like these, where we need to be able to access the node that precedes a given node, a doubly linked list is useful.
  - In a doubly linked list, the nodes are linked in both directions as shown in the following figure.



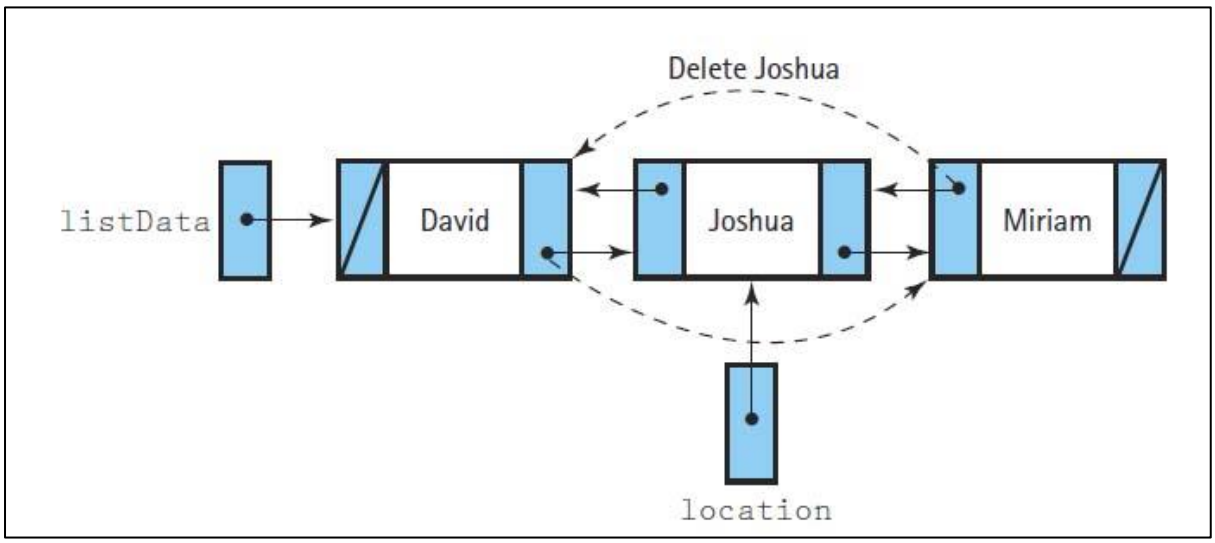
A linear doubly linked lsit

  - Each node of a doubly linked list contains three parts :
    - `info`: the data stored in the node
    - `next`: the reference to the following node
    - `back`: the reference to the preceding node

  - Although our search phase is simpler, the algorithms for the insertion and deletion operations on a doubly linked list are somewhat more complicated than for a singly linked list. The reason is clear: There are more references to keep track of in a doubly linked list as shown in the following figure.



Inserting into doubly linked list



Deleting form a doubly linked list

PAST YEAR EXAM QUESTION – Sample 1

Given the following `Worker` and `LinkedList` ADTs:

```
public class Worker
{
      private int empNumber;
      private String name;
      private String position;
      private double grossIncome;
      private double deduction;

      public Worker()   { … }
      public Worker(int e,String n, int p, double i, double d){ … }
      public String getEmpNumber()   { … }
      public String getName ()   { … }
      public int getPosition()  { … }
      public double getGrossIncome()  { … }
      public double getDeduction() { … }
      public String toString() { … }
}


public class LinkedList
{
      public LinkedList() {…}

      public void insert(Object elem) {…}

      public Object getFirst() {…}

      public Object getNext() {…}

      //definition for other methods
}
```

Write a java application program to solve the following problems:

a) Declare one (1) LinkedList named as `listWorker1` .

(1 marks)

b) Input twenty (20) workers into `listWorker1`.

(4 marks)

c) Calculate the amount of KWSP that has to deduct by the worker if the monthly deduction is 11% of nett income. Find and display the information of worker with the highest and the lowest KWSP deduction.

(6 marks)

Answer Scheme (Suggestion):

```
public class AppWorker
{
      public static void main(String args[])
      {
      a)    LinkedList listWorker1 = new LinkedList(); [1 marks]
                                                              (1 marks)

      b)    Scanner scanner = new Scanner (System.in);
            String lineSeparator = System.getProperty
                                   ("line.separator");
            scanner.useDelimiter (lineSeparator);

            for(int a = 0; a<20; a++)
            {
                  System.out.print ("Enter employee number: ");
                  int e = scanner.nextInt();
                  System.out.print ("Enter employee name: ");
                  String n = scanner.next();
                  System.out.print ("Enter employee position: ");
                  String p = scanner.next();
                  System.out.print ("Enter employee income: ");
                  double i = scanner.nextDouble();
                  System.out.print ("Enter employee deduction: ");
                  double d = scanner.nextDouble();
                                                      [3 marks]

                  Worker W = new Worker(e, n, p, I, d); [1/2 marks]
                  listWorker1.insert(W);                [1/2 marks]

            }
                                                              (4 marks)
      c)    Worker maxWorker = new Worker();
            Worker minWorker = new Worker();

            Object obj = listWorker1.getFirst();

            double kwsp;
            Worker Wor;
            double max, min;
            Wor = (Worker)obj;

            kwsp = (Wor.getIncome() -  Wor.getDeduction()) * 0.11;
            max = min = kwsp;
            maxWorker = minWorker = Wor;

            if (obj!=null)
            {
                  Wor = (Worker)obj;

                  if((Wor.getIncome() -  Wor.getDeduction()) * 0.11 >
                      max){
                        max = (Wor.getIncome()- Wor.getDeduction())*
                              0.11;
```

```
                maxWorker = Wor;
        }

        if((Wor.getIncome() -  Wor.getDeduction()) * 0.11 <
            min){
              min =(Wor.getIncome() -  Wor.getDeduction()) *
                    0.11;
              minWorker = Wor;
        }

}

obj = listWorker1.getNext();

while(obj != null)
{
      Wor = (Worker)obj;

      if((Wor.getIncome() -  Wor.getDeduction()) * 0.11 >
          max){
            max =(Wor.getIncome() -  Wor.getDeduction()) *
                    0.11;
            maxWorker = Wor;
      }

      if((Wor.getIncome() -  Wor.getDeduction()) * 0.11 <
          min){
            min =(Wor.getIncome() -  Wor.getDeduction()) *
                    0.11;
            90ineworker = Wor;
      }

      obj = listWorker1.getNext();
}

System.out.println("\n\nThe highest payment");
System.out.println(""+maxWorker.toString());

System.out.println("\n\nThe lowest payment");
System.out.println(""+90ineworker.toString());
```

(6 marks)

PAST YEAR EXAM QUESTION – Sample 2

Given the following `Worker` and `LinkedList` ADTs:

```
public class Invoice
{
      private int orderID;
      private String custName;
      private String prodName;
      private int prodQuantity;
      private double unitPrice;

      public Invoice()    { … }
      public void setData(int oid, String cn, String pn,
                          int pq, double up)   { … }
      public int getOrderID ()    { … }
      public String getCustName()    { … }
      public String getProdName ()    { … }
      public int getProdQuantity()   { … }
      public double getUnitPrice()   { … }
}

public class LinkedList
{
      public LinkedList() {…}

      public void insert(Object elem) {…}

      public Object getFirst() {…}

      public Object getNext() {…}

      //definition for other methods
}
```

Write a java application program to solve the following problems:

d) Input ten (10) invoices into a linked list.

(4 marks)

e) Display the information of invoice that makes the highest payment. The payment is calculated by multiplying `prodQuantity` and `unitPrice`. At the end of the process, the information of all invoices must remain in the original linked list.

(6 marks)

f) Count the number of invoices where the payment is more than RM 5000, and also display the information of those invoices.

(4 marks)

Answer Scheme (Suggestion):

```
public class AppInvoice
{
      public static void main(String args[])
      {
```

a)

```
½ marks     List listInv = new List();

            for(int i = 0; i<3; i++)
            {
              int oid = Integer.parseInt(
              JOptionPane.showInputDialog("Order ID: "));
2 marks       String cn = JOptionPane.showInputDialog("Customer Name: ");
              String pn = JOptionPane.showInputDialog("Product Name: ");
              int pq = Integer.parseInt(
                        JOptionPane.showInputDialog("Product Quantity: "));
              double up = Double.parseDouble(
                        JOptionPane.showInputDialog("Unit Price: "));

½ marks       Invoice inv = new Invoice();
½ marks       inv.setData(oid,cn,pn,pq,up);
½ marks       listInv.insert(inv);
            }
```

b)

```
            Invoice maxInvoice = new Invoice();
            List listTemp = new List();
1 marks     Object obj = listInv.remove();
            Invoice i = (Invoice)obj;

½ marks     double maxPayment = i.getProdQuantity() *  i.getUnitPrice();

½ marks     listTemp.insert(i);
            while(!listInv.isEmpty())
            {
              obj = listInv.remove();
              i = (Invoice)obj;

2 marks       if(i.getProdQuantity()*i.getUnitPrice() > maxPayment){
                  maxPayment = i.getProdQuantity()*i.getUnitPrice();
                  maxInvoice = i;
              }
              listTemp.insert(i);
            }

            System.out.println("\n\nThe highest payment");
            System.out.println("Order ID: "+maxInvoice.getOrderID());
            System.out.println("Customer name: "+maxInvoice.getCustName());
1 marks     System.out.println("Product name: "+maxInvoice.getProdName());
            System.out.println("Product Quantity: "
                       +maxInvoice.getProdQuantity());
            System.out.println("Unit Price: "+maxInvoice.getUnitPrice());
```

```
obj = listTemp.remove();
i = (Invoice)obj;

listInv.insert(i);
while(!listTemp.isEmpty())
{
      obj = listTemp.remove();
      i = (Invoice)obj;
      listInv.insert(i);
}
```

1 marks

c)

```
int cnt = 0;
while(!listInv.isEmpty())
{
      obj = listInv.remove();
      i = (Invoice)obj;

      if(i.getProdQuantity()*i.getUnitPrice() > 5000.00)
      {
        cnt += 1;
        System.out.println("\n\nThe highest payment");
        System.out.println("Order ID: "
                  +maxInvoice.getOrderID());
        System.out.println("Customer name: "
                  +maxInvoice.getCustName());
        System.out.println("Product name: "
                  +maxInvoice.getProdName());
        System.out.println("Product Quantity: "
                  +maxInvoice.getProdQuantity());
        System.out.println("Unit Price: "
                  +maxInvoice.getUnitPrice());
      }
}
System.exit(0);
    }
}
```

½ marks

½ marks

1 marks

½ marks

1 ½ marks

(14 marks)

CHAPTER 3

Stack

3.1 INTRODUCTION

- Try to think the following scenario:
  - In a cafeteria, you can see stacks of dishes placed in spring-loaded containers. Usually several dishes are visible above the top of the container, and the rests are inside the container.
  - You can access only the dish that is on top of the stack.
  - If you want to place more dishes on the stack, you can place the dishes on top of those that are already there.
  - Another example, at a library you can see stacks of books placed on the table, before it is categorized into the specific rack, such as fiction and non-fiction. You can access only the book that is on top of the stack if not, the book will collapse and so, if you want to place more books on the stack, you can place the book on the top of those already there. The following figure shows the stack application:

Stack of books

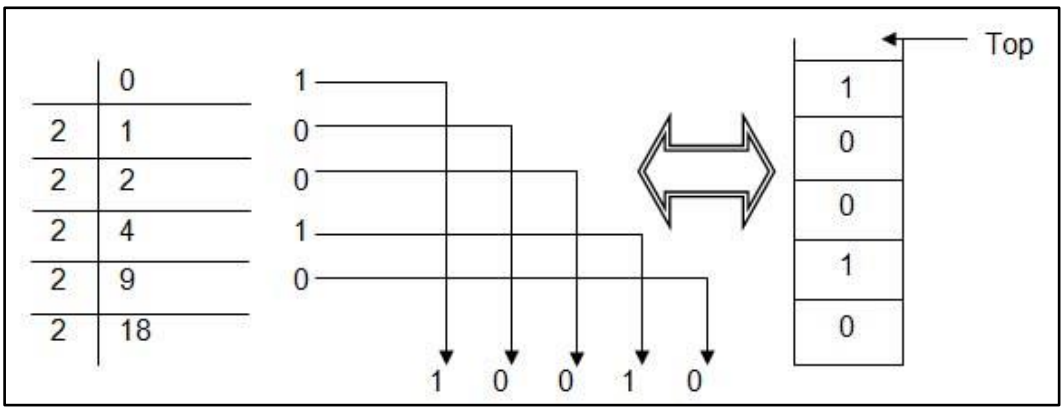
- In programming, a stack is a data structure with the property that only the top element of the stack is accessible.

STACK

A stack is a sequence of elements in which the only element that can be removed, accessed, or modified is the element that recently inserted: - That element is referred to as the **top** element on the stack.

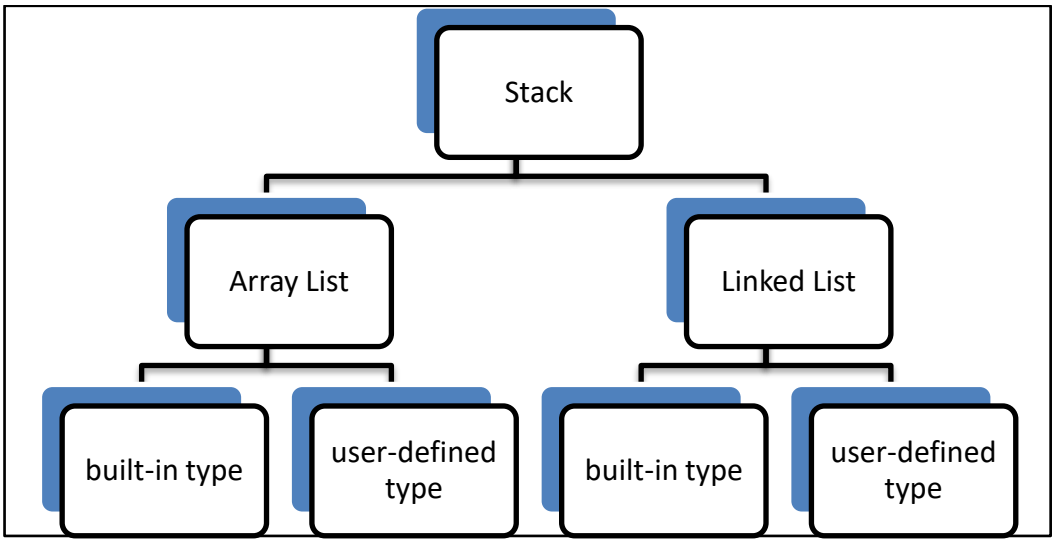
- For that reason, a stack is referred to as ***a last-in, first-out (LIFO)***.
- The primary methods for manipulating a stack are **push** and **pop**.
- Method **push** adds a new element to the top of the stack.
- Method **pop** removes a node from the top of the stack and returns the data from the popped list.
- ADT operations:
  - Create a new stack.
  - Determine empty stacks.
  - Insert new elements into the stack (push).
  - Remove elements from stack (pop).

- Example of the problem needed to use stack:
  - Conversion from decimal number to binary number.



Conversion from decimal to binary value

- The stack can be implemented by using the Array List or Linked List as shown in the following figure:



Stack implementation: Array list and Linked list

3.2 DESIGN AND IMPLEMENTATION OF THE `STACK` CLASS: *BUILT-IN* ARRAY LIST

- The stack class have six methods: ***push, pop, peek, size, isEmpty,*** and a ***default constructor***:
- The following table shows the methods of `Stack` class.

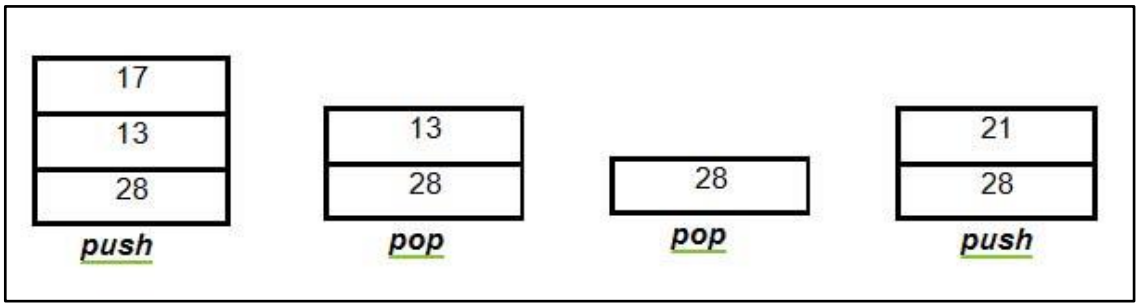
Method	Description
`public Stack();`	**Postcondition :** this Stack object is empty
`public int size();`	**Postcondition :** the number of elements in this Stack object has been returned.
`public boolean isEmpty();`	**Postcondition :** true has been returned if this Stack object has no elements. Otherwise, false has been returned
`public Object push (Object element);`	**Postcondition :** element has been inserted at the top of this Stack object and returned. The averageTime(n) is constant. The

	worstTime(n) is O(n), but for n consecutive calls to push, worstTime(n) – for the entire sequence - is still only O(n).
public Object pop();	**Precondition :** this Stack object is not empty. Otherwise, an exception will be thrown **Postcondition :** the element that had been at the top of this Stack object before this method was called has been removed and returned.
public Object peek()	**Precondition :** this Stack object is not empty. Otherwise, an exception will be thrown **Postcondition :**(a reference to) the top element in this Stack object has been returned.

The Stack class

- **Example:**

```
Stack stack = new Stack();
stack.push(new Integer(28));
stack.push(new Integer(13));
stack.push(new Integer(17));
stack.pop();
stack.pop();
stack.push(new Integer(21));
```



Stack representation illustrated using Diagram

EXAMPLE OF PROGRAM – PRIMITIVE DATA TYPE

To store primitive data type objects onto stack. This program shows and manipulates some of the methods in Stack class.

```
import java.util.*;

public class StackTest
{
    public StackTest()
    {
       Stack stack = new Stack();
       //create objects to store in the stack
       Boolean bool = Boolean.TRUE;
       Character character = new Character('$');
       Integer integer = new Integer(34567);
       String string = "hello";

       //use push method
       stack.push(bool);
       printStack(stack);
       stack.push(character);
       printStack(stack);
       stack.push(integer);
       printStack(stack);
       stack.push(string);
       printStack(stack);

      //remove items from stack
      try {
          Object removedObject = null;

          while(true){
              removedObject = stack.pop(); //use pop method

              System.out.println(removedObject.toString()
                     + " poped");
              printStack(stack);
          }
      }

      catch (EmptyStackException emptyStackException) {
                    emptyStackException.printStackTrace();
      }
    }

    private void printStack(Stack stack)
    {
      //the stack is empty
      if(stack.isEmpty() )
        System.out.print("stack is empty");
```

```
        else{
          System.out.print( "stack contains: ");
          Enumeration items = stack.elements();


          while( items.hasMoreElements())
             System.out.print(items.nextElement() + " ");
        }
        System.out.println("\n");  //go to the next line
      }

      public static void main(String[] args)
      {
          StackTest st = new StackTest();
      }

  }//end class StactTest
```

3.3 DESIGN AND IMPLEMENTATION OF THE *STACK* CLASS: *BUILT-IN* LINKED LIST

- Methods and constructor use for the stack in `LinkedList` class as shown in the following table.

Method
1. public LinkedList();
2. public LinkedList(Collection c);
3. public Object getFirst();
4. public Object getLast();
5. public void addFirst(Object o);
6. public void addLast(Object o);
7. public void removeFirst();
8. public void removeLast();

Methods in `LinkedList` class

EXAMPLE OF PROGRAM – PRIMITIVE DATA TYPE

To store integer numbers onto stack using the linked list. This program manipulates some of the methods in `LinkedList` class inorder to implement the stack linked list.

```
//LinkedListStack.java - A Stack built from A LinkedList
import java.util.LinkedList;
import java.util.EmptyStackException;

public class LinkedListStack
{
   private LinkedList listDelegate = new LinkedList();
   public boolean empty()
   {
      return listDelegate.isEmpty();
   }
   //return the element on top of the stack without
   //removing it
   public Object peek()
   {
      if(!this.empty()) {
        return listDelegate.getFirst();
      }
      throw new EmptyStackException();
   }

   //return the element on top of the stack and remove it
   public Object pop()
   {
      if(!this.empty()) {
        return listDelegate.removeFirst();
      }
      throw new EmptyStackException();
   }

   //place an object on top of the stack
   public Object push(Object item)
   {
     listDelegate.addFirst(item);
     return item;
   }

  //look for an object in the stack
  public int search(Object item)
  {
     return listDelegate.indexOf(item);
  }
```

```
  public static void main(String[] args)
  {
     LinkedListStack stack = new LinkedListStack();

     //push 5 elements onto the stack
     for (int i=0; i<5; i++)
     {
         stack.push(new Integer(i));
     }

     System.out.println("Peek at the top of the stack: "
            +stack.peek() );
     // empty the stack
     while( !stack.empty())
     {
       System.out.println("Pop: " +stack.pop() );
     }
   }
}
```

3.4 DESIGN AND IMPLEMENTATION OF THE *STACK* CLASS: *USER DEFINED TYPE* LINKED LIST.

EXAMPLE OF PROGRAM 1– PRIMITIVE DATA TYPE

To store primitive data type objects onto stack using the user defined linked list. This program shows and manipulates some of the user defined methods in order to implement stack linked list.

```
// ListNode class to represent one node in a list
public class ListNode {
   // package access members; List can access these directly
   Object data;
   ListNode nextNode;

   // create a ListNode that refers to object
   ListNode( Object object )
   {
      this( object, null );
   }

   // create ListNode that refers to Object and to next ListNode
   ListNode( Object object, ListNode node )
   {
      data = object;
      nextNode = node;
   }

   // return reference to data in node
   Object getObject()
   {
      return data; // return Object in this node
   }

   // return reference to next node in list
   ListNode getNext()
   {
      return nextNode; // get next node
   }

} // end class ListNode

//*************************************************************

// class List definition
public class List {
   private ListNode firstNode;
   private ListNode lastNode;
   private String name;  // string like "list" used in printing
```

```
   // construct empty List with "list" as the name
   public List()
   {
      this( "list" );
   }
   // construct an empty List with a name
   public List( String listName )
   {
      name = listName;
      firstNode = lastNode = null;
   }

   // insert Object at front of List
   public synchronized void insertAtFront( Object insertItem )
   {
      // firstNode and lastNode refer to same object
      if ( isEmpty() )
         firstNode = lastNode = new ListNode( insertItem );

      else // firstNode refers to new node
         firstNode = new ListNode( insertItem, firstNode );
   }

   // remove first node from List
   public synchronized Object removeFromFront() throws
   EmptyListException{

      if ( isEmpty() ) // throw exception if List is empty
         throw new EmptyListException( name );

      // retrieve data being removed
      Object removedItem = firstNode.data;

      // update references firstNode and lastNode
      if ( firstNode == lastNode )
         firstNode = lastNode = null;
      else
         firstNode = firstNode.nextNode;

      return removedItem; // return removed node data
   } // end method removeFromFront

   // determine whether list is empty
   public synchronized boolean isEmpty()
   {
      return firstNode == null; // return true if List is empty
   }

   // output List contents
   public synchronized void print()
   {
      if ( isEmpty() ) {
         System.out.println( "Empty " + name );
         return;
      }
```

```
      System.out.print( "The " + name + " is: " );
      ListNode current = firstNode;

      // while not at end of list, output current node's data
      while ( current != null ) {
         System.out.print( current.data.toString() + " " );
         current = current.nextNode;
      }
      System.out.println( "\n" );
   }
} // end class List


//*************************************************************

// Class EmptyListException definition.

public class EmptyListException extends RuntimeException {
   // no-argument constructor
   public EmptyListException()
   {
      this( "List" );// call other EmptyListException constructor
   }

   // constructor
   public EmptyListException( String name )
   {
      super( name + " is empty" ); // call superclass constructor
   }

} // end class EmptyListException


//*************************************************************
// Derived from class List
public class StackInheritance extends List
{
  //Construct stack
  public StackInheritance()
  {
      super("stack");
  }

  //add object to stack
  public synchronize void push(Object object)
  {
      insertAtFront(object);
  }

  //remove object from stack
  public synchronized Object pop() throws EmptyListException
  {
      return removeFromFront();
  }
}
```

```
//*************************************************************

public class StackInheritanceTest
{
    public static void main(String[] args)
    {
        StackInheritance stack = new StackInheritance();

       //create objects to store in the stack
       Boolean bool = Boolean.TRUE;
       Character character = new Character( '$' );
       Integer integer = new Integer( 34567 );
       String string = "hello";

       //use push method
       stack.push( bool );
       stack.print();
       stack.push( character );
       stack.print();
       stack.push( integer );
       stack.print()
       stack.push( string );
       stack.print();

       //removed items from stack
       try {
           Object removedObject = null;

           while(true) {
              removedObject = stack.pop();  // use pop method
              System.out.println( removedObject.toString()
                     + " popped" );
              Stack.print();
           }
       }

       //catch exception if stack is empty when item popped
       catch ( EmptyListException emptyListException ) {
            emptyListException.printStackTrace();
       }
   }
} // end class StackInheritanceTest
```

EXAMPLE OF PROGRAM 2 – PRIMITIVE DATA TYPE

```
/*_Author: Mazidah Puteh
Objective:
         1. Push integer value into the stack
         2. Display all numbers in the stack
         3. Calculate and display sum of all numbers
*/

class EmptyListException extends RuntimeException {
   public EmptyListException( String name )
   {
      super( "The " + name + " is empty" );
   }
}

class ListNode {
   // package access data so class List can access it directly
   Object data;
   ListNode next;

   // Constructor: Create a ListNode that refers to Object o.
   ListNode( Object o ) { this( o, null ); }

   // Constructor: Create a ListNode that refers to Object o
   // and to the next ListNode in the List.
   ListNode( Object o, ListNode nextNode )
   {
      data = o;          // this node refers to Object o
      next = nextNode;  // set next to refer to next
   }

   // Return a reference to the Object in this node
   Object getObject() { return data; }

   // Return the next node
   ListNode getLink() { return next; }
}

// Class List definition
class LinkedList {
   private ListNode firstNode;
   private ListNode lastNode;
   private ListNode currNode; // use to traverse the list
   private String name;  // String like "list" used in printing

   // Constructor: Construct an empty List with s as the name
   public LinkedList( String s )
   {
      name = s;
      firstNode = lastNode = currNode = null;
   }
```

```
   // Constructor: Construct an empty List with "list"
   // as the name
   public LinkedList() { this( "list" ); }

   // Insert an Object at the front of the List
   // If List is empty, firstNode and lastNode will refer to
   // the same object. Otherwise, firstNode refers to new node.
   public void insertAtFront( Object insertItem )
   {
      if ( isEmpty() )
         firstNode = lastNode = new ListNode( insertItem );
      else
         firstNode = new ListNode( insertItem, firstNode );
   }

   // Insert an Object at the end of the List
   // If List is empty, firstNode and lastNode will refer to
   // the same Object. Otherwise, lastNode's next instance
   // variable refers to new node.
   public void insertAtBack( Object insertItem )
   {
      if ( isEmpty() )
         firstNode = lastNode = new ListNode( insertItem );
      else
         lastNode = lastNode.next = new ListNode( insertItem );
   }

   // Remove the first node from the List.
   public Object removeFromFront()throws EmptyListException
   {
      Object removeItem = null;

      if ( isEmpty() )
         throw new EmptyListException( name );

      removeItem = firstNode.data;  // retrieve the data

      // reset the firstNode and lastNode references
      if ( firstNode.equals( lastNode ) )
         firstNode = lastNode = null;
      else
         firstNode = firstNode.next;

      return removeItem;
   }

   // Remove the last node from the List.
   public Object removeFromBack()throws EmptyListException
   {
      Object removeItem = null;

      if ( isEmpty() )
         throw new EmptyListException( name );
```

```
      removeItem = lastNode.data;  // retrieve the data
      // reset the firstNode and lastNode references
      if ( firstNode.equals( lastNode ) )
         firstNode = lastNode = null;
      else {
         ListNode current = firstNode;

         while ( current.next != lastNode )  // not last node
            current = current.next;      // move to next node

         lastNode = current;
         current.next = null;
      }

      return removeItem;
   }

   // Return true if the List is empty
   public boolean isEmpty()
   { return firstNode == null; }

   // Return First element
   public Object getFirst()
   {
     if (isEmpty())
        return null;
     else
     {
       currNode = firstNode;
       return currNode.data;
     }
   }

   public Object getNext()
   {
     if (currNode != lastNode)
     {
         currNode = currNode.next;
         return currNode.data;
     }
     else
         return null;
   }

   // Output the List contents
   public void print()
   {
      if ( isEmpty() ) {
         System.out.println( "Empty " + name );
         return;
      }

      System.out.print( "The " + name + " is: " );

      ListNode current = firstNode;
```

```
      while ( current != null ) {
         System.out.print( current.data.toString() + " " );
         current = current.next;
      }

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

class PrimStack extends LinkedList {
   public PrimStack() { }

   public void Push(Object elem)
   {
     insertAtFront( elem);
   }

   public Object Pop ()
   {
     return removeFromFront();
   }

   public Object Peek ()
   {
     return getFirst();
   }
}

public class PrimitiveStack {
   public static void main( String args[] )
   {
      // create the List container
      PrimStack stack = new PrimStack();
      // Create objects to store in the List
      String a = "10";
      String b = "12";
      String c = "14";
      String d = "16";

      // Use the List insert methods
      stack.Push( a );
      stack.Push( b );
      stack.Push( c );
      stack.Push( d );

      //To display top data
      System.out.println("Top: "+stack.Peek().toString());

     //To sum all numbers in Stack
      int sum = 0;
      int num;
      System.out.println("Contents of stack: \n");

      while (!stack.isEmpty())
```

```
      {
         num = Integer.parseInt(stack.Pop().toString());
         sum += num;
         System.out.println (""+num);
      }

      System.out.println ("SUM of all numbers: "+sum);
      if (stack.isEmpty())
        System.out.println ("Stack is Empty..");
      else
       System.out.println ("Stack is Not Empty..");
   }
}
```

3.5 STACK APPLICATION FOR MATHEMATIC EXPRESSION: INFIX, PREFIX AND POSTFIX

- For the most high-level-language, arithmetic expression is written in infix notation which the operator is placed between its operands.
- **Example:**

  (i) A + B – C

  (ii) A * B / C

- Most of the compiler will change the infix notation into postfix notation which the operator will be placed after both operands.
- **Example:**

  A + B → *infiks*

  A B + → *posfiks*

- In infix notation, the parenthesis will determine the precedence of the operator.
- **Example:**

  5 * (2 + 3) → infix

  5 2 3 + * → postfix

- **How to evaluate postfix notation?**
  - **Example:** 16 2 / 5 3 - +
    - The suitable infix notation:- 16/2 + (5 – 3)
    - The evaluation rules for postfix:
      1. Evaluate expression from left to right.
      2. At each occurrence of an operator, apply it to the two operands to the immediate left and replace the sequence of two operands and one operator with the resulting value.

    - First operator is ***/*** and its operands are **16 and 2.**

      16 2 / 5 3 - +

      ✓ Replace the value with **8 (16 / 2)**. The expression will look like as follows:

      8 5 3 - +

    - The second operator is **–** and its operands are **5 and 3**

      8 **5 3 -** +

      ✓ Replace the value with **2 (5 - 3)**. The expression will look like as follows:

      8 2 +

    - Last operator is **+** and its operands are **8 and 2**

      8 2 +

      ✓ Replace the value with **10 (8 + 2)**. The expression will look like as follows:

      10

    - The result is **10**

3.5.1 Algorithm For Evaluating Postfix Notation

- The evaluation rules for postfix notation can be elegantly expressed using the stack.
- Each time an operand is encountered, it is pushed onto the stack; upon reaching an operator, the last two operands are popped off the stack, and the operator is applied.

Algorithm:

1. Create an empty stack.

2. Traversal until to the end of expression.

2.1 Get the next postfix notation token.

2.2 If the token is an operand,

2.2.1 Push the operand onto the stack.

2.3 If the token is an operator,

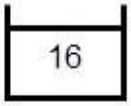
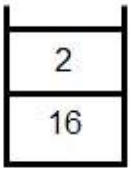
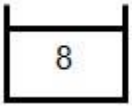
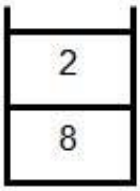
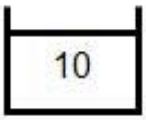
2.3.1 The last two operands are popped off the stack (if less than 2 operands- throws exception errors). Pop the first operand off the stack and placed on the right and second operand on the left.

2.3.2 Evaluate the operands based on the operator.

2.3.3 Pushed the value onto stack.

2.4 *If reach at the end of the postfix notation, pop the stack and return the result :- the result (value) at the top of the stack.*

Algorithm to evaluate postfix notation

Expression	Stack	Description
16 2 / 5 3 - +		Push 16 onto the stack
2 / 5 3 - +		Push 2 onto the stack
/ 5 3 - +		Pop off 2 & 16 from stack, get the result of dividing 16/2, push the result (8) onto the stack
5 3 - +		Push 5 onto stack
3 - +		Push 3 onto the stack
- +		Pop off 3 & 5 from stack, get the result of subtracting 5 – 3, push the result (2) onto the stack
+		Pop off 2 & 8 from stack, get the result of adding 8 + 2, push the result (10) onto the stack

Evaluating postfix notation using stack illustrated by the diagram

3.5.2 Algorithm To Convert Infix Notation Into Postfix Notation

- Read infix notation from left to right.
- At each occurrence of an operand, display its as output.
- Each time an operator is encountered; it is pushed onto the stack, but it depends on the precedence of operators.

Algorithm:

1. Create an empty stack.

2. While no error and not at the end of expression.

2.1 Get the next token.

2.2 If the token is

2.2.1 ' (' : push the operator onto stack.

2.2.2 ') ' : popped off all elements from the stack and display the results until the operator ' (' is reached, but don't need to display ' (' – throws exception if the '(' couldn't find.

*2.2.3 Operator *, +, -, /*

If empty stack OR the token has a higher precedence than the token on the top element stack:

- *Pushed the token onto the stack.*

else

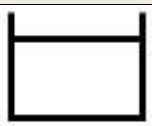
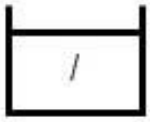
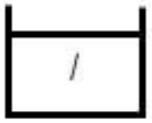
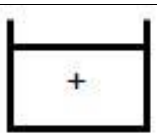
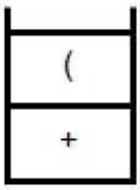
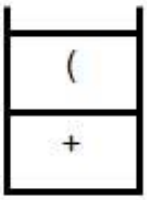
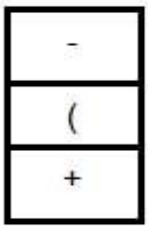
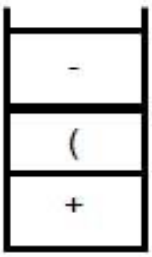
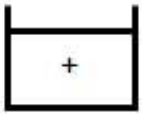
While the stack is not empty and the precedence of the token is less than or equal to the precedence on the top element stack

- *Popped off the elements from the stack and display.*

2.2.4 Operand: display.

2.3 *If the end of the stack, popped off elements from the stack and display.*

Algorithm to convert infix notation to postfix notation

Expression	Stack	Output	Description
16 / 2 + (5 - 3)		16	Display 16
/ 2 + (5 - 3)		16	Push / onto the stack
2 + (5 - 3)		16 2	Display 2
+ (5 – 3)		16 2 /	Pop / and display. Push + onto the stack
(5 – 3)		16 2 /	Push (onto the stack
5 – 3)		16 2 / 5	Display 5
- 3)		16 2 / 5	Push - onto the stack
3)		16 2 / 5 3 -	Display 3
)		16 2 / 5 3 -	Pop off all elements until pop (off of stack and display.
		16 2 / 5 3 - +	Pop + off and display

Evaluating infix notation using stack illustrated by the diagram

PAST YEAR EXAM QUESTION – Sample 1

Given the following `Motorcycle`, `Queue` and `Stack` ADTs:

```
public class Motorcycle
{
      private String brand; //brand of motorcycle e.g: Honda, Kriss
      private int year; //year of registration
      private double price; //price of motorcycle

      public Motorcycle(){…}
      public void setData(String b, int y, double p){…}
      public String getBrand() {…}
      public int getYear() {…}
      public double getPrice() {…}

}

public class Queue
{
      public Queue()  {…}
      public void enqueue(Object elem) {…}
      public Object dequeue()  {…}
      public boolean isEmpty() {…}

//definition for other methods
}

public class Stack
{
      public Stack()  {…}
      pulic void push(Object elem)  {…}
      public Object pop()  {…}
      public boolean isEmpty()  {…}

//definition for other methods
}
```

Write a Java application to solve the following problems:

a) Create a Queue object named as `qMotorcycle`.

(1 marks)

b) Create two Stack objects named as `sHonda` and `sOthers`.

(2 marks)

c) Input thirty (30) motorcycles into `qMotorcycle`.

(3 marks)

d) Get all motorcycles from `qMotorcycle`. Store Honda motorcycles into a stack called `sHonda` and other motorcycles into a stack called `sOthers`. At the end of the process, all objects must remain in the original queue `qMotorcycle`.

(6 marks)

e) Count the number of motorcycles where the year of registration is greater than 2002 and also display the information of those motorcycles.

(4 marks)

Suggestion Answer Scheme:

```
public class AppQSMot
      {
            public static void main(String args[])
            {
```

a)

1 marks

```
Queue qMotorcycle = new Queue();
```

b)

1 marks
1 marks

```
Stack sHonda = new Stack();
Stack sOthers = new Stack();
```

c)

½ marks
1 marks
½ marks
½ marks
½ marks

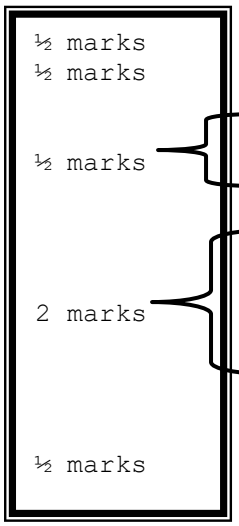
```
for(int i=0; i<30; i++)
{
  String brand = JOptionPane.showInputDialog("Brand: ");
  int year = Integer.parseInt(
             JOptionPane.showInputDialog("Year: "));
  double price = Double.parseDouble(
             JOptionPane.showInputDialog("Price: "));

  Motorcycle m= new Motorcycle();
  m.setData(brand,year,price);
  qMotorcycle.enqueue(m);
}
```

d)

½ marks
½ marks
½ marks
½ marks
½ marks
1 marks
1 marks
½ marks
½ marks
½ marks

```
Queue qTemp = new Queue();
while(!qMotorcycle.isEmpty())
{
  Object obj = qMotorcycle.dequeue();
  Motorcycle mot = (Motorcyle)obj;
  qTemp.enqueue(mot);

  if(mot.getBrand().equalsIgnoreCase("Honda"))
      sHonda.push(mot);
  else
      sOthers.push(mot);
}

while(!qTemp.isEmpty())
{
  Object obj = qTemp.dequeue();
  qMotorcycle.enqueue(obj);
}
```

e)




```
int cnt = 0;
while(!qMotorcycle.isEmpty())
{
  Object obj = qMotorcycle.dequeue();
  Motorcycle mot = (Motorcycle)obj;

  if(mot.getYear() >2002){
      cnt += 1;
      System.out.println("Brand: "+mot.getBrand());
      System.out.println("Year: "+mot.getYear());
      System.out.println("Price: "+mot.getPrice());
  }

}
System.out.println("\nThe number of motorcycle for year
       grater than 2002 :" +cnt);

System.exit(0);
    }
  }
```

(16 Marks)

PAST YEAR EXAM QUESTION – Sample 2

Given the following arithmetic expressions:

Y1= (A+B) * (C/D * E + F) / (G-H)

Y2 = A B * C - D / E F * +

i. Convert the infix expression Y1 to postfix and prefix notations.

(4 marks)

ii. Evaluate the expression Y2 by showing the contents of the stack if the following values are given:
A = 2, B = 6, C = 3, D = 3, E = 5, F =5

(4 marks)

Suggestion Answer Scheme:

i)

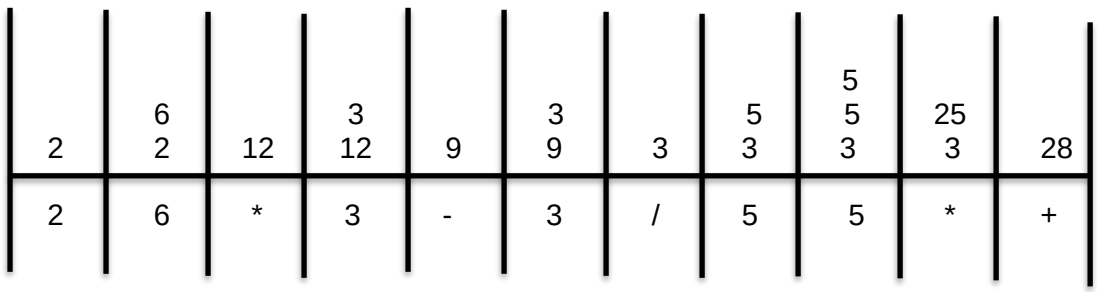
postfix : A B + C D / E * F + * G H / /

(2 marks)

prefix : / * + A B + * / C D E F / G H

(2 marks)

ii)



(4 marks)

CHAPTER 4

Queue

4.1 INTRODUCTION

- The easiest way to visualize a queue is to think of a line of customers waiting for service, for example like standing in line at a movie theater counter as shown in the figure below:

Queue: customer waiting for service

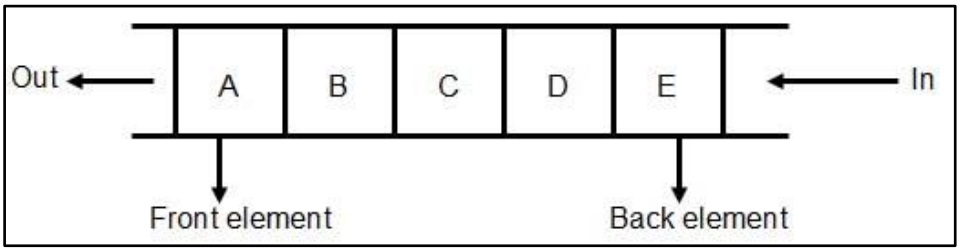
- Usually, the next person to be served is the one who has been waiting the longest, and the latecomers are added to the end of the line.
- The queue gets its name from the fact that such a waiting line is called a queue in English-speaking countries.

4.2 QUEUE CONCEPTS

- Like the stack, the queue is a list-like structure that provides restricted access to its elements.
- Queue elements may only be inserted at the back (called an **enqueue** operation) and removed from the front (called a **dequeue** operation).
- For this reason, a queue is a **FIRST-IN, FIRST-OUT (FIFO)** data structure.

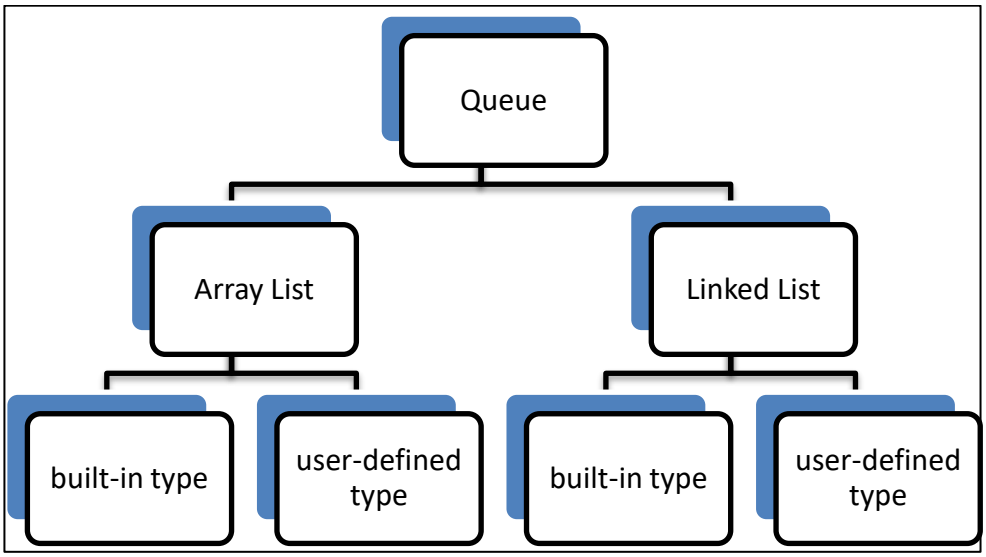
- So we can simplifies that a queue is a sequence of elements in which:
  i) Insertion is allowed only at the back of the sequence.
  ii) Removal, retrieval, and modification are allowed only at the front of the sequence.
  iii) **ENQUEUE** – The term used for inserting an element at the back of a queue.
  iv) **DEQUEUE** – The term used for removing an element from a queue.
  v) **FRONT** – The term used to access that first element in a queue.

- **Example:**



Queue represented using array diagram

- The queue can be implemented using array based and linked list, as shown in the figure below:



Queue implementation

4.3 DESIGN AND IMPLEMENTATION OF THE QUEUE CLASS: *BUILT-IN* ARRAY LIST

- There are six methods that queue class must have as shown in the following table:

Method	Description
`public Queue();`	**Postcondition**: this Queue object has been initialized
`public int size();`	**Postcondition**: the number of elements in this Queue object has been returned.
`public boolean isEmpty();`	**Postcondition**: true has been returned if this Queue object has no elements. Otherwise, false has been returned.
`public void enqueu(Object element);`	**Postcondition**: A copy of element has been inserted at the back of this Queue object. The averageTime(n) is constant and worstTime(n) is O(n)
`public Object dequeue();`	**Precondition**: this queue object is not empty. Otherwise, NoSuchElementException will be thrown **Postcondition**: The element that was at the front of this Queue object, just before this method was called- has been remove from this Queue object and returned. The worstTime(n) is constant.
`public Object front();`	**Precondition**: This Queue object is not empty. Otherwise, NoSuchElementException will be thrown **Postcondition**: The element at the front of this Queue object has been returned. The worstTime (n) is constant.

The built-in `Queue` class

- **Example:**

```
Queue queue = new Queue()
queue.enqueue ("Brian");
queue.enqueue ("Jane");
queue.enqueue ("Karen");
queue.enqueue("Bob");
queue.dequeue ();
```

4.4 DESIGN AND IMPLEMENTATION OF THE *QUEUE* CLASS: *BUILT-IN* LINKED LIST

- The following table shows the methods and constructor for Queue in `LinkedList` class:

Method
1. public LinkedList();
2. public LinkedList(Collection c);
3. public Object getFirst();
4. public void addLast(Object o);
5. public void removeFirst();

Method for Queue in `LinkedList` class

EXAMPLE OF PROGRAM – PRIMITIVE DATA TYPE

```
//Queue.java - A Queue built from A LinkedList
import java.util.*;

public class Queue
{
   protected LinkedList list ;

   public Queue()
   {
     list = new LinkedList();
   } // default constructor

   public boolean isEmpty()
   {
      return list.isEmpty();
   }  // method isEmpty
```

```
    public int size()
    {
       return list.size();
    }  //method size

    public void  enqueue (Object element)
    {
       list.addLast(element);
    }  //method enqueue

    public Object dequeue()
    {
       return list.removeFirst();
    }  //method dequeue

    public Object front ()
    {
       return list.getFirst();
    }  //method front

    public static void main(String[] args)
    {
        Queue queue = new Queue();

        //push 5 elements onto queue stack
        for (int i=0; i<5; i++)
        {
            queue.enqueue(new Integer(i));
        }

        // empty the stack
        while( !queue.isEmpty())
        {
            System.out.println("value: " +queue.dequeue() );
        }
    }
}
```

4.5 DESIGN AND IMPLEMENTATION OF THE QUEUE CLASS: USER-DEFINED TYPE LINKED LIST

EXAMPLE OF PROGRAM– PRIMITIVE DATA TYPE

```
/*    Author     : Mazidah Puteh
      Objective  :
                 1.  Enqueue integer value into queue
                 2.  Display all numbers in the queue
                 3.  Calculate and display sum of all numbers
*/
class EmptyListException extends RuntimeException {
   public EmptyListException( String name )
   {
      super( "The " + name + " is empty" );
   }
}

class ListNode {
   // package access data so class List can access it directly
   Object data;
   ListNode next;

   // Constructor: Create a ListNode that refers to Object o.
   ListNode( Object o ) { this( o, null ); }

   // Constructor: Create a ListNode that refers to Object o and
   // to the next ListNode in the List.
   ListNode( Object o, ListNode nextNode )
   {
      data = o;          // this node refers to Object o
      next = nextNode;  // set next to refer to next
   }

   // Return a reference to the Object in this node
   Object getObject() { return data; }

   // Return the next node
   ListNode getLink() { return next; }
}



// Class List definition
class LinkedList {
   private ListNode firstNode;
   private ListNode lastNode;
   private ListNode currNode; // use to traverse the list
   private String name;  // String like "list" used in printing
```

```
   // Constructor: Construct an empty List with s as the name
   public LinkedList( String s )
   {
      name = s;
      firstNode = lastNode = currNode = null;
   }

   // Constructor: Construct an empty List with
   // "list" as the name
   public LinkedList() { this( "list" ); }

   // Insert an Object at the front of the List If List is empty,
   // firstNode and lastNode will refer to the same object.
   // Otherwise, firstNode refers to new node.
   public void insertAtFront( Object insertItem )
   {
      if ( isEmpty() )
         firstNode = lastNode = new ListNode( insertItem );
      else
         firstNode = new ListNode( insertItem, firstNode );
   }

   // Insert an Object at the end of the List. If List is empty,
   // firstNode and lastNode will refer to the same Object.
   // Otherwise, lastNode's next instance variable refers to
   // new node.
   public void insertAtBack( Object insertItem )
   {
      if ( isEmpty() )
         firstNode = lastNode = new ListNode( insertItem );
      else
         lastNode = lastNode.next = new ListNode( insertItem );
   }

   // Remove the first node from the List.
   public Object removeFromFront() throws EmptyListException
   {
      Object removeItem = null;

      if ( isEmpty() )
         throw new EmptyListException( name );

      removeItem = firstNode.data;  // retrieve the data

      // reset the firstNode and lastNode references
      if ( firstNode.equals( lastNode ) )
         firstNode = lastNode = null;
      else
         firstNode = firstNode.next;
      return removeItem;
   }
```

```
// Remove the last node from the List.
public Object removeFromBack() throws EmptyListException
{
   Object removeItem = null;

   if ( isEmpty() )
      throw new EmptyListException( name );

   removeItem = lastNode.data;  // retrieve the data
   // reset the firstNode and lastNode references
   if ( firstNode.equals( lastNode ) )
      firstNode = lastNode = null;
   else {
      ListNode current = firstNode;

      while ( current.next != lastNode )  // not last node
         current = current.next;      // move to next node

      lastNode = current;
      current.next = null;
   }

   return removeItem;
}

// Return true if the List is empty
public boolean isEmpty()
{ return firstNode == null; }

// Return First element
public Object getFirst()
{
    if (isEmpty())
        return null;
    else
    {
        currNode = firstNode;
        return currNode.data;
    }
}

public Object getNext()
{
    if (currNode != lastNode)
    {
        currNode = currNode.next;
        return currNode.data;
    }
    else
        return null;
}
```

```
   // Output the List contents
   public void print()
   {
      if ( isEmpty() ) {
         System.out.println( "Empty " + name );
         return;
      }
      System.out.print( "The " + name + " is: " );

      ListNode current = firstNode;


      while ( current != null ) {
         System.out.print( current.data.toString() + " " );
         current = current.next;
      }
      System.out.println( "\n" );
   }
}

class PrimQueue extends LinkedList {
      public PrimQueue() { }

      public void Enqueue(Object elem)
      {
            insertAtBack( elem);
      }

      public Object Dequeue()
      {
            return removeFromFront();
      }

      public Object getFront()
      {
            return getFirst();
      }

      public Object getEnd()
      {
            Object O = removeFromBack();
            insertAtBack(O);
            return O;
      }
}
```

```
public class PrimitiveQueue {
   public static void main( String args[] )
   {
      // create the List container
      PrimQueue queue = new PrimQueue();

      // Create objects to store in the List
      String a = "10";
      String b = "20";
      String c = "30";
      String d = "40";

      // Use the List insert methods
      queue.Enqueue( a );
      queue.Enqueue( b );
      queue.Enqueue( c );
      queue.Enqueue( d );

      //To display first and last data
      System.out.println("First: "+queue.getFront().toString());
      System.out.println("Last: "+queue.getEnd().toString())
     //To sum all numbers in Queue
      int sum = 0;
      int num;
      System.out.println("Contents of queue: \n");
      while (!queue.isEmpty())
      {
           num = Integer.parseInt(queue.Dequeue().toString());
           sum += num;
           System.out.println (" "+num);
      }

      System.out.println ("SUM of all numbers: "+sum);
      if (queue.isEmpty())
           System.out.println ("Queue is Empty..");
      else
           System.out.println ("Queue is Not Empty..");
   }
}
```

PAST YEAR EXAM QUESTION – Sample 1

Given the following `Queue` ADT and Java application:

```
public class Queue
{
     public void enqueue (Object elem) {…}
     public Object dequeue () {…}
     public boolean isEmpty () {…}

     //definition of other methods
}

public class QueueApp
{
     public static void main (String[] args)
     {
          int [] intArr = {6, 9, 13, 5, 11, 0, 7, 3, 19, 4, 16};
          LinkedQueue qA = new linkedqueue();
          LinkedQueue qB = new LinkedQueue();

          int i, n;

          for (i =0; i < intArr.lenth; i++)
                  qA.enqueue (intArr[i]);
          System.out.println("Size of qA is " +queueSize(qA));
          n = queueSize(qA);

          i = 1;
          while (i <= n/2)
          {
            qA.enqueue (qA.dequeue());
            i++;
          }
          while (i <= n)
          {
            qB.enqueue (qA.dequeue());
            i++;
          }
          System.out.println("Elements in qA: " + qA);
          System.out.println("Elements in qB: " + qB);
     }
            //***********
     System.exit(0);
}
```

a) Trace and state the output for the above program. Draw the diagram of all queues at line marked with //***********

(7 marks)

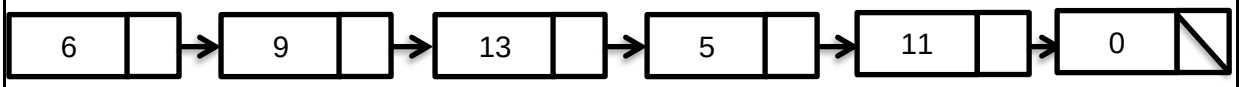
b) Write a Java program segment to append all numbers in qA and store them into qB. At the end of the process, qA must cntain all the numbers in its original order.

(6 marks)

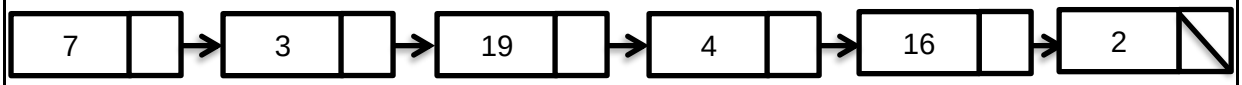
Answer Scheme (Suggestion):

a)

```
Size of qA is 12        [1 mark]

Elements in qA: [6, 9, 13, 5, 11, 0]      [1 mark]

Drawing the output in LinkedQueue for qA    [2 marks]
```




```
Elements in qB: [7, 3, 19, 4, 16, 2]      [1 mark]

Drawing the output in LinkedQueue for qB    [2 marks]
```



(7 marks)

b)

```
Queue temp = new Queue();
Object obj;
while (!qA.empty())       [1 mark]
{
      Obj = qA.dequeue();    [1 mark]
      N = (number) obj;
      if (N.append())
            qB.enqueue(N);      [1 mark]
      temp.enqueue(N);
}

while ( !temp.empty())      [1 mark]
{
       Obj = temp.dequeu();  [1 mark]
       qA.enqueue(obj);      [1 mark]
}
```

(6 marks)

PAST YEAR EXAM QUESTION – Sample 2

Given the following `ChildNursery`, `Queue` and `Stack` ADTs:

```
public class ChildNursery
{
      private String name;
      private String race;
      private int age;
      private String sex;

      public ChildNursery(){…}
      public void setData(String n, String r, int a, String s){…}
      public String getName() {…}
      public String getRace() {…}
      public int getAge() {…}
      public Sting getSex() {…}

}

public class Queue
{
      public Queue()  {…}
      public void enqueue(Object elem) {…}
      public Object dequeue()  {…}
      public boolean isEmpty() {…}

      //definition for other methods
}

public class Stack
{
      public Stack()  {…}
      pulic void push(Object elem)  {…}
      public Object pop()  {…}
      public boolean isEmpty()  {…}

      //definition for other methods
}
```

Write a Java application to solve the following problems:

f) Create a Queue object named as `qChildren`.

(1 marks)

g) Create two Stack objects named as `sMale` and `sFemale`.

(2 marks)

h) Input twenty (20) children into `qChildren`.

(3 marks)

i) Calculate total payment received by the nursery if the payment based on the age. The payment table as follows:

Age	Payment
Less than 1	RM 200
1 or less than 3	RM 180
3 to 4	RM 170
5 to 6	RM 150

(6 marks)

j) Get all children from `qChildren`. Store male children into a stack called `sMale` and female children into a stack called `sFemale`.

(5 marks)

Answer Scheme (Suggestion):

```
public class AppQSChildNursery
{
      public static void main(String args[])
      {
      a)    Queue qChildren = new Queue();          [1 marks]
                                                              (1 marks)

      b)    Stack sMale = new Stack();              [1 marks]
            Stack sFemale = new Stack();            [1 marks]
                                                              (2 marks)
      c)    Scanner scanner = new Scanner (System.in);
            String lineSeparator = System.getProperty
                                   ("line.separator");
            scanner.useDelimiter (lineSeparator);

            for(int i=0; i<3; i++)
            {
                  System.out.print("Enter name: ");
                  String name = scanner.next();
                  System.out.print("Enter race: ");
                  String race = scanner.next();
                  System.out.print("Enter age: ");
                  int age = scanner.nextInt();
                  System.out.print("Enter sex: ");
                  String sex = scanner.next();          [1 ½ marks]
                  ChildNursery CN= new ChildNursery();  [1/2 marks]
                  CN.setData(name,race, age, sex);      [1/2 marks]
                  qChildren.enqueue(CN);                [1/2 marks]
            }
                                                              (3 marks)


      d)    double totPay = 0.0;                        [1/2 marks]
            Queue qTemp = new Queue();                  [1/2 marks]
            while(!qChildren.isEmpty())
            {
                  Object obj = qChildren.dequeue();
                  ChildNursery CN = (ChildNursery)obj;  [1 marks]
                  qTemp.enqueue(CN);                    [1/2 marks]
```

```
                if(CN.getAge() < 1)
                      totPay = totPay + 200;
                else if (CN.getAge ()<3)
                      totPay = totPay + 180;
                else if (CN.getAge ()<4)
                      totPay = totPay + 170;
                else if (CN.getAge () <6)
                      totPay = totPay + 150;          [2 marks]
          }

          System.out.println("Total payment: "+totPay);[1/2 marks]

          while(!qTemp.isEmpty())
          {
                Object obj = qTemp.dequeue();
                qChildren.enqueue(obj);               [1 marks]
          }
```

(6 marks)

```
    e)    while(!qChildren.isEmpty())                [1/2 marks]
          {
                Object obj = qChildren.dequeue();
                ChildNursery CN = (ChildNursery)obj;  [1 marks]

                if(CN.getSex().equalsIgnoreCase("Male"))
                      sMale.push(CN);                 [1 ½  marks]
                else
                      sFemale.push(CN);               [1 marks]
          }
```

(5 marks)

```
          System.exit(0);
    }
}
```

Recursion

CHAPTER 5

5.1 INTRODUCTION

- You can use recursion to solve many kinds of programming problems that would be very difficult to conceptualize and solve without recursion.
- Computer scientists in the field of artificial intelligence (AI) often use recursion to write programs that exhibit intelligent behavior: playing games such as chess, proving mathematics theorems, recognizing patterns, and so on.
- Recursive algorithms and methods can be used to perform common mathematical operations such as computing a factorial or a greatest common divisor.

> **Recursion**
> The process of solving a problem by reducing it to smaller versions of itself.

5.2 RECURSIVE THINKING

- Recursion is a problem solving approach that can be used to generate simple solutions to a certain kinds of problems that would be difficult to solve in other ways.
- In a recursive algorithm the original problem is split into one or more simpler version of itself.
- So, we can say that any problems which can be split into one or more simpler version of itself can be considered to use recursive approach. Unfortunately, you also have to consider other factors such as the complexities of the problem given.

- **Example:**

  The factorial of an integer is defined as follows:

```
0!= 1,
n! =n ×(n -1)!, if  n >0

     Let n = 3;
     3! = 3 x 2!
     2! = 2 x 1!
     1! = 1 x 0!
     0! = 1
     The answer: 6
```

- The following shows the general approach for a recursive algorithm:

```
if the problem can be solved for the current value of n
     Solve it.
else
     Recursively apply the algorithm to one or more
     problems involving smaller values of n
     Combine the solutions to the smaller problems to get
     the solution to the original.
```

5.3 RECURSIVE CHARACTERICTICS

- The characteristics of a recursive solution:
  1. There must be at least one case (the base case), for a small value of *n*, that can be solved directly.
  2. A problem of a given (say, *n*) can be split into one or more smaller versions of the same problem (the recursive case).

- Therefore, to design a recursive algorithm, we must:
  1. Recognize the base case and provide a solution it.
  2. Devise a strategy to split the problem into smaller version of itself. Each recursive case must make progress toward the base case.
  3. Combine the solutions to the smaller problems in such a way that each larger problem is solved correctly.

5.4 RECURSIVE METHOD

- **Recursive definition:** A definition in which something is defined in terms of a smaller version of itself.
- **Example:**

$$f(n) = \begin{cases} 0, if\ n = 0 \\ n\ x\ (n-1), if\ n > 0 \end{cases}$$

Recursive definition

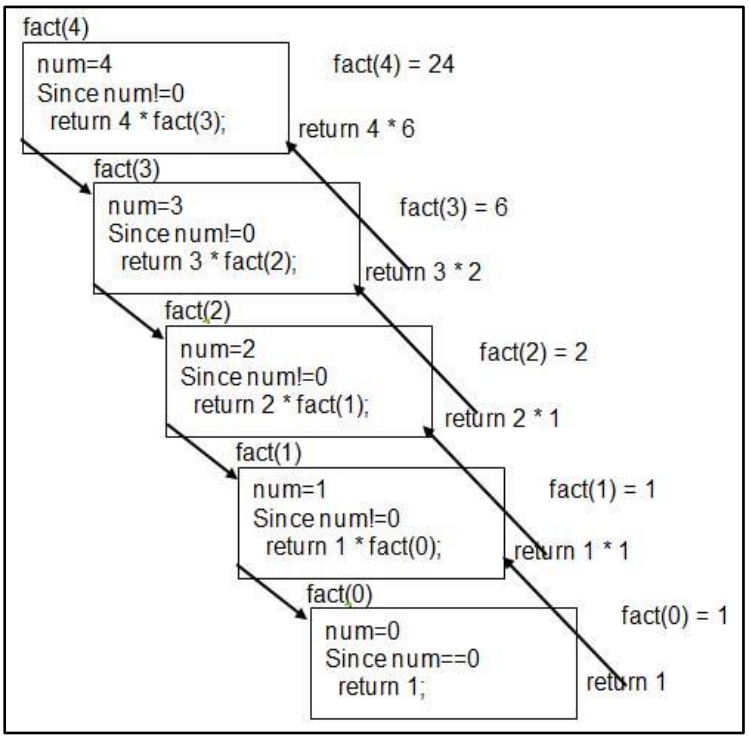
- A method that calls itself is called **Recursive Method.**

- Example of recursive method to implement the factorial definition:

```
public static int fact(int num)
{
      if (num == 0)
            return 1;
      else
            return num*fact(num-1);
}
```

➢ The following figure traces the execution of the following statement

```
System.out.println(fact(4));
```



The execution of recursive method

NOTE:

- The downward arrows represent the successive calls to the method fact.
- The upward arrows represent the values returned to the caller.

5.5 INFINITE RECURSION

- If every recursive call results in another recursive call (non-stop), then the recursive method (algorithm) is said to have **infinite recursion**. Its can happen when the recursive call didn't find the base case.
    - Infinite recursion executes forever.

- So, you must be carefull when design a recursive method by make sure your solution must have at least one base case and general cases (recursive call) will meet this base case during the execution process. Otherwise, it will cause the infinite loop.

- To design a recursive method, you must:
    1. Understand the problem requirements.
    2. Determine the limiting conditions. For example, for a list, the limiting condition is determined by the number of elements in the list.
    3. Identify the base cases and provide a direct solution to each base case.
    4. Identify the general cases and provide a solution to each general case in terms of a smaller version of itself.

5.6 EXAMPLE OF RECURSIVE ALGORITHM

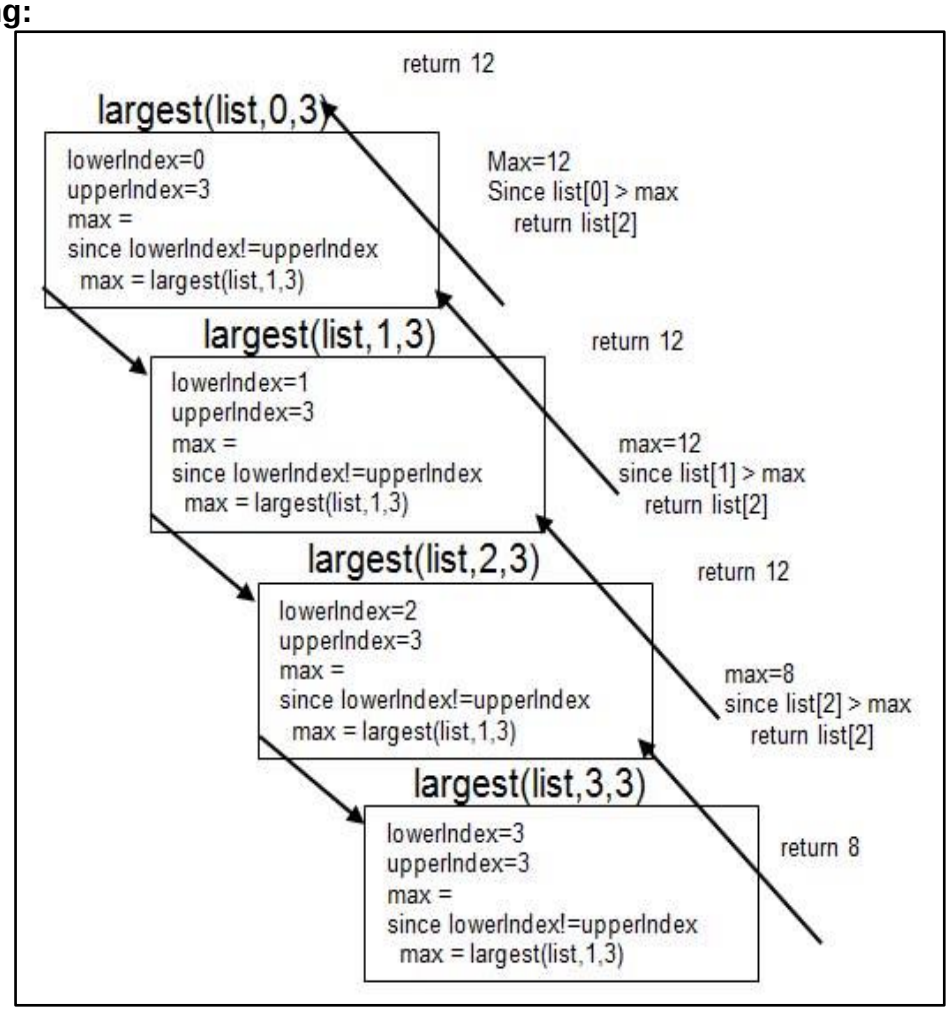
EXAMPLE OF PROGRAM 1

To find the largest number.

```
//Recursion: largest Element in an Array
import java.io.*;
public class largestElementInArray
{
      public static void main(String args[])
      {
         int intArray[] ={5,10,12,8};
         System.out.println("The largest element in intArray:"
                    +largest(intArray,0,intArray.length-1));
}
```

```
public static int largest(int list[], int lowerIndex,
                          int upperIndex)
{
      int max;

      if(lowerIndex == upperIndex)
            return list[lowerIndex];




      else
      {
            max=largest(list,lowerIndex+1,upperIndex);
            if(list[lowerIndex] >= max)
                  return list[lowerIndex];
            else
                  return max;
      }
}
```

Tracing:



The execution of recursive method

EXAMPLE OF PROGRAM 2

To calculate the Fibonacci number.

$$rFibNum(a,b,n) = \begin{cases} a, & if\ n = 1 \\ b, & if\ n = 2 \\ rFinNum(b, n-1) + rFibNum(a,b,n-2), & if\ n > 2 \end{cases}$$

Recursive method for Fibonacci number

```
//Recursion: Fibonacci Number
import java.io.*;
public class FibonacciNumber
{
    static BufferedReader keyboard = new
             BufferedReader(new InputStreamReader(System.in));

    public static void main(String[] args) throws IOException
    {
       int firstFibNum;
       int secondFibNum;
       int nth;

       System.out.print("Enter the first Fibonacci number: ");
       firstFibNum = Integer.parseInt(keyboard.readLine());
       System.out.println();

       System.out.print("Enter the second Fibonacci number:");
       secondFibNum = Integer.parseInt(keyboard.readLine());
       System.out.println();

       System.out.print("Enterthe desired Fibonacci number:");
       nth = Integer.parseInt(keyboard.readLine());
       System.out.println();

       System.out.println("The Fibonacci number at position "
                   + nth + " is: "
                   + rFibNum(firstFibNum, secondFibNum, nth));
    }

    public static int rFibNum(int a, int b, int n)
    {
       if(n == 1)
         return a;
       else if(n == 2)
         return b;
       else
         return rFibNum(a, b, n - 1) + rFibNum(a, b, n - 2);
    }
}
```

Tracing:

Let; rFibNum(2,5,4)

rFibNum(2,5,4) = rFibNum(2,5,3) + rFibNum(2,5,2)

Let's first:

1.a *rFibNum(2,5,3) = rFibNum(2,5,2) + rFibNum(2,5,1)*

1.a.1 *rFibNum(2,5,2) = 5*

1.a.2 *rFibNUm(2,5,1) = 2*

We substitute the values of rFibNum(2,5,2) and rFibNum(2,5,1) into (1.a) to get:

rFibNum(2,5,3) = 5 + 2 = 7

Next we determine rFibNum(2,5,2);

rFibNum(2,5,2) = 5

So;

rFibNum(2,5,4) = 7 + 5 = 12

Calculating the Fibonnaci number

5.7 RECURSION VERSUS ITERATION.

- You may have noticed that there are some similarities between recursion and iteration.
- Both techniques enable us to repeat a compound statement.
- In an iteration, a loop repetition condition in the loop header determines whether we repeat the loop body or exit from the loop. We repeat the loop body while the repetition condition is true.
- In recursion, the condition usually test for a base case. We stop the recursion when the base case is reached (the condition is true), and we execute the method body again when the condition is false.
- We can always write an iterative solution to a problem that solvable by recursion.
- However, the recursive algorithm may be easier to conceptualize and may, therefore, lead to a method that is easier to write, read, and debug.
- There are usually two ways to solve a particular problem – iteration and recursion.

- The obvious question is, which method is better- iteration or recursion?
  - There is no simple answer.
  - In addition to the nature of a problem, the other key factor in determining the best solution method is **efficient**.

- As a general rule:
  1. If you think an iterative solution is more obvious and easier to understand that a recursive solution, use the iterative solution, which is more efficient.
  2. The power of recursion, if the definition of a problem is inherently recursive, then you should consider a recursive solution.

PAST YEAR EXAM QUESTION – Sample

Given the following recursive program:

```
import java.util.*;

public class Recursive
{
      public static void main(String args[])
      {
            Scanner scanner = new Scanner(System.in);
            System.out.print("Enter level: ");
            int lvl = scanner.nextInt();
            printStuff(lvl);
      }

      static void printStuff(int level)
      {
       if (level == 0)
          System.out.print("*");

      else
      {
         System.out.print("[");
         printStuff(level - 1);
         System.out.print(",");
         printStuff(level - 1);
         System.out.print("]");
       }
  }
}
```

a) What is the output for the above program if the input is `printStuff(3)`?

(4 marks)

b) Write a recursive method `myPower` that computes X^n.

Example:

$2^0 = 1$
$4^3 = 64$
$3^5 = 243$

(6 marks)

Suggestion Answer Scheme:

a) [[[*,*] , [*,*]], [[*,*] , [*,*]]]

(4 marks)

b)

```
static int myPower(int X, int n)          [1 marks]
{
     if (n == 0)                          [2 marks]
        return 1;

     else
        return X * myPower(X, n-1);       [3 marks]
}
```

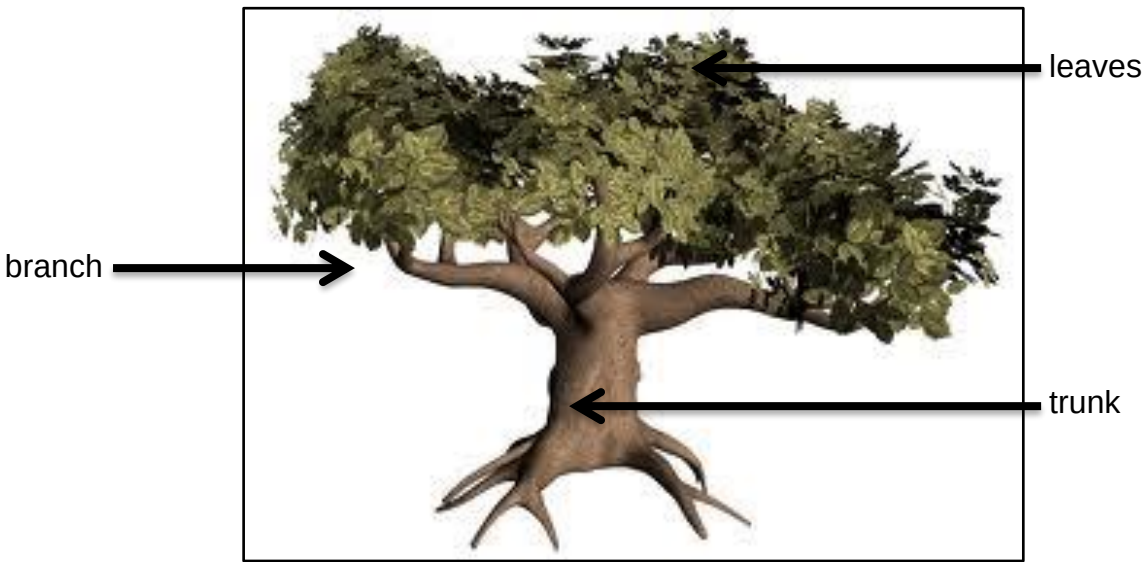
(6 marks)

Binary Tree

CHAPTER 6

6.1 INTRODUCTION

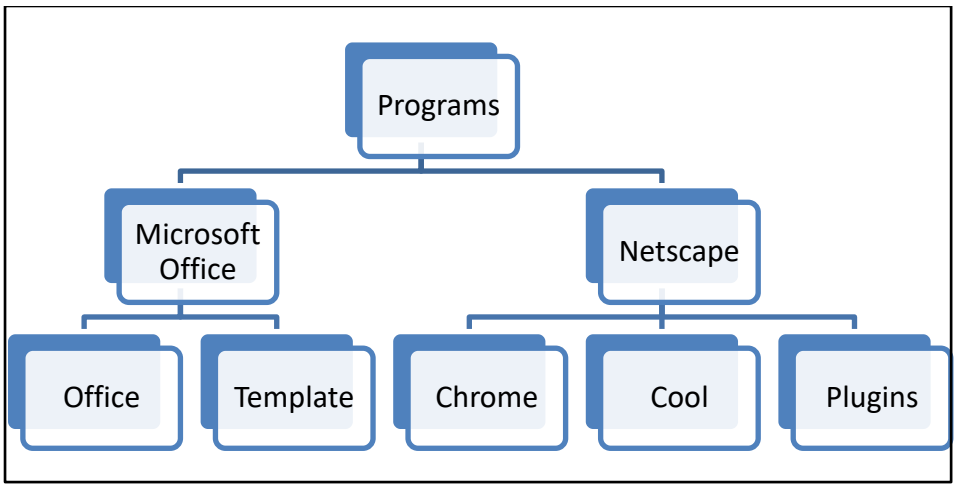
- The data organizations you studied so far are linear in that each element has only one predecessor or successor. For example, linked lists, stacks and queues are linear data structure.
- Accessing all the elements in the sequence is an $O(n)$ process.
- A tree is a non-linear or hierarchical. Instead of having just one successor, a node in a tree can have multiple successors; but it has just one predecessor.
- A tree in computer science is like a natural tree, which has a single trunk that may split off into two or more main branches.
- The predecessor of each main branch is the trunk.
- Each main branch may spawn several secondary branches (successors of the main branches).
- The predecessor of each secondary branch is a main branch.



A tree

- In computer science, we draw a tree from the top down, so the root is at the top of the diagram instead of the bottom.

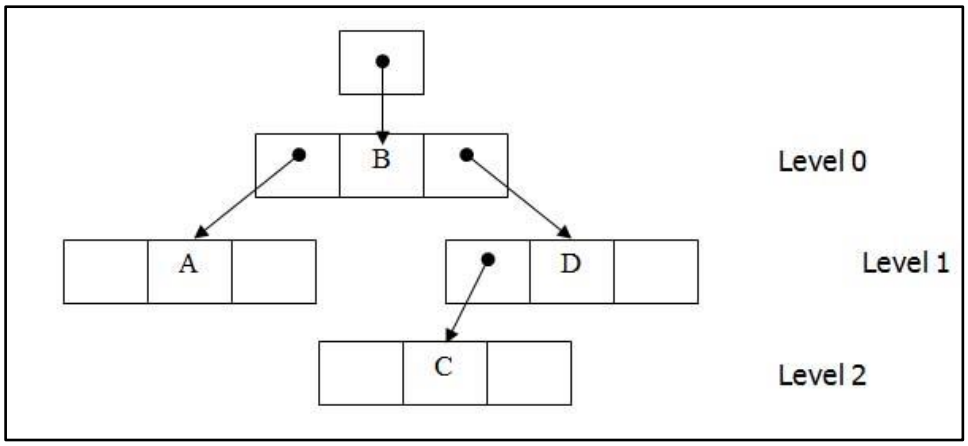
- Because trees have a hierarchical structure, we use them to represent hierarchical organizations of information, such as class hierarchy.
- Example:



A class hierarchy

6.2 TREE TERMINOLOGY

- We use the same terminology to describe trees in computer science as we do trees in nature.
- A computer science tree consists of a collection of elements or nodes, with each node linked to its successors.
- The node at the top of a tree is called its root because computer science trees grow from the top-down. The links from a node to its successors are called branches. The successors of a node are called its children. The predecessor of a node is called its parent. Each node in a tree has exactly one parent except for the root node, which has no parent. Nodes that have the same parent are siblings. A node that has no children is a leaf node. Leaf nodes are also known as external nodes, and non leaf nodes are known as internal nodes.
- Tree nodes contain two or more links.



Graphical tree representation

- The following table simplifies the terminology use to describe trees:

Terminology	Description
root	The first node in a tree.
child	Each node link to the root node.
left child	The first node in the **left subtree** (also known as the root node of the left subtree).
right child	The first node in the **right subtree** (also known as the root node of the right subtree).
siblings	The children of a specific node.
branch	The line from a root element to a subtree.
leaf	An element whose associated left and right subtrees are both empty.
height	The number of branches between the root and the farthest leaf.
depth	The level of the root element is 0, and the height of a tree is equal to the highest level in the tree. An element's level is also referred to as that element's depth.

Terminology use to describe trees

- **Example:**

Root (parent) → Node B

Child → Node A and D

Siblings → Node A and D

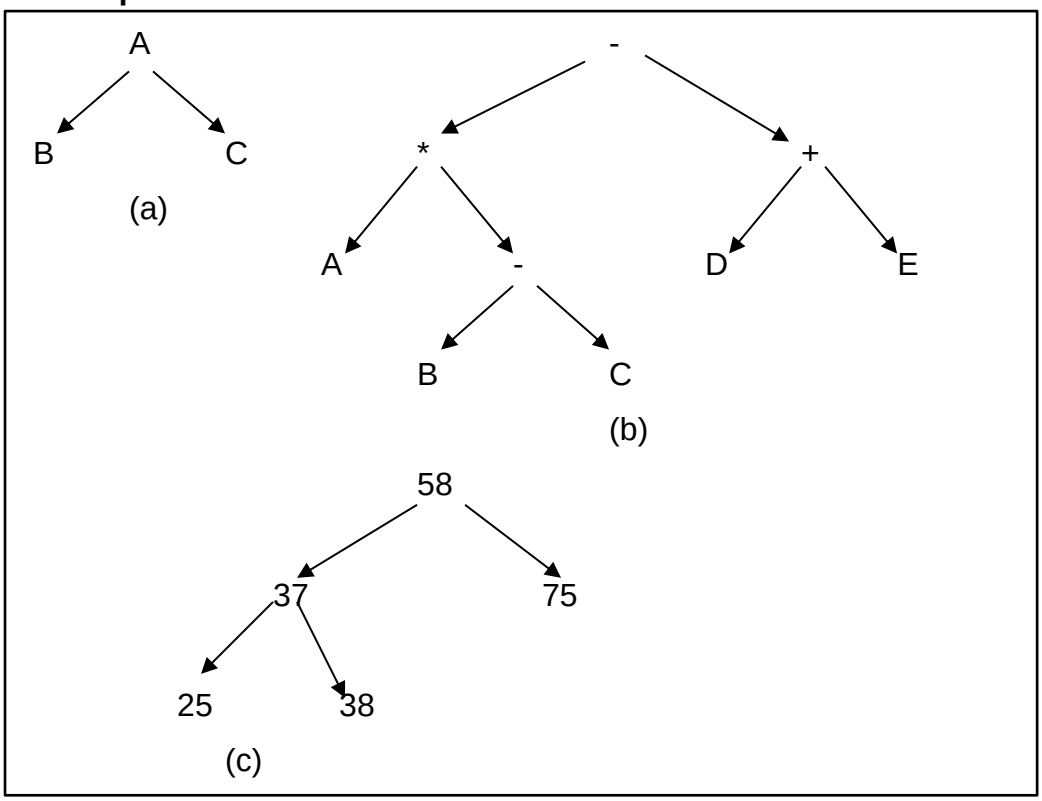
Leaf → Node C and A

Height → 2

Depth → 2

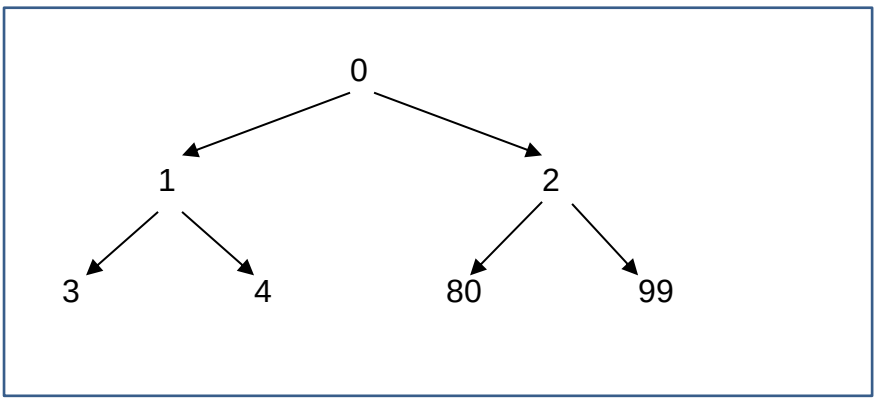
6.3 BINARY TREE

- **A binary tree (*t*)** is either empty or consists of an element, called the root element, and two distinct binary trees, called the *left subtree* and *right subtree* of *t.*
- **Example:**



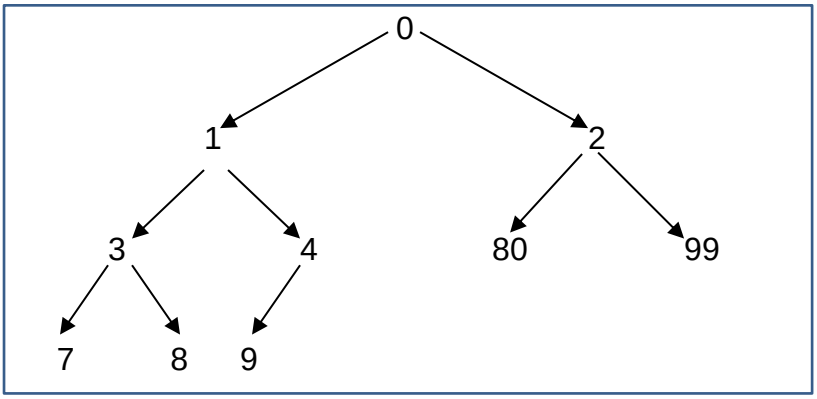
Binary tree

- A binary tree (*t)* is **full** if *t* is a two-tree with all its leaves on the same level. This is called Strictly/Full Binary Tree.
- **Example:**



Full binary tree

- A binary tree (*t)* is **complete** if *t* is full through the next-to-lowest level and all the leaves at the lowest level are as far to the left as possible. This is called Complete Binary Tree
- **Example:**



Complete binary tree

6.4 TAVERSAL OF A BINARY TREE

- A traversal of a binary tree (*t)* is an algorithm that accesses each element in *t* exactly once.
- We identify three different kinds of traversals.

Traversal 1: *inOrder traversal [Left-Node-Right]*

- Here is the algorithm:
- Assume that *t* is a binary tree

```
inOrder(t) {
     if(t is not empty){
          inOrder(leftTree(t));
          Access the root element of t;
          inOrder(rightTree(t));
      }
}
```

Method `inOrder`

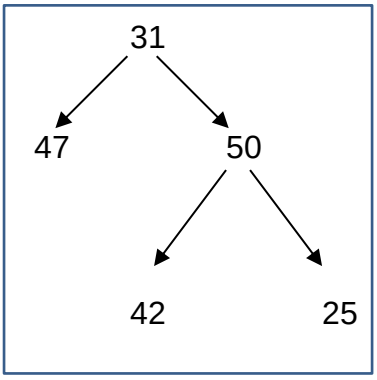
- Traverse the left subtree with a call to `inOrder`.
- Process the value in the node.
- Traverse the right subtree with a call to `inOrder`.

So, the sequence is:

L : Left
N : Node
R : Right

} LNR (InOrder traversal)

- The inOrder traversal does not process the value of a node until the values in that node's left subtree are processed.
- **Example:**



Output: 47 31 42 50 25

Traversal 2: *postOrder traversal [Left-Right-Node]*

- The algorithm, with *t* a binary tree is:

```
postOrder(t) {
      if(t is not empty){
            postOrder(leftTree(t));
            postOrder(rightTree(t));
            Access the root element of t;
      }
}
```

Method `postOrder`

- Traverse the left subtree with a call to `postOrder`.
- Traverse the right subtree with a call to `postOrder`.
- Process the value in the node.

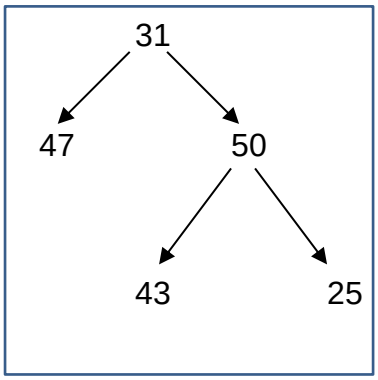
So, the sequence is:

L : Left
R : Right
N : Node

} LRN (post Order traversal)

- The postOrder traversal processes the value in each node after the values of all that node's children are processed.

- **Example:**



Output: 47 43 25 50 31

Traversal 3: *preOrder traversal [Node-Left-Right]*

- The algorithm, with *t* a binary tree is:

```
preOrder(t) {
      if(t is not empty){
            Access the root element of t;
            preOrder(leftTree(t));
            preOrder(rightTree(t));
      }
}
```

Method `preOrder`

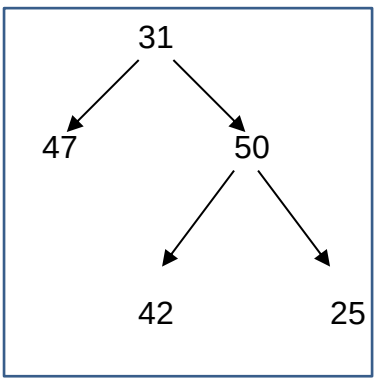
- Process the value in the node.
- Traverse the left subtree with a call to `preOrder`.
- Traverse the right subtree with a call to .

So, the sequence is:

N : Node
L : Left
R : Right

} NRL (pre Order traversal)

- The preOrder traversal processes the value in each node as the node is visited. After processing the value of a given node, the preorder traversal processes the values in the left tree, then the values in the right subtree.
- **Example:**



Output: 31 47 50 42 25

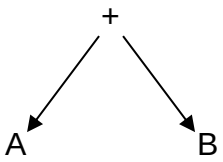
6.5 EXPRESSION TREE

- Used to represent arithmetic expression.
- Each non-leaf is a binary operator whose operands is the associated left and right subtrees. Operand are stored in leaf nodes.
- Parentheses are not stored in the tree, because the tree structure dictates the order of operator evaluation.
- Operator in nodes at higher levels are evaluated after operators in nodes at lower levels.
- If a node contains a binary operator, its left subtree represents the operator's left operand and its right subtree represents the operator's right operand.
- Consider the basic arithmetic expression as follows:

Operand1	operator	operand2
A	+	B

  - Expression tree representation as follows:

Operator	→	root
Operand1	→	left subtree
Operand2	→	right subtree



- **Example:**

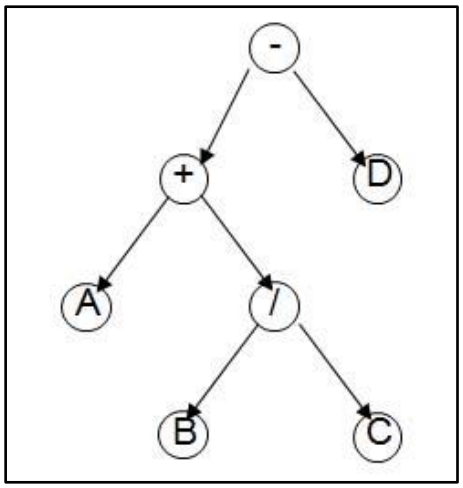
  (A + (B/C)) – D

  Operand1 = (A + (B/C))

  Operator = -

  Operand2 = D

  - By using the same method, operand1 can be simplified to make it into simpler form (basic arithmetic expression).



Arithmetic expression tree

- Based on the expression tree we can get prefix, infix and postfix notation by traverse the trees using pre order, in order and post order traversal.
    - In Order traversal, we can get infix notation:

      A + B / C - D

    - Pre Order traversal, we can get prefix notation:

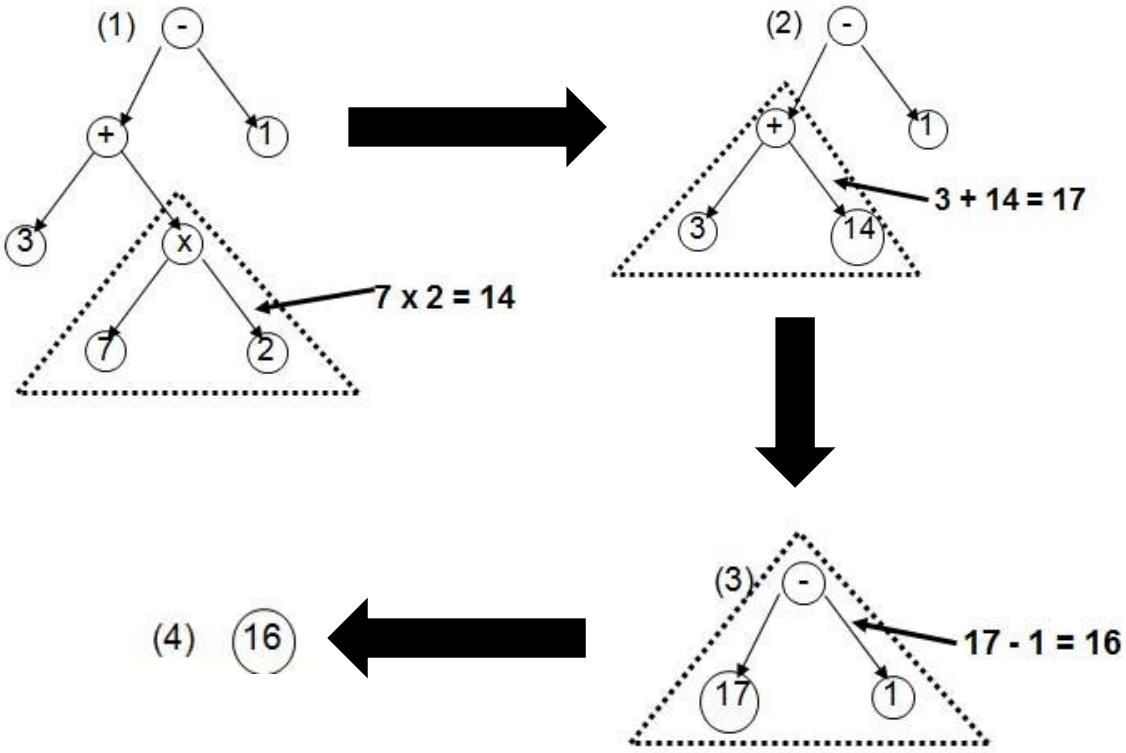
      - + A / B C D

    - Post Order traversal, we can get postfix notation:

      A B C / - D +

6.6 EVALUATE EXPRESION TREE BY USING IN ORDER TRAVERSAL

- We can evaluate the expression tree by using in order traversal, therefore we will get the infix notation.
- We also can use the following steps to evaluate the expression tree:
  1. Select the highest-level leaves subtree, perform the operation by using the operator at the root node and replace the value (result) on that root node.
  2. Repeat the process, until a single node that contains the value of expression.
- **Example:**

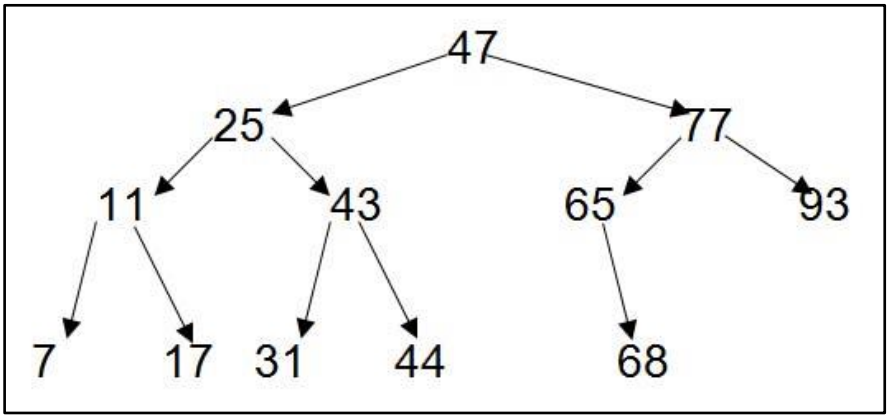


So, **the value of expression** = 16

In order traversal = (3 + (7 * 2)) – 1 = 16

6.7 BINARY SEARCH TREE

- A binary search tree (with no duplicate value) has the characteristic that **the values in any left subtree are less than the value in that subtree's parent node,** and **the values in any right subtree are greater than the value in that subtree's parent node**.
- **Example:**



Binary Search Tree

- The binary search tree facilitates duplicate elimination.
  - While building a tree, the insertion operation recognizes attempts to insert a duplicate value, because a duplicate follows the same "go left" or "go right" decisions on each comparison as the original value did.

- Searching a binary tree for a value that matches a key value is fast, especially for tightly packed (or balanced) trees.
- Based on the binary search tree we can traverse the trees using pre order, in order and post order traversal.
- When you traverse the binary search tree using in order traversal you will get the results in sequence (ascending or descending order).
- Based on the above figure you will get the following result:

In Order traversal: 7 11 17 25 31 43 44 47 65 68 77 93 (ascending order)

Pre Order traversal: 47 25 11 7 17 43 31 44 77 65 68 93

Post Order traversal: 7 17 11 31 44 43 25 68 65 93 77 47

EXAMPLE OF PROGRAM – PRIMITIVE DATA TYPE

```
/* Author : Mazidah Puteh
   Objective:
      Writing Program for Tree Application
      - print inorder, preorder, postorder
      - count number of nodes ( count without condition)
      - calculate sum of data in nodes (calc without
        condition)
      - count number of value > 50 ( count with condition)
      - calculate sum of all data minus discount ( calculate
        with new value)


// Class TreeNode definition
class TreeNode {
   // package access members
   TreeNode left;   // left node
   int data;        // data item
   TreeNode right;  // right node

   // Constructor: initialize data to d and make this a leaf
   //node
   public TreeNode( int d )
   {
      data = d;
      left = right = null;  // this node has no children
   }

   // Insert a TreeNode into a Tree that contains nodes.
   // Ignore duplicate values.
   public void insert( int d )
   {
      if ( d < data ) {
         if ( left == null )
            left = new TreeNode( d );
         else
            left.insert( d );
      }
      else if ( d > data ) {
         if ( right == null )
            right = new TreeNode( d );
         else
            right.insert( d );
      }
   }
}

// Class Tree definition
class BSTree {
   private TreeNode root;

   // Construct an empty Tree of integers
   public BSTree() { root = null; }
```

```
   // Insert a new node in the binary search tree. If the
   // root node is null, create the root node here.
   // Otherwise, call the insert method of class TreeNode.
   public void insertNode( int d )
   {
      if ( root == null )
         root = new TreeNode( d );
      else
         root.insert( d );
   }

   // Preorder Traversal
   public void preorderTraversal()
   { preorderHelper( root ); }
   // Recursive method to perform preorder traversal
   private void preorderHelper( TreeNode node )
   {
      if ( node == null )
         return;

      System.out.print( node.data + " " );
      preorderHelper( node.left );
      preorderHelper( node.right );
   }

   // Inorder Traversal
   public void inorderTraversal()
   { inorderHelper( root ); }

   // Recursive method to perform inorder traversal
   private void inorderHelper( TreeNode node )
   {
      if ( node == null )
         return;

      inorderHelper( node.left );
      System.out.print( node.data + " " );
      inorderHelper( node.right );
   }

   // Postorder Traversal
   public void postorderTraversal()
   { postorderHelper( root ); }

   // Recursive method to perform postorder traversal
   private void postorderHelper( TreeNode node )
   {
      if ( node == null )
         return;

      postorderHelper( node.left );
      postorderHelper( node.right );
      System.out.print( node.data + " " );
   }
```

```
public void calcSize()
{
  System.out.println("Size of tree is: "
                     +calcSizeAll( root ));
}

// Recursive method to perform counting without condition
private int calcSizeAll( TreeNode node )
{
   if ( node == null )
      return 0;
   return 1 + calcSizeAll(node.left)
          + calcSizeAll(node.right);
}
public int calcSum()
{ return calcSumAll( root ); }

//Recursive method to perform calculations without
//condition
private int calcSumAll( TreeNode node )
{
   if ( node == null )
      return 0;

   return node.data + calcSumAll(node.left)
          +calcSumAll( node.right );
}

public int countVal()
{ return countValAll( root ); }

// Recursive method to perform counting with condition
private int countValAll( TreeNode node )
{
   if ( node == null )
      return 0;

   if (node.data > 50)
        return 1 + countValAll( node.left )
                 + countValAll( node.right );
   else
        return countValAll (node.left)
                 + countValAll(node.right);
 }

 public double calcNewVal()
 { return calcNewAll( root ); }
```

```
    // Recursive method to perform calculations with
    // condition
    private double calcNewAll( TreeNode node )
    {
      if ( node == null )
         return 0;

      double comm = 0.80 * node.data;
      return comm + calcNewAll( node.left )
             + calcNewAll( node.right );
    }

    public void printcat()
    {
       System.out.println("Number that multiple of 2");
       int num1 = countcategory(root,2);
       System.out.println (num1);
       System.out.println("Number that multiple of 5");
       int num2 = countcategory(root,5);
       System.out.println (num2);
       System.out.println("Number that multiple of 10");
      int num3 = countcategory(root,10);
      System.out.println (num3);
    }

    // Recursive method to perform calculations with
    // condition
    private int countcategory( TreeNode node, int num )
    {
       if ( node == null )
         return 0;

       if (node.data % num == 0)
           return 1+countcategory( node.left,num )+
                           countcategory( node.right,num );
       else
           return countcategory (node.left,num)+
                           countcategory(node.right,num);
    }
}
```

```
// Java application program of Binary Search Tree
public class BSTTest {
   public static void main( String args[] )
   {
      BSTree tree = new BSTree();
      int intVal;

      System.out.println("Inserting the following values:");

      for ( int i = 1; i <= 10; i++ ) {
         intVal = ( int ) ( Math.random() * 100 );
         System.out.print( intVal + " " );
         tree.insertNode( intVal );
      }

      System.out.println ("\n Calculation Size Of Tree");
      tree.calcSize();

      System.out.println("\nNumber of values with
                        different category");
      tree.printcat();

      System.out.println ("\nSum Of Tree "+tree.calcSum());

      System.out.println ("\nNumber of value > 50 is "
                          +tree.countVal());

      System.out.println ("\nSum of value minus 20% discount
                          is" +tree.calcNewVal());

      System.out.println ( "\n\nPreorder traversal" );
      Tree.preorderTraversal();

      System.out.println ( "\n\nInorder traversal" );
      tree.inorderTraversal();

      System.out.println ( "\n\nPostorder traversal" );
      tree.postorderTraversal();
      System.out.println();
   }
}
```

PAST YEAR EXAM QUESTION – Sample 1

Given the following `Hotel, TreeNode` and `BSTHotel` ADTs:

```
public class Hotel
{
      private String custName;
      private int roomNumber;
      private String type;
      private double rate;
      private int day;

      public Hotel(){…}

      public Hotel(String cn, int rn,String t,double r,int d)
      {…}
      public String getCustName() {…}
      public int getRoomNumber()  {…}
      public String getType()  {…}
      public double getRate()  {…}
      public int getDay()  {…}
}

public class TreeNode
{
      TreeNode left;
      Hotel elem;
      TreeNode right;

      // definition for other methods
}

public class BSTHotel
{
      TreeNode root;

      public BSTHotel(){…}

      public void countType(){…}

      public double calcPayment() {…}

      public Object findInformation(int roomNumber){…}

      public void displayAll(){…}

      //definition for other methods

}
```

Based on the information in the following table:

cusName	roomNumber	Type	Rate (RM)	day
David Low	0201	Family	350.00	2
Manaf Abu Bakar	0110	Standard	128.00	5
Himilton Daves	0108	Deluxe	247.50	3
Kumar Maniam	0210	Standard	128.00	4
Hamidah Roslan	0209	Standard	128.00	5
Jamal Jamaldin	0190	Deluxe	247.50	4
Tan Boon	0211	Family	350.00	3
Mary Brown	0220	Superior	210.00	3

a) Draw a binary Search Tree (BST) diagram according to the `roomNumber`.

(4 marks)

b) Write the definition of method `countType` to count and display the number of each type of hotel reservation (Family, Standard, Deluxe and Superior).

(4 marks)

c) Write the definition of method `calcPayment` to calculate the payment that has to be paid by each customer.

(4 marks)

d) Write the definition of method `findInformation` to find and return the information of any hotel reservation based on the room number.

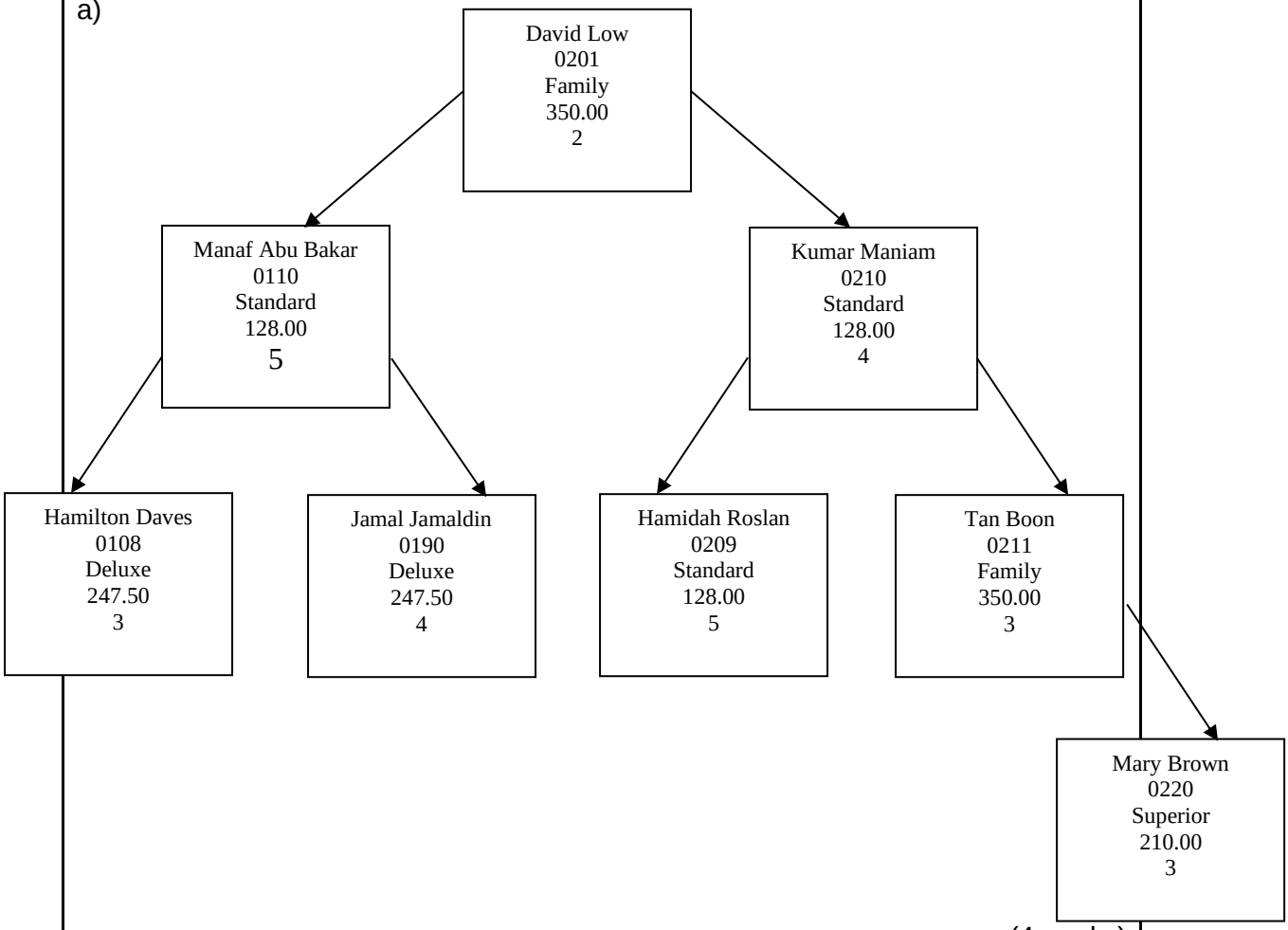
(4 marks)

e) Write the definition of method `displayAll` to display the information stored in the binary search tree in ascending order.

(4 marks)

Answer Scheme (Suggestion):

a)



(4 marks)

b)

```
public void countType()
{
  System.out.println("The number of room (Family): "
                  +cntHotel(root, "Family"));
  System.out.println("The number of room (Standard): "
                  +cntHotel(root, "Standard"));
  System.out.println("The number of room (Deluxe): "
                  +cntHotel(root, "Deluxe"));
  System.out.println("The number of room (Superior): "
                  +cntHotel(root, "Superior"));
}

public int cntHotel(TreeNode node, String typ)
{
  if(node == null)
      return 0;

  if(node.elem.getType().equalsIgnoreCase(typ))
      return 1 + cntHotel(node.left, typ) +
                 cntHotel(node.right, typ);
  else
      return cntHotel(node.left, typ) +
             cntHotel(node.right, typ);
```

1 marks

½ marks

½ marks

1 marks

1 marks

```
}
```

c)

```
public double calcPayment()
{
      return calcPay(root);
}

public double calcPay(TreeNode node)
{
  if (node == null)
      return 0.0;
  else
      return node.elem.getRate() * node.elem.getDay() +
             calcPay(node.left) + calcPay(node.right);
}
```

1 marks

½ marks

½ marks

2 marks

d)

```
public Object findInformation(int rn)
{
      return findInfo(root, rn);
}

public Object findInfo(TreeNode node, int rn)
{
      Object ob = null;

      if (node == null)
            return null;

      if(node.elem.getRoomNumber()==rn)
      {
            ob = node.elem;
      }
      findInfo(node.left, rn);
      findInfo(node.right, rn);
      return ob;
}
```

½ marks

½ marks

½ marks

½ marks

1 marks

1 marks

e)

```
public void displayAll()
{
      display(root);
}

public void display(TreeNode node)
{
      if (node == null)
            return;
      else{
            display(node.left);
            System.out.println("Customer Name: "
                  +node.elem.getCusName());
            System.out.println("Room Number: "
                  +node.elem.getRoomNumber());
            System.out.println("Type: "+node.elem.getType());

            System.out.println("Rate: RM "
                  +node.elem.getRate());
            System.out.println("Day: "+node.elem.getDay());
            display(node.right);
      }
}
```

1 marks

½ marks

½ marks

½ marks

1 marks

½ marks

(20 marks)

PAST YEAR EXAM QUESTION – Sample 2

Given the following infix expression:

(1+S) * (1 + (M/2*23)) * 2 + (E – 127)

i. Draw the expression tree for the above expression.

(5 marks)

ii. What is the output if the preorder traversal is implemented using the above expression tree?.

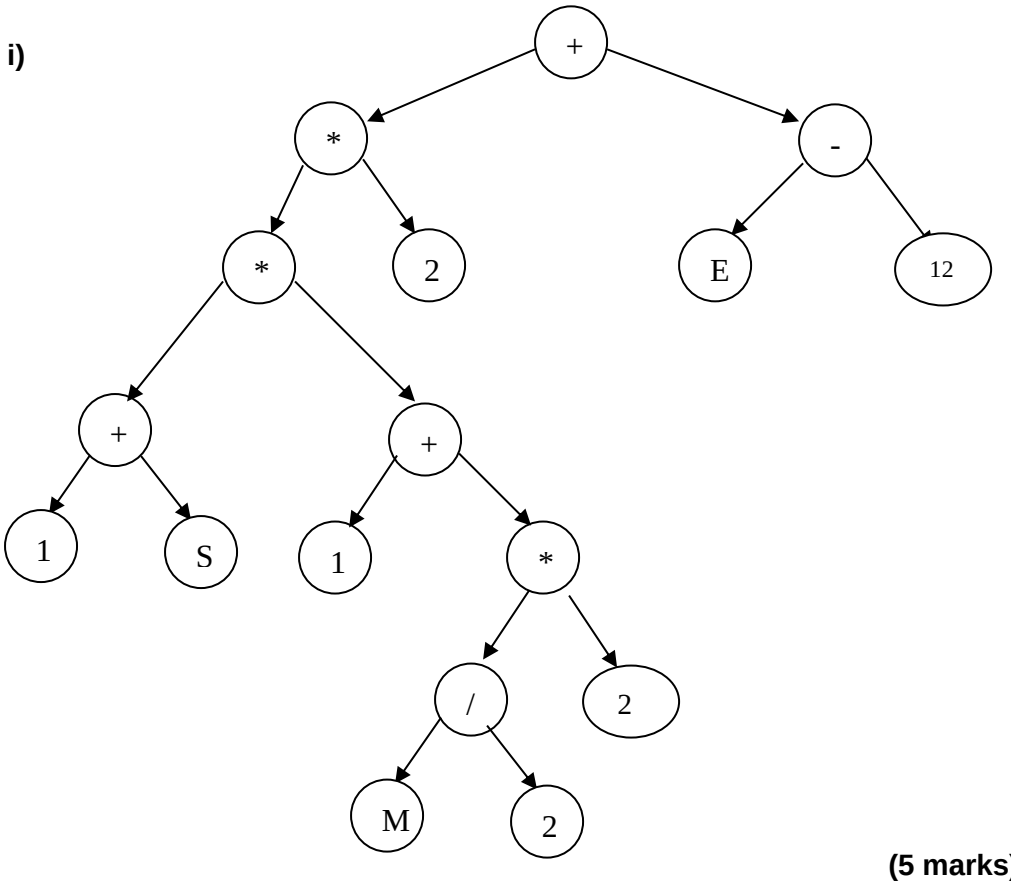
(3 marks)

iii. What is the depth of the above tree?

(1 marks)

Answer Scheme (Suggestion):

i)



(5 marks)

ii)

+ * * + 1 S + / * / M 2 23 2 - E 127

(3 marks)

iii)

6

(1 marks)