Higher Nationals in Computing

Unit 19: Data Structures & Algorithms

ASSIGNMENT 1

Learner's name: TRINH THI DIEU HUYEN

Assessor name: **HO HAI VAN**

Class: GCS0801B.1

ID: GDD18606

Subject code: 1649

Assignment due: Assignment submitted:

ASSIGNMENT 1 BRIEF

Qualification	**BTEC Level 5 HND Diploma in Business**		
Unit number	Unit 19: Data Structures and Algorithms		
Assignment title	Examine and specify ADT and DSA		
Academic Year			
Unit Tutor	Ho Hai Van		
Issue date		**Submission date**	
IV name and date			

Submission Format:
Format: The submission is in the form of an individual written report and a presentation. This should be written in a concise, formal business style using single spacing and font size 12. You are required to make use of headings, paragraphs and subsections as appropriate, and all work must be supported with research and referenced using the Harvard referencing system. Please also provide a bibliography using the Harvard referencing system. *Submission:* Students are compulsory to submit the assignment in due date and in a way requested by the Tutors. The form of

submission will be a soft copy in PDF posted on corresponding course of http://cms.greenwich.edu.vn/

Note: The Assignment *must* be your own work, and not copied by or from another student or from books etc. If you use ideas, quotes or data (such as diagrams) from books, journals or other sources, you must reference your sources, using the Harvard style. Make sure that you know how to reference properly, and that understand the guidelines on plagiarism. *If you do not, you definitely get fail*

Assignment Brief and Guidance:

Scenario: You work as in-house software developer for Soft.net Development Ltd, a software body-shop providing network provisioning solutions. Your company is part of a collaborative service provisioning development project and your company has won the contract to design and develop a middleware solution that will interface at the front-end to multiple computer provisioning interfaces including SOAP, HTTP, JML and CLI, and the back-end telecom provisioning network via CLI.

Your account manager has assigned you a special role that is to inform your team about designing and implementing abstract data types. You have been asked to create a presentation for all collaborating partners on how ADTs can be utilised to improve software design, development and testing. Further, you have been asked to write an introductory report for distribution to all partners on how to specify abstract data types and algorithms in a formal notation.

Tasks

Part 1

You will need to prepare a presentation on how to create a design specification for data structures, explaining the valid operations that can be carried out on the structures using the example of:

1. A stack ADT, a concrete data structure for a First in First out (FIFO) queue.
2. Two sorting algorithms.

3. Two network shortest path algorithms.

Part 2

You will need to provide a formal written report that includes the following:

1. Explanation on how to specify an abstract data type using the example of software stack.
2. Explanation of the advantages of encapsulation and information hiding when using an ADT.
3. Discussion of imperative ADTs with regard to object orientation.

Learning Outcomes and Assessment Criteria		
Pass	**Merit**	**Distinction**
LO1 Examine abstract data types, concrete data structures and algorithms		**D1** Analyse the operation, using illustrations, of two network shortest path algorithms, providing an example of each.
P1 Create a design specification for data structures explaining the valid operations that can be carried out on the structures. **P2** Determine the operations of a memory stack and how it is used to implement function calls in a computer.	**M1** Illustrate, with an example, a concrete data structure for a First in First out (FIFO) queue. **M2** Compare the performance of two sorting algorithms.	
LO2 Specify abstract data types and algorithms in a formal notation		**D2** Discuss the view that imperative ADTs are a basis for object orientation and, with justification, state whether you agree.
P3 Using an imperative definition, specify the abstract data type for a software stack.	**M3** Examine the advantages of encapsulation and information hiding when using an ADT.	

Table of Contents

Table of Figures

Table of Tables

ASSIGNMENT 1 ANSWERS

LO1 Examine abstract data types, concrete data structures and algorithms

P1 Create a design specification for data structures explaining the valid operations that can be carried out on the structures.

1. Create a design specification for data structures.

1.1. What is an ADT ?

ADT - refers to an Abstract Data Type that represent some data and operations on that data. It consists of Declaration of data and Declaration of operations. ADT defines a particular data structure in terms of data and operations. Besides, it also offers an interface of the objects, that is instances of an ADT. The behavior of each operation is determined by its inputs and outputs. The implementation details are hidden from the user of the ADT and protected from outside accessed, it just can be manipulated only by means of operations - This is referred to as encapsulation of data.

1.2. How to implement the ADT ?

To implement the ADT needs to go from definition to implementation. The first, problem definition and identification of ADTs, mean, identify data or attributes. Consists, specify ADT interactions and specify ADT operations. Then, identify object hierarchy (if using OOP) and implement ADTs. The implementation consists of a concrete representation of the abstract data in terms of existing data types and off-the-shelf ADT operations the implementation may include program code to implements the allowable operations.

It's difficult to specify a single list Abstract Data Type (ADT) that covers both arrays and linked lists. One issue is the representation of Array position is represented by integer and Linked list position is represented by pointer. They are completely different. In case of an array with n elements, a "position" is simply an integer in the range from 0 to n-1. In linked list, a position can be a pointer to one of the nodes in the list, but there are some subtleties involved.

2. Explaining the valid operations that can be carried out on the structures.

2.1. What common operations can be carried out on the ADT ?

Table 1: Some common operations can be carried out on the ADT

Some common operations can be carried out on the ADT	
`getFirst()`	Returns the first element in the list
`getLast()`	Returns the last element in the list
`getNext(p)`	Returns the next element (here it is `p`) in the list
`getPrev(p)`	Returns the previous element (here it is `p`) in the list
`get(p)`	Returns the element at the specified position (here it is `p`) in the list
`set(p, x)`	Fixes the number at position `p` to `x`
`insert(p, x)`	Add to position `p` the number `x`
`remove(p)`	Removes the element at the specified position (here it is `p`) in the list
`removeFirst()`	Remove an item from the beginning of the list
`removeLast()`	Remove an item from the end of the list
`removeNext(p)`	Remove an item (here it is `p`) from the next of the list
`removePrev(p)`	Remove an item (here it is `p`) from the previous of the list
`find(x)`	Searches the specified pattern or the expression (here it is `x`) in the given sequence string and returns true otherwise it returns false
`size()`	Return the number of objects in the list

2.2. How to implement the operations ?

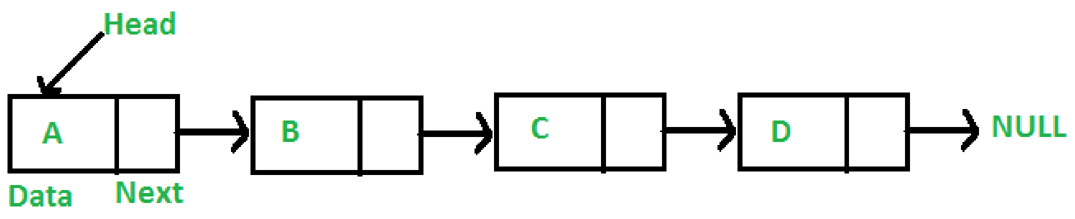


Figure 1: Linked List

A linked structure is a collection of nodes storing data and links to other nodes. In this way, nodes can be located anywhere in memory, and passing from one node of the linked structure to another is accomplished by storing the reference(s) to other node(s) in the structure. Although linked structures can be implemented in a variety of ways, the most flexible implementation is by using a separate object for each node.

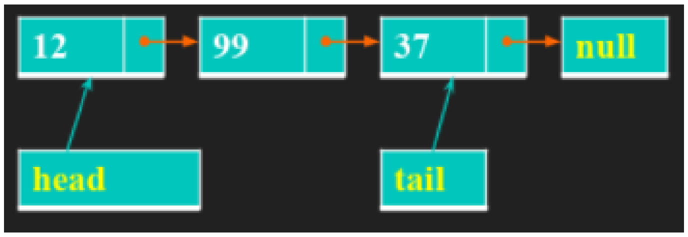
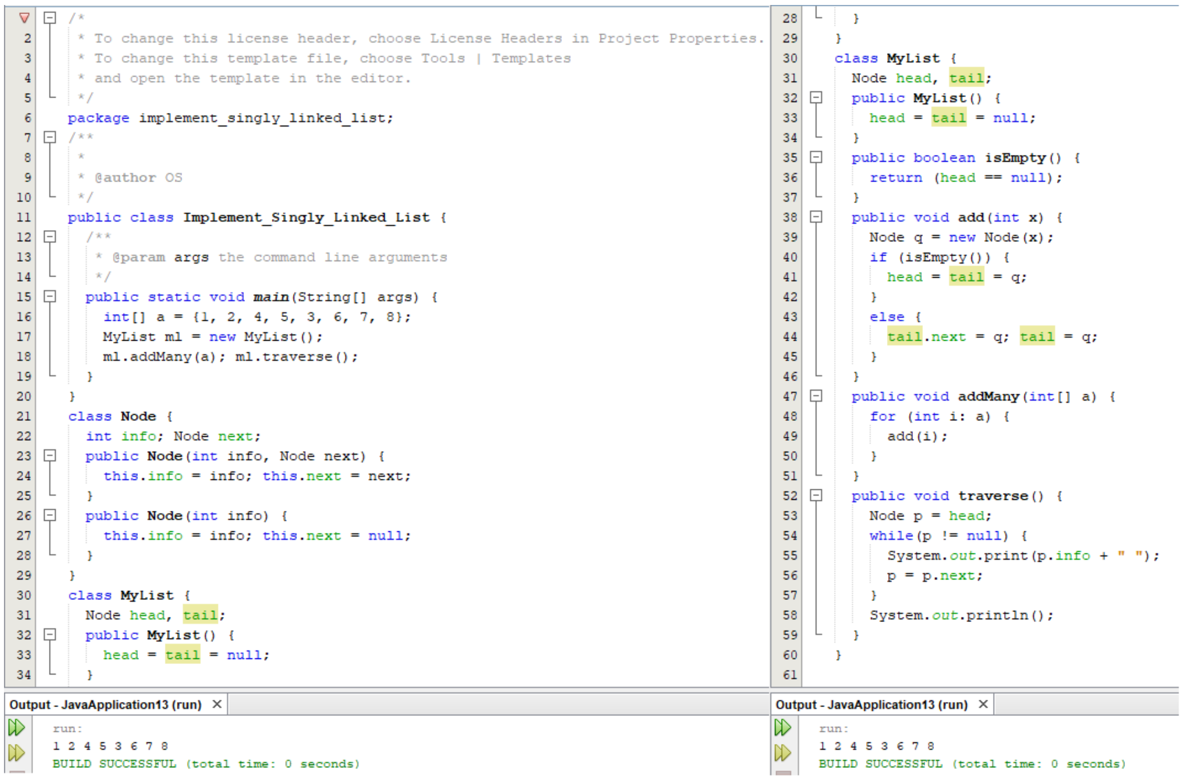


Figure 2: Singly Linked List

Have two types of linked lists are Singly-linked list and Doubly-linked list. Singly linked lists node contains two data fields are info and next. Info stores information which is usable by user. Next links it to its successor in the sequence. Below is some example how to implement the operations (Singly Linked List).



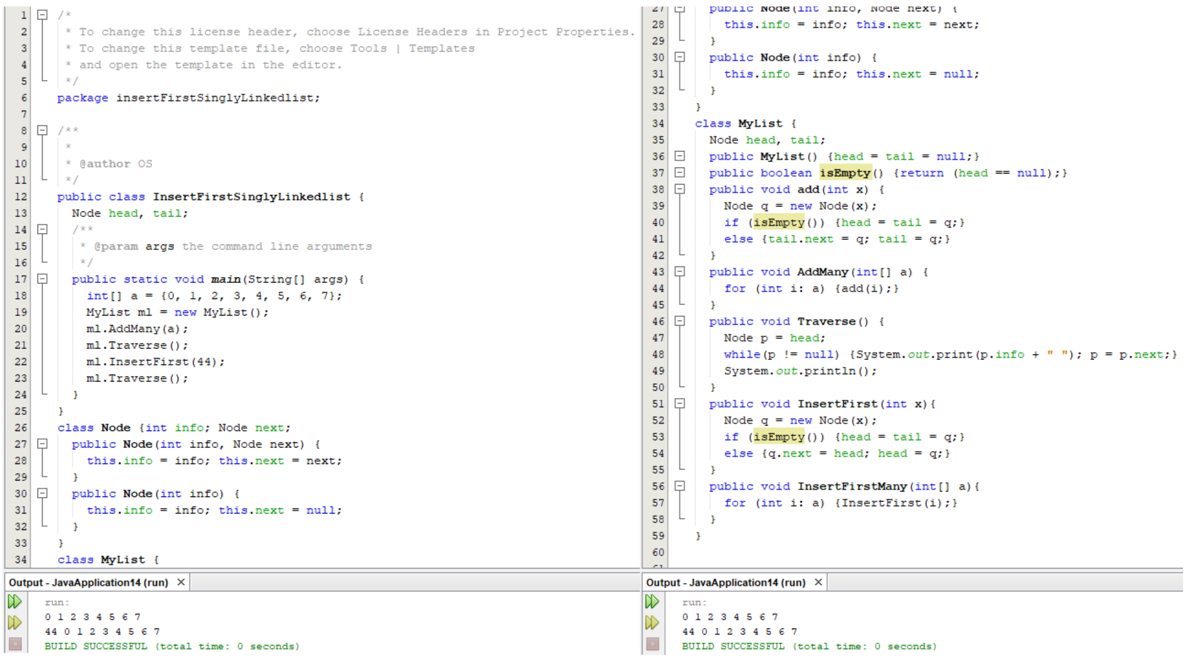
```java
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package implement_singly_linked_list;
/**
 *
 * @author OS
 */
public class Implement_Singly_Linked_List {
  /**
   * @param args the command line arguments
   */
  public static void main(String[] args) {
    int[] a = {1, 2, 4, 5, 3, 6, 7, 8};
    MyList ml = new MyList();
    ml.addMany(a); ml.traverse();
  }
}
class Node {
  int info; Node next;
  public Node(int info, Node next) {
    this.info = info; this.next = next;
  }
  public Node(int info) {
    this.info = info; this.next = null;
  }
}
class MyList {
  Node head, tail;
  public MyList() {
    head = tail = null;
  }
```

```
Output - JavaApplication13 (run)
run:
1 2 4 5 3 6 7 8
BUILD SUCCESSFUL (total time: 0 seconds)
```

```java
  }
}
class MyList {
  Node head, tail;
  public MyList() {
    head = tail = null;
  }
  public boolean isEmpty() {
    return (head == null);
  }
  public void add(int x) {
    Node q = new Node(x);
    if (isEmpty()) {
      head = tail = q;
    }
    else {
      tail.next = q; tail = q;
    }
  }
  public void addMany(int[] a) {
    for (int i: a) {
      add(i);
    }
  }
  public void traverse() {
    Node p = head;
    while(p != null) {
      System.out.print(p.info + " ");
      p = p.next;
    }
    System.out.println();
  }
}
```

```
Output - JavaApplication13 (run)
run:
1 2 4 5 3 6 7 8
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 3: Implement Java Code for Singly Linked List


Output - JavaApplication14 (run)
Output - JavaApplication14 (run)

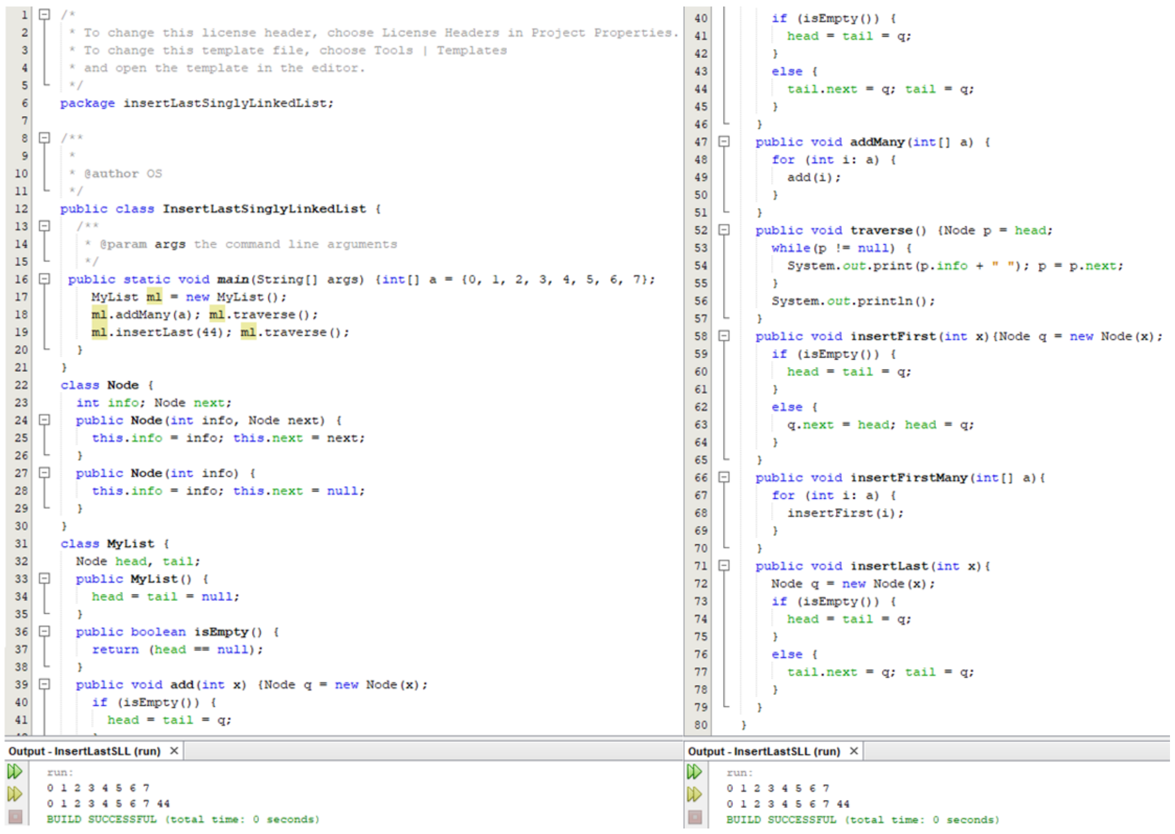
```java
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package insertFirstSinglyLinkedlist;

/**
 *
 * @author OS
 */
public class InsertFirstSinglyLinkedlist {
  Node head, tail;
  /**
   * @param args the command line arguments
   */
  public static void main(String[] args) {
    int[] a = {0, 1, 2, 3, 4, 5, 6, 7};
    MyList ml = new MyList();
    ml.AddMany(a);
    ml.Traverse();
    ml.InsertFirst(44);
    ml.Traverse();
  }
}
class Node {int info; Node next;
  public Node(int info, Node next) {
    this.info = info; this.next = next;
  }
  public Node(int info) {
    this.info = info; this.next = null;
  }
}
class MyList {
```

```
run:
0 1 2 3 4 5 6 7
44 0 1 2 3 4 5 6 7
BUILD SUCCESSFUL (total time: 0 seconds)
```

```java
  public Node(int info, Node next) {
    this.info = info; this.next = next;
  }
  public Node(int info) {
    this.info = info; this.next = null;
  }
}
class MyList {
  Node head, tail;
  public MyList() {head = tail = null;}
  public boolean isEmpty() {return (head == null);}
  public void add(int x) {
    Node q = new Node(x);
    if (isEmpty()) {head = tail = q;}
    else {tail.next = q; tail = q;}
  }
  public void AddMany(int[] a) {
    for (int i: a) {add(i);}
  }
  public void Traverse() {
    Node p = head;
    while(p != null) {System.out.print(p.info + " "); p = p.next;}
    System.out.println();
  }
  public void InsertFirst(int x){
    Node q = new Node(x);
    if (isEmpty()) {head = tail = q;}
    else {q.next = head; head = q;}
  }
  public void InsertFirstMany(int[] a){
    for (int i: a) {InsertFirst(i);}
  }
}
```

```
run:
0 1 2 3 4 5 6 7
44 0 1 2 3 4 5 6 7
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 4: Implement Java Code for inserting Node at the beginning of Singly Linked List



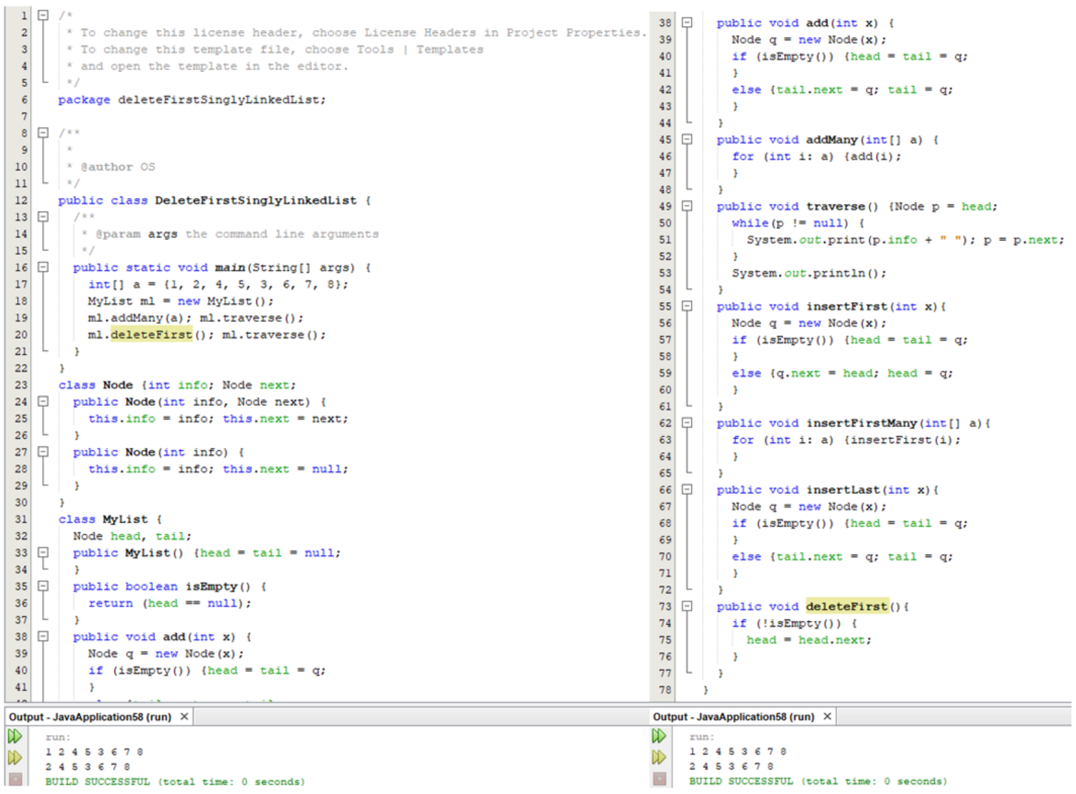
```java
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package insertLastSinglyLinkedList;

/**
 *
 * @author OS
 */
public class InsertLastSinglyLinkedList {
  /**
   * @param args the command line arguments
   */
 public static void main(String[] args) {int[] a = {0, 1, 2, 3, 4, 5, 6, 7};
    MyList ml = new MyList();
    ml.addMany(a); ml.traverse();
    ml.insertLast(44); ml.traverse();
  }
}
class Node {
  int info; Node next;
  public Node(int info, Node next) {
    this.info = info; this.next = next;
  }
  public Node(int info) {
    this.info = info; this.next = null;
  }
}
class MyList {
  Node head, tail;
  public MyList() {
    head = tail = null;
  }
  public boolean isEmpty() {
    return (head == null);
  }
  public void add(int x) {Node q = new Node(x);
    if (isEmpty()) {
      head = tail = q;
```

```
run:
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7 44
BUILD SUCCESSFUL (total time: 0 seconds)
```

```java
    if (isEmpty()) {
      head = tail = q;
    }
    else {
      tail.next = q; tail = q;
    }
  }
  public void addMany(int[] a) {
    for (int i: a) {
      add(i);
    }
  }
  public void traverse() {Node p = head;
    while(p != null) {
      System.out.print(p.info + " "); p = p.next;
    }
    System.out.println();
  }
  public void insertFirst(int x){Node q = new Node(x);
    if (isEmpty()) {
      head = tail = q;
    }
    else {
      q.next = head; head = q;
    }
  }
  public void insertFirstMany(int[] a){
    for (int i: a) {
      insertFirst(i);
    }
  }
  public void insertLast(int x){
    Node q = new Node(x);
    if (isEmpty()) {
      head = tail = q;
    }
    else {
      tail.next = q; tail = q;
    }
  }
}
```

```
run:
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7 44
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 5: Implement Java Code for inserting Node at the end of Singly Linked List



```java
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package deleteFirstSinglyLinkedList;

/**
 *
 * @author OS
 */
public class DeleteFirstSinglyLinkedList {
  /**
   * @param args the command line arguments
   */
  public static void main(String[] args) {
    int[] a = {1, 2, 4, 5, 3, 6, 7, 8};
    MyList ml = new MyList();
    ml.addMany(a); ml.traverse();
    ml.deleteFirst(); ml.traverse();
  }
}
class Node {int info; Node next;
  public Node(int info, Node next) {
    this.info = info; this.next = next;
  }
  public Node(int info) {
    this.info = info; this.next = null;
  }
}
class MyList {
  Node head, tail;
  public MyList() {head = tail = null;
  }
  public boolean isEmpty() {
    return (head == null);
  }
  public void add(int x) {
    Node q = new Node(x);
    if (isEmpty()) {head = tail = q;
    }
```

```java
  public void add(int x) {
    Node q = new Node(x);
    if (isEmpty()) {head = tail = q;
    }
    else {tail.next = q; tail = q;
    }
  }
  public void addMany(int[] a) {
    for (int i: a) {add(i);
    }
  }
  public void traverse() {Node p = head;
    while(p != null) {
      System.out.print(p.info + " "); p = p.next;
    }
    System.out.println();
  }
  public void insertFirst(int x){
    Node q = new Node(x);
    if (isEmpty()) {head = tail = q;
    }
    else {q.next = head; head = q;
    }
  }
  public void insertFirstMany(int[] a){
    for (int i: a) {insertFirst(i);
    }
  }
  public void insertLast(int x){
    Node q = new Node(x);
    if (isEmpty()) {head = tail = q;
    }
    else {tail.next = q; tail = q;
    }
  }
  public void deleteFirst(){
    if (!isEmpty()) {
      head = head.next;
    }
  }
}
```

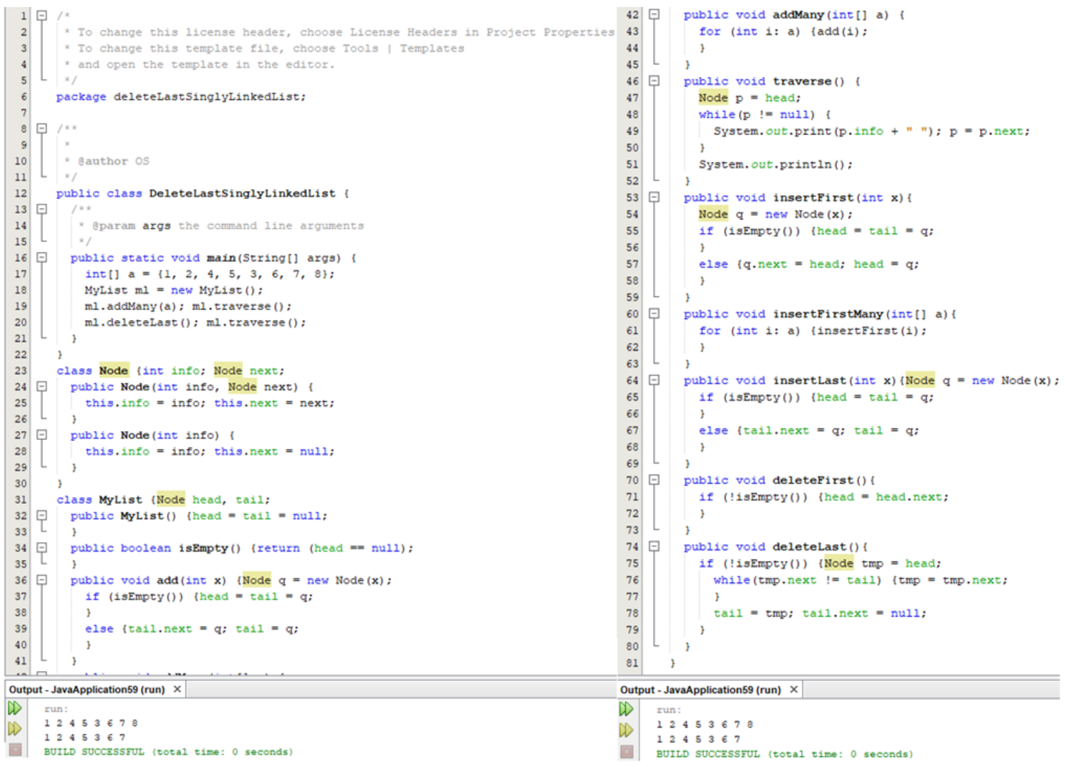
Output - JavaApplication58 (run)

```
run:
1 2 4 5 3 6 7 8
2 4 5 3 6 7 8
BUILD SUCCESSFUL (total time: 0 seconds)
```

Output - JavaApplication58 (run)

```
run:
1 2 4 5 3 6 7 8
2 4 5 3 6 7 8
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 6: Implement Java Code for deleting Node at the beginning of Singly Linked List



```java
/*
 * To change this license header, choose License Headers in Project Properties
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package deleteLastSinglyLinkedList;

/**
 *
 * @author OS
 */
public class DeleteLastSinglyLinkedList {
  /**
   * @param args the command line arguments
   */
  public static void main(String[] args) {
    int[] a = {1, 2, 4, 5, 3, 6, 7, 8};
    MyList ml = new MyList();
    ml.addMany(a); ml.traverse();
    ml.deleteLast(); ml.traverse();
  }
}
class Node {int info; Node next;
  public Node(int info, Node next) {
    this.info = info; this.next = next;
  }
  public Node(int info) {
    this.info = info; this.next = null;
  }
}
class MyList {Node head, tail;
  public MyList() {head = tail = null;
  }
  public boolean isEmpty() {return (head == null);
  }
  public void add(int x) {Node q = new Node(x);
    if (isEmpty()) {head = tail = q;
    }
    else {tail.next = q; tail = q;
    }
  }
```

```java
  public void addMany(int[] a) {
    for (int i: a) {add(i);
    }
  }
  public void traverse() {
    Node p = head;
    while(p != null) {
      System.out.print(p.info + " "); p = p.next;
    }
    System.out.println();
  }
  public void insertFirst(int x){
    Node q = new Node(x);
    if (isEmpty()) {head = tail = q;
    }
    else {q.next = head; head = q;
    }
  }
  public void insertFirstMany(int[] a){
    for (int i: a) {insertFirst(i);
    }
  }
  public void insertLast(int x){Node q = new Node(x);
    if (isEmpty()) {head = tail = q;
    }
    else {tail.next = q; tail = q;
    }
  }
  public void deleteFirst(){
    if (!isEmpty()) {head = head.next;
    }
  }
  public void deleteLast(){
    if (!isEmpty()) {Node tmp = head;
      while(tmp.next != tail) {tmp = tmp.next;
      }
      tail = tmp; tail.next = null;
    }
  }
}
```

Output - JavaApplication59 (run)

```
run:
1 2 4 5 3 6 7 8
1 2 4 5 3 6 7
BUILD SUCCESSFUL (total time: 0 seconds)
```

Output - JavaApplication59 (run)

```
run:
1 2 4 5 3 6 7 8
1 2 4 5 3 6 7
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 7: Implement Java Code for deleting Node at the end of Singly Linked List

P2 Determine the operations of a memory stack and how it is used to implement function calls in a computer.

1. Determine the operations of a memory stack.

1.1. What is a stack ?

A stack is a linear data structure that can be accessed only at one of its ends for storing and retrieving data. It is a Last In/First Out (LIFO) data structure. Anything added will go to the top. Anything removed will be taken from the top and things are removed in the reverse order that they were inserted. In other words, such a stack resembles a stack of trays in a cafeteria. New trays are put on the top of the stack and taken off the top. The last tray put on the stack is the first tray removed from the stack. For this reason, a stack is called a LIFO structure: Last In/First Out.

1.2. What are operations on stack ?

According to the above explanation, a tray can be taken only if there are trays on the stack, and a tray can be added to the stack only if there is enough room; that is, if the stack is not too high. Therefore, a stack is defined in terms of operations that change its status and operations that check this status. Below table is the operations on stack.

Table 2: Operations on stack

Operations on stack	
`clear()`	Clear the stack
`isEmpty()`	Check to see if the stack is empty
`push(el)`	Put the element el on the top of the stack
`pop()`	Take the topmost element from the stack
`topEl()`	Return the topmost element in the stack without removing it

Operations pop and top cannot be performed if the stack is empty. If attempting the execution of pop or top on an empty stack, will throws a StackEmptyException. In addition, operations push sometimes cannot be performed if it’s not enough memory. If attempting the execution of push when there is not enough memory, will throws an OutOfMemoryError.

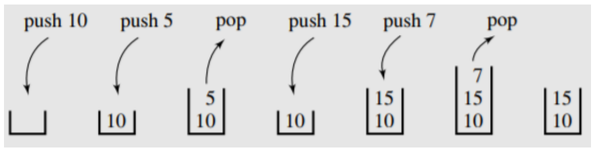
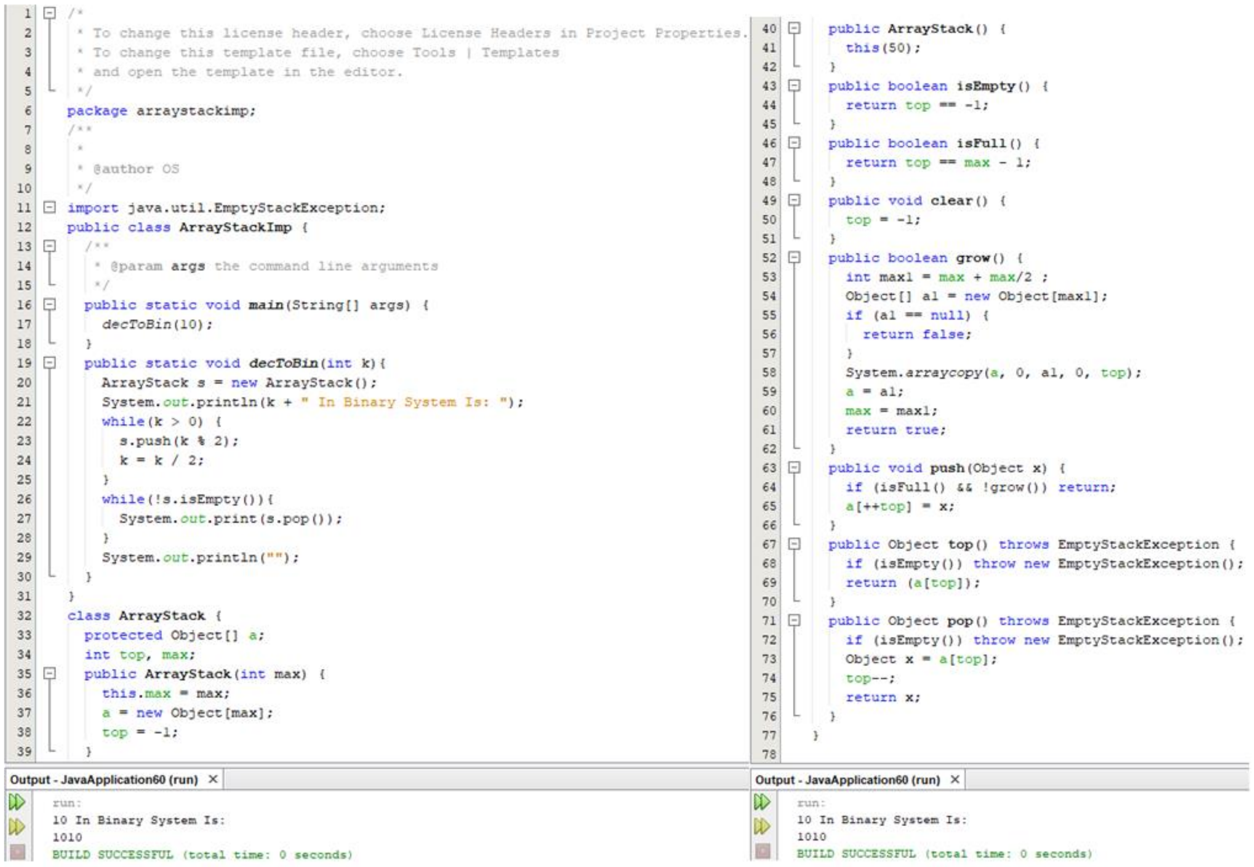


Figure 8: A series of operations executed on a stack

➢ For example: *A series of push and pop operations is shown in Figure 8. After pushing number 10 onto an empty stack, the stack contains only this number. After pushing 5 on the stack, the number is placed on top of 10 so that, when the popping operation is executed, 5 is removed from the stack, because it arrived after 10, and 10 is left on the stack. After pushing 15 and then 7, the topmost element is 7, and this number is removed when executing the popping operation, after which the stack contains 10 at the bottom and 15 above it.*

1.3. How to implement stacks ?



```java
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package arraystackimp;
/**
 *
 * @author OS
 */
import java.util.EmptyStackException;
public class ArrayStackImp {
  /**
   * @param args the command line arguments
   */
  public static void main(String[] args) {
    decToBin(10);
  }
  public static void decToBin(int k){
    ArrayStack s = new ArrayStack();
    System.out.println(k + " In Binary System Is: ");
    while(k > 0) {
      s.push(k % 2);
      k = k / 2;
    }
    while(!s.isEmpty()){
      System.out.print(s.pop());
    }
    System.out.println("");
  }
}
class ArrayStack {
  protected Object[] a;
  int top, max;
  public ArrayStack(int max) {
    this.max = max;
    a = new Object[max];
    top = -1;
  }
  public ArrayStack() {
    this(50);
  }
  public boolean isEmpty() {
    return top == -1;
  }
  public boolean isFull() {
    return top == max - 1;
  }
  public void clear() {
    top = -1;
  }
  public boolean grow() {
    int max1 = max + max/2 ;
    Object[] a1 = new Object[max1];
    if (a1 == null) {
      return false;
    }
    System.arraycopy(a, 0, a1, 0, top);
    a = a1;
    max = max1;
    return true;
  }
  public void push(Object x) {
    if (isFull() && !grow()) return;
    a[++top] = x;
  }
  public Object top() throws EmptyStackException {
    if (isEmpty()) throw new EmptyStackException();
    return (a[top]);
  }
  public Object pop() throws EmptyStackException {
    if (isEmpty()) throw new EmptyStackException();
    Object x = a[top];
    top--;
    return x;
  }
}
```

Output - JavaApplication60 (run)

```
run:
10 In Binary System Is:
1010
BUILD SUCCESSFUL (total time: 0 seconds)
```

Output - JavaApplication60 (run)

```
run:
10 In Binary System Is:
1010
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 9: Array Implementation of Stack - The demo program that can convert 10 to binary

2. **How it is used to implement function calls in a computer.**

2.1. Applications of stacks.

Table 3: Applications of Stacks

Some applications of stacks	
1	Any sort of nesting (such as parentheses)
2	Evaluating arithmetic expressions (and other sorts of expression)
3	Implementing function or method calls
4	Keeping track of previous choices (as in backtracking)
5	Keeping track of choices yet to be made (as in creating a maze)
6	Undo sequence in a text editor
7	Auxiliary data structure for algorithms
8	Component of other data structure
9	Any sort of nesting (such as parentheses)
10	Evaluating arithmetic expressions (and other sorts of expression)
11	Implementing function or method calls
12	Keeping track of previous choices (as in backtracking)
13	Keeping track of choices yet to be made (as in creating a maze)
14	Undo sequence in a text editor
15	Auxiliary data structure for algorithms
16	Component of other data structure

Table 4: Functionalities of the function calls

Functionalities of the function calls		
Common characteristics	`Object[] a`	Array of Object
	`int top,max`	Top is the number currently highest on the stack
		Max is the number that the stack can hold
Constructors	`ArrayStack(int max)`	Initialize an empty stack with a specified max
	`ArrayStack()` – `default max=50`	Initialize an empty stack with a specified max is 50
Common behaviors	`isEmpty()`	Check to see if the stack is empty
	`isFull()`	Tests if the stack is full or not
	`clear()`	Clear the stack
	`grow()`	Increase the size of the internal array
	`push()`	Pushes an item onto the top of the stack
	`top()`	Return the topmost element in the stack without removing it
	`pop()`	Take the topmost element from the stack

2.2. What is method calls and how they work.

If the method has formal parameters, they have to be initialized to the values passed as actual parameters. In addition, the system has to know where to resume execution of the program after the method has finished.

The method can be called by other methods or by the main program (the method `main()`). The information indicating where it has been called from has to be remembered by the system. This could be done by storing the return address in main memory in a place set aside for return addresses, but we do not know in advance how much space might be needed, and allocating too much space for that purpose alone is not efficient.

For a method call, more information has to be stored than just a return address. Therefore, dynamic allocation using the run-time stack is a much better solution.

Each time a method is called, an activation record (AR) is allocated for it and this record contains the following information. *Firstly*, parameters and local variables used in the called method. *Secondly*, dynamic link - a pointer to the caller's activation record. *Thirdly*, return address to resume control by the caller - address of instruction immediately following the call. *Finally*, return value for a method not declared as void, since the size of AR may vary from one call to another so returned value is placed right above the AR of the caller.

Each new AR is placed on top of the run-time stack. When a method terminates, its AR is removed from the top of the run-time stack. Thus, the first AR placed on the stack is the last one removed.

Creating an activation record whenever a method is called allows the system to handle recursion properly. Recursion is calling a method that happens to have the same name as the caller. Therefore, a recursive call is not literally a method calling itself, but rather an instantiation of a method calling another instantiation of the same original. These invocations are represented internally by different activation records and are thus differentiated by the system.

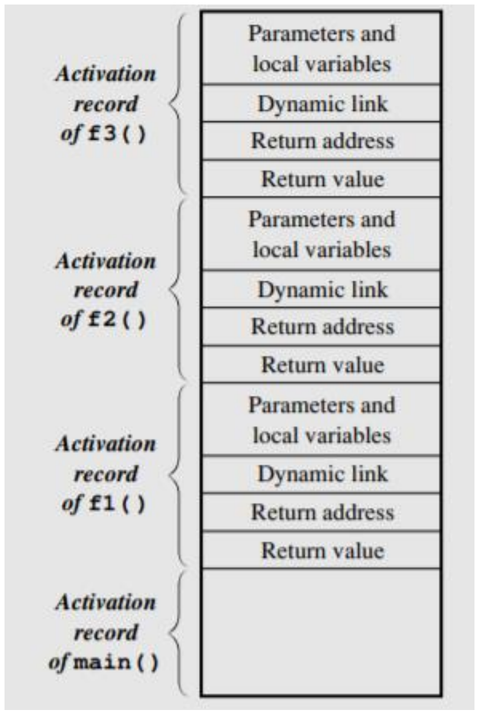

Activation record of f3()
Parameters and local variables
Dynamic link
Return address
Return value
Activation record of f2()
Parameters and local variables
Dynamic link
Return address
Return value
Activation record of f1()
Parameters and local variables
Dynamic link
Return address
Return value
Activation record of main()

Figure 10: How to implement method calls by stack - Contents of the run-time stack when `main`() calls method f1(), f1() calls f2(), and f2() calls f3()

- For example: *Suppose that* `main`*() calls method f1(), f1() calls f2(), and f2() in turn calls f3(). If f3() is being executed, then the state of the run-time stack is as shown in Figure 10 By the nature of the stack, if the activation record for f3() is popped by moving the stack pointer right below the return value of f3(), then f2() resumes execution and now has free access to the private pool of information necessary for reactivation of its execution. On the other hand, if f3() happens to call another method f4(), then the run-time stack increases its height because the activation record for f4() is created on the stack and the activity of f3() is suspended.*

M1 Illustrate, with an example, a concrete data structure for a First in First out (FIFO) queue.

FIFO is an abbreviation for first in, first out. It is a method for handling data structures where the first element is processed first and the newest element is processed last.

FIFO used in data structures, disk scheduling, communications and networking. In data structures, certain data structures like Queue and other variants of Queue uses FIFO approach for processing data. In disk scheduling, disk controllers can use the FIFO as a disk scheduling algorithm to determine the order in which to service disk I/O requests. Finally, in communications and networking, communication network bridges, switches and routers used in computer networks use FIFOs to hold data packets end route to their next destination.

Examples of queues are very much in practice. A typical example of a queue is cars lined up to enter a toll booth. Vehicles that arrive first will be charged in advance and can continue the journey. It can be seen similarly that the way the queue works is to operate first in first out. Below are pictures depicting how the queuing operation works.

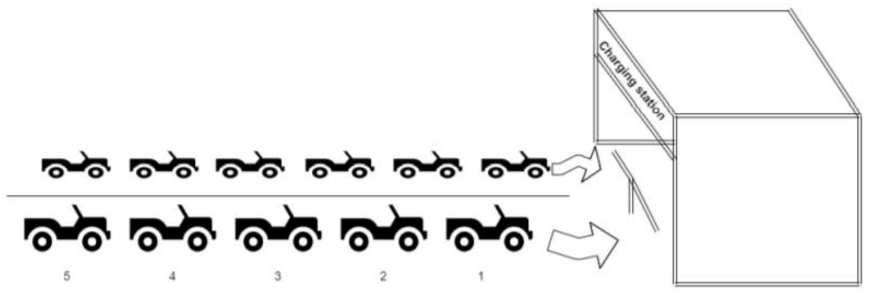


Figure 11: Simulation of queue 1 example

This is a simulated image of cars queuing up in turn to enter the toll booth. Here will number each vehicle to make it easier to track the progress. The cars numbered 1 to 5 and the vehicle numbered 1 are the cars preparing to enter the tollgate.

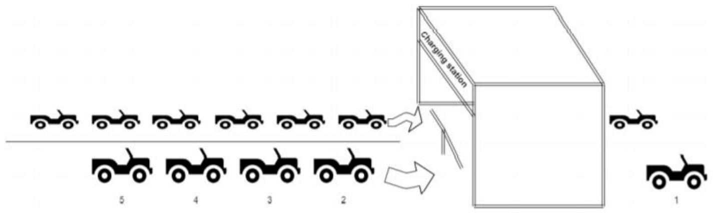
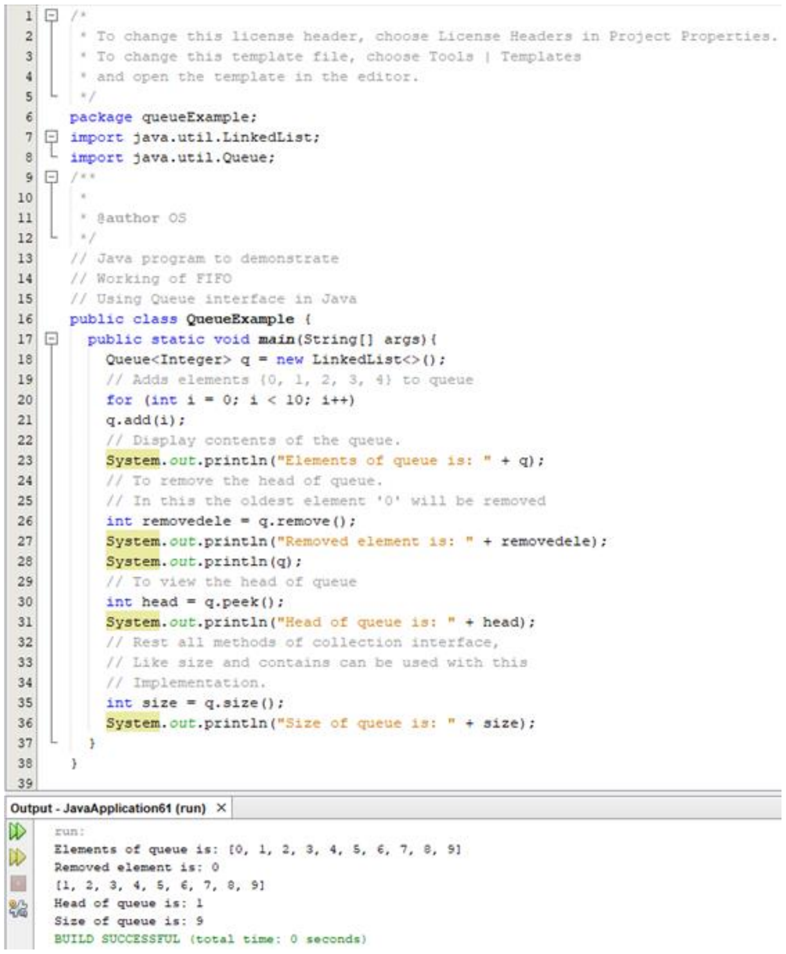


Figure 12: Simulation of queue 2 example

After paying the fee, the car No. 1 got out of the row and continued the journey. The No. 2 car continues to move to the station entrance and the process continues like that first in first out.



```java
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package queueExample;
import java.util.LinkedList;
import java.util.Queue;
/**
 *
 * @author OS
 */
// Java program to demonstrate
// Working of FIFO
// Using Queue interface in Java
public class QueueExample {
  public static void main(String[] args){
    Queue<Integer> q = new LinkedList<>();
    // Adds elements {0, 1, 2, 3, 4} to queue
    for (int i = 0; i < 10; i++)
    q.add(i);
    // Display contents of the queue.
    System.out.println("Elements of queue is: " + q);
    // To remove the head of queue.
    // In this the oldest element '0' will be removed
    int removedele = q.remove();
    System.out.println("Removed element is: " + removedele);
    System.out.println(q);
    // To view the head of queue
    int head = q.peek();
    System.out.println("Head of queue is: " + head);
    // Rest all methods of collection interface,
    // Like size and contains can be used with this
    // Implementation.
    int size = q.size();
    System.out.println("Size of queue is: " + size);
  }
}
```

Output - JavaApplication61 (run)

```
run:
Elements of queue is: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Removed element is: 0
[1, 2, 3, 4, 5, 6, 7, 8, 9]
Head of queue is: 1
Size of queue is: 9
BUILD SUCCESSFUL (total time: 0 seconds)
```

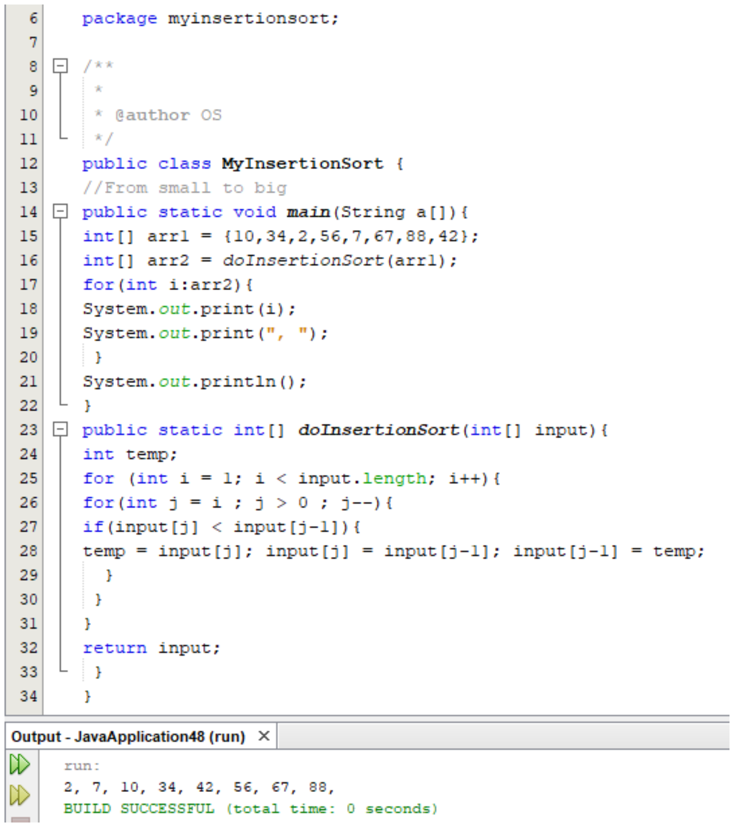
Figure 13: Implement Java Code for First in First out (FIFO) queue

M2 Compare the performance of two sorting algorithms.

There are many sorting algorithms, each one has its advantages and disadvantages. However, I will explain and comparison sorting algorithms Insertion sort and Merge sort.

1. Insertion sort

An insertion sort starts by considering the two first elements in an array A (0-based index). If the element A[1] is less than the element A[0], they are swapped to make them in order. Then, the third element A[2] is considered and inserted into its proper place, if it is lesser than the previous two elements then the previous elements are shifted to the right by one position and put A[2] to the shifted empty position. Making A[0] at position 1, A[1] at position 2 and A[2] at position 0. If A[2] is not less than both its predecessors, it stays in its current position. Summary, each element A[i] is inserted into its proper location j such that such that 0 ≤ j ≤ i and all elements greater than A[i] is shifted to the right by one position.



```
package myinsertionsort;

/**
 *
 * @author OS
 */
public class MyInsertionSort {
//From small to big
public static void main(String a[]){
int[] arr1 = {10,34,2,56,7,67,88,42};
int[] arr2 = doInsertionSort(arr1);
for(int i:arr2){
System.out.print(i);
System.out.print(", ");
 }
System.out.println();
}
public static int[] doInsertionSort(int[] input){
int temp;
for (int i = 1; i < input.length; i++){
for(int j = i ; j > 0 ; j--){
if(input[j] < input[j-1]){
temp = input[j]; input[j] = input[j-1]; input[j-1] = temp;
  }
 }
}
return input;
 }
}
```

Output - JavaApplication48 (run)

```
run:
2, 7, 10, 34, 42, 56, 67, 88,
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 14: Implement Java Code for Insertion Sort

Insertion sort runs well for partially sorted array. Because when we are considering an element A[i] with i > 0 during insertion sort, it means that all the elements from A[0…i-1] is already sorted. Initially i = 1, this means that the sub-array A[0…0] is sorted. It is trivial that A[0] is sorted, then A[1] is put into its proper location, by shifting the element A[0] if it is greater than A[1] to the right by one position, making the sub-array A[0…1] sorted. Then i = 2, this means sub-array A[0…1] is sorted which is true based on the previous step. Then A[2] is put into its proper location, by shifting the elements in A[0…1] that is greater than A[2] to the right by one position, making the sub-array A[0…2] sorted. For i = j such that 2 < j < n, it follows that A[0…j-1] is already sorted and A[j] is put into its proper location making the sub-array A[0…j] sorted. From that we can see that, when we are considering an element A[i] with i > 0 during insertion sort and the sub-array A[0…i-1] is sorted, if A[i] > A[i-1] then A[i] is already in its proper location hence no swapping of elements is performed. The best case for insertion sort is when the array is already sorted. One comparison is made for each i, starting at i = 1, so there is n – 1 comparison making it time complexity O(n). The worst case for insertion sort is when the array is already sorted but in reverse order. In this case, for every element A[i] where i > 0, it is always lesser than elements from A[0]…[i-1]. This means there is i comparisons for each i from i = 1 to i = n-1. Making the sum:

$$\sum_{i=1}^{n-1} i = 1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2} = O(n^2)$$

The above formula gives the upper-bound for the worst-case time complexity which is O(n2). In the average-case, this depends how far away the element A[i] that we are considering from its proper location. If it is already in proper location, then only perform one compare between A[i] and A[i-1]. If it is one location away from its proper location, then two comparisons are performed between A[i], A[i-1] and A[i-2]. Summary, if it is j positions away then we do j+1 comparison. Assume that every element in the array have uniform probability of being in its proper location. So, the average number of comparisons for A[i] of iteration i is:

$$\frac{1 + 2 + \cdots + i - 1}{i} = \frac{\frac{1}{2} i(i+1)}{i} = \frac{i+1}{2}$$

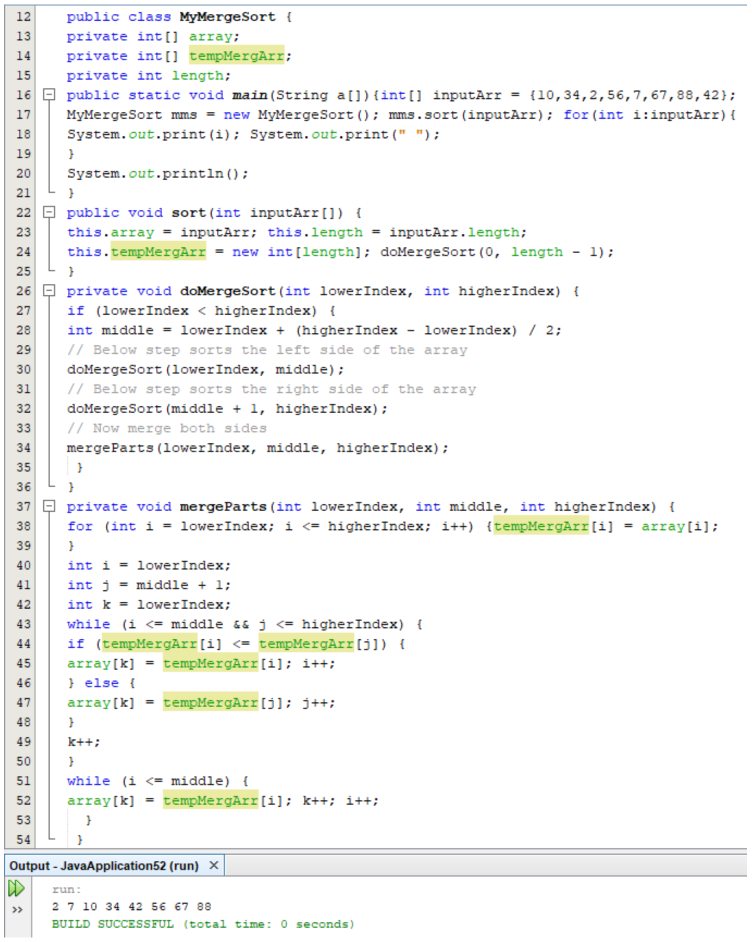
Next, we can calculate the total number of comparisons by summing it through all the i from 1 to n-1:

$$\sum_{i=1}^{n-1}\frac{i+1}{2}=\frac{1}{2}\sum_{i=1}^{n-1}i+\sum_{i=1}^{n-1}\frac{1}{2}=\frac{\frac{1}{2}n(n-1)}{2}+\frac{1}{2}(n-1)=\frac{n^2+n-2}{4}$$

The upper-bound average-case running-time for insertion O(n2) and approximately one-half of the complexity in the worst case.

2. Merge sort

Merge sort different completely from insertion sort, it works by following the divide-and-conquer paradigm. The sorting takes place by sorting two halves of the array and merge it into one. But this requires the two halves to be already sorted which can be accomplish by merging the sorted two halves of these halves. Can see that, there is a recurrence relation in merge sort.



```java
public class MyMergeSort {
private int[] array;
private int[] tempMergArr;
private int length;
public static void main(String a[]){int[] inputArr = {10,34,2,56,7,67,88,42};
MyMergeSort mms = new MyMergeSort(); mms.sort(inputArr); for(int i:inputArr){
System.out.print(i); System.out.print(" ");
}
System.out.println();
}
public void sort(int inputArr[]) {
this.array = inputArr; this.length = inputArr.length;
this.tempMergArr = new int[length]; doMergeSort(0, length - 1);
}
private void doMergeSort(int lowerIndex, int higherIndex) {
if (lowerIndex < higherIndex) {
int middle = lowerIndex + (higherIndex - lowerIndex) / 2;
// Below step sorts the left side of the array
doMergeSort(lowerIndex, middle);
// Below step sorts the right side of the array
doMergeSort(middle + 1, higherIndex);
// Now merge both sides
mergeParts(lowerIndex, middle, higherIndex);
 }
}
private void mergeParts(int lowerIndex, int middle, int higherIndex) {
for (int i = lowerIndex; i <= higherIndex; i++) {tempMergArr[i] = array[i];
}
int i = lowerIndex;
int j = middle + 1;
int k = lowerIndex;
while (i <= middle && j <= higherIndex) {
if (tempMergArr[i] <= tempMergArr[j]) {
array[k] = tempMergArr[i]; i++;
} else {
array[k] = tempMergArr[j]; j++;
}
k++;
}
while (i <= middle) {
array[k] = tempMergArr[i]; k++; i++;
  }
 }
```

```
Output - JavaApplication52 (run)
run:
2 7 10 34 42 56 67 88
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 15: Implement Java Code for Merge Sort

Suppose n is a power of 2 and T(n) denotes the worst-case running-time for merge sorting an array of n elements. Deduced following equation:

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n + n = 2T\left(\frac{n}{2}\right) + 2n$$

Based on the above equation, will get the following recurrence relation (where the constant term is the time it takes to copy the elements into the auxiliary array and merge the two halves array):

$$\begin{cases} T(1) = 1 & if\ n = 1 \\ T(n) = 2T\left(\frac{n}{2}\right) + 2n\ ifn > 1 \end{cases}$$

Next, can calculate the worst-case time complexity as then:

$$T(n) = 2T\left(\frac{n}{2}\right) + 2n = 4T\left(\frac{n}{4}\right) + 4n = 8T\left(\frac{n}{8}\right) + 6n = \cdots = 2^kT\left(\frac{n}{2^k}\right) + 2kn$$

Based on the last equation, deduced:

$$\Leftrightarrow \frac{n}{2^k} = 1 \Leftrightarrow n = 2^k \Leftrightarrow \log_2 n = k \Longrightarrow 2^{\log_2 n}T(1) + 2n\log_2 n = n + 2n\log_2 n$$
$$= n(1 + 2\log_2 n) = O(n\log_2 n)$$

From that can see that, the worst-case time complexity for merge sort is O(nlog2n). The best case running-time and average running-time for merge sort is the same as its worst-case time since it also need to divide the array into its smallest sub-array size and start merging from there. However, still one drawback from merge sort it is that it requires to create additional storage for storing copy of the array which can affect a lot on its running time when sorting large array. In addition, can make merge sort more efficient by implementing it with iterative method instead of recursion and use insertion sort for small size array.

3. Compare the performance of insertion and bubble sort

Based on the given running-time of two sorting algorithms, we can come up with the following comparison table.

Table 5: Compare the performance of two sorting algorithms

Type	**Time**		
	Best	Average	Worst
Insertion Sort	O(n)	O(n^2)	O(n^2)
Merge Sort	O(nlog2n)	O(nlog2n)	O(nlog2n)

From the above explanations, it can be seen that: merge sort out performs insertion sort for large number n while insertion sort is better than merge sort when the array is already sorted or partially sorted. So, can combine both merge sort and insertion sort to get the best out efficiency in both algorithms. Because when sorting small size array, the merge sort time complexity is always the same for any case while insertion sort can perform better since it can be the case that the array is partially sorted so it will be more efficient. While for larger array size, merge sort certainly performs better due to it logarithmic time complexity. Though we should note that this is just the upper-bound for the algorithms which means it only really matters for large n. For small size array their running-time should be relatively as efficient.

D1 Analyze the operation, using illustrations, of two network shortest path algorithms, providing an example of each.

1. Dijkstra's Algorithm

Dijkstra's Algorithm is the widely used algorithm to find the shortest routes in a graph or a tree. Given a weighted graph and a starting (source) vertex in the graph, Dijkstra's algorithm is used to find the shortest distance from the source node to all the other nodes in the graph. As a result of the running Dijkstra's algorithm on a graph, we obtain the shortest path tree (SPT) with the source vertex as root. In Dijkstra's algorithm, we maintain two sets or lists. One contains the vertices that are a part of the shortest-path tree (SPT) and the other contains vertices that are being evaluated to be included in SPT. Hence for every iteration, we find a vertex from the second list that has the shortest path. The pseudocode for the Dijkstra's shortest path algorithm is given below.

```
procedure dijkstra(G, S)
     G-> graph; S->starting vertex
begin
    for each vertex V in G
//initialization; initial path set to infinite
        path[V] <- infinite
        previous[V] <- NULL
        If V != S, add V to Priority Queue PQueue
    path [S] <- 0
    while PQueue IS NOT EMPTY
        U <- Extract MIN from PQueue
        for each unvisited adjacent_node  V of U
            tempDistance <- path [U] + edge_weight(U, V)
            if tempDistance < path [V]
                path [V] <- tempDistance
                previous[V] <- U
    return path[], previous[]
end
```

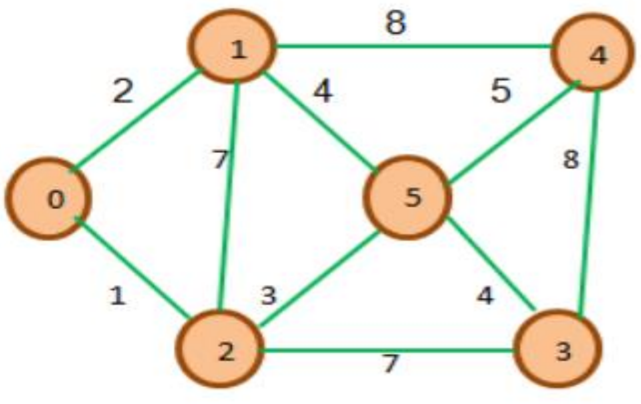
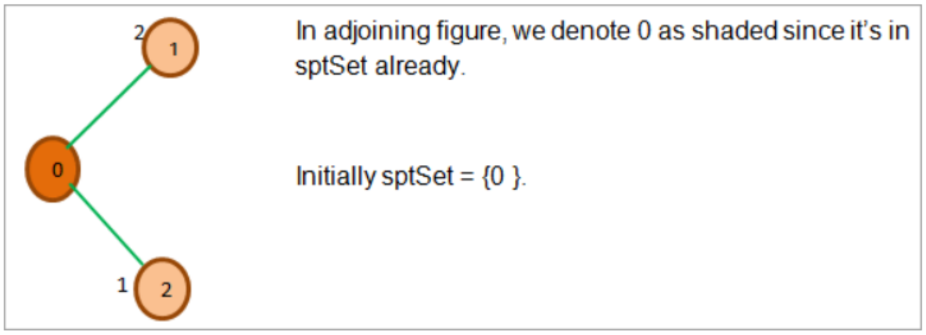


Figure 16: Sample graph the Dijkstra's shortest path algorithm

Initially, the SPT (Shortest Path Tree) set is set to infinity. So, to begin with we put the vertex 0 in `sptSet`. Means, `sptSet = {0, INF, INF, INF, INF, INF}`. Next with vertex 0 in `sptSet`, we will explore its neighbors. Vertices 1 and 2 are two adjacent nodes of 0 with distance 2 and 1 respectively.



In adjoining figure, we denote 0 as shaded since it's in sptSet already.

Initially sptSet = {0 }.

Figure 17: Analyze sample graph

In the above figure, we have also updated each adjacent vertex are 1 and 2 with their respective distance from source vertex 0. Now we see that vertex 2 has a minimum distance. So next we add vertex 2 to the `sptSet`. Also, we explore the neighbors of vertex 2.

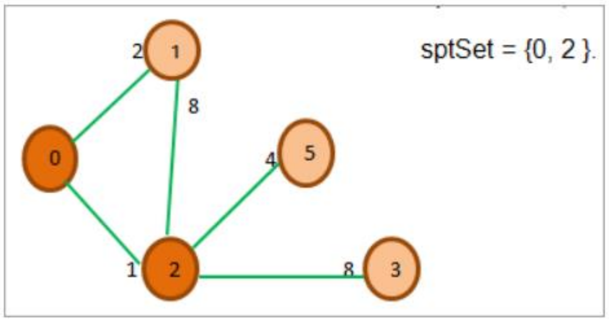


Figure 18: Analyze sample graph

Now we look for the vertex with minimum distance and those that are not there in `spt`. We pick vertex 1 with distance 2.

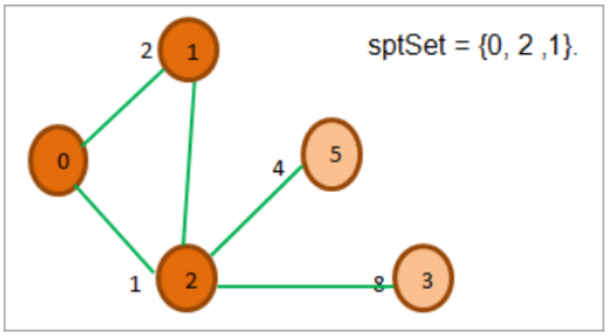


Figure 19: Analyze sample graph

As we see in the above figure, out of all the adjacent nodes of 2, 0, and 1 are already in `sptSet` so we ignore them. Out of the adjacent nodes 5 and 3, 5 have the least cost. So, we add it to the `sptSet` and explore its adjacent nodes.

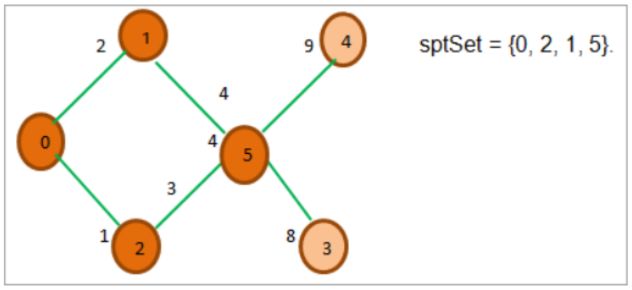


Figure 20: Analyze sample graph

In the above figure, we see that except for nodes 3 and 4, all the other nodes are in `sptSet`. Out of 3 and 4, node 3 has the least cost. So, we put it in `sptSet`.

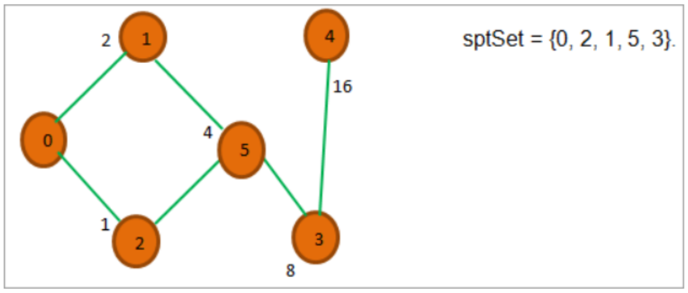
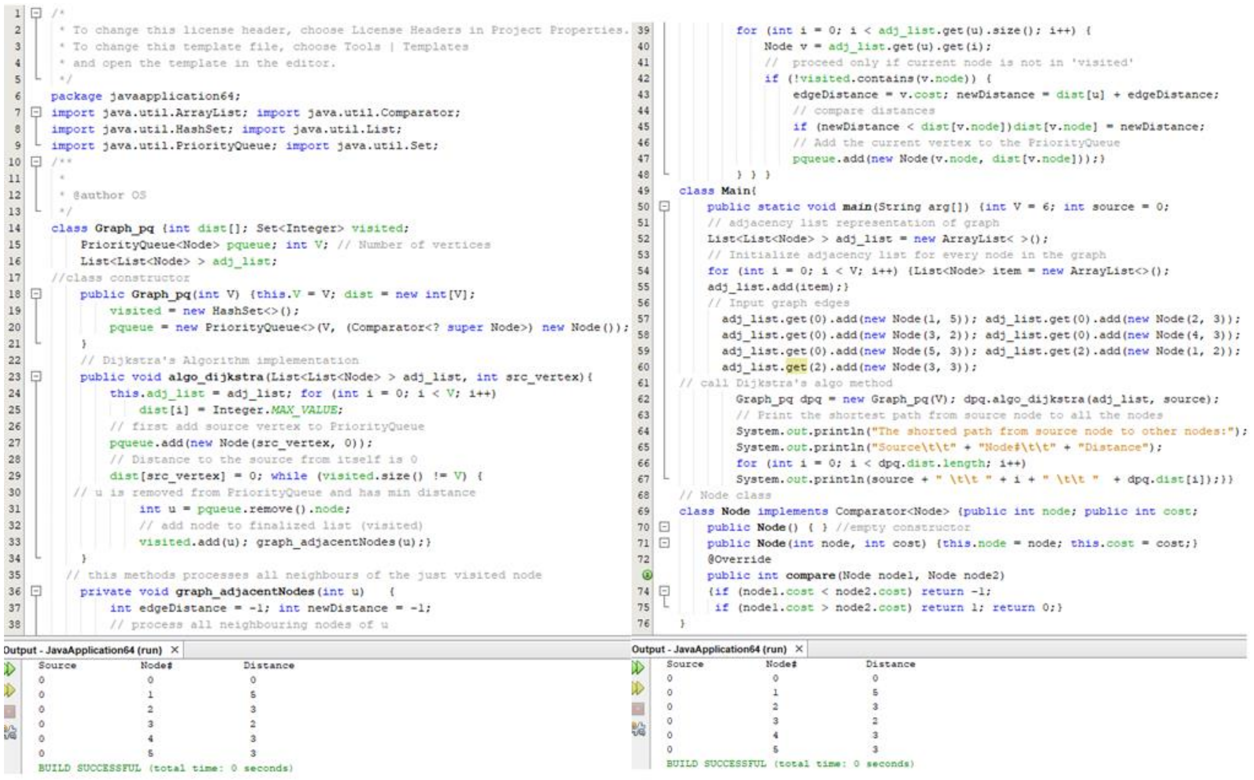


Figure 21: Analyze sample graph

As shown above, now we have only one vertex left i.e. 4 and its distance from the root node is 16. Finally, we put it in `sptSet` to get the final `sptSet` = {0, 2, 1, 5, 3, 4} that gives us the distance of each vertex from the source node 0.

Implementation of Dijkstra's shortest path algorithm in Java can be achieved using two ways. We can either use priority queues and adjacency list or we can use adjacency matrix and arrays.



```java
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package javaapplication64;
import java.util.ArrayList; import java.util.Comparator;
import java.util.HashSet; import java.util.List;
import java.util.PriorityQueue; import java.util.Set;
/**
 *
 * @author OS
 */
class Graph_pq {int dist[]; Set<Integer> visited;
    PriorityQueue<Node> pqueue; int V; // Number of vertices
    List<List<Node> > adj_list;
//class constructor
    public Graph_pq(int V) {this.V = V; dist = new int[V];
        visited = new HashSet<>();
        pqueue = new PriorityQueue<>(V, (Comparator<? super Node>) new Node());
    }
    // Dijkstra's Algorithm implementation
    public void algo_dijkstra(List<List<Node> > adj_list, int src_vertex){
        this.adj_list = adj_list; for (int i = 0; i < V; i++)
            dist[i] = Integer.MAX_VALUE;
        // first add source vertex to PriorityQueue
        pqueue.add(new Node(src_vertex, 0));
        // Distance to the source from itself is 0
        dist[src_vertex] = 0; while (visited.size() != V) {
      // u is removed from PriorityQueue and has min distance
            int u = pqueue.remove().node;
            // add node to finalized list (visited)
            visited.add(u); graph_adjacentNodes(u);}
    }
   // this methods processes all neighbours of the just visited node
    private void graph_adjacentNodes(int u)    {
        int edgeDistance = -1; int newDistance = -1;
        // process all neighbouring nodes of u
        for (int i = 0; i < adj_list.get(u).size(); i++) {
            Node v = adj_list.get(u).get(i);
            //  proceed only if current node is not in 'visited'
            if (!visited.contains(v.node)) {
                edgeDistance = v.cost; newDistance = dist[u] + edgeDistance;
                // compare distances
                if (newDistance < dist[v.node])dist[v.node] = newDistance;
                // Add the current vertex to the PriorityQueue
                pqueue.add(new Node(v.node, dist[v.node]));}
        } } }
class Main{
    public static void main(String arg[]) {int V = 6; int source = 0;
    // adjacency list representation of graph
    List<List<Node> > adj_list = new ArrayList< >();
    // Initialize adjacency list for every node in the graph
    for (int i = 0; i < V; i++) {List<Node> item = new ArrayList<>();
    adj_list.add(item);}
    // Input graph edges
      adj_list.get(0).add(new Node(1, 5)); adj_list.get(0).add(new Node(2, 3));
      adj_list.get(0).add(new Node(3, 2)); adj_list.get(0).add(new Node(4, 3));
      adj_list.get(0).add(new Node(5, 3)); adj_list.get(2).add(new Node(1, 2));
      adj_list.get(2).add(new Node(3, 3));
// call Dijkstra's algo method
        Graph_pq dpq = new Graph_pq(V); dpq.algo_dijkstra(adj_list, source);
        // Print the shortest path from source node to all the nodes
        System.out.println("The shorted path from source node to other nodes:");
        System.out.println("Source\t\t" + "Node#\t\t" + "Distance");
        for (int i = 0; i < dpq.dist.length; i++)
        System.out.println(source + " \t\t " + i + " \t\t "  + dpq.dist[i]);}}
// Node class
class Node implements Comparator<Node> {public int node; public int cost;
    public Node() { } //empty constructor
    public Node(int node, int cost) {this.node = node; this.cost = cost;}
    @Override
    public int compare(Node node1, Node node2)
    {if (node1.cost < node2.cost) return -1;
     if (node1.cost > node2.cost) return 1; return 0;}
}
```

Output - JavaApplication64 (run)

```
Source          Node#           Distance
0                0               0
0                1               5
0                2               3
0                3               2
0                4               3
0                5               3
BUILD SUCCESSFUL (total time: 0 seconds)
```

Output - JavaApplication64 (run)

```
Source          Node#           Distance
0                0               0
0                1               5
0                2               3
0                3               2
0                4               3
0                5               3
BUILD SUCCESSFUL (total time: 0 seconds)
```

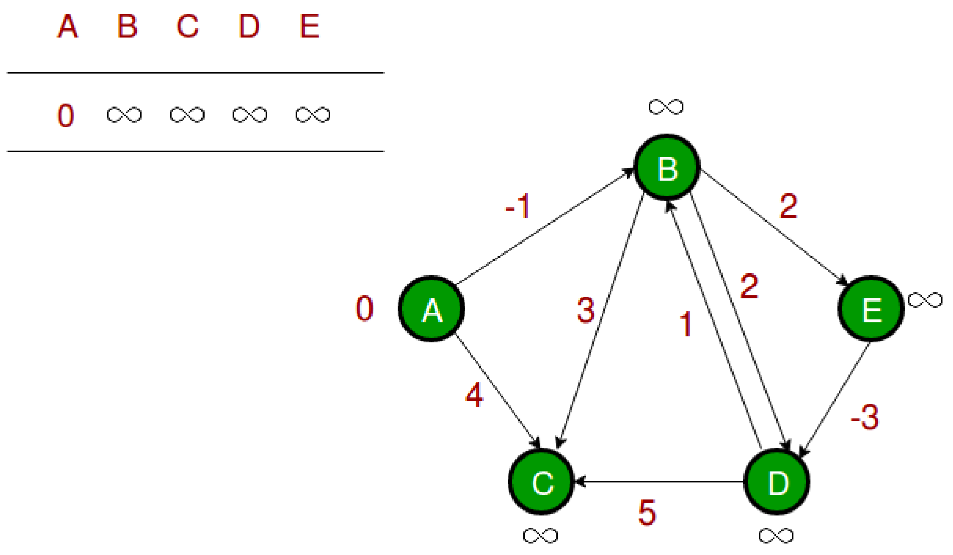
Figure 22: Implementation of Dijkstra's Algorithm in Java - Using A Priority Queue

2. Bellman Ford's Algorithm

Bellman Ford's algorithm is used to find the shortest paths from the source vertex to all other vertices in a weighted graph. It depends on the following concept, shortest path contains at most `n - 1` edges, because the shortest path couldn't have a cycle. Bellman-Ford do work for graphs with negative weight edges. It is also simpler than Dijkstra and suites well for distributed systems. But time complexity of Bellman-Ford is *O*(*VE*), which is more than Dijkstra.

<u>Algorithm Steps</u>: The outer loop traverses from 0 --> `n - 1`. Loop over all edges, check if the next node distance > current node distance + edge weight, in this case update the next node distance to "current node distance + edge weight". This algorithm depends on the relaxation principle where the shortest distance for all vertices is gradually replaced by more accurate values until eventually reaching the optimum solution. In the beginning all vertices have a distance of "Infinity", but only the distance of the source vertex = 0, then update all the connected vertices with the new distances (source vertex distance + edge weights), then apply the same concept for the new vertices with new distances and so on.

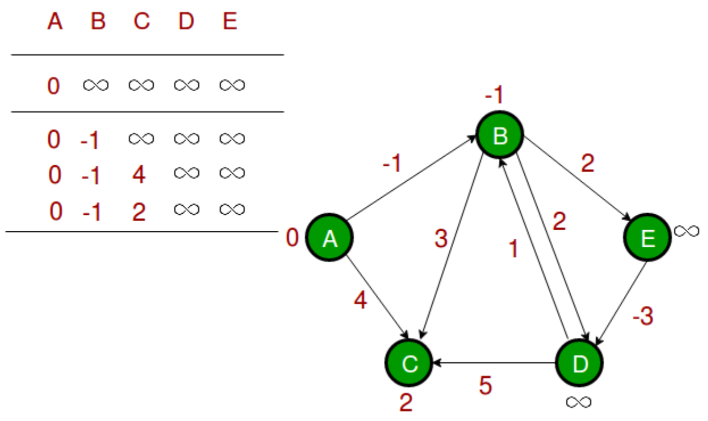
<u>For example</u>: Let the given source vertex be 0. Initialize all distances as infinite, except the distance to the source itself. Total number of vertices in the graph is 5, so all edges must be processed 4 times.



A	B	C	D	E
0	∞	∞	∞	∞

Figure 23: Sample graph the Bellman Ford's shortest path algorithm

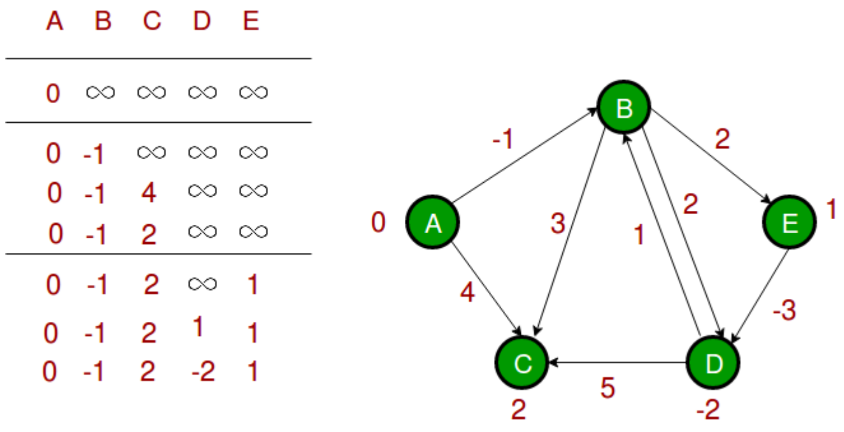
Let all edges are processed in the following order: (B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D). We get the following distances when all edges are processed the first time. The first row shows initial distances. The second row shows distances when edges (B, E), (D, B), (B, D) and (A, B) are processed. The third row shows distances when (A, C) is processed. The fourth row shows when (D, C), (B, C) and (E, D) are processed.



A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞

Figure 24: Analyze sample graph

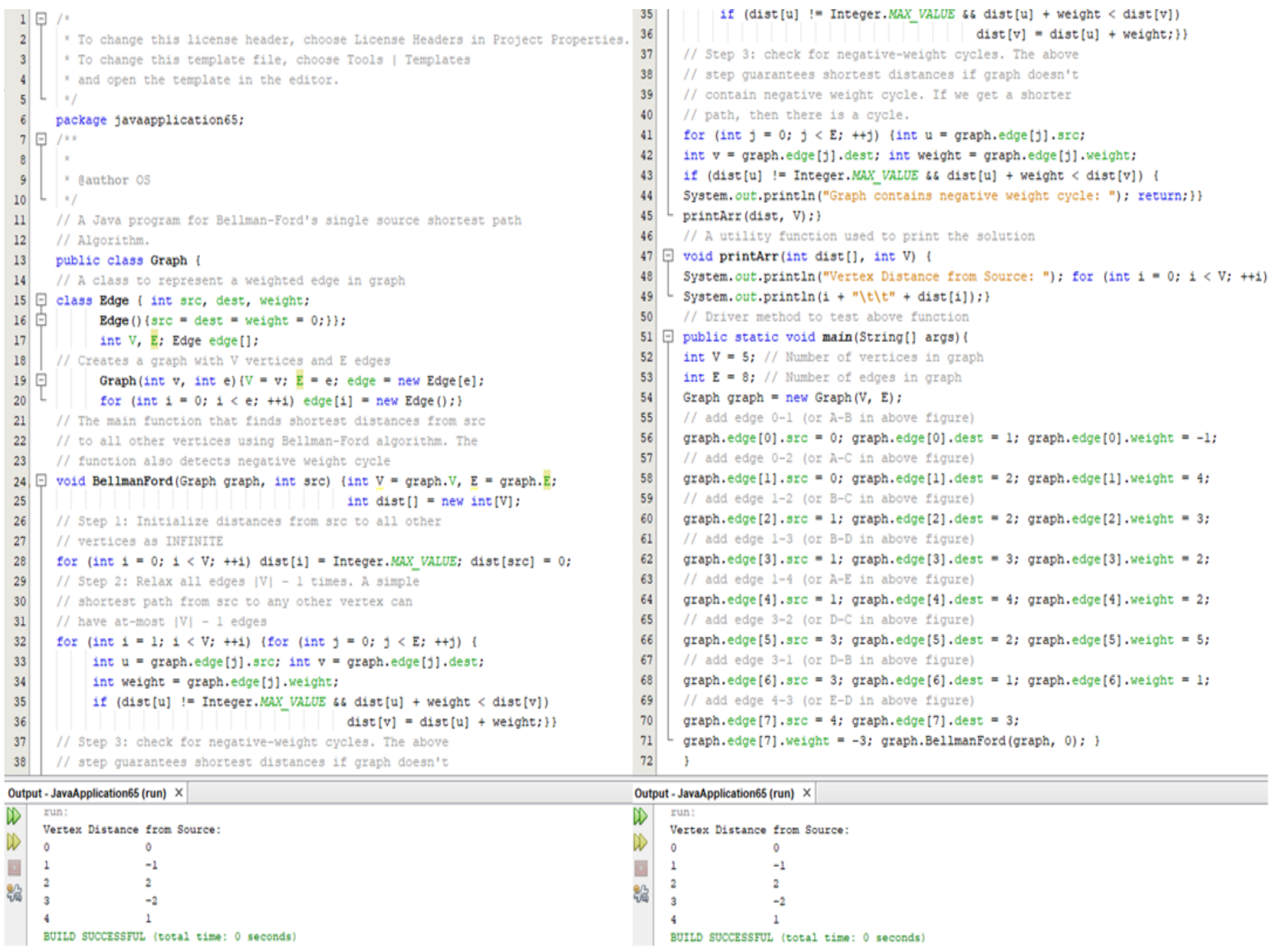
The first iteration guarantees to give all shortest paths which are at most 1 edge long. We get the following distances when all edges are processed second time - The last row shows final values.



A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1
0	-1	2	-2	1

Figure 25: Analyze sample graph

The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations don't update the distances.



```java
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package javaapplication65;
/**
 *
 * @author OS
 */
// A Java program for Bellman-Ford's single source shortest path
// Algorithm.
public class Graph {
// A class to represent a weighted edge in graph
class Edge { int src, dest, weight;
        Edge(){src = dest = weight = 0;}};
        int V, E; Edge edge[];
// Creates a graph with V vertices and E edges
        Graph(int v, int e){V = v; E = e; edge = new Edge[e];
        for (int i = 0; i < e; ++i) edge[i] = new Edge();}
// The main function that finds shortest distances from src
// to all other vertices using Bellman-Ford algorithm. The
// function also detects negative weight cycle
void BellmanFord(Graph graph, int src) {int V = graph.V, E = graph.E;
                                        int dist[] = new int[V];
// Step 1: Initialize distances from src to all other
// vertices as INFINITE
for (int i = 0; i < V; ++i) dist[i] = Integer.MAX_VALUE; dist[src] = 0;
// Step 2: Relax all edges |V| - 1 times. A simple
// shortest path from src to any other vertex can
// have at-most |V| - 1 edges
for (int i = 1; i < V; ++i) {for (int j = 0; j < E; ++j) {
    int u = graph.edge[j].src; int v = graph.edge[j].dest;
    int weight = graph.edge[j].weight;
    if (dist[u] != Integer.MAX_VALUE && dist[u] + weight < dist[v])
                                        dist[v] = dist[u] + weight;}}
// Step 3: check for negative-weight cycles. The above
// step guarantees shortest distances if graph doesn't
```

```java
    if (dist[u] != Integer.MAX_VALUE && dist[u] + weight < dist[v])
                                        dist[v] = dist[u] + weight;}}
// Step 3: check for negative-weight cycles. The above
// step guarantees shortest distances if graph doesn't
// contain negative weight cycle. If we get a shorter
// path, then there is a cycle.
for (int j = 0; j < E; ++j) {int u = graph.edge[j].src;
int v = graph.edge[j].dest; int weight = graph.edge[j].weight;
if (dist[u] != Integer.MAX_VALUE && dist[u] + weight < dist[v]) {
System.out.println("Graph contains negative weight cycle: "); return;}}
printArr(dist, V);}
// A utility function used to print the solution
void printArr(int dist[], int V) {
System.out.println("Vertex Distance from Source: "); for (int i = 0; i < V; ++i)
System.out.println(i + "\t\t" + dist[i]);}
// Driver method to test above function
public static void main(String[] args){
int V = 5; // Number of vertices in graph
int E = 8; // Number of edges in graph
Graph graph = new Graph(V, E);
// add edge 0-1 (or A-B in above figure)
graph.edge[0].src = 0; graph.edge[0].dest = 1; graph.edge[0].weight = -1;
// add edge 0-2 (or A-C in above figure)
graph.edge[1].src = 0; graph.edge[1].dest = 2; graph.edge[1].weight = 4;
// add edge 1-2 (or B-C in above figure)
graph.edge[2].src = 1; graph.edge[2].dest = 2; graph.edge[2].weight = 3;
// add edge 1-3 (or B-D in above figure)
graph.edge[3].src = 1; graph.edge[3].dest = 3; graph.edge[3].weight = 2;
// add edge 1-4 (or A-E in above figure)
graph.edge[4].src = 1; graph.edge[4].dest = 4; graph.edge[4].weight = 2;
// add edge 3-2 (or D-C in above figure)
graph.edge[5].src = 3; graph.edge[5].dest = 2; graph.edge[5].weight = 5;
// add edge 3-1 (or D-B in above figure)
graph.edge[6].src = 3; graph.edge[6].dest = 1; graph.edge[6].weight = 1;
// add edge 4-3 (or E-D in above figure)
graph.edge[7].src = 4; graph.edge[7].dest = 3;
graph.edge[7].weight = -3; graph.BellmanFord(graph, 0); }
}
```

Output - JavaApplication65 (run)

```
run:
Vertex Distance from Source:
0               0
1               -1
2               2
3               -2
4               1
BUILD SUCCESSFUL (total time: 0 seconds)
```

Output - JavaApplication65 (run)

```
run:
Vertex Distance from Source:
0               0
1               -1
2               2
3               -2
4               1
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 26: Implementation of Bellman Ford's Algorithm in Java

LO2 Specify abstract data types and algorithms in a formal notation

P3 Using an imperative definition, specify the abstract data type for a software stack.

1. What is an imperative definition ?

An abstract data type is defined as a mathematical model of the data objects that make up a data type as well as the functions that operate on these objects. There are no standard conventions for defining them. A broad division may be drawn between "imperative" and "functional" definition styles.

Imperative-style definition: In the philosophy of imperative programming languages, an abstract data structure is conceived as an entity that is mutable - meaning that it may be in different states at different times. Some operations may change the state of the ADT. Therefore, the order in which operations are evaluated is important, and the same operation on the same entities may have different effects if executed at different times - just like the instructions of a computer, or the commands and procedures of an imperative language. To underscore this view, it is customary to say that the operations are executed or applied, rather than evaluated. The imperative style is often used when describing abstract algorithms.

Abstract variable: Imperative-style definitions of ADT often depend on the concept of an abstract variable, which may be regarded as the simplest non-trivial ADT. An abstract variable V is a mutable entity that admits two operations are `store`(*V*, *x*) and `fetch`(*V*). In it, `store`(*V*, *x*), where x is a value of unspecified nature, and `fetch`(*V*) that yields a value with the constraint that `fetch`(*V*) always returns the value x used in the most recent `store`(*V*, *x*) operation on the same variable *V*. As in so many programming languages, the operation `store`(*V*, *x*) is often written *V* $\leftarrow$ *x* or some similar notation, and `fetch`(*V*) is implied whenever a variable *V* is used in a context where a value is required.

- For example: *V ← V + 1 is commonly understood to be a shorthand for `store` (V, `fetch`(V) + 1).*

In this definition, it is implicitly assumed that storing a value into a variable U has no effect on the state of a distinct variable *V*. To make this assumption explicit, one could add the constraint that, if *U* and *V* are distinct variables, the sequence {`store` (*U*, *x*); `store` (*V*, *y*)} is equivalent to {`store` (*V*, *y*); `store` (*U*, *x*)}. More generally, ADT definitions often assume that any operation that changes the state of one ADT instance has no effect on the state of any other instance including other instances of the same ADT - unless the ADT axioms imply that the two instances are connected/aliased in that sense.

- For example: *When extending the definition of abstract variable to include abstract records, the operation that selects a field from a record variable R must yield a variable V that is aliased to that part of R*.

The definition of an abstract variable V may also restrict the stored values *x* to members of a specific set *X*, called the range or type of *V*. As in programming languages, such restrictions may simplify the description and analysis of algorithms, and improve their readability. Note that this definition does not imply anything about the result of evaluating *fetch* (*V*) when V is un-initialized, that is, before performing any store operation on *V*. An algorithm that does so is usually considered invalid, because its effect is not defined. However, there are some important algorithms whose efficiency strongly depends on the assumption that such a fetch is legal, and returns some arbitrary value in the variable's range.

<u>*Instance creation*</u>: Some algorithms need to create new instances of some ADT such as new variables, or new stacks. To describe such algorithms, one usually includes in the ADT definition a create() operation that yields an instance of the ADT, usually with axioms equivalent to the result of create () is distinct from any instance in use by the algorithm. This axiom may be strengthened to exclude also partial aliasing with other instances. In other words, this axiom still allows implementations of create () to yield a previously created instance that has become inaccessible to the program.

2. Using an imperative definition, specify the abstract data type for a software stack

<u>Example about imperative for abstract stack</u>: An imperative-style definition of an abstract stack could specify that the state of a stack *S* can be modified only by the operations push(*S*, *x*) and pop (*S*). In it, push (*S*, *x*), where x is some value of unspecified nature and pop (S) that yields a value as a result with the constraint for any value *x* and any abstract variable *V*, the sequence of operations {push (*S*, *x*); *V* ← pop(*S*) } is equivalent to *V* ← *x*. Since the assignment *V* ← *x*, by definition, cannot change the state of *S*, this condition implies that *V* ← pop (*S*) restores *S* to the state it had before the push (*S*, *x*). From this condition and from the properties of abstract variables it follows.

- <u>For example</u>: that the sequence { push (*S*, *x*); push (*S*, *y*); *U* ← pop (*S*); push (*S*, *z*); *V* ← pop (*S*); *W* ← pop (*S*) } where *x*, *y*, and *z* are any values, and *U*, *V*, *W* is pairwise distinct variables, is equivalent to { *U* ← *y*; *V* ← *z*; *W* ← *x* }.

Here it is implicitly assumed that operations on a stack instance do not modify the state of any other ADT instance including other stacks for any values *x*, *y*, and any distinct stacks *S* and *T*, the sequence {push (*S*, *x*); push (*T*, *y*)} is equivalent to { push (*T*, *y*); push (*S*, *x*)}. An abstract stack definition usually includes also a Boolean-valued function empty(*S*) and a create() operation that returns a stack instance with axioms equivalent to create() ≠ *S* for any prior stack *S* - a newly created stack is distinct from all previous stacks, `empty` (`create` ()) - a newly created stack is empty, `not empty`(`push` (*S*, *x*)) - pushing something into a stack makes it non-empty.

<u>A software stack:</u> Is a collection of independent components that work together to support the execution of an application. The components, which may include an operating system, architectural layers, protocols, run-time environments, databases and function calls, are stacked one on top of each other in a hierarchy. Typically, the lower level components in the hierarchy interact with hardware, while the higher-level components in the hierarchy perform specific tasks for the end user. Components communicate directly with the application through a series of complex instructions that traverse the stack.

<u>Three examples of software stacks</u>: LAMP, MEAN and Apache CloudStack.

- ✓ **LAMP (Linux, Apache, MySQL, PHP)** - *a well-known software stack for web development. The lowest layer of the stack's hierarchy is the Linux operating system. The highest layer of the hierarchy is the scripting language -- in this case, PHP. Besides, the "P" may also stand for the programming languages Python or Perl. LAMP stacks are popular because the components are all open source and the stack can run on commodity hardware. Unlike monolithic software stacks, which are often tightly coupled and often built for a particular operating system, a LAMP stack is loosely coupled. This simply means that while the components were not originally designed to work together, they have proven to be complementary and are often used together. Today, LAMP components are now included in almost all Linux distributions.*
- ✓ **MEAN (MongoDB, Express, Angular and Node)** - *a stack of development tools known for helping to eliminate the language barriers often experienced in software development. In a MEAN stack, the foundation is MongoDB, which is a NoSQL document datastore. The HTTP server is Express and Angular is the framework for front-end JavaScript. The highest layer of the stack is Node, a platform for server-side scripting.*
- ✓ **Apache CloudStack** - *an open source cloud management stack used by large enterprise customers and service providers who provide*

Infrastructure-as-a-Service (IaaS). CloudStack provides developers with multiple layers of optional services as well as support for a variety of hypervisors and application program interfaces (APIs).

M3 Examine the advantages of encapsulation and information hiding when using an ADT.

Data hiding and encapsulation both are the important concept of object-oriented programming. Encapsulation means wrapping the implementation of data member and methods inside a class. When implementation of all the data member and methods inside a class are encapsulated, the method name can only describe what action it can perform on an object of that class. Information hiding means that when someone uses that interface, they will still be able to use it without knowing any implementation details of the interface. Encapsulation and information hiding are usually come hand in hand. This is usually the case in OOP languages because of the availability of classes.

- For example: *In the Java language, when we are implementing a class, putting the data members and methods inside that class is essentially performing encapsulation. Furthermore, using specific keywords such as "private", "protected" means we are effectively performing information hiding. "Private" means the data member or method with this keyword will not available for access anywhere while "protected" means any inheritance of this class can see that data member or method, which is usually used for custom implementation or modification of the original class.*

In addition, there are major differences between the two this concept. The encapsulation also involves the way the class is hidden from users and how the process can be reversed by unlocking the capsule using methods. In data encapsulation, the capsule and the object inside it can be classified as either private or public, unless specified or programmed by the programmer. On the other hand, data hiding is the process of hiding the details of an object or function. It is also a potent technique in programming that results in data security and less data complexity. One of the manifestations of data hiding is that it is used as a method of hiding information inside a computer code after the code is broken down and hidden from the object. All objects in the state of data hiding are in isolated units, which is the main concept of object-oriented programming. The data inside is classified as private or non-accessible from other objects, classes, and API's in the system. The data appear as invisible to outsiders – whether objects, other classes, or users. Data encapsulation is one of the chief mechanisms of data hiding. Data hiding works by nesting the data or arranging it into capsules. Hiding the physical storage layout for data and avoiding linking to incorrect data (if a programmer does link to said data, the program will display an error to protect the content).

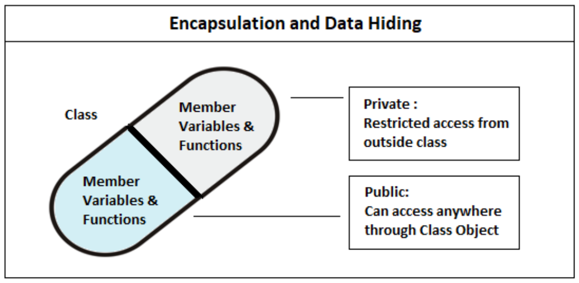

Encapsulation and Data Hiding
Class
Member Variables & Functions
Member Variables & Functions
Private : Restricted access from outside class
Public: Can access anywhere through Class Object

Figure 27: Encapsulation and Data Hiding

Table 6: Advantage of encapsulation and data hiding

Encapsulation	Data hiding
Improve the maintainability of the application	Restrict access to data from outside
Users can use the system in a very flexible way	Avoid the risk of data theft or information disclosure
Help application developers organize their coding better	Hiding data reduces some system complexity by limiting software interdependence
The packaging code is quite flexible and easily changes before new requirements	Data hiding is also a method to help protect from hackers
Help change a part of the code without affecting other parts of the code	Data hiding data is always private and inaccessible
Increased security and more convenient for application maintenance	
Abstraction hides the irrelevant details found in the code	
Simplify the representation of domain models	

D2 Discuss the view that imperative ADTs are a basis for object orientation and, with justification, state whether you agree.

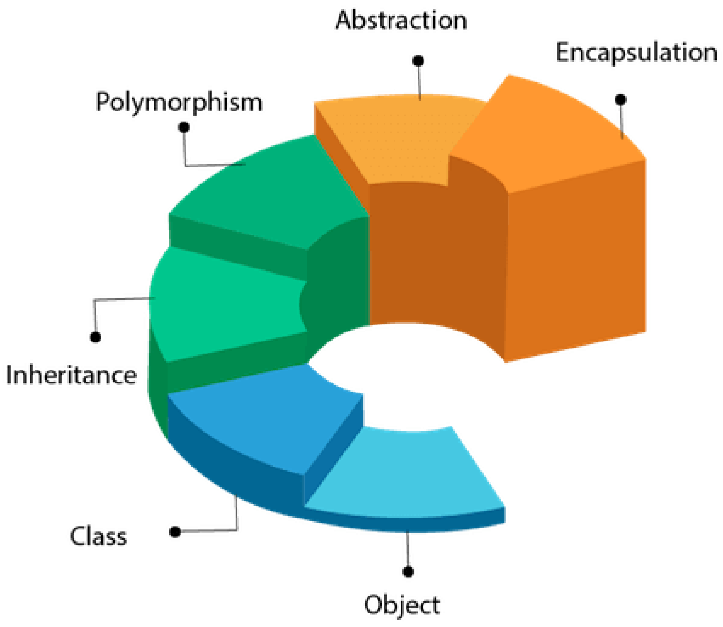


Figure 28: OOPs - Object Oriented Programming System

I disagree with the view that mandatory ADT is the basis for object orientation, although this concept influenced the design of many later languages to which object-oriented labels were attached. The basic ideas of OOP are encapsulation (local storage, state process protection and hiding), late binding in everything and the message.

According to Simula, an object is basically defined by the name of the messages it receives and the values it encapsulates. The behavior of those methods is subject to change. And you don't know what will actually be done in response to a message or a method call. While it is reasonable to expect a return value of a certain type, in Simula-style OOP, that can at least be guaranteed.

- *Simula is the name of two simulation programming languages, Simula I and Simula 67, developed in the 1960s at the Norwegian Computing Center in Oslo, by Ole-Johan Dahl and Kristen Nygaard. Syntactically, it is a fairly faithful superset of ALGOL 60 (Algorithmic Language 1960 was the first language implementing nested function definitions with lexical scope), it also influenced by the design of Simscript (is a free-form, English-like* *general-purpose simulation language conceived by Harry Markowitz and Bernard Hausner at the RAND Corporation in 1962). Simula 67 introduced objects, classes, inheritance and subclasses, virtual procedures, coroutines, and discrete event simulation, and features garbage collection. Also, other forms of subtyping (besides inheriting subclasses) were introduced in Simula derivatives. Simula is considered the first object-oriented programming language. As its name suggests, the first Simula version by 1962 was designed for doing simulations; Simula 67 though was designed to be a general-purpose programming language and provided the framework for many of the features of object-oriented languages today. Simula has been used in a wide range of applications such as simulating very-large-scale integration (VLSI) designs, process modeling, communication protocols, algorithms, and other applications such as typesetting, computer graphics, and education. The influence of Simula is often understated, and Simula-type objects are reimplemented in C++, Object Pascal, Java, C#, and many other languages. Computer scientists such as Bjarne Stroustrup, creator of C++, and James Gosling, creator of Java, have acknowledged Simula as a major influence.*

An abstract data type is determined by the values it can act on and the operations that can be performed on them. Inheritance is not part of the concept. Parametric polymorphism is required. Encapsulation is required but without the concept of an object containing its own activities, the semantics of the activities are an essential part of the definition of ADT and cannot be changed.

Java's design is influenced by CLU- a language created by one of the people who named ADT and their official definition is clearly designed to allow programming with ADT. Java's own methods, final methods, public metrics, and interfaces that allow you to effectively create things are ADT. It still doesn't give the concept of ADT differentiation - it arises from the choices you make in defining an abstract class or interface.

ASSIGNMENT 1 REFERENCES

[1] Adam Drozek, 2010. *Data Structures And Algorithms In Java*. 2nd ed. [ebook] Available at: <https://pdfs.semanticscholar.org/03fb/c6e2cd9ef11dfe000696849786b372fdd777.pdf?_ga=2.95085468.1647234384.1596762023-1381977231.1596762023> [Accessed 7 August 2020].

[2] En.wikipedia.org. 2020. *Abstract Data Type*. [online] Available at: <https://en.wikipedia.org/wiki/Abstract_data_type> [Accessed 15 August 2020].

[3] Margaret Rouse, 2018. *What Is Software Stack? - Definition From Whatis.Com*. [online] SearchAppArchitecture. Available at: <https://searchapparchitecture.techtarget.com/definition/software-stack> [Accessed 15 August 2020].

[4] Granville Barnett and Luca Del Tongo, 2008. *DSA Data Structures And Algorithms Annotated Reference With Examples*. 1st ed. [ebook] Available at: <https://www.academia.edu/30843807/DSA_Data_Structures_and_Algorithms_Annotated_Reference_with_Examples> [Accessed 15 August 2020].

[5] C.Thomas Wu, 2010. *An Introduction To Object-Oriented Programming With Javatm*. 5th ed. [ebook] Available at: <https://www.programming-book.com/an-introduction-to-object-oriented-programming-with-java-5th-edition/> [Accessed 15 August 2020].

[6] Code_r, andrew1234 and 29AjayKumar, 2020. *FIFO (First-In-First-Out) Approach In Programming - Geeksforgeeks*. [online] GeeksforGeeks. Available at: <https://www.geeksforgeeks.org/fifo-first-in-first-out-approach-in-programming/> [Accessed 16 August 2020].

[7] Tech Differences. 2016. *Difference Between Data Hiding And Encapsulation (With Comparison Chart) - Tech Differences*. [online] Available at: <https://techdifferences.com/difference-between-data-hiding-and-encapsulation.html> [Accessed 16 August 2020].

[8] Softwaretestinghelp.com. 2020. *How To Implement Dijkstra'S Algorithm In Java*. [online] Available at: <https://www.softwaretestinghelp.com/dijkstras-algorithm-in-java/> [Accessed 16 August 2020].

[9] GeeksforGeeks. 2020. *Bellman–Ford Algorithm | DP-23 - Geeksforgeeks*. [online] Available at: <https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/> [Accessed 16 August 2020].

[10] HackerEarth. 2020. *Shortest Path Algorithms Tutorials & Notes | Algorithms | Hackerearth*. [online] Available at: <https://www.hackerearth.com/fr/practice/algorithms/graphs/shortest-

path-algorithms/tutorial/> [Accessed 16 August 2020].

[11] Bjarne Stroustrup, 2018. The C++ Programming Language. Upper Saddle River [i pozostałe]: Pearson.

[12] Drozdek, A., 2012. Data Structures And Algorithms In C++.

[13] Insertion Algorithms, 2020. *Insertion Sort Vs Bubble Sort Algorithms*. [online] Stack Overflow. Available at: <https://stackoverflow.com/questions/17270628/insertion-sort-vs-bubble-sort-algorithms#:~:text=In%20insertion%20sort%20elements%20are,out%20of%20the%20unsorted%20section.> [Accessed 18 August 2020].

[14] Cpp.thiyagaraaj.com. 2020. Data Hiding And Encapsulation Using Access Specifiers In C++ - C++ Programming Concepts. [online] Available at: <https://www.cpp.thiyagaraaj.com/home/blog-1/datahidingandencapsulation> [Accessed 18 August 2020].

[15] Medium. 2020. Undestanding Encapsulation And Information Hiding. [online] Available at: <https://medium.com/@scottc130/undestanding-encapsulation-and-information-hiding-914a11d89b6f> [Accessed 17 August 2020].

[16] Differencebetween.net. 2020. *Difference Between Data Hiding And Data Encapsulation | Difference Between*. [online] Available at: <http://www.differencebetween.net/technology/software-technology/difference-between-data-hiding-and-data-encapsulation/> [Accessed 18 August 2020].

[17] Bruce Richardson and Mark Miller, 2018. *Is It True That Abstract Data Types (ADT) Is One Of The Basic Ideas Behind Object Orientation? - Quora*. [online] Quora.com. Available at: <https://www.quora.com/Is-it-true-that-abstract-data-types-ADT-is-one-of-the-basic-ideas-behind-object-orientation> [Accessed 18 August 2020].

[18] Bowdoin.edu. 2020. *Mergesort And Recurrences*. [online] Available at: <http://www.bowdoin.edu/~ltoma/teaching/cs231/fall14/Lectures/02-recurrences/recurrences.pdf> [Accessed 21 August 2020].