# ASSIGNMENT 1 FRONT SHEET

| | | | |
|---|---|---|---|
| **Qualification** | **BTEC Level 5 HND Diploma in Computing** | | |
| **Unit number and title** | Unit 19: Data Structures and Algorithms | | |
| **Submission date** | 24/04/2021 | **Date Received 1st submission** | |
| **Re-submission Date** | | **Date Received 2nd submission** | |
| **Student Name** | PHAM CAO NGUYEN | **Student ID** | TRAN THI KIM KHANH |
| **Class** | GCC0801 | **Assessor name** | GCC18074 |
| **Student declaration**<br><br>I certify that the assignment submission is entirely my own work and I fully understand the consequences of plagiarism. I understand that making a false declaration is a form of malpractice. | | | |
| | | **Student's signature** | CAONGUYEN |

**Grading grid**

| P1 | P2 | P3 | M1 | M2 | M3 | D1 | D2 |
|---|---|---|---|---|---|---|---|
| | | | | | | | |

| | | |
|---|---|---|
| ❒ **Summative Feedback:** | | ❒ **Resubmission Feedback:** |
| **Grade:** | **Assessor Signature:** | **Date:** |
| **Internal Verifier's Comments:** | | |
| **IV Signature:** | | |

# 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** | | | |
| **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.<br>*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/<br>*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 Softnet 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.

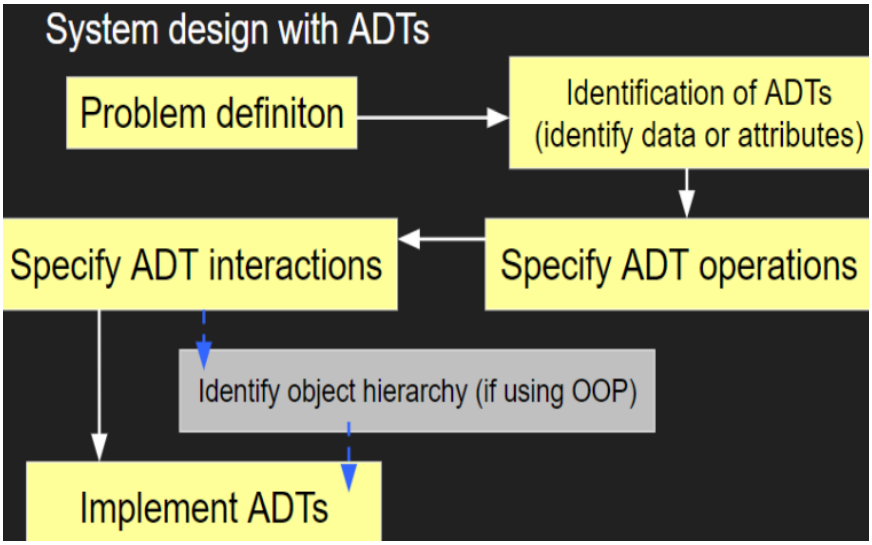
| Learning Outcomes and Assessment Criteria | | |
|---|---|---|
| **Pass** | **Merit** | **Distinction** |
| **LO1** Examine abstract data types, concrete data structures and algorithms | | **D1** Analyse the operation, using illustrations, of two network shortest path algorithms, providing an example of each. |
| **P1** Create a design specification for data structures explaining the valid operations that can be carried out on the structures.<br><br>**P2** Determine the operations of a memory stack and how it | **M1** Illustrate, with an example, a concrete data structure for a First In First out (FIFO) queue.<br>**M2** Compare the performance of two sorting algorithms. | |

<table>
<tr><td>is used to implement function calls in a computer.</td><td></td><td></td></tr>
<tr><td colspan="2">LO2 Specify abstract data types and algorithms in a formal notation</td><td rowspan="2">D2 Discuss the view that imperative ADTs are a basis for object orientation and, with justification, state whether you agree.</td></tr>
<tr><td>P3 Using an imperative definition, specify the abstract data type for a software stack.</td><td>M3 Examine the advantages of encapsulation and information hiding when using an ADT.</td></tr>
</table>

# Contents

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

1. **What is ADT? [1]**
   - An Abstract Data Type (ADT) is the specification of a data type within some programming language, independent of an implementation. The interface for 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. (Sedgewick, 2002)
   - In computer science, an abstract data type (ADT) is a mathematical model for data types. An abstract data type is determined by its behavior (semantics) from the user's point of view, of the data, namely about the possible values, the possible operations on the data belonging to this type, and the behavior of these operations. This mathematical model contrasts with the data structure in specific representations of the data and is the point of view of the implementer, not the user.
   - A data structure is an implementation for an ADT. In an object-oriented language like Java, an ADT and its implementation together make up a class. Each operation associated with the 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.
   - Concept of "Abstraction": This allows us to consider the high-level characteristics of something without getting bogged down in the details.
   - An ADT consists of:
     - Declaration of data.
     - Declaration of operations.
     - Encapsulation of data and operations: data is hidden from the user and can be manipulated only by means of operations.
2. **How to implement ADT? [2]**
   - There can be different ways to implement an ADT, for example, the List ADT can be implemented using arrays, or singly linked list or doubly linked list. Similarly, stack ADT and Queue ADT can be implemented using arrays or linked lists.
   - ADTs support modular design which is very important in software development




- Designing an abstract type involves choosing good operations and determining how they should behave. Here are a few rules of thumb.
- It's better to have a few, simple operations that can be combined in powerful ways, rather than lots of complex operations.
- Each operation should have a well-defined purpose, and should have a coherent behavior rather than a panoply of special cases. We probably shouldn't add a sum operation to List, for example. It might help clients who work with lists of Integers, but what about lists of Strings? Or nested lists? All these special cases would make sum a hard operation to understand and use.
- The set of operations should be adequate in the sense that there must be enough to do the kinds of computations clients are likely to want to do. A good test is to check that every property of an object of the type can be extracted. For example, if there were no get operation, we would not be able to find out what the elements of a list are. Basic information should not be inordinately difficult to obtain. For example, the size method is not strictly necessary for List, because we could apply get on increasing indices until we get a failure, but this is inefficient and inconvenient.
- The type may be generic: a list or a set, or a graph, for example. Or it may be domain-specific: a street map, an employee database, a phone book, etc. But it should not mix generic and domain-specific features.

**3. Operations of ADT [3]**

➢ **The operations of an abstract data type are classified as follows:**

- Creators create new objects of the type. A creator may take an object as an argument, but not an object of the type being constructed.
- Producers create new objects from old objects of the type. The concat method of String, for example, is a producer. It takes two strings and produces a new one representing their concatenation.
- Observers take objects of the abstract type and return objects of a different type. The size method of List, for example, returns an int.

- Mutators change objects. The add method of List, for example, mutates a list by adding an element to the end.

**4. ADTs in Java**

- Java library has Abstract Data Types such as List, Stack, Queue, Set, Map as inbuilt interfaces which are being implemented using various data structures.
- In Java, Abstract Data Types extend the Collections Interface which represents the data type. It is part of the Java Collections framework and is the root interface in the collection hierarchy. A collection represents a group of objects, known as its elements.
- The JDK does not provide any direct implementations of this interface. It provides implementations of more specific sub interfaces like List, Set. This interface is typically used to pass collections around and manipulate them where maximum generality is desired.

**5. List ADT [4]**

- List Abstract Data Type is a collection of elements that have a linear relationship with each other. A linear relationship means that, except for the first one, each element on the list has a unique successor. Also, lists have a property intuitively called size, which is simply the number of elements on the list.
- List is mutable. List is also an interface, which means that other classes provide the actual implementation of the data type. These classes include ArrayList which is implemented internally using Arrays and LinkedList which is implemented internally using LinkedList data structure.
- get (int index) – Returns an element at a particular index from the list.
- add (E e) – Appends the specified element to the end of this list.
- remove (Object o) – Remove the first occurrence of the specified element from the list.
- remove (int index) – Removes the element at the specified index from the list.
- size () – Returns number of elements of the list.
- isEmpty () – Return true if the list is empty, else return false.

**6. Stack ADT [5]**

- Stack ADT is a collection with homogeneous data items (elements), in which all insertions and deletions occur at one end, called the top of the stack. A stack is a LIFO "Last In, First Out" structure. Analogy to Stack is a stack of plates.
- Stacks are managed mainly using two functions like below
    - PUSH – places an element on top of the stack.
    - POP – removes an element from the stack.
- In Java, Stack ADT class extends the Vector class which is a growable array of Objects and it can be accessed using integer index
- Java library provides the below following operations which can be performed on
    - push (E e) – Inserts an element at the top of stack.
    - pop () – Removes an element from the top of the stack if it is not empty.
    - peek () – Returns the top element of stack without removing it.
    - size () – Returns the size of the stack.

- isEmpty () – Returns true if the stack is empty, else it returns false.

**7. Queue ADT [6]**

- Queue ADT is a collection in which the elements of the same type are arranged sequentially. The operations can be performed at both ends with insertion being done at rear end deletion being done at the front end for a single-ended queue. Theoretically, Queue is a FIFO "First In, First Out" structure.
- add(E e) – Queues an element at the end of queue.
- remove() – Dequeues an element from the head of the queue.
- peek() – Returns the element of the queue without removing it.
- offer(E e) – Inserts the specified element into this queue if it is possible to do without violating capacity restrictions.
- size() – Returns the size of the queue.

**8. Which Java ADT to choose**

- Since Stack is a Last In First data structure, the implementations of Stack ADT need to be chosen in scenarios where the most recently inserted elements need to be accessible first. One of the common example where this kind of LIFO data structure is required is the function call stack of every programming language where the most recent function in the stack needs to be executed first.
- Queue is a First In First Out structure and data structures implementing Queue ADT need to be chosen in scenarios where the elements need to be accessed in their order of insertion i.e where fairness needs to be ensured. Example of one such scenario is request handling by web servers. Web servers facilitate fairness of request handling according to their order of arrival by maintaining an internal queue for the requests.

**9. Deployment**

- A data structure is the implementation for an ADT. In an object-oriented language like Java, an ADT and its implementation together make up a class. Each operation associated with the 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.

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

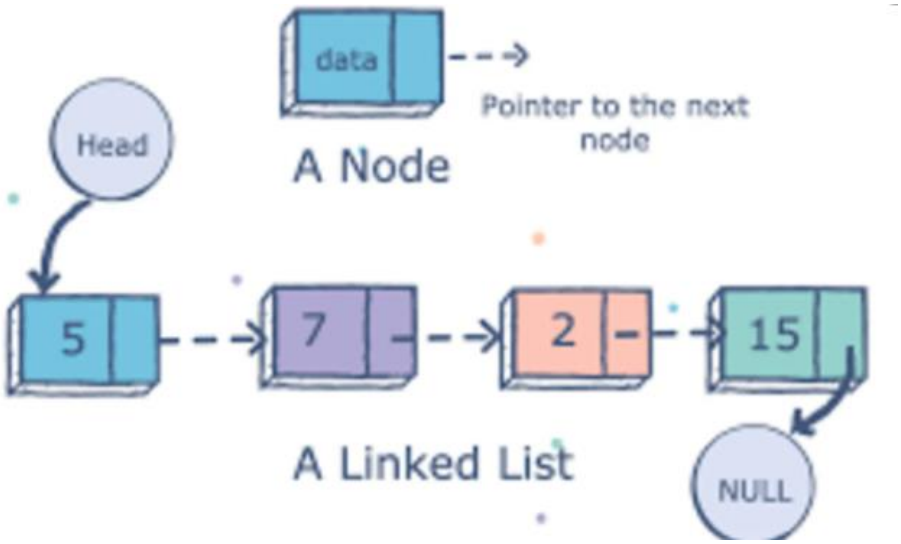
➢ **Let us see some operations of those mentioned ADT:**

- Stack
  - isFull(), This is used to check whether stack is full or not
  - isEmpry(), This is used to check whether stack is empty or not
  - push(x), This is used to push x into the stack
  - pop(), This is used to delete one element from top of the stack
  - peek(), This is used to get the top most element of the stack
  - size(), this function is used to get number of elements present into the stack
- Queue
  - isFull(), This is used to check whether queue is full or not

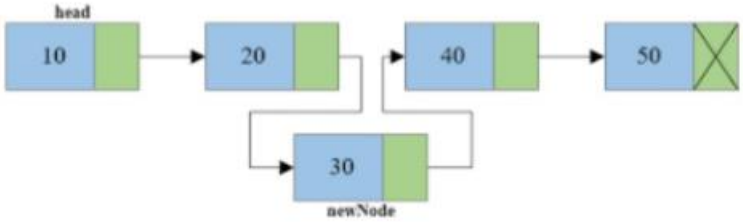
  - isEmpry(), This is used to check whether queue is empty or not
  - insert(x), This is used to add x into the queue at the rear end
  - delete(), This is used to delete one element from the front end of the queue
  - size(), this function is used to get number of elements present into the queue
- List
  - size(), this function is used to get number of elements present into the list
  - insert(x), this function is used to insert one element into the list
  - remove(x), this function is used to remove given element from the list
  - get(i), this function is used to get element at position i
  - replace(x, y), this function is used to replace x with y value

**11. How to implement the operations?**

- **A Linked List** is an ADT (Abstract Data Type) that consists of a series of nodes that are related in some way. A linked list is used to keep track of a changing set of data. Each node of a linked list basically contains only two parts data part and the address part. The data portion of the node contains the actual data, while the address portion contains the connection or address to the node to which it is attached next. Since each list (node) is linked to another list, the linked list gets its name (node).




- A linked list is composed mainly of nodes and each node contains the address of its next connected node
- Type of linked lists: Singly-linked list and Doubly-linked list. **[7]**
- **A singly linked list** is the most basic kind of linked list. A singly linked list is a set of nodes connected in a sequential manner, with each node containing a data field and an address field containing the connection to the next node. A singly linked list can have several data fields, but it must have at least one address field leading to the next node in the chain.




- To execute any operation on a linked list, we must keep track of the first node, which can be referred to using the head pointer attribute. In a singly connected list, the last node's address field must contain a NULL value indicating the list's end.
- The basic structure of a singly linked list: A singly connected list's nodes all have the same fundamental structure. We can store several data fields in a node, but we must have at least one address field to store the address of the next connected node. **[8]**

```
public class node {
    public int info;
    public node next;

    public node(int info, node next) {
        this.info = info;
        this.next = next;
    }
    public node(int info) {
        this(info,null);
    }
}
```

```
public class LinkedList {
    public node head, tail;
    public LinkedList(){
        head = tail = null;
    }
    public boolean isEmpty(){
        return (head==null);
    }
    public void add(int a){
        node n = new node(a);
        if(isEmpty()){
            head = tail = n;
        }else{
            tail.next = n;
            tail = n;
        }
    }
    public void addMany(int[] a){
        for(int item: a){
            add(item);
        }
    }
```

```
    public void traverse(){
        node p = head;
        while(p!=null){
            System.out.println(p.info);
            p = p.next;//p= p.getNext();
        }
    }
    public void deleteLast(){
        if(head.next==null){
            head = tail = null;
            System.out.println("Empty List");
        }
        else{
            node temp = head;
            while(temp.next.next!=null){
                temp = temp.next;
            }
            temp.next = null;
            tail = temp;
        }
    }

    public static void main(String[] args){
        int[] a = {9,8,-11,3,6,24};
        LinkedList list = new LinkedList();
        list.addMany(a);
        list.traverse();
        System.out.println("After add last:");
        list.add(7);
        list.traverse();
        System.out.println("After delete last:");
        list.deleteLast();
        list.traverse();
    }
}


"C:\Program Files\Java\jdk-15.0.2\bin\java.exe" "-javaagent:C:\Program
Files\JetBrains\IntelliJ IDEA Community Edition
2020.3.3\lib\idea_rt.jar=63657:C:\Program Files\JetBrains\IntelliJ IDEA Community
Edition 2020.3.3\bin" -Dfile.encoding=UTF-8 -classpath
C:\Users\lambe\IdeaProjects\Java1\out\production\Java1 LinkedList
9
8
-11
3
6
24
After add last:
9
8
-11
3
6
24
7
```

```
After delete last:
9
8
-11
3
6

Process finished with exit code 0
```

- **addMany**
  - For the function that adds multiple elements to the end of the list (addMany), it accepts the array a = int. Run a for loop with the value item = a, adding a to the end of the list for all item values.
- **isEmpty**
  - Build a function LinkedList with two variables head and tail, where head and tail is the two variable null.
  - Constructs the isEmpty function to check if an element already exists in the list and it will end the function with a null head value.
- **Add**
  - For the function that add an element to the end of the list (add), it takes the value an of int where the node is a new Node = a. When the list is empty, head and tail = n, otherwise tail = n.
- **Delete Last**
  - It first checks whether the head is null (empty list) then, it will return from the function as there is no node present in the list
  - If the list is not empty, it will check whether list has only one node
  - If the list has only one node, it will set both head and tail to null.
  - Now, current will point to the node previous to tail. Make current as new tail and tail will point to head thus, deletes the node from last.
- **Add Last**
  - Create another class LinkedList which has two attributes: head and tail.
  - addLast() will add a new node to the Last of the list:
  - It first checks, whether the last is equal to null which means the list is empty.
  - If the list is empty, both last and tail will point to a newly added node.
  - If the list is not empty then, create temporary node temp will point to last.
  - Add the new node as the last of the list.

```java
public class node {
    public int info;
    public node next;

    public node(int info, node next) {
        this.info = info;
        this.next = next;
    }
    public node(int info) { this(info, next: null); }
}
```

```java
  public void addLast(int d){
      if(!isEmpty()){
           newNode temp =
       new newNode(d,  next: null, tail);
        tail.setNext(temp);
        tail = temp;
       }
       else {
          head = tail = new newNode(d);
       }
   }
```

```java
    public static void main(String[] args){
        int[] a = {9,8,-11,3,6,24};
        LinkedList list = new LinkedList();
```

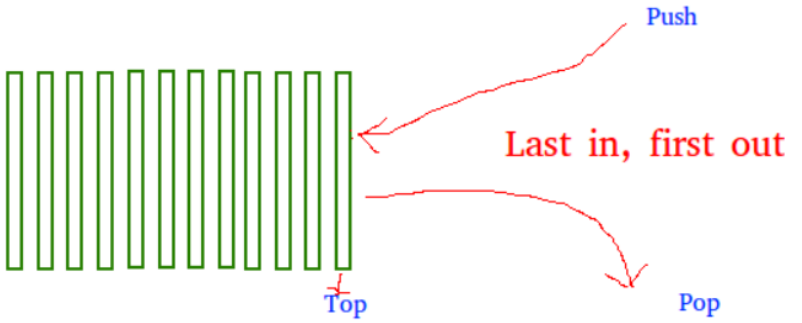
```java
     System.out.println("After add last:");
     list.add(7);
```

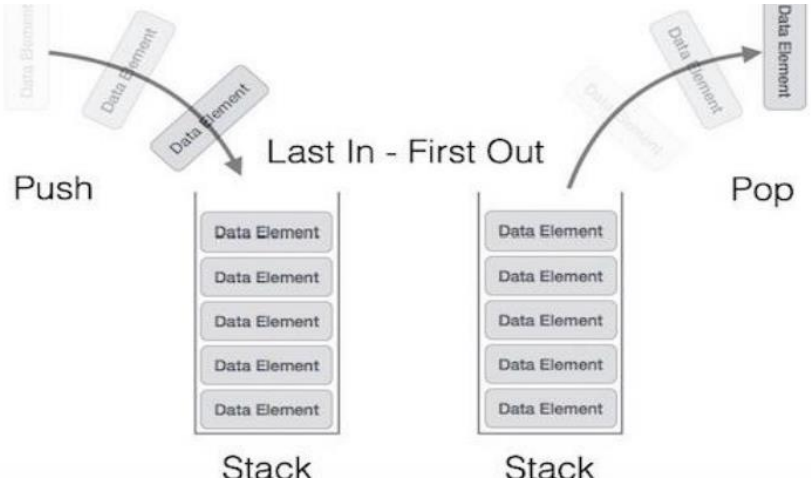
```
After add last:
9
8
-11
3
6
24
7
```

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

1. **What is a stack? [9]**

- Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO (Last In First Out) or FILO (First In Last Out).
- A stack is an abstract data form in computer science that acts as a set of elements with two major operations:
    - push, which adds an element to the collection
    - pop, which removes the most recently added element that was not yet removed




- The order in which elements come off a stack gives rise to its alternative name, LIFO (last in, first out). A peek operation can also provide access to the top of the stack without altering it. This style of arrangement is called a "stack" because it resembles a stack of physical objects piled on top of each other. This arrangement makes it simple to remove an item from the top of the stack while removing an item from the bottom of the stack will necessitate removing several other items first.

**2. What are operations on the stack? [10]**

- The stack primarily performs the following simple operations:
    - 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
- Stack exceptions

- Operations pop and top cannot be performed if the stack is empty
- Operations push sometimes cannot be performed if it's not enough memory

**3. How to implement stacks? [11]**

- There are two ways to implement a stack:
  - Using array
    - The stack can be implemented from the linked list, but stack using an array is the easiest way to understand and implement. I will write code to implement a stack using a linked list in the upcoming article.
  - Using linked list
    - A stack can be easily implemented through the linked list. In stack Implementation, a stack contains a top pointer. which is "head" of the stack where pushing and popping items happens at the head of the list. first node have null in link field and second node link have first node address in link field and so on and last node address in "top" pointer.

```
package stackPK;

public interface AbstractStack<E> {
    public boolean isEmpty();
    public void push(E element);
    public E pop();
    public E peek();
    public int getSize();
    public void traverse();

}
```

```
package stackPK;

import home.LinkedListPK.Nodeint;
import home.LinkedListPK.SinglyListint;

public class StackInt implements AbstractStack<Nodeint> {
    private int size;
    private Nodeint top;
    private SinglyListint list = new SinglyListint();

    public StackInt(){
        this.top = null;
        this.size = 0;
    }

    @Override
    public boolean isEmpty() {
        return this.top == null;
        //return this.size == 0;
        //return list.isEmpty();
    }

    @Override
```

```
    public void push(Nodeint element) {
        list.addFirst(element);
        this.top = list.getHead();
        this.size++;
    }

    @Override
    public Nodeint pop() {
        if(isEmpty())
            return null;
        //throw new emtyStackException();
        else{
            Nodeint old = this.top;//old top
            list.deleteFirst();
            this.size--;
            this.top = list.getHead();
            return old;
        }
    }

    @Override
    public Nodeint peek() {
        if(isEmpty())
        return null;
        else
            return this.top;
    }

    @Override
    public int getSize() {
        return this.size;
    }

    @Override
    public void traverse() {
        if(isEmpty())
            System.out.println("Stack is empty");
        else
            list.traverse();

    }
}
```

```
package home.LinkedListPK;

public class Nodeint {
    private Integer info;
    private Nodeint next;

    public Nodeint(Integer info) {
        this(info,null);
    }

    public Nodeint(Integer info, Nodeint next) {
        this.info = info;
```

```
        this.next = next;
    }

    public Integer getInfo() {
        return info;
    }

    public void setInfo(Integer info) {
        this.info = info;
    }

    public Nodeint getNext() {
        return next;
    }

    public void setNext(Nodeint next) {
        this.next = next;
    }
}
```

```
package home.LinkedListPK;

public class SinglyListint extends SinglyList<Nodeint>{

    @Override
    public void addFirst(Nodeint element) {
        if(!isEmpty()){
            element.setNext(super.getHead());
        }
        super.setHead(element);
    }

    @Override
    public void addLast(Nodeint element) {
        if(isEmpty()){
            super.setHead(element);
            super.setTail(element);
        }else{
            Nodeint tail = super.getTail();//old tail
            tail .setNext(element);//add new temp tail
            super.setTail(element);//
        }
    }

    @Override
    public void deleteFirst() {
        Nodeint temp = super.getHead();
        super.setHead(temp.getNext());
        System.out.println("Deleting "+temp.getInfo());
    }

    @Override
    public void deleteLast() {
        Nodeint temph = super.getHead();
        if (temph.getNext()==null){
            super.setHead(null);
```

```
            System.out.println("Empty List");
        }else {
            while (temph.getNext().getNext() != null) {
                temph = temph.getNext();
            }
            temph.setNext(null);
        }
    }

    @Override
    public void traverse() {
        Nodeint temp = super.getHead();
        while (temp!=null){
            System.out.println(temp.getInfo());
            temp = temp.getNext();
        }
    }
}
```

```
package stackPK;

import home.LinkedListPK.Nodeint;

import stackPK.StackInt;

import java.util.Scanner;

public class TestStack {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String line = "";
        StackInt stack = new StackInt();

        while(!line.equals("end")){
            System.out.print("inter your node");
            Nodeint n = new Nodeint(Integer.parseInt(sc.nextLine()));
            stack.push(n);
            line = sc.nextLine();
        }
        System.out.println("Stack is:");
        stack.traverse();

        System.out.println("Top of Stack"+stack.peek().getInfo());

        System.out.println("Pop 3 nodes");
        System.out.println("Popping node of"+stack.pop().getInfo());
        System.out.println("Popping node of"+stack.pop().getInfo());
        System.out.println("Popping node of"+stack.pop().getInfo());

        System.out.println("Top of Stack"+stack.peek().getInfo());
    }
}
```

```
"C:\Program Files\Java\jdk-15.0.2\bin\java.exe" "-javaagent:C:\Program
Files\JetBrains\IntelliJ IDEA Community Edition
2020.3.3\lib\idea_rt.jar=57521:C:\Program Files\JetBrains\IntelliJ IDEA Community
Edition 2020.3.3\bin" -Dfile.encoding=UTF-8 -classpath
C:\Users\lambe\IdeaProjects\Java1\out\production\Java1 stackPK.TestStack
inter your node6

inter your node7

inter your node8

inter your node9
end
Stack is:
9
8
7
6
Top of Stack9
Pop 3 nodes
Deleting 9
Popping node of9
Deleting 8
Popping node of8
Deleting 7
Popping node of7

Process finished with exit code 0
```

- Creates size object with int value, on top with Nodeint value and SingListint named “list”.
- For the StackInt function there is top = null and size = 0.
- For the isEmpty function will return top = null.
- The feature that adds the item to the stack (push) gets the entity value. With the values element top = head, size = size + 1, call the function to add an element to the top of the stack.
- To remove the object from the stack, use this feature (pop). The stack returns null if isEmpty = null. Establish a temporary value old = top, then call the function to delete the stack’s first object, size = size -1, top = head, and return the old value.
- The method verifies that the stack’s first item is still there (peek). Tests if isEmpty = null, the stack returns null. Otherwise, the stack returns value top.
- In order to use the stack traverse feature (traverse). If the stack is empty, print “Stack is empty” on the screen; otherwise, call the traverse method from SinglyList.
- For the function that checks size of stacks (getSize), when the function is called, it will return the size of the stack.
- Main() is called when the program is run, and a new stack is generated. Run the line and execute the line “Insert your node” that is printed to the screen, which will create a new node n that will

collect the numeric value from the keyboard. Add a new button to the stack and enter the desired characters in the row, once the "end" appears in the row, the loop is over.

- Print to the screen "Stack is:" then call the stack's traverse function. "Top of Stack:" print to the computer, then call the feature to get the value of the top of the stack and show it. "Pop 2 nodes" is printed to the screen, and it calls the operation that deletes values from the stack and displays them. "Stack is:" is printed to the computer, followed by a call to the stack's traverse feature.

**4. Applications of stacks [12]**

- Any sort of nesting (such as parentheses)
- Evaluating arithmetic expressions (and other sorts of expression)
- Implementing function or method calls
- Keeping track of previous choices (as in backtracking)
- Keeping track of choices yet to be made (as in creating a maze)
- Undo sequence in a text editor
- Auxiliary data structure for algorithms
- Component of other data structure

**5. How do method calls function and what are they? [13]**

- A Java method is a collection of statements that are grouped together to perform an operation.
- For using a method, it should be called. There are two ways in which a method is called i.e., method returns a value or returning nothing (no return value).
- The process of method calling is simple. When a program calls a process, the program control gets transferred to the called method. In two cases, the named process restores power to the sender.
    - The return statement is executed.
    - It reaches the method ending closing brace.
- How does recursion actually work?
    - Each time a method is called, an activation record (AR) is allocated for it.
    - This record contains the following information
        - Parameters and local variables used in the called method
        - Dynamic-link: a pointer to the caller's activation record
        - Return address to resume control by the caller (address of the instruction immediately following the call)
        - For a process that isn't declared as void, this is the return value. (Because the size of AR varies from call to call, the returned value is set directly above the caller's AR.)
- Each new AR is placed on top of the run-time stack)
- The AR of a process is deleted from the top of the run-time stack as it terminates.
- As a result, the first AR on the stack is the last one to be replaced.

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

1. **Imperative definition in ADT? [14]**
   - 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.
   - An abstract data structure is conceived as a mutable object in the theory of imperative programming languages, meaning that it can be in different states at different times. Some operations can alter the state of the ADT; as a result, the order in which operations are evaluated is critical, and the same operation on the same entities can have different results if performed at various times, just like computer instructions or imperative language commands and procedures. To emphasize this point, the operations are typically described as being executed or applied rather than being evaluated. When discussing complex algorithms, the imperative form is often used. (For more information, see Donald Knuth's The Art of Computer Programming.)
   - Imperative-style definitions of ADT often depend on the concept of an abstract variable, which may be regarded as the simplest non-trivial ADT. (Cowling, 2001)
2. **What is Software Stack? [15]**
   - A software stack is a group of programs that work in tandem to produce a result or achieve a common goal. Software stack also refers to any set of applications that works in a specific and defined order toward a common goal, or any group of utilities or routine applications that work as a set. Installable files, software definitions of products and patches can be included in a software stack. One of popular Linux-based software stack is LAMP (Linux, Apache, MYSQL, Perl or PHP or Python). WINS (Windows Server, Internet Explorer, .NET, SQL Server) is a popular Windows-based software stack.
   - Software stacks have many benefits:
     - They provide predefined solutions to problems and at times are the best solutions.
     - They provide the minimum software needed to achieve the intended goals.
     - Software stacks can be installed on individual systems or added to computer templates for automatic installation.
     - Software stack installation and functioning is the same for the same configurable systems. As such, the solutions provided are also consistent.
     - Most of the software stacks come with support for the entire package. Some also have community forums.
   - There are many types of technology stacks. Some are constructed to deliver a technology product or service, while others support a business purpose. Examples include:
     - a server stack, which includes the hardware, operating system and supporting software such as runtime environments, database software, and software for various web services;
     - a storage stack, which includes servers and server software, virtualization and networking components;

  - a cloud infrastructure stack, which includes abstracted infrastructure (physical and virtualized hardware resources), platform infrastructure (application servers and databases), applications, and services. A vast array of more granular services and functions can be added throughout this stack, such as machine learning, security and monitoring; and
  - a marketing stack, which includes an email platform, a content management system, customer relationship management software, and analytics and tracking tools.
- There can be different ways to implement an ADT, for example, the List ADT can be implemented using arrays, or singly linked list, or a doubly-linked list. Similarly, stack ADT and Queue ADT can be implemented using arrays or linked lists.
- Do they implement ADT?
  - Representation invariant: Object → Boolean
    - Indicates whether a data structure is well-formed
    - Only well-formed representations are meaningful
    - Defines the set of valid values of the data structure
  - Abstraction function: Object → abstract value
    - What the data structure means (as an abstract value)
    - How the data structure is to be interpreted
    - How do you compute the inverse, abstract value → Object?

# References

[1] Sedgewick, R., 2002. *Algorithms in Java, Parts 1-4*. Addison-Wesley Professional.

[2] http://web.mit.edu/6.005/www/fa14/classes/08-abstract-data-types/

[3] , [4], [7], [8], [11] Goodrich, M.T., Tamassia, R. and Goldwasser, M.H., 2014. *Data structures and algorithms in Java*. John Wiley & Sons.

[5] Carrano, F.M. and Savitch, W.J., 2003. *Data structures and abstractions with Java*. Upper Saddle River, NJ, USA: Prentice Hall.

[6] Dale, N.B., Joyce, D.T. and Weems, C., 2002. *Object-oriented data structures using Java*. Jones & Bartlett Learning.

[9], [12] Lafore, R., 2017. *Data structures and algorithms in Java*. Sams publishing.

[10] Meng, M., Meinicke, J., Wong, C.P., Walkingshaw, E. and Kästner, C., 2017, February. A choice of variational stacks: exploring variational data structures. In *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems* (pp. 28-35).

[13] Urma, R.G., Fusco, M. and Mycroft, A., 2014. *Java 8 in action*. Manning publications.

[14] Cowling, A.J., 2001, February. Teaching data structures and algorithms in a software engineering degree: Some experience with java. In *Proceedings 14th Conference on Software Engineering Education and Training.'In search of a software engineering profession'(Cat. No. PR01059)* (pp. 247-257). IEEE.

[15] Weiss, M.A., 1995. *Data structures and algorithm analysis*. Benjamin-Cummings Publishing Co., Inc…