Name: John Patrick F. Narvasa **DSA Assignment 1**
Year and Course: 1 BSCS2

Data Structure Types

Data Structures - A way of organizing a collection of data in order to process or handle information in a program effectively and efficiently.

Overview of the data structure types:

A. Primitive Data Structure

- Integer
- Float
- Character
- Boolean

B. Non Primitive Data Structure

- Linear
    - Array
    - Linked List
    - Stacks
    - Queues
- Non-Linear
    - Trees
    - Graphs
    - Hash Table

A. Primitive Data Structure -

A primitive data structure is a **basic type of data** that is already built-in to most programming languages. It can **only store one value** only on one data type. Examples of primitive data structures are integer, float, character and boolean.

1. Integer -

A data type that stores whole numbers. In the Java programming language it can store numbers from -2,147,483,648 to 2,147,483,647.

Syntax/ initialization:

```
int variableName = value;
```

Sample Code:

```
//Example:
int age = 18;
```

2. Float - A data type that stores decimal values.

Syntax/ initialization:

```
float variableName = value;
```

Sample Code:

```
//Example:
float price =3.99f;
```

3. Character - A data type that only stores a single character. It is declared using single quotation marks (‘ ’).

Syntax/ initialization:

```
char variableName = 'c’;
```

Sample Code:

```
//Example:
char Grade = ‘A’;
```

4. Boolean - A data type that stores truth values such as true or false. It is typically used in testing conditions.

Syntax/ initialization:

```
boolean variableName = true; or
boolean variableName = false;
```

Sample Code:

```
//Example:
boolean isActive = true;
```

B. Non Primitive Data Structure - A data structure that you can **store multiple values.** There are two types of non-primitive data structure: **linear** and **non-linear data structures.**

1. Linear data structure - A data structure that is being stored in a continuous manner. Examples of this data structures are arrays and linked lists

1.1. Array - A collection of data of the same type that is being **stored in a contiguous manner**. Each item inside the array is called an element. Can be one dimensional or multidimensional.

Syntax/ initialization:

```
data type[] arrayName = new type[size]; or
data type[] arrayName = {element1, element2, …};
```

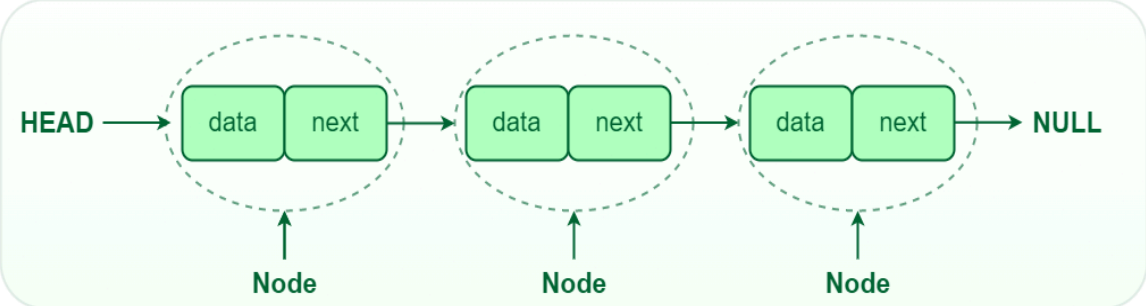
Sample Code:

```
int[] numbers = {1, 2, 3, 4, 5};
for (int n : numbers) {
        System.out.print(n + " ");
}
```

Output:
1 2 3 4 5

1.2. Linked List - A linear data structure where elements are **not stored** in an **adjacent manner**. Instead, the data is **stored using pointers** which form a connection of a series of nodes.

Sample illustration of a linked list:



Syntax/ initialization:

```
LinkedList<dataType> variableName = new LinkedList<dataType>();
```

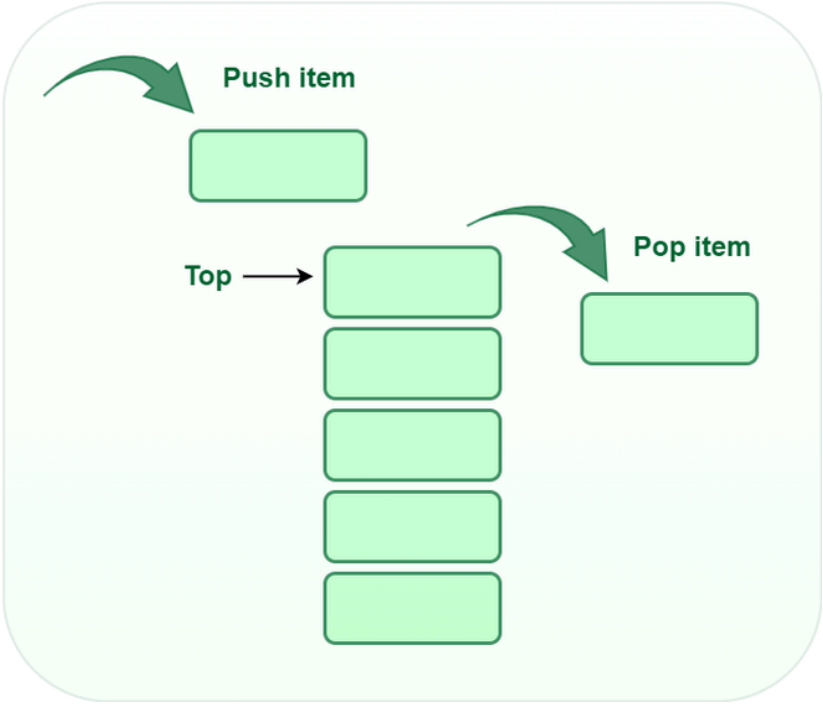
Sample Code (in Java):

```
import java.util.LinkedList;

public class ToDo {
  public static void main(String[] args) {
    LinkedList<String> tasks = new LinkedList<>();
    tasks.add("Complete DSA assignment");
    tasks.add("Eat Lunch");
    tasks.add("Study for IP");

    for (String task : tasks) {
            System.out.println(task);
        }
  }
}
```

Output:
Complete DSA assignment
Eat Lunch
Study for IP

1.3. Stacks - A data structure that follows the **Last In First Out (LIFO)** principle. Has basic operations such as **push** (adding new element) and **pop** (removing the top element).

Sample illustration of a stack:



Syntax/ initialization:

```
Stack<dataType> variableName = new Stack<>();
```

Sample Code (in Java):

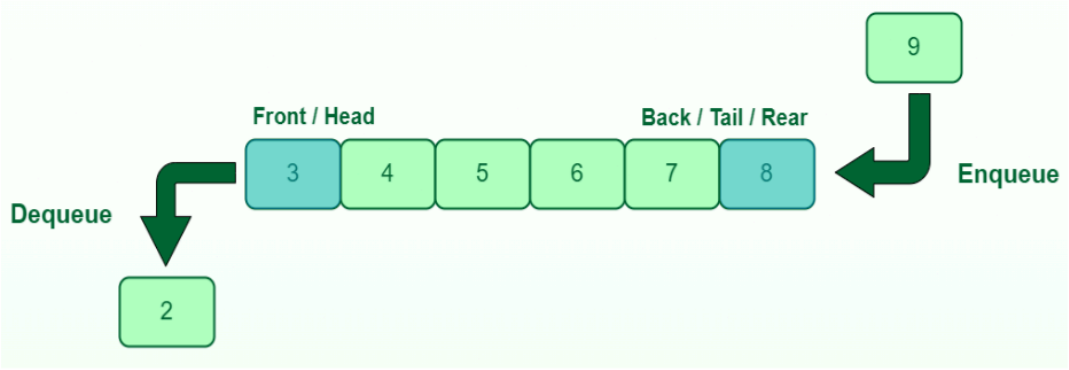
```
import java.util.Stack;

public class DsaBooks {
        public static void main(String[] args) {
            Stack<String> books = new Stack<>();
            books.push("Java Language");
            books.push("Data Structures");
            books.push("Algorithms");

            System.out.println("Top Book: " + books.peek());
            books.pop();

            System.out.println("After removing top book: " + books.peek());
            books.pop();

            System.out.println("Bottom Book: " + books.peek());
```

```
        }
}
```

Output:
Top Book: Algorithms
After removing top book: Data Structures
Bottom Book: Java Language

1.4. Queues - A queue follows the **First In First Out (FIFO)** principle in which it functions like a waiting line. Has basic operations such as **enqueue** (adding new element) and **dequeue** (removing the first or front element).

Sample illustration of a queue:



Queue Data Structure

Syntax/ initialization:

```
Queue<dataType> variableName = new LinkedList<>();
```

Sample Code (in Java):

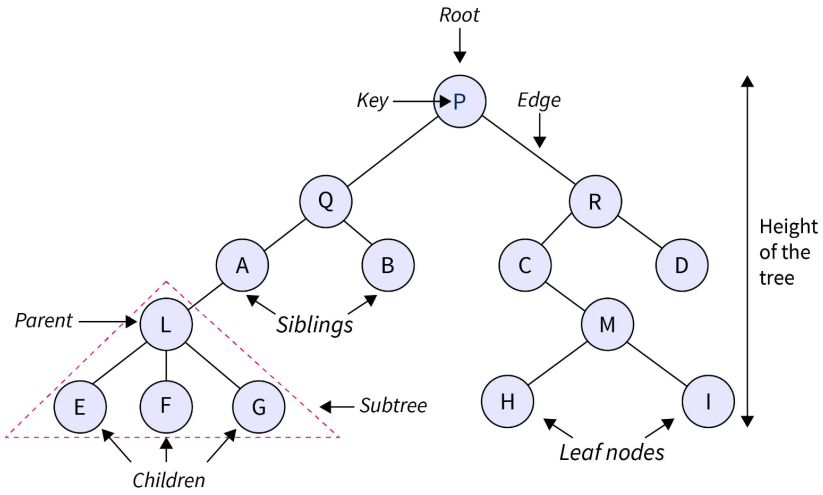
```
import java.util.LinkedList;
import java.util.Queue;

public class QueueExample {
        public static void main(String[] args) {
        Queue<String> customers = new LinkedList<>();
        customers.add("Bob");
        customers.add("Oggy");
        customers.add("Jack");

        System.out.println("Next Customer: " + customers.poll());
        }
}
```

Output:
Next Customer: Bob

2. Non Linear data structure - A type of non primitive data structure that stores data in a non sequential manner.

2.1 Trees - A tree is a **hierarchical** data structure with nodes connected by edges, where **each node has a parent** (except for the root) and children. Trees are often used to represent hierarchical structures such as file systems.

Sample illustration of a Tree data structure:



SCALER Topics

Sample Syntax/ initialization (in Java):

```
class Node {
        int data;
        Node left, right;

public Node(int item) {
        data = item;
        left = right = null;
        }
  }
```

Sample Code (in Java):

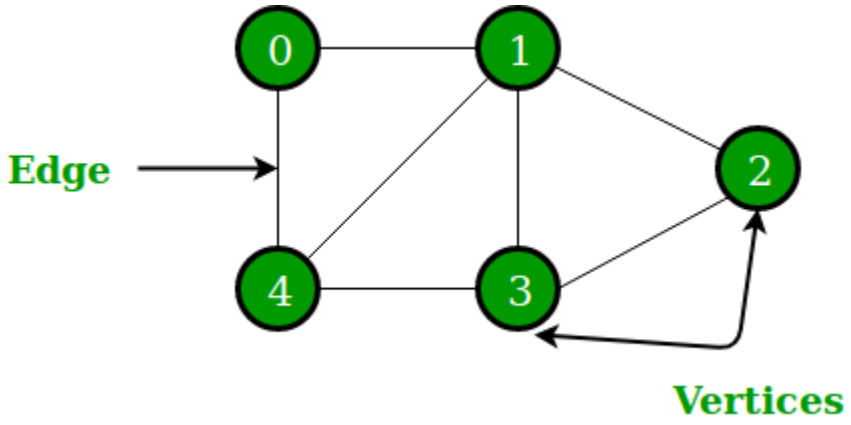
```
public class FamilyTree {

   static class Person {
      String name;
      Person leftChild, rightChild;

      public Person(String name) {
         this.name = name;
         leftChild = rightChild = null;
      }
   }
   public static void printFamilyTree(Person person, int level) {
      if (person == null) {
         return;
      }
      printFamilyTree(person.rightChild, level + 1);
      for (int i = 0; i < level; i++) {
         System.out.print("\t\t");
      }
      System.out.println(person.name);
      printFamilyTree(person.leftChild, level + 1);
   }

   public static void main(String[] args) {
      Person grandparent = new Person("Grandparent");
      grandparent.leftChild = new Person("Parent A");
      grandparent.rightChild = new Person("Parent B");
      grandparent.leftChild.leftChild = new Person("Child 1");
      grandparent.leftChild.rightChild = new Person("Child 2");
      grandparent.rightChild.leftChild = new Person("Child 3");
      grandparent.rightChild.rightChild = new Person("Child 4");

      System.out.println("Family Tree:");
      printFamilyTree(grandparent, 0);
   }
}
```

Output:

```
Family Tree:
                        Child 4
                Parent B
                        Child 3
Grandparent
                        Child 2
                Parent A
                        Child 1
```

2.2. Graphs - A graph is a data structure that has a **collection of nodes** (vertices) and edges where nodes can have **any number of relationships**.

Sample illustration of a Graph data structure:



Sample Syntax/ initialization (in Java): import java.util.*;

```
import java.util.*;

class Graph {
   private Map<dataType, List<dataType>> adjList = new HashMap<>();

   public void addEdge(int source, int destination) {
         adjList.computeIfAbsent(source, k -> newArrayList<>()).add(destination);
         }
   }
```

Sample Code (in Java):

```
import java.util.*;

public class SocialNetwork {
   static class Graph {
      private Map<String, List<String>> friendConnections = new HashMap<>();


      public void addFriendship(String person1, String person2) {
         friendConnections.computeIfAbsent(person1, k -> new
ArrayList<>()).add(person2);
         friendConnections.computeIfAbsent(person2, k -> new
ArrayList<>()).add(person1); // Bidirectional friendship
      }
```

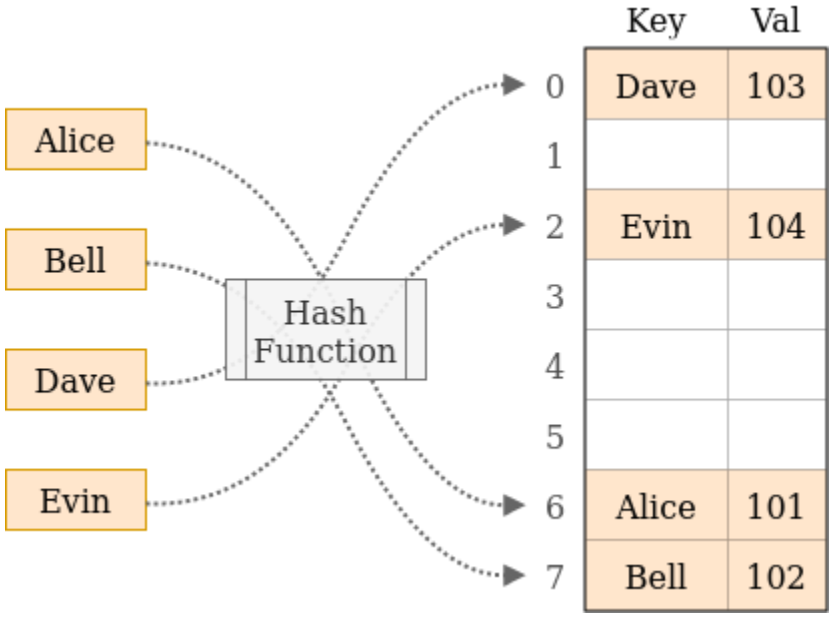
```
    public void printNetwork() {
        System.out.println("Social Network: \n");
        for (Map.Entry<String, List<String>> entry : friendConnections.entrySet()) {
            System.out.println(entry.getKey() + " is friends with " + entry.getValue());
        }
    }
}

public static void main(String[] args) {
    Graph socialGraph = new Graph();

    socialGraph.addFriendship("Patrick", "Vem");
    socialGraph.addFriendship("Patrick", "Reb");
    socialGraph.addFriendship("Patrick", "Marc");
    socialGraph.addFriendship("Reb", "Marc");
    socialGraph.addFriendship("Reb", "Vem");

    socialGraph.printNetwork();
}
}
```

Output:

```
Social Network:

Marc is friends with [Patrick, Reb]
Patrick is friends with [Vem, Reb, Marc]
Reb is friends with [Patrick, Marc, Vem]
Vem is friends with [Patrick, Reb]
```

2.3. Hash Table - A data structure that **stores key-value pairs** that allows fast data retrieval.

Sample illustration of a Hash Table data structure:



Syntax/ initialization:

```
HashMap<KeyType, ValueType> map = new HashMap<>();
```

Sample Code (in Java):

```
import java.util.HashMap;

public class GadgetStore {
        public static void main(String[] args) {
        HashMap<String, Integer> prices = new HashMap<>();

        prices.put("Laptop", 800);
        prices.put("Phone", 500);

        System.out.println("Laptop Price: $" + prices.get("Laptop"));
        }
}
```

```
Output:
Laptop Price: $800
```

References:

https://www.simplilearn.com/tutorials/data-structure-tutorial/what-is-data-structure

https://www.scaler.in/difference-between-primitive-and-non-primitive-data-structure/

https://www.ccbp.in/blog/articles/primitive-data-structure

https://www.w3schools.com/java/java_data_types.asp

https://www.geeksforgeeks.org/what-is-linked-list/

https://www.geeksforgeeks.org/basic-operations-in-stack-data-structure-with-implementations/

https://www.geeksforgeeks.org/what-is-queue-data-structure/

https://www.tutorialspoint.com/data_structures_algorithms/tree_data_structure.htm

https://www.geeksforgeeks.org/applications-of-graph-data-structure/

https://simplerize.com/data-structures/hash-table-introduction