Unit 1
Part 2
Introduction to Data Structures

Introduction

- A *data structure* is an arrangement of data either in computer's memory or on the disk storage.
- Some common examples of data structures are arrays, linked lists, queues, stacks, binary trees, graphs, and hash tables.
- Data structures are widely applied in areas like:
  - ✓ Compiler design
  - ✓ Operating system
  - ✓ Statistical analysis package
  - ✓ DBMS
  - ✓ Numerical analysis
  - ✓ Simulation
  - ✓ Artificial Intelligence

Classification of Data Structures

- **Primitive data structures** are the fundamental data types which are supported by a programming language. Some basic data types are integer, real, and boolean. The terms ‘data type’, ‘basic data type’, and ‘primitive data type’ are often used interchangeably.

- **Non-primitive data structures** are those data structures which are created using primitive data structures. Examples of such data structures include linked lists, stacks, trees, and graphs.

- Non-primitive data structures can further be classified into two categories: ***linear*** and ***non-linear*** data structures.

Classification of Data Structures

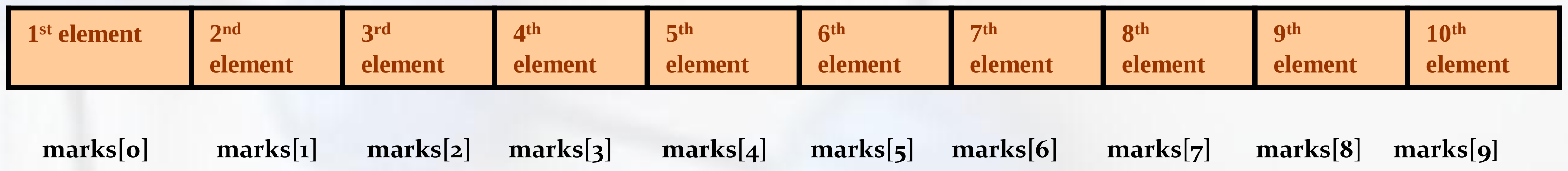
- If the elements of a data structure are stored in a linear or sequential order, then it is a ***linear* data structure**.
- Examples are arrays, linked lists, stacks, and queues.

- If the elements of a data structure are not stored in a sequential order, then it is a ***non-linear* data structure**.
- Examples are trees and graphs.

Arrays

- An array is a collection of similar data elements.
- The elements of an array are stored in consecutive memory locations and are referenced by an index (also known as the subscript).
- Arrays are declared using the following syntax:

```
type   name[size];
```

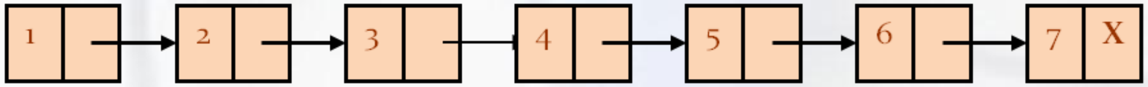


Linked Lists

- A linked list is a very flexible dynamic data structure in which elements can be added to or deleted from anywhere.
- In a linked list, each element (called a *node*) is allocated space as it is added to the list.
- Every node in the list points to the next node in the list. Therefore, in a linked list every node contains two types of information:

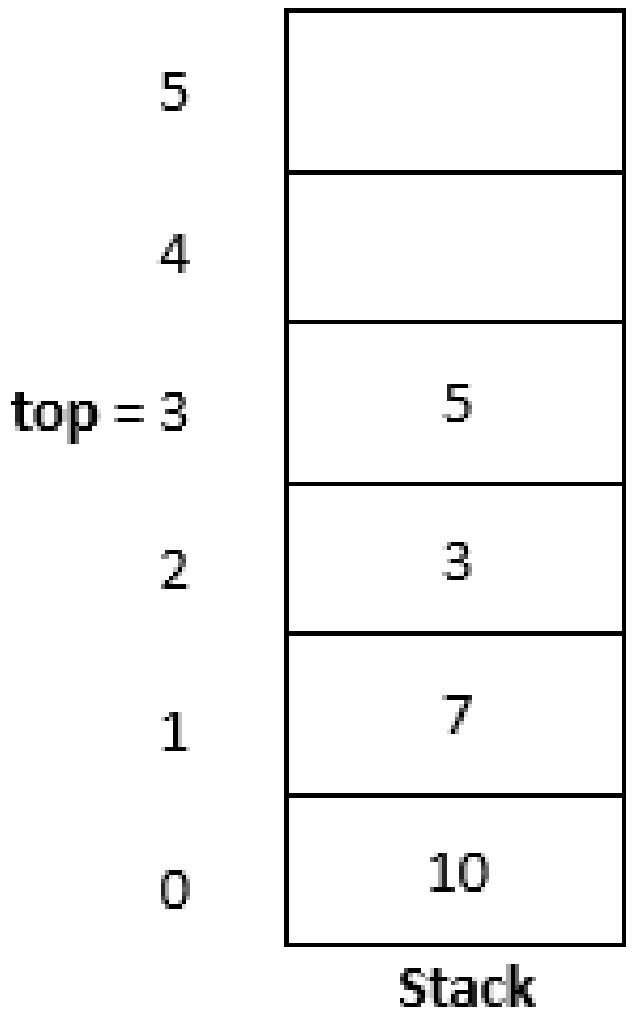
✓ The data stored in the node

✓ A pointer or link to the next node in the list



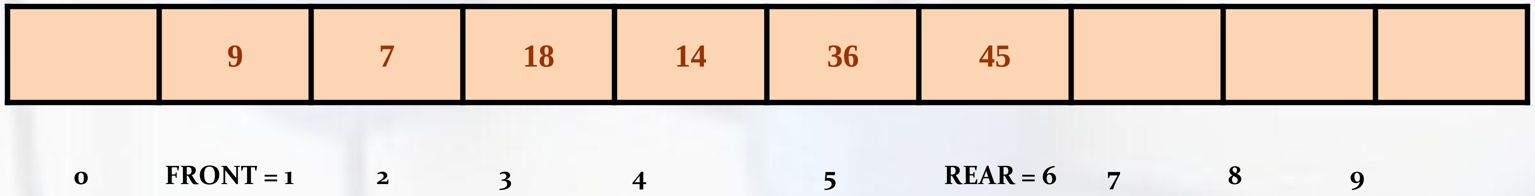
Stack

- A stack is a last-in, first-out (LIFO) data structure in which insertion and deletion of elements are done only at one end, known as TOP of the stack.
- Every stack has a variable TOP associated with it, which is used to store the address of the topmost element of the stack.
- If TOP = NULL, then it indicates that the stack is empty.
- If TOP = MAX-1, then the stack is full.



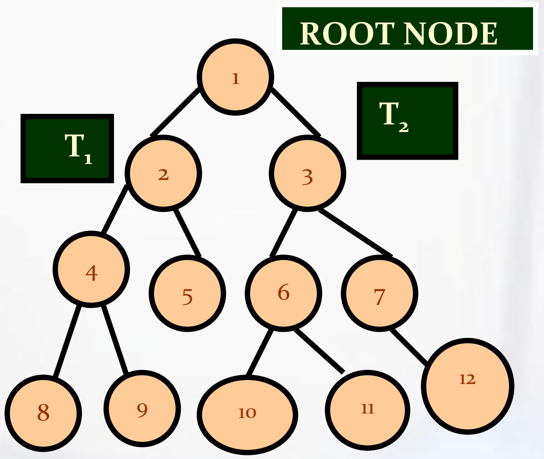
Queue

- A queue is a FIFO (first-in, first-out) data structure in which the element that is inserted first is the first one to be taken out.
- The elements in a queue are added at one end called the REAR and removed from the other one end called FRONT.
- When REAR = MAX – 1, then the queue is full.
- If FRONT = NULL and Rear = NULL, this means there is no element in the queue.



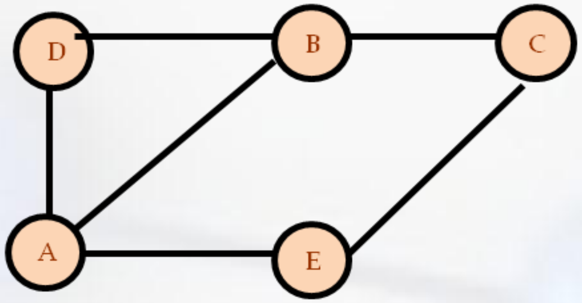
Tree

- A tree is a non-linear data structure which consists of a collection of nodes arranged in a hierarchical order.
- One of the nodes is designated as the root node, and the remaining nodes can be partitioned into disjoint sets such that each set is the sub-tree of the root.
- A binary tree is the simplest form of tree which consists of a root node and left and right sub-trees.
- The root element is pointed by 'root' pointer.
- If root = NULL, then it means the tree is empty.



Graph

- A graph is a non-linear data structure which is a collection of *vertices* (also called *nodes*) and *edges* that connect these vertices.
- A graph is often viewed as a generalization of the tree structure, where instead of a having a purely parent-to-child relationship. between nodes, any kind of complex relationship can exist.
- Every node in the graph can be connected with any other node.
- When two nodes are connected via an edge, the two nodes are known as neighbors.



Abstract Data Type

- An ***Abstract Data Type*** **(ADT)** is the way at which we look at a data structure, focusing on what it does and ignoring how it does its job.

- For example, stacks and queues are perfect examples of an abstract data type. We can implement both these ADTs using an array or a linked list. This demonstrates the "abstract" nature of stacks and queues.

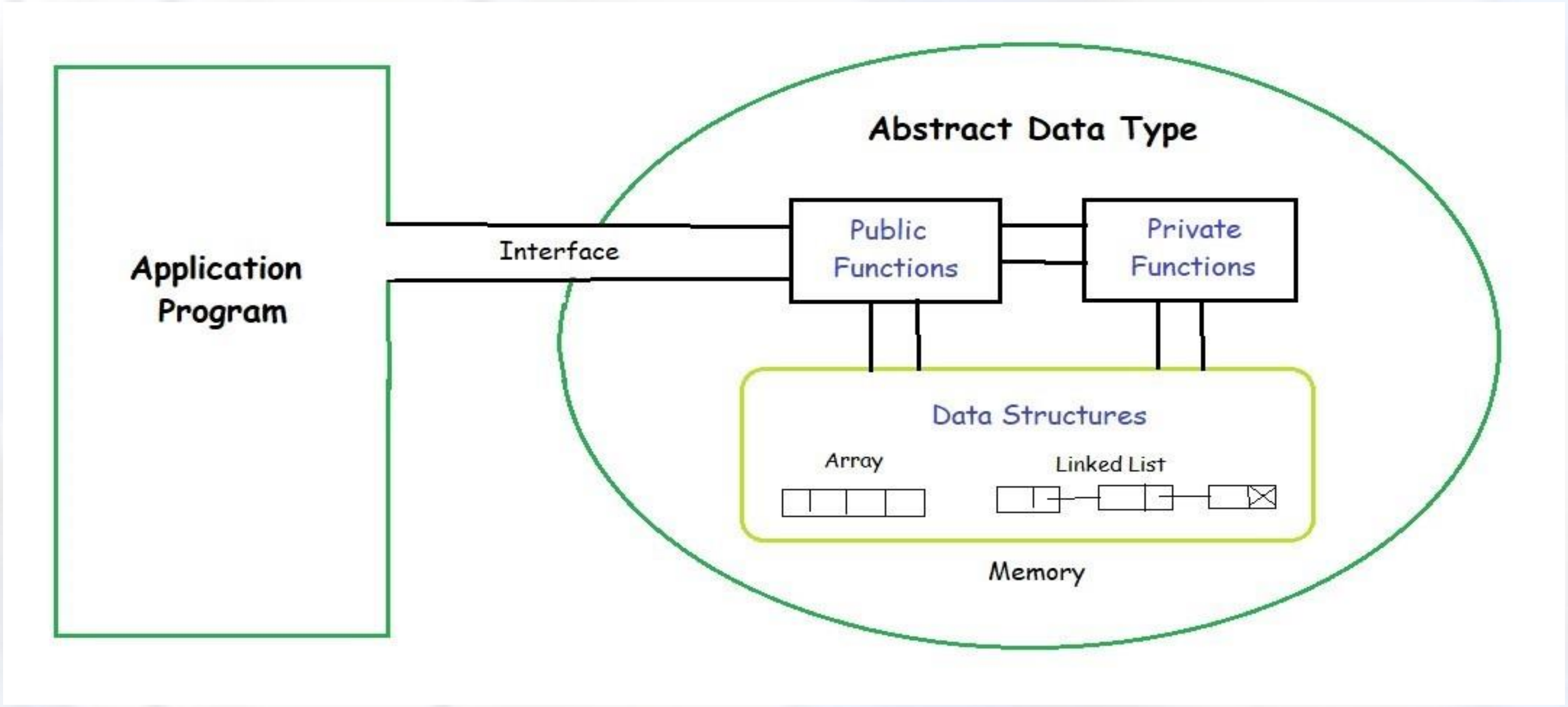
Abstract Data Type

- In C, an Abstract Data Type can be a structure considered without regard to its implementation. It can be thought of as a "description" of the data in the structure with a list of operations that can be performed on the data within that structure.

- The end user is not concerned about the details of how the methods carry out their tasks. They are only aware of the methods that are available to them and are concerned only about calling those methods and getting back the results but not HOW they work.

Abstract Data Type

- Applications that use ADT is shielded from implementation complexities of the data structure(DS), by using only the public interface of the DS.



E.g.1: Stack ADT

Possible interface functions/operation:

- push() – Insert an element at one end of the stack called top.
- pop() – Remove and return the element at the top of the stack, if it is not empty.
- peek() – Return the element at the top of the stack without removing it, if the stack is not empty.
- size() – Return the number of elements in the stack.
- isEmpty() – Return true if the stack is empty, otherwise return false.
- isFull() – Return true if the stack is full, otherwise return false.

E.g. 2: Queue ADT

Possible interface functions/operations:

- enqueue() – Insert an element at the end of the queue.
- dequeue() – Remove and return the first element of the queue, if the queue is not empty.
- peek() – Return the element of the queue without removing it, if the queue is not empty.
- size() – Return the number of elements in the queue.
- isEmpty() – Return true if the queue is empty, otherwise return false.
- isFull() – Return true if the queue is full, otherwise return false.

E.g. 3: List ADT

Possible interface functions/operations:

- get() – Return an element from the list at any given position.
- insert() – Insert an element at any position of the list.
- remove() – Remove the first occurrence of any element from a non-empty list.
- removeAt() – Remove the element at a specified location from a non-empty list.
- replace() – Replace an element at any position by another element.
- size() – Return the number of elements in the list.
- isEmpty() – Return true if the list is empty, otherwise return false.
- isFull() – Return true if the list is full, otherwise return false.