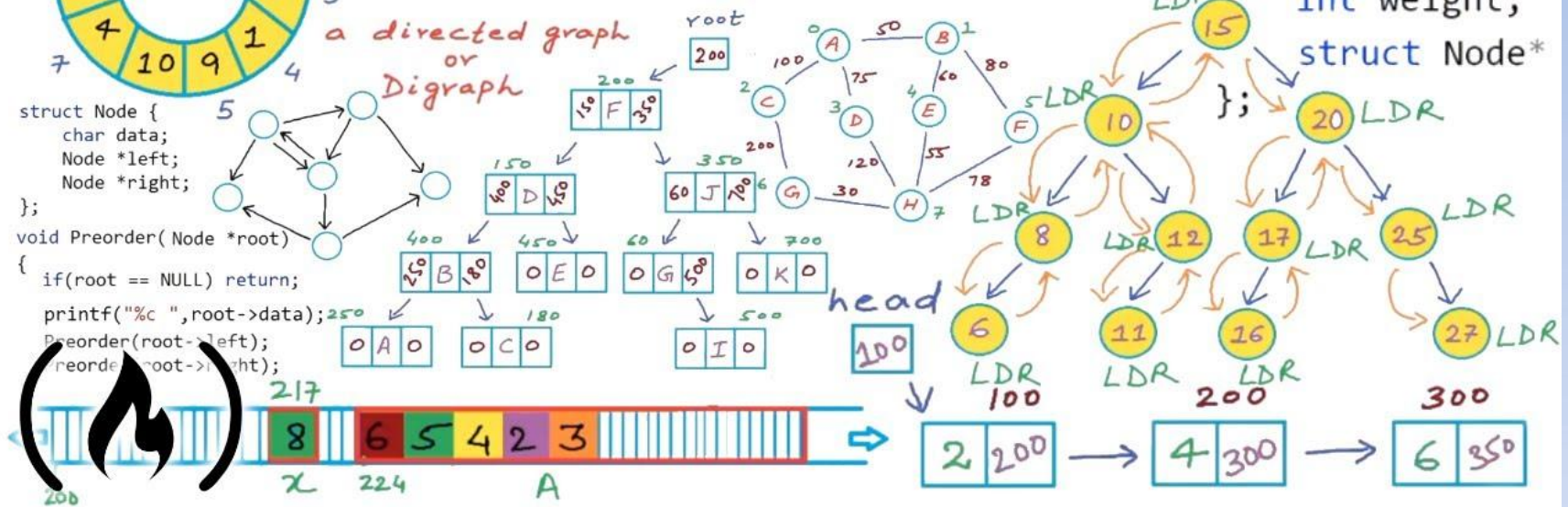


# C and C++

```
struct Node {
    int data;
    int weight;
    struct Node*
};
```



1

# **MODULE 1**

## **Introduction**

# Data Structure

- A data structure is a specialized format for **organizing, processing, retrieving** and **storing** data.
- While there are several basic and advanced structure types, any data structure is designed **to arrange data to suit a specific purpose** so that it can be accessed and worked with in appropriate ways.

# Data Structure

- In computer programming, a data structure may be selected or designed to store data for the purpose of working on it with various algorithms.
- Each data structure contains information about the data values, relationships between the data and functions that can be applied to the data.

# Data Structure

- The data structure is basically a **technique of organizing and storing** of different types of data items in computer memory.
- It is considered as **not only the storing of data elements but also the maintaining of the logical relationship existing** between individual data elements.
- The Data structure can also be defined **as a mathematical or logical model**, which relates to a particular organization of different data elements.<sup>5</sup>

# Data Structure

- ***Data:***
  - Data is the basic entity of fact that is used in calculations or manipulation process.
  - The way of **organizing of the data** & **performing the operations** is called as data structure.  
**Data structure=organized data+ operations**
  - Operations
    - Insertion
    - Deletions
    - Searching
    - Traversing

# Data Structure

- The organization must be convenient for users.
- Data structures are implemented in the real time in the following situations:
  - Car park
  - File storage
  - Machinery
  - Shortest path
  - Sorting
  - Networking
  - Evaluation of expressions

# Data Structure

- Specification of data structure :
  - Data structures are considered as the main building blocks of a **computer program**.
    - **Organization** of data
    - **Accessing methods**
    - **Degree of associativity**
    - **Processing alternatives for information**

**Algorithm + Data Structure = Program**



# Data Structure

- At the time of selection of data structure we should follow these two things so that our selection is efficient enough to solve our problem.
  - The data structure **must be powerful** enough to handle the different relationship existing between the data.
  - The structure of data also **to be simple**, so that we can efficiently process data when required.

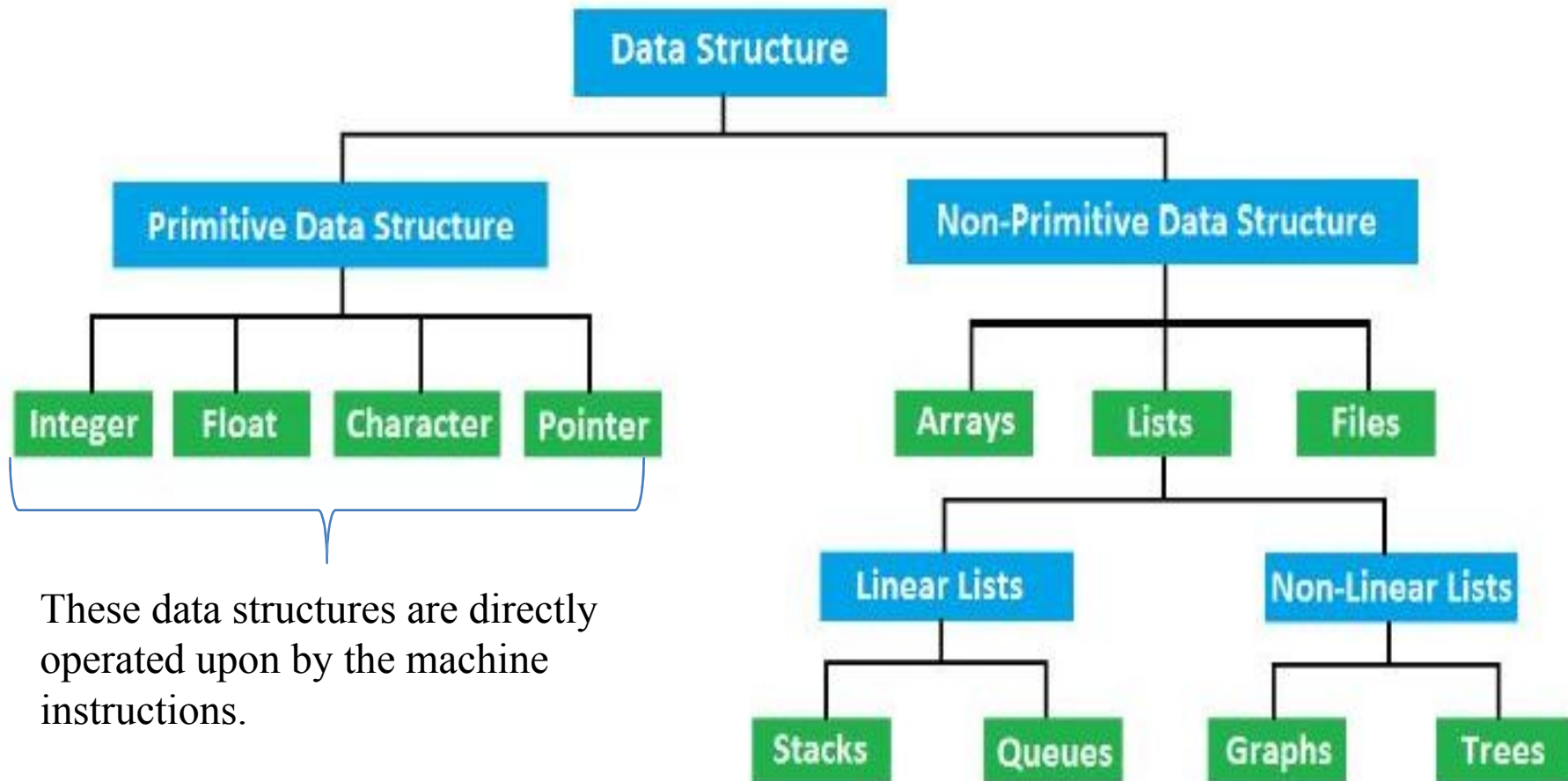
# Characteristics of data structures

- **Linear or non-linear:** This characteristic describes whether the data items are arranged in chronological sequence, such as with an **array**, or in an **unordered** sequence, such as with a **graph**.
- **Homogeneous or non-homogeneous:** This characteristic describes whether all data items in a given repository are of the same type or of various types.

# Characteristics of data structures

- **Static or dynamic:** This characteristic describes how the data structures are compiled. Static data structures have **fixed sizes**, structures and memory locations at compile time.
- Dynamic data structures have sizes, structures and memory locations that can shrink or **expand** depending on the use.

# Types of data structures



# Types of data structures

- *Primitive data structure :*
  - The primitive data structures are known as basic data structures.
  - These data structures are **directly operated upon by the machine instructions.**
  - The primitive data structures have different representation on different computers.

# Types of data structures

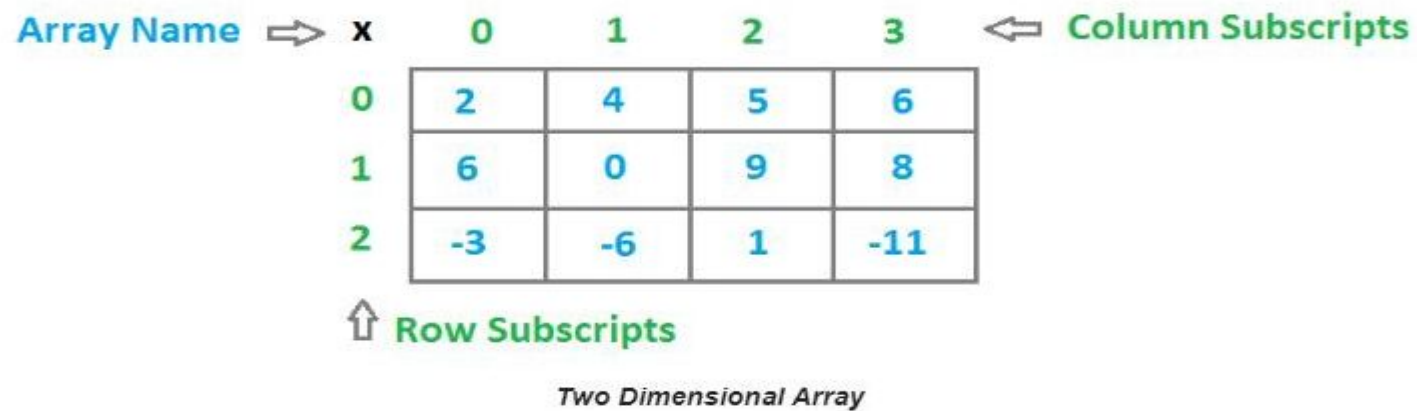
- *Non-Primitive data structure :*
  - The non-primitive data structures are highly developed **complex** data structures.
  - Basically these are developed from the primitive data structure.
  - The non-primitive data structure is responsible for organizing the group of homogeneous and heterogeneous data elements.

# Types of data structures

- Data structure types are determined by what types of operations are required or what kinds of algorithms are going to be applied.
- *Arrays-*
  - An array stores a collection of items at adjoining memory locations.
  - Items that are the same type get stored together so that the position of each element can be calculated or retrieved easily.
  - Arrays can be fixed or flexible in length.

# Types of data structures

- Arrays-



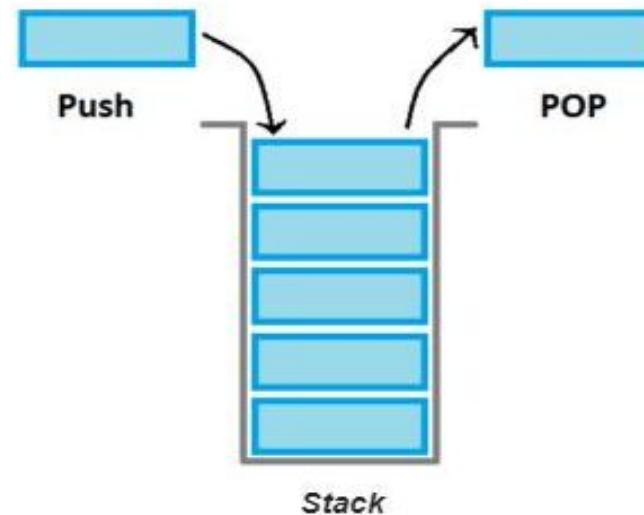


# Types of data structures

- **Stacks-**

We can implement a stack using 2 ways:

- Static implementation (using arrays)
- Dynamic implementation (using pointers)



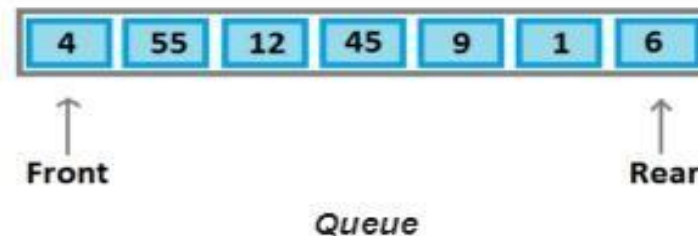
# Types of data structures

- **Queues-**

- A queue stores a collection of items similar to a stack; however the operation order can only be first in first out.

We can also implement queues using 2 ways :

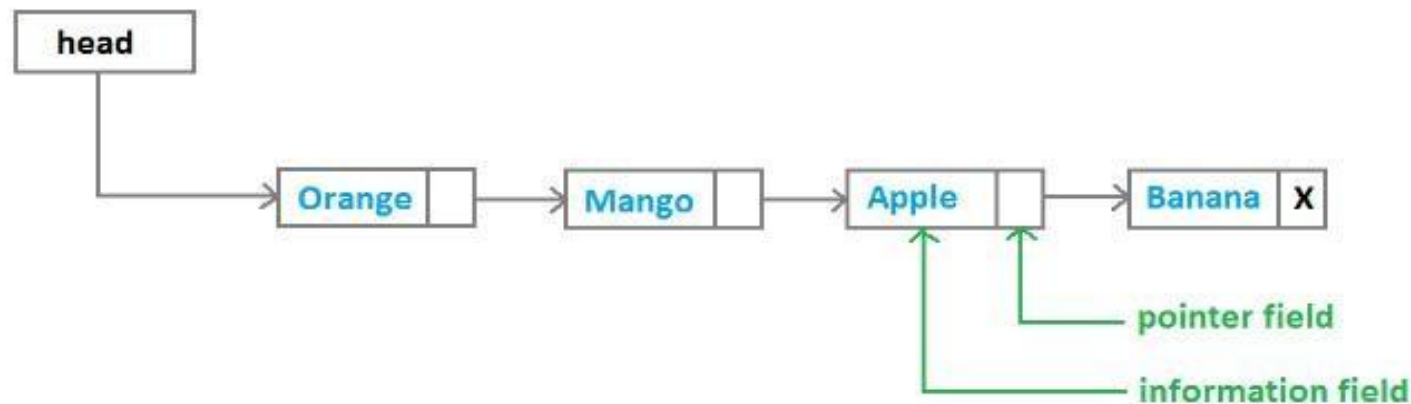
- Using arrays
- Using pointers



# Types of data structures

- **Linked lists-**

- A linked list stores a collection of items in a linear order. Each element or node in a linked list contains a **data item as well as a reference or link to the next item** in the list.

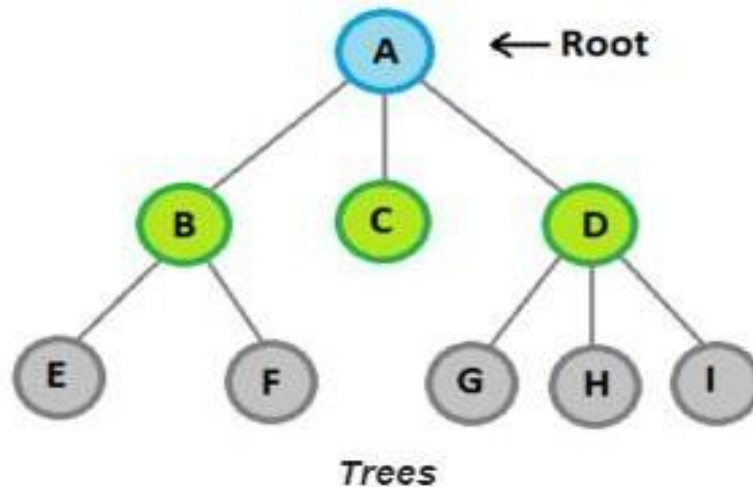


*Linear linked lists*

# Types of data structures

- **Trees-**

- A tree stores a collection of items in an abstract **hierarchical** way.
- Each node is linked to other nodes and can have multiple sub-values also known as children.



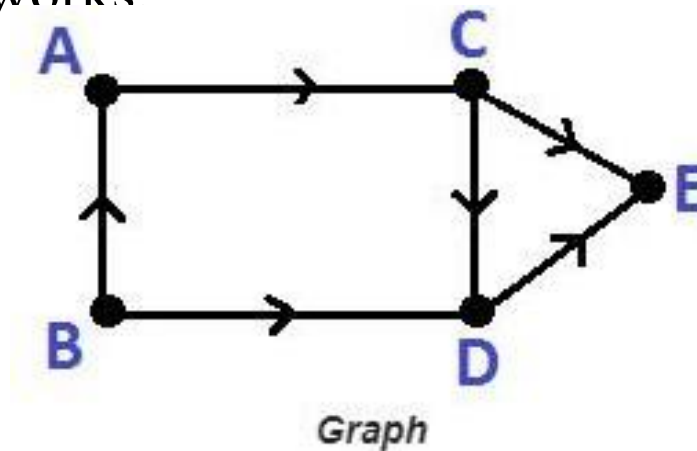
# Types of data structures

- *A Tree has the following characteristics :*
  - The top item in a hierarchy of a tree is referred as the **root** of the tree.
  - The remaining data elements are partitioned into a number of mutually exclusive subsets and they itself a tree and are known as the **subtree**.
  - Unlike natural trees trees in the data structure always grow in length towards the bottom.

# Types of data structures

- **Graphs-**

- A graph stores a collection of items in a non-linear fashion.
- Graphs are made up of a finite set of nodes also known as **vertices** and lines that connect them also known as **edges**.
- These are useful for representing real-life systems such as computer networks



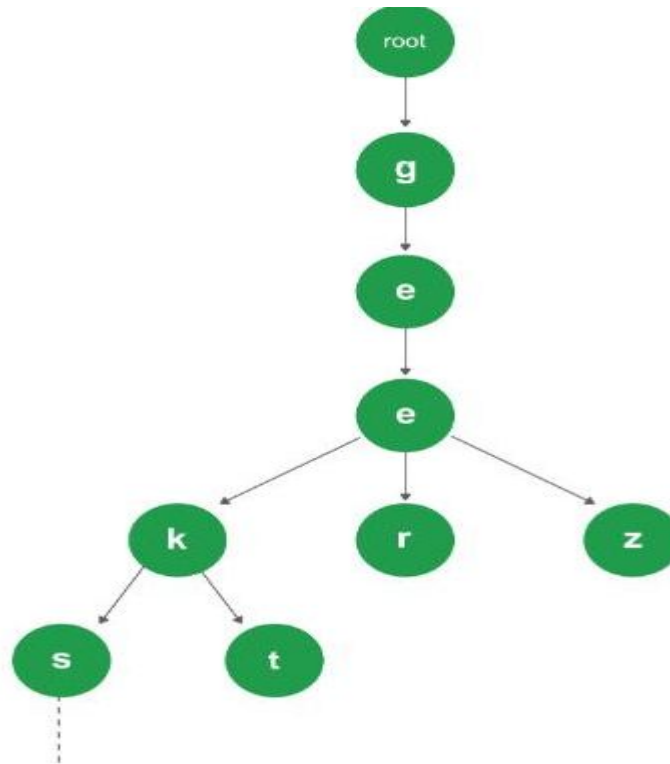
# Types of data structures

- *The different types of Graphs are :*
  - Directed Graph
  - Non-directed Graph
  - Connected Graph
  - Non-connected Graph
  - Simple Graph
  - Multi-Graph

# Types of data structures

- **Tries-**

- A trie or keyword tree, is a data structure that stores strings as data items that can be organized in a visual graph.





# Types of data structures

- **Hash tables-**

- A hash table or a hash map stores a collection of items in an **associative array** that **plots keys to values**.
- A hash table uses a **hash function** to convert an index into an array of buckets that contain the desired data item.
- Overcoming the drawbacks of linear data structures hashing is introduced.

# Types of data structures

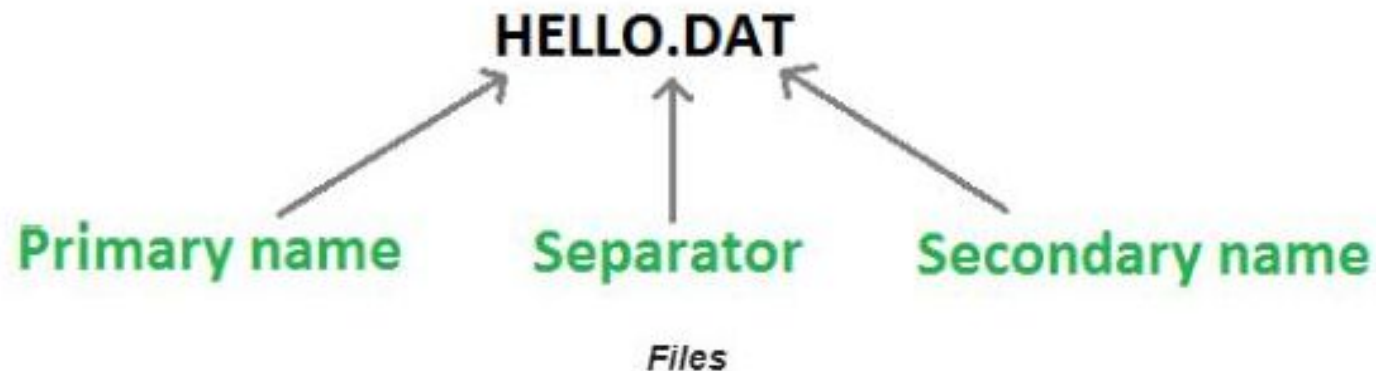
- ***Files :***

- Files contain data or information, stored permanently in the secondary storage device such as Hard Disk and Floppy Disk.
- It is useful when we have to store and process a large amount of data.
- A file stored in a storage device is identified by a unique file name using a file extension like **HELLO.DAT** or **TEXTNAME.TXT** and so on.

# Types of data structures

- ***Files :***

- A file name normally contains a primary and a secondary name which is separated by a dot(.).



# Fundamentals of data structures:

- **Fundamental Data Structures**

- The following four data structures are used ubiquitously in the description of algorithms and serve as basic building blocks for realizing more complex data structures.
  - Sequences (also called as lists)
  - Dictionaries
  - Priority Queues
  - Graphs
- Dictionaries and priority queues can be classified under a broader category called *dynamic sets*.
- Binary and general trees are very popular building blocks for implementing dictionaries and priority queues.

# Fundamentals of data structures:

## Dictionaries

- A *dictionary* is a general-purpose data structure for storing a group of objects.
- A dictionary has a set of *keys* and each key has a single associated *value*.
- When presented with a key the dictionary will return the associated value.
- A dictionary is also called a *hash*, a *map*, a *hashmap* in different programming languages.

# Fundamentals of data structures:

## Dictionaries

- For example, the results of a classroom test could be represented as a dictionary with pupil's names as keys and their scores as the values
- ```
results = { 'Detra' : 17,  
            'Nova' : 84,  
            'Charlie' : 22,  
            'Henry' : 75,  
            'Roxanne' : 92,  
            'Elsa' : 29 }
```
- Instead of using the numerical index of the data we can use the dictionary names to return values
- ```
>>> results['Nova']  
84
```
- ```
>>> results['Elsa']  
29
```

# Fundamentals of data structures:

## Dictionaries

- The keys in a dictionary must be simple types (such as integers or strings) while the values can be of any type.
- Different languages enforce different type restrictions on keys and values in a dictionary.
- Dictionaries are often implemented as hash tables.
- Keys in a dictionary must be **unique**
- An attempt to create a duplicate key will typically overwrite the existing value for that key.

# Fundamentals of data structures:

## Dictionaries

- Dictionary is an abstract data structure that supports the following operations:
  - search(K key) (returns the value associated with the given key)
  - insert(K key, V value)
  - delete(K key)
- Each element stored in a dictionary is identified by a key of type K.
- Dictionary represents a mapping from keys to values.



# Fundamentals of data structures:

## Dictionaries

- Dictionaries have numerous applications.
  - contact book
    - key: name of person; value:
  - telephone number table of program variable identifiers
    - key: identifier; value: address in memory
  - property-value collection
    - key: property name; value: associated value
  - natural language dictionary
    - key: word in language X; value: word in language Y

# Fundamentals of data structures:

## *operations on dictionaries*

- Dictionaries typically support several operations:
  - **retrieve a value** (depending on language, attempting to retrieve a missing key may give a default value or throw an exception)
  - **insert or update a value** (typically, if the key does not exist in the dictionary, the key-value pair is inserted; if the key already exists, its corresponding value is overwritten with the new one)
  - **remove a key-value pair**
  - test for existence of a key
- Note that items in a dictionary are unordered, so loops over dictionaries will return items in an arbitrary order.

# Abstract Data Type

**A collection of related data is known as an *abstract data type* (ADT)**

**Data Structure = ADT + Collection of functions that operate on the ADT**

**Abstract data types are the entities that are definitions of data and operations but do not have implementation details**

- Consist of the data **structure** definition and a collection of functions that operate on the **struct**
  - **We will never access the struct directly!**
- Separate *what* you can do with data from *how* it is represented
- Other parts of the program interacts with data through provided operations according to their specifications
- Implementation chooses how to represent data and implement its operations

# Abstract Data Type

To handle the complex problems, the computer programmer uses **abstraction** to focus on what it does and ignoring how it does its job. ADTs are purely theoretical entities, used to simplify the description of algorithms, used to classify and evaluate the data structures.

An abstract data type (ADT) is an object with a **generic description independent of implementation details**. This description includes a specification of the components from which the object is made and also the behavioral details of the object.

# ADTs as interfaces

- **abstract data type (ADT):** A specification of a collection of data and the operations that can be performed on it.
  - Describes *what* a collection does, not *how* it does it.
- Java's collection framework uses interfaces to describe ADTs:
  - `Collection`, `Deque`, `List`, `Map`, `Queue`, `Set`
- An ADT can be implemented in multiple ways by classes:
  - `ArrayList` and `LinkedList` implement `List`
  - `HashSet` and `TreeSet` implement `Set`
  - `LinkedList`, `ArrayDeque`, etc. implement `Queue`

# Why use ADTs?

- Why would we want more than one kind of list, queue, etc.?
- **Answer:** Each implementation is more efficient at certain tasks.
  - `ArrayList` is faster for adding/removing at the end;  
`LinkedList` is faster for adding/removing at the front/middle.  
Etc.
  - You choose the optimal implementation for your task, and if the rest of your code is written to use the ADT interfaces, it will work.
  - Abstract data type in data structure type allows **reusability** of a **code** i.e. it makes it convenient for a programmer to write a shorter code.

## **Abstract Data Type (ADT)**

Type defined in terms of its data items and associated operations, **not its implementation.**

## **Abstract Data Types (ADT)**

- We are well acquainted with data types by now, like integers, arrays, and so on.
- To access the data, we've used operations defined in the programming language for the data type, for instance by accessing array elements by using the square bracket notation.
- An abstract data type is a data type whose representation is hidden from, and of no concern to the application code.



**An Abstract Data Type (ADT) is a blueprint for creating a data structure that defines the behavior and interface of the structure, without specifying how it is implemented.**

**An ADT is a set of operations that can be performed on a set of values. This set of operations actually defines the behavior of the data structure, and they are used to manipulate the data in a way that suits the needs of the program. ADT allows the programmers to use the functions while hiding the implementation details.**

**Examples of abstract data type in data structures are List, Stack, Queue, etc.**

# Abstract Data Types (ADT)

- For example, when writing application code, we don't care how strings are represented: we just declare variables of type string, and manipulate them by using string operations.
- Once an abstract data type has been designed, the programmer responsible for implementing that type is concerned only with choosing a suitable data structure and coding up the methods.
- On the other hand, application programmers are concerned only with using that type and calling its methods without worrying much about how the type is implemented.

# The Array As An ADT

- An array is probably the most versatile or fundamental Abstract Data Type
- An **array** is a finite sequence of storage cells, for which the following operations are defined:
  - $\text{create}(A, N)$  creates an array  $A$  with storage for  $N$  items;
  - $A[i] = \text{item}$  stores item in the  $i^{\text{th}}$  position in the array  $A$ ; and
  - $A[i]$  returns the value of the item stored in the  $i^{\text{th}}$  position in the array  $A$ .

# Showing that an array is an ADT

Let A be an array of type T and has n elements then it satisfied the following operations

1. **CREATE(A)**: Create an array A
2. **INSERT(A,X)**: Insert an element X into an array A in any location
3. **DELETE(A,X)**: Delete an element X from an array A
4. **MODIFY(A,X,Y)**: modify element X by Y of an array A
5. **TRAVELS(A)**: Access all elements of an array A
6. **MERGE(A,B)**: Merging elements of A and B into a third array C

Thus by using 1D array we can perform above operations thus an array acts as an ADT.

## Definition

- A stack is an *ordered collection of items* into which new items may be *inserted* and from which items may be *deleted* at one end, called the *top* of the stack.
- Stack is a linear data structure where all the *insertions* and *deletions* are done at end rather than in the middle.

# Stack as an abstract data type

- A stack of elements of type  $T$  is a finite sequence of elements together with the operations
  1. **CreateEmptyStack( $S$ )**: create or make stack  $S$  be an empty stack
  2. **Push( $S, x$ )**: Insert  $x$  at one end of the stack, called its **top**
  3. **Top( $S$ )**: If stack  $S$  is not empty; then retrieve the element at its **top**
  4. **Pop( $S$ )**: If stack  $S$  is not empty; then delete the element at its **top**
  5. **IsFull( $S$ )**: Determine if  $S$  is full or not. Return **true** if  $S$  is full stack; return **false** otherwise
  6. **IsEmpty( $S$ )**: Determine if  $S$  is empty or not. Return **true** if  $S$  is an empty stack; return **false** otherwise.

# Operations

- Push – Place something on stack on the top
- Pop – Remove the very top item
- Top – examine without removing top item
- IsEmpty – Tells us whether the stack is empty
- size – how many elements are there?



## The advantages of ADT in Data Structures are:

- Provides **abstraction**, which simplifies the complexity of the data structure and **allows users to focus on the functionality**.
- Enhances **program modularity** by allowing the data structure implementation to be separate from the rest of the program.
- Enables **code reusability** as the same data structure can be used in multiple programs with the same interface.
- Promotes the concept of **data hiding** by encapsulating data and operations into a single unit, which **enhances security and control over the data**.
- Supports **polymorphism**, which allows the same interface to be used with different underlying data structures, providing **flexibility and adaptability** to changing requirements.



```
typedef struct {  
    double real;  
    double imag;  
} complex;
```

Let us also illustrate the implementation of some arithmetic routines on complex numbers:

```
complex cadd ( complex z1 , complex z2 )  
{  
    complex z;  
    z.real = z1.real + z2.real;  
    z.imag = z1.imag + z2.imag;  
    return z;  
}
```

```
complex cmul ( complex z1 , complex z2 )  
{  
    complex z;  
    z.real = z1.real * z2.real - z1.imag * z2.imag;  
    z.imag = z1.real * z2.imag + z1.imag * z2.real;  
    return z;  
}
```

```
complex conj ( complex z1 )  
{  
    complex z;  
    z.real = z1.real;  
    z.imag = -z1.imag;  
    return z;  
}
```

```
void cprn ( complex z )  
{  
    printf("(%.1f) + i(%.1f)", z.real, z.imag);  
}
```

## Complex Numbers ADT Implementation

# Matrix ADT Implementation

```
#define MAXROW 10
#define MAXCOL 15
typedef struct {
    int rowdim;
    int coldim;
    complex entry[MAXROW][MAXCOL];
} matrix;
```

in order to define an ADT we need to specify:

- The components of an object of the ADT.
- A set of procedures that provide the behavioral description of objects belonging to the ADT.

Let us now implement some basic arithmetic operations on these matrices.

```
matrix msetid ( int n )
```

```
{
    matrix C;
    int i, j;

    if ((n > MAXROW) || (n > MAXCOL)) {
        fprintf(stderr, "msetid: Matrix too big\n");
        C.rowdim = C.coldim = 0;
        return C;
    }
    C.rowdim = C.coldim = n;
    for (i = 0; i < C.rowdim; ++i) {
        for (j = 0; j < C.coldim; ++j) {
            A.entry[i][j].real = (i == j) ? 1 : 0;
            A.entry[i][j].imag = 0;
        }
    }
    return C;
}
```

```
matrix madd ( matrix A , matrix B )
```

```
{
    matrix C;
    int i, j;

    if ((A.rowdim != B.rowdim) ||
        (A.coldim != B.coldim)) {
        fprintf(stderr, "madd: Matrices of
incompatible dimensions\n");
        C.rowdim = C.coldim = 0;
        return C;
    }

    C.rowdim = A.rowdim;
    C.coldim = A.coldim;
    for (i = 0; i < C.rowdim; ++i)
        for (j = 0; j < C.coldim; ++j)
            C.entry[i][j] =
cadd(A.entry[i][j],B.entry[i][j]);
    return C;
}
```

```
matrix mmul ( matrix A , matrix B )
```

```
{  
    matrix C;  
    int i, j, k;  
    complex z;  
  
    if (A.coldim != B.rowdim) {  
        fprintf(stderr, "mmul: Matrices of incompatible dimensions\n");  
        C.rowdim = C.coldim = 0;  
        return C;  
    }  
    C.rowdim = A.rowdim;  
    C.coldim = B.coldim;  
    for (i = 0; i < A.rowdim; ++i) {  
        for (j = 0; j < B.coldim; ++j) {  
            C.entry[i][j].real = 0;  
            C.entry[i][j].imag = 0;  
            for (k = 0; k < A.coldim; ++k) {  
                z = cmul(A.entry[i][k], B.entry[k][j]);  
                C.entry[i][j] = cadd(C.entry[i][j], z);  
            }  
        }  
    }  
    return C;  
}
```

Any questions?

