Week 1; Lecture 1

Data Structures: Introduction

Class

- Lecture will introduce new concepts
- Tutorials will test your understanding the concepts
- Labs will teach you to program
  - One learns programming only by programming
    - We can tell you things, but the words only have meaning when you program

Lectures and Tutorial

- Each week, you will get a list of questions, which are answered in the lecture.
- During tutorial, you will have the opportunity to check your answers
  - You will have a quiz comprising a selection of the questions
- The sessionals and finals will comprise a selection of these questions

Lab

- **The lab is where you learn programming**
- Each lab will be a viva where you demonstrate the work you have done on the backlog.
  - You may work on your program during lab, but most work is done outside class.
- Programs from lab comprise the majority of sessionals and the final.

Class Content

- Data Structures
  - Organization of data for efficient algorithms
  - Implementation of operations on algorithms
- Analysis of Algorithms
  - Definition and measurement of algorithmic efficiency

Data Structures

- An important step in problem solving
  1)Selection the appropriate data structure
  2)Design a suitable algorithm

- Overall program performance depends on:
  - Choice of data structure
  - Design of algorithm to use the data structure

Example

- As a teacher, I want to track student attendance, so I know when students bunking.
  - Data needed: serial number, name, attendance
- Two approaches
  - Three arrays:
    - int sno[30]; char *name[30]; int attend[30]
  - Array of structures
    - Struct student {

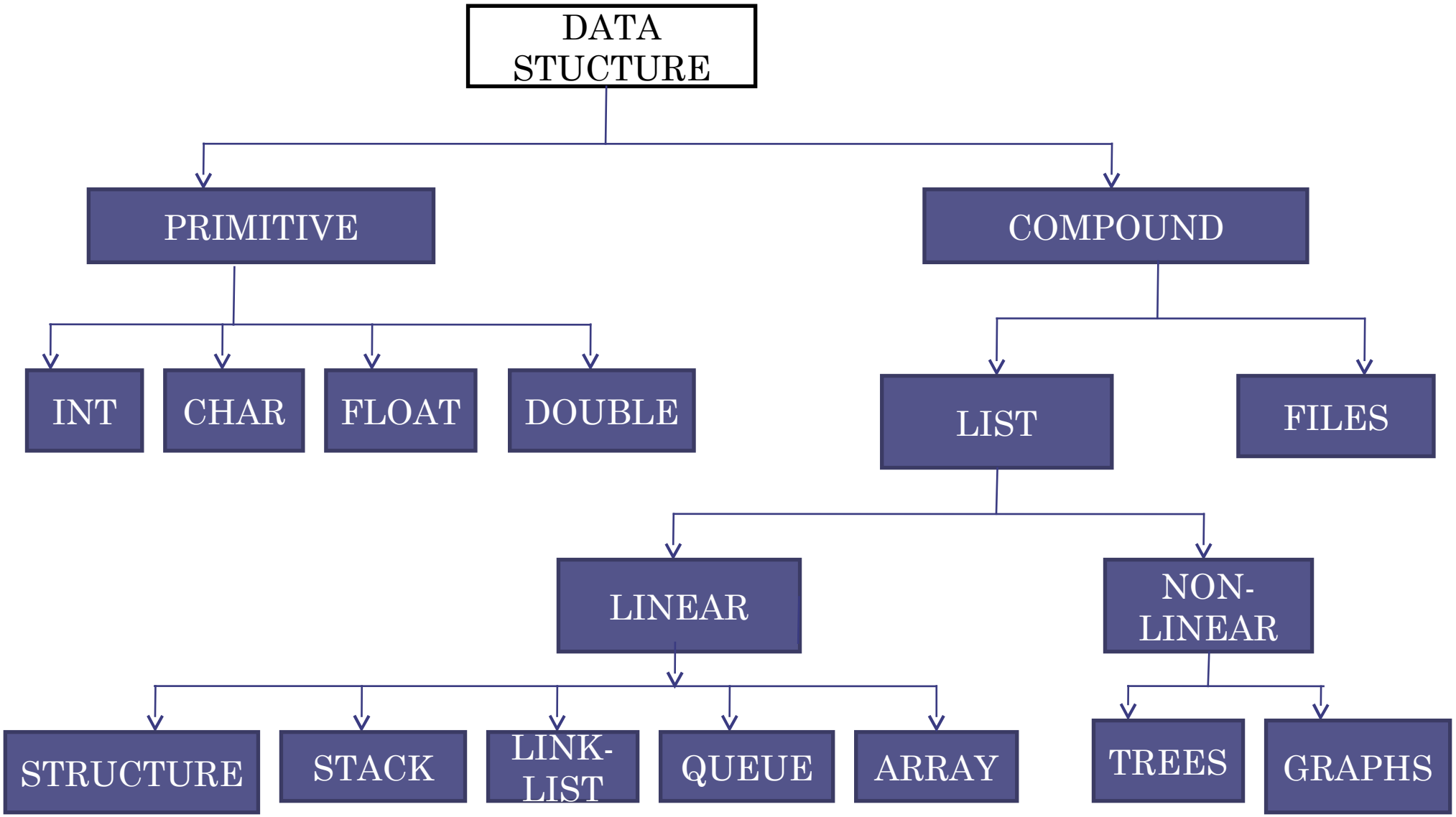
      int sno;

      char *name

      int attend

      } db[30]

Types of Data Structures



Types of Data Structures

- Primitive Data Structures
  - Defined by hardware
  - E.g., int, int *, float, float * char, char *, double, double * ...
- Compound Data Structures
  - Build from Primitive Data Structures
  - Array: multiple primitive DS
  - Structs: collection of primitive DS
  - Lists, Stacks, Trees, etc.: build using pointers

Linear Lists

- Sequence of items with a linear ordering
  - Each element is followed by at most one additional element
  - i.e., no element may be followed

Types Of Linear Lists:

ARRAYS:

An array is used to store elements of the same type.

The number of elements that can be stored in a array can be calculated as-

(upperbound-lowerbound)+1

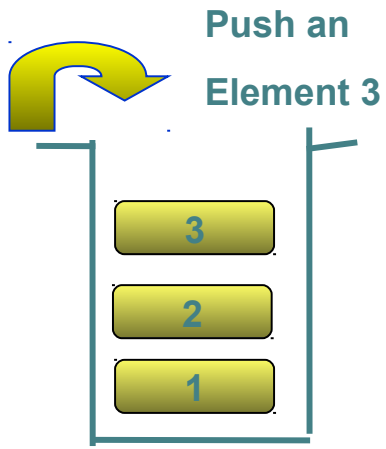
int a[10]

1	2	3	4	5	6	7	8	9	10

a[0] a[1] ----- ---------- ---------- -a[9]

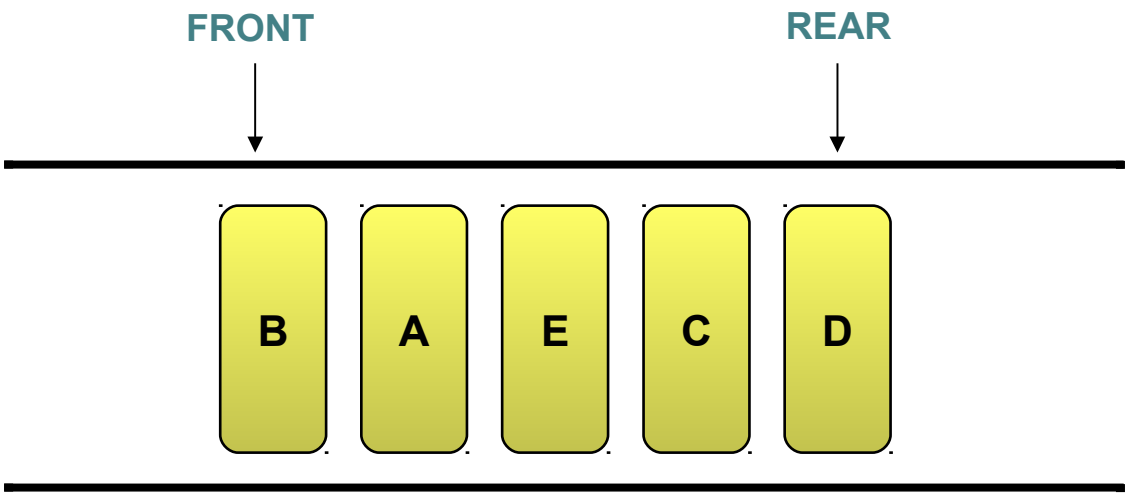
STACKS

Collection of elements like array with a special feature that deletion and insertion of elements can be done only from one end, called the TOP (Top of stack). Due to this property it is also called LIFO data structure i.e. **L**ast-**I**n-**F**irst-**O**ut data structure.



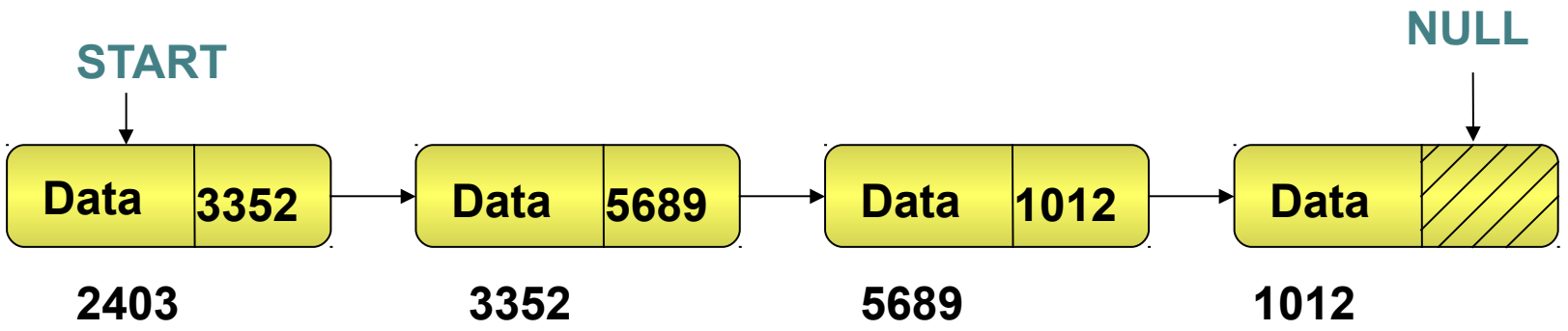
QUEUES

Queues are **F**irst -**I**n-**F**irst-**O**ut (FIFO) data. In a queue new elements are added to the queue from one end called Rear end, and the elements are deleted from another end called Front end. Eg. the queue in a ticket counter.



LINK LIST

◦A link list is a linear collection of elements called nodes. The linear order is maintained by pointers. Each node is divided into two or more parts; one data field and another pointer field.



STRUCTURE

- Collections of related variables (aggregates) of under one name.
- Can contain variables of different data types
- Commonly used to define records to be stored in files

Eg.

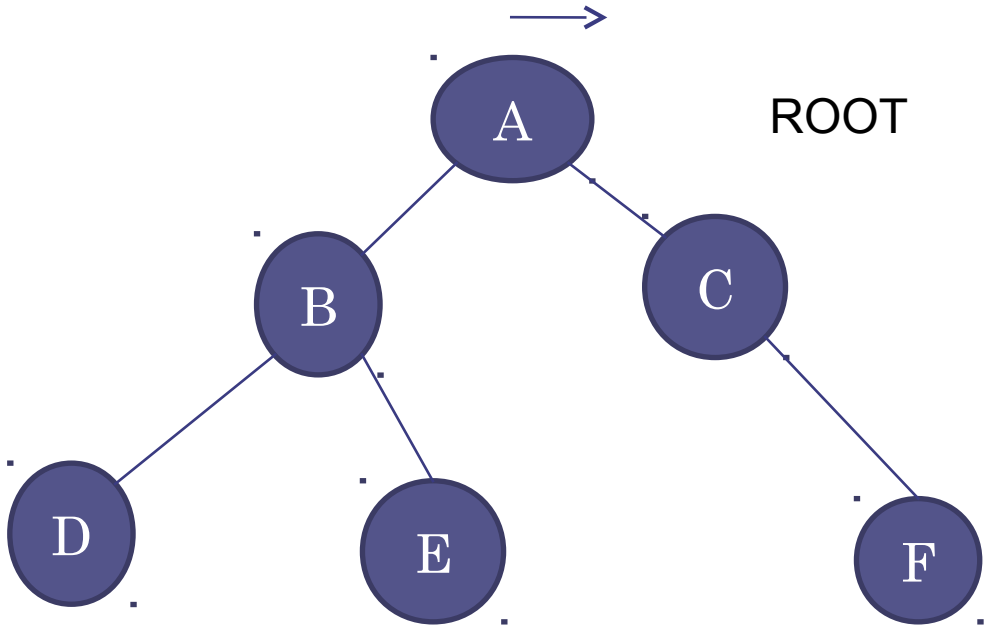
```
struct ADate
    {
int month;
int day;
int year;
} Date;
```

Types of Non Linear Lists

TREES

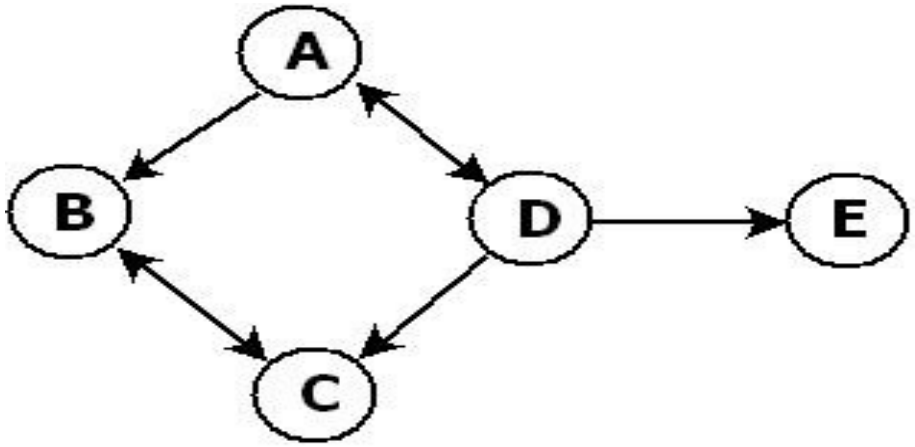
A tree can be defined as finite set of data items called nodes.

- Trees are non linear type of data structures in which data items are arranged or stored in a sorted sequence.
- It represents the hierarchical relationship between various elements.
- There is a special data item at the top called Root of tree.



GRAPHS

Graph is non-linear data structure capable of representing kinds of physical structures.
A graph G(V,E) is a set of vertices V and a set of edges E connecting vertices.



ALGORITHM

An ***Algorithm***, is a finite sequence of instructions, each of which has a clear meaning and can be executed with a finite amount of effort in finite time.

DESIRABLE FEATURES OF AN ALGORITHM

1. Each step of algorithm should be simple.

1. It should be unambiguous in the sense that the logic should be *crisp* and *clear*.

1. It should be effective i.e., it must lead to unique solution of the problem.

1. It must end in a finite number of steps.

1. It should be as efficient as possible.

CHARACTERISTICS OF AN ALGORITHM

1. **INPUT:** This part of the algorithm reads the data of the given problem.

1. **PROCESS:** This part of the algorithm does the required computations in simple steps.

1. **FINITENESS:** The algorithm must come to an end after a finite number of steps.

1. **EFFECTIVENESS:** Every step of the algorithm must be accurate and precise. It should also be executable within a desired period of time on the target machine.

1. **OUTPUT:** It must produce the desired output.

HOW TO DEVELOP AN ALGORITHM

1. Understand the problem.

1. Identify the **output** of the problem.

3. Identify **inputs** required by the problem and choose the associated data structure.

4. Design a **logic** that will produce the desired output from the given inputs.

5. **Test** the algorithm for different sets of input data.

6. Repeat steps 1 to 5 till the algorithm produces the desired results for all types of input and rules.

Describing an Algorithm

- State the name and inputs to the algorithm
  - e.g., Rectangle_area(length, width)
- Describe the computations using C-like language
  - Clarity trumps syntax: if the steps are clear, it does not matter if it is legal C syntax
    - E.g., Variables need not be declared if their content is clear

Test Driven Development

1) Write a test: Map inputs to outputs
2) Run all tests: Check tests
3) Write program: Add logic to map inputs to outputs
4) Run all test: Check logic
5) Refactor: Make logic clear

Example: Area of Rectangle

- Tests

Length	Width	Area
2	3	6
4	5	20

- Algorithm

```
Rectangle_Area(length, width)
{
      Return length * width
}
```

Example: Biggest

- Tests

Num1	Num2	Num3	Biggest
1	2	3	3
2	2	1	2
1	2	2	2

- Algorithm

```
Biggest(num1, num2, num3)
{
      If (num1 > num2 and num1 > num3)
            Return num1
      Else if (num2 > num1 and num2 > num3)
            Return num2
      Else if (num3 > num1 and num3 > num2)
            Return num3
      Else if (num1 > num2)
            Return num1
      Else
            Return num2
}
```

Example: Biggest in array

- Tests

a[0]	a[1]	a[2]	a[3]	a[4]	Biggest
1	2	3	4	5	5
5	4	3	2	1	5
1	3	2	5	4	5

- Algorithm

```
Biggest(a, size_of_a)
{
        Largest = a[0];
        for(i = a to size_of_a)
                If (Largest < a[i])
                        Largest = a[i]
        Return Largest
}
```

Stepwise Refinement: Top-Down

- Break a complex problem into smaller problems
  - Write a stub for each smaller problem
- Fill in algorithm for each sub-step.
  - Possibly using stepwise refinement for the sub-step

- Advantages
  - Bugs are easier to find
    - adding less code in each refinement
  - Functions might be reusable