DATA STRUCTURES USING C

MODULE1

Introduction – Basic Terms

Data

- A value or set of values.
- It specifies value of a variable or constant.
- Example: name,address

Data Item:- Single unit of values

- **Elementary Data :-** Data items that can't be divided into sub items. Eg: Mark, age
- **Grouped Data :-** Data items that can be divided into sub items. Eg; Address,name

Entity

- An entity is an object which has a set of properties called attributes.
- We can assign values to this attribute.
- Example: Employee

Entity Set

- Entities with similar attributes
- Example: Employees in an organization

Information

Meaningful data or processed data

Field:- A field is a single elementary unit of information representing an attribute of an entity.Eg: name,number,class,age,mark

Record:- a record is a collection of field values of a given entity.Eg: details of a single student(name,number,age,mark)

File:- a file is the collection of records of the entities in a given entity set.

Eg: If there are 50 students in a class, it will contains 50 records.

Primary key.:-Each record in a file may contain many field items, but the value in a certain field may uniquely determine the record in the file. Such a field is called primary key.

Data Structure Definitions

- A data structure is a group of data elements which comes under one name.

- Particular way of storing and organizing data in a computer so that it can be used efficiently.
- The logical or mathematical model of a particular organization of data.
- Data structures are building blocks of programs.
- Data Structure is a way to store and organize data so that it can be used efficiently.

Types of Data Structures

There are two types of data structures:

- Primitive data structure
- Non-primitive data structure

Primitive Data structure

- Consists of fundamental data types which are supported by a programming language.
- The int, char, float, double are the primitive data structures that can hold a single value.
- In C the basic data types used are

DATA TYPE	SIZE (IN BYTE)
char	1
short int	2
int	4
long int	4
float	4
double	8
long double	12
void	MEANING LESS

Non-primitive data structure

- Created using primitive data structures
- Example: Stack,Queue,Tree,Graph
- Non primitive data structures are divided into
  - Linear data structures
  - Non linear data structures

Linear Data Structures

- If the elements of the data structures are stored in a linear or sequential order then it is called as linear data structure.
- Example: Array, Stack , Queue, Linked list
- Stored in two ways
- Stored in sequential memory location
- Elements are connected using links

Non Linear Data Structures

- If the elements of a data structure are not stored in sequential order are called non linear data structure.
- Example: Tree,Graph

ARRAYS

- An Array is a finite ordered homogeneous set of elements.
- The elements of the array are stored in consecutive memory locations.
- Elements are accessed using an index or subscripts.

Syntax:

Data_type arrayname[max_size];

- Also called subscripted variables.
- Example : int a[10];
- Here a is the name of the array which can store a maximum of integer elements
- In C the index starts from zero. i.e, the elements are a[0] to a[9].

Memory Representation

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

STACK

- Linear data structure where insertion and deletion of elements are done at only one end which is known as the **top** of the stack.

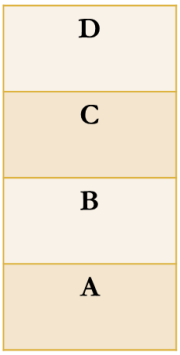
- STACK is a Last in First Out(LIFO) structure because the last element which is added to the stack is the first element which is deleted from the stack.
- A stack support two operations

1. **Push :** Inserting an element to the top of the stack.
2. **Pop :** Removing an element from the top of the stack.

- **A** Stack can be implemented using arrays or linked list.

Implementation of a stack using Array

- Let MAX be the number of elements in a stack.
- Let top represent the position of the top element in the stack.



Stack Overflow and Underflow conditions

- **If top=MAX-1 then the stack is full. This is called overflow condition.**
- **We cant insert an element if the stack is already full.**
- **If top=null or -1 then the stack is empty.**
- **We cant delete an element if the stack is empty(underflow condition)**

QUEUE

- **A queue is a First In First Out(FIFO) 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 rear end and are removed from the other end called front.**
- **Queues also can be implemented using arrays and linked list.**

- **Here front=0, rear=4**
- **If we want to add one more value to the queue then the rear is incremented by one.**

- **i.e; front=0 and rear=5**
- **If we want to delete one element from the queue then the value of front will be incremented by one.**
- **i.e; front=1 and rear=5**

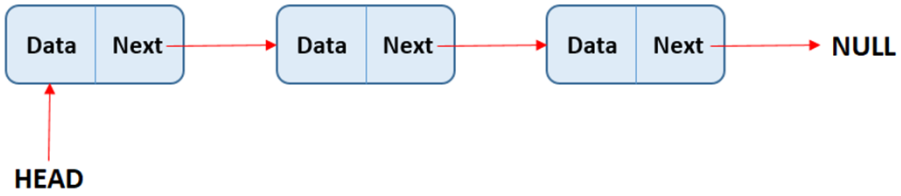
Queue

- **Overflow ……..> check whether rear=MAX-1(full)**
- **Underflow ……..> Queue is already empty.**

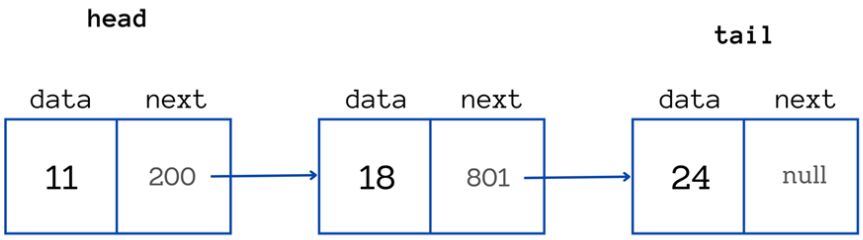
  front=null and rear=null.

Linked List

- **A linked list is a linear dynamic data structure.**
- **The elements in a linked list are called nodes.**
- **A node contains 2 parts :**
- **Data part: value stored in the node**
- **Link or pointers: Points to the next node in the list.**



- **A linked list is dynamic in nature because each node is allocated space when it is added to the list.**
- **The last node in the list contains a null pointer(tail of the list).**
- **Since the memory of the node is dynamically allocated to the total number of nodes that can be added to a list depends on the amount of memory available.**



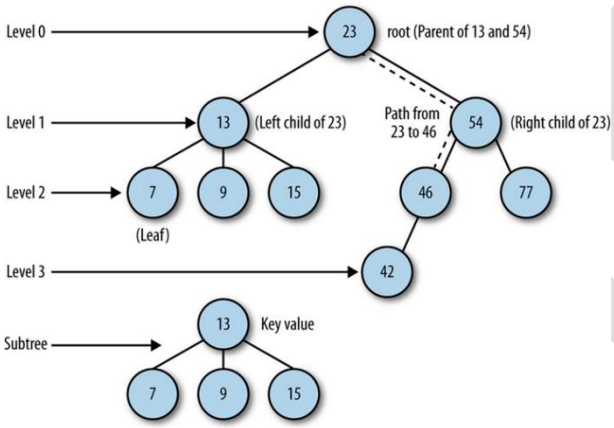
Non linear Data Structures

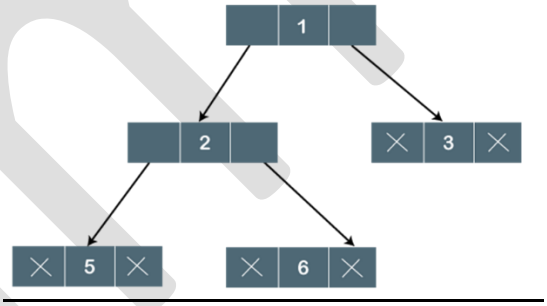
- **In non linear data structure the elements of a data structure are not stored in sequential order.**
- **Example : Tree, Graph**

Tree

- **A tree is non linear data structure which consists of collection of nodes arranged in a hierarchical order.**
- **One of the node is taken as the root node.**
- **The remaining nodes are partitioned into sub trees of the root.**
- **The simplest form of tree is a binary tree.**

Binary Trees



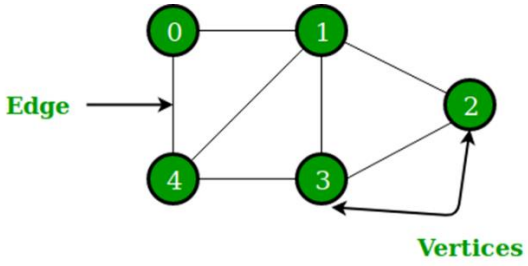


- In a tree parent node can have any number of children but a child can have only one parent.
- A binary tree consists of a root node and left and right sub trees are also binary trees.
- Each node consists of a data element, a left pointer which points to left sub tree and right pointer which points to right sub tree.
- The top most node is always the root.
- If root=NULL then the tree is empty.

Graph

- A graph is a non linear data structure which is a collection of vertices and edges that connect these vertices.
- A graph is similar to a tree but it can have more complex relationships.
- Here a parent node can be connected to any number of nodes and vice versa.
- Graphs do not have any root node.

- Every node in a graph can be connected to any other nodes in the graph.



Applications of Data Structures

- Data structures are used in almost every program or software system.
- Each field uses different type of data structure.
- It depends on how the data should be organized, accessed, inserted and deleted.
- Arrays, stacks, queues, linked lists, tree and graphs are used depending on the application.

Some of the common areas where data structures are applied:

- Compiler design
- DBMS
- Operating system
- Graphics
- Artificial Intelligence
- Numerical analysis
- Simulations

Arrays:- RDBMS, Matrices, Vectors used to implement stacks , queues , linked lists etc.

Linked List:- In web browsers, which creates a linked list of web pages visited.

Stack :- Recursion, Evaluation of expressions.

Queue:- CPU Scheduling, Disk Scheduling, waiting list for processes.

Tree:- DBMS hierarchical model, system directories, compiler design.

Graph:- DBMS network model.

Algorithm

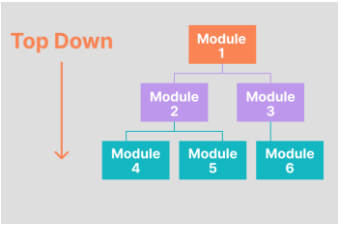
- An algorithm is a set of steps for solving a particular problem.
- The main characteristics of an algorithm are

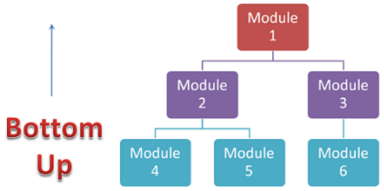
1. Each step should be simple and precise.

2. Each step should be finite, it should be carried out in a particular time.
3. An algorithm can accept zero or more inputs.
4. It should produce desired output.

Different approaches to Design an algorithm

- A complex algorithm can be divided into smaller units called modules.
- This process of dividing an algorithm into modules is called modularization.
- The main advantages of modularization are
  - Complex algorithm can be made simple to design and implement.
  - There are two ways of design an algorithm





Bottom Up approach

- It is the reverse of top down approach.
- Here first the basic modules are designed.
- Then these sub modules are grouped together to form a higher level module.
- The higher level modules are again grouped and this process is repeated until the complete algorithm is obtained.

Algorithm Complexity: Time-space trade off

- The efficiency of an algorithm depends upon two measures time and space.
- The complexity of an algorithm is a function **f(n)** which measures time and or space used by an algorithm in terms of the input size n.
- Time complexity means the time taken to run a program which is expressed as a function of input size.
- Space complexity means the amount of computer memory required during program execution expressed as a function of input size.
- We can express the time complexity in 3 ways.

1.Worst case running time

- This is the maximum time taken by an algorithm.

- For searching an algorithm if the item to be searched is the last element in an array, then it is called the **worst case**.
- Search reaches the upper bound.

2.Best case running time

- It is the shortest time taken.
- In a searching algorithm the element to be searched is the first element in the array.
- Search takes the lower bound.

3.Average case running time

- It is the running time of an average input.
- i.e the element may be around the middle of the array.
- Number of comparisons can be 1,2,3,…….n
- So probability p=1/n
- C(n) =1.1/n+2.1/n+……n.1/n=(1+2+3+…….n).1/n
- =n(n+1)/2*1/n=(n+1)/2

Rate of growth of functions

- M- algorithm
- n – size of input data
- Complexity f(n) increases as n increases
- Usually f(n) will be some standard functions such as $\log_2 n$, $n \log_2 n$, n^2, n^3, 2^n.

g(n) / n	log n	n	n log₂ n	n^2	n^3	2^n
5	3	5	15	25	125	32
10	4	10	40	100	10^3	10^3
100	7	100	700	10^4	10^6	10^{30}
1000	10	10^3	10^4	10^6	10^9	10^{300}

Asymptotic notations for complexity

- There are many asymptotic notations to represent the complexity of algorithms.

Big O notations (o)

- The Big O notation defines the upper bound of an algorithm, it bounds a function only from above.
- It describes the worst-case scenario.

- It can be used to describe the execution time required or the space taken by an algorithm.

Big O Notation

Formal Definition:

- Let f(n) and g(n) are functions defined on positive integers.
- F(n) is bounded by some multiple of g(n) for almost all n.
- Let n_0 be a positive integer and M be a positive number such that for all $n>n_0$, we have f(n)<=M.g(n), then we can write
- F(n) = O(g(n)) means that **f(n) is of order g(n).**

Omega notation(Ω)

- **Big Omega** is used to give a **lower bound** for the growth of a function.
- It's defined in the same way as Big O
- But with the inequality sign turned around. It takes the best case.

Formal Definition:

- Let f(n) and g(n) are functions defined on positive integers.
- F(n) is bounded by some multiple of g(n) for almost all n.
- Let n_0 be a positive integer and M be a positive number such that for all $n>n_0$, we have f(n)>=M.g(n), then we can write
- F(n)= Ω (g(n)) means that **f(n) is omega of g(n).**

Theta notation (θ)

- It is used when the function f(n) is bounded both from above and below by the function g(n).

Formal Definition:

- Let f(n) and g(n) are functions defined on positive integers.
- F(n) is bounded by some multiple of g(n) for almost all n.
- Let n_0 be a positive integers and C_1 and C_2 are two positive constants such that for all $n>n_0$, we have
- $C_1g(n)<=f(n)<=C_2g(n)$, then we can write
- F(n)= θ g(n)) means that **f(n) is theta of g(n)**
- Function g(n) is both an upper bound and lower bound for the function f(n).

- F(n) is such that f(n)=O(g(n)) and f(n)= Ω (g(n))

Little Oh notations(o)

- F(n) = o(g(n)) (read as f(n) is little oh of g(n)) iff
- f(n) = O(g(n))
- f(n) $\neq$ Ω (g(n))