PG-DAC SEPT-2021

ALGORITHMS & DATA STRUCTURES

SACHIN G. PAWAR

SUNBEAM INSTITUTE OF INFORMATION & TECHNOLOGIES

PUNE & KARAD

Data Structures: Introduction

Name of the Module : Algorithms & Data Structures Using Java.

Prerequisites: Knowledge of programming in C/C++/Java with object oriented concepts.

Weightage : 100 Marks (Theory Exam : 40% + Lab Exam : 40% + Mini Project : 20%).

 # Importance of the Module:

1. CDAC - Syllabus
2. To improve programming skills
3. Campus Placements
4. Applications in Industry work

Data Structures: Introduction

Q. Why there is a need of data structure?

- There is a need of data structure to achieve 3 things in programming:
    1. **efficiency**
    2. **abstraction**
    3. **reusability**

Q. What is a Data Structure?

Data Structure is **a way to store data elements into the memory** (i.e. into the main memory) in **an organized manner** so that operations like **addition, deletion, traversal, searching, sorting** etc... can be performed on it efficiently.

Data Structures: Introduction

Two types of **Data Structures** are there:

1. Linear / Basic data structures : data elements gets stored / arranged into the memory in a **linear manner** (e.g. sequentially) and hence can be accessed linearly / sequentially.

- **Array**
- **Structure & Union**
- **Class**
- **Linked List**
- **Stack**
- **Queue**

2. Non-Linear / Advanced data structures : data elements gets stored / arranged into the memory in a **non-linear manner** (e.g. hierarchical manner) and hence can be accessed non-linearly.

- **Tree (Hierarchical manner)**
- **Graph**
- **Hash Table(Associative manner)**
- **Binary Heap**

Data Structures: Introduction

+ Array: It is a **basic/linear data structure** which is **a collection/list of logically related similar type of data elements** gets stored/arranged into the memory at **contiguos locations**.

+ Structure: It is a **basic/linear data structure** which is **a collection/list of logically related similar and disimmilar type of data elements** gets stored/arranged into the memory **collectively i.e. as a single entity/record.**

sizeof of the structure = sum of size of all its members.

+ Union: Union is same like structure, except, memory allocation i.e. size of union is the size of max size member defined in it and that memory gets shared among all its members for effective memory utilization (can be used in a special case only).

Data Structures: Introduction

Q. What is a Program?

- A Program is **a finite set of instructions written in any programming language** (either in a high level programming language like C, C++, Java, Python or in a low level programming language like assembly, machine etc...) given to the machine to do specific task.

Q. What is an Algorithm?

- An algorithm is **a finite set of instructions written in any human understandable language (like english)**, if followed, acomplishesh a given task.

- **Pseudocode :** It is a **special form of an algorithm**, which is a finite set of instructions written in any human understandable language (like english) **with some programming constraints**, if followed, acomplishesh a given task.

- An algorithm is a template whereas a program is an implementation of an algorithm.

Data Structures: Introduction

 # Algorithm : to do sum of all array elements

Step-1: initially take value of sum is 0.

Step-2: traverse an array sequentially from first element till last element and add each array element into the sum.

Step-3: return final sum.

 # Pseudocode : to do sum of all array elements

```
Algorithm ArraySum(A, n){//whereas A is an array of size n
    sum=0;//initially sum is 0
    for( index = 1 ; index <= size ; index++ ) {
    sum += A[ index ];//add each array element into the sum
    }
    return sum;
}
```

Data Structures: Introduction

- There are two types of Algorithms OR there are two approaches to write an algorithm:

1. iterative (non-recursive) approach :

```
Algorithm ArraySum( A, n){//whereas A is an array of size n
    sum = 0;
    for( index = 1 ; index <= n ; index++ ){
          sum += A[ index ];
    }
       return sum;
}
```

```
for( exp1 ; exp2 ; exp3 ){
    statement/s
}
```

exp1 => initialization
exp2 => termination condition
exp3 => modification

Data Structures: Introduction

2. recursive approach:
While writing recursive algo: we need to take care about 3 things
1. initialization: at the time first time calling to recursive function
2. base condition/termination condition : at the begining of recursive function
3. modification: while recursive function call

Example:

```
Algorithm RecArraySum( A, n, index )
{
   if( index == n )//base condition
   return 0;

   return ( A[ index ] + RecArraySum(A, n, index+1) );
}
```

Data Structures: Introduction

Recursion : it is a process in which we can give call to the function within itself.

function for which recursion is used => recursive function

- there are two types of recursive functions:

1. tail recursive function : recursive function in which recursive function call is the last executable statement.

```
void fun( int n )
{
   if( n == 0 )
      return;

   printf(“%4d”, n);
   fun(n--);//rec function call
}
```

2. non-tail recursive function : recursive function in which recursive function call is not the last executable statement

```
void fun( int n )
{
   if( n == 0 )
      return;

   fun(n--);//rec function call
   printf("%4d", n);
}
```

Data Structures: Introduction

- An Algorithm is a solution of a given problem.
- Algorithm = Solution
- One problem may has many solutions.

For example: Problem => **Sorting :** to arrange data elements in a collection/list of elements either in an ascending order or in descending order.

A1 : Selection Sort
A2 : Bubble Sort
A3 : Insertion Sort
A4 : Quick Sort
A5 : Merge Sort
etc...

- When one problem has many solutions/algorithms, in that case we need to select an efficient solution/algo, and to decide efficiency of an algo's we need to do their analysis.

Data Structures: Introduction

- **Analysis of an algorithm** is a work of determining how much **time** i.e. computer time and **space** i.e. computer memory it needs to run to completion.

- There are two **measures** of an **analysis of an algorithms:**

1. Time Complexity of an algorithm is the amount of **time i.e. computer time** it needs to run to completion.

2. Space Complexity of an algorithm is the amount of **space i.e. computer memory** it needs to run to completion.

Data Structures: Introduction

 # Space Complexity of an algorithm is the amount of space i.e. computer memory it needs to run to completion.

Space complexity = code space + data space + stack space (applicable only for recursive algo)
code space = space required for an instructions
data space = space required for **simple variables, constants & instance variables.**
stack space = space required for **function activation records.**

- Space complexity has **two components:**
1. fixed component: data space (space required for simple vars & constants) and code space.
2. variable component : instance characteristics (i.e. space required for instance vars) and stack space (which is applicable only in recursive algorithms).

Data Structures: Introduction

 # Calculation of space complexity of non-recursive algo:

```
Algorithm ArraySum( A, n){//whereas A is an array of size n
   sum = 0;
   for( index = 1 ; index <= n ; index++ ){
      sum += A[ index ];
   }
   return sum;
}
```

Sp = data space + instance charactristics
simple vars => formal param: A & local vars: sum, index
constants : 0 & 1
instance variable = n, input size of an array = **n units**
data space = 3 units (for simple vars => A, sum & index) + 2 units (for constants => 0 & 1)
=> data space = **5 units**
Sp = (n + 5) units.

Data Structures: Introduction

S = C (code space) + Sp
S = C + (n+5)
S >= (n + 5) ... (as C is constant, it can be neglected)
S >= O(n) => O(n)
Space required for an algo = O(n) => whereas n = input size array.

 # Calculation of space complexity of recursive algorithm:

```
Algorithm RecArraySum( A, n, index ){
    if( index == n )//base condition
    return 0;
    return ( A[ index ] + RecArraySum(A, n, index+1) );
}
```

space complexity = code space + data space + stack space (applicable only in recursive algo)
code space = space required for instructions
data space = space required for variables, constants & instance characteristics
stack space = space required for FAR's.

Data Structures: Introduction

- When any function gets called one entry gets created onto the stack for that function call, referred as **function activation record / stack frame,** it contains **formal params, local vars, return addr, old frame pointer etc...**
In our example of recursive algorithm:
3 units (for A, index & n) + 2 units (for constants 0 & 1) = total 5 **units** of memory is required per function call.
- for size of an array = **n,** algo gets called **(n+1) no. of times.**
Hence, total space required = **5 * (n+1)**
S = 5n + 5
S >= 5n.
S >= 5n
S ~= 5n => O(n), wheras n = size of an array

Data Structures: Introduction

 # Time Complexity:
time complexity = compilation time + execution time
Time complexity has two components :
1. fixed component : compilation time
2. variable component : execution time => it depends on instance char of an algorithm.
Example :

```
Algorithm ArraySum( A, n){//whereas A is an array of size n
    sum = 0;
    for( index = 1 ; index <= n ; index++ ){
    sum += A[ index ];
    }

    return sum;
}
```

Data Structures: Introduction

- for size of an array = 5 => instruction/s inside for loop will execute 5 no. of times
- for size of an array = 10 => instruction/s inside for loop will execute 10 no. of times
- for size of an array = 20 => instruction/s inside for loop will execute 20 no. of times
- for size of an array = n => instruction/s inside for loop will execute n no. of times

 # Scenario-1

Machine-1 : Pentium-4 : Algorithm : input size = 10
Machine-2 : Core i5 : Algorithm : input size = 10

 # Scenario-2

Machine-1 : Core i5 : Algorithm : input size = 10 : system fully laoded with other processes
Machine-2 : Core i5 : Algorithm : input size = 10 : system not fully laoded with other processes.

- it is observed that, execution time is not only depends on instance chars, it also depends on some external factors like hardware on which algorithm is running as well as other conditions, and hence it is not a good practice to decide efficiency of an algo i.e. calculation of time complexity on the basis of an execution time and compilation time, and hence to do analysis of an algorithms **asymptotic analysis** is preferred.

Data Structures: Introduction

 # Asymptotic Analysis : It is a **mathematical way** to calculate time complexity and space complexity of an algorithm **without implementing it in any programming language.**

- In this type of analysis, analysis can be done on the basis of **basic operation** in that algorithm.

e.g. in searching & sorting algorithms **comparison** is the basic operation and hence analysis can be done on the basis of no. of comparisons, in addition of matrices algorithm **addition** is the basic operation and hence on the basis of addition operation analysis can be done.

"Best case time complexity": if an algo takes **min** amount of time to run to completion then it is referred as best case time complexity.

"Worst case time complexity": if an algo takes **max** amount of time to to run to completion then it is referred as worst case time complexity.

"Average case time complexity": if an algo takes **neither min nor max** amount of time to run to completion then it is referred as an average case time complexity.

Data Structures: Introduction

 # Asympotic Notations:

1. Big Omega (Ω) : this notation is used to denote **best case time complexity** – also called as **asymptotic lower bound,** running time of an algorithm cannot be less than its asymptotic lower bound.

2. Big Oh (O) : this notation is used to denote **worst case time complexity** - also called as **asymptotic upper bound,** running time of an algorithm cannot be more than its asymptotic upper bound.

3. Big Theta (θ) : this notation is used to denote an **average case time complexity** - also called as **asymptotic tight bound,** running time of an algorithm cannot be less than its asymptotic lower bound and cannot be more than its asymptotic upper bound i.e. it is **tightly bounded.**

Data Structures: Searching Algorithms

1. Linear Search / Sequential Search:

 # Algorithm :

Step-1 : accept key from the user

Step-2 : start traversal of an array and compare value of the key with each array element sequentially from first element either till match is not found or max till last element, if key is matches with any of array element then return true otherwise return false if key do not matches with any of array element.

```
# Pseudocode:
Algorithm LinearSearch(A, size, key){
   for( int index = 1 ; index <= size ; index++ ){
   if( arr[ index ] ==  key )
   return true;
   }
   return false;
}
```

Data Structures: Searching Algorithms

Best Case: If key is found at very first position in only 1 no. of comparison then it is considered as a best case and running time of an algorithm in this case is **O(1)** => and hence time complexity = **Ω(1)**

Worst Case: If either key is found at last position or key does not exists, in this case maximum **n** no. of comparisons takes place, it is considered as a worst case and running time of an algorithm in this case is **O(n)** => and hence time complexity = **O(n)**

Average Case: If key is found at any in between position it is considered as an average case and running time of an algorithm in this case is **O(n/2) => O(n)** => and hence time complexity = **θ(n)**