Data Structures and Algorithms

Sankita J Patel
sjp@coed.svnit.ac.in

Introduction to Data Structures

- Data Structure
  - A systematic way to organize data in order to use it efficiently.
- A few terms
  - Interface
    - Set of operations that a data structure supports.
    - An interface only provides the list of supported operations, type of parameters they can accept and return type of these operations.
  - Implementation
    - The internal representation of a data structure.
    - The definition of the algorithms used in the operations of the data structure.

Characteristics of a Data Structure

- Correctness
  - Data Structure implementation should implement its interface correctly.
- Time Complexity
  - Running time or execution time of operations of data structure must be as small as possible.
- Space Complexity
  - Memory usage of a data structure operation should be as little as possible.

Need for Data Structures

- **Data Search**
  - Consider searching of a particular item from an inventory of 1 million(10^6) items of a store.
- **Processor speed**
  - Processor speed although being very high, falls limited if data grows to billion records.
- **Multiple requests**
  - As thousands of users can search data simultaneously on a web server, even very fast server fails while searching the data.

Data Structures - Algorithms Basics

- Algorithm
  - A step by step procedure, which defines a set of instructions to be executed in certain order to get the desired output.
  - Independent of underlying languages.
- A few important categories of algorithms from data structure point of view
  - **Search** − Algorithm to search an item in a data structure.
  - **Sort** − Algorithm to sort items in certain order
  - **Insert** − Algorithm to insert item in a data structure
  - **Update** − Algorithm to update an existing item in a data structure
  - **Delete** − Algorithm to delete an existing item from a data structure

Characteristics of an Algorithm

- Unambiguous
  - Each of its steps (or phases), and their input/outputs should be clear and must lead to only one meaning.
- Input
  - Should have 0 or more well defined inputs.
- Output
  - Should have 1 or more well defined outputs, and should match the desired output.
- Finiteness
  - Must terminate after a finite number of steps.
- Feasibility
  - Should be feasible with the available resources.
- Independent
  - An algorithm should have step-by-step directions which should be independent of any programming code.

How to write algorithm

- No well-defined standards for writing algorithms
  - Rather, it is problem and resource dependent
  - Never written to support a particular programming code
- Basic code constructs like loops (do, for, while), flow-control (if-else) etc. *can* be used to write an algorithm.
- We write algorithms in step by step manner, but it is not always the case.
  - Algorithm writing is a process and is executed after the problem domain is well-defined.
  - That is, we should know the problem domain, for which we are designing a solution.

How to write algorithm…

- Algorithm to add two numbers and display result (method 1)

step 1 – START
step 2 – declare three integers a, b & c
step 3 – define values of a & b
step 4 – add values of a & b
step 5 – store output of step 4 to c
step 6 – print c
step 7 – STOP

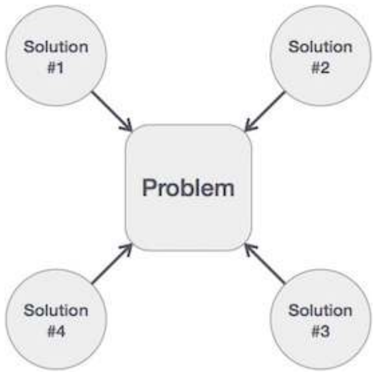
How to write algorithm…

- Algorithm to add two numbers and display result (method 2)
- We shall use method 2 to throughout the course

step 1 – START ADD
step 2 – get values of a & b
step 3 – c ← a + b
step 4 – display c
step 5 – STOP

Algorithm analysis

- What if there are multiple solutions to a problem ??



Algorithm analysis…

- Efficiency of an algorithm can be analyzed at two different stages:
  - A priori analysis
    - Theoretical analysis of an algorithm
    - Efficiency is measured by assuming that all other factors e.g. processor speed, are constant and have no effect on implementation
  - A posterior analysis
    - Empirical analysis of an algorithm
    - The selected algorithm is implemented using programming language
    - This is then executed on target computer machine
    - In this analysis, actual statistics like running time and space required, are collected.

Algorithm analysis…

- Algorithm Complexity
  - Suppose X is an algorithm and n is the size of input data, the time and space used by the Algorithm X are the two main factors which decide the efficiency of X.
- Time Factor
  - The time is measured by counting the number of key operations
  - E.g. comparisons in sorting algorithm
- Space Factor
  - The space is measured by counting the maximum memory space required by the algorithm.
- The complexity of an algorithm f(n) gives the running time and / or storage space required by the algorithm in terms of n as the size of input data.

Algorithm analysis…

- Space Complexity
  - represents the amount of memory space required by the algorithm in its life cycle.
  - Space required by an algorithm is equal to the sum of the following two components –
    - A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example simple variables & constant used, program size etc.
    - A variable part is a space required by variables, whose size depends on the size of the problem. For example dynamic memory allocation, recursion stack space etc.
- Space complexity **S(P)** of any algorithm **P** is **S(P) = C + Sp(I)** Where **C** is the fixed part and **Sp(I)** is the variable part of the algorithm which depends on instance characteristic **I**.

Algorithm analysis…

- Space Complexity
  - S(P)=1+3
  - Depends on the type of variables and constants and multiplied accordingly

Algorithm: SUM(A, B)
Step 1 – START
Step 2 - C ← A + B + 10
Step 3 - Stop

Algorithm analysis…

- Time Complexity
  - represents the amount of time required by the algorithm to run to completion
  - Can be defined as a numerical function **T(n)**, where **T(n)** can be measured as the number of steps, provided each step consumes constant time.
- For example, addition of two n-bit integers takes n steps. Consequently, the total computational time is **T(n) = c*n**, where **c** is the time taken for addition of two bits. Here, we observe that **T(n)** grows linearly as input size increases.

Data Structures: Basic Concepts

- Data types
  - A way to classify various types of data such as integer, string etc.
  - Determines,
    - the values that can be used with the corresponding type of data
    - the type of operations that can be performed on the corresponding type of data
- Categories of Data type
  - Built-in Data Type
  - Derived Data Type

Data types…

- Built-in Data Type
  - Data types for which a language has built-in support, e.g.
    - Integers
    - Boolean (true, false)
    - Floating (Decimal numbers)
    - Character and Strings
- Derived Data Type
  - Data types which are implementation independent
  - Normally built by combination of primary or built-in data types and associated operations on them. E.g.,
    - List
    - Array
    - Stack
    - Queue

Basic Operations

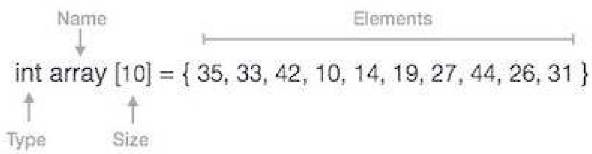
- Traversing
- Searching
- Insertion
- Deletion
- Sorting
- Merging

Data Structure: Arrays

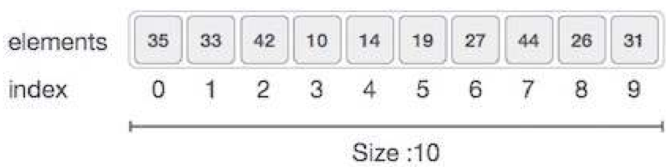
- Array is a container which can hold fix number of items
- These items should be of same type.
- Most of the data structures make use of array to implement their algorithms.
- Important terms to understand the concepts of Array are,
  - Element
    - Each item stored in an array is called an element.
  - Index
    - Each location of an element in an array has a numerical index which is used to identify the element.

Data Structure: Arrays

- Array declaration



- Array representation



Arrays: Basic Operations

- Traverse
  – print all the array elements one by one.
- Insertion
  – add an element at given index.
- Deletion
  – delete an element at given index.
- Search
  – search an element using given index or by value.
- Update
  – update an element at given index.

Arrays: Basic Operations…

- Insertion operation
  – Insertion at the beginning of an array
  – Insertion at the end of an array
  – Insertion at given index of an array
  – Insertion after the given index of an array
  – Insertion before the given index of an array

Arrays: Insertion at beginning of an array

- A: an array with N elements
- MAX: The maximum numbers of elements A can store
- First check if array has any empty space to store any element and then proceed with insertion of ITEM.

INSERT_BEG(A,N,MAX, ITEM)

1. START
2. IF N = MAX -1
3. THEN return
4. ELSE N ← N + 1
5. ENDIF
6. FOR i←N-2 downto 0
7. DO A[i+1] ←A[i]
8. ENDFOR
9. A[0] ← ITEM
10. STOP

Arrays: Insertion at the end of an array

- A: an array with N elements
- MAX: The maximum numbers of elements A can store

INSERT_END(A,N,MAX, ITEM)

1. START
2. IF N = MAX
3. THEN return
4. ELSE N ← N + 1
5. A[N] ← ITEM
6. STOP

Arrays: Insertion at given index of an array

- A: array with N elements
- K: positive integer such that K<=N
- ITEM: data item to be inserted is inserted into the K^{th} position of A

INSERT(A,N,MAX, ITEM,K)

1. START
2. J ← N
3. N ← N+1
4. Repeat steps 5 and 6 while J >= K
5. A[J+1] ← A[J]
6. J ← J-1
7. A[K] ← ITEM
8. STOP

Arrays: Insertion after given index of an array :Tutorial

Arrays: Insertion after given index of an array :Tutorial Solution

- A: array with N elements
- K: positive integer such that K<=N
- ITEM: data item to be inserted is inserted into the K^{th} position of A

INS_AFTER_IND(A,N,MAX, ITEM,K)
1. START
2. IF N = MAX
3. THEN return
4. ELSE N ← N + 1
5. FOR i=N-1 downto K +1
6. DO A[i+1]← A[i]
7. A[K+1]←ITEM
8. STOP

Arrays: Insertion before given index of an array :Tutorial

Arrays: Insertion before given index of an array :Tutorial

- A: array with N elements
- K: positive integer such that K<=N
- ITEM: data item to be inserted is inserted into the K^{th} position of A

INS_BEFORE_IND(A,N,MAX, ITEM,K)
1. START
2. IF N = MAX
3. THEN return
4. ELSE N ← N + 1
5. FOR i=N-1 downto K-1
6. DO A[i+1]← A[i]
7. A[K-1]←ITEM
8. STOP

Arrays: Deletion

- Removing an existing element from the array and re-organizing all elements of an array

A: array with N elements
K: positive integer such that K<=N

DELETE(A,N,K)
1. START
2. FOR i←K+1 to N-1
3. DO A[i-1]← A[i]
4. STOP

Arrays: Searching

- Searching of an array element based on its value or index

A: array with N elements
ITEM: element with ITEM to be found

SEARCH(A,N,ITEM)
1. START
2. FOR i←0 to N-1
3. IF A[i]= ITEM
4. THEN RETURN i
5.
6. STOP

Arrays: Update : Tutorial

- Update an array element at kth position with new value

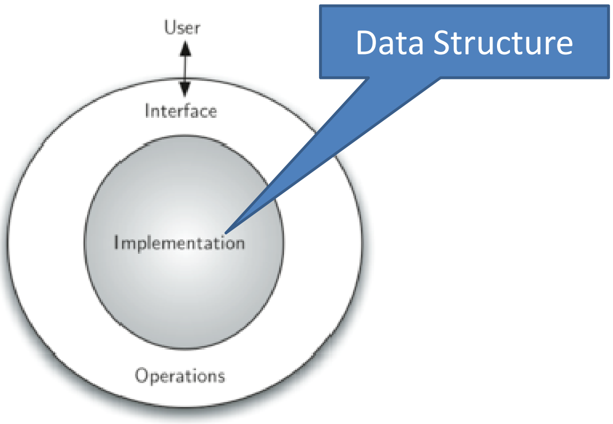
A: array with N elements
ITEM: new element
K: index

DATA STRUCTURE: STACK

Abstract Data Type (ADT)

- What is procedural abstraction ?
- Data Abstraction
  - Hiding implementation details of data and operations
- Abstract Data Type (ADT)
  - a logical description of how we view the data and the operations that are allowed without regard to how they will be implemented.
  - We are encapsulating the data or provide information hiding

Abstract Data Type (ADT)…



Abstract Data Type (ADT)…

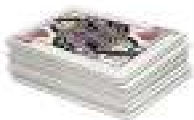
- Tutorial : List out advantages of ADT.
- Tutorial : Give a few examples of ADT that you have come across.

Stack

- A stack is an abstract data type (ADT), commonly used in most programming languages.
- Named stack as it behaves like a real-world stack, for example − deck of cards or pile of plates etc.

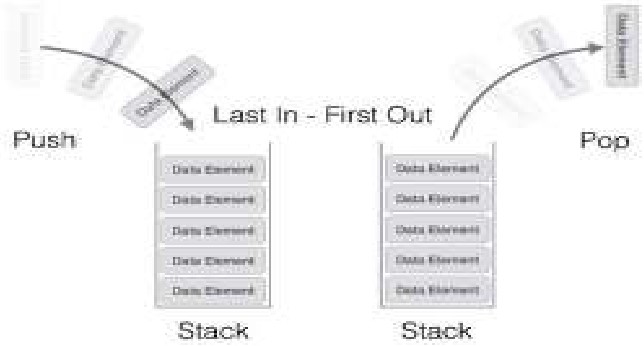
Stack…

- A real-world stack allows operations at one end only.
- For example, we can place or remove a card or plate from top of the stack only.
- Similar concept for Stack ADT. We can access only top element of stack.



Stack…

- LIFO (Last In First Out) data structure
- Insertion operation : PUSH
- Removal operation: POP



Stack representation

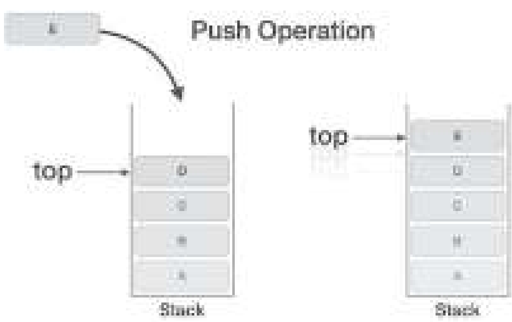
- A stack can be implemented by means of Array, Structure, Pointer and Linked-List.
- Stack can either be a fixed size one or it may have a sense of dynamic resizing.
- Stack using arrays
  - a fixed size stack implementation.

Stack operations

- Push() : Insert an element at the top of the stack
- Pop(): Removes a top element of the stack
- IsEmpty(): Checks if the stack is empty or not
- IsFull(): Checks if the stack is full or not
- A pointer to the last element inserted to the stack : TOP

Stack operation: Push()

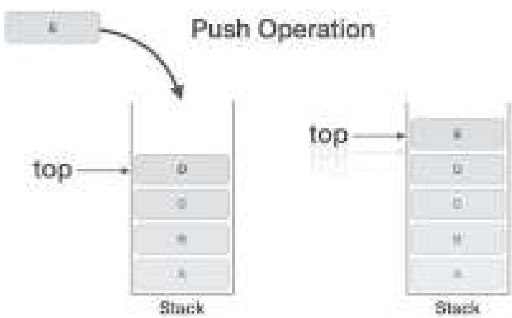
- **Step 1** – Check if stack is full.
- **Step 2** – If stack is full, produce error and exit.
- **Step 3** – If stack is not full, increment **TOP** to point next empty space.
- **Step 4** – Add data element to the stack location, where top is pointing.
- **Step 5** – return success.



Stack operation: Push()

Push(stack, data)

1. START
2. If stack is full
3. Then display "stack is full"
4. Return 0
5. TOP←TOP +1
6. stack[TOP]← data
7. Return stack



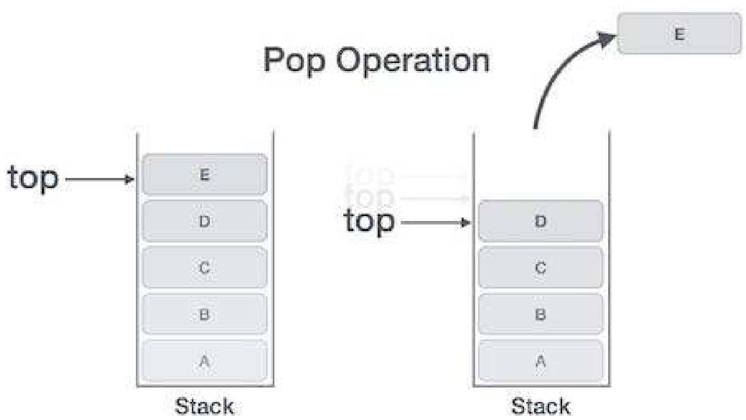
Stack operation:Pop()

- Accessing the content while removing it from stack, is known as pop operation.
- In array implementation of pop() operation, data element is not actually removed, instead **TOP** is decremented to a lower position in stack to point to next value.
- But in linked-list implementation, pop() actually removes data element and deallocates memory space.

Stack operation:Pop()…

- **Step 1** – Check if stack is empty.
- **Step 2** – If stack is empty, produce error and exit.
- **Step 3** – If stack is not empty, access the data element at which **top** is pointing.
- **Step 4** – Decrease the value of top by 1.
- **Step 5** – return success.

Stack operation:Pop()…



Stack operation:Pop()…

Pop(stack)

1. START
2. If stack is
3. Then display "Empty stack"
4. Return 0
5. TOP←TOP -1
6. Return stack

Stack operations

- Tutorial:
  - Write algorithms for the stack operations IsEmpty() and IsFull()