Data Structures

Important Points

→ Big Theta notation is used to denote (tight) asymptotic bound (upper and lower bound)

→ Big Omega notation is used to denote asymptotic lower bound.

→ Big O notation is used to denote an asymptotic upper bound.

Stack ADT operations

- Push - public
- Pop - public
- Peek - public
- isFull - public or private
- isEmpty - public or private
- Size - public or private

Internal data structure should be either array or linked list or both.

→ The main memory is organized for access and storage in row major order and column major order.

→

Binary search	↔	Log base 2
Last in first out	↔	Recursion

→

Modularity	↔	Testing of modules separate from rest of the system
Encapsulation	↔	Independence of module's behaviour
Separation of concerns	↔	Module responsible for one and only one feature
Abstraction	↔	Hiding details of the implementation

→ Stack ADT supports LIFO/FILO operation

→

Peek	⟷	Return value of latest push operation
Top of Stack	⟷	Latest insertion in the stack
Pop	⟷	Decrement top of stack
Push	⟷	Increment top of stack

→ Queue supports FIFO/LILO model.

→ Like circular queue, implementing a circular stack is possible.

→ Having elements of a queue at contiguous locations does not reduce the time complexity of its operations.

→ Stack is better suited than queue for reversing the order of elements that are sorted.

→

Null pointer	⟷	tail → next
Traversing a linked list	⟷	strictly linear order
Contiguous memory locations	⟷	Access via indexing
Element of type struct	⟷	both Linked List and Array:

→ Space Complexity : Memory space required by an algorithm in its execution cycle. Space needed by an algorithm is equal to the sum of the following two components :

- Fixed part : Space required to store data and variables that are not dependent of the size of input.
- Variable part : Space required by variables, whose size is dependent on the size of the problem.
- Space complexity is usually denoted by Big O notation.

→ Postfix Examples

1. 62+8/23+5/*

Infix method

= (62+)8/23+5/*
= (6+2)8/23+5/*
= (88/)23+5/*
= (8/8)23+5/*
= 1(23+)5/*
= 1(2+3)5/*
= 1(55/)*
= 1(5/5)*
= 11* = 1 //

2. 62/7*23-5+*

Infix method

= (62/)7*23-5+*
= (6/2)7*23-5+*
= (37*)23-5+*
= (3*7)(23-)5+*
= 21(2-3)5+*
= 21(-15+)*
= 21(-1+5)*
= 214*
= 21*4
= 84 //

3. 46+7+9+2-

Infix method

= (46+)7+9+2-
= (4+6)7+9+2-
= (107+)9+2-
= (10+7)9+2-
= (179+)2-
= (17+9)2-
= (262-)
= 26-2 = 24 //

4. 32*955+2//+

Infix method

= (32*)955+2//+

= (3*2)9(55+)2//+

= 69(5+5)2//+

= 69(102/)/+

= 69(10/2)/+

~~= 6(9/5)÷~~

= 6(95/)+

= 6(9/5)+

= (61.8)+

= (6+1.8) = 7.8//

Lookup operation

2D Array & 3D Array

R = Total Rows

C = Total Columns

D = Total Depth

arr2[r][c],

index = r*C + c

arr3[d][r][c],

index = d*R*C+
r*C+c

r = Current row
(range 0 to R-1)

c = Current column
(range 0 to C-1)

d = current depth
(range 0 to D-1)