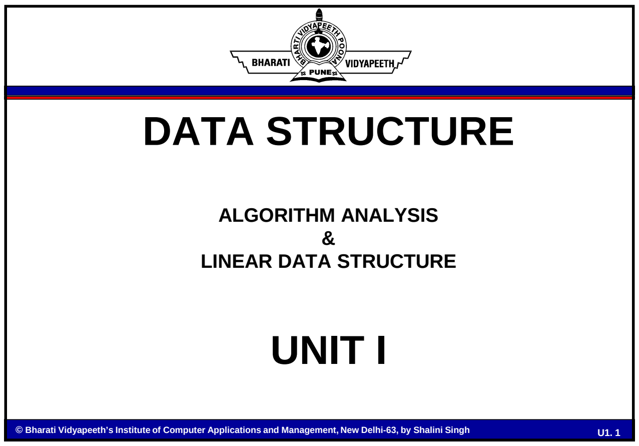
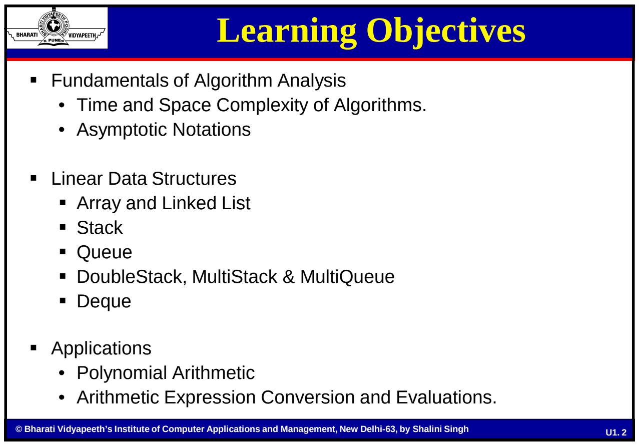


DATA STRUCTURE

ALGORITHM ANALYSIS
&
LINEAR DATA STRUCTURE

UNIT I

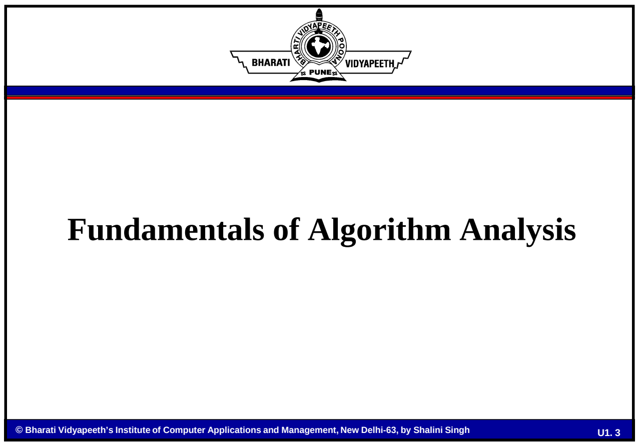
U1. 1



Learning Objectives

- Fundamentals of Algorithm Analysis
  - Time and Space Complexity of Algorithms.
  - Asymptotic Notations
- Linear Data Structures
  - Array and Linked List
  - Stack
  - Queue
  - DoubleStack, MultiStack & MultiQueue
  - Deque
- Applications
  - Polynomial Arithmetic
  - Arithmetic Expression Conversion and Evaluations.

U1. 2



Fundamentals of Algorithm Analysis

U1. 3

Objective

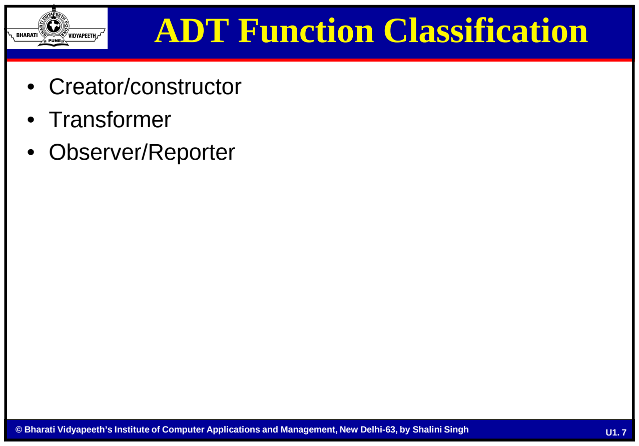
- How to create program
- ADT
- Space Complexity
- Time Complexity
- Common Growth Rate of Complexity
- Asymptotic Notation
- Algorithm Analysis

How to Create Programs

- Requirements
- Analysis: bottom-up vs. top-down
- Design: data objects and operations
- Refinement and Coding
- Verification
  - Program Proving
  - Testing
  - Debugging

Data Type

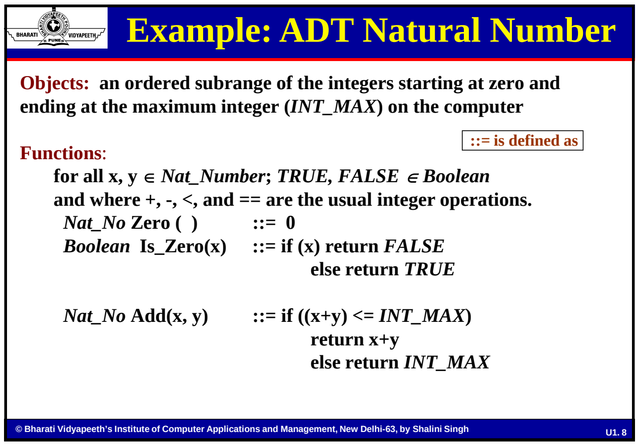
- Data Type
  - A *data type* is a collection of *objects* and a set of *operations* that act on those objects.
- Abstract Data Type
  - A set of data values and associated operations that are precisely specified independent of any particular implementation.



ADT Function Classification

- Creator/constructor
- Transformer
- Observer/Reporter

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 7



Example: ADT Natural Number

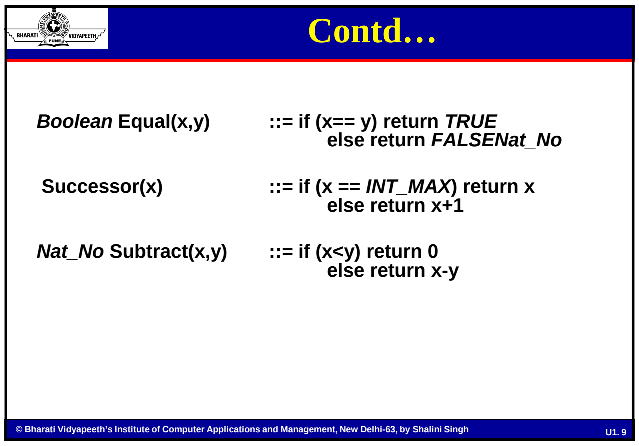
Objects: **an ordered subrange of the integers starting at zero and ending at the maximum integer (*INT_MAX*) on the computer**

::= is defined as

Functions:

```
for all x, y ∈ Nat_Number; TRUE, FALSE ∈ Boolean
and where +, -, <, and == are the usual integer operations.
  Nat_No Zero ( )        ::=  0
  Boolean  Is_Zero(x)    ::= if (x) return FALSE
                                else return TRUE

  Nat_No Add(x, y)       ::= if ((x+y) <= INT_MAX)
                                return x+y
                                else return INT_MAX
```

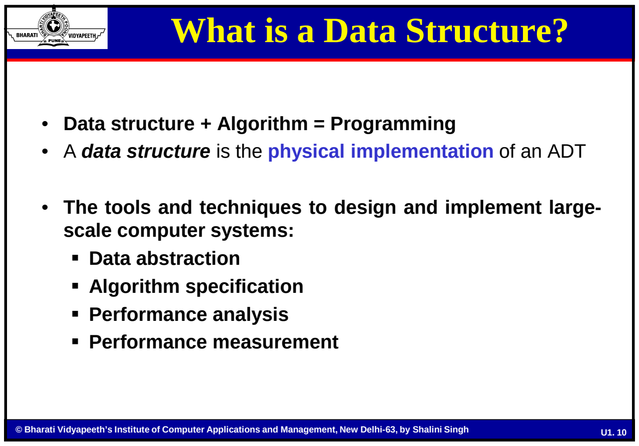
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 8



Contd…

```
Boolean Equal(x,y)        ::= if (x== y) return TRUE
                                 else return FALSENat_No

 Successor(x)             ::= if (x == INT_MAX) return x
                                 else return x+1

Nat_No Subtract(x,y)      ::= if (x<y) return 0
                                 else return x-y
```

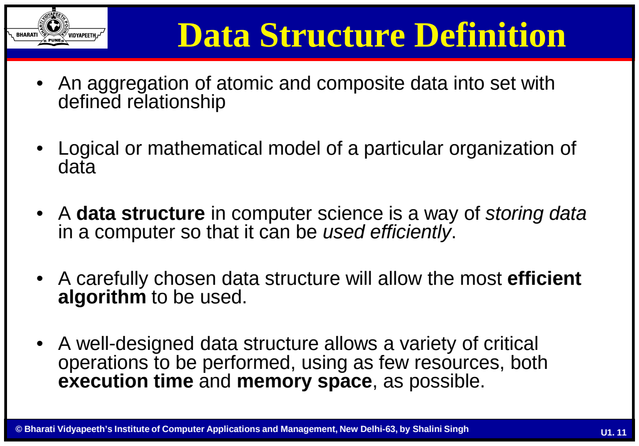
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 9



What is a Data Structure?

- **Data structure + Algorithm = Programming**
- A ***data structure*** is the **physical implementation** of an ADT
- **The tools and techniques to design and implement large-scale computer systems:**
  - **Data abstraction**
  - **Algorithm specification**
  - **Performance analysis**
  - **Performance measurement**

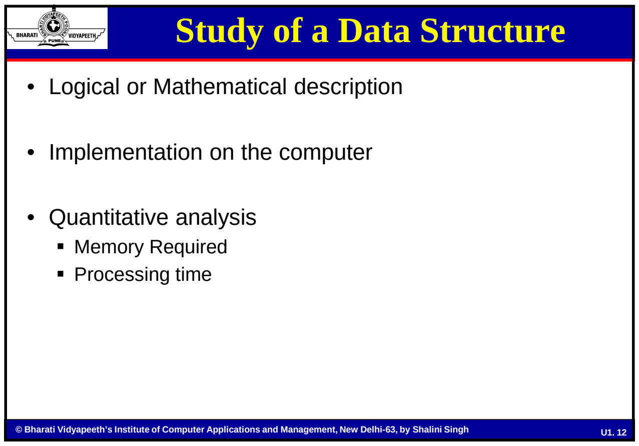
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 10



Data Structure Definition

- An aggregation of atomic and composite data into set with defined relationship
- Logical or mathematical model of a particular organization of data
- A **data structure** in computer science is a way of *storing data* in a computer so that it can be *used efficiently*.
- A carefully chosen data structure will allow the most **efficient algorithm** to be used.
- A well-designed data structure allows a variety of critical operations to be performed, using as few resources, both **execution time** and **memory space**, as possible.

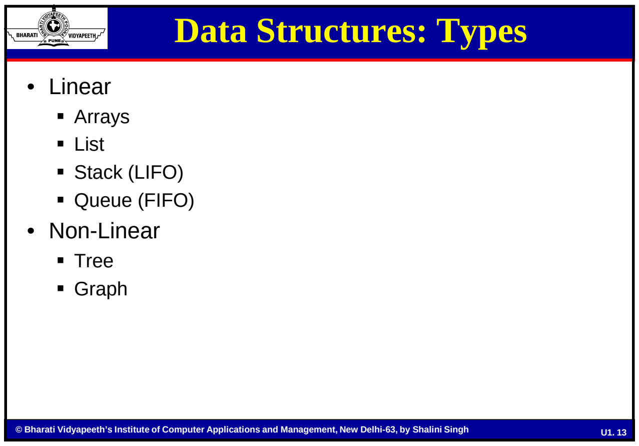
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 11



Study of a Data Structure

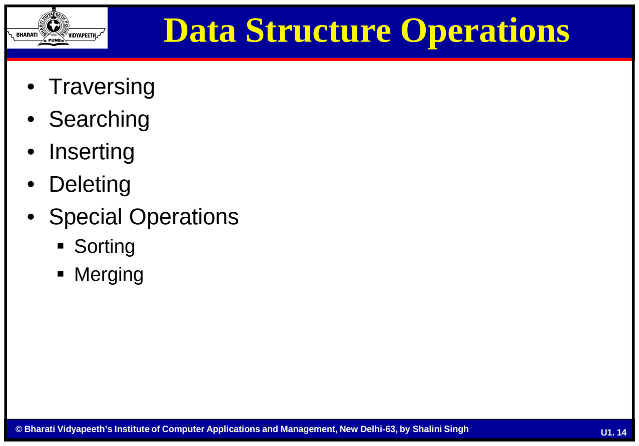
- Logical or Mathematical description
- Implementation on the computer
- Quantitative analysis
  - Memory Required
  - Processing time

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 12



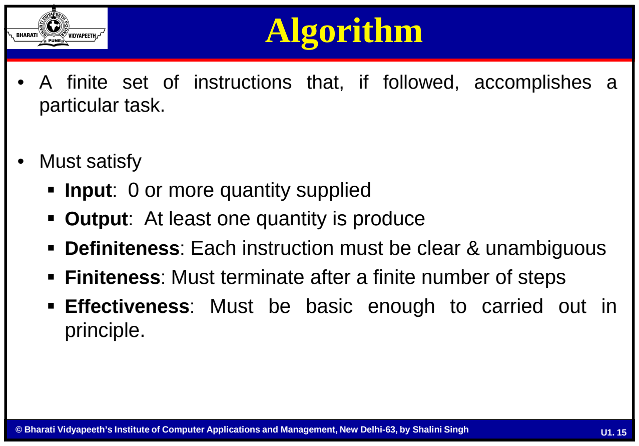
Data Structures: Types

- Linear
  - Arrays
  - List
  - Stack (LIFO)
  - Queue (FIFO)
- Non-Linear
  - Tree
  - Graph



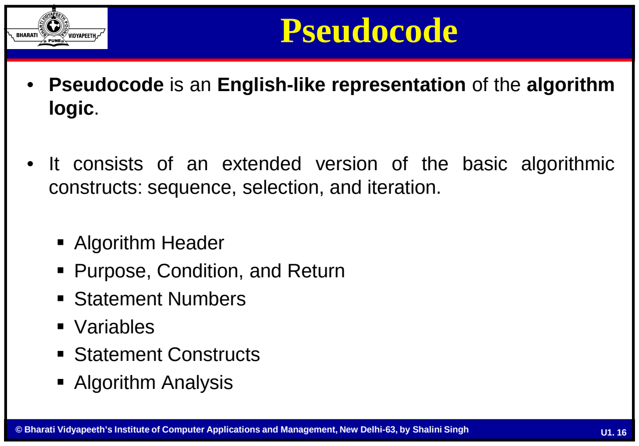
Data Structure Operations

- Traversing
- Searching
- Inserting
- Deleting
- Special Operations
  - Sorting
  - Merging



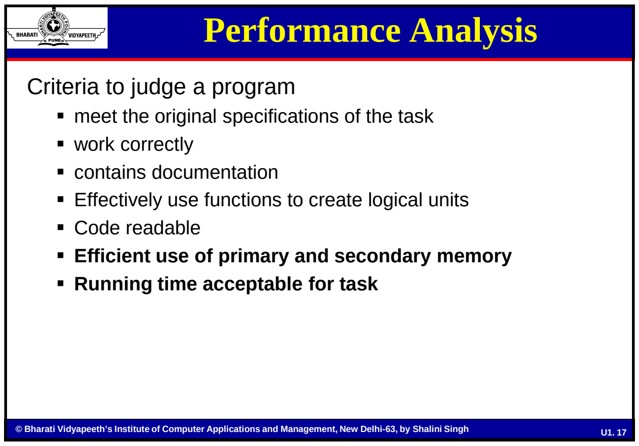
Algorithm

- A finite set of instructions that, if followed, accomplishes a particular task.
- Must satisfy
  - **Input**: 0 or more quantity supplied
  - **Output**: At least one quantity is produce
  - **Definiteness**: Each instruction must be clear & unambiguous
  - **Finiteness**: Must terminate after a finite number of steps
  - **Effectiveness**: Must be basic enough to carried out in principle.



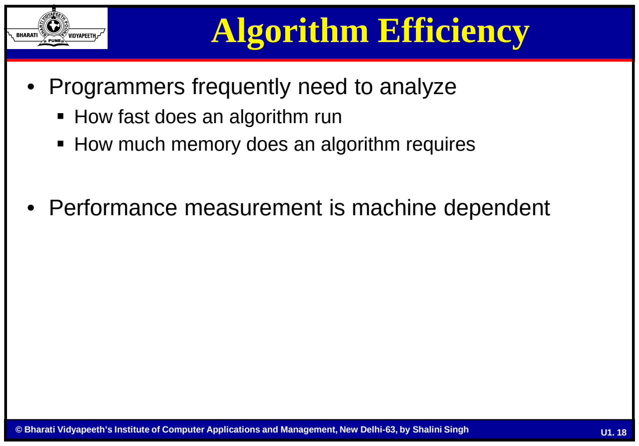
Pseudocode

- **Pseudocode** is an **English-like representation** of the **algorithm logic**.
- It consists of an extended version of the basic algorithmic constructs: sequence, selection, and iteration.
  - Algorithm Header
  - Purpose, Condition, and Return
  - Statement Numbers
  - Variables
  - Statement Constructs
  - Algorithm Analysis



Performance Analysis

Criteria to judge a program
- meet the original specifications of the task
- work correctly
- contains documentation
- Effectively use functions to create logical units
- Code readable
- **Efficient use of primary and secondary memory**
- **Running time acceptable for task**



Algorithm Efficiency

- Programmers frequently need to analyze
  - How fast does an algorithm run
  - How much memory does an algorithm requires
- Performance measurement is machine dependent

Different Aspects of Program Efficiency

Need to analyze algorithms for

- Correctness
- Efficiency

Efficiency is measured in terms of

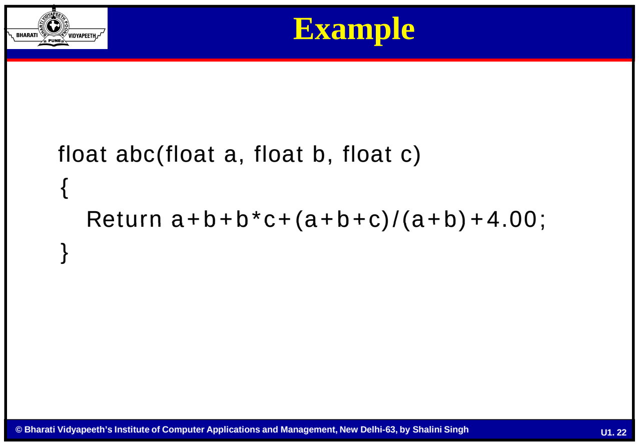
- Space utilization
- Time Utilization

Complexity Definition

- Space complexity of a program is:
  - the amount of memory that it needs to run to completion
- Time complexity of a program is:
  - the amount of computer time that it needs to run to completion

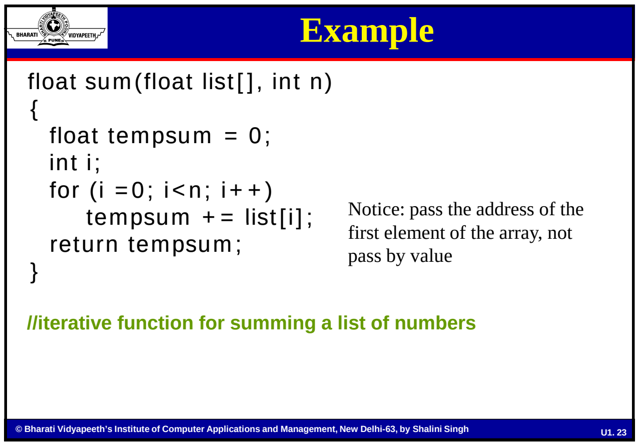
Space Complexity

- Fixed component c
  - ✓Independent of characteristics of I/O
  - ✓Includes
    - Instruction space
    - Space for simple variables
    - Space for constants; etc.
- Instance dependent component $S_p(I)$
  - ✓Dependent upon the particular problem instance being solved.
  - ✓Include
    - Dynamically created variables
    - Recursion stack space
- Total space requirement S(P) for any program
  - ✓$S(P) = c + S_p(I)$



Example

```
float abc(float a, float b, float c)
{
   Return a+b+b*c+(a+b+c)/(a+b)+4.00;
}
```

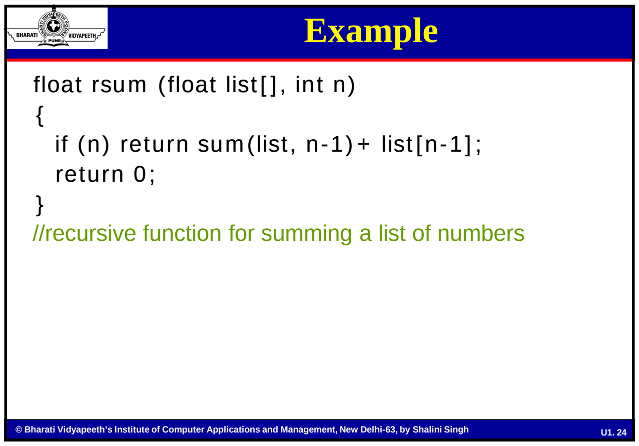


Example

```
float sum(float list[], int n)
{
  float tempsum = 0;
  int i;
  for (i =0; i<n; i++)
      tempsum += list[i];
  return tempsum;
}
```

Notice: pass the address of the first element of the array, not pass by value

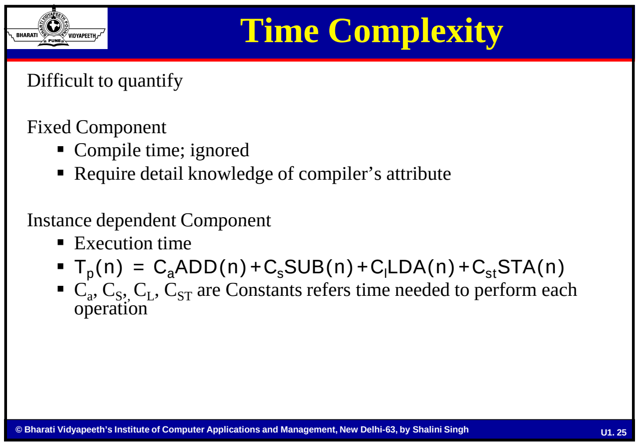
//iterative function for summing a list of numbers



Example

```
float rsum (float list[], int n)
{
  if (n) return sum(list, n-1)+ list[n-1];
  return 0;
}
```

//recursive function for summing a list of numbers



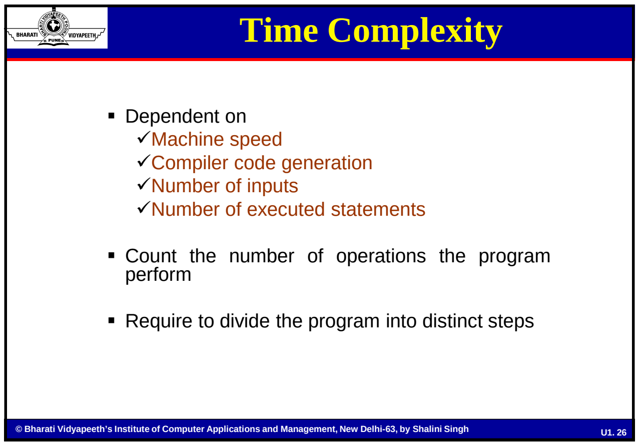
Time Complexity

Difficult to quantify

Fixed Component
- Compile time; ignored
- Require detail knowledge of compiler's attribute

Instance dependent Component
- Execution time
- $T_p(n) = C_a ADD(n) + C_s SUB(n) + C_l LDA(n) + C_{st} STA(n)$
- C_a, C_S, C_L, C_{ST} are Constants refers time needed to perform each operation

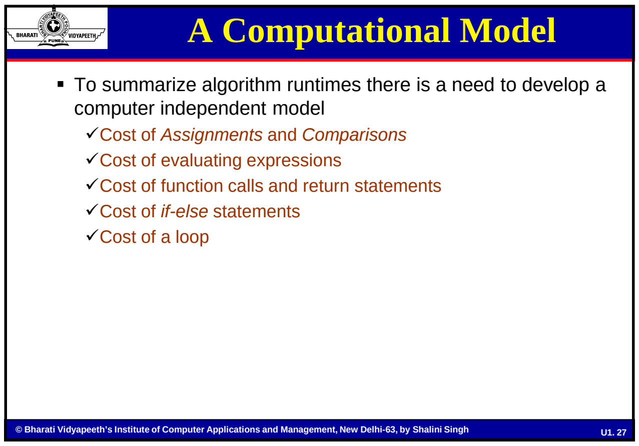
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 25



Time Complexity

- Dependent on
  - ✓Machine speed
  - ✓Compiler code generation
  - ✓Number of inputs
  - ✓Number of executed statements
- Count the number of operations the program perform
- Require to divide the program into distinct steps

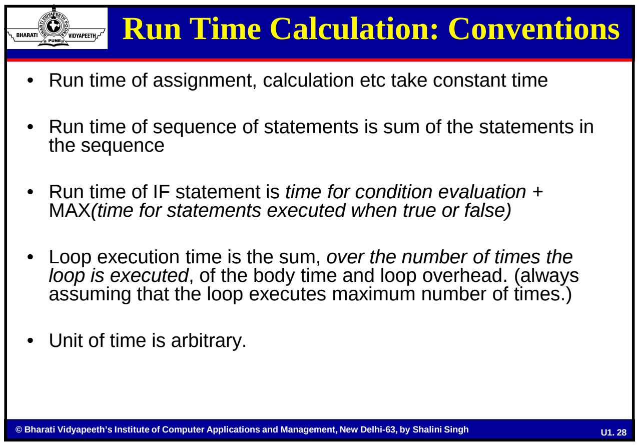
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 26



A Computational Model

- To summarize algorithm runtimes there is a need to develop a computer independent model
  - ✓Cost of *Assignments* and *Comparisons*
  - ✓Cost of evaluating expressions
  - ✓Cost of function calls and return statements
  - ✓Cost of *if-else* statements
  - ✓Cost of a loop

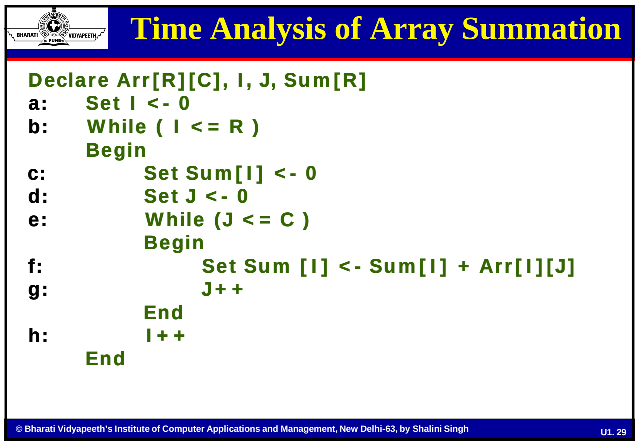
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 27



Run Time Calculation: Conventions

- Run time of assignment, calculation etc take constant time
- Run time of sequence of statements is sum of the statements in the sequence
- Run time of IF statement is *time for condition evaluation* + MAX*(time for statements executed when true or false)*
- Loop execution time is the sum, *over the number of times the loop is executed*, of the body time and loop overhead. (always assuming that the loop executes maximum number of times.)
- Unit of time is arbitrary.

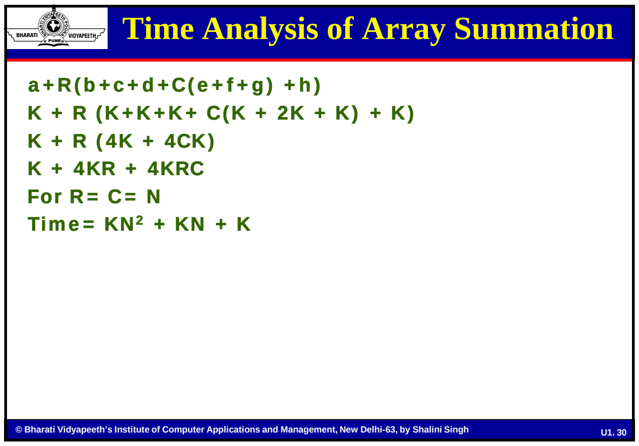
U1. 28



Time Analysis of Array Summation

```
Declare Arr[R][C], I, J, Sum[R]
a:      Set I <- 0
b:      While ( I <= R )
        Begin
c:          Set Sum[I] <- 0
d:          Set J <- 0
e:          While (J <= C )
            Begin
f:              Set Sum [I] <- Sum[I] + Arr[I][J]
g:              J++
            End
h:          I++
        End
```

U1. 29



Time Analysis of Array Summation

a+R(b+c+d+C(e+f+g) +h)

K + R (K+K+K+ C(K + 2K + K) + K)

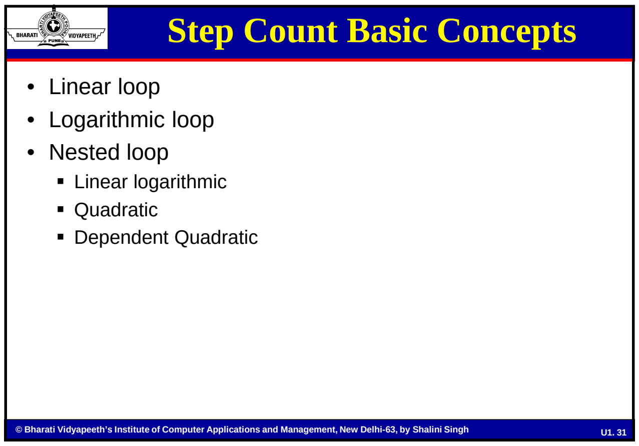
K + R (4K + 4CK)

K + 4KR + 4KRC

For R= C= N

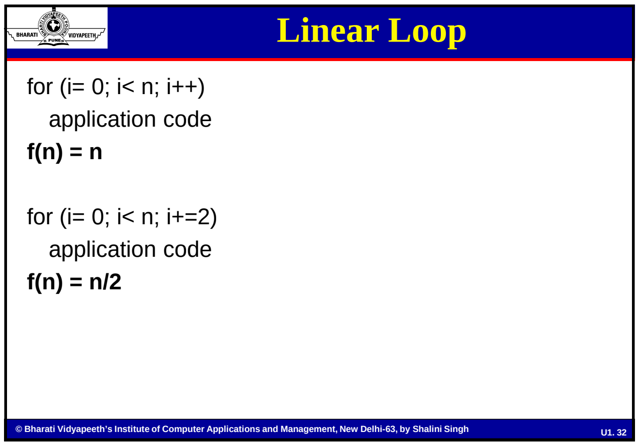
Time= KN^2 + KN + K

U1. 30



Step Count Basic Concepts

- Linear loop
- Logarithmic loop
- Nested loop
  - Linear logarithmic
  - Quadratic
  - Dependent Quadratic



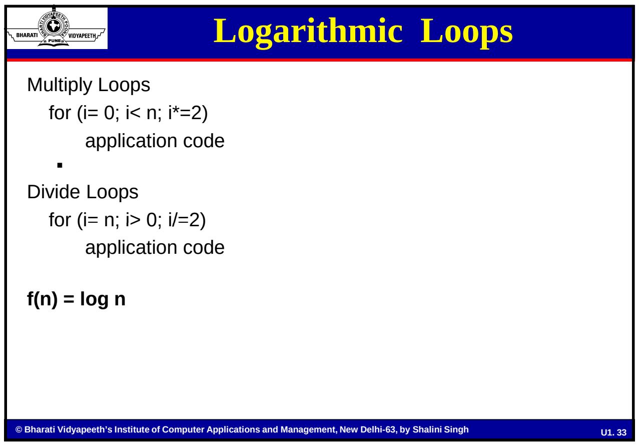
Linear Loop

```
for (i= 0; i< n; i++)
   application code
```

f(n) = n

```
for (i= 0; i< n; i+=2)
   application code
```

f(n) = n/2



Logarithmic Loops

Multiply Loops

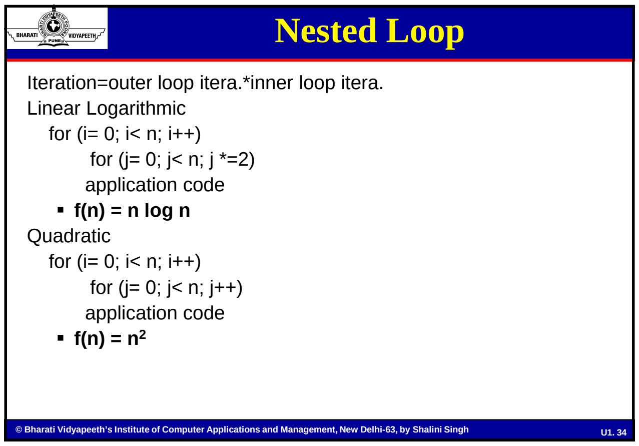
```
for (i= 0; i< n; i*=2)
     application code
```

▪

Divide Loops

```
for (i= n; i> 0; i/=2)
     application code
```

f(n) = log n



Nested Loop

Iteration=outer loop itera.*inner loop itera.

Linear Logarithmic

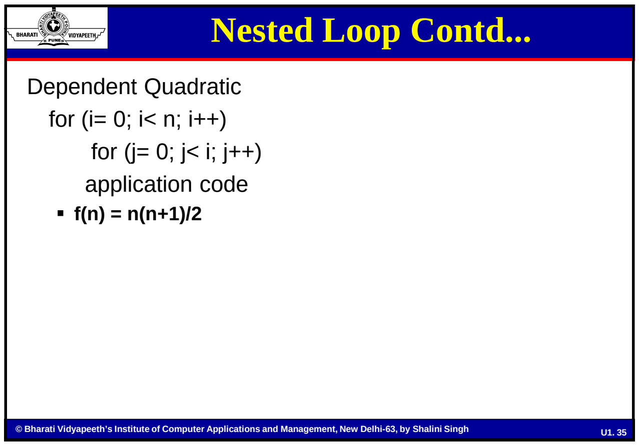
```
   for (i= 0; i< n; i++)
         for (j= 0; j< n; j *=2)
        application code
```

- **f(n) = n log n**

Quadratic

```
   for (i= 0; i< n; i++)
         for (j= 0; j< n; j++)
        application code
```

- **f(n) = n^2**

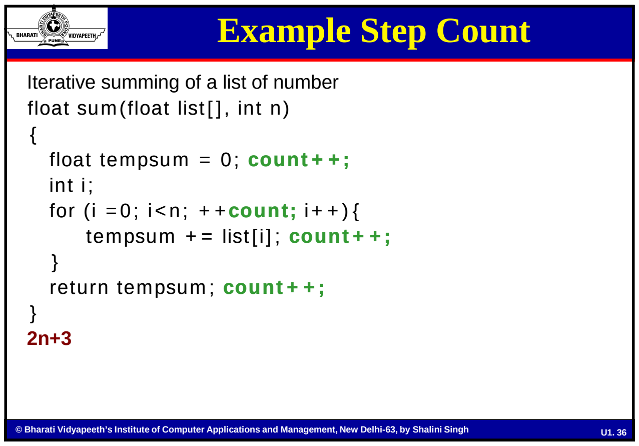


Nested Loop Contd...

Dependent Quadratic

```
   for (i= 0; i< n; i++)
         for (j= 0; j< i; j++)
        application code
```

- **f(n) = n(n+1)/2**

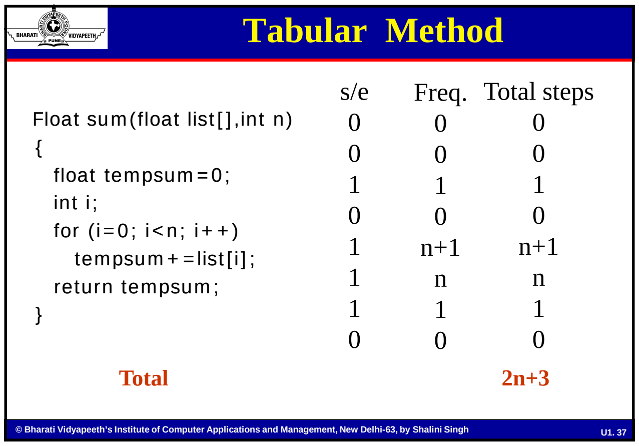


Example Step Count

Iterative summing of a list of number

```
float sum(float list[], int n)
{
   float tempsum = 0; count++;
   int i;
   for (i =0; i<n; ++count; i++){
        tempsum += list[i]; count++;
   }
   return tempsum; count++;
}
```

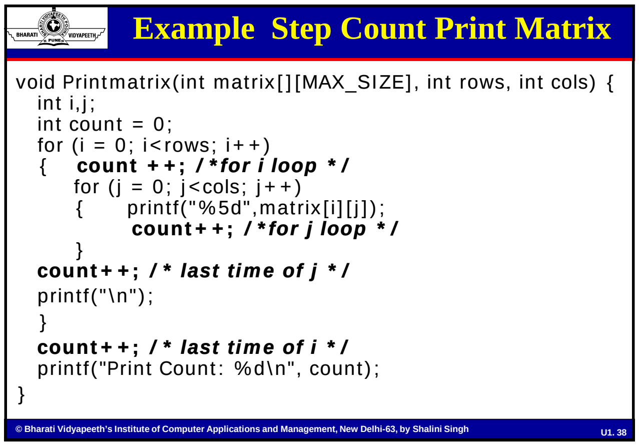
2n+3



Tabular Method

	s/e	Freq.	Total steps
`Float sum(float list[],int n)`	0	0	0
`{`	0	0	0
`float tempsum=0;`	1	1	1
`int i;`	0	0	0
`for (i=0; i<n; i++)`	1	n+1	n+1
`tempsum+=list[i];`	1	n	n
`return tempsum;`	1	1	1
`}`	0	0	0
Total			**2n+3**

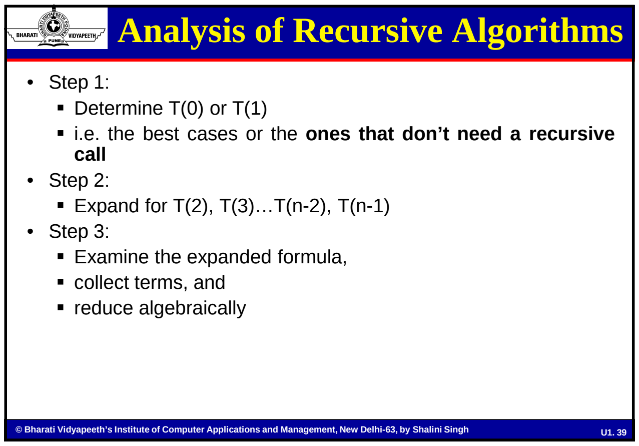
U1. 37



Example Step Count Print Matrix

```
void Printmatrix(int matrix[][MAX_SIZE], int rows, int cols) {
   int i,j;
   int count = 0;
   for (i = 0; i<rows; i++)
   {    count ++; /*for i loop */
        for (j = 0; j<cols; j++)
        {      printf("%5d",matrix[i][j]);
               count++; /*for j loop */
        }
   count++; /* last time of j */
   printf("\n");
   }
   count++; /* last time of i */
   printf("Print Count: %d\n", count);
}
```

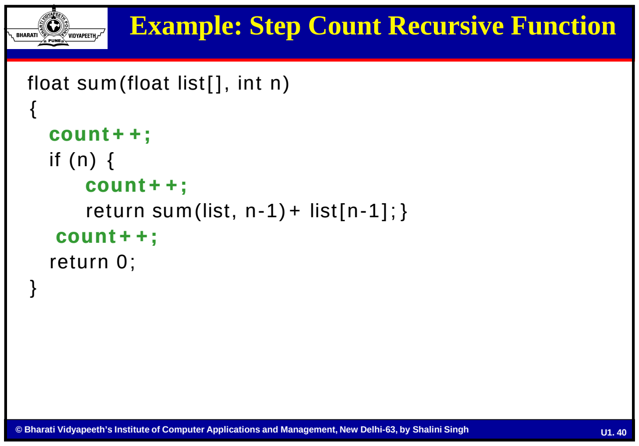
U1. 38



Analysis of Recursive Algorithms

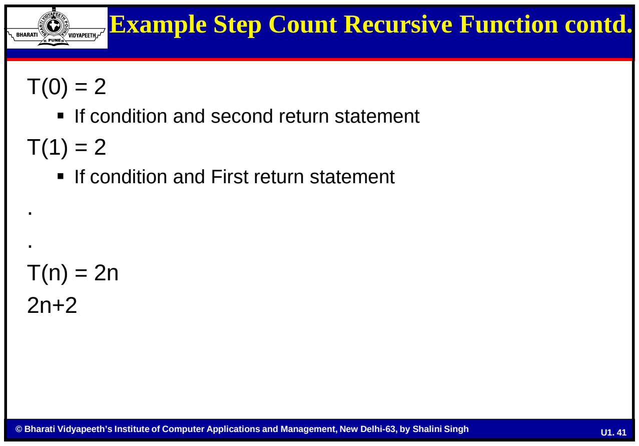
- Step 1:
  - Determine T(0) or T(1)
  - i.e. the best cases or the **ones that don't need a recursive call**
- Step 2:
  - Expand for T(2), T(3)…T(n-2), T(n-1)
- Step 3:
  - Examine the expanded formula,
  - collect terms, and
  - reduce algebraically

U1. 39



Example: Step Count Recursive Function

```
float sum(float list[], int n)
{
  count++;
  if (n) {
       count++;
       return sum(list, n-1)+ list[n-1];}
   count++;
  return 0;
}
```



Example Step Count Recursive Function contd.

T(0) = 2

- If condition and second return statement

T(1) = 2

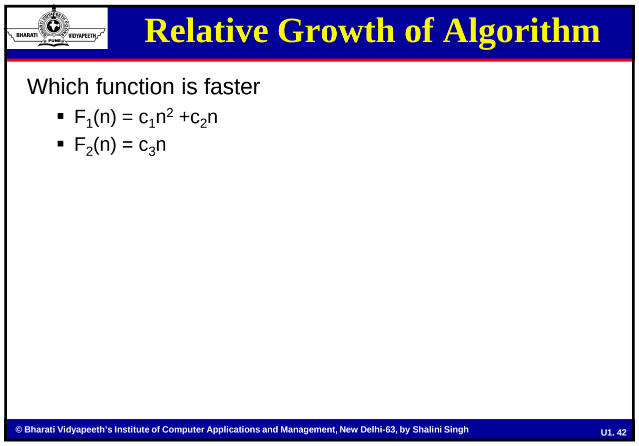
- If condition and First return statement

.

.

T(n) = 2n

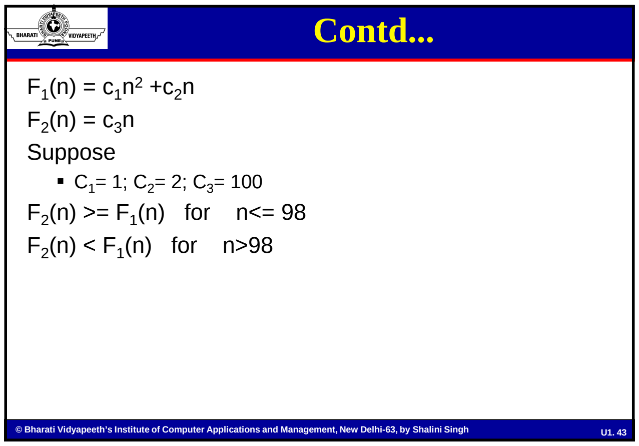
2n+2



Relative Growth of Algorithm

Which function is faster

- $F_1(n) = c_1n^2 + c_2n$
- $F_2(n) = c_3n$



Contd...

$F_1(n) = c_1n^2 + c_2n$

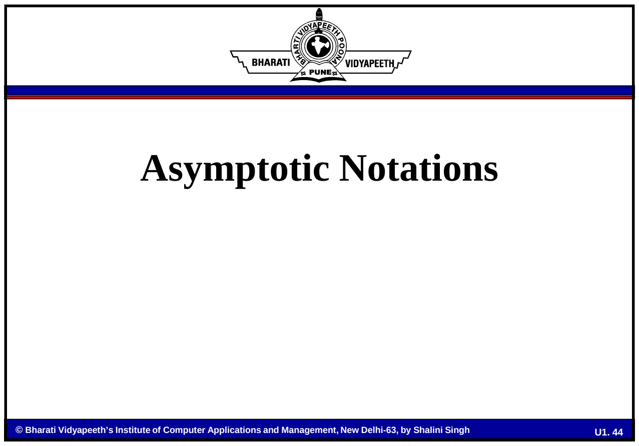
$F_2(n) = c_3n$

Suppose

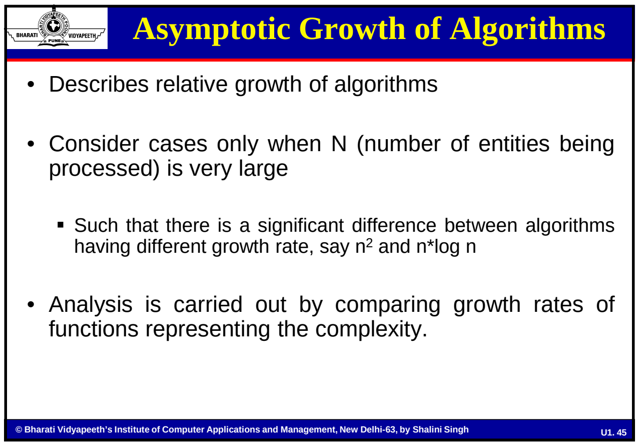
- $C_1 = 1$; $C_2 = 2$; $C_3 = 100$

$F_2(n) >= F_1(n)$ for $n <= 98$

$F_2(n) < F_1(n)$ for $n > 98$

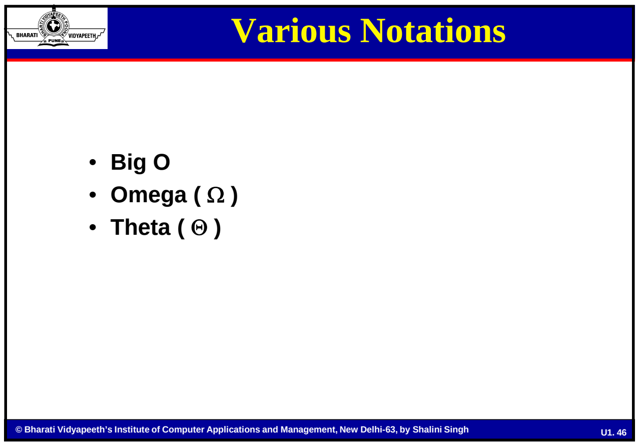


Asymptotic Notations



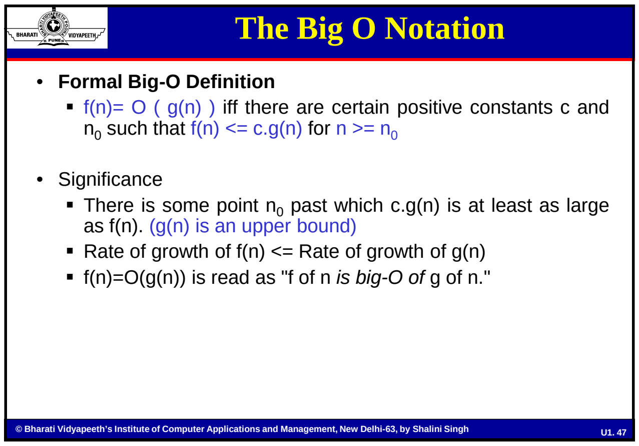
Asymptotic Growth of Algorithms

- Describes relative growth of algorithms
- Consider cases only when N (number of entities being processed) is very large
  - Such that there is a significant difference between algorithms having different growth rate, say n^2 and n*log n
- Analysis is carried out by comparing growth rates of functions representing the complexity.



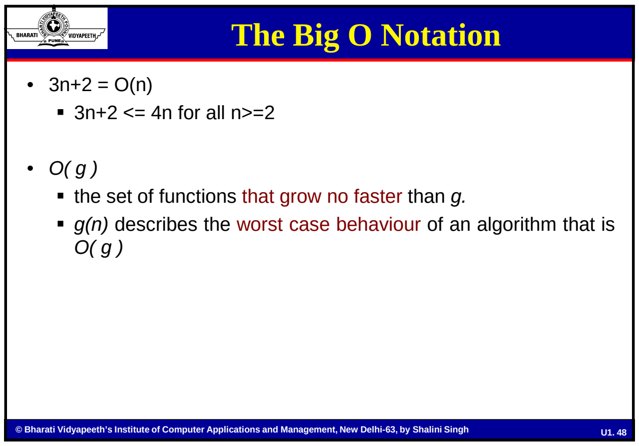
Various Notations

- **Big O**
- **Omega (Ω)**
- **Theta (Θ)**



The Big O Notation

- **Formal Big-O Definition**
  - f(n)= O (g(n)) iff there are certain positive constants c and n_0 such that f(n) <= c.g(n) for n >= n_0
- Significance
  - There is some point n_0 past which c.g(n) is at least as large as f(n). (g(n) is an upper bound)
  - Rate of growth of f(n) <= Rate of growth of g(n)
  - f(n)=O(g(n)) is read as "f of n *is big-O of* g of n."



The Big O Notation

- 3n+2 = O(n)
  - 3n+2 <= 4n for all n>=2
- *O(g)*
  - the set of functions that grow no faster than *g.*
  - *g(n)* describes the worst case behaviour of an algorithm that is *O(g)*

The Big O Notation Example

3n+2 = O(n)

- 3n+2 <= 4n for all n>=2

The Omega (Ω) Notation

- **Formal Definition**
  - f(n)= Ω (g(n)) iff there exists positive constants c and n_0 such that f(n)>=c.g(n) for all n, n>=n_0.
- Significance
  - When the data size n is large enough then there will be a function c.g(n) which is smaller than f(n). (g(n) is a lower bound)
  - Rate of growth of f(n) >= rate of growth of g(n)
  - f(n)= Ω (g(n)) is read as "f of n *is omega of* g of n.“
  - The largest function which is a lower limit is chosen for g(n). Example: 3n+3= Ω(1) is true, but not as informative of 3n+3= Ω(n).

The Omega (Ω) Notation Example

3n+2 = Ω(n)

- 3n+2 >= 3n for all n>=1

The Theta (Θ) Notation

- When the upper and lower bound are the same then theta notation can be used to describe the complexity class.
- It is used because it is more precise.
- **3n+2 = Θ(n)**
  - **3n+2 >= 3n for all n>=1 and 3n+2<=4n for all n>=2 so c_1 = 3 and c_2 = 4 and n_0 =2**

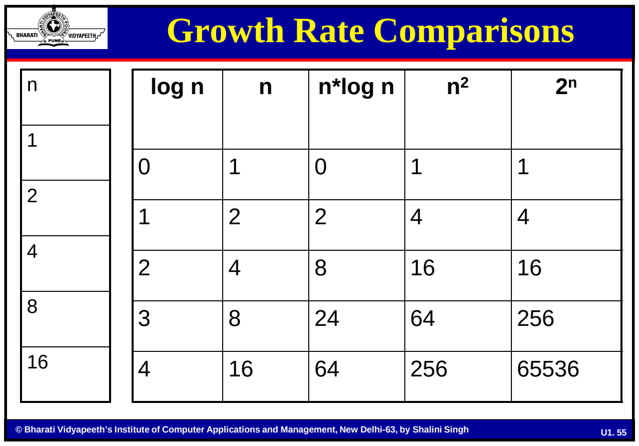
The Theta (Θ) Notation

- **Formal Definition**
  - f(n)= Θ(g(n)) iff there exists positive constants c1, c2, and n_0 such that c1*g(n) <= f(n) <= c2*g(n) for all n, n>=n_0.
- Significance
  - Growth rate of f(n) is the same as the growth rate of g(n), i.e. g(n) is both an upper and the lower bound on f(n)

The Theta (Θ) Notation Example

3n+2 = Θ(n)

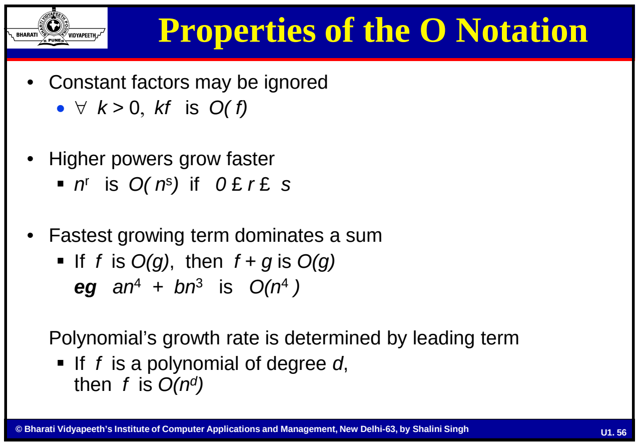
- 3n+2 >= 3n for all n>=1 and 3n+2<=4n for all n>=2 so c_1 = 3 and c_2 = 4 and n_0 =2



Growth Rate Comparisons

n
1
2
4
8
16

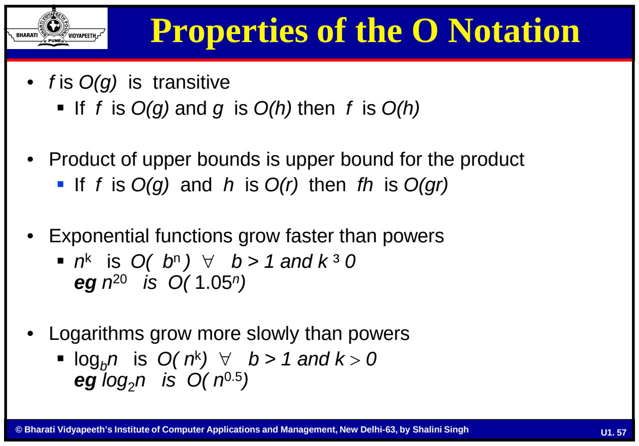
log n	n	n*log n	n^2	2^n
0	1	0	1	1
1	2	2	4	4
2	4	8	16	16
3	8	24	64	256
4	16	64	256	65536



Properties of the O Notation

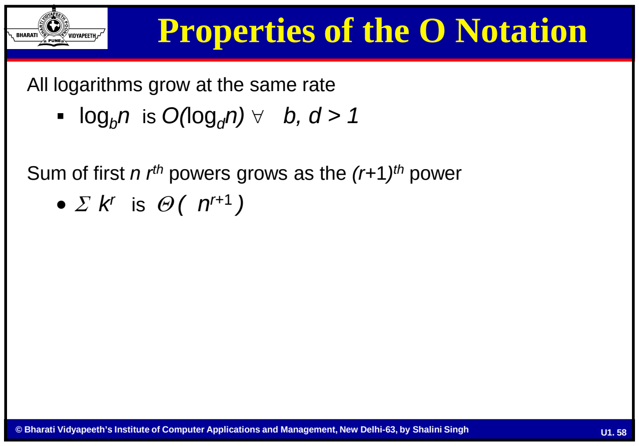
- Constant factors may be ignored
  - $\forall\ k > 0,\ kf$ is $O(f)$
- Higher powers grow faster
  - n^r is $O(n^s)$ if $0 £ r £ s$
- Fastest growing term dominates a sum
  - If f is $O(g)$, then $f + g$ is $O(g)$
    eg $an^4 + bn^3$ is $O(n^4)$

Polynomial's growth rate is determined by leading term
  - If f is a polynomial of degree d,
    then f is $O(n^d)$



Properties of the O Notation

- f is $O(g)$ is transitive
  - If f is $O(g)$ and g is $O(h)$ then f is $O(h)$
- Product of upper bounds is upper bound for the product
  - If f is $O(g)$ and h is $O(r)$ then fh is $O(gr)$
- Exponential functions grow faster than powers
  - n^k is $O(b^n)\ \forall\ b > 1$ and $k ³ 0$
    eg n^{20} is $O(1.05^n)$
- Logarithms grow more slowly than powers
  - $\log_b n$ is $O(n^k)\ \forall\ b > 1$ and $k > 0$
    eg $\log_2 n$ is $O(n^{0.5})$



Properties of the O Notation

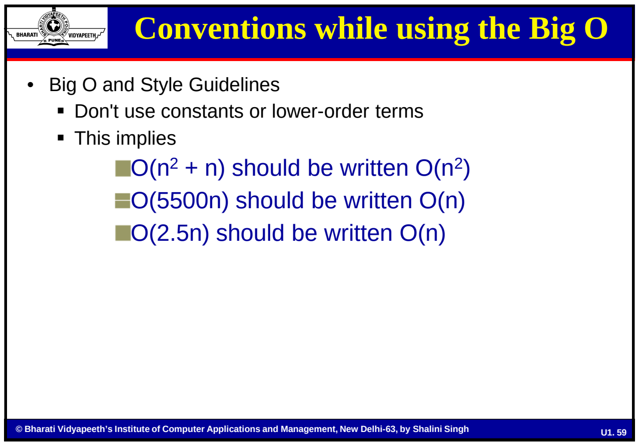
All logarithms grow at the same rate

- $\log_b n$ is $O(\log_d n)$ $\forall$ $b, d > 1$

Sum of first n r^{th} powers grows as the $(r+1)^{th}$ power

- $\Sigma\ k^r$ is $\Theta(\ n^{r+1})$

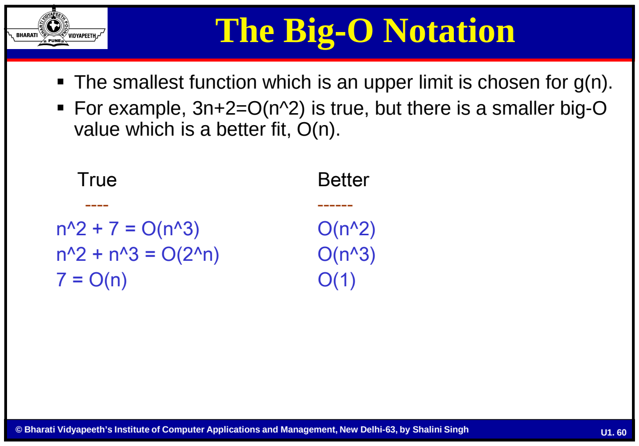
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 58



Conventions while using the Big O

- Big O and Style Guidelines
  - Don't use constants or lower-order terms
  - This implies
    - $O(n^2 + n)$ should be written $O(n^2)$
    - O(5500n) should be written O(n)
    - O(2.5n) should be written O(n)

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 59



The Big-O Notation

- The smallest function which is an upper limit is chosen for g(n).
- For example, 3n+2=O(n^2) is true, but there is a smaller big-O value which is a better fit, O(n).

True	Better
----	------
n^2 + 7 = O(n^3)	O(n^2)
n^2 + n^3 = O(2^n)	O(n^3)
7 = O(n)	O(1)

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 60

Common Growth Rates for Runtime

Some common complexity classes are;

- Constant O(1)
- Logarithmic (log n)
- Linear O(n)
- Linear Logarithmic O(n log n)
- Quadratic O(n^2)
- Exponential O(2^n)

Analyzing an Algorithm

- Simple statement sequence

  ```
  s1; s2; …. ; sk
  ```

  - *O(1)* as long as *k* is constant
  - Runtime is constant.
  - The running time of the statement will not change in relation to N.
- Simple loops

  ```
  for(i=0;i<n;i++) { s; }
  ```

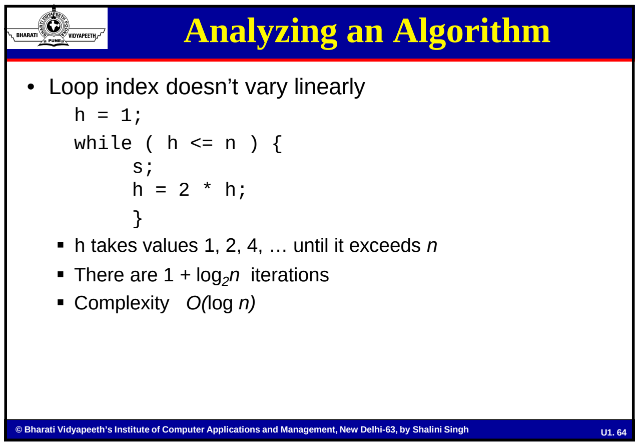
  where s is *O(1)*
  - Time complexity is *n O(1)* or *O(n)*
  - Runtime is linear.
  - The running time of the loop is directly proportional to N.
  - When N doubles, so does the running time.

Analyzing an Algorithm

- Nested loops

  ```
  for(i=0;i<n;i++)
     for(j=0;j<n;j++) { s; }
  ```

  - Complexity is *n O(n)* or *O(n^2)*
  - Runtime is quadratic.
  - The running time of the two loops is proportional to the square of N.
  - When N doubles, the running time increases by N * N.

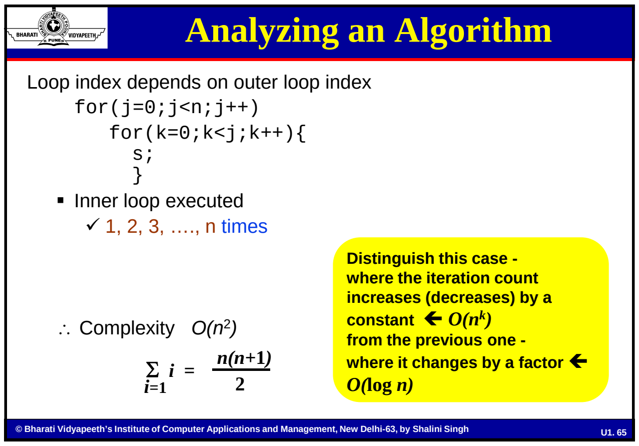


Analyzing an Algorithm

- Loop index doesn't vary linearly

```
h = 1;
while ( h <= n ) {
     s;
     h = 2 * h;
     }
```

  - h takes values 1, 2, 4, … until it exceeds *n*
  - There are 1 + $\log_2 n$ iterations
  - Complexity *O*(log *n)*



Analyzing an Algorithm

Loop index depends on outer loop index

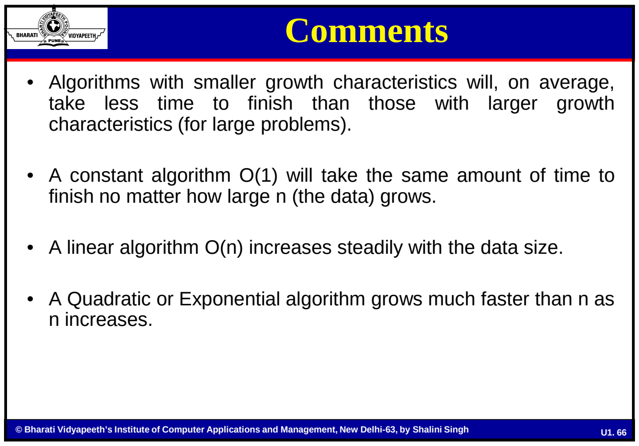
```
for(j=0;j<n;j++)
   for(k=0;k<j;k++){
     s;
     }
```

- Inner loop executed
  - ✓ 1, 2, 3, …., n times

∴ Complexity *O*(n^2)

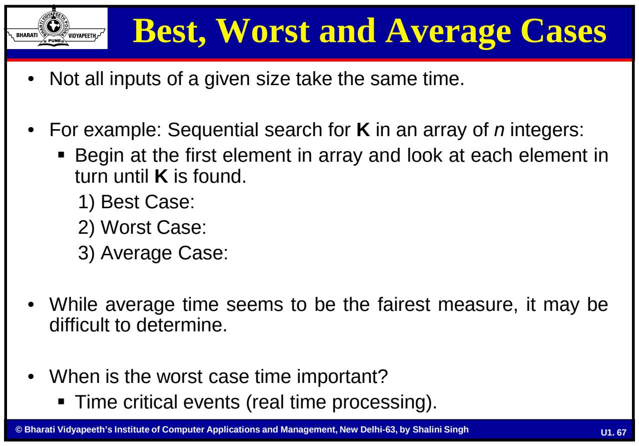
$$\sum_{i=1} i = \frac{n(n+1)}{2}$$

Distinguish this case - where the iteration count increases (decreases) by a constant ← *O*(n^k) from the previous one - where it changes by a factor ← *O*(log *n)*



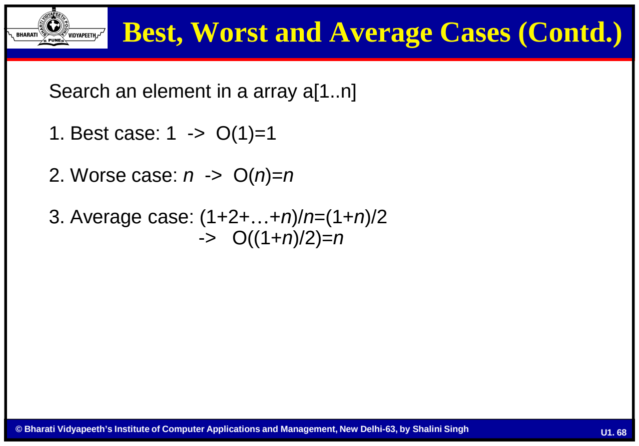
Comments

- Algorithms with smaller growth characteristics will, on average, take less time to finish than those with larger growth characteristics (for large problems).
- A constant algorithm O(1) will take the same amount of time to finish no matter how large n (the data) grows.
- A linear algorithm O(n) increases steadily with the data size.
- A Quadratic or Exponential algorithm grows much faster than n as n increases.



Best, Worst and Average Cases

- Not all inputs of a given size take the same time.
- For example: Sequential search for **K** in an array of *n* integers:
  - Begin at the first element in array and look at each element in turn until **K** is found.
    1) Best Case:
    2) Worst Case:
    3) Average Case:
- While average time seems to be the fairest measure, it may be difficult to determine.
- When is the worst case time important?
  - Time critical events (real time processing).



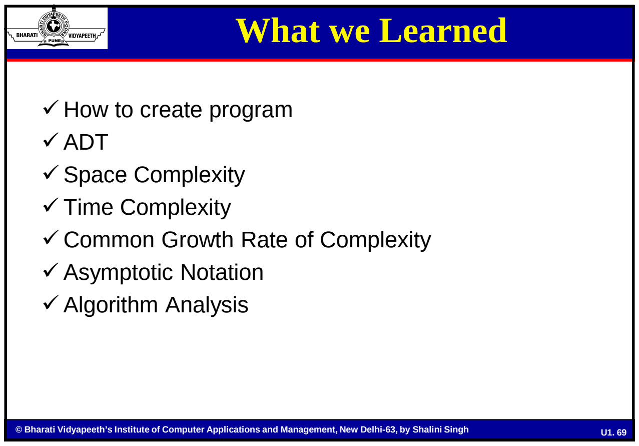
Best, Worst and Average Cases (Contd.)

Search an element in a array a[1..n]

1. Best case: 1 -> O(1)=1

2. Worse case: *n* -> O(*n*)=*n*

3. Average case: (1+2+…+*n*)/*n*=(1+*n*)/2
   -> O((1+*n*)/2)=*n*



What we Learned

✓ How to create program
✓ ADT
✓ Space Complexity
✓ Time Complexity
✓ Common Growth Rate of Complexity
✓ Asymptotic Notation
✓ Algorithm Analysis

LINEAR DATA STRUCTURE

U1. 70

Objectives

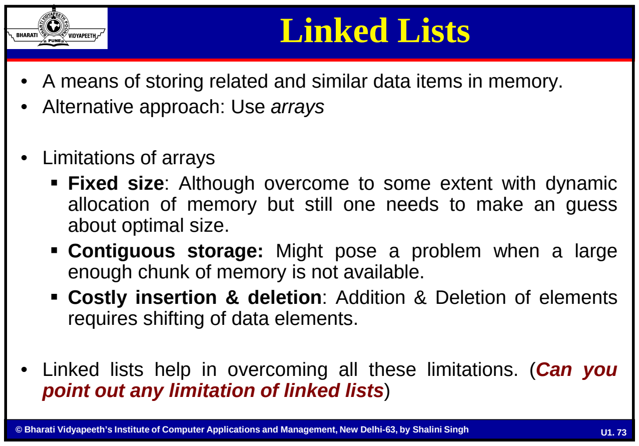
- What is Linear List?
- Comparison with Array
- Linked List Basic Operations
- Linked List Implementation
- Assignment based on Advance Operations
- Variation in Linked List
  - Doubly Linked List
  - Circular Linked List

U1. 71

Linear List

- List in which each element has a unique successor
- Two Categories
  - Restricted
    - ✓STACK (LIFO)
    - ✓QUEUE(FIFO)
  - General

U1. 72

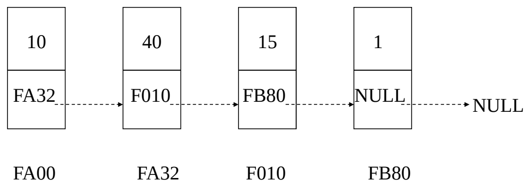


Linked Lists

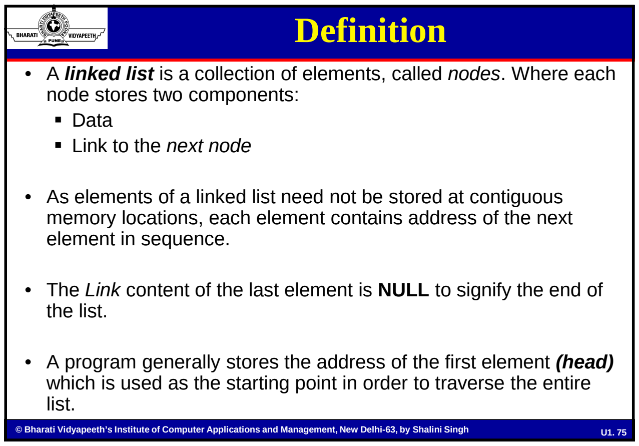
- A means of storing related and similar data items in memory.
- Alternative approach: Use *arrays*
- Limitations of arrays
  - **Fixed size**: Although overcome to some extent with dynamic allocation of memory but still one needs to make an guess about optimal size.
  - **Contiguous storage:** Might pose a problem when a large enough chunk of memory is not available.
  - **Costly insertion & deletion**: Addition & Deletion of elements requires shifting of data elements.
- Linked lists help in overcoming all these limitations. (***Can you point out any limitation of linked lists***)

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 73

Memory Representation



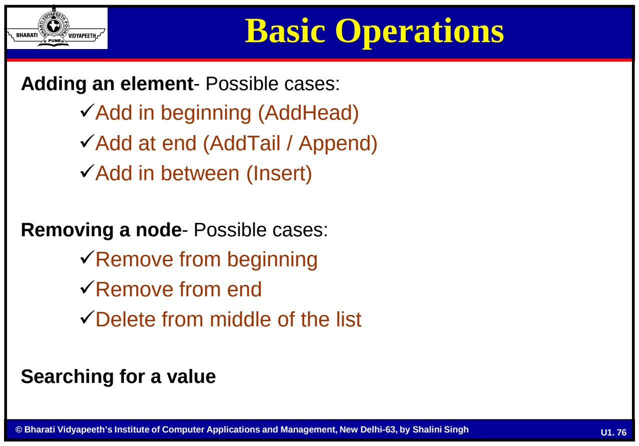
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 74



Definition

- A ***linked list*** is a collection of elements, called *nodes*. Where each node stores two components:
  - Data
  - Link to the *next node*
- As elements of a linked list need not be stored at contiguous memory locations, each element contains address of the next element in sequence.
- The *Link* content of the last element is **NULL** to signify the end of the list.
- A program generally stores the address of the first element ***(head)*** which is used as the starting point in order to traverse the entire list.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 75



Basic Operations

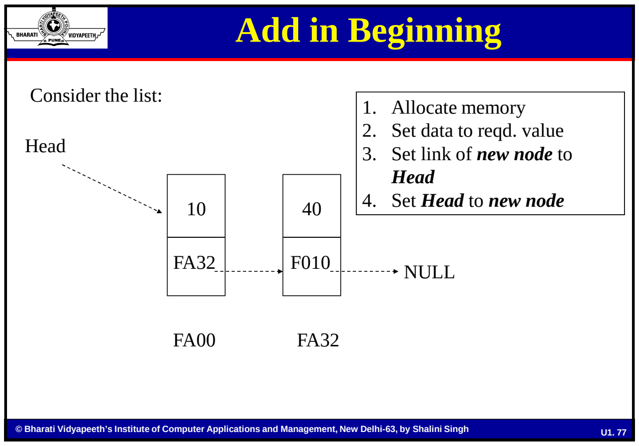
Adding an element- Possible cases:

- ✓Add in beginning (AddHead)
- ✓Add at end (AddTail / Append)
- ✓Add in between (Insert)

Removing a node- Possible cases:

- ✓Remove from beginning
- ✓Remove from end
- ✓Delete from middle of the list

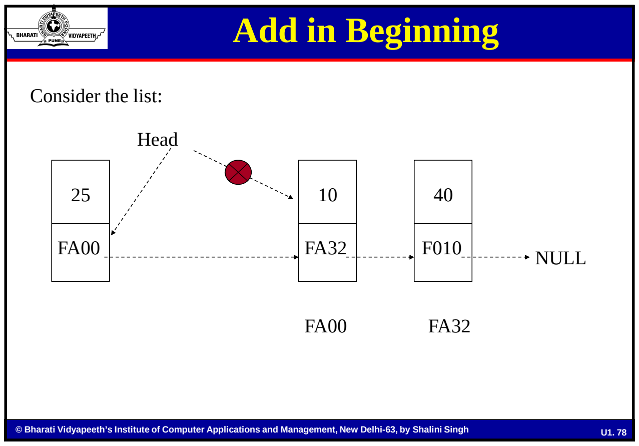
Searching for a value



Add in Beginning

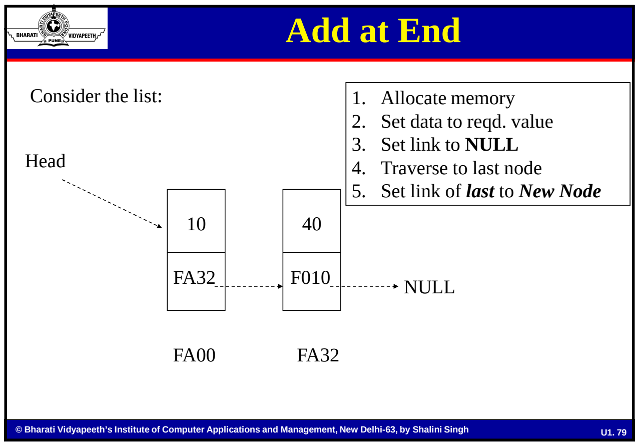
Consider the list:

1. Allocate memory
2. Set data to reqd. value
3. Set link of ***new node*** to ***Head***
4. Set ***Head*** to ***new node***



Add in Beginning

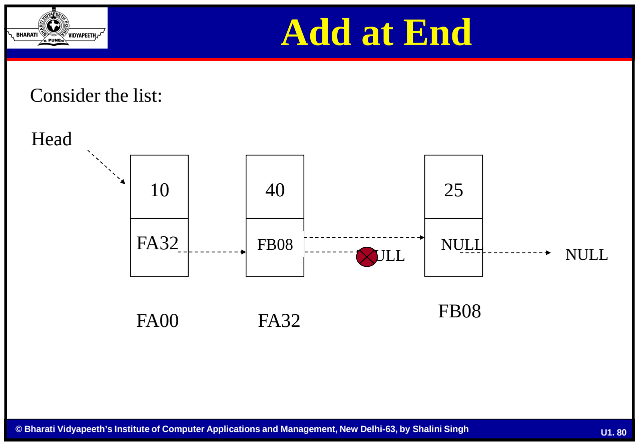
Consider the list:



Add at End

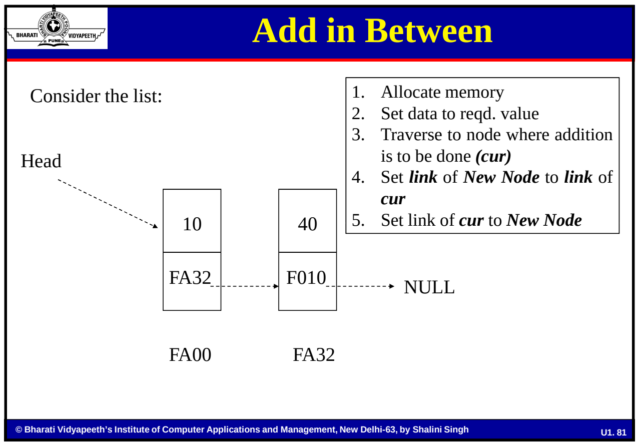
Consider the list:

1. Allocate memory
2. Set data to reqd. value
3. Set link to **NULL**
4. Traverse to last node
5. Set link of ***last*** to ***New Node***



Add at End

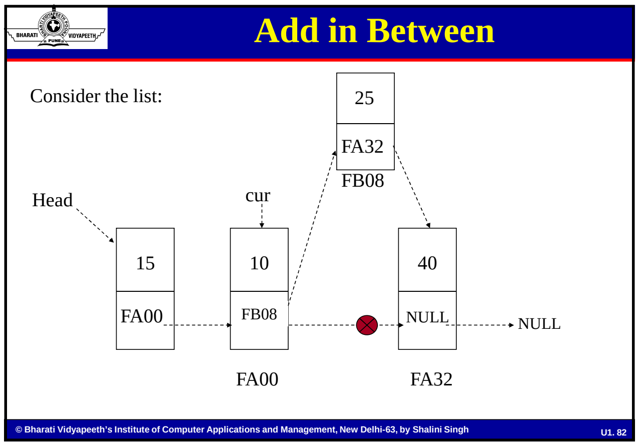
Consider the list:



Add in Between

Consider the list:

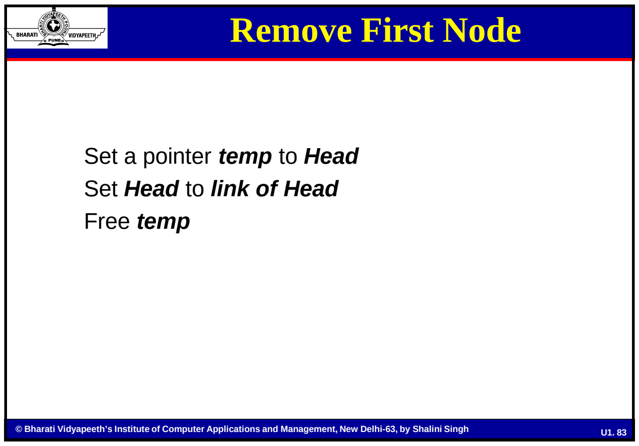
1. Allocate memory
2. Set data to reqd. value
3. Traverse to node where addition is to be done *(cur)*
4. Set ***link*** of ***New Node*** to ***link*** of ***cur***
5. Set link of ***cur*** to ***New Node***



Add in Between

Consider the list:

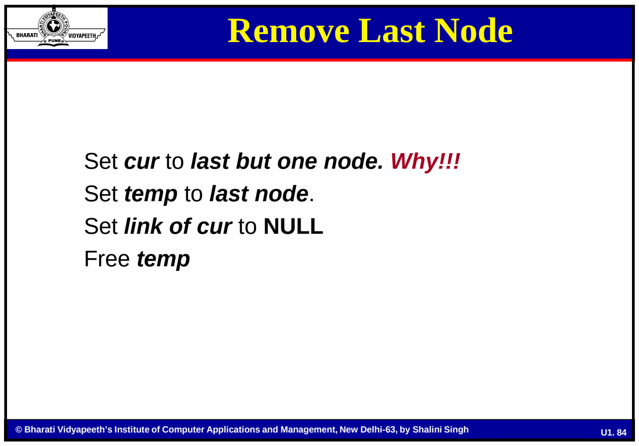
U1. 82



Remove First Node

Set a pointer ***temp*** to ***Head***
Set ***Head*** to ***link of Head***
Free ***temp***

U1. 83



Remove Last Node

Set ***cur*** to ***last but one node. Why!!!***
Set ***temp*** to ***last node***.
Set ***link of cur*** to **NULL**
Free ***temp***

U1. 84

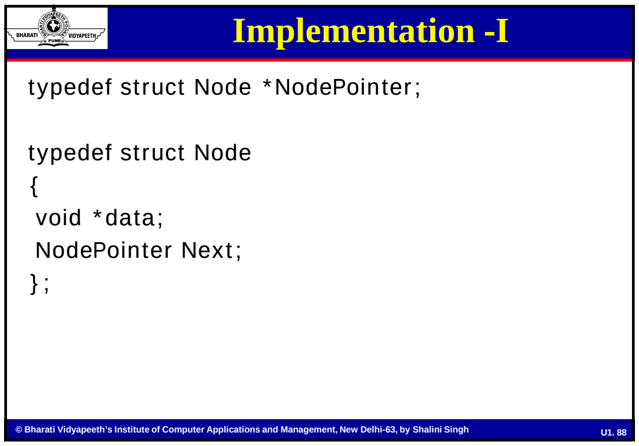
Remove Intermediate Node

Set ***cur*** to ***the node prior to the one that is to be deleted. Why!!!***
Set ***temp*** to ***the node to be deleted***.
Set ***link of cur*** to ***link of temp***
Free ***temp***

Search

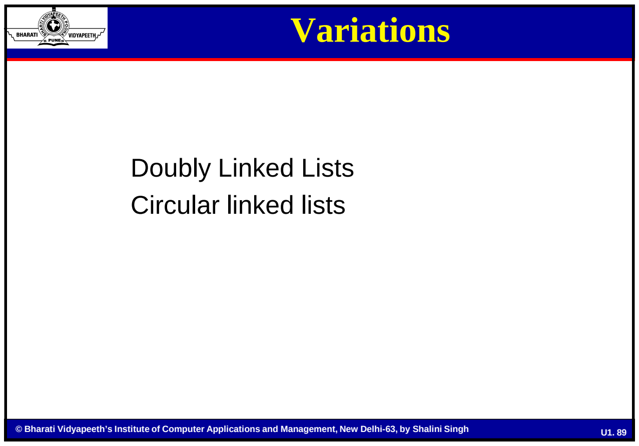
```
1. Set Pre to Null
2. Set Loc to Head
3. If Target is less than Head data Return False
4. While Loc and Loc data < Target
     i.  Set Pre to Loc
     ii. Set Loc to Next
5. If not Loc Return False
6. If Loc Data is equal to target Return True
7. Else Return False
```

IMPLEMENTATION



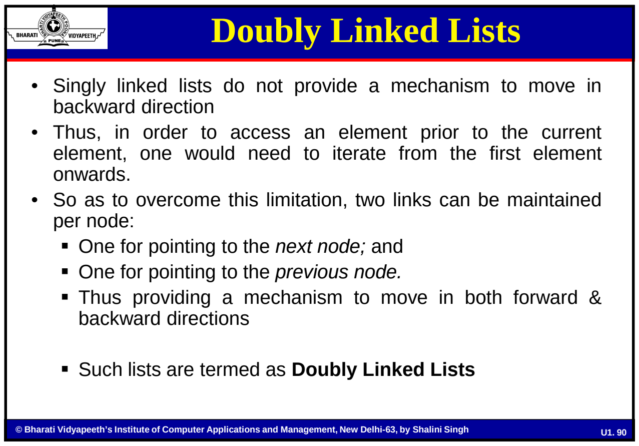
Implementation -I

```
typedef struct Node *NodePointer;

typedef struct Node
{
 void *data;
 NodePointer Next;
};
```



Variations

Doubly Linked Lists
Circular linked lists

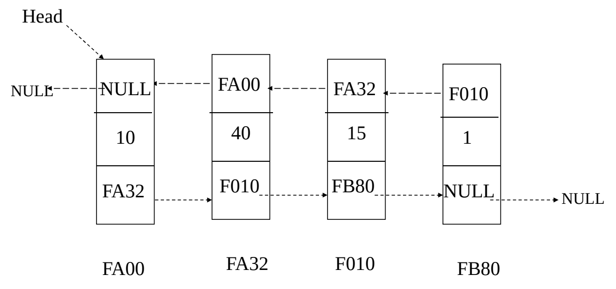


Doubly Linked Lists

- Singly linked lists do not provide a mechanism to move in backward direction
- Thus, in order to access an element prior to the current element, one would need to iterate from the first element onwards.
- So as to overcome this limitation, two links can be maintained per node:
  - One for pointing to the *next node;* and
  - One for pointing to the *previous node.*
  - Thus providing a mechanism to move in both forward & backward directions

  - Such lists are termed as **Doubly Linked Lists**

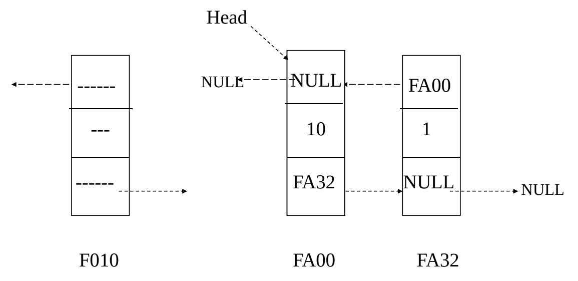
Doubly Linked Lists

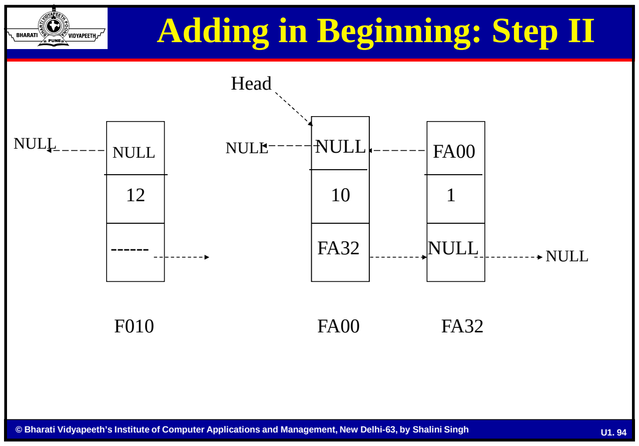


Basic Operations

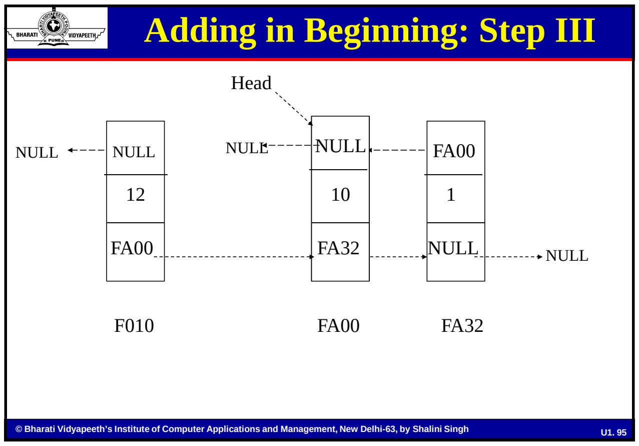
- Similar to Singly Linked Lists
- Addition
  - In Beginning
  - In Middle
  - At End
- Removal
  - From Beginning
  - From Middle
  - From End
- Searching

Adding in Beginning: Step I

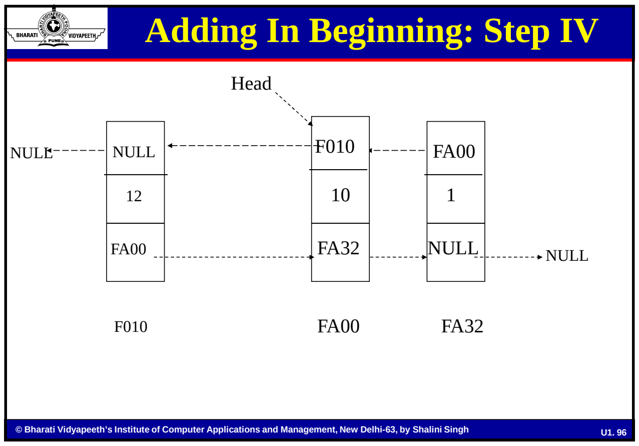




Adding in Beginning: Step II

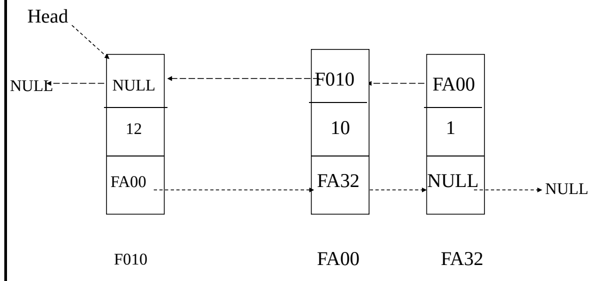


Adding in Beginning: Step III



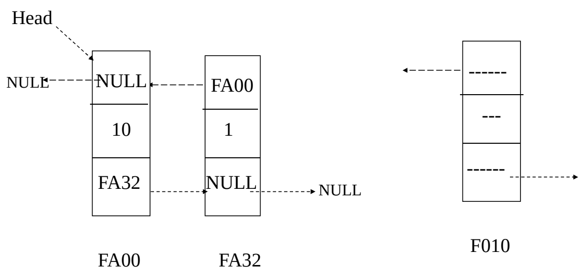
Adding In Beginning: Step IV

Adding in Beginning: Step V



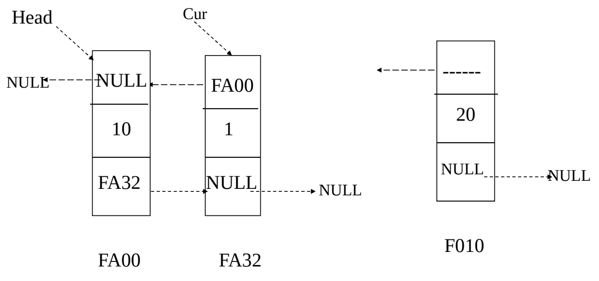
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 97

Adding at End: Step I

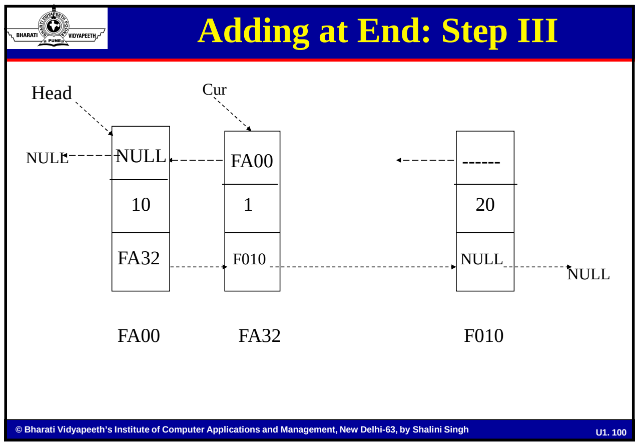


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 98

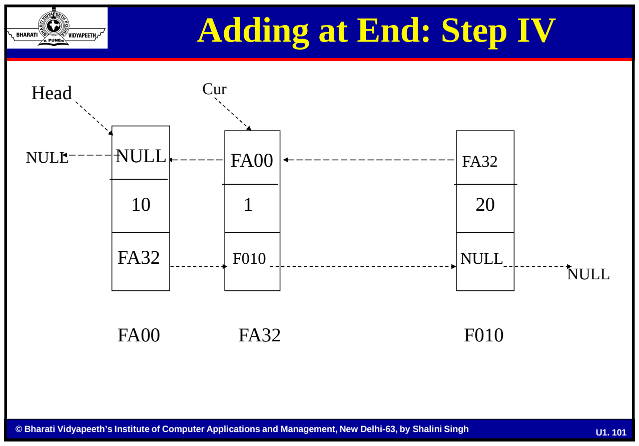
Adding at End: Step II



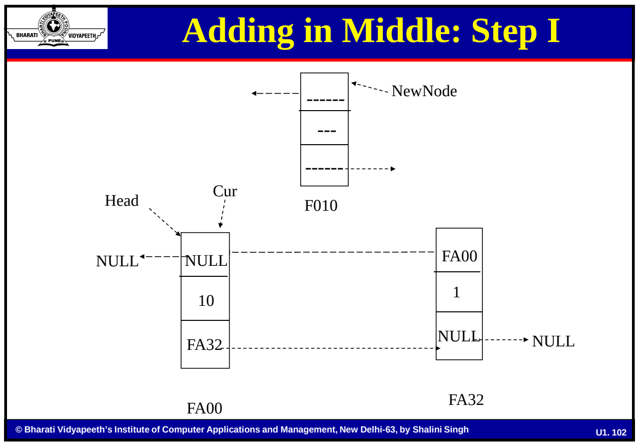
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 99



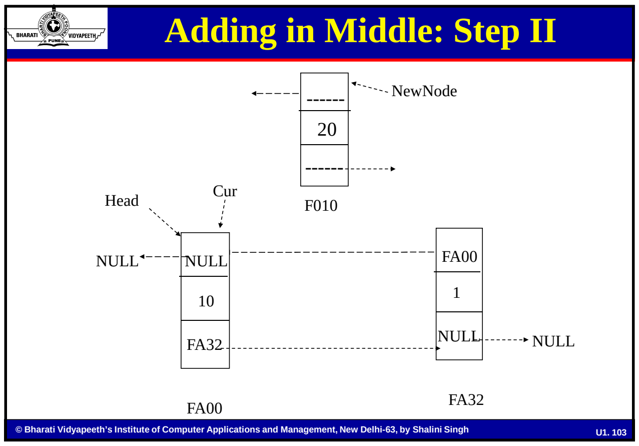
Adding at End: Step III



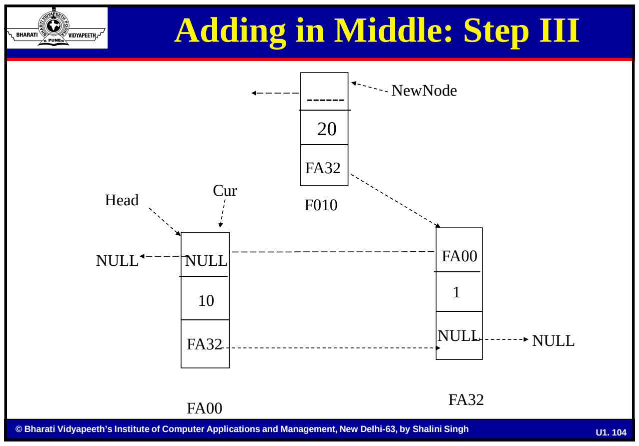
Adding at End: Step IV



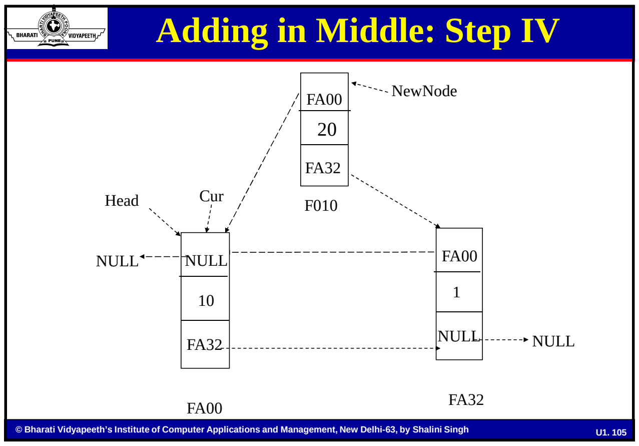
Adding in Middle: Step I



Adding in Middle: Step II

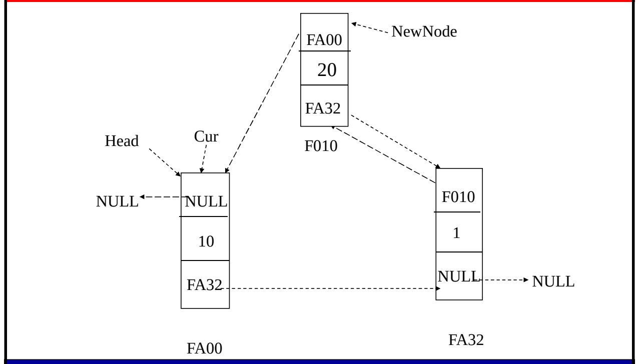


Adding in Middle: Step III

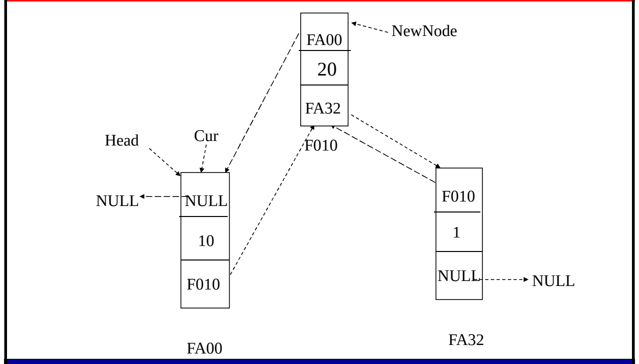


Adding in Middle: Step IV

Adding in Middle: Step V

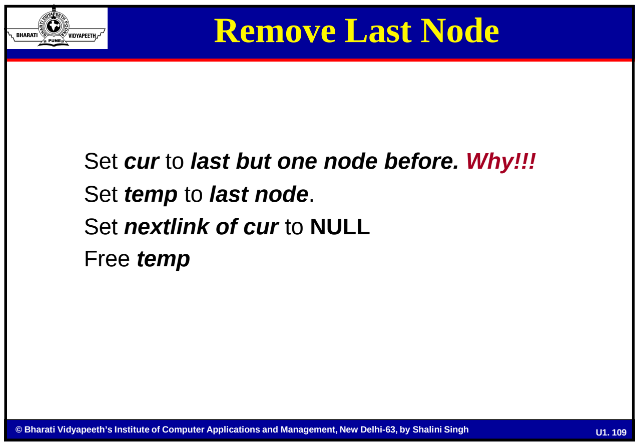


Adding in Middle: Step VI



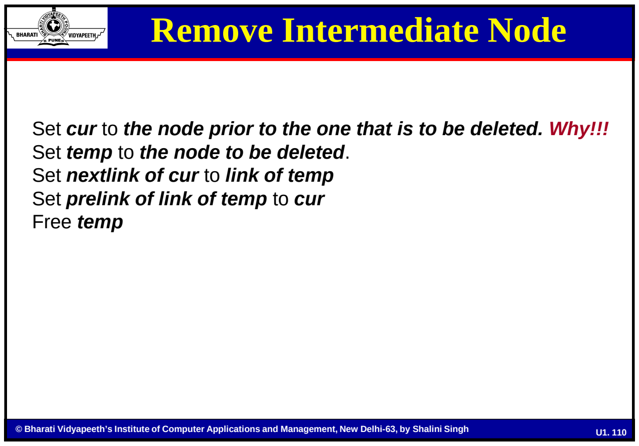
Remove First Node

Set a pointer ***temp*** to ***Head***
Set ***Head*** to ***Nextlink of Head***
Set ***Prelink*** of ***Head*** to ***Null***
Free ***temp***



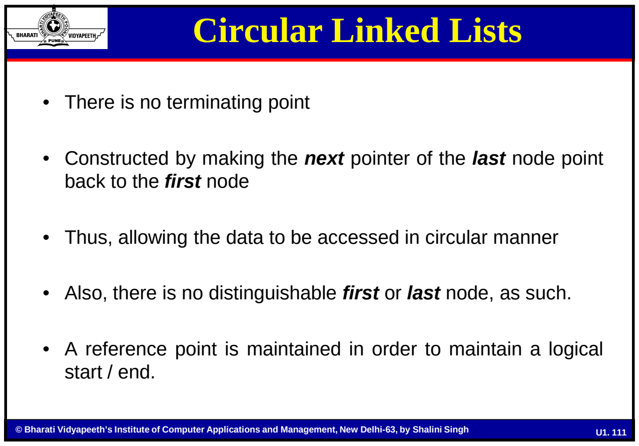
Remove Last Node

Set ***cur*** to ***last but one node before. Why!!!***
Set ***temp*** to ***last node***.
Set ***nextlink of cur*** to **NULL**
Free ***temp***



Remove Intermediate Node

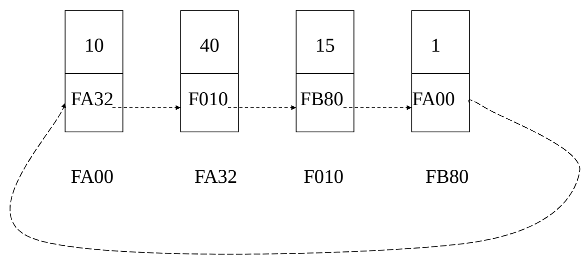
Set ***cur*** to ***the node prior to the one that is to be deleted. Why!!!***
Set ***temp*** to ***the node to be deleted***.
Set ***nextlink of cur*** to ***link of temp***
Set ***prelink of link of temp*** to ***cur***
Free ***temp***



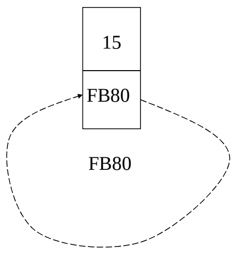
Circular Linked Lists

- There is no terminating point
- Constructed by making the ***next*** pointer of the ***last*** node point back to the ***first*** node
- Thus, allowing the data to be accessed in circular manner
- Also, there is no distinguishable ***first*** or ***last*** node, as such.
- A reference point is maintained in order to maintain a logical start / end.

Memory Representation



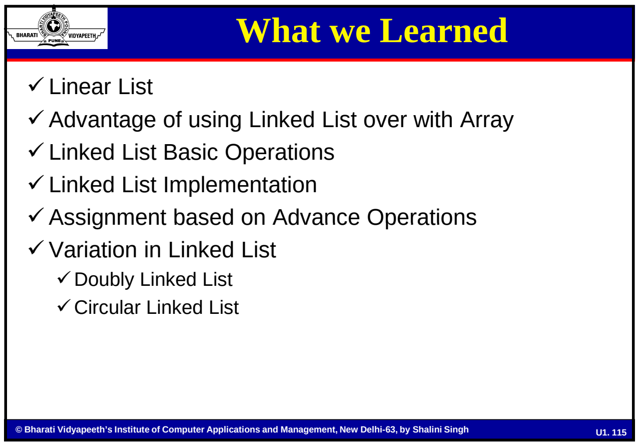
Circular Linked List with a Single Node



http://www.ceglug.org/index.php/labs/67-circularly-singly-linked-list

Applications

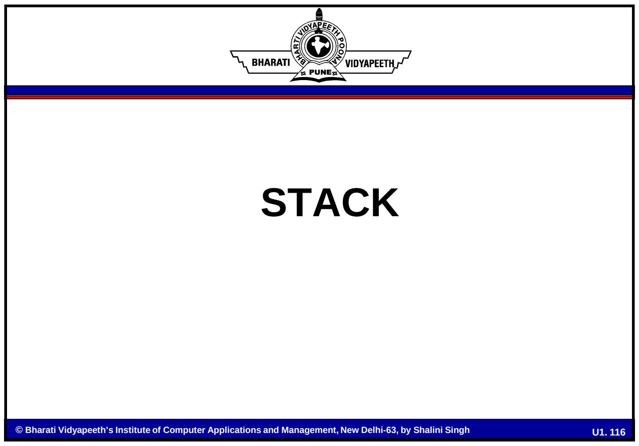
- Implementation of linked ***stacks***
- Implementation of linked ***queues***
- Representing polynomials
- Etc.



What we Learned

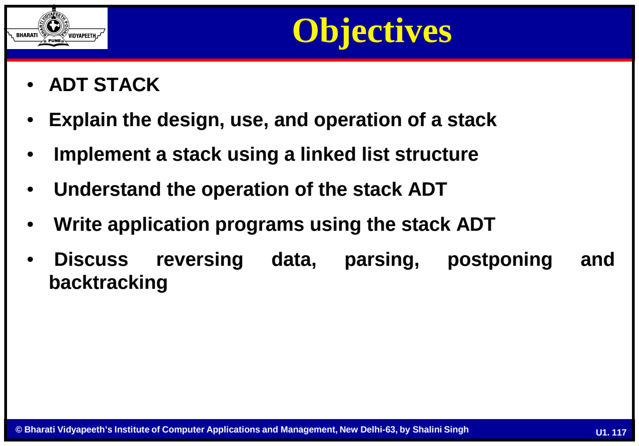
- ✓ Linear List
- ✓ Advantage of using Linked List over with Array
- ✓ Linked List Basic Operations
- ✓ Linked List Implementation
- ✓ Assignment based on Advance Operations
- ✓ Variation in Linked List
  - ✓ Doubly Linked List
  - ✓ Circular Linked List

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 115



STACK

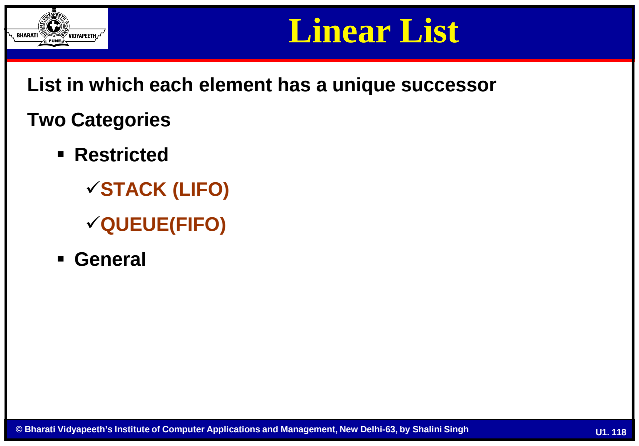
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 116



Objectives

- **ADT STACK**
- **Explain the design, use, and operation of a stack**
- **Implement a stack using a linked list structure**
- **Understand the operation of the stack ADT**
- **Write application programs using the stack ADT**
- **Discuss reversing data, parsing, postponing and backtracking**

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 117



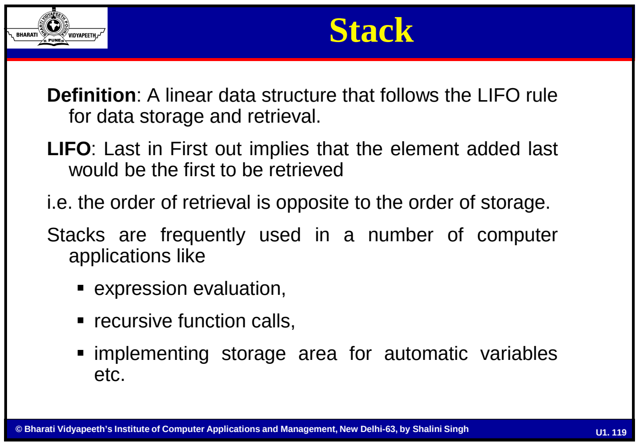
Linear List

List in which each element has a unique successor

Two Categories

- **Restricted**
  - ✓**STACK (LIFO)**
  - ✓**QUEUE(FIFO)**
- **General**

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 118



Stack

Definition: A linear data structure that follows the LIFO rule for data storage and retrieval.

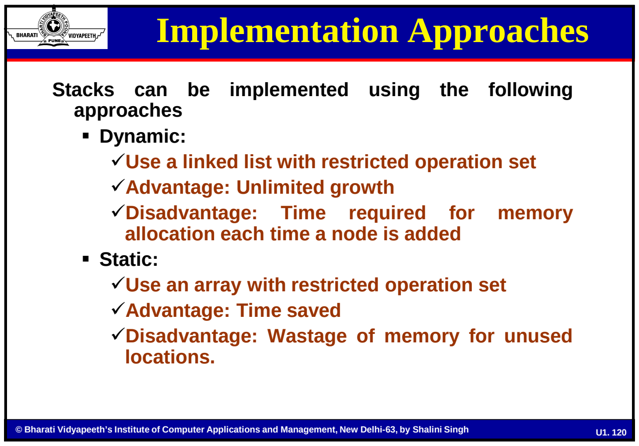
LIFO: Last in First out implies that the element added last would be the first to be retrieved

i.e. the order of retrieval is opposite to the order of storage.

Stacks are frequently used in a number of computer applications like

- expression evaluation,
- recursive function calls,
- implementing storage area for automatic variables etc.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 119

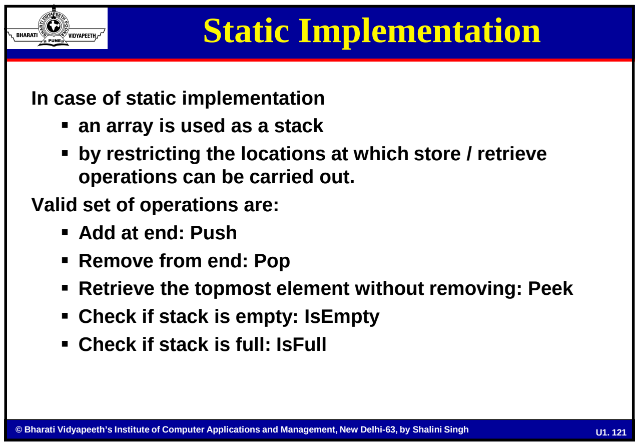


Implementation Approaches

Stacks can be implemented using the following approaches

- **Dynamic:**
  - ✓**Use a linked list with restricted operation set**
  - ✓**Advantage: Unlimited growth**
  - ✓**Disadvantage: Time required for memory allocation each time a node is added**
- **Static:**
  - ✓**Use an array with restricted operation set**
  - ✓**Advantage: Time saved**
  - ✓**Disadvantage: Wastage of memory for unused locations.**

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 120



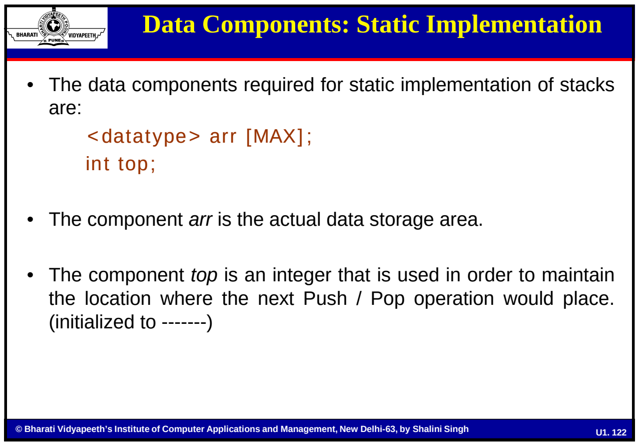
Static Implementation

In case of static implementation

- **an array is used as a stack**
- **by restricting the locations at which store / retrieve operations can be carried out.**

Valid set of operations are:

- **Add at end: Push**
- **Remove from end: Pop**
- **Retrieve the topmost element without removing: Peek**
- **Check if stack is empty: IsEmpty**
- **Check if stack is full: IsFull**

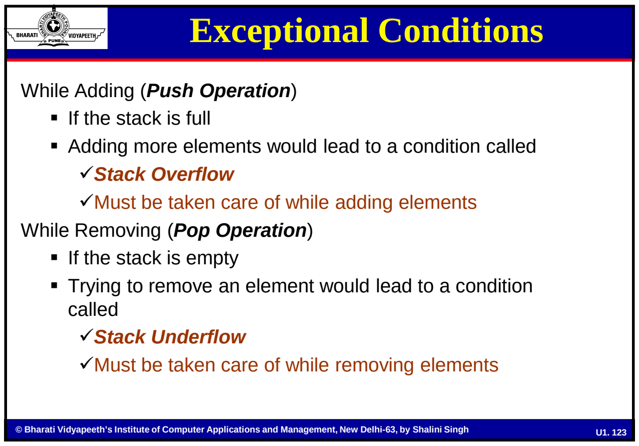


Data Components: Static Implementation

- The data components required for static implementation of stacks are:

```
<datatype> arr [MAX];
int top;
```

- The component *arr* is the actual data storage area.
- The component *top* is an integer that is used in order to maintain the location where the next Push / Pop operation would place. (initialized to -------)



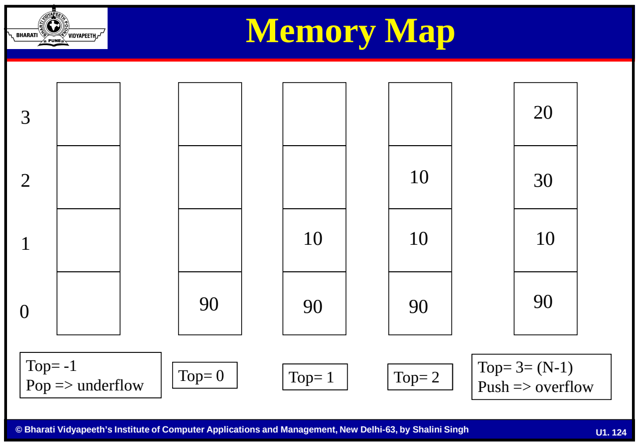
Exceptional Conditions

While Adding (***Push Operation***)

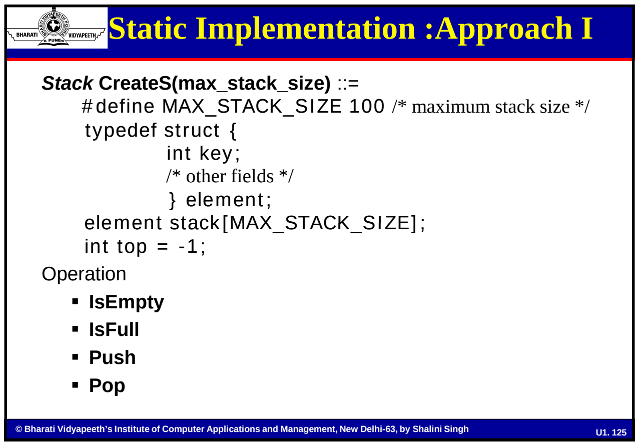
- If the stack is full
- Adding more elements would lead to a condition called
  - ✓***Stack Overflow***
  - ✓Must be taken care of while adding elements

While Removing (***Pop Operation***)

- If the stack is empty
- Trying to remove an element would lead to a condition called
  - ✓***Stack Underflow***
  - ✓Must be taken care of while removing elements



Memory Map



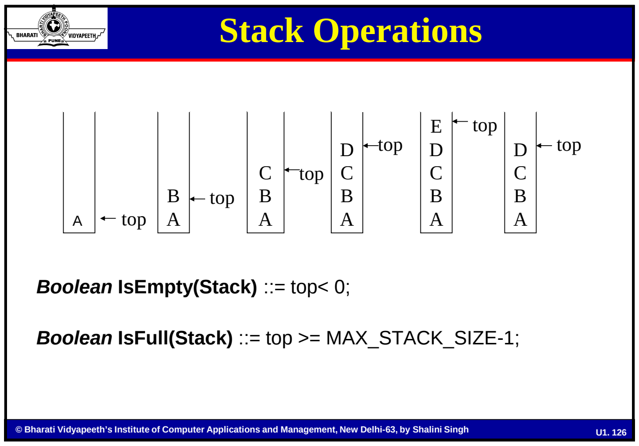
Static Implementation :Approach I

Stack **CreateS(max_stack_size)** ::=

```
    #define MAX_STACK_SIZE 100 /* maximum stack size */
    typedef struct {
            int key;
            /* other fields */
            } element;
    element stack[MAX_STACK_SIZE];
    int top = -1;
```

Operation

- **IsEmpty**
- **IsFull**
- **Push**
- **Pop**



Stack Operations

Boolean **IsEmpty(Stack)** ::= top< 0;

Boolean **IsFull(Stack)** ::= top >= MAX_STACK_SIZE-1;

Stack Add Operation

```
void Push(int *top, element item)
   {
    /* add an item to the global stack */
       if (*top >= MAX_STACK_SIZE-1)  {   /* if(IsFull(*top)) */
           stack_full( );
            return;
       }
       stack[++*top] = item;
   }
```

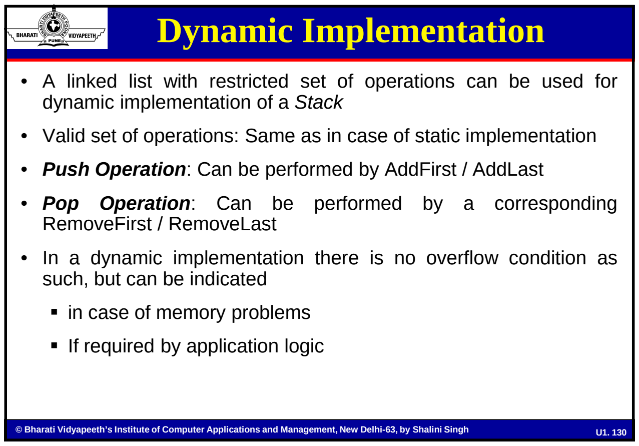
Stack Delete Operation

```
element pop(int *top)
   {
    /* return the top element from the stack */
       if (*top == -1)  /* if(IsEmpty(*top)) */
              return stack_empty( );   /* returns and error key
  */
       return stack[(*top)--];
    }
```

Static Implementation :Approach II

```
#define MAX_SIZE …
typedef struct STACKSTRU {
        int top;
        element elem[MAX_SIZE];
} stack;
stack s;
```

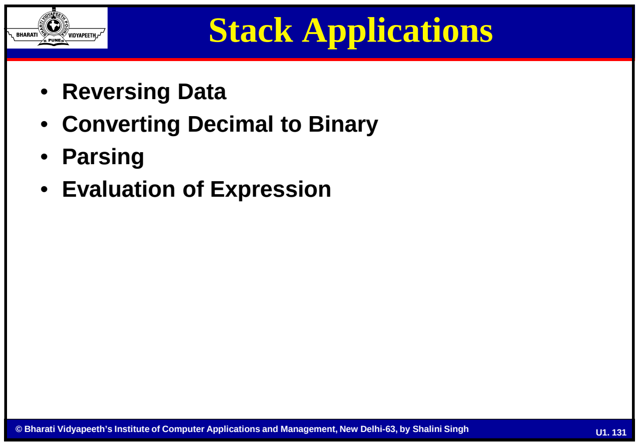
Implement Operations



Dynamic Implementation

- A linked list with restricted set of operations can be used for dynamic implementation of a *Stack*
- Valid set of operations: Same as in case of static implementation
- ***Push Operation***: Can be performed by AddFirst / AddLast
- ***Pop Operation***: Can be performed by a corresponding RemoveFirst / RemoveLast
- In a dynamic implementation there is no overflow condition as such, but can be indicated
  - in case of memory problems
  - If required by application logic

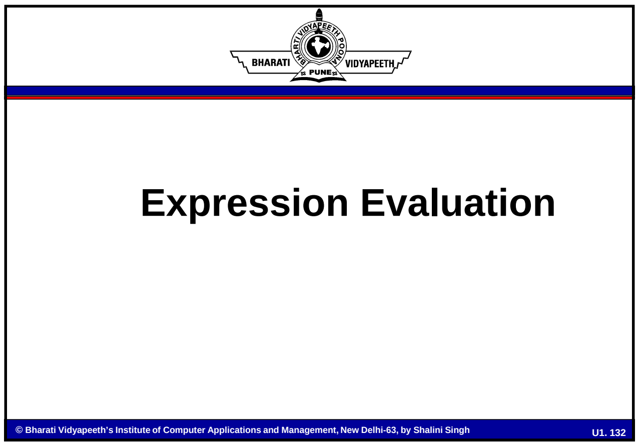
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 130



Stack Applications

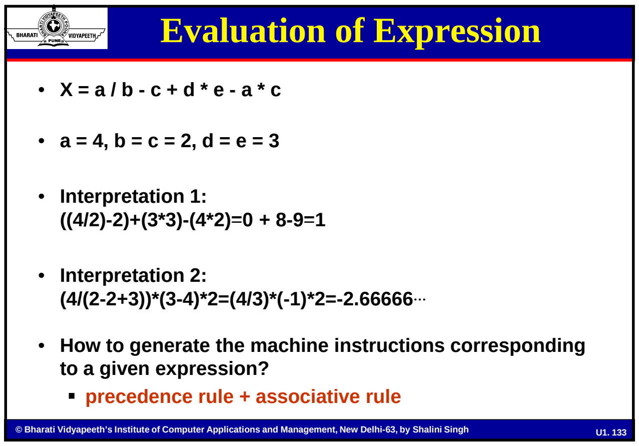
- **Reversing Data**
- **Converting Decimal to Binary**
- **Parsing**
- **Evaluation of Expression**

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 131



Expression Evaluation

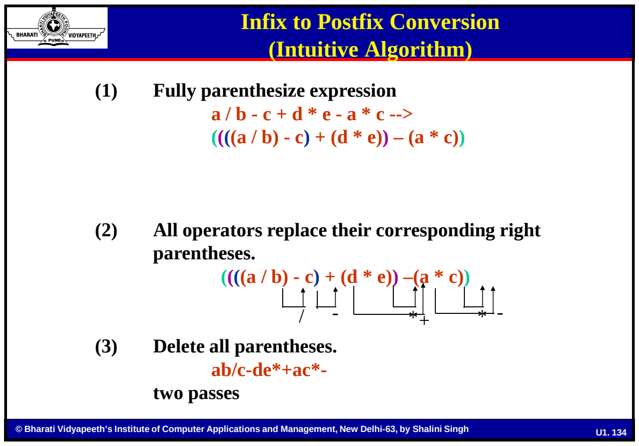
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 132



Evaluation of Expression

- X = a / b - c + d * e - a * c
- a = 4, b = c = 2, d = e = 3
- Interpretation 1:
  ((4/2)-2)+(3*3)-(4*2)=0 + 8-9=1
- Interpretation 2:
  (4/(2-2+3))*(3-4)*2=(4/3)*(-1)*2=-2.66666…
- How to generate the machine instructions corresponding to a given expression?
  - precedence rule + associative rule

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 133



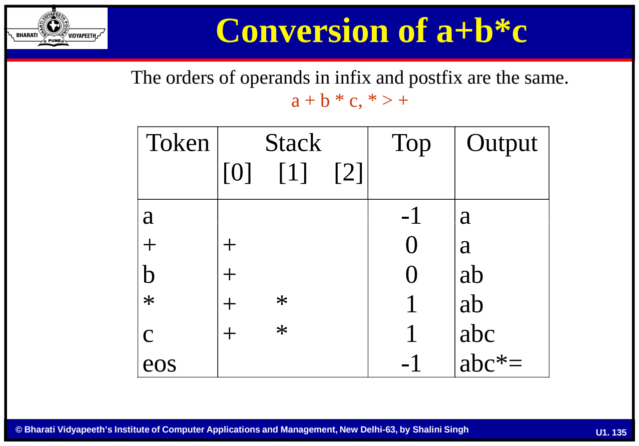
Infix to Postfix Conversion (Intuitive Algorithm)

(1) Fully parenthesize expression
a / b - c + d * e - a * c -->
((((a / b) - c) + (d * e)) – (a * c))

(2) All operators replace their corresponding right parentheses.
((((a / b) - c) + (d * e)) –(a * c))

(3) Delete all parentheses.
ab/c-de*+ac*-
two passes

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 134

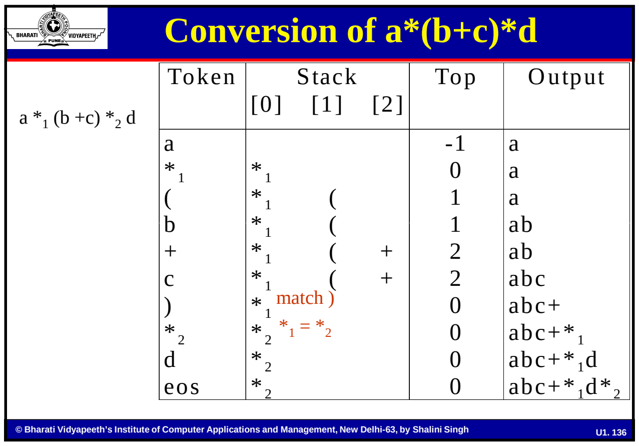


Conversion of a+b*c

The orders of operands in infix and postfix are the same.
a + b * c, * > +

Token	Stack [0]	[1]	[2]	Top	Output
a				-1	a
+	+			0	a
b	+			0	ab
*	+	*		1	ab
c	+	*		1	abc
eos				-1	abc*=

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 135

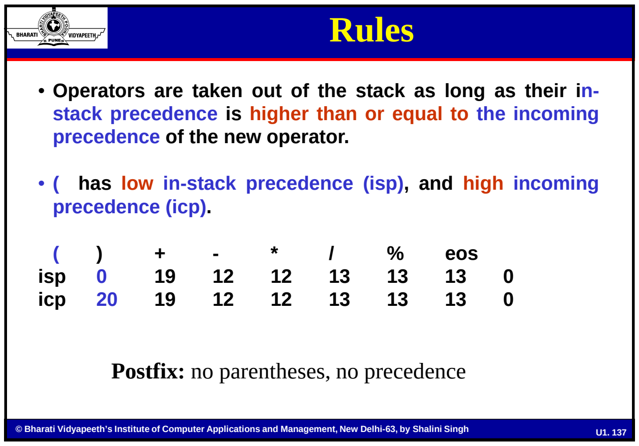


Conversion of a*(b+c)*d

$a *_1 (b + c) *_2 d$

Token	Stack [0]	Stack [1]	Stack [2]	Top	Output
a				-1	a
$*_1$	$*_1$			0	a
(	$*_1$	(		1	a
b	$*_1$	(		1	ab
+	$*_1$	(	+	2	ab
c	$*_1$	(	+	2	abc
)	$*_1$	match)		0	abc+
$*_2$	$*_2$	$*_1 = *_2$		0	$abc+*_1$
d	$*_2$			0	$abc+*_1d$
eos	$*_2$			0	$abc+*_1d*_2$

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 136



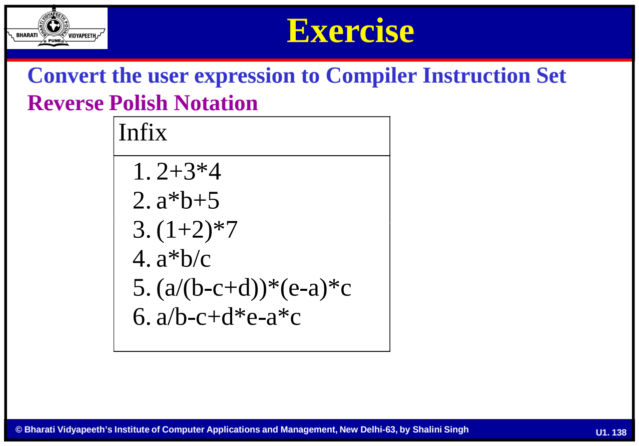
Rules

- Operators are taken out of the stack as long as their in-stack precedence is higher than or equal to the incoming precedence of the new operator.
- (has low in-stack precedence (isp), and high incoming precedence (icp).

	(	)	+	-	*	/	%	eos
isp	0	19	12	12	13	13	13	0
icp	20	19	12	12	13	13	13	0

Postfix: no parentheses, no precedence

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 137

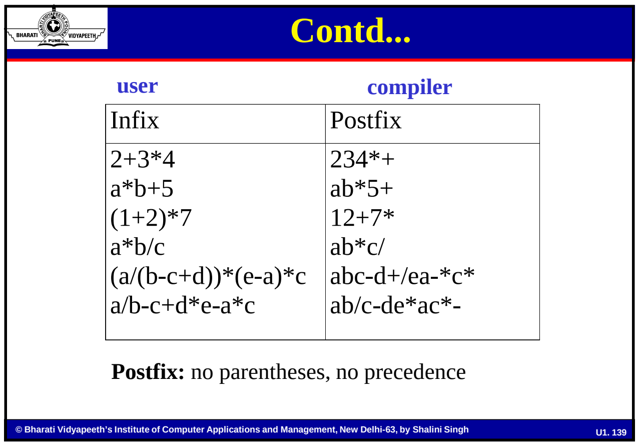


Exercise

Convert the user expression to Compiler Instruction Set
Reverse Polish Notation

Infix
1. 2+3*4 2. a*b+5 3. (1+2)*7 4. a*b/c 5. (a/(b-c+d))*(e-a)*c 6. a/b-c+d*e-a*c

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 138

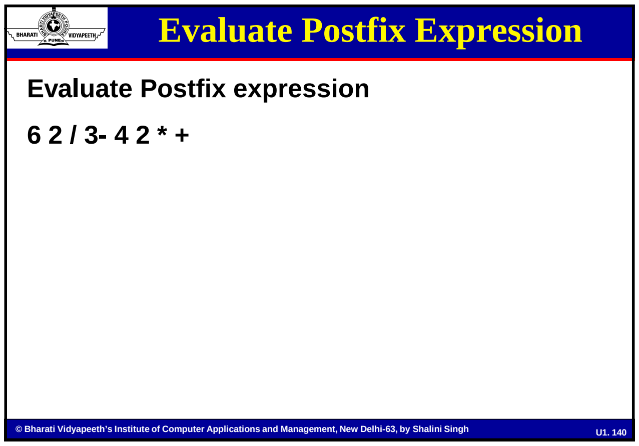


Contd...

user	**compiler**
Infix	Postfix
2+3*4	234*+
a*b+5	ab*5+
(1+2)*7	12+7*
a*b/c	ab*c/
(a/(b-c+d))*(e-a)*c	abc-d+/ea-*c*
a/b-c+d*e-a*c	ab/c-de*ac*-

Postfix: no parentheses, no precedence

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 139

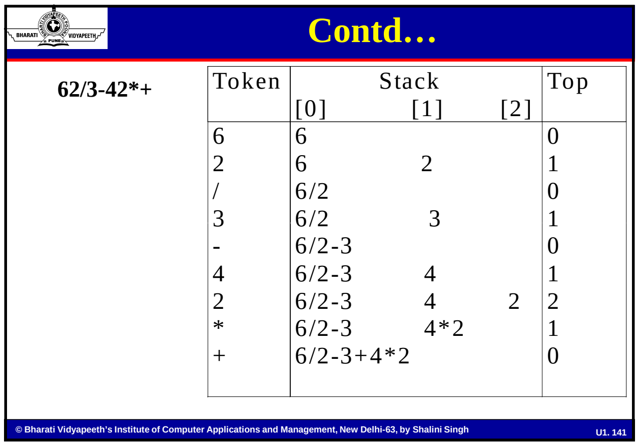


Evaluate Postfix Expression

Evaluate Postfix expression

6 2 / 3- 4 2 * +

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 140

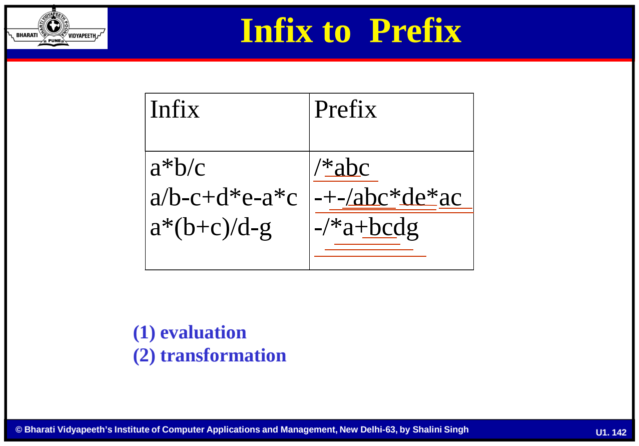


Contd…

62/3-42*+

Token	Stack [0]	[1]	[2]	Top
6	6			0
2	6	2		1
/	6/2			0
3	6/2	3		1
-	6/2-3			0
4	6/2-3	4		1
2	6/2-3	4	2	2
*	6/2-3	4*2		1
+	6/2-3+4*2			0

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 141

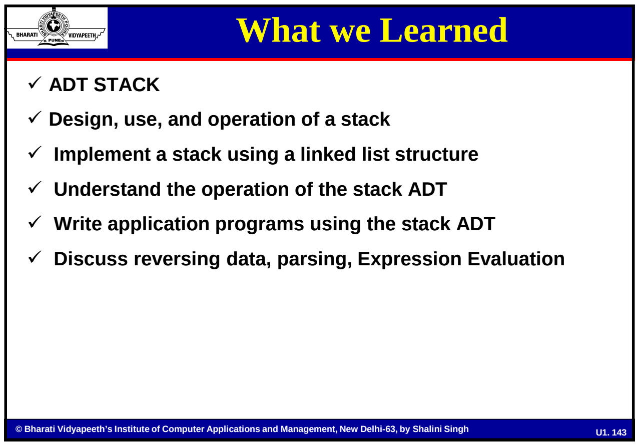


Infix to Prefix

Infix	Prefix
a*b/c a/b-c+d*e-a*c a*(b+c)/d-g	/*abc -+-/abc*de*ac -/*a+bcdg

(1) evaluation
(2) transformation

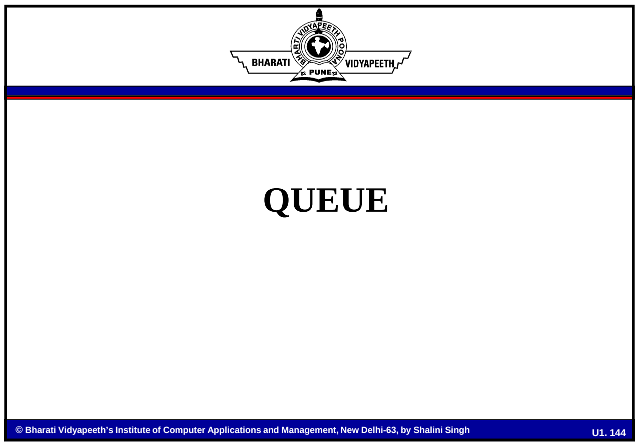
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 142



What we Learned

- ✓ ADT STACK
- ✓ Design, use, and operation of a stack
- ✓ Implement a stack using a linked list structure
- ✓ Understand the operation of the stack ADT
- ✓ Write application programs using the stack ADT
- ✓ Discuss reversing data, parsing, Expression Evaluation

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 143



QUEUE

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 144

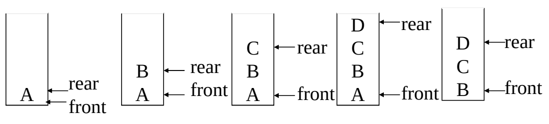
Objectives

- ADT Queue
- Explain the design, use, and operation of a Queue
- Array Implementation of Queue
- Understand the operation of the Queue ADT
- Implement a Queue using a linked list structure
- Multi Stack
- Multi Queue

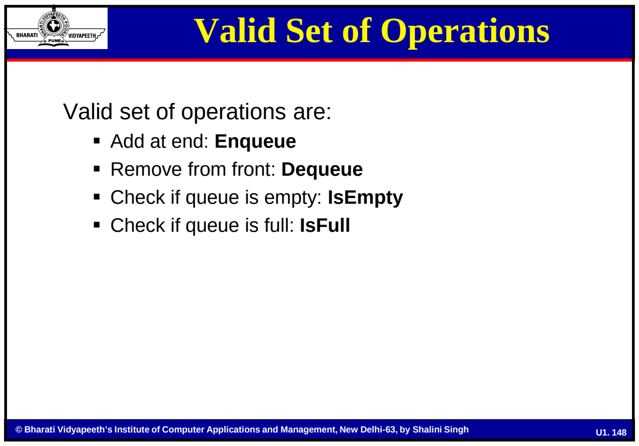
Queue

- **Definition**: A linear data structure that follows the FIFO rule for data storage and retrieval.
- **FIFO**: First in First out implies that the element added first would be the first to be retrieved
- i.e. the order of retrieval is the same as the order of storage.
- Queues are frequently used in a number of computer applications like
  - Sequential processing of data
  - Process Scheduling etc.

Queue: a First-In-First-Out (FIFO) List



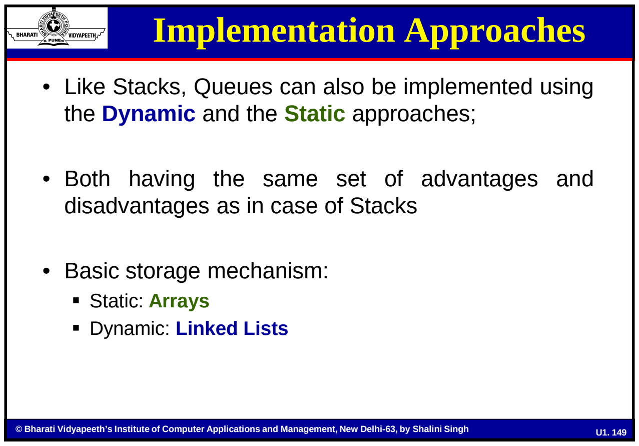
Inserting and deleting elements in a queue



Valid Set of Operations

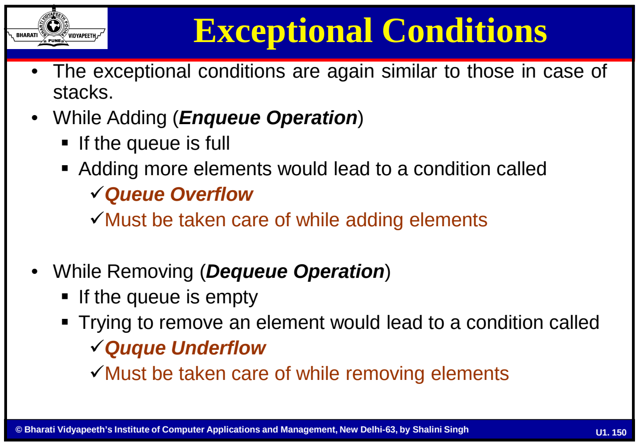
Valid set of operations are:

- Add at end: **Enqueue**
- Remove from front: **Dequeue**
- Check if queue is empty: **IsEmpty**
- Check if queue is full: **IsFull**



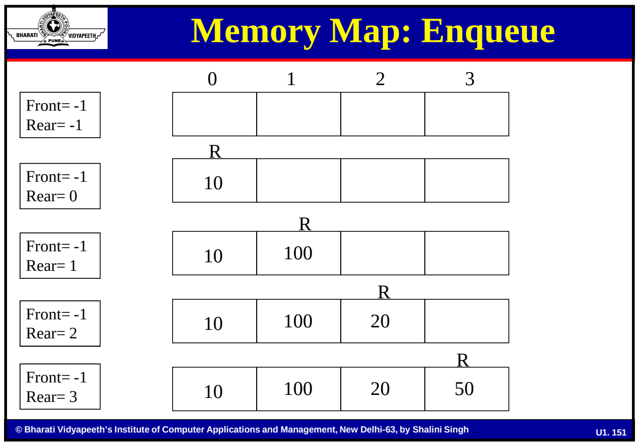
Implementation Approaches

- Like Stacks, Queues can also be implemented using the **Dynamic** and the **Static** approaches;
- Both having the same set of advantages and disadvantages as in case of Stacks
- Basic storage mechanism:
  - Static: **Arrays**
  - Dynamic: **Linked Lists**



Exceptional Conditions

- The exceptional conditions are again similar to those in case of stacks.
- While Adding (***Enqueue Operation***)
  - If the queue is full
  - Adding more elements would lead to a condition called
    - ✓***Queue Overflow***
    - ✓Must be taken care of while adding elements
- While Removing (***Dequeue Operation***)
  - If the queue is empty
  - Trying to remove an element would lead to a condition called
    - ✓***Ququе Underflow***
    - ✓Must be taken care of while removing elements



Memory Map: Enqueue

	0	1	2	3

Front= -1
Rear= -1				

	R			

Front= -1
Rear= 0
	10			

		R		

Front= -1
Rear= 1
	10	100		

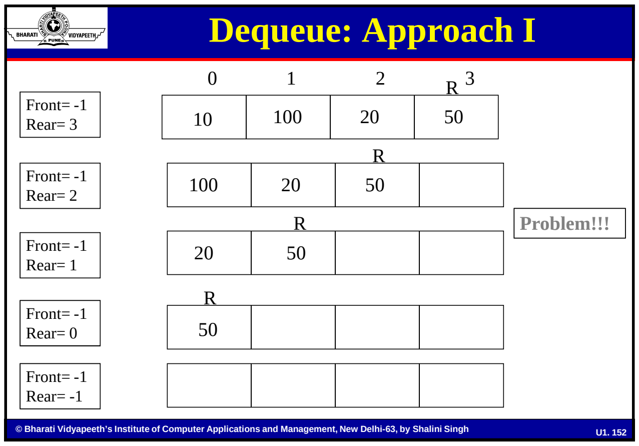
			R	

Front= -1
Rear= 2
	10	100	20	

				R

Front= -1
Rear= 3
	10	100	20	50

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 151



Dequeue: Approach I

	0	1	2	3
				R

Front= -1
Rear= 3
	10	100	20	50

			R	

Front= -1
Rear= 2
	100	20	50	

		R		

Front= -1
Rear= 1
	20	50		

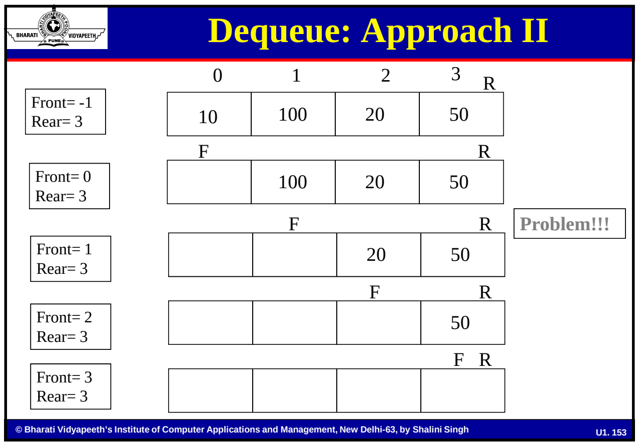
	R			

Front= -1
Rear= 0
	50			

Front= -1
Rear= -1				

Problem!!!

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 152



Dequeue: Approach II

	0	1	2	3
				R

Front= -1
Rear= 3
	10	100	20	50

	F			R

Front= 0
Rear= 3
		100	20	50

		F		R

Front= 1
Rear= 3
			20	50

			F	R

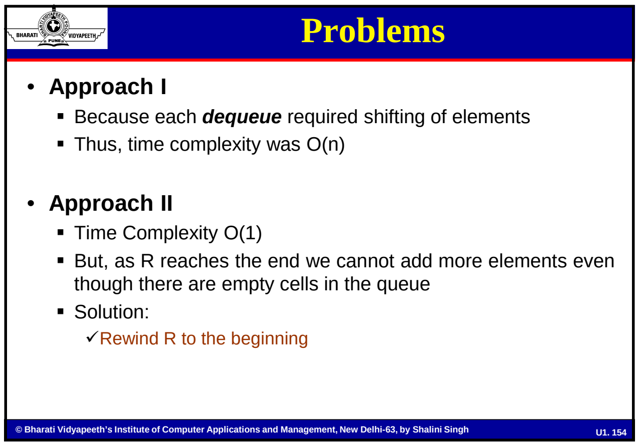
Front= 2
Rear= 3
				50

				F R

Front= 3
Rear= 3				

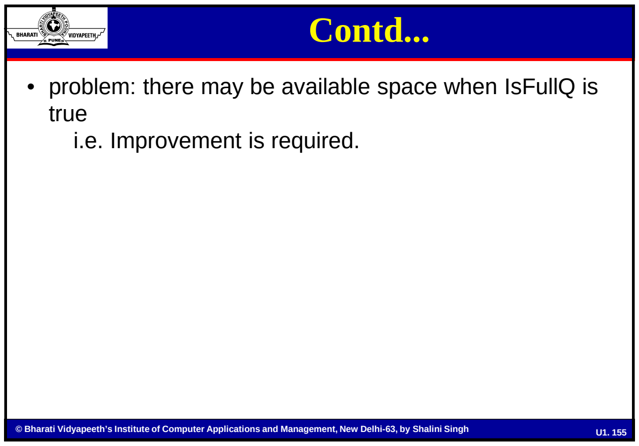
Problem!!!

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 153



Problems

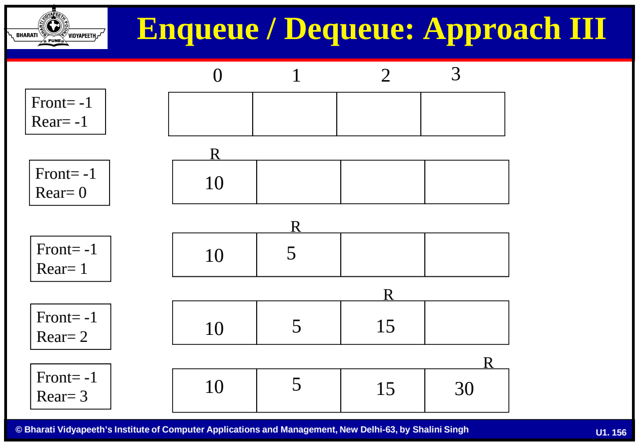
- **Approach I**
  - Because each ***dequeue*** required shifting of elements
  - Thus, time complexity was O(n)
- **Approach II**
  - Time Complexity O(1)
  - But, as R reaches the end we cannot add more elements even though there are empty cells in the queue
  - Solution:
    - ✓Rewind R to the beginning



Contd...

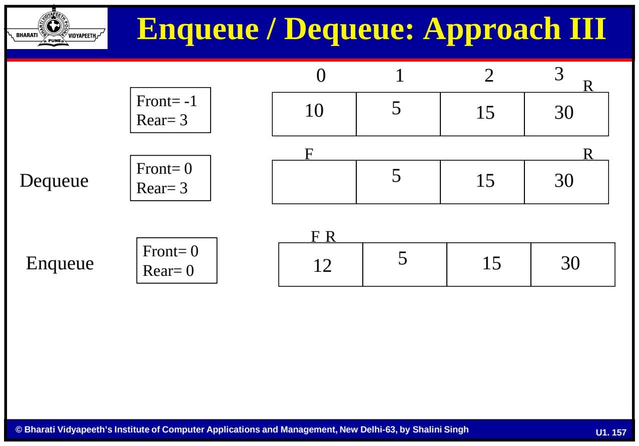
- problem: there may be available space when IsFullQ is true

  i.e. Improvement is required.



Enqueue / Dequeue: Approach III

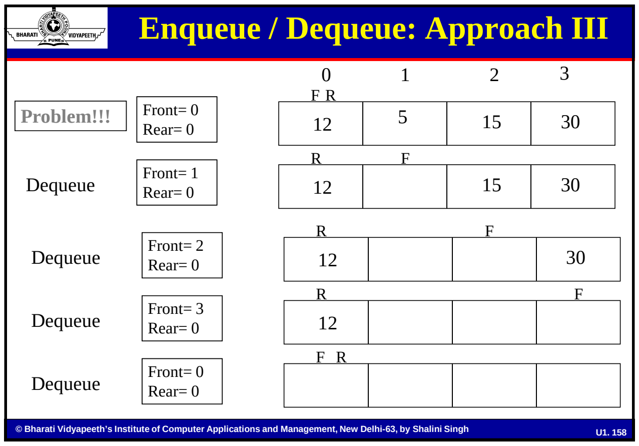
	0	1	2	3
Front= -1 Rear= -1				
Front= -1 Rear= 0	10 (R)			
Front= -1 Rear= 1	10	5 (R)		
Front= -1 Rear= 2	10	5	15 (R)	
Front= -1 Rear= 3	10	5	15	30 (R)



Enqueue / Dequeue: Approach III

		0	1	2	3
					R
	Front= -1 Rear= 3	10	5	15	30
		F			R
Dequeue	Front= 0 Rear= 3		5	15	30
		F R			
Enqueue	Front= 0 Rear= 0	12	5	15	30

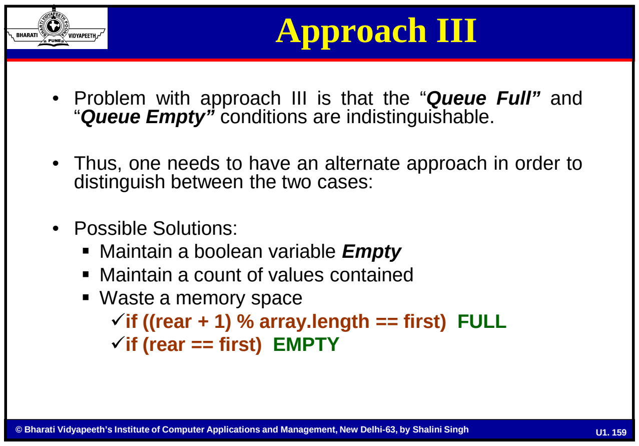
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 157



Enqueue / Dequeue: Approach III

		0	1	2	3
		F R			
Problem!!!	Front= 0 Rear= 0	12	5	15	30
		R	F		
Dequeue	Front= 1 Rear= 0	12		15	30
		R		F	
Dequeue	Front= 2 Rear= 0	12			30
		R			F
Dequeue	Front= 3 Rear= 0	12			
		F R			
Dequeue	Front= 0 Rear= 0				

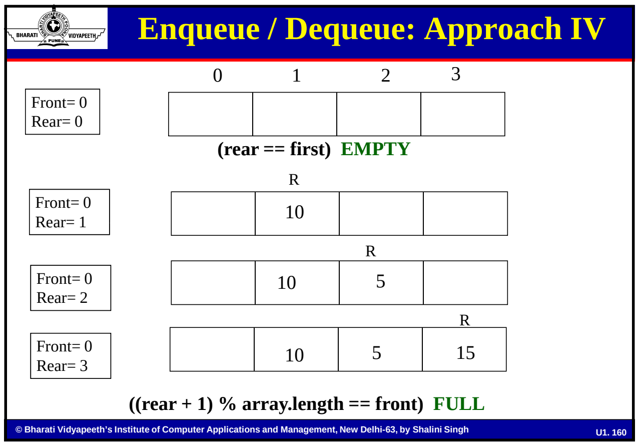
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 158



Approach III

- Problem with approach III is that the "***Queue Full"*** and "***Queue Empty"*** conditions are indistinguishable.
- Thus, one needs to have an alternate approach in order to distinguish between the two cases:
- Possible Solutions:
  - Maintain a boolean variable ***Empty***
  - Maintain a count of values contained
  - Waste a memory space
    - ✓**if ((rear + 1) % array.length == first) FULL**
    - ✓**if (rear == first) EMPTY**

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 159



Enqueue / Dequeue: Approach IV

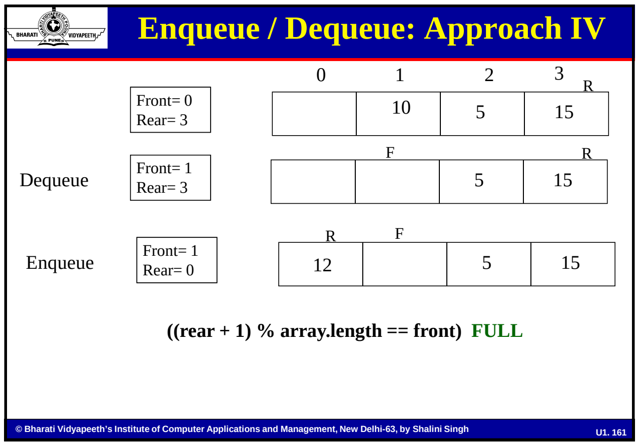
	0	1	2	3
Front= 0 Rear= 0				

(rear == first) EMPTY

		R		
Front= 0 Rear= 1		10		
			R	
Front= 0 Rear= 2		10	5	
				R
Front= 0 Rear= 3		10	5	15

((rear + 1) % array.length == front) FULL

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 160

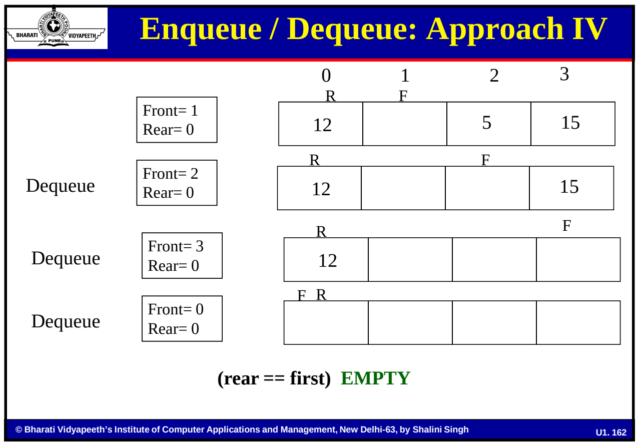


Enqueue / Dequeue: Approach IV

		0	1	2	3
					R
	Front= 0 Rear= 3		10	5	15
			F		R
Dequeue	Front= 1 Rear= 3			5	15
		R	F		
Enqueue	Front= 1 Rear= 0	12		5	15

((rear + 1) % array.length == front) FULL

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 161



Enqueue / Dequeue: Approach IV

		0	1	2	3
		R	F		
	Front= 1 Rear= 0	12		5	15
		R		F	
Dequeue	Front= 2 Rear= 0	12			15
		R			F
Dequeue	Front= 3 Rear= 0	12			
		F R			
Dequeue	Front= 0 Rear= 0				

(rear == first) EMPTY

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 162

Data Components: Static Implementation

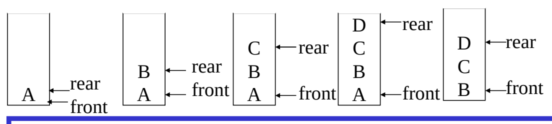
- As data storage and retrieval take place at two opposite ends:
  - ✓<datatype> arr [MAX];
  - ✓int front;
  - ✓int rear;
- The component ***arr*** is the actual data storage area.
- The component ***front*** is an integer maintains the location where the next **Dequeue** operation would place.
- The component ***rear*** is an integer maintains the location where the next **Enqueue** operation would place.

Implementation : Using Array

Queue CreateQ(*max_queue_size*) ::=

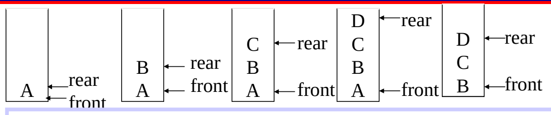
```
# define MAX_QUEUE_SIZE 100

typedef struct {
                int key;
                /* other fields */
                } element;
element queue[MAX_QUEUE_SIZE];
int rear = -1;
int front = -1;
```

Contd...



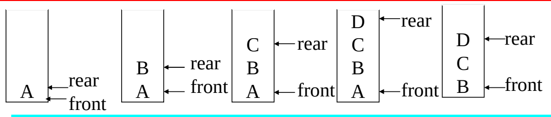
```
int rear = -1;
int front = -1;
Boolean IsEmpty(queue) ::=
        front == rear
Boolean IsFullQ(queue) ::=
        rear == MAX_QUEUE_SIZE-1
```

Add to a Queue



```
void addq(int *rear, element item)
{/* add an item to the queue */
   if (*rear == MAX_QUEUE_SIZE_1) {
      queue_full( );
      return;
  }
  queue [++*rear] = item;
}
```

Delete From a Queue



```
element deleteq(int *front, int rear)
{/* remove element at the front of the queue */
   if ( *front == rear)
       return queue_empty( );     /* return an error
key */
   return queue [++ *front];
}
```

Dynamic Implementation

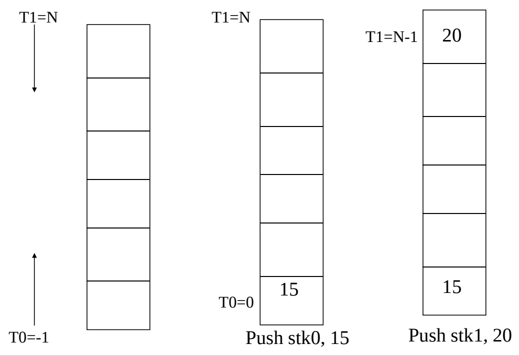
- A linked list with restricted set of operations can be used for dynamic implementation of a *Queue* with the same set of valid operations as in case of static implementation
- ***Enqueue Operation***: Can be performed by AddFirst / AddLast
- ***Dequeue Operation***: Can be performed by a corresponding RemoveLast / RemoveFirst
- Limited Memory or application logic can again be used to indicate overflow condition if required.

Double Stack

- Useful when there are two categories of data items
- Total number of entities is fixed, but proportion isn't known
- Same memory area shared related variables

- Operations
  - As the same memory area is logically divided into two stacks
  - Each end serves as the reference point ***(top)*** for one of the two stacks
  - ***Push*** and ***Pop*** operations can take place on either end
  - While performing an operation, user needs to specify the stack number on which the operation is to be carried out.

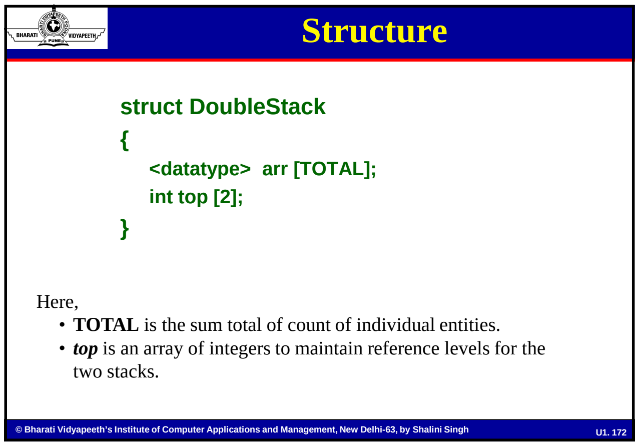
Memory Representation



Exceptional Conditions

Overflow Condition
- Should take crossover into consideration
- ------------------------

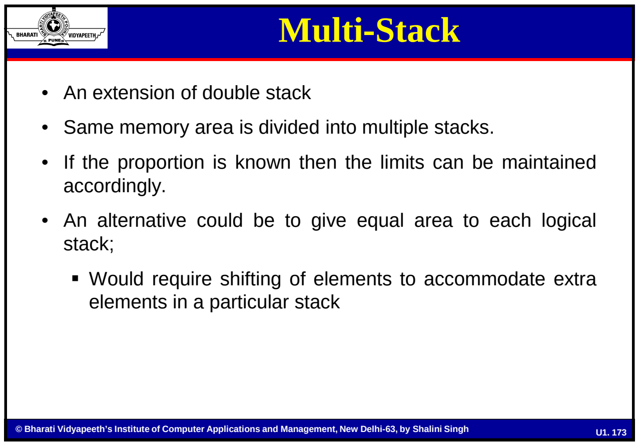
Underflow Condition
- Needs to be maintained individually for each stack
- For Stack 0->
- For Stack 1->



Structure

```
struct DoubleStack
{
    <datatype>  arr [TOTAL];
    int top [2];
}
```

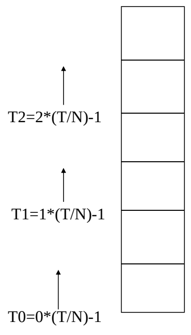
Here,
- **TOTAL** is the sum total of count of individual entities.
- ***top*** is an array of integers to maintain reference levels for the two stacks.

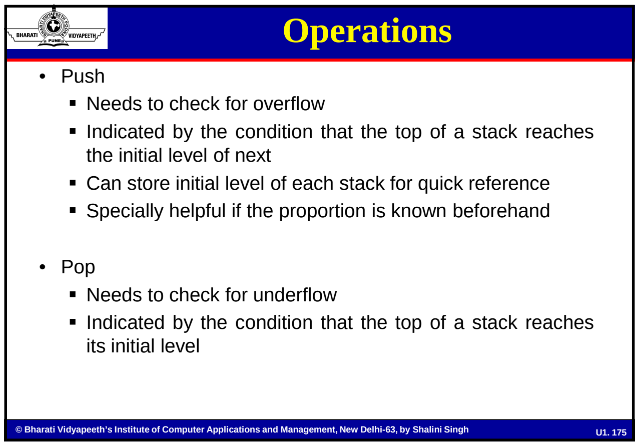


Multi-Stack

- An extension of double stack
- Same memory area is divided into multiple stacks.
- If the proportion is known then the limits can be maintained accordingly.
- An alternative could be to give equal area to each logical stack;
  - Would require shifting of elements to accommodate extra elements in a particular stack

Memory Representation

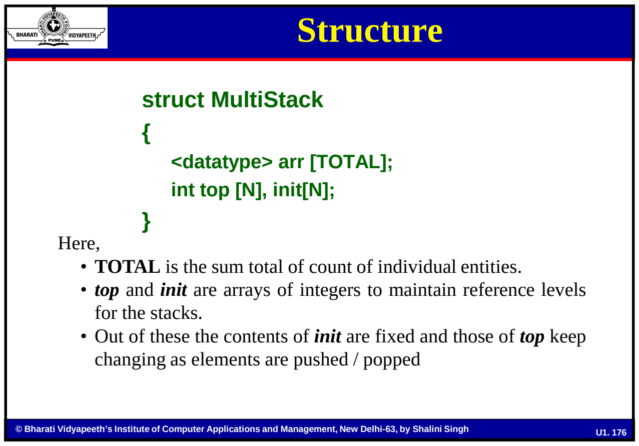




Operations

- Push
  - Needs to check for overflow
  - Indicated by the condition that the top of a stack reaches the initial level of next
  - Can store initial level of each stack for quick reference
  - Specially helpful if the proportion is known beforehand
- Pop
  - Needs to check for underflow
  - Indicated by the condition that the top of a stack reaches its initial level

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 175



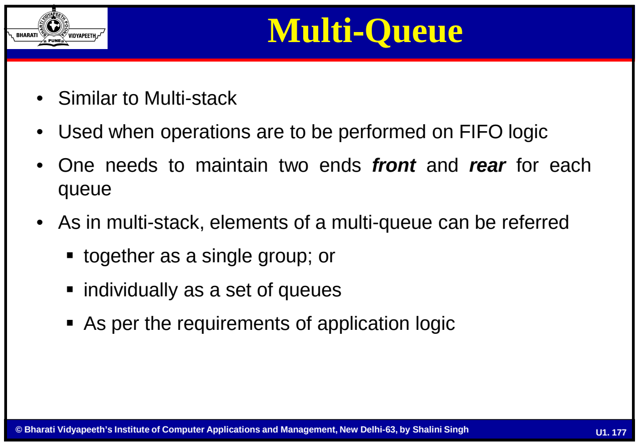
Structure

```
struct MultiStack
{
    <datatype> arr [TOTAL];
    int top [N], init[N];
}
```

Here,

- **TOTAL** is the sum total of count of individual entities.
- ***top*** and ***init*** are arrays of integers to maintain reference levels for the stacks.
- Out of these the contents of ***init*** are fixed and those of ***top*** keep changing as elements are pushed / popped

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 176



Multi-Queue

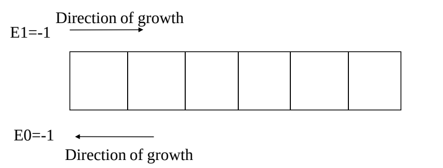
- Similar to Multi-stack
- Used when operations are to be performed on FIFO logic
- One needs to maintain two ends ***front*** and ***rear*** for each queue
- As in multi-stack, elements of a multi-queue can be referred
  - together as a single group; or
  - individually as a set of queues
  - As per the requirements of application logic

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 177

Deque

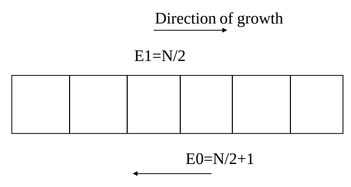
- A deque: is a data structure that allows input or output operations at either of its ends.
  - i.e. elements can be added or removed from both ends but not from in-between.
- Can be used as a stack or a queue as per the requirements of application logic by restricting operations at the ends.
- Variations:
  - Input restricted ***Deque***
  - Output restricted ***Deque***

Approach I

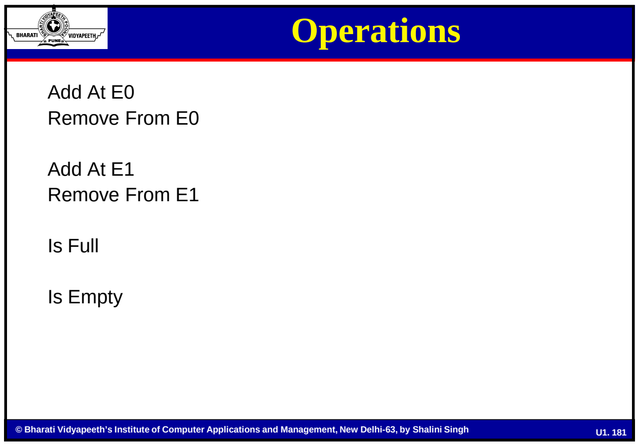


Direction of growth
=> Need to shift / wrap around while adding data initially

Approach II



Direction of growth
=> Initial addition doesn't require shifting



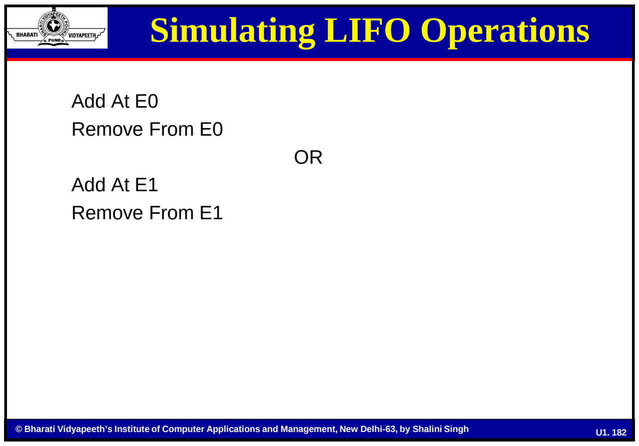
Operations

Add At E0
Remove From E0

Add At E1
Remove From E1

Is Full

Is Empty

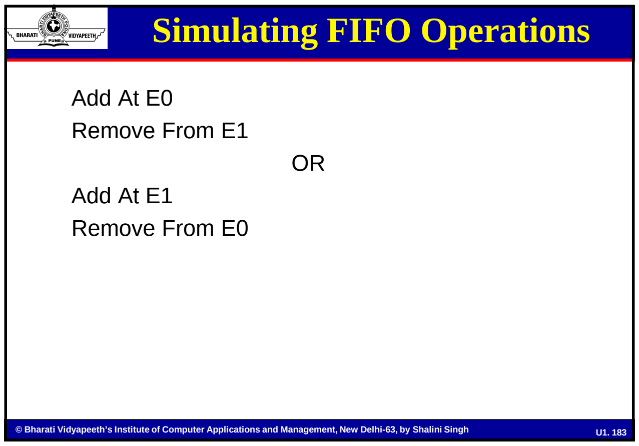


Simulating LIFO Operations

Add At E0
Remove From E0

OR

Add At E1
Remove From E1

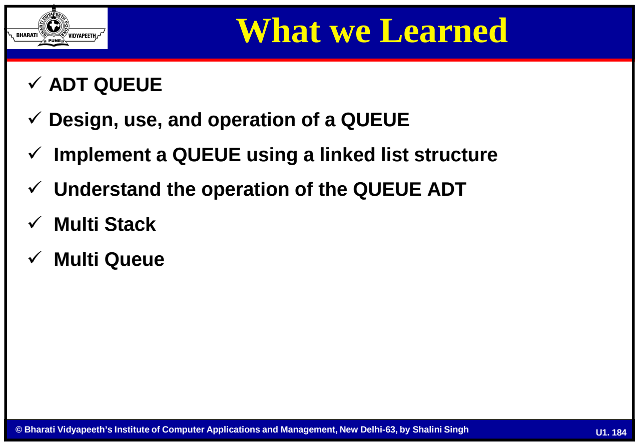


Simulating FIFO Operations

Add At E0
Remove From E1

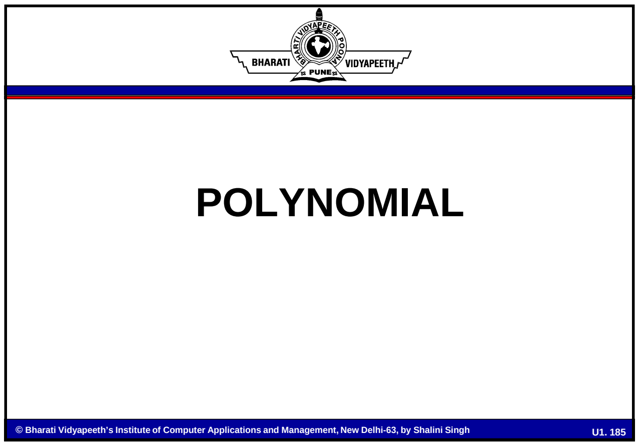
OR

Add At E1
Remove From E0

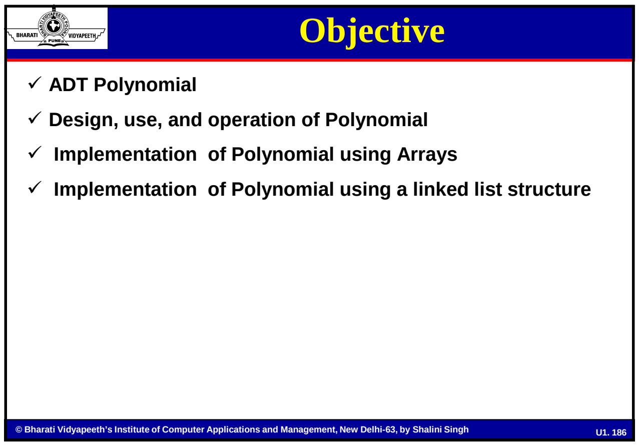


What we Learned

- ✓ ADT QUEUE
- ✓ Design, use, and operation of a QUEUE
- ✓ Implement a QUEUE using a linked list structure
- ✓ Understand the operation of the QUEUE ADT
- ✓ Multi Stack
- ✓ Multi Queue



POLYNOMIAL



Objective

- ✓ ADT Polynomial
- ✓ Design, use, and operation of Polynomial
- ✓ Implementation of Polynomial using Arrays
- ✓ Implementation of Polynomial using a linked list structure

Polynomial

- homogeneous ordered list of pairs <*exponent,coefficient*>, where each coefficient is unique.

Operations include

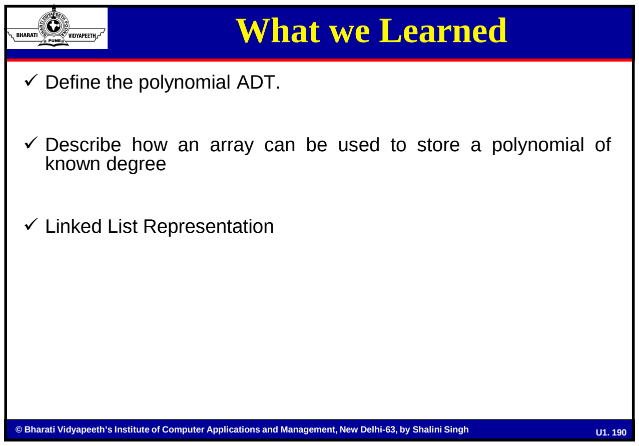
- **Returning the degree**
- **Extracting the coefficient for a given exponent**
- **Addition**
- **Multiplication**

Array Implementation

- One may map choose to map array Indices with the power of the term
  - Helpful if the number of missing terms is minimal
  - Helps in saving time but wastes memory if missing terms are large.
- In case this mapping is not used
  - memory is saved but
  - Time is wasted in
    - ✓searching for the required term
    - ✓Adding or deleting terms with intermediate powers

Linked List Representation

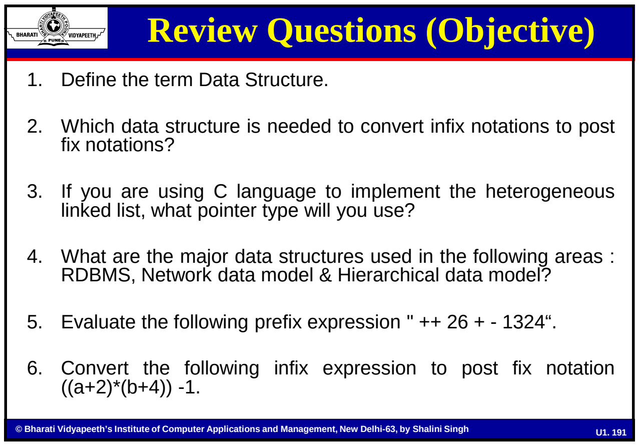
```
Typedef struct Node *PtrToNode;

struct Term{
    int Coefficient;
    int Exponent;
};

struct Node
{
        struct Tern term;
        PtrToNode Next;
};
Typedef PtrToNode Polynomial;
```



What we Learned

- ✓ Define the polynomial ADT.
- ✓ Describe how an array can be used to store a polynomial of known degree
- ✓ Linked List Representation

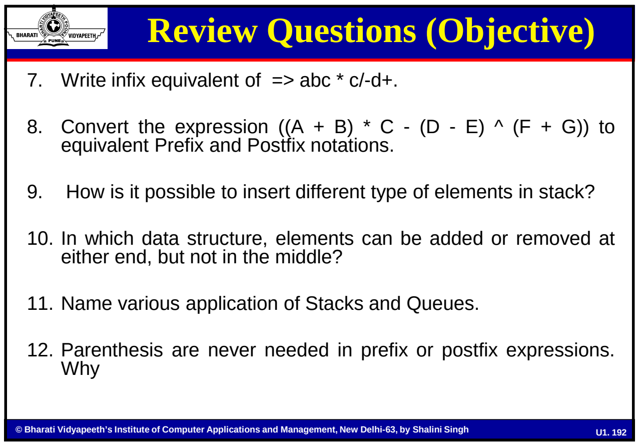
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 190



Review Questions (Objective)

1. Define the term Data Structure.
2. Which data structure is needed to convert infix notations to post fix notations?
3. If you are using C language to implement the heterogeneous linked list, what pointer type will you use?
4. What are the major data structures used in the following areas : RDBMS, Network data model & Hierarchical data model?
5. Evaluate the following prefix expression " ++ 26 + - 1324“.
6. Convert the following infix expression to post fix notation ((a+2)*(b+4)) -1.

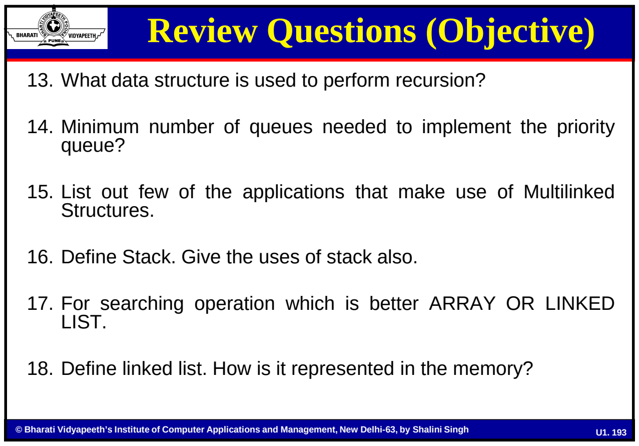
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 191



Review Questions (Objective)

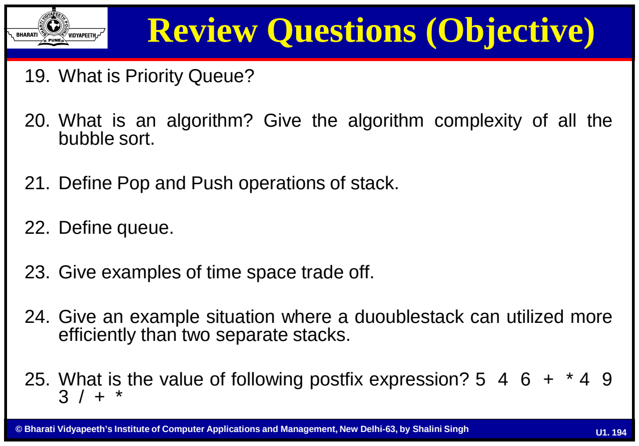
7. Write infix equivalent of => abc * c/-d+.
8. Convert the expression ((A + B) * C - (D - E) ^ (F + G)) to equivalent Prefix and Postfix notations.
9. How is it possible to insert different type of elements in stack?
10. In which data structure, elements can be added or removed at either end, but not in the middle?
11. Name various application of Stacks and Queues.
12. Parenthesis are never needed in prefix or postfix expressions. Why

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Shalini Singh U1. 192



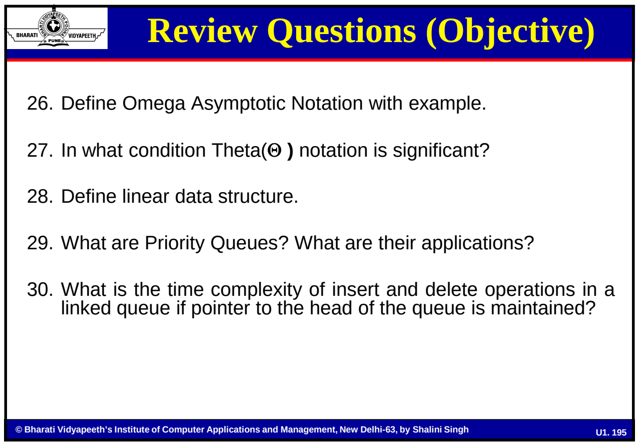
Review Questions (Objective)

13. What data structure is used to perform recursion?
14. Minimum number of queues needed to implement the priority queue?
15. List out few of the applications that make use of Multilinked Structures.
16. Define Stack. Give the uses of stack also.
17. For searching operation which is better ARRAY OR LINKED LIST.
18. Define linked list. How is it represented in the memory?



Review Questions (Objective)

19. What is Priority Queue?
20. What is an algorithm? Give the algorithm complexity of all the bubble sort.
21. Define Pop and Push operations of stack.
22. Define queue.
23. Give examples of time space trade off.
24. Give an example situation where a duoublestack can utilized more efficiently than two separate stacks.
25. What is the value of following postfix expression? 5 4 6 + * 4 9 3 / + *



Review Questions (Objective)

26. Define Omega Asymptotic Notation with example.
27. In what condition Theta(Θ) notation is significant?
28. Define linear data structure.
29. What are Priority Queues? What are their applications?
30. What is the time complexity of insert and delete operations in a linked queue if pointer to the head of the queue is maintained?

Review Questions (Objective)

31. While deleting a node from a linked list one must remember to ------.
32. Why do we need to pass the starting node of the list as a reference parameter to the function AddAtBegin?
33. Name an operation that can be performed more efficiently on an array as compared to a linked list.
34. What are the exception conditions for adding and deleting item from a stack?
35. Write an application where we use the Circular Linked List.

U1. 196

Review Questions (Short Type)

1. What is recursion and what are its overheads?
2. Give the difference between linked list and array.
3. What are polish notations? Explain their advantages.
4. Give the difference between field, record and file.
5. What is an entity? Give example.
6. Compare linear and binary search.
7. Define the operations of data structure with example.

U1. 197

Review Questions (Short Type)

8. What does the term abstract data type mean? Give Example.
9. Why do we go for dynamic data storage ?
10. List out the areas in which data structures are applied extensively?
11. Write an algorithm to check whether a given string is palindrome or not using stacks.
12. In what way, double linked list is better than single link list? Give example.
13. Write a non-recursive algorithm to insert a node into a single-list list.

U1. 198

Review Questions (Short Type)

14. Given an Array A(20.. 50,20..40). The elements are stored in Column Major Order. What is the starting location of A (32,23)

15. Write an algorithm for
    1. Delete item from
    2. Linear queue
    3. Circular queue

16. Write an algorithm to add items to A) Linear queue b) Circular queue.

17. Write algorithm to perform following operations
    1. Concatenate two lists
    2. Reverse a list

Review Questions (Short Type)

18. Give the algorithm of inserting the node into beginning of linked list. With example.

19. What data structure would you mostly likely see in a non recursive implementation of a recursive algorithm?

20. Give the algorithm of searching in linked list when list is sorted.

21. Write an algorithm to invert a given linked list if the pointer to first node (First) is given.

22. Define Queue and its type. Give insertion and deletion algorithms of linear queue.

23. Define garbage collection with example.

Review Questions (Short Type)

24. Define the overflow and underflow conditions in circular queue .with example.

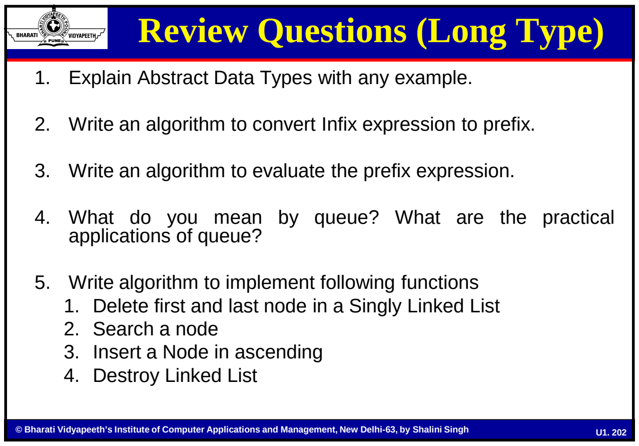
25. Write the algorithm of deleting the node following a given node in linked list.

26. What is memory leakage?

27. Given an integer K, write a procedure which deletes the K^{th} element from linked list.

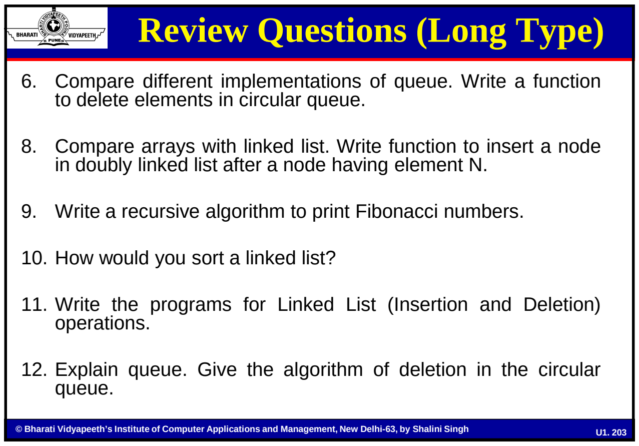
28. Give the difference between linear and circular linked list

29. Write the algorithm of inserting the node in a linear linked list.



Review Questions (Long Type)

1. Explain Abstract Data Types with any example.
2. Write an algorithm to convert Infix expression to prefix.
3. Write an algorithm to evaluate the prefix expression.
4. What do you mean by queue? What are the practical applications of queue?
5. Write algorithm to implement following functions
   1. Delete first and last node in a Singly Linked List
   2. Search a node
   3. Insert a Node in ascending
   4. Destroy Linked List



Review Questions (Long Type)

6. Compare different implementations of queue. Write a function to delete elements in circular queue.

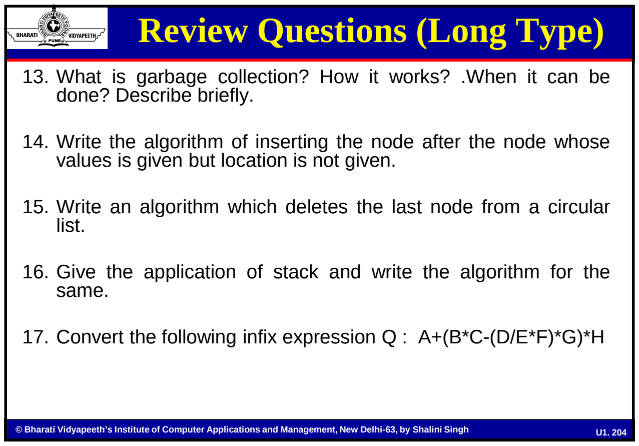
8. Compare arrays with linked list. Write function to insert a node in doubly linked list after a node having element N.

9. Write a recursive algorithm to print Fibonacci numbers.

10. How would you sort a linked list?

11. Write the programs for Linked List (Insertion and Deletion) operations.

12. Explain queue. Give the algorithm of deletion in the circular queue.



Review Questions (Long Type)

13. What is garbage collection? How it works? .When it can be done? Describe briefly.

14. Write the algorithm of inserting the node after the node whose values is given but location is not given.

15. Write an algorithm which deletes the last node from a circular list.

16. Give the application of stack and write the algorithm for the same.

17. Convert the following infix expression Q : A+(B*C-(D/E*F)*G)*H

Review Questions (Long Type)

18. Define data structure. List common operations which can be performed on data structures. Discuss the complexity of an algorithm. How can it be measured. Discuss in brief.

19. What do you mean by linked list. What are its advantages. Write an algorithm to insert and delete a node from a linked list

20. Suppose there are two sorted lists L1 and L2. Write an algorithm to store the result of both L1 L2 in the List L3. The list L3 must be in sorted order. What will be the time complexity of the algorithm?

References

- E. Horowitz and S. Sahani, "Fundamentals of Data Structures in C", 2nd Edition, Universities Press, 2008.
- Mark Allen Weiss, "Data Structures and Algorithm Analysis in C", 2nd Edition Addison-Wesley, 1997.
- Schaum's Outline Series, "Data Structure", TMH, Special Indian Ed., Seventeenth Reprint, 2009.
- Y. Langsam et. al., "Data Structures using C and C++", PHI, 1999.
- N. Dale and S.C. Lilly, D.C. Heath and Co., "Data Structures", 1995.
- Mary E. S. Loomes, "Data Management and File Structure", PHI, 2nd Ed., 1989.

References

- http://www.ceglug.org/index.php/labs/67-circularly-singly-linked-list
- http://www.brpreiss.com/books/opus4/html/page158.html
- http://cset.sp.utoledo.edu/cset3150/chap42.pdf
- http://frank.mtsu.edu/~csci217/manual/lab10/lab10.html