Data Structures and Algorithms

Before We Start

Water : Container ⇔ Data : Data Structure




What is Data Structure?

- It is the structure of the data
- It is the container of the data
- It is the conceptual form of the data
- It is a mathematical model of the data

Data Structure and its Importance

- A data structure is a particular way of **organizing data in a computer** so that it can be used **effectively**.
- With a sufficient understanding of data structure **data can be organized and stored in a proper manner.**
- It is an **effective way of performing various operations** related to data management.
- It is a key part of many computer algorithms as it allows the programmers to do **data management in an efficient way.**
- A right selection of data structure can **enhance the efficiency of computer program or algorithm** in a better way.

Types of Data Structure

- Linear and Non-Linear
- **Linear**
  - Array,
  - Stack,
  - Queue,
  - Linked list
- **Non-Linear**
  - Tree
  - Graph

Linear Data Structure	Non-Linear Data Structure
Linear order.	Hierarchical manner.
Single level	Multiple levels
Easy implementation	Complex Implementation
Single run traversal	Multi-run traversal
Time complexity increases with increase in size.	Time complexity often remains same with increase in size.

Abstract Data Type (ADT)

- To process data, we need to define the data type and the operation to be performed on the data.
- The definition of the data type and the definition of the operation to be applied to the data is part of the idea behind an Abstract Data Type (ADT).
- ADT means to **hide how the operation is performed on the data.**
- In other words, the user of an ADT needs only to know that a set of operations are available for the data type, but does not need to know how they are applied.

Another Resemblance!

- An ADT is like a ATM machine. No matter what is the design of the machine, as long as it is working fine, nobody cares what the design really is. Similarly, until the methods change the internal details of an ADT remains a mystery to the end user.

ADT

- ADT is the basis of programs
- Are of two types Built in and User defined
- Operate as a black box to the client application programmer.

Built In ADTs in C

- Array
- Structure

User defined ADTs

- Single Linked List
- Double Linked List
- Circular Linked List
- Binary Tree
- Binary Search Tree
- Tree
- Stack
- Queue

Methods of ADT

- Active Methods
- Passive Methods

Active Methods

- The methods that change the contents of the ADT.
- The methods to add/modify/delete the content of the ADT.

Passive Methods

- Methods that don’t change the content of the ADT.
- Normally returns some information about the ADT by just peeking at the data without changing it.

Active Methods Examples

- ADT Linked List
  - Active Methods
    - `push_back()`
    - `push_front()`
    - `insert() ….etc`

- ADT Trie
  - Active Methods
    - `AddAWord()`
    - `DeleteAWord()`

Passive Method Example

- ADT Stack
  - Passive Methods
    - isEmpty()
    - isFull()
- ADT Queue
  - Passive Methods
    - isEmpty()
    - isFull()

Introduction to Algorithms

- The theoretical study of design and analysis of computer algorithms.
- Basic goals for an algorithm:

➢ always correct

➢ always terminates

Design and Analysis of Algorithms

- ***Analysis:*** predict the cost of an algorithm in terms of resources and performance

- ***Design:*** design algorithms which minimize the cost

Analysis of Algorithms

- An algorithm is a **finite set of precise instructions** for performing a computation or for solving a problem.
- What is the goal of Analysis of Algorithm?

➢ to compare algorithms mainly in terms of running time and other factors such as memory requirement, programmer's effort, etc.

- What is Running Time Analysis?

➢ determining how running time increases as the size of problem increases.

Types of Analysis

- **Worst Case**

➔ Provides an upper bound on running time.

➔ An absolute guarantee that the algorithm would not run longer,no matter what the inputs are.

- **Best Case**

➔ Provides a lower bound on running time.

➔ Input is the one for which the algorithm runs the fastest.

- **Average Case**

➔ Prives the prediction of the running time

➔ Assumes that the input is random

Data Structures and Algorithms

Mathematics of Computing

Indian Institute of Information Technology Sri City, India

March 28, 2021

Today's Contents

- Exponentials
- Logarithms
- Arithmetic Sums
- Geometric Sums
- Permutations and Binomials
- Proof by induction
- Proof by counterexample
- Proof by contradiction
- Recap to recursion

Exponentials

$$X^A X^B = X^{A+B} \tag{1}$$

$$\frac{X^A}{X^B} = X^{A-B} \tag{2}$$

$$\left(X^A\right)^B = X^{AB} \tag{3}$$

$$X^N + X^N = 2X^N \neq X^2 \tag{4}$$

$$2^N + 2^N = 2^{N+1} \tag{5}$$

Logarithms

- In computer Science, all logarithms are to the base 2 unless specified otherwise.

- $X^A = B$ if and only if $log_X B = A$

Theorem 1.1:

$$\log_A B = \frac{\log_C B}{\log_C A}; \quad A, B, C > 0, A \neq 1 \tag{6}$$

Proof Let $X = log_C B$, $Y = log_C A$, and $Z = log_A B$.
Then, by the definition of logarithms, $C^X = B, C^Y = A$, and $A^Z = B$.
Combining these three equalities yields B = $C^X = (C^Y)^Z$.
Therefore, X = YZ, which implies Z = X/Y, proving the theorem.

Logarithms

Theorem 1.2 $logAB = logA + logB; A, B > 0$

Proof

Let X = log A, Y = log B, and Z = log AB. Then, assuming the default base of 2, $2^X = A, 2^Y = B, and 2^Z = AB$.

Combining the last three equalities yields $2^X 2^Y = AB = 2^Z$.

Therefore, X + Y = Z, which proves the theorem. Some other useful formulas, which can all be derived in a similar manner, follow:

- $logA/B$ =$logA$ - $logB$
- $log(A^B) = BlogA$
- log$X < X forall X > 0$
- log 1 = 0, log 2 = 1, log 1,024 = 10, log 1,048,576 = 20

Logarithms

- If we start with a value N, divide it by 2, then that result we divide it by 2, and so on, until reaching 1 or less.
- **Question**: How many times did we divide before reaching 1 or less?

Logarithms

- If we start with a value N, divide it by 2, then that result we divide it by 2, and so on, until reaching 1 or less.
- **Question**: How many times did we divide before reaching 1 or less?
- **Answer**: consider, Start with 1, then double it, then double it again, until reaching N. If we doubled k times, then $N = 2k$, or, $k = \lg(N)$

Arithmetic Sums

$$\sum_{k=1}^{n} k = 1 + 2 + 3 + \cdots + (n-1) + n \tag{7}$$

$$= n + (n-1) + (n-2) + \cdots + 2 + 1$$

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2} \tag{8}$$

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6} \tag{9}$$

Geometric Sums

$$\sum_{k=0}^{n} a^k \tag{10}$$

Let's use S to denote the sum,

$$\begin{aligned} S &= 1 + a + a^2 + a^3 + \cdots + a^{n-1} + a^n \\ aS &= a + a^2 + a^3 + \cdots + a^{n-1} + a^n + a^{n+1} \\ &= S + a^{n+1} - 1 \end{aligned} \tag{11}$$

From $aS = S + a^{n+1} - 1$, we solve for S, obtaining:

$$\sum_{k=1}^{n} a^k = \frac{a^{(n+1)} - 1}{a - 1} \tag{12}$$

Often enough, you see it written with both signs reversed (makes more sense when $|a| < 1$)

Permutations and Binomials

Binomial coefficients, the so-called "n choose k" and denoted

$$\left(\begin{array}{c} n \\ k \end{array} \right) \tag{13}$$

are closely related to factorials and permutations; the value of that expression is given by:

$$\left(\begin{array}{c} n \\ k \end{array} \right) = \frac{n!}{k!(n-k)!} \tag{14}$$

An important application is the Binomial Expansion — to obtain the n^{th} power of (a+b):

$$(a+b)^n = \sum_{k=0}^{n} \left(\begin{array}{c} n \\ k \end{array} \right) a^k b^{n-k} \tag{15}$$

Proof by Induction

- A proof by induction has two standard parts. The first step is proving a base case, that is, establishing that a theorem is true for some small (usually degenerate) value(s); this step is almost always trivial.
- Next, an **inductive hypothesis** is assumed. Generally this means that the theorem is assumed to be true for all cases up to some limit k. Using this assumption, the theorem is then shown to be true for the next value, which is typically k + 1. This proves the theorem (as long as k is finite).

Proof by Induction: Example

Theorem If N $\geq$ 1 then,

$$\sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6} \tag{16}$$

Proof

The proof is by induction. For the basis, it is readily seen that the theorem is true when N = 1. For the inductive hypothesis, assume that the theorem is true for 1 $\leq$ k $\leq$ N. We will establish that, under this assumption, the theorem is true for N + 1. We have,

$$\sum_{i=1}^{N+1} i^2 = \sum_{i=1}^{N} i^2 + (N+1)^2 \tag{17}$$

Proof by Induction: Example

Applying the inductive hypothesis, we obtain

$$\sum_{i=1}^{N+1} i^2 = \frac{N(N+1)(2N+1)}{6} + (N+1)^2 \tag{18}$$

$$\begin{aligned} &= (N+1)\left[\tfrac{N(2N+1)}{6} + (N+1)\right] \\ &= (N+1)\tfrac{2N^2+7N+6}{6} \\ &= \tfrac{(N+1)(N+2)(2N+3)}{6} \end{aligned} \tag{19}$$

Thus,

$$\sum_{i=1}^{N+1} i^2 = \frac{(N+1)[(N+1)+1][2(N+1)+1]}{6} \tag{20}$$

proving the theorem.

Proof by Counter example

- This proof structure allows us to prove that a property is not true by providing an example where it does not hold.
- For example, to prove that "not all triangles are obtuse", we give the following counter example: the equilateral triangle having all angles equal to sixty. In this case, there are infinitely many counterexample. However, it only takes one to show it.

Proof by Contradiction

- Proof by contradiction proceeds by assuming that the theorem is false and showing that this assumption implies that some known property is false, and hence the original assumption was erroneous.
- A classic example is the proof that there is an infinite number of primes.
- To prove this, we assume that the theorem is false, so that there is some largest prime P_k.
- Let $P_1, P_2, ..., P_k$ be all the primes in order and consider,

$$N = P_1 P_2 .... P_k + 1 \tag{21}$$

Proof by Contradiction

- Clearly, N is larger than P_k, so, by assumption, N is not prime.
- However, none of $P_1, P_2, ..., P_k$ divides N exactly, because there will always be a remainder of 1.
- This is a contradiction, because every number is either prime or a product of primes.
- Hence, the original assumption, that P_k is the largest prime, is false, which implies that the theorem is true.

A brief recap to Recursion

- A function that is defined in terms of itself is called recursive.

- When writing recursive routines, it is crucial to keep in mind the four basic rules of recursion:
  1. Base cases. You must always have some base cases, which can be solved without recursion.
  2. Making progress. For the cases that are to be solved recursively, the recursive call must always be to a case that makes progress toward a base case.
  3. Design rule. Assume that all the recursive calls work.
  4. Compound interest rule. Never duplicate work by solving the same instance of a problem in separate recursive calls.

A brief recap to Recursion: example

```
1  void printOut( int n )  // Print nonnegative n
2  {
3      if( n >= 10 )
4          printOut( n / 10 );
5      printDigit( n % 10 );
6  }
```

Figure 1. Recursive routine to print an integer.

Book

- Mark Allen Weiss, "Data Structures and Algorithm Analysis in C++", Third Edition, Addison Wesley, 2006.
- Cormen, Leiserson, Rivest, Stein, "Introduction to Algorithms", Third Edition, MIT Press, 2009.

Data Structures and Algorithms

Algorithm Analysis

Indian Institute of Information Technology Sri City, India

March 29, 2021

Today's Contents

- Algorithm Analysis
- Asymptotic Notations
- Bounds of Algorithmic Analysis

Content Acknowledgments

- "Algorithms", Dr. Rajendra Prasath, Autumn 2018, IIIT Sri City
- "Introduction to Algorithms",Cormen, Leiserson, Rivest, Stein, MIT Press, 2009.

Two Steps to Remember before Algorithm Analysis

- Data Structures
- The choice of Data Structures
  - Built-in Data Structures (Primitive)
  - User Defined Data Structures (Abstract)
- Computational Efficiency
  - Time Complexity
  - Space Complexity
- Problem / Solution Specific Constraints
- Best Practices / Efficient Approaches

Algorithm Analysis

- Scale the algorithm for sufficiently large n ?
- How fast the algorithm could behave when we scale the input size n sufficiently large enough??
- Performance degradation
- Analysis complex properties
- How to prepare the solution to work for the given problem with sufficiently large input size ?

Algorithm Analysis: Example

- Consider Swapping of two integers
  - Input: x = 5, y = 7
  - Output: x = 7, y = 5
- A simple Solution: Use of temporary variable to hold the intermediate value.
- Any Other Solution?
  - Bitwise
  - Add / Subtract with no intermediate value.
- Explore Complexity : Time and Space needed

Time Complexity

- **Computation of TIME**
  - Consider a unique list of 10 numbers (unsorted):
    5, 8, 21, 43, 35, 17, 94, 68, 54, 81
    (Traversal must be left to right)
- **Task:**
  - Find any one (??) number in the list
- **Best Case**
  - The minimum (at least) time taken for solving the problem
  - Finding the number: 5 in the list
- **Average Case**
  - Average time taken to solve the problem
- **Worst Case**
  - The maximum (at most) time taken for any input size
  - Finding the number 81 in the list

Space Complexity

A measure of the amount of working storage an algorithm needs.

- This means how much memory, in the worst case, is needed at any point by the underlying algorithm?
- We always focus on how the space needs grow, in big-Oh terms, as the size N of the input problem grows.
- Explore different aspects of time and space complexities

Algorithm Analysis: Algorithm Pattern Nomenclature

- Name (Descriptive Name of the algorithm)
- Context (illustrating an essential part of the Algo)
- Facts (Properties that could cause anyone to choose the algorithm specifically)
- Consequences (Advantages and Disadvantages)
- Analysis of the algorithm (understanding the behavior of the algorithm
  - Why does an algorithm behave like this!!
  - Can we come up with lemmas and proofs to explain the behavior of the algorithm?
- Alternatives (find and compare other competitive variations of solutions to the same problem)

Algorithm Analysis: Example 2

Task: Find the element in a given list L

- The given list, say L has n elements
- We have to find the element in the list.
- Issues?
- n elements can be unique (no repetitions)
- There can be more than one instance of the element.
- The element can be seen in the worst case scenario

Algorithm Analysis: Example 2 (contd)

Task: Find the element in a given list L

- Solution: look at every number in the list by comparing with key

High-level description (Linear Scan):

- If there are no numbers in the list then there is no searching element (key).
- Start with the first number in the list by comparing with the key
- if found return else
- For each remaining number in the list: continue searching by comparison
- When there are no numbers left in the list to scan through, consider that element is not found in the list

Algorithm Analysis: Example 2 (contd)

Task: Find the element in a given list L

- Let us consider the list of n elements

1	4	9	15	7	12	13	6

Solution:

```
Algorithm findNumber(list, num)
  begin
    for each element t in the list, do
      if (list[i] = t) then
          return true
    return false;
  end
```

Search - Complexity

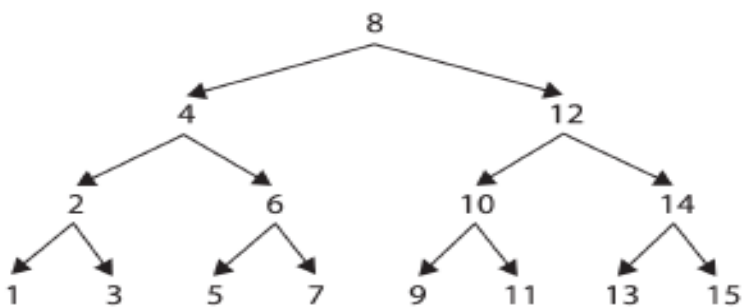
- **Best :** O(1) – Constant time
- **Average :** O(n) – Linear time
- **Worst :** O(n) – Linear time
- Which Data Structure is used . . . ??
- Choice of the data structure makes this complexity different

How to make Search easier?

- Given an array of n elements
- List

1	4	9	15	7	12	13	6

- Linked list
- Trees and its variants



Computational Complexity

- Running Times of the algorithms may vary based on n.
  - Fastest Algorithms – Less time
  - Slowest Algorithms – More time

- Order of Growth (Magnitude)
  - **How fast the running time grows with the input size, say n?**

- How to perform accurate analysis of algorithms in terms of its computational complexities:
  - Worst, average and Best cases

Running Time Estimation

SIZE / COMPLEXITY	20	50	100	200	500	1000
$1000n$	.02 sec	.05 sec	.1 sec	.2 sec	.5 sec	1 sec
$1000n \lg n$	.09 sec	.3 sec	.6 sec	1.5 sec	4.5 sec	10 sec
$100n^2$	.04 sec	.25 sec	1 sec	4 sec	25 sec	2 min
$10n^3$	.02 sec	1 sec	10 sec	1 min	21 min	2.7 hr
$n^{\lg n}$	.4 sec	1.1 hr	220 DAYS	125 CENT	5×10^8 CENT	
$2^{n/3}$	.0001 sec	.1 sec	2.7 hr	3×10^4 CENT		
2^n	1 sec	35 YR	3×10^4 CENT			
3^n	58 min	2×10^9 CENT				

- One step takes one Microsecond. lg n denotes $\log_2 n$

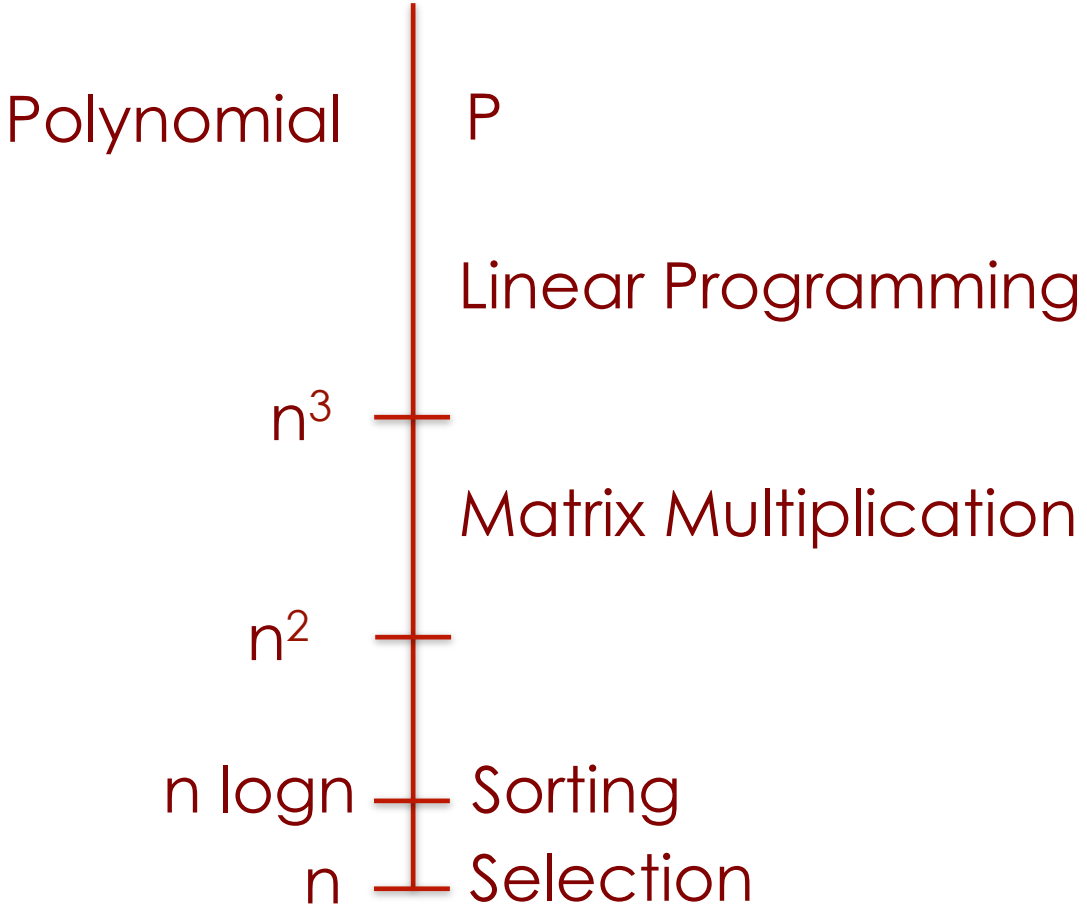
Computational Complexity

TIME / COMPLEXITY	1sec	10^2 sec (1.7 min)	10^4 sec (2.7 hr)	10^6 sec (12 DAYS)	10^8 sec (3 YEARS)	10^{10} sec (3 CENT.)
$1000n$	10^3	10^5	10^7	10^9	10^{11}	10^{13}
$1000n \lg n$	1.4×10^2	7.7×10^3	5.2×10^5	3.9×10^7	3.1×10^9	2.6×10^{11}
$100n^2$	10^2	10^3	10^4	10^5	10^6	10^7
$10n^3$	46	2.1×10^2	10^3	4.6×10^3	2.1×10^4	10^5
$n \lg n$	22	36	54	79	112	156
$2^{n/3}$	59	79	99	119	139	159
2^n	19	26	33	39	46	53
3^n	12	16	20	25	29	33

- As the problem size increases, polynomial time algorithm become unusable gradually.
- A factor of ten increase in machine speed corresponds to a factor of ten increase in time.

The Spectrum of Computational Complexity

TRACKTABLE



The Spectrum of Computational Complexity

INTRACKTABLE

Undecidable (with no algorithms)	Hilbert's Tenth Problem
Superexponential	Presburger Arithmetic
Exponential	Circularity of Attribute Grammers
	NP-Complete Problems

Efficiency of Algorithms

- Assume that we have a processor that executes a million high-level instructions per second and we have algorithms with polynomial running-time

How to define Efficiency?

- An algorithm is efficient if, when implemented, it runs quickly on real input instances.
- An algorithm is efficient if it achieves qualitatively better worst-case performance, at an analytical level, than brute-force search
- An algorithm is efficient if it has a polynomial running time
  - **Growing Polynomials:**
    $1 < n < n \log_2 n < n^2 < n^3 < 1.5n < 2^n < n!$

Asymptotic Upper Bounds

"Big-O" Notation

(introduced in P. Bachmann's 1892 book Analytische Zahlentheorie_

- Let $f(n)$ be a function
  - Say the worst case running time of a certain algorithm on an input of size n
- and $g(n)$ be another function

- We say that $f(n)$ is O $(g(n))$ for sufficiently large n, the function $f(n)$ is bounded by a constant multiple of $g(n)$

- More Precisely, $f(n)$ is O $(g(n))$ if **there exist** constants c > 0 and $n_0 \geq 0$ so that for all $n \geq n_0$, we have $f(n) \leq c \cdot g(n)$

- The constant c can not depend on n

Asymptotic Lower Bounds

"Omega (Ω)" Notation

- Let $f(n)$ be a function and $g(n)$ be another function

- We say that $f(n)$ is $\Omega(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ so that for all $n \geq n_0$, we have $f(n) \geq c \cdot g(n)$

- Note that the constant c must be fixed, independent of n.

Asymptotic Tight Bounds

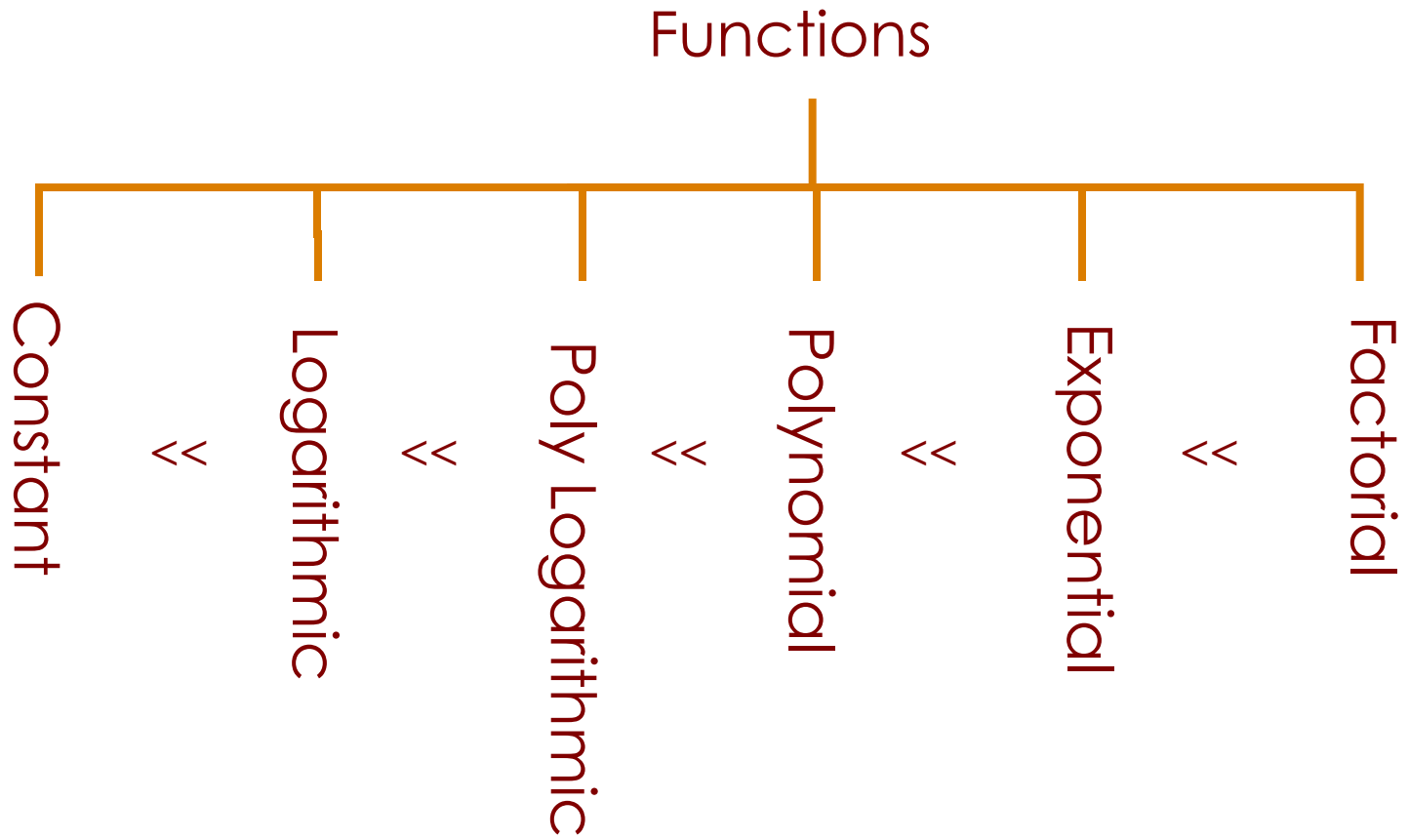
"Theta (Θ)" Notation

- Let $f(n)$ be a function and let $g(n)$ be another function.
- We say that $f(n)$ is $\Theta(g(n))$ if $f(n)$ if both $O(g(n))$ and $\Omega(g(n))$.
- Asymptotically the tight bounds characterize the worst case performance of an algorithm precisely upto constant factors.
- It closes the gap between an upper bound and a lower bound.

Asymptotic Growth Rates

- A way of comparing functions that ignores constant factors and small input sizes
- $O(g(n))$: class of functions $f(n)$ that grow no faster than $g(n)$
- $\Theta(g(n))$: class of functions $f(n)$ that grow at same rate as $g(n)$
- $\Omega(g(n))$: class of functions $f(n)$ that grow at least as fast as $g(n)$

Computational Complexity



$5 \ll 5 \log n \ll (\log n)^5 \ll n^5 \ll 2^{5n} \ll n!$

Identify the Constants

- Yes • 5
- Yes • 1,000,000,000,000
- Yes • 0.000000000001
- Yes • -5
- Yes • 0
- No • 8 + sin(n)

The running time of the algorithm is a "Constant" if it does not depend significantly on the input size

Quadratic Functions?

- n^2
- $0.001\ n^2$
- $1000\ n^2$
- $5n^2 + 3n + 2\log n$

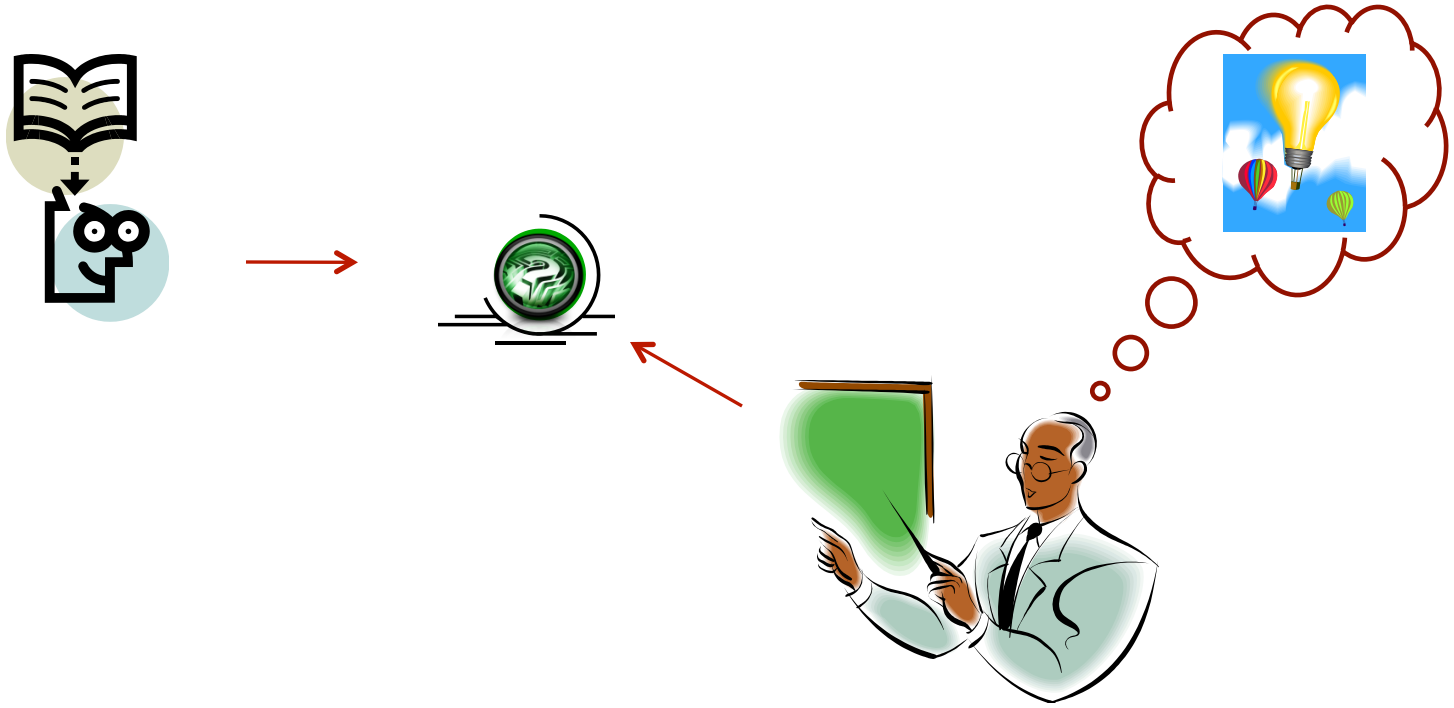
Ignore low-order terms
Ignore multiplicative constants
Ignore "small" values of n

Write $\Theta(n^2)$

Time Efficiency of Non-recursive Algorithms

- Decide on parameter n indicating input size
- Identify algorithm's basic operation
- Determine worst, average, and best cases for input of size n
- Set up a sum for the number of times the basic operation is executed
- Simplify the sum using standard formulas and rules

Thanks …



… Questions ???

Data Structures and Algorithms

Algorithm Analysis

Indian Institute of Information Technology Sri City, India

April 3, 2021

Today's Contents

- Algorithmic thinking - Problem Solving Approaches
- Algorithm Design
- Models of Computation
- Computational Complexities

Content Acknowledgments

- "Algorithms", Dr. Rajendra Prasath, Autumn 2018, IIIT Sri City
- "Introduction to Algorithms",Cormen, Leiserson, Rivest, Stein, MIT Press, 2009.

Algorithmic thinking - Problem Solving Approaches

- **Computation of TIME Recap**
  - Consider a unique list of 10 numbers (unsorted):
    5, 8, 21, 43, 35, 17, 94, 68, 54, 81
    (Traversal must be left to right)
- **Task:**
  - Find any one (??) number in the list
- **Best Case**
  - The minimum (at least) time taken for solving the problem
  - Finding the number: 5 in the list
- **Average Case**
  - Average time taken to solve the problem
- **Worst Case**
  - The maximum (at most) time taken for any input size
  - Finding the number 81 in the list

Algorithmic thinking - Problem Solving Approaches

- How to prepare the solution to work for the given problem with sufficiently large input size ?
- Whether the Solutions (Algorithms) you proposed could work for sufficiently large input size ?
- What are the issues in scaling up the solution?
- The given problem has to be solved with the resources in hand (!!)

Algorithms

- Definition: An algorithm can be defined as a collection of unambiguous (precise) executable instructions, whose step by step execution leads to a predefined goal, within a finite number of steps.
- Algorithmic Thinking
  - Enables a step by step method of solving a problem
  - Helps to design efficient algorithms
  - A complex intellectual process of thinking, combining facts, skills, and so on
  - Effective algorithms: Finite time and space

Good or Efficient Algorithms?

- Easy to understand
- Easy to Implement
- Efficient Algorithms
  - Provides an added insight to the problem
  - Solutions must be Simple and Elegant
- Needs a model of computation
- Develop a denotational definition of complexity
- Of course Too much details will **Obscure the most beautiful** Algorithms

Algorithm Design

A method or even a Mathematical process for problem solving.
Design an algorithm with efficient run time.
Steps:

- Problem Definition
- Develop a Mathematical model of computation
- Specification of the algorithm
- Design the algorithm
- Check the correctness of the algorithm
- Analysis of the algorithm
- Choice of proper data Structures and implementation
- Perform Program Testing
- Documentation of the above procedure

Good Algorithm- simplicity and its elegance.

Classification of Algorithms

- By implementation
- By Design Paradigm
- By Domain Specific study
- By Computational Complexity

Models of Computation : Turing Machine

Turing Machine – The First computing machine

- consists of a finite state control
- a two-way infinite memory tape divided into squares, each of which can hold one symbol and a read / write head
- In one step, the machine can
  - Read the contents of one tape square
  - Write a new symbol in that square
  - Move the head one square left or right and
  - Change the state of the control
- Such Turing Machine are useful in high-level theoretical studies of computational complexity
- But not realistic to follow the accurate analysis of algorithms

Models of Computation : Random-access machine (RAM) model

In the RAM model, instructions are executed one after another, with no concurrent operations.

- RAM model contains instructions commonly found in real computers: arithmetic (such as add, subtract, multiply, divide, remainder, floor, ceiling), data movement (load, store, copy), and control (conditional and unconditional branch, subroutine call and return).
- Each such instruction takes a constant amount of time.
- The running time of an algorithm on a particular input is the number of primitive operations or "steps" executed.

Computational Complexities

- Time Complexity
- Space Complexity

Computational Complexities: Analsysis Example

Linear Search:

```
 1  int Linear_Search(int a[],int n, int key)
 2  {
 3      int i;
 4      for (i=0;i<n;i++)
 5      {
 6          if(a[i]==key)
 7          {
 8              return i;
 9          }
10      return 0;
11  }
```

Computational Complexities

The analysis of this fragment is simple.

- The declarations count for no time. (Line 3)
- Line 6 count for one unit.
- Line 4 has the hidden costs of initializing i, testing i N, and incrementing i. The total cost of all these is 1 to initialize, N+1 for all the tests, and N for all the increments, which is 2N + 2.
- We ignore the costs of calling the function and returning, for a total of 2N + 3.
- We have 2N+3, but ignoring the coefficients and constants we get N.
- So, we say that this function has complexity of O(N).

Algorithm Computational Complexities Analysis: General rules

General rules:

- **Rule 1— FOR loops**
  The running time of a for loop is at most the running time of the statements inside the for loop (including tests) times the number of iterations.
- **Rule 2—Nested loops**
  Analyze these inside out. The total running time of a statement inside a group of nested loops is the running time of the statement multiplied by the product of the sizes of all the loops. As an example, the following program fragment is O(N2):
  for(i = 0; i ¡ n; ++i)
  for(j = 0; j ¡ n; ++j)
  ++k;

Algorithm Computational Complexities Analysis: General rules

General rules:

- **Rule 3—Consecutive Statements**
  These just add (which means that the maximum is the one that counts).
- As an example, the following program fragment, which has O(N) work followed by O(N^2) work, is also O(N^2):

```
for( i = 0; i ¡ n; ++i )
    a[ i ] = 0;
for( i = 0; i ¡ n; ++i )
  for( j = 0; j ¡ n; ++j )
      a[ i ] += a[ j ] + i + j;
```

Algorithm Computational Complexities Analysis: General rules

General rules:

- **Rule 4—If/Else** For the fragment

  ```
  if( condition )
    S1
  else
    S2
  ```

  the running time of an if/else statement is never more than the running time of the test plus the larger of the running times of S1 and S2.

Algorithm Analysis: Example 2 (contd)

Task: Find the element in a given list L

- Solution: look at every number in the list by comparing with key

High-level description (Linear Scan): A method to characterize the execution time of an algorithm:

- Adding two square matrices is O(n2)
- Searching in a dictionary is O(log n)
- Sorting a vector is O(n log n)
- Solving Towers of Hanoi is O(2n)
- Multiplying two square matrices is O(n3)
- Count the dominating terms . . . discard constants

Estimating Runtime – 1

- What is the runtime of g(n)?

```
void g(int n) {
        for (int i = 0; I < n; ++i)
                f( );
}
```

Runtime g(n) ≈ n • runtime (f(n))

Estimating Runtime – 1

- What is the runtime of g(n)?

```
void g(int n) {
        for (int i = 0; I < n; ++i)
                f( );
}
```

Runtime g(n) ≈ n • runtime (f(n))

Estimating Runtime – 2

- What is the runtime of g(n)?

```
void g(int n) {
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            f( );
}
```

Runtime g(n) $\approx n^2 \bullet$ runtime (f(n))

Estimating Runtime – 3

- What is the runtime of g(n)?

```
void g(int n) {
    for (int i= 0; i < n; ++i)
        for(int j = 0; j <= i; ++j)
            f( );
}
```

Runtime g(n) $\approx (1 + 2 + 3 + \bullet\bullet\bullet + n) \bullet$ runtime (f(n))

$\approx (n\ (n+1)\ /\ 2) \bullet$ runtime (f(n))

Complexity: $O(n^2)$

Estimating Runtime – 4

$T(n)$	Complexity
$5n^3 + 200n^2 + 15$	$O(n^3)$
$3n^2 + 2^{300}$	$O(n^2)$
$5\log_2 n + 15\ln n$	$O(\log n)$
$2\log n^3$	$O(\log n)$
$4n + \log n$	$O(n)$
2^{64}	$O(1)$
$\log n^{10} + 2\sqrt{n}$	$O(\sqrt{n})$
$2^n + n^{1000}$	$O(2^n)$

Book

- Mark Allen Weiss, "Data Structures and Algorithm Analysis in C++", Third Edition, Addison Wesley, 2006.
- Cormen, Leiserson, Rivest, Stein, "Introduction to Algorithms", Third Edition, MIT Press, 2009.

Data Structures and Algorithms

Recursion and Recurrence Relations

Indian Institute of Information Technology Sri City, India

April 3, 2021

Today's Contents

- Recursion
- Algorithm Design
- Models of Computation
- Computational Complexities
- Time and Space Complexities

Content Acknowledgments

- "Algorithms", Dr. Rajendra Prasath, Autumn 2018, IIIT Sri City
- "Introduction to Algorithms",Cormen, Leiserson, Rivest, Stein, MIT Press, 2009.

Recursions

- **Definition:**
  - A program is recursive when it contains a call to itself.
  - Recursion can substitute iteration in program design:
  - Generally, recursive solutions are simpler than iterative solutions.
  - Generally, recursive solutions are slightly less efficient than the iterative ones unless the recursive calls are optimized
  - There are natural recursive solutions that can be extremely inefficient . . . Be aware of such recursions !

Algorithmic thinking - Problem Solving Approaches

- How to we define a recursive approach?
  - In terms of the input size n
  - The rate of change of n
- Two Steps:
  - Basic Steps
  - Recursive Steps

Examples

- gcd(a,b)

```
int gcd(int a, int b) {
    if ( b == 0 ) return a;
    return gcd(b, a%b);
}
```

- Print N numbers

```
// Infinite Recursion
int printNum(int n) {
    printf (" %d ", n);
    return printNum(n+1);
}
```

Recurrence Relations

- How to get a closed form of a recurrence relation
  - In terms of the input size n
  - The rate of change of n
- Compute Factorial of n:

```
/* Pre: n >= 0; returns n! */

int factorial(int n) { // iterative solution
    int f = 1, i= 0;

    while(i < n) {
        i= i+ 1;
        f = f*i;
    }
    return f;
}
```

Factorial

- Definition:
  $n! = n \cdot (n-1) \cdot (n-2) \cdot \cdot \cdot 3 \cdot 2 \cdot 1$
- Recursive Definition:
  $n! = n \cdot (n–1)!, \quad if \quad n > 0;$
  $= 1, \quad if \quad n = 0;$

Code:

```
int factorial(int n) { //recursive soln.
    if(n == 0) return 1;
    else return n * factorial(n -1);
}
```

Factorial

- How to work out Factorial (5)?

factorial (5)
= 5 * factorial (4)
= 5 * 4 * factorial (3)
= 5 * 4 * 3* factorial (2)
= 5 * 4 * 3 * 2 * factorial (1)
= 5 * 4 * 3 * 2 * 1 * factorial (0)
= 5 * 4 * 3 * 2 * 1 * 1
= 120

Recursion: Important Points

- Each time a function is called, a new instance of the function is created
- Each time a function "returns", its instance is destroyed
- The creation of a new instance only requires the allocation of memory space for data
- The instances of a function are destroyed in reverse order to their creation, i.e. the first instance to be created will be the last to be destroyed.

Complexity Analysis - Recursion

```
void f(int n) {
    if(n > 0) {
        DoSomething(n);        // O(n)
        f(n/2);
    }
}
```

$$T(n) = n + T(n/2)$$

$$T(n) = n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots + 2 + 1$$

$$2 \cdot T(n) = 2n + n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots + 4 + 2$$

$$2 \cdot T(n) - T(n) = T(n) = 2n - 1 \quad \rightarrow \text{O(n)}$$

Complexity Analysis - Recursion

```
void f(int n) {
    if(n > 0) {
        DoSomething(n);        // O(n)
        f(n/2);    f(n/2);
    }
}
```

$$
\begin{aligned}
T(n) &= n + 2 \cdot T(n/2) \\
&= n + 2 \cdot \frac{n}{2} + 4 \cdot \frac{n}{4} + 8 \cdot \frac{n}{8} + \cdots \\
&= \underbrace{n + n + n + \cdots + n}_{\log_2 n} = n \log_2 n
\end{aligned}
$$

➔ O(n logn)

Complexity Analysis - Recursion

```
void f(int n) {
    if(n > 0) {
        DoSomething(n);        // O(n)
        f(n - 1);
    }
}
```

$$T(n) = n + T(n-1)$$

$$T(n) = n + (n-1) + (n-2) + \cdots + 2 + 1$$

$$T(n) = \frac{n^2 + n}{2}$$

➔ $O(n^2)$

Complexity Analysis - Recursion

```
void f(int n) {
    if(n > 0) {
        DoSomething(); // O(1)
        f(n - 1); f(n - 1);
    }
}
```

$$
\begin{aligned}
T(n) &= 2 \cdot T(n-1) \\
&= 2 \cdot 2 \cdot T(n-2) \\
&= 2 \cdot 2 \cdot 2 \cdot T(n-3) \\
&\vdots \\
&= \underbrace{2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdots 2}_{n} = 2^n \rightarrow O(2^n)
\end{aligned}
$$

Solving Recurrance Relations

- How to find the closed form solution for T(n)
- Several Methods
  - Substitution Method
  - Recursion-tree method
  - Master Method

Substitution Method

- The Basic Idea
  - Make a guess of a possible solution
  - Use Induction to prove it.
- - Look at the following Recursive Definition:

T(n) $= 1$, if n $= 1$
$= 2$T(n/2) $+$ n, for all $n > 1$

- How to solve this recurrence relation?

Substitution Method

- Solving the following recursive relation:

$T(n) = 1$, if $n = 1$
$= 2T(n/2) + n$, for all $n > 1$

Solution:

1) $T(n) = n \log n + n$

2) Proof by Induction

Basis: if $n = 1$, $\Rightarrow 1 \log 1 = 1 \rightarrow$ **$T(n) = 1$, true for $n = 1$**

Induction Step:

$T(n) = 2T(n/2) + n = 2((n/2) * \log (n/2) + n/2) + n$

$= n \log (n/2) + n + n$

$= n (\log n - \log 2) + n + n = n \log n - n + n + n$

$T(n) = n \log n + n$ is true for for all $n > 1$

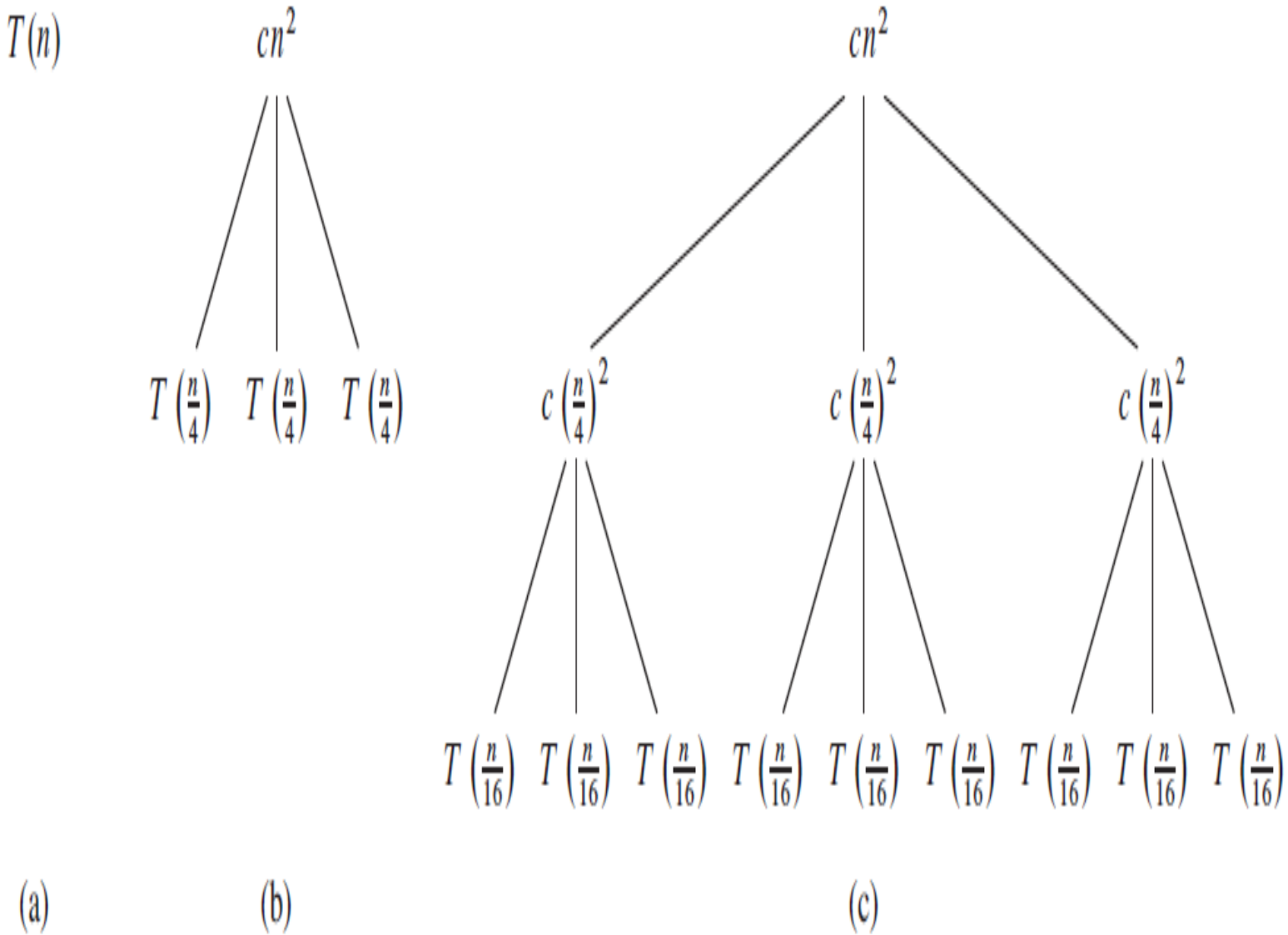
Recursion Tree

- In a recursion tree, each node represents the cost of a single subproblem somewhere in the set of recursive function invocations.
- We sum the costs within each level of the tree to obtain a set of per-level costs, and then we sum all the per-level costs to determine the total cost of all levels of the recursion.
- A recursion tree is best used to generate a good guess, which you can then verify by the substitution method.

Recursion Tree

- **Example:** Let us see how a recursion tree would provide a good guess for the recurrence $T(N) = 3T(\lfloor N/4 \rfloor) + \Theta(n^2)$
- We start by focusing on finding an upper bound for the solution.
- Because we know that floors and ceilings usually do not matter when solving recurrences, we create a recursion tree for the recurrence $T(N) = 3T(N/4) + cn^2$, having written out the implied constant coefficient $c > 0$.

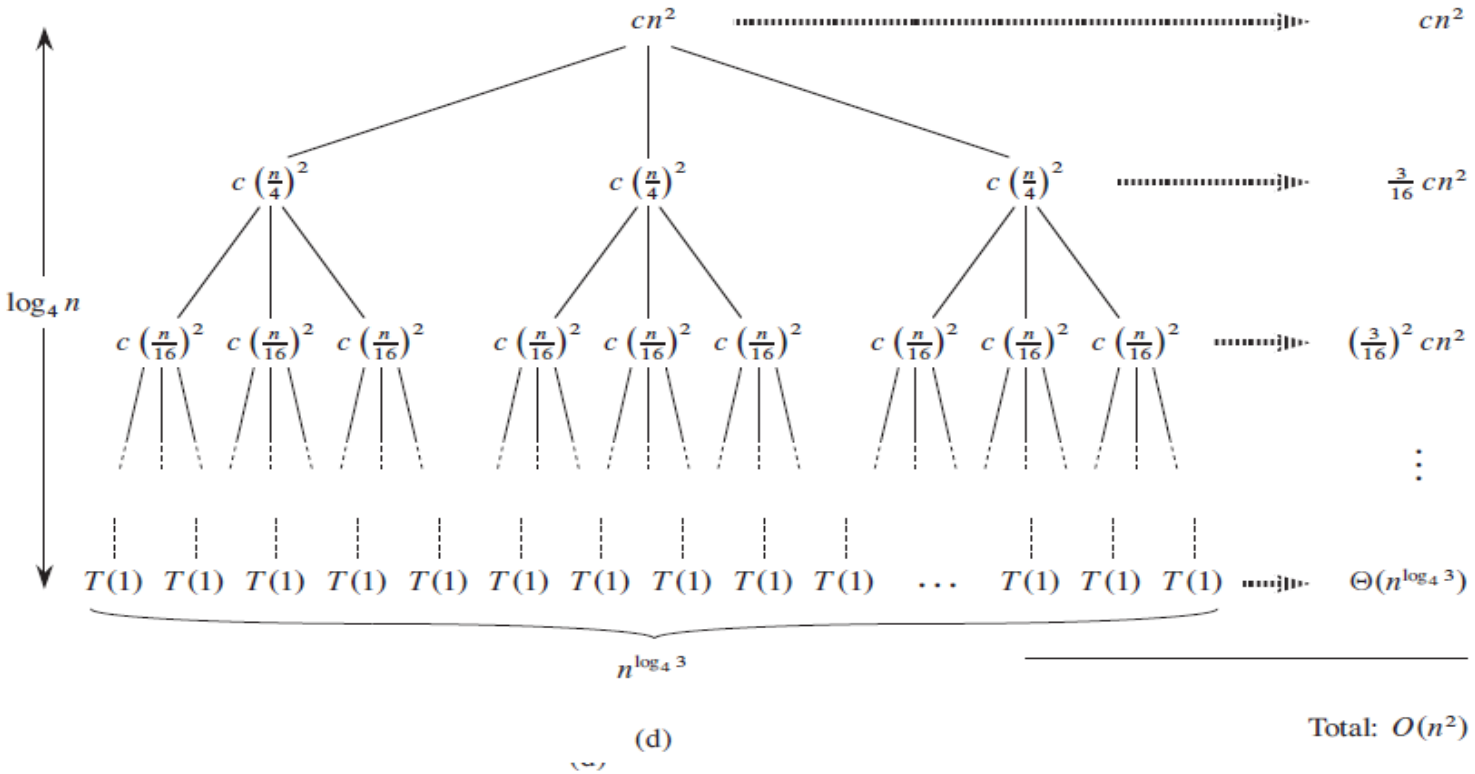
Recurrence Tree - Example



Recursion Tree

- For convenience, we assume that n is an exact power of 4 so that all subproblem sizes are integers.
- Part (a) of the figure shows T (n)
- In part (b) we expand the part (a) into an equivalent tree representing the recurrence.
- The cn^2 term at the root represents the cost at the top level of recursion, and the three subtrees of the root represent the costs incurred by the subproblems of size $n/4$.
- Part (c) shows this process carried one step further by expanding each node with cost T $(n/4)$ from part (b). The cost for each of the three children of the root is $c(n/4)^2$.

Recurrence Tree - Example



Recurrence Tree - Example

- The sub problem size for a node at depth i is $n/4^i$.
- Thus, the subproblem size hits n$=1$ when $n/4^i = 1$ or, equivalently, when i$=\log_4$ n.
- Thus, the tree has $\log_4$ n $+ 1$ levels (at depths 0, 1, 2, ..., $\log_4$ n)
- Next we determine the cost at each level of the tree.
- Each level has three times more nodes than the level above, and so the number of nodes at depth i is 3^i . Because subproblem sizes reduce by a factor of 4 for each level we go down from the root, each node at depth i, for i $= 0, 1, 2, ..., \log_4$ n-1, has a cost of c(n$/4^i)^2$
- Multiplying, we see that the total cost over all nodes at depth i, for i $= 0, 1, 2, ..., \log_4$ n-1, is $3^i c$(n$/4^i)^2 = (3/16)^i cn^2$

Recurrence Tree - Example

- The sub problem size for a node at depth i is $n/4^i$.
- The bottom level, at depth $\log_4$ n, has $3^{\log_4 n} = n^{\log_4 3}$ nodes, each contributing cost T(1), for a total cost of $n^{\log_4 3}$T(1), which is $.n^{\log_4 3}$, since we assume that T(1) is a constant.
- Now we add up the costs over all levels to determine the cost for the entire tree:

$$T(n) = cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta\left(n^{\log_4 3}\right) \tag{1}$$

.

- This is a geometric progression with ratio 5/16
- Upper Bound is:

$$\frac{16}{13}cn^2 + \Theta\left(n^{\log_4 3}\right) \tag{2}$$

- Which is

$$T(n) = O\left(n^2\right) \tag{3}$$

Master Theorm

- Consider a recurrence of the form:

  $T(n) = aT(n/b) + f(n)$ where $a \geq 1$ and $b > 1$

- Recursive Tree based approach
- The running time is influenced by:
  - cost at **leaf nodes:**

    If $f(n) = \Theta(n^c)$ where $c < \text{Log}_b a$ then $T(n) = \Theta(n^{\text{Log}_b a})$

  - cost **evenly distributed** throughout the tree:

    If $f(n) = \Theta(n^c)$ where $c = \text{Log}_b a$ then $T(n) = \Theta(n^c \text{ Log } n)$

  - cost at **root nodes**:

    If $f(n) = \Theta(n^c)$ where $c > \text{Log}_b a$ then $T(n) = \Theta(f(n))$

Master Theorem: Formal Definition

- For any recurrence relation of the form

  $T(n) = a\,T(n/b) + cn^k$, $T(1) = c$, the following relationships hold

$$\mathbf{T}(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{if } a > b^k \\ \Theta(n^k \log n) & \text{if } a = b^k \\ \Theta(n^k) & \text{if } a < b^k \end{cases}$$

- Apply this recurrence whenever appropriate without deriving the solution for the recurrence

Master Theorem: Example 1

$$\mathbf{T}(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{if } a > b^k \\ \Theta(n^k \log n) & \text{if } a = b^k \\ \Theta(n^k) & \text{if } a < b^k \end{cases}$$

- Apply the theorem to solve

  $T(n) = 3\,T(n/5) + cn^2,\ T(1) = c$

- Here a = 2, b = 5, c = 8 and k = 2

- Check whether $3 < 5^2$??

- As per case 3: The solution is

  $T(n) = \Theta(n^2)$

Master Theorem: Example 2

$$\mathbf{T}(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{if } a > b^k \\ \Theta(n^k \log n) & \text{if } a = b^k \\ \Theta(n^k) & \text{if } a < b^k \end{cases}$$

- Apply the theorem to solve

  $T(n) = 2\,T(n/2) + n,\ T(1) = 1$

- Here a = 2, b = 2, c = 1 and k = 1

- Can we find $2 = 2^1$??

- As per case 2: The solution is

  $T(n) = \Theta(n \log n)$

Book

- Mark Allen Weiss, "Data Structures and Algorithm Analysis in C++", Third Edition, Addison Wesley, 2006.
- Cormen, Leiserson, Rivest, Stein, "Introduction to Algorithms", Third Edition, MIT Press, 2009.

Data structures and algorithms

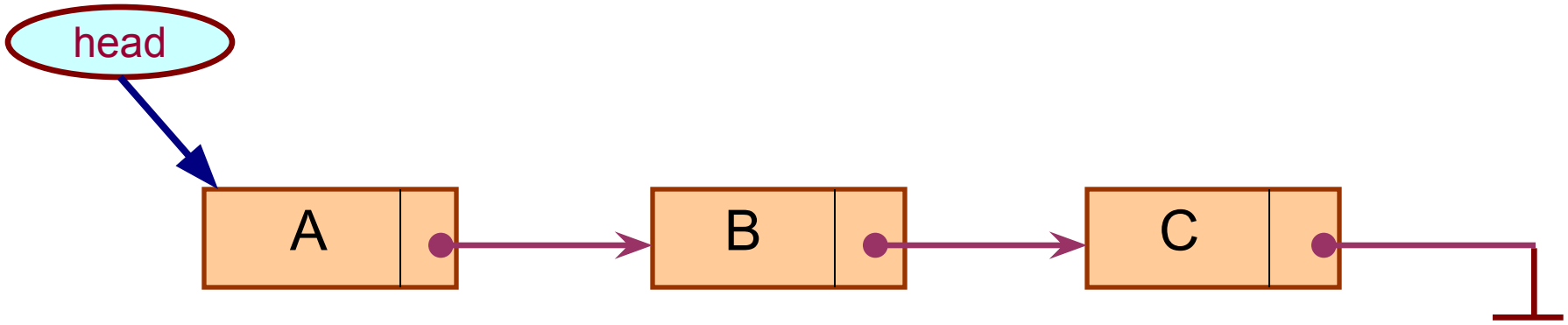
Indian Institute of Information Technology Sri City

List is an Abstract Data Type

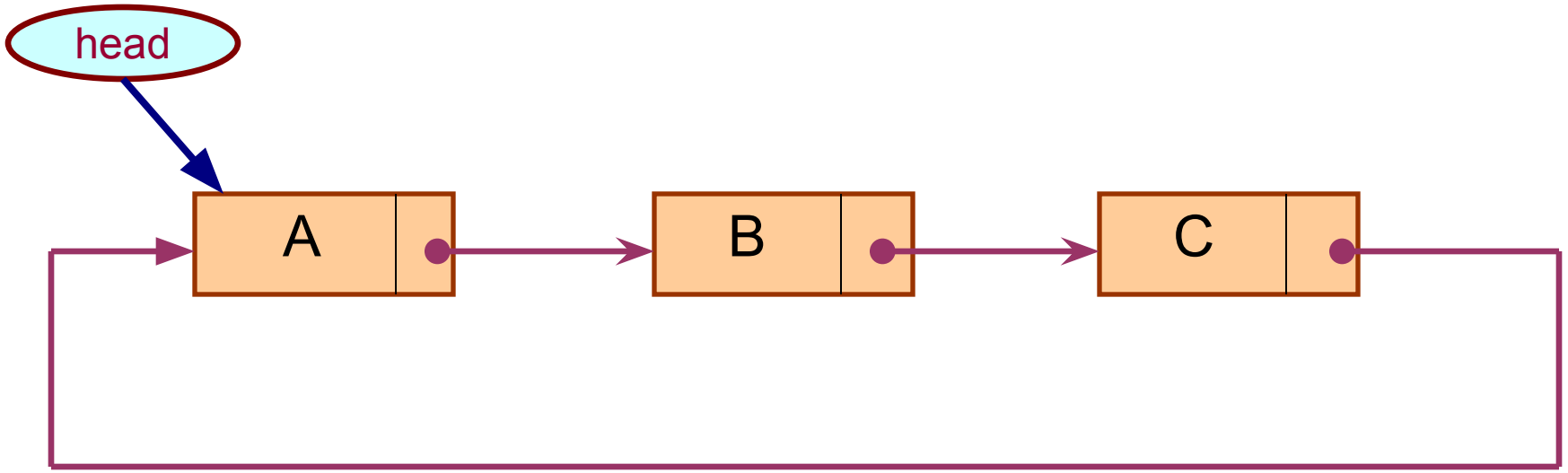
- What is an abstract data type?
  - It is a data type defined by the user.
  - Typically more complex than simple data types like *int*, *float*, etc.
- Why abstract?
  - Because details of the implementation are hidden.
  - When you do some operation on the list, say insert an element, you just call a function.
  - Details of how the list is implemented or how the insert function is written is no longer required.

Types of Lists

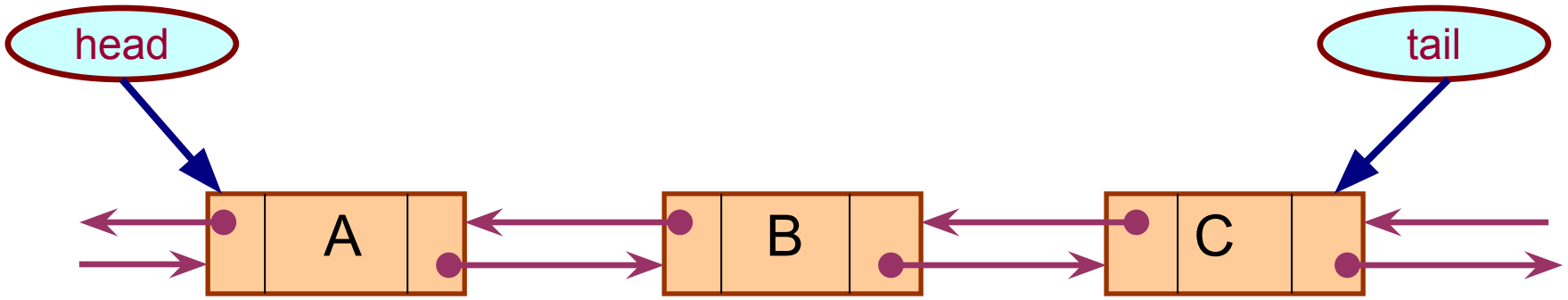
- Depending on the way in which the links are used to maintain adjacency, several different types of linked lists are possible.
  - Linear singly-linked list (or simply linear list)
    - One we have discussed so far.



- Circular linked list
  - The pointer from the last element in the list points back to the first element.



- Doubly linked list
  - Pointers exist between adjacent nodes in both directions.
  - The list can be traversed either forward or backward.
  - Usually two pointers are maintained to keep track of the list, *head* and *tail*.



Basic Operations on a Linear List using Dynamic Memory Allocation

- Creating a list
- Traversing the list
- Inserting an item in the list
- Deleting an item from the list
- Concatenating two lists into one

Example: Working with linked list

- Consider the structure of a node as follows:

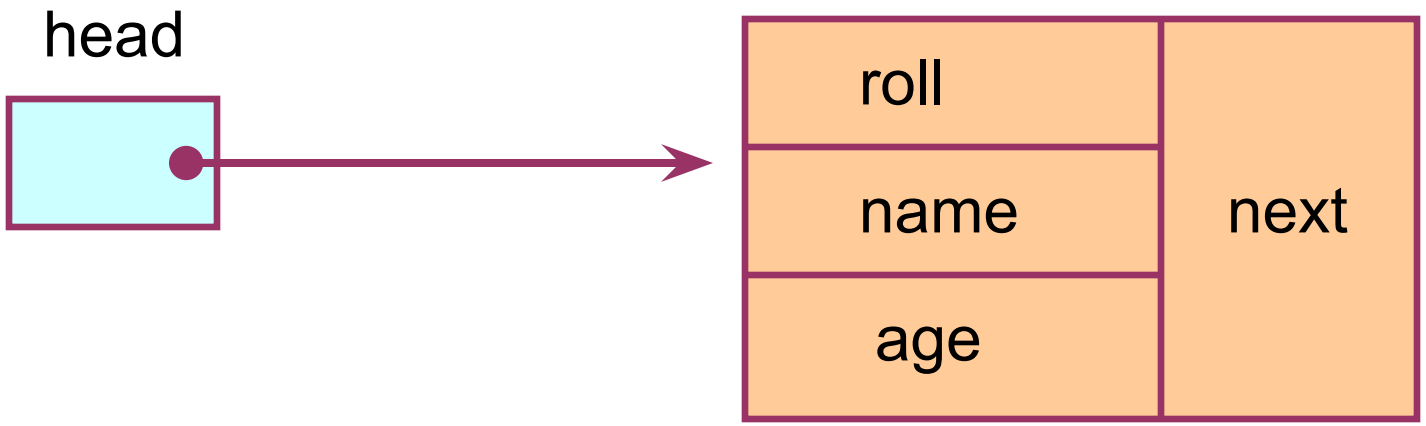
```
struct stud {
              int   roll;
              char  name[25];
              int   age;
              struct stud *next;
            };

 /* A user-defined data type called "node" */

typedef struct stud node;
node *head;
```

Creating a List

How to begin?

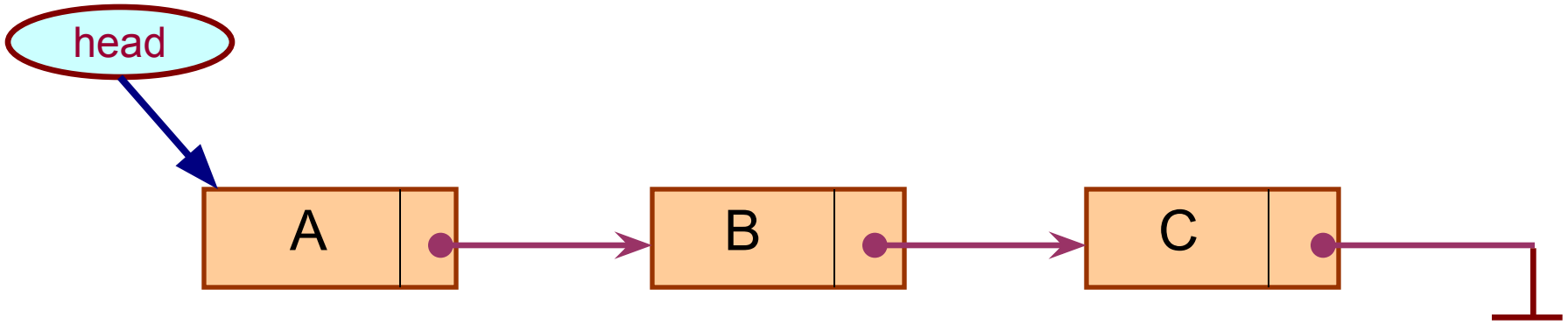
- To start with, we have to create a node (the first node), and make `head` point to it.

```
head = (node *) malloc(sizeof(node));
```



Creating a list cont'd.

- If there are n number of nodes in the initial linked list:
  - Allocate n records, one by one.
  - Read in the fields of the records.
  - Modify the links of the records so that the chain is formed.



Creating a list cont'd.

```
node *create_list()
{
    int  k, n;
    node  *p, *head;

    printf  ("\n How many elements to enter?");
     scanf ("%d", &n);

    for  (k=0; k<n; k++)
    {
        if (k == 0) {
          head = (node *) malloc(sizeof(node));
          p = head;
         }
         else {
               p->next  = (node *) malloc(sizeof(node));
               p = p->next;
              }

        scanf ("%d %s %d", &p->roll, p->name, &p->age);
    }

    p->next  =  NULL;
    return (head);
}
```

- To be called from `main()` function as:

```
    node *head;
……
head = create_list();
```

Traversing the List

What is to be done?

- Once the linked list has been constructed and *head* points to the first node of the list,
  - Follow the pointers.
  - Display the contents of the nodes as they are traversed.
  - Stop when the *next* pointer points to NULL.

```
void display (node *head)
{
  int  count = 1;
  node  *p;

  p = head;
  while (p != NULL)
  {
    printf ("\nNode %d: %d %s %d", count,
                    p->roll, p->name, p->age);
    count++;
    p = p->next;
  }
  printf ("\n");
}
```

- To be called from `main()` function as:

```
    node *head;
.....
display (head);
```

Inserting a Node in a List

How to do?

- The problem is to insert a node *before a specified node*.
  - Specified means some value is given for the node (called *key*).
  - In this example, we consider it to be `roll`.
- Convention followed:
  - If the value of roll is given as *negative*, the node will be inserted at the *end* of the list.

Inserting a node in list Cont'd.

- When a node is added at the beginning,
  - Only one next pointer needs to be modified.
    - *head* is made to point to the new node.
    - New node points to the previously first element.
- When a node is added at the end,
  - Two next pointers need to be modified.
    - Last node now points to the new node.
    - New node points to NULL.
- When a node is added in the middle,
  - Two next pointers need to be modified.
    - Previous node now points to the new node.
    - New node points to the next node.

```
void insert (node *head)
{
    int  k = 0, rno;
    node *p, *q, *new;

    new = (node *) malloc(sizeof(node));

    printf ("\nData to be inserted: ");
      scanf ("%d %s %d", &new->roll, new->name, &new->age);
    printf ("\nInsert before roll (-ve for end):");
      scanf ("%d", &rno);

    p = head;

    if (p->roll == rno)       /* At the beginning */
    {
        new->next = p;
        head = new;
    }
```

```
  else
     {
    while ((p != NULL) && (p->roll != rno))
         {
             q = p;
             p = p->next;
         }

         if   (p == NULL)          /* At the end */
         {
             q->next = new;
             new->next = NULL;
         }
     else if   (p->roll  == rno)
                              /* In the middle */
                 {
                     q->next = new;
                     new->next = p;
                 }
     }
}
```

The pointers q and p always point to consecutive nodes.

- To be called from `main()` function as:

```
    node *head;
.....
insert (head);
```

Deleting a node from the list

What is to be done?

- Here also we are required to delete a specified node.
  - Say, the node whose `roll` field is given.
- Here also three conditions arise:
  - Deleting the first node.
  - Deleting the last node.
  - Deleting an intermediate node.

```
void  delete (node **head)
{
    int  rno;
    node  *p, *q;

    printf ("\nDelete for roll :");
      scanf ("%d", &rno);

    p = *head;
    if  (p->roll == rno)
            /* Delete the first element */
    {
        *head = p->next;
        free (p);
    }
```

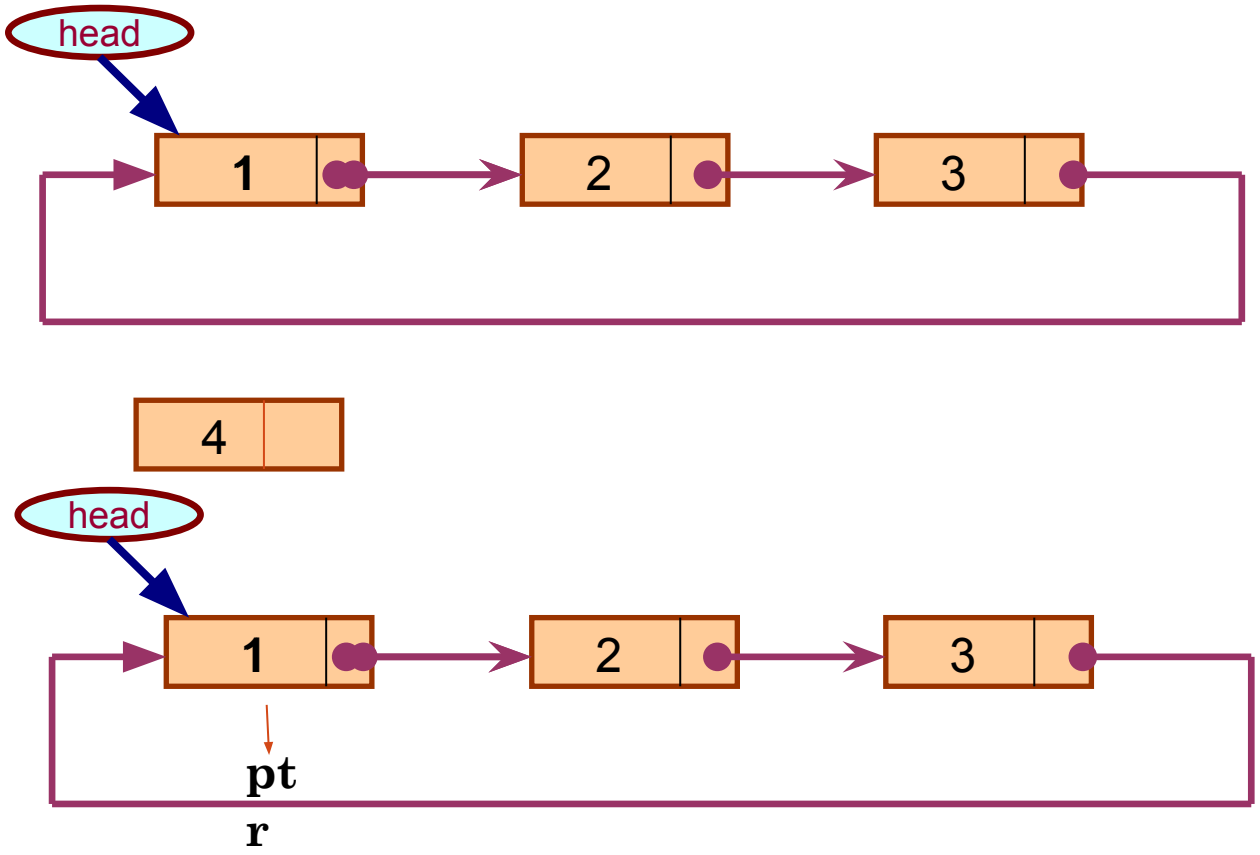
```
  else
     {
          while  ((p != NULL) && (p->roll != rno))
          {
               q = p;
               p  =  p->next;
          }

          if   (p == NULL)       /* Element not found */
             printf ("\nNo match :: deletion failed");

          else if (p->roll == rno)
                         /* Delete any other element */
               {
                    q->next  =  p->next;
                    free (p);
               }
     }
}
```

Circular linked list

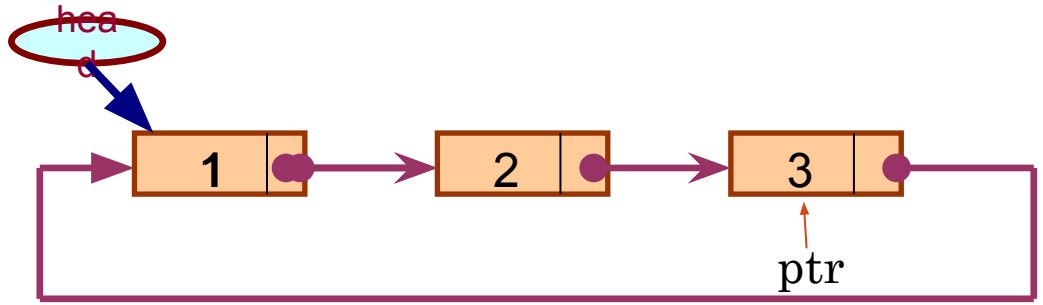
- Last node points to the header node.
- Advantages over single linked list.
  - **Accessibility of a member node in the list:** In linear linked list member node is accessible only through hear node, whereas the circular linked list is accessible through any node by merely chaining through the list.
  - **Null link problem**: In circular linked list we need not to maintain the null pointer.
  - **Easy implementation** of operations like merging, deletion, etc.

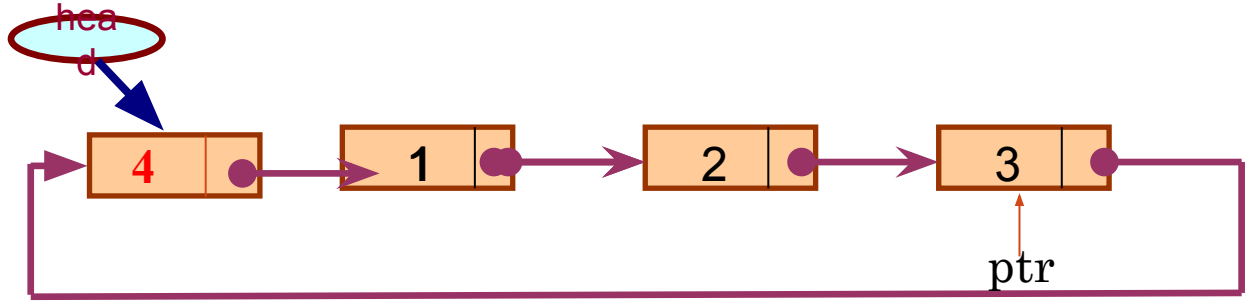
Insertion in Circular linked list

- Inserting at the beginning of the circular linked list



Insertion in Circular linked list





Insertion in Circular linked list

```
void insertion_At_begin (int data)
{
  struct node *new_node, *cur_Node;
  if (head == NULL)
    {
      printf ("The List is empty");
    }
  else
    {
      new_node = (struct node *) malloc
(sizeof (struct node));
      new_node->data = data;
      new_node->next = head;
      cur_Node = head;
      while (cur_Node->next != head)
       {
         cur_Node = cur_Node->next;
       }
      cur_Node->next = new_node;
      head = new_node;
    }
}
```

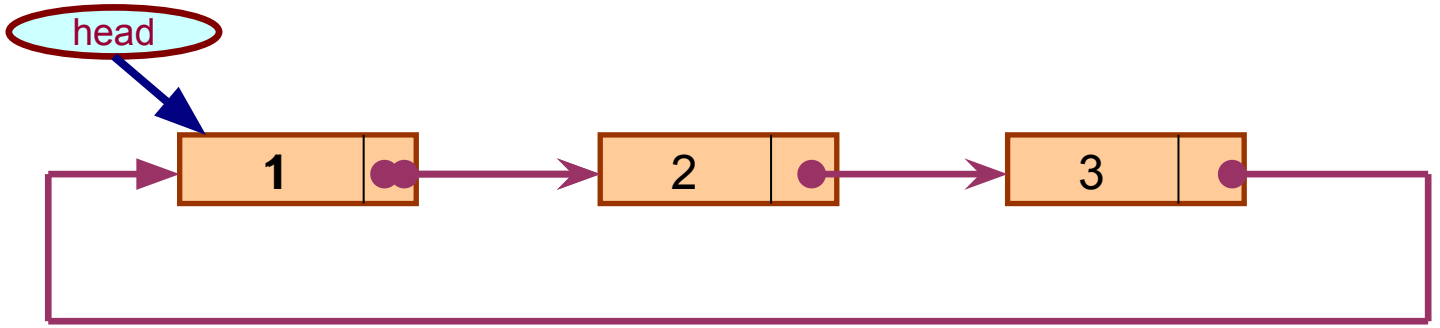
Insertion in Circular linked list

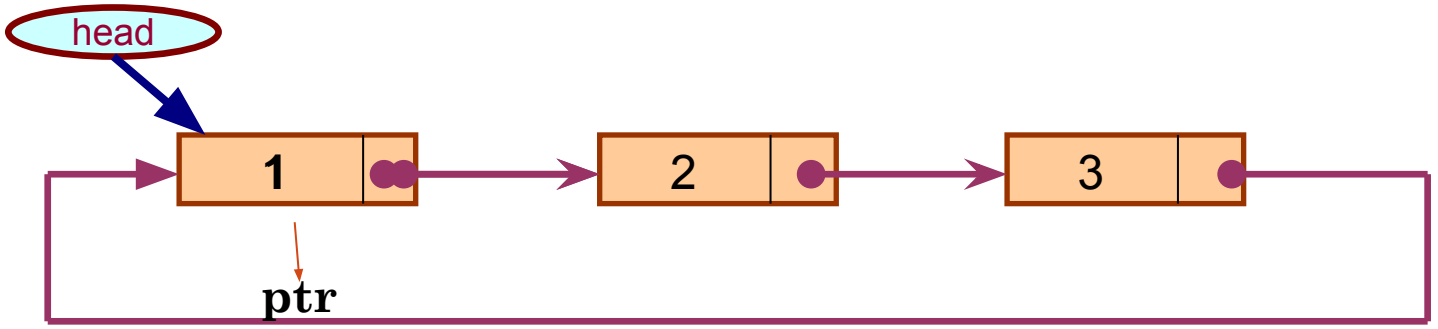
Practice Problems:

1. Inserting at the end of the circular linked list.
2. Inserting at the nth position of the circular linked list.

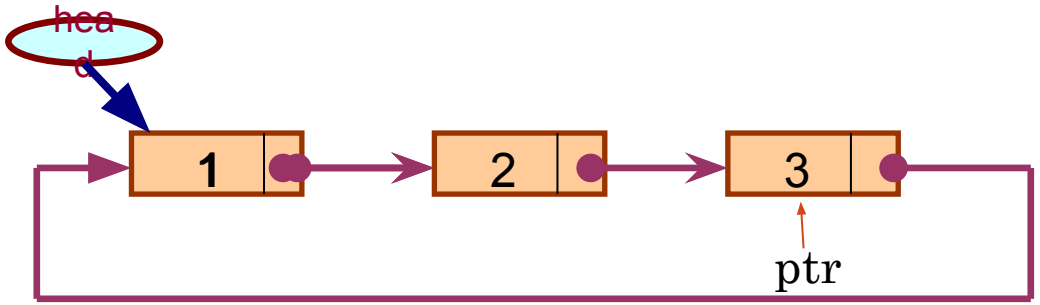
Deleting from circular linked list

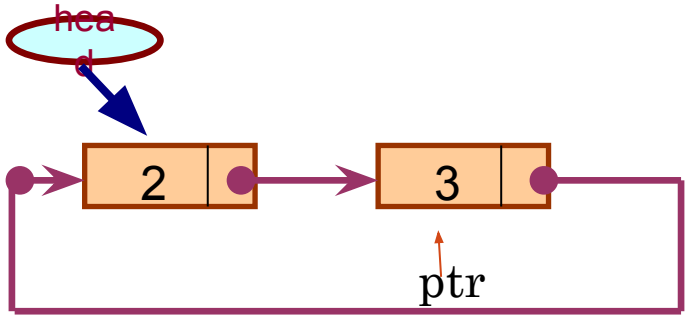
- Deleting from the beginning of the circular linked list





Deleting from circular linked list





Deleting from circular linked list

```
void deletion_beginning
{
  struct node *temp;
  if(head == NULL)
{
  printf("\n UNDERFLOW\n");
}
  else if(head->next == head)
{
  head = NULL;
  free(head);
  printf("\nNode Deleted\n");
}
```

```
else
{
ptr = head;
  while(ptr -> next != head)
{
  ptr = ptr -> next;
}
  ptr -> next = head -> next;
  ptr = head;
  head = head -> next;
    free(ptr);
  printf("\nNode Deleted\n");
}
}
```

Deleting from circular linked list

Practice Problems:

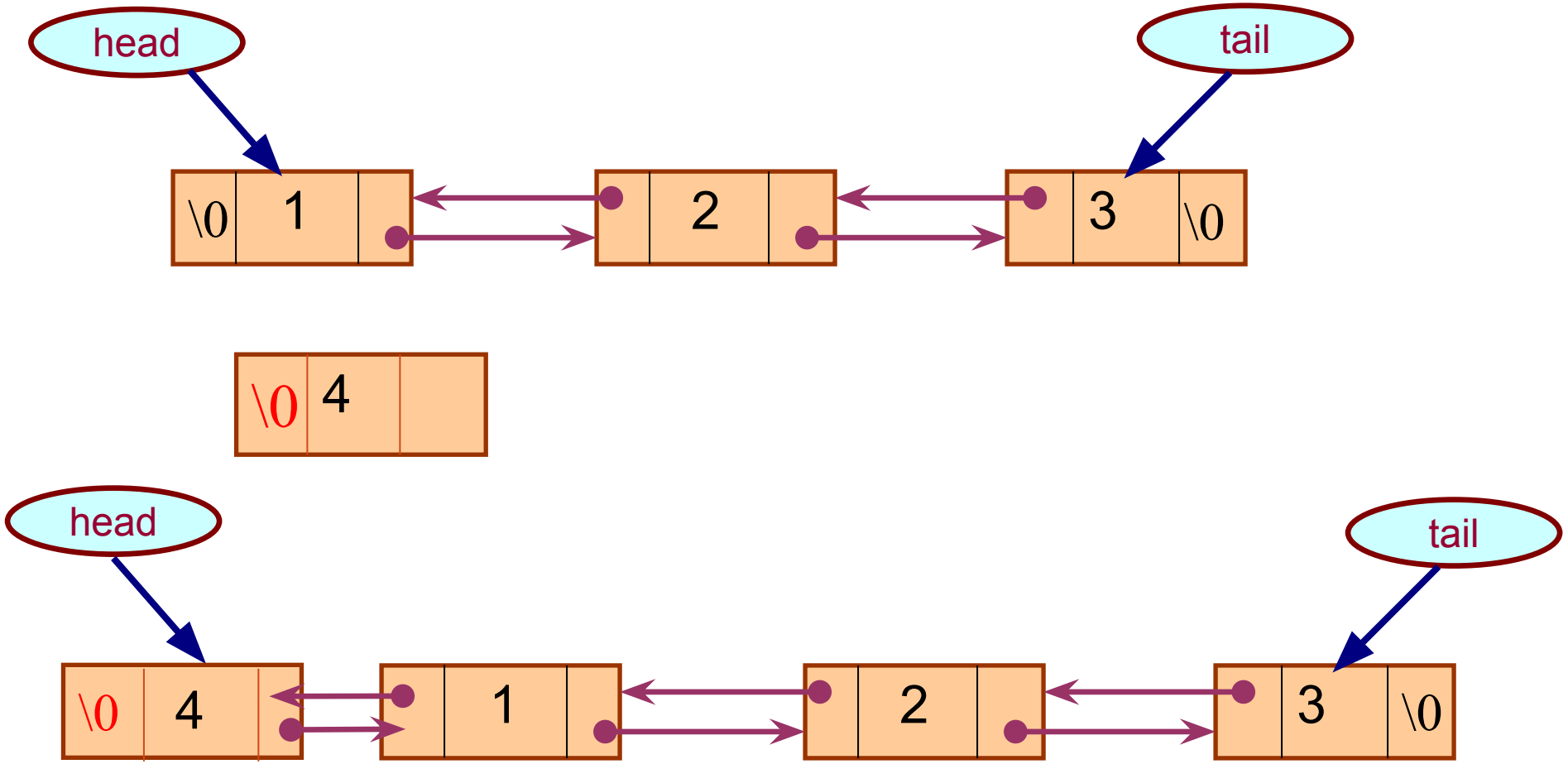
1. Deleting from the end of the circular linked list.
2. Deleting from the nth position of the circular linked list.

Doubly Linked List

- Contains pointer to the next and to the previous node.
- The structure of a doubly linked list can be given as:

```
struct node
{
        struct node *prev;
        int data;
        struct node *next;
}
```

Insertion in Doubly linked list



Insertion in Doubly linked list

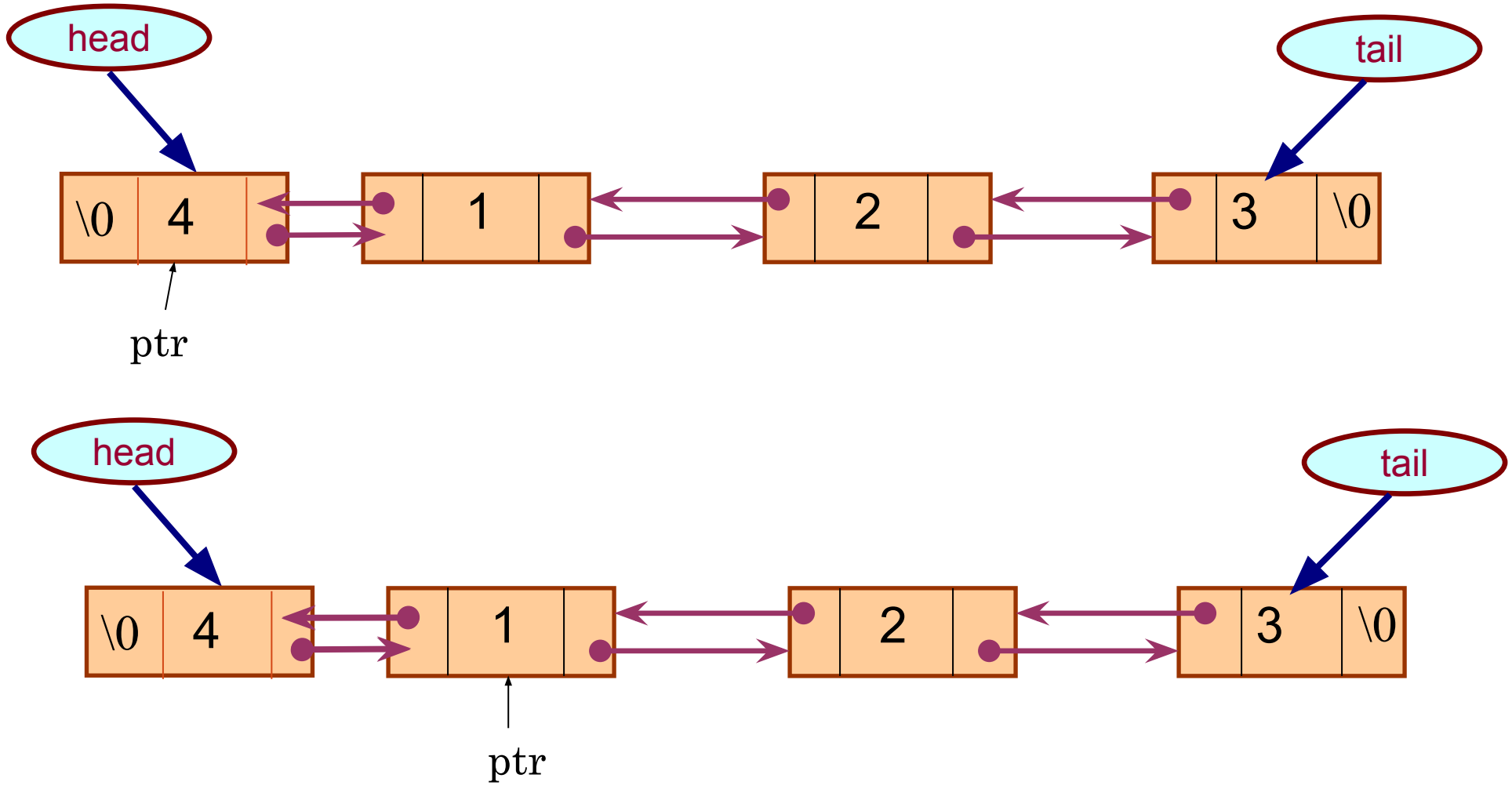
```
void insert(struct Node** head, int data){
struct Node* newNode = (struct Node* malloc(sizeof(struct Node));
  newNode->data = data;
  newNode->next = *head;
  newNode->prev = NULL;
if(*head !=NULL)
  (*head)->prev = newNode;
*head = newNode;
}
```

Insertion in Doubly linked list

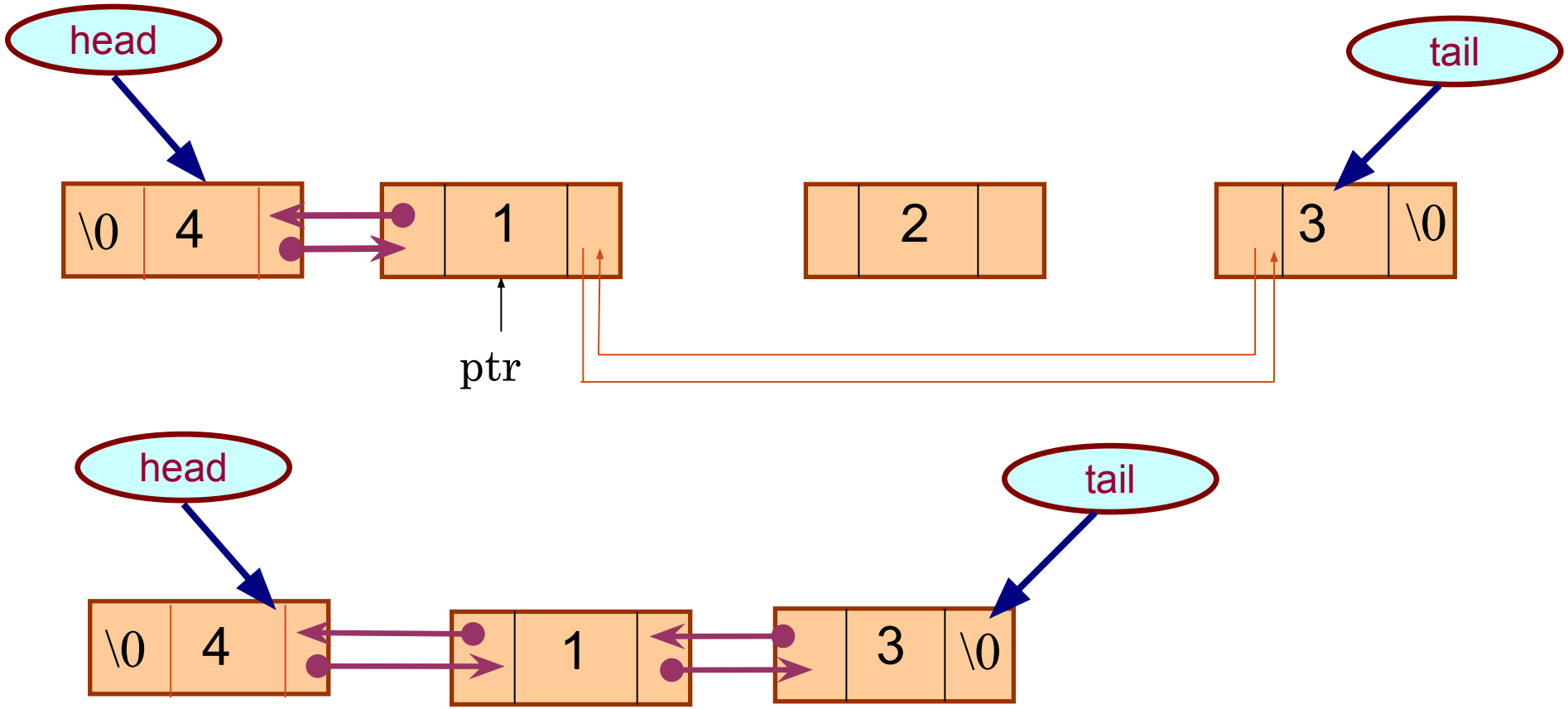
Practice Problems:

1. Inserting at the end of the doubly linked list.
2. Inserting at the nth position of the doubly linked list.

Deletion in doubly linked list



Deletion in doubly linked list



Deletion in doubly linked list

```
void delete(struct Node** head, int delVal)
{
  struct Node*ptr= *head;
  struct Node* previous = (struct Node*) malloc(sizeof(struct Node));
 if(ptr->next == NULL)
  {
    *head = NULL;
    free(ptr);
    printf("Value %d, deleted \n",delVal);
    return;

  }
```

Contd...

```
if(ptr!=NULL && ptr -> data==delVal)
  {
*head = ptr -> next;              //changing head to next in the list
    (*head)->prev = NULL;    //assign new head's previous value to NULL
free(ptr);
    printf("Value %d, deleted \n",delVal);
    return;
  }
```

Contd...

```
//run until we find the value to be deleted in the list
  while (ptr != NULL && ptr>data != delVal)
  {
    //store previous link node as we need to change its next val
    previous = ptr;
    ptr = ptr->next;
  }
  //if value is not present then
  //ptr will have traversed to last node NULL
    if(ptr==NULL)
  {
    printf("Value not found\n");
    return;
  }
```

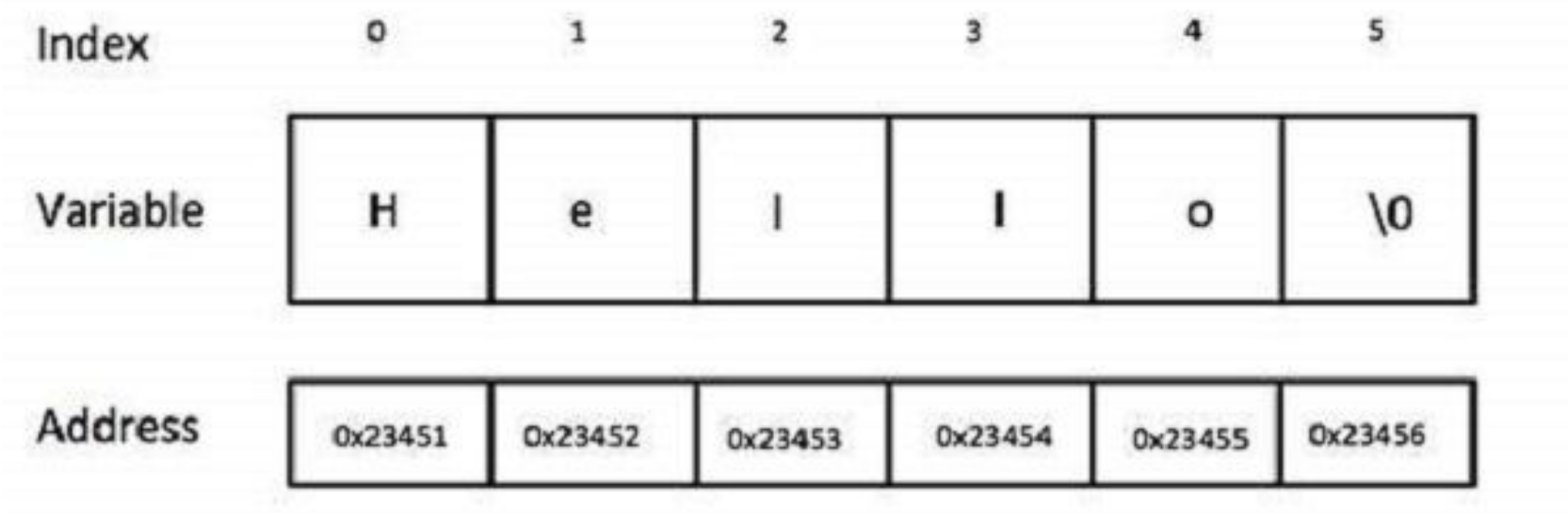
Data Structures and Algorithms

Module 2

Indian Institute of Information Technology Sri City, Chittoor

Data Structures with Strings

- Strings are defined as an *array of characters.*
- The difference between a character array and a string is the *string is terminated with a special character '\0'.*



Abstract Data Type

- Abstract Data Type (ADT) is a design tool.
- Concerns only on the important concept or model.
- No concern on implementation details.
- **Stack & Queue** is an example of ADT.
- An array is not ADT.

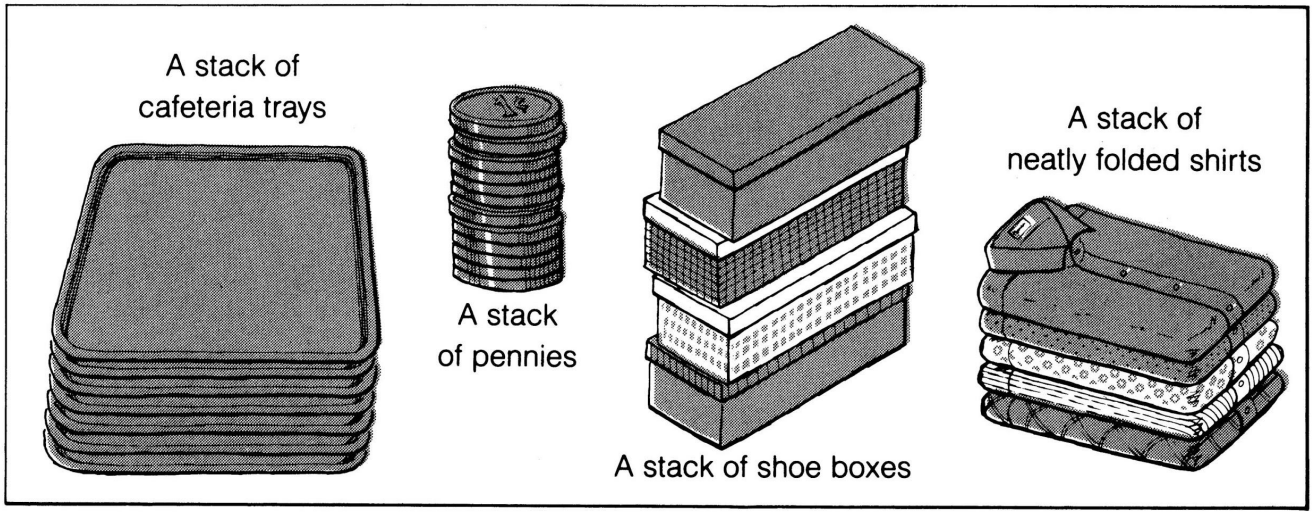
What is the Difference?

- **Stack & Queue vs. Array**
  - Arrays are data storage structures while stacks and queues are specialized DS and used as programmer's tools.

- Stack – a container that allows push and pop.
- Queue - a container that allows enqueue and dequeue.
- No concern on implementation details.
- In an array any item can be accessed, while in these data structures access is restricted.
- They are more abstract than arrays.

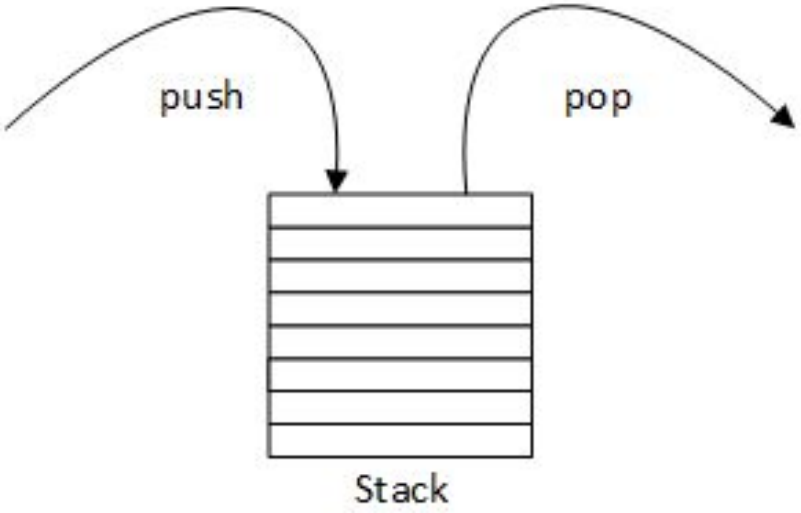
Stacks

- It is an *ordered group of homogeneous items of elements.*
- Elements are added to and removed from the *top of the stack* (the most recently added items are at the top of the stack).
- The last element to be added is the first to be removed (*LIFO: Last In, First Out*).



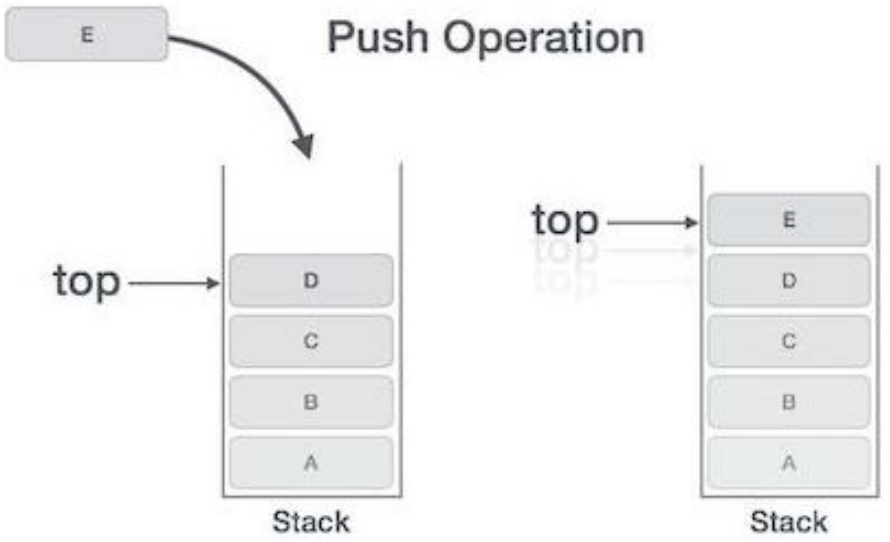
Stack operations

- Placing a data item on the top is called "*pushing*", while removing an item from the top is called "*popping*" it.
- *push* and *pop* are the primary stack operations.
- Some of the applications are: microprocessors, some older calculators etc.

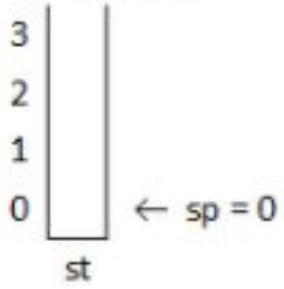
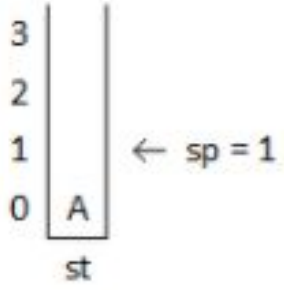
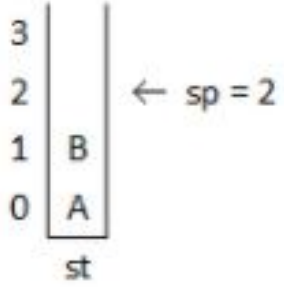
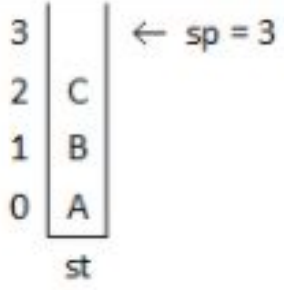


Push Operation

- **Step 1** − Checks if the stack is full.
- **Step 2** − If the stack is full, produces an error and exit.
- **Step 3** − If the stack is not full, increments **top** to point next empty space.
- **Step 4** − Adds data element to the stack location, where top is pointing.
- **Step 5** − Returns success.



Contd…

Operation	Picture	Execution
Stack is empty		
`push('A');`		`st[0] = 'A';` `sp = 0 + 1;`
`push('B');`		`st[1] = 'B';` `sp = 1 + 1;`
`push('C');`		`st[2] = 'C';` `sp = 2 + 1;`

Algorithm and Implementation

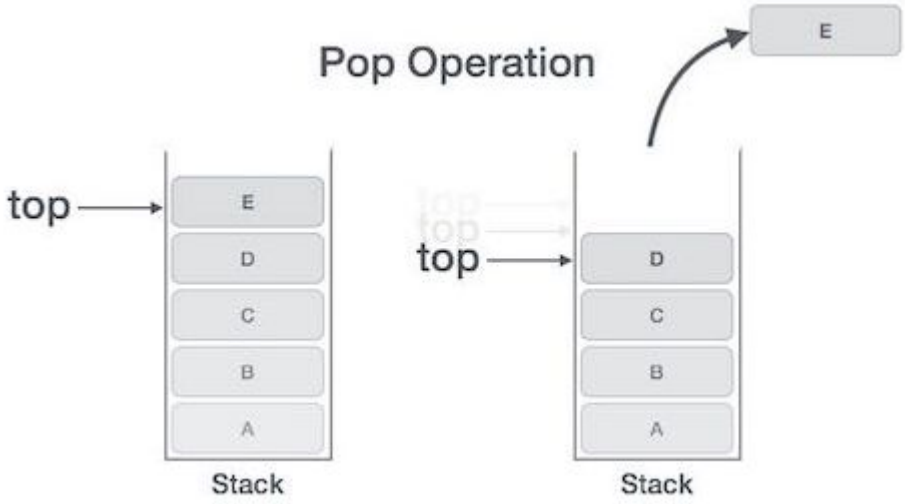
```
begin procedure push: stack, data

   if stack is full
      return null
   endif

   top ← top + 1
   stack[top] ← data

end procedure
```

Time Complexity : O(1)

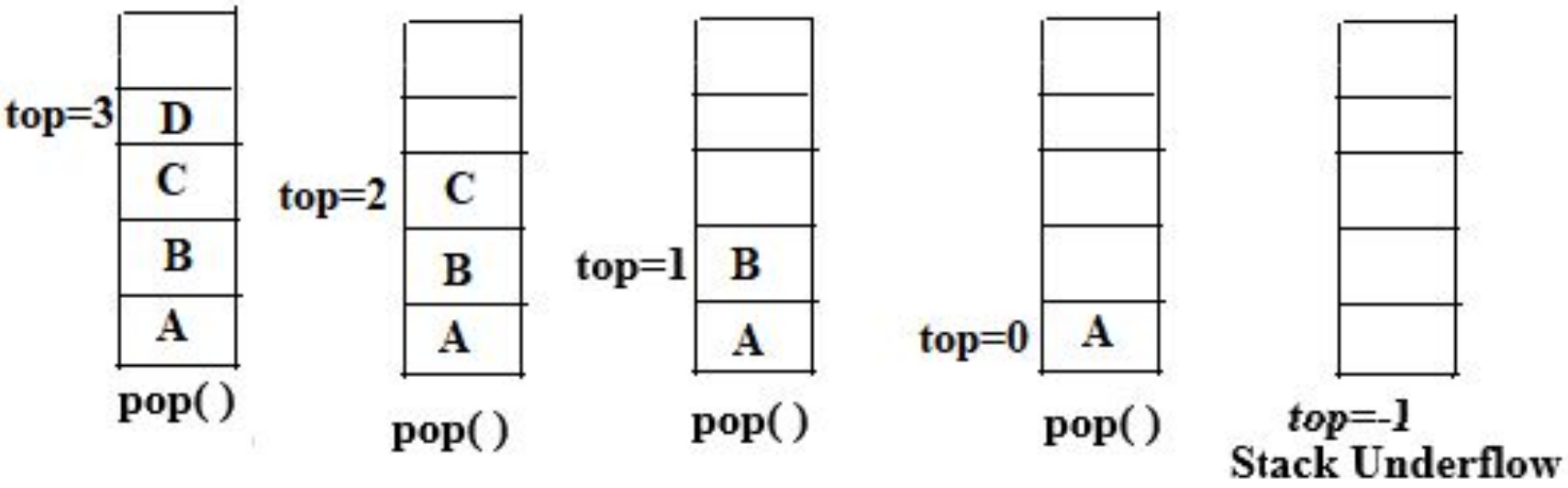
```
void push(int data) {
   if(!isFull()) {
      top = top + 1;
      stack[top] = data;
   } else {
      printf("Could not insert data, Stack is full.\n");
   }
}
```

Pop Operation

- **Step 1** − Checks if the stack is empty.
- **Step 2** − If the stack is empty, produces an error and exit.
- **Step 3** − If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** − Decreases the value of top by 1.
- **Step 5** − Returns success.



Contd...


top=3
D
C
B
A
pop( )
top=2
C
B
A
pop( )
top=1
B
A
pop( )
top=0
A
pop( )
top=-1
Stack Underflow

Algorithm and Implementation

```
begin procedure pop: stack

   if stack is empty
      return null
   endif

   data ← stack[top]
   top ← top - 1
   return data

end procedure
```

Time Complexity : O(1)

```
int pop(int data) {

   if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   } else {
      printf("Could not retrieve data, Stack is empty.\n");
   }
}
```

Peek Operation

```
int peek()
{
   if (top == -1)
   {
      printf("Underflow");
      return 0;
   }
   else
   {
      return stack [top];
   }
}
```

Time complexity: O(1)

Stack overflow

- The condition resulting from trying to push an element onto a full stack.

```
if(stack.IsFull())
  stack.Push(item);
```

Stack underflow

- The condition resulting from trying to pop an empty stack.

```
if(stack.IsEmpty())
  stack.Pop(item);
```

Applications of Stack

1. Expression Evaluation
2. Expression Conversion
    i. Infix to Postfix
    ii. Infix to Prefix
3. Balanced Parenthesis
4. Memory Management

Expression Evaluation

Example

Evaluate the postfix expression
5 3 2 * + 4 - 5 +

(a) Input so far (shaded):
5 3 2 * + 4 - 5 +

(b) Input so far (shaded):
5 3 2 * + 4 - 5 +

(c) Input so far (shaded):
5 3 2 * + 4 - 5 +

(d) Input so far (shaded):
5 3 2 * + 4 - 5 +

(e) Input so far (shaded):
5 3 2 * + 4 - 5 +

(f) Input so far (shaded):
5 3 2 * + 4 - 5 +

(g) Input so far (shaded):
5 3 2 * + 4 - 5 +

(h) Input so far (shaded):
5 3 2 * + 4 - 5 +

(i) Input so far (shaded):
5 3 2 * + 4 - 5 +

The result of the computation is 12.

Infix to Postfix Conversion

Let, X is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression Y.

1. Push "(" onto Stack, and add ")" to the end of X.
2. Scan X from left to right and repeat Step 3 to 6 for each element of X until the Stack is empty.
3. If an operand is encountered, add it to Y.
4. If a left parenthesis is encountered, push it onto Stack.
5. If an operator is encountered ,then:
    1. Repeatedly pop from Stack and add to Y each operator (on the top of Stack) which has the same precedence as or higher precedence than operator.
    2. Add operator to Stack.
    [End of If]
6. If a right parenthesis is encountered ,then:
    1. Repeatedly pop from Stack and add to Y each operator (on the top of Stack) until a left parenthesis is encountered.
    2. Remove the left Parenthesis.
    [End of If]
    [End of If]
7. END.

Expression Conversion

Convert **3*3/(4-1)+6*2** expression into **postfix** form

Expression	Stack	Output
3	Empty	3
*	*	3
3	*	33
/	/	33*
(	/(	33*
4	/(	33*4
-	/(-	33*4
1	/(-	33*41
)	-	33*41-
+	+	33*41-/
6	+	33*41-/6
*	+*	33*41-/62
2	+*	33*41-/62
	Empty	**33*41-/62*+**

Infix to Prefix Conversion

Expression = **(A+B^C)*D+E^5**

Step 1. Reverse the infix expression.

5^E+D*)C^B+A(

Step 2. Make Every '(' as ')' and every ')' as '('

5^E+D*(C^B+A)

Step 3. Convert expression to postfix form. (shown in the next slide)

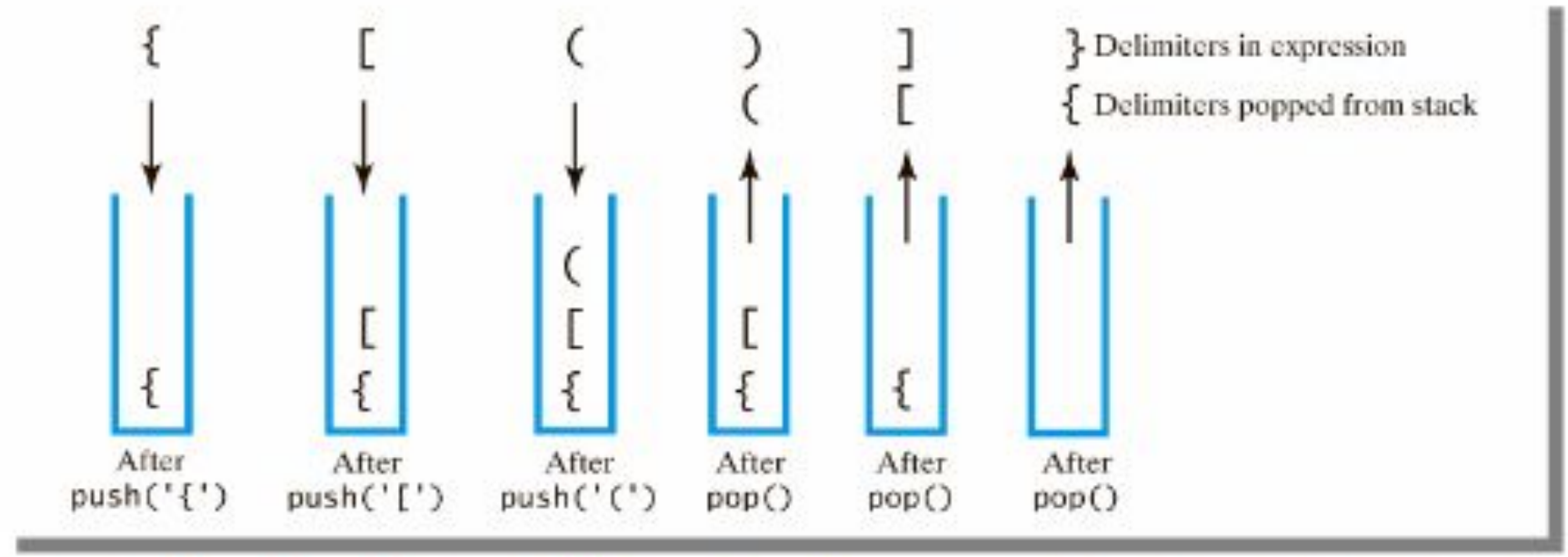
5E^DCB^A+*+

Step 4. Reverse the expression

+*+A^BCD^E5

Expression	Stack	Output	Comment
5^E+D*(C^B+A)	Empty	-	Initial
^E+D*(C^B+A)	Empty	5	Print
E+D*(C^B+A)	^	5	Push
+D*(C^B+A)	^	5E	Push
D*(C^B+A)	+	5E^	Pop And Push
*(C^B+A)	+	5E^D	Print
(C^B+A)	+*	5E^D	Push
C^B+A)	+*(	5E^D	Push
^B+A)	+*(	5E^DC	Print
B+A)	+*(^	5E^DC	Push
+A)	+*(^	5E^DCB	Print
A)	+*(+	5E^DCB^	Pop And Push
)	+*(+	5E^DCB^A	Print
End	+*	5E^DCB^A+	Pop Until '('
End	Empty	5E^DCB^A+*+	Pop Every element

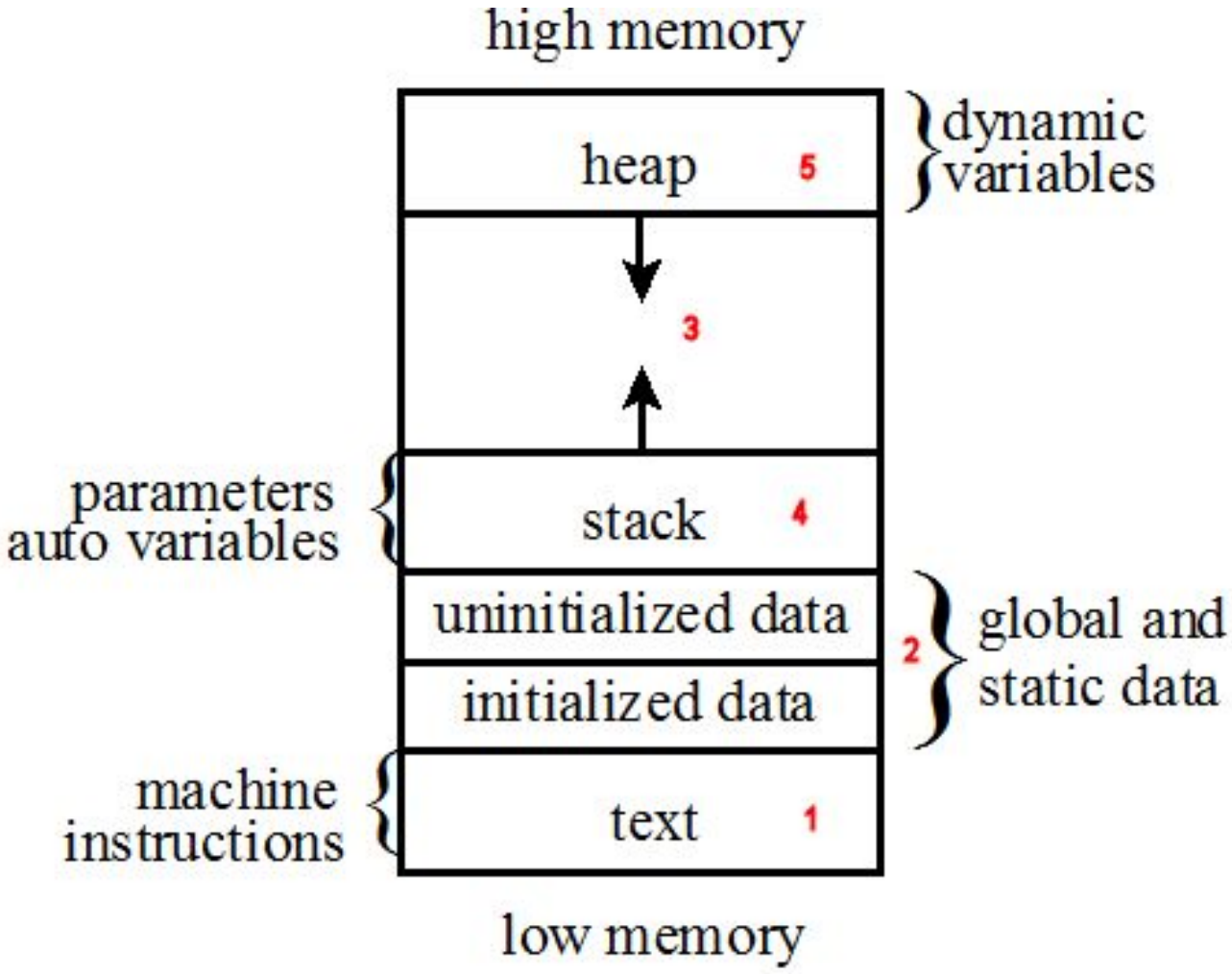
Balanced Parenthesis



The contents of a stack during the scan of an expression that contains the balanced delimiters {[()]}

```
a{b[c(d+e)/2 - f] +1}
```

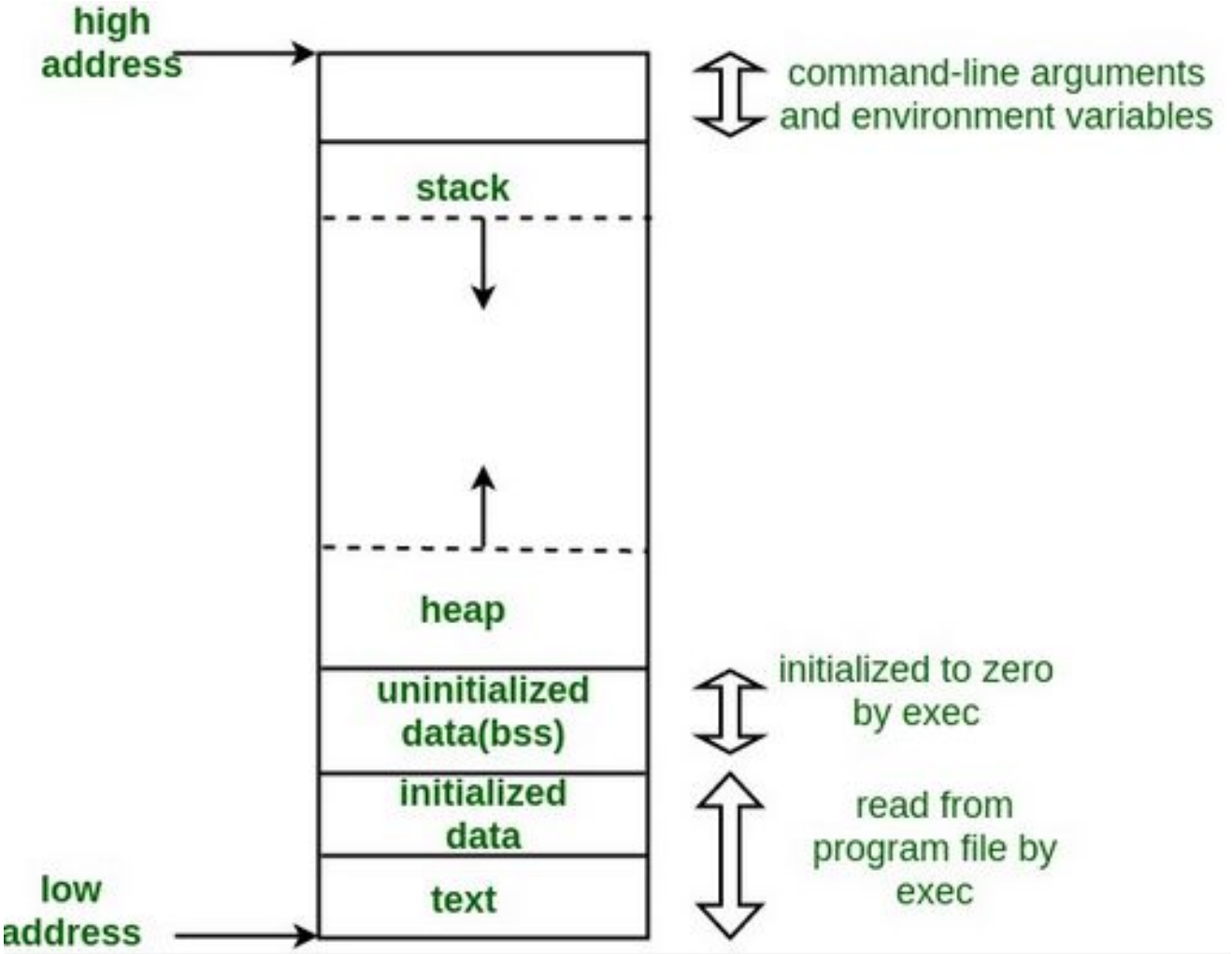
Memory Management using Stack



Contd…

1. The text area contains the program's machine instructions (i.e., the executable code).
2. Global variables are defined in global scope outside of any function or object; static variables have the keyword static included as part of their definition. The memory that holds global and static variables is typically allocated at program startup.
3. The space illustrated here was actually allocated but unused on a segmented-memory system, and provided space in to which the stack and heap could grow. On a paged-memory system this space is not actually allocated but signifies that there is space available for stack and heap growth.
4. The stack (sometimes called the runtime stack) contains all of the automatic (i.e., non-static) variables.
5. Memory is allocated from and returned to the heap with with the new and delete operators respectively.

Contd...



bss = **block started by symbol**

Data Structures and Algorithms

Module 2

Indian Institute of Information Technology Sri City, Chittoor

Queues

- Queue is an ADT data structure similar to stack, except that the *first item to be inserted is the first one to be removed*.
- This mechanism is called *First-In-First-Out (FIFO).*
- Placing an item in a queue is called "insertion or enqueue", which is done at the end of the queue called "rear".
- Removing an item from a queue is called "deletion or dequeue", which is done at the other end of the queue called "front".
- Some of the applications are: printer queue, keystroke queue, etc.

Bus Stop Queue

front rear

Bus Stop Queue

front rear

Bus Stop Queue

front rear

Bus Stop Queue

front

rear

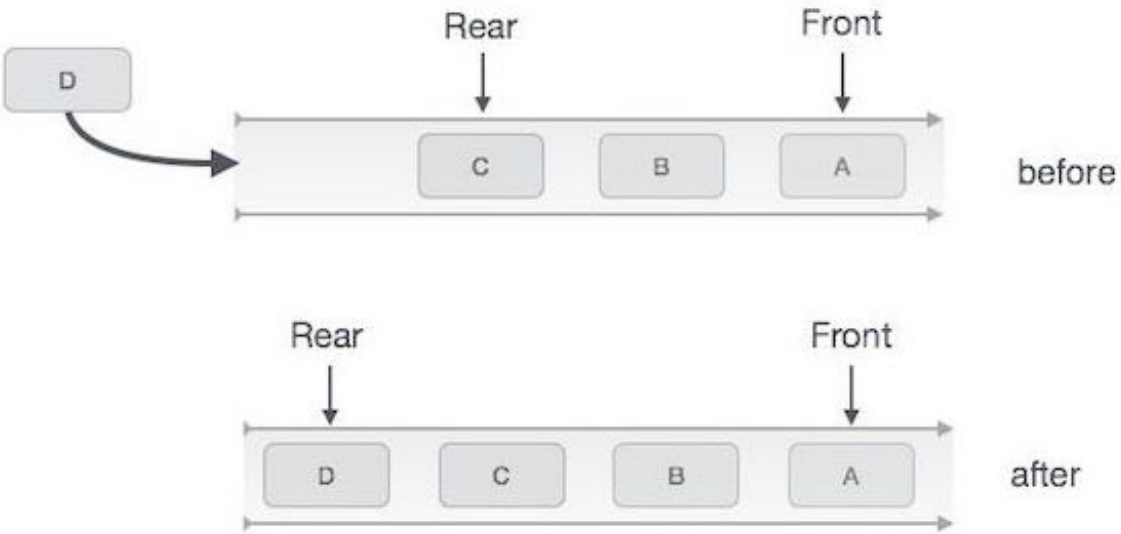
Queue Operations

- **enqueue()** − add (store) an item to the queue.
- **dequeue()** − remove (access) an item from the queue.
- **peek()** − gets the element at the front of the queue without removing it.
- **isfull()** − checks if the queue is full.
- **isempty()** − checks if the queue is empty.

Enqueue Operation

Queues maintain two data pointers, **front** and **rear**.

- **Step 1** − Check if the queue is full.
- **Step 2** − If the queue is full, produce overflow error and exit.
- **Step 3** − If the queue is not full, increment **rear** pointer to point the next empty space.
- **Step 4** − Add data element to the queue location, where the rear is pointing.
- **Step 5** − return success.



Algorithm and Implementation

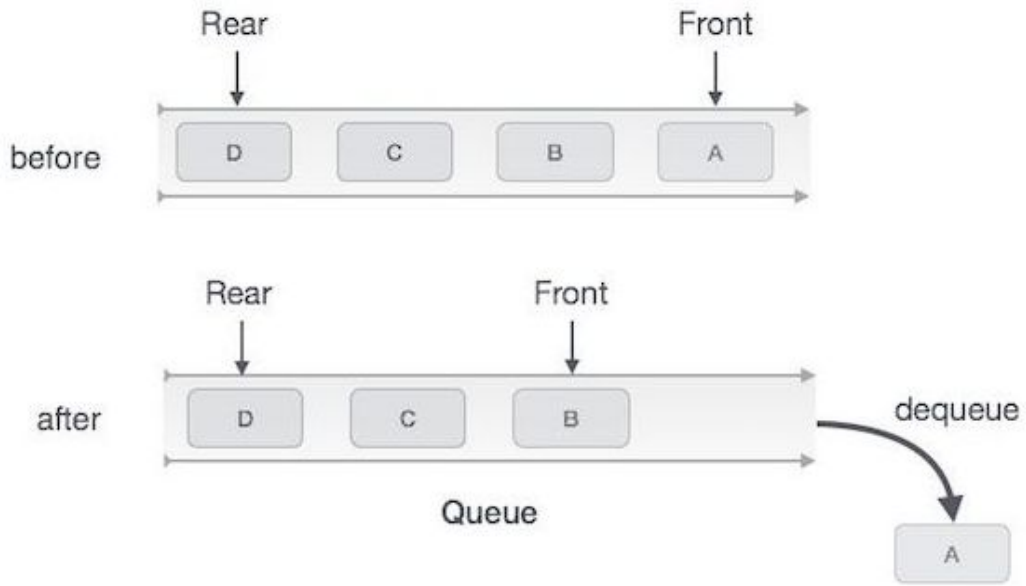
```
procedure enqueue(data)

   if queue is full
      return overflow
   endif

   rear ← rear + 1
   queue[rear] ← data
   return true

end procedure
```

Time Complexity: O(1)

```
int enqueue(int data)
   if(isfull())
      return 0;

   rear = rear + 1;
   queue[rear] = data;

   return 1;
end procedure
```

Dequeue Operation

- **Step 1** − Check if the queue is empty.
- **Step 2** − If the queue is empty, produce underflow error and exit.
- **Step 3** − If the queue is not empty, access the data where **front** is pointing.
- **Step 4** − Increment **front** pointer to point to the next available data element.
- **Step 5** − Return success.



Algorithm and Implementation

```
procedure dequeue

   if queue is empty
      return underflow
   end if

   data = queue[front]
   front ← front + 1
   return true

end procedure
```

Time Complexity: O(1)

```
int dequeue() {
   if(isempty())
      return 0;

   int data = queue[front];
   front = front + 1;

   return data;
}
```

Peek Operation

```
begin procedure peek
   return queue[front]
end procedure
```

```
int peek() {
   return queue[front];
}
```

Isfull Operation

```
begin procedure isfull

   if rear equals to MAXSIZE
      return true
   else
      return false
   endif

end procedure
```

```
bool isfull() {
   if(rear == MAXSIZE - 1)
      return true;
   else
      return false;
}
```

Isempty Operation

```
begin procedure isempty

   if front is less than MIN  OR front is greater than rear
      return true
   else
      return false
   endif

end procedure
```

```
bool isempty() {
   if(front < 0 || front > rear)
      return true;
   else
      return false;
}
```

Applications of Queue

- When a resource is shared among multiple consumers. Examples include ***CPU Scheduling, Disk Scheduling.***

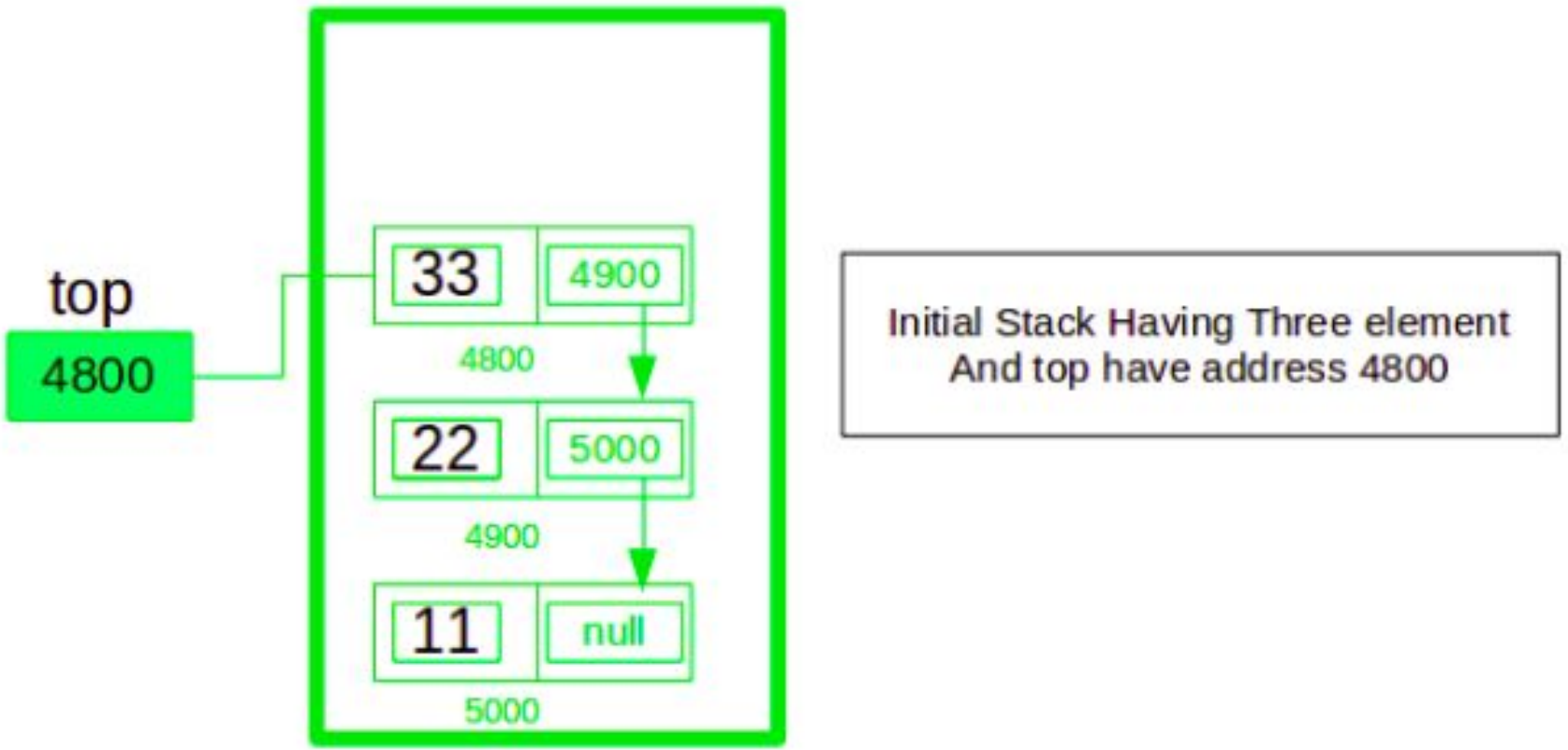
- When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include ***IO Buffers, pipes, file IO, etc.***

Data Structures and Algorithms

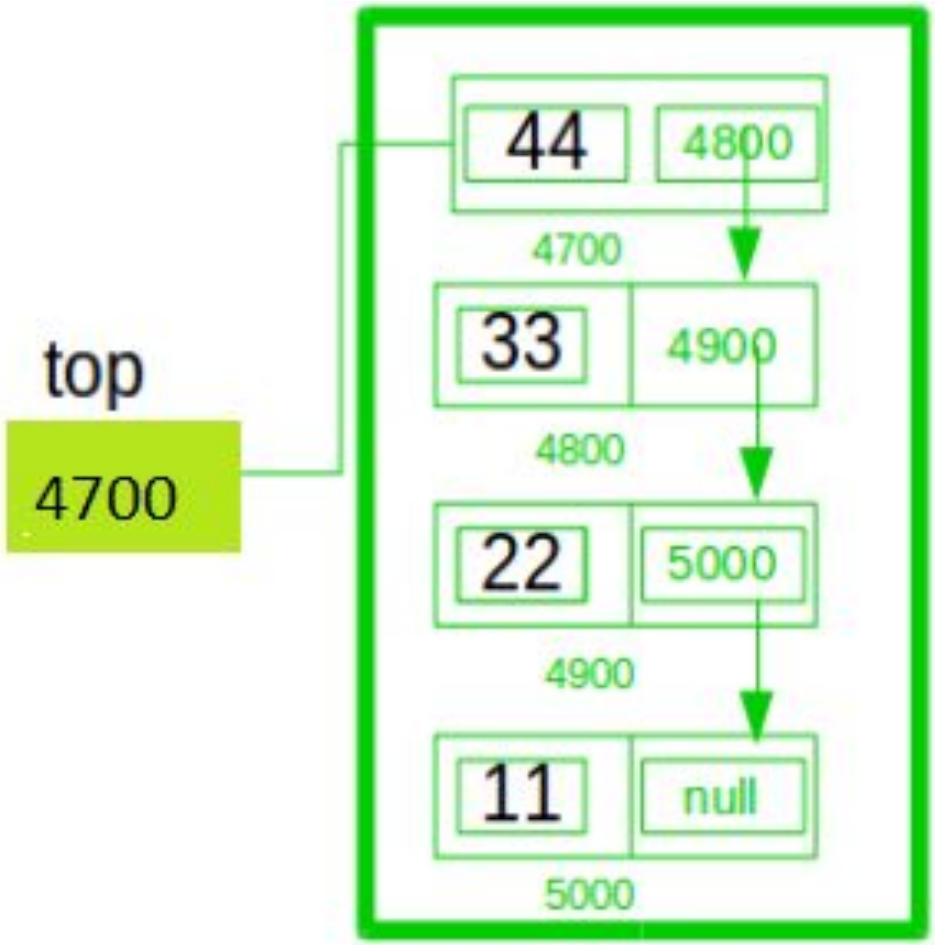
Module 2

Indian Institute of Information Technology Sri City, Chittoor

Stack using Linked List



Insert an element at the top of stack



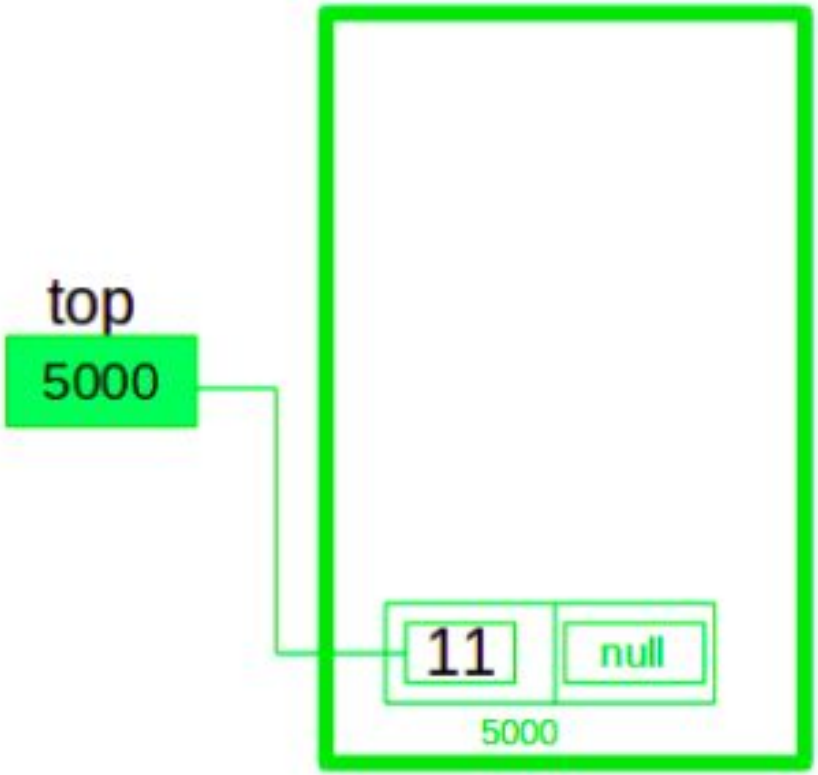
First create a temp node. Assign 44 into the data field and copy top into the link field. Then assign temp into the top.

Time Complexity: O(1)

Implementation

```
void push(int value)
{
   struct Node *newNode;
   newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   if(top == NULL)
      newNode->next = NULL;
   else
      newNode->next = top;
   top = newNode;
   printf("\nInsertion is Success!!!\n");
}
```

Delete top three elements




```
Pop three  element
    temp = top;
    top = top->link;
    temp->link = NULL;
    free(temp);
```

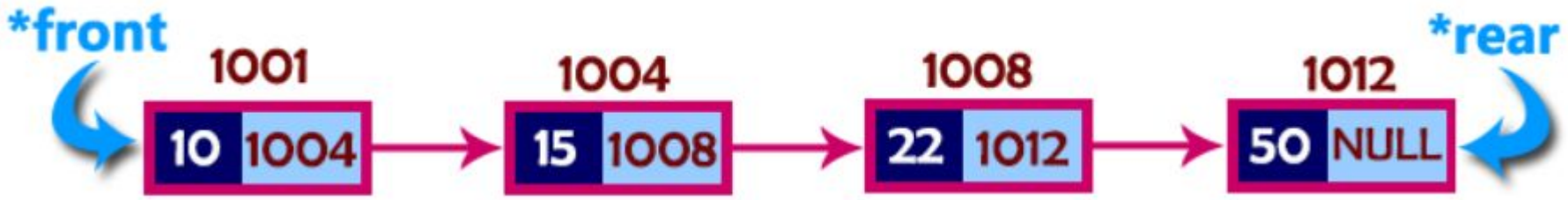
Implementation

```
void pop()
{
   if(top == NULL)
      printf("\nStack is Empty!!!\n");
   else{
      struct Node *temp = top;
      printf("\nDeleted element: %d", temp->data);
      top = temp->next;
      free(temp);
   }
}
```

Time Complexity: O(1)

Queue using Linked List

- The Queue implemented using linked list can organize as many data values as we want.
- In linked list implementation of a queue, the last inserted node is always pointed by '**rear**' and the first node is always pointed by '**front**'.



Enqueue: Algorithm

- **Step 1:** Allocate the space for the new node PTR
- **Step 2:** SET PTR -> DATA = VAL
- **Step 3:** IF FRONT = NULL
  SET FRONT = REAR = PTR
  SET FRONT -> NEXT = REAR -> NEXT = NULL
  ELSE
  SET REAR -> NEXT = PTR
  SET REAR = PTR
  SET REAR -> NEXT = NULL
  [END OF IF]
- **Step 4:** END

Time Complexity: O(1)

Implementation

```
void insert(int value)
{
   struct Node *newNode;
   newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   newNode -> next = NULL;
   if(front == NULL)
      front = rear = newNode;
   else{
      rear -> next = newNode;
      rear = newNode;
   }
   printf("\nInsertion is Success!!!\n");
}
```

Dequeue: Algorithm

- **Step 1:** IF FRONT = NULL
  Write " Underflow "
  Go to Step 5
  [END OF IF]
- **Step 2:** SET PTR = FRONT
- **Step 3:** SET FRONT = FRONT -> NEXT
- **Step 4:** FREE PTR
- **Step 5:** END

Time Complexity: O(1)

Implementation

```
void delete()
{
    if(front == NULL)
        printf("\nQueue is Empty!!!\n");
    else{
        struct Node *temp = front;
        front = front -> next;
        printf("\nDeleted element: %d\n", temp->data);
        free(temp);
    }
}
```

Data Structures and Algorithms

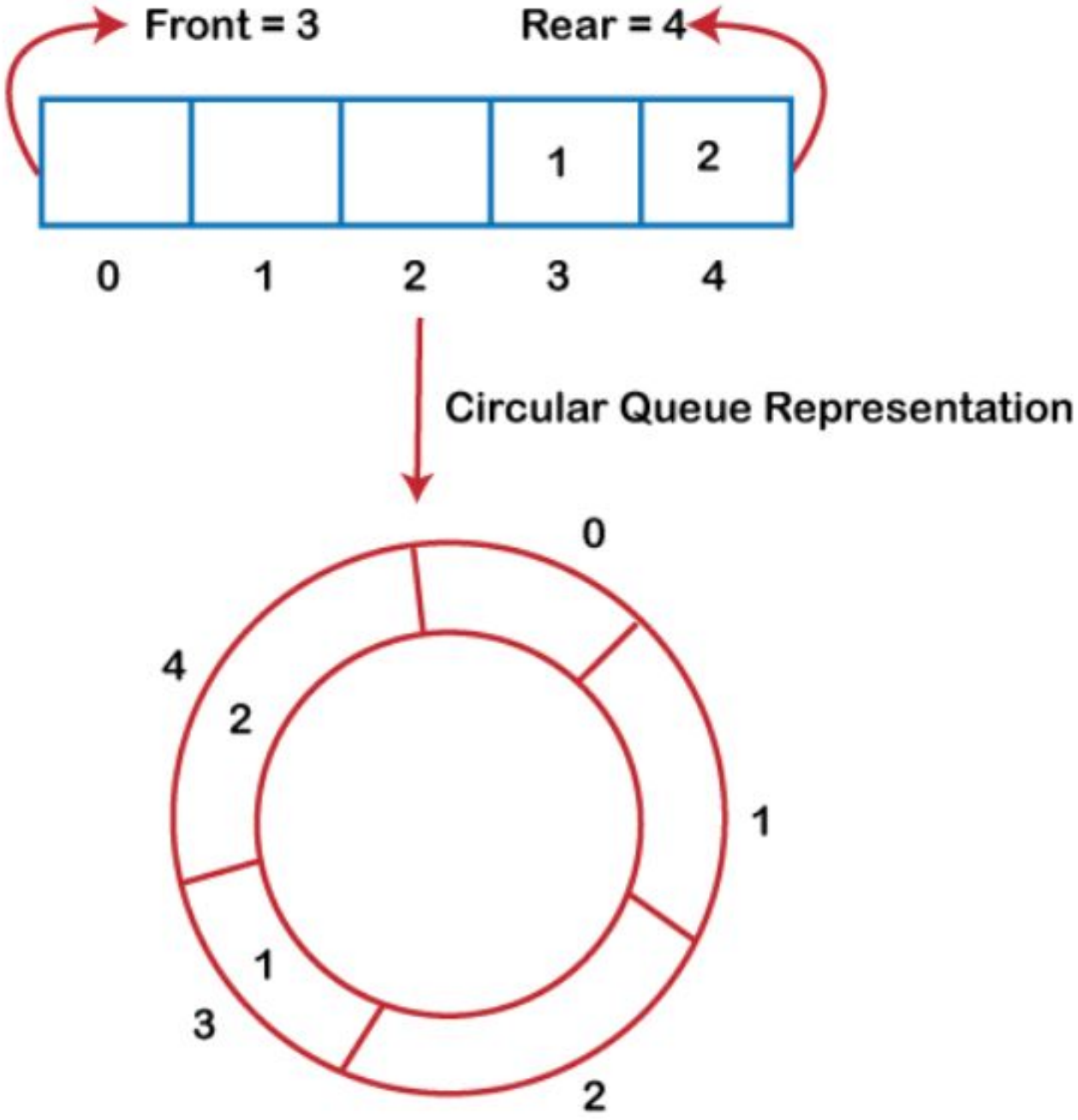
Module 2

Indian Institute of Information Technology Sri City, Chittoor

Circular Queue

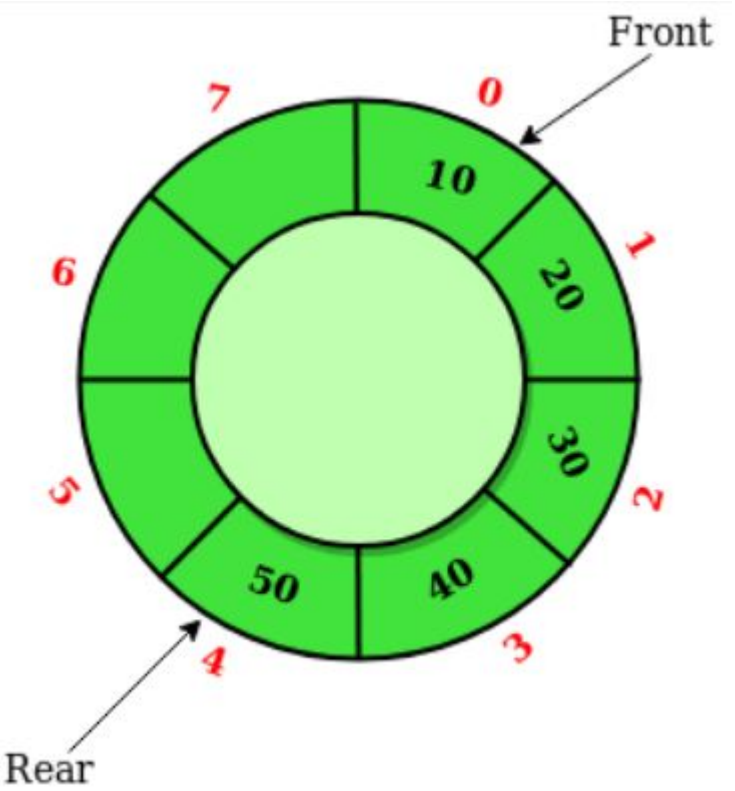
Why was the concept of the Circular Queue introduced?

- There was one limitation in the array implementation of *Queue.*
- ➢ If the rear reaches to the end position of the Queue then there might be possibility that *some vacant spaces are left in the beginning* which cannot be utilized.
- So, to overcome such limitations, the concept of the circular queue was introduced.

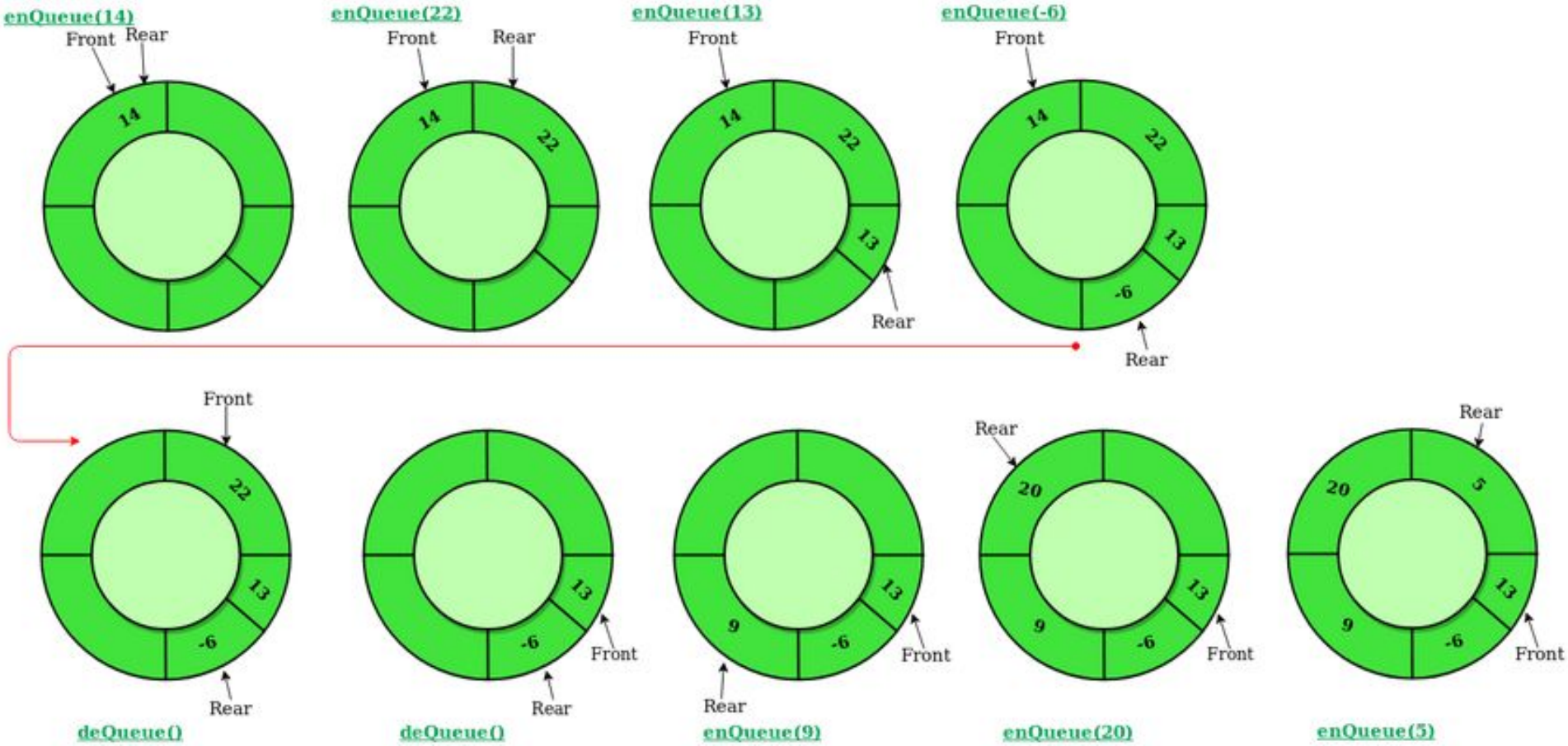

Front = 3
Rear = 4
1
2
0
1
2
3
4
Circular Queue Representation
0
4
2
1
1
3
2

Circular Queue

- Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the *last position is connected back to the first position* to make a circle. It is also called **'Ring Buffer'**.



- In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we can not insert the next element even if there is a space in front of queue.



Operations on Circular Queue

- **Front:** Get the front item from queue.
- **Rear:** Get the last item from queue.
- **enQueue(value):** This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at *Rear position*.
- **deQueue():** This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from *Front position.*

Enqueue Operation

The steps of enqueue operation are given below:

- First, check whether the **Queue is full or not.**
- Initially the front and rear are set to -1. When we insert the first element in a Queue, front and rear both are set to 0.
- When we insert another new element, the rear gets incremented, i.e., ***rear=rear+1***.

Scenarios for inserting an element

There are two scenarios in which queue is not full:

- **If rear != max - 1**, then rear will be incremented to **mod(maxsize)** and the new value will be inserted at the rear end of the queue.
- **If front != 0 and rear = max - 1**, it means that queue is not full, then set the value of rear to 0 and insert the new element there.

Contd...

There are two cases in which the element cannot be inserted:

- When **front == 0 && rear = max-1**, which means that front is at the first position of the Queue and rear is at the last position of the Queue.
- **front == rear + 1;**

Algorithm: Enqueue Operation

```
Step 1: IF (REAR+1)%MAX = FRONT
Write " OVERFLOW "
Goto step 4
[End OF IF]

Step 2: IF FRONT = -1 and REAR = -1
SET FRONT = REAR = 0
ELSE IF REAR = MAX - 1 and FRONT ! = 0
SET REAR = 0
ELSE
SET REAR = (REAR + 1) % MAX
[END OF IF]

Step 3: SET QUEUE[REAR] = VAL

Step 4: EXIT
```

Time Complexity : O(1)

Dequeue Operation

The steps of dequeue operation are given below:

- First, we check whether the **Queue is empty or not.** If the queue is empty, we cannot perform the dequeue operation.
- When the element is deleted, the ***value of front gets incremented by 1.***
- If there is only one element left which is to be deleted, then the ***front and rear are reset to -1.***

Algorithm

```
Step 1: IF FRONT = -1
Write " UNDERFLOW "
Goto Step 4
[END of IF]

Step 2: SET VAL = QUEUE[FRONT]

Step 3: IF FRONT = REAR
SET FRONT = REAR = -1
ELSE
IF FRONT = MAX -1
SET FRONT = 0
ELSE
SET FRONT = FRONT + 1
[END of IF]
[END OF IF]

Step 4: EXIT
```

Time Complexity : O(1)

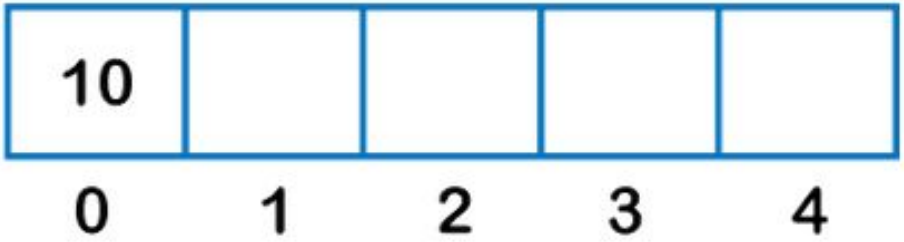
The enqueue and dequeue operation through the diagrammatic representation

Initially

Front = -1

Rear = -1

Insert 10

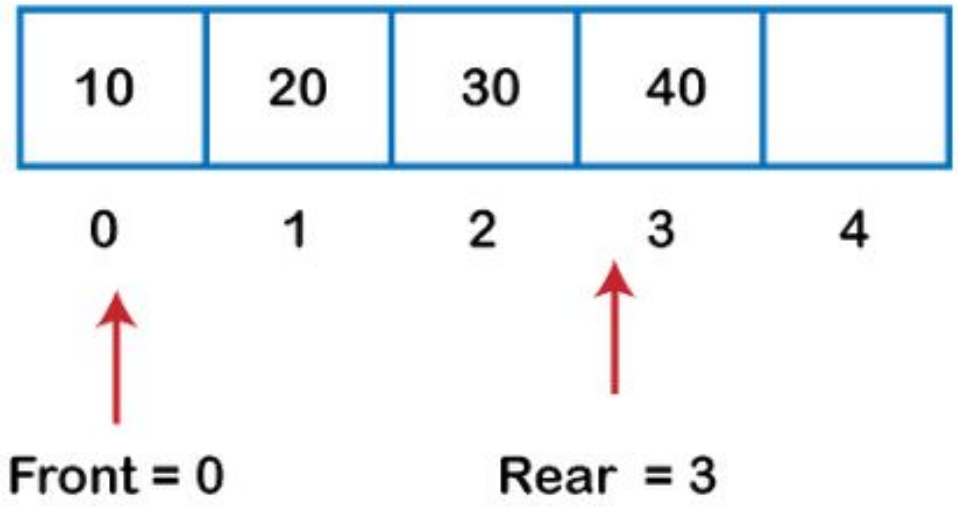


Front = 0

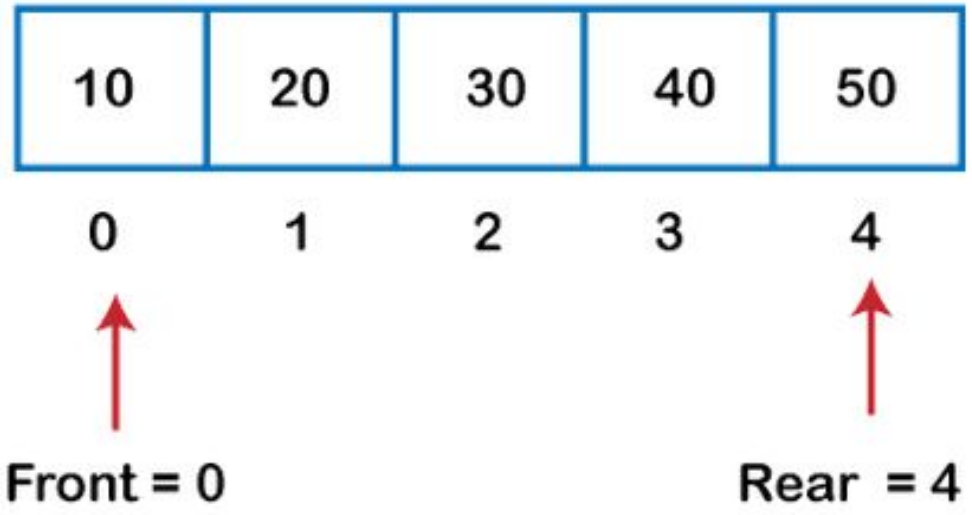
Rear = 0

Contd...

Insert 20, 30, and 40

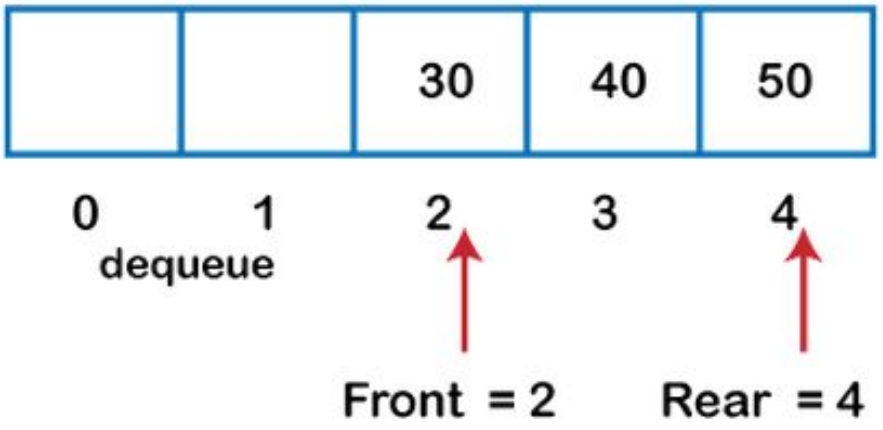


Insert 50

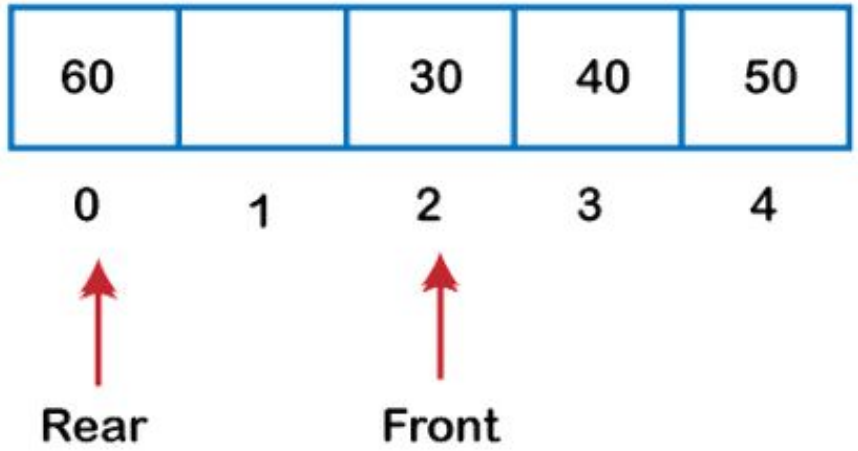


Contd...

Delete 10 and 20

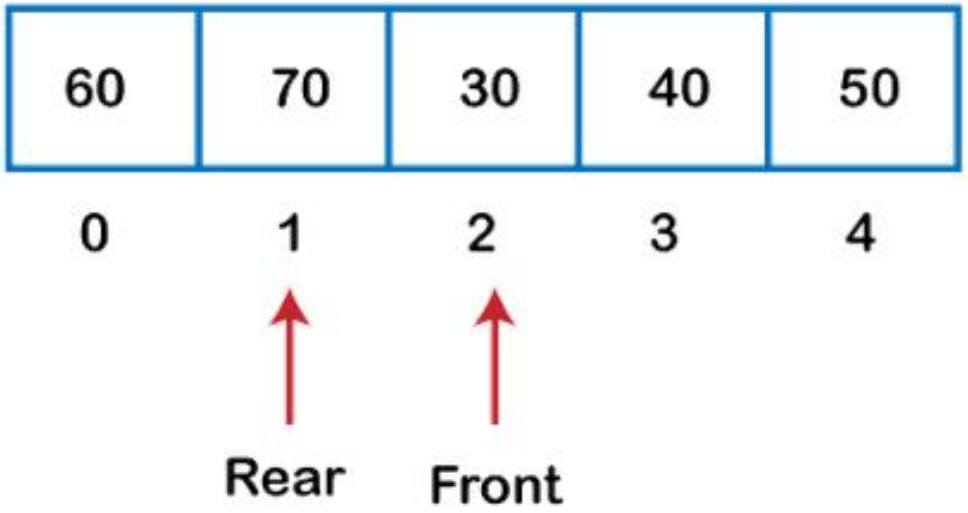


Insert 60



Contd...

Insert 70



Implementation of Insertion in circular queue using Array

```
#include <stdio.h>

# define max 6
int queue[max];  // array declaration
int front=-1;
int rear=-1;
// function to insert an element in a circular queue
void enqueue(int element)
{
   if(front==-1 && rear==-1)   // condition to check queue is empty
   {
      front=0;
      rear=0;
      queue[rear]=element;
   }
   else if((rear+1)%max==front)  // condition to check queue is full
   {
      printf("Queue is overflow..");
   }
   else
   {
      rear=(rear+1)%max;       // rear is incremented
      queue[rear]=element;     // assigning a value to the queue at the rear position.
   }
}
```

Implementation of Deletion in circular queue using Array

```
// function to delete the element from the queue
int dequeue()
{
   if((front==-1) && (rear==-1))  // condition to check queue is empty
   {
      printf("\nQueue is underflow..");
   }
 else if(front==rear)
{
  printf("\nThe dequeued element is %d", queue[front]);
  front=-1;
  rear=-1;
}
else
{
   printf("\nThe dequeued element is %d", queue[front]);
  front=(front+1)%max;
}
}
```

Display the elements using Array

```
void display()
{
   int i=front;
   if(front==-1 && rear==-1)
   {
      printf("\n Queue is empty..");
   }
   else
   {
      printf("\nElements in a Queue are :");
      while(i<=rear)
      {
         printf("%d,", queue[i]);
         i=(i+1)%max;
      }
   }
}
```

Implementation of circular queue using linked list

- As we know that linked list is a linear data structure that stores two parts, i.e., **data part and the address part** where address part contains the address of the next node.
- Here, linked list is used to implement the circular queue; therefore, the linked list follows the properties of the Queue.
- When we are implementing the circular queue using linked list then both the ***enqueue and dequeue*** operations take ***O(1)*** time.

Declaration of struct type node

```
struct node
{
   int data;
   struct node *next;
};
struct node *front=-1;
struct node *rear=-1;
```

Insert an element in the Queue

```
void enqueue(int x)
{
   struct node *newnode;  // declaration of pointer of struct node type.
   newnode=(struct node *)malloc(sizeof(struct node));  // allocating the memory to the newnode
   newnode->data=x;
   newnode->next=0;
   if(rear==-1)  // checking whether the Queue is empty or not.
   {
      front=rear=newnode;
      rear->next=front;
   }
   else
   {
      rear->next=newnode;
      rear=newnode;
      rear->next=front;
   }
}
```

Delete an element from the Queue

```
void dequeue()
{
   struct node *temp;   // declaration of pointer of node type
   temp=front;
   if((front==-1)&&(rear==-1))  // checking whether the queue is empty or not
   {
      printf("\nQueue is empty");
   }
   else if(front==rear)  // checking whether the single element is left in the queue
   {
      front=rear=-1;
      free(temp);
   }
   else
   {
      front=front->next;
      rear->next=front;
      free(temp);
   }
}
```

Get the front element of the Queue

```
int peek()
{
   if((front==-1) &&(rear==-1))
   {
      printf("\nQueue is empty");
   }
   else
   {
      printf("\nThe front element is %d", front->data);
   }
}
```

Display all the elements of the queue

```
void display()
{
   struct node *temp;
   temp=front;
   printf("\n The elements in a Queue are : ");
   if((front==-1) && (rear==-1))
   {
      printf("Queue is empty");
   }

   else
   {
      while(temp->next!=front)
      {
         printf("%d,", temp->data);
         temp=temp->next;
      }
      printf("%d", temp->data);
   }
}
```

Notations & Conversions

Infix: Infix expression is the normal expression that consists of operands and operators. For example, **A+B**

Postfix: Postfix expression consists of operands followed by operators. For example, **AB+**

Prefix: Prefix expression consists of operators followed by operands. For example, **+AB**

Recap

Infix to Postfix Conversion

Infix Expression: (A/(B-C)*D+E)

Symbol Scanned	Stack	Output
(	(	-
A	(	A
/	(/	A
(	(/(	A
B	(/(	AB
-	(/(-	AB
C	(/(-	ABC
)	(/	ABC-
*	(*	ABC-/
D	(*	ABC-/D
+	(+	ABC-/D*
E	(+	ABC-/D*E
)	Empty	ABC-/D*E+

Postfix Expression: ABC-/D*E+

Infix to Prefix Conversion

***** Recall the steps that we have discussed in the last week.***

Infix Expression : **A + (B * C - (D / E ^ F) * G) * H**

1. Reverse the infix expression :

   H *) G *) F ^ E / D (- C * B (+ A

1. Make Every " (" as ") " and every ") " as " ("

   H * (G * (F ^ E / D) - C * B) + A

1. Convert expression to postfix form: Try it yourself

   You will get: **HGFE^D/*CB*-*A+**

1. Reverse the postfix expression : **+A*-*BC*/D^EFGH**

Postfix to Infix Conversion

- Scan the given postfix expression from **left to right** character by character.
- If the character is an operand, push it into the stack.
- But if the character is an operator, pop the top two values from stack.

  Concatenate this operator with these two values (**2^{nd} top value+operator+1^{st} top value**) to get a new string.
- Now push this resulting string back into the stack.
- Repeat this process untill the end of postfix expression. Now the value in the stack is the infix expression.

Example

Input String	Postfix Expression	Stack (Infix)
ab*c+	b*c+	a
ab*c+	*c+	ab
ab*c+	c+	(a*b)
ab*c+	+	(a*b)c
ab*c+		((a*b)+c)

Prefix to Infix Conversion

- Scan the given prefix expression from **right to left** character by character.
- If the character is an operand, push it into the stack.
- But if the character is an operator, pop the top two values from stack.

  Concatenate this operator with these two values (**1st top value+operator+2nd top value**) to get a new string.
- Now push this resulting string back into the stack.
- Repeat this process untill the end of prefix expression. Now the value in the stack is the desired infix expression.

Example

Input String	Prefix Expression	Stack (Infix)
*-A/BC-/AKL	*-A/BC-/AK	L
*-A/BC-/AKL	*-A/BC-/A	LK
*-A/BC-/AKL	*-A/BC-/	LKA
*-A/BC-/AKL	*-A/BC-	L(A/K)
*-A/BC-/AKL	*-A/BC	((A/K)-L)
*-A/BC-/AKL	*-A/B	((A/K)-L)C
*-A/BC-/AKL	*-A/	((A/K)-L)CB
*-A/BC-/AKL	*-A	((A/K)-L)(B/C)
*-A/BC-/AKL	*-	((A/K)-L)(B/C)A
*-A/BC-/AKL	*	((A/K)-L)(A-(B/C))
*-A/BC-/AKL		((A-(B/C))*((A/K)-L))

Thank You !!!