NEHRU COLLEGE OF ENGINEERING AND RESEARCH CENTRE
(NAAC Accredited)
(Approved by AICTE, Affiliated to APJ Abdul Kalam Technological University, Kerala)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

COURSE MATERIAL

CS 205 DATASTRUCTURES

VISION OF THE INSTITUTION

To mould true citizens who are millennium leaders and catalysts of change through excellence in education.

MISSION OF THE INSTITUTION

NCERC is committed to transform itself into a center of excellence in Learning and Research in Engineering and Frontier Technology and to impart quality education to mould technically competent citizens with moral integrity, social commitment and ethical values.

We intend to facilitate our students to assimilate the latest technological know-how and to imbibe discipline, culture and spiritually, and to mould them in to technological giants, dedicated research scientists and intellectual leaders of the country who can spread the beams of light and happiness among the poor and the underprivileged.

ABOUT DEPARTMENT

- Established in: 2002
- Courses offered : B.Tech in Computer Science and Engineering

  M.Tech in Computer Science and Engineering

  M.Tech in Cyber Security
- Approved by AICTE New Delhi and Accredited by NAAC
- Affiliated to the A P J Abdul Kalam Technological University.

DEPARTMENT VISION

Producing Highly Competent, Innovative and Ethical Computer Science and Engineering Professionals to facilitate continuous technological advancement.

DEPARTMENT MISSION

1. To Impart Quality Education by creative Teaching Learning Process
2. To Promote cutting-edge Research and Development Process to solve real world problems with emerging technologies.
3. To Inculcate Entrepreneurship Skills among Students.
4. To cultivate Moral and Ethical Values in their Profession.

PROGRAMME EDUCATIONAL OBJECTIVES

PEO1: Graduates will be able to Work and Contribute in the domains of Computer Science and Engineering through lifelong learning.

PEO2: Graduates will be able to Analyse, design and development of novel Software Packages, Web Services, System Tools and Components as per needs and specifications.

PEO3: Graduates will be able to demonstrate their ability to adapt to a rapidly changing environment by learning and applying new technologies.

PEO4: Graduates will be able to adopt ethical attitudes, exhibit effective communication skills, Teamworkand leadership qualities.

PROGRAM OUTCOMES (POS)

Engineering Graduates will be able to:

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

COURSE OUTCOMES

SUBJECT CODE: C204	
COURSE OUTCOMES	
C205.1	**Analyze** performance of algorithms and design efficient programs to solve problems.
C205.2	**Use** appropriate data structures like arrays,linked list,stacks and queues to solve real world problems efficiently.
C205.3	**Categorize** different memory management techniques and the implementations of linear datastructures.
C205.4	**Represent** and manipulate data using nonlinear data structures like trees and graphs to design algorithms for various applications.
C205.5	**Illustrate and understand** various techniques for searching and sorting.
C205.6	**Illustrate** various hashing algorithms

PROGRAM SPECIFIC OUTCOMES (PSO)

PSO1: Ability to Formulate and Simulate Innovative Ideas to provide software solutions for Real-time Problems and to investigate for its future scope.

PSO2: Ability to learn and apply various methodologies for facilitating development of high quality System Software Tools and Efficient Web Design Models with a focus on performance optimization.

PSO3: Ability to inculcate the Knowledge for developing Codes and integrating hardware/software
products in the domains of Big Data Analytics, Web Applications and Mobile Apps to create innovative career path and for the socially relevant issues.

CO PO MAPPING

Note: H-Highly correlated=3, M-Medium correlated=2, L-Less correlated=1

CO'S	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
C205.1	3	3	3	2	2							2
C205.2	3	3	3	2	3							2
C205.3	3	3	3	2	3							2
C205.4	3	3	3	2	2							2
C205.5	3	3		2	3							2
C205.6	3	2	3	2	2							2
C205	3	2.83	3	2	2.5							2

CO PSO MAPPING

CO'S	PSO1	PSO2	PSO3
C205.1	3	3	2
C205.2	2	3	3
C205.3		3	2
C205.4		3	
C205.5	3		
C205.6		3	
C205	2.67	3	2.33

APPENDIX 1

CONTENT BEYOND THE SYLLABUS

MODULE NOTES & QUESTION BANK

Course code	Course Name	L-T-P-Credits	Year of Introduction
CS205	Data Structures	3-1-0-4	2016

Pre-requisite: B101-05 Introduction to Computing and Problem Solving

Course Objectives

1. To impart a thorough understanding of linear data structures such as stacks, queues and their applications.
2. To impart a thorough understanding of non-linear data structures such as trees, graphs and their applications.
3. To impart familiarity with various sorting, searching and hashing techniques and their performance comparison.
4. To impart a basic understanding of memory management.

Syllabus

Introduction to various programming methodologies, terminologies and basics of algorithms analysis, Basic Abstract and Concrete Linear Data Structures, Non-linear Data Structures, Memory Management, Sorting Algorithms, Searching Algorithms, Hashing.

Expected Outcome:

Students will be able to

1. compare different programming methodologies and define asymptotic notations to analyze performance of algorithms.
2. use appropriate data structures like arrays, linked list, stacks and queues to solve real world problems efficiently.
3. represent and manipulate data using nonlinear data structures like trees and graphs to design algorithms for various applications.
4. illustrate and compare various techniques for searching and sorting.
5. appreciate different memory management techniques and their significance.
6. illustrate various hashing techniques.

Text Books:

1. Samanta D., Classic Data Structures, Prentice Hall India, 2/e, 2009.
2. Richard F. Gilberg, Behrouz A. Forouzan, Data Structures: A Pseudocode Approach with C, 2/e, Cengage Learning, 2005.

References

1. Horwitz E., S. Sahni and S. Anderson, Fundamentals of Data Structures in C, University Press (India), 2008.
2. Aho A. V., J. E. Hopcroft and J. D. Ullman, Data Structures and Algorithms, Pearson Publication,1983.
3. Tremblay J. P. and P. G. Sorenson, Introduction to Data Structures with Applications, Tata McGraw Hill, 1995.
4. Peter Brass, Advanced Data Structures, Cambridge University Press, 2008
5. Lipschuts S., Theory and Problems of Data Structures, Schaum's Series, 1986.
6. Wirth N., Algorithms + Data Structures = Programs, Prentice Hall, 2004.
7. Hugges J. K. and J. I. Michtm, A Structured Approach to Programming, PHI, 1987.
8. Martin Barrett, Clifford Wagner, And Unix: Tools For Software Design, John Wiley, 2008 reprint.

COURSE PLAN			
Module	**Contents**	**Hours (56)**	**Sem. Exam Marks**
I	Introduction to programming methodologies – structured approach, stepwise refinement techniques, programming style, documentation – analysis of algorithms: frequency count, definition of Big O notation, asymptotic analysis of simple algorithms. Recursive and iterative algorithms.	**9**	**15%**
II	Abstract and Concrete Data Structures- Basic data structures – vectors and arrays. Applications, Linked lists:- singly linked list, doubly linked list, Circular linked list, operations on linked list, linked list with header nodes, applications of linked list: polynomials,.	**9**	**15%**
III	Applications of linked list (continued): Memory management, memory allocation and de-allocation. First-fit, best-fit and worst-fit allocation schemes Implementation of Stacks and Queues using arrays and linked list, DEQUEUE (double ended queue). Multiple Stacks and Queues, Applications.	**9**	**15%**
IV	String: - representation of strings, concatenation, substring searching and deletion. Trees: - m-ary Tree, Binary Trees – level and height of the tree, complete-binary tree representation using array, tree traversals (Recursive and non-recursive), applications. Binary search tree – creation, insertion and deletion and search operations, applications.	**10**	**15%**
V	Graphs – representation of graphs, BFS and DFS (analysis not required) applications. Sorting techniques – *Bubble sort, Selection Sort,* Insertion sort, Merge sort, Quick sort, Heaps and Heap sort. Searching algorithms (Performance comparison expected. Detailed analysis not required)	**09**	**20%**
VI	Linear and Binary search. (Performance comparison expected. Detailed analysis not required) Hash Tables – Hashing functions – Mid square, division, folding, digit analysis, collusion resolution and Overflow handling techniques.	**10**	**20%**

Question Paper Pattern:

1. There will be *five* parts in the question paper – A, B, C, D, E
2. Part A
    a. Total marks : 12
    b. *Four* questions each having *3* marks, uniformly covering module I and II; All *four* questions have to be answered.
3. Part B
    a. Total marks : 18
    b. *Three* questions each having *9* marks, uniformly covering module I and II; T*wo* questions have to be answered. Each question can have a maximum of three subparts
4. Part C
    a. Total marks : 12
    b. *Four* questions each having *3* marks, uniformly covering module III and IV; All *four* questions have to be answered.
5. Part D
    a. Total marks : 18
    b. *Three* questions each having *9* marks, uniformly covering module III and IV; T*wo* questions have to be answered. Each question can have a maximum of three subparts
6. Part E
    a. Total Marks: 40
    b. *Six* questions each carrying 10 marks, uniformly covering modules V and VI; *four* questions have to be answered.
    c. A question can have a maximum of three sub-parts.
7. There should be at least 60% analytical/numerical/design questions.

QUESTION BANK

MODULE I			
Q:NO:	**QUESTIONS**	**CO**	**KL**
1	Describe complexity of an algorithm? Write worst case and best case complexity of linear search.	CO1	K2
2	Define the terms	CO1	K3
3	a) Frequency count.	CO1	K2
4	b) Stepwise refinement technique.	CO1	K3
5	Describe the different notations used to describe the asymptotic running time of an algorithm.	CO1	K5
6	Explain stepwise refinement techniques.	CO1	K2
7	Explain performance analysis measurements.	CO1	K5
8	Define Big O notation. Show that $4n^2=O(n^3)$	CO1	K2
9	Explain the criteria that you will keep in mind while choosing an appropriate algorithm to solve a particular problem.	CO1	K2
10	Explain different programming methodologies**.**	CO1	K4
11	Write an algorithm to find the position of smallest number in an array.	CO1	K2
12	Explain the criteria that you will keep in mind while choosing an appropriate algorithm to solve a particular problem.	CO1	K2
13	Explain different programming methodologies**.**	CO1	K5
14	Write an algorithm to find the position of smallest number in an array.	CO1	K5
MODULE II			
1	Explain Vectors and arrays.	CO2	K2
2	How a linked list can be used to represent a polynomial $5x^3+4x^2+3x+2$? Give an algorithm to perform addition of two polynomials using linked list.	CO2	K4
3	Explain Vectors and arrays.	CO2	K2
4	Write an algorithm to find the position of smallest number in an array.	CO2	K5

5	Write an algorithm to delete a node at first position in doubly linked list.	CO2	K5
6	Compare the approaches for designing an algorithm	CO2	K3
7	Compare a linked list with an array.	CO2	K3
8	Write short notes on Modular programming and structured programming	CO2	K4
9	Explain the control structures used in algorithms.	CO2	K2
10	Compare the approaches for designing an algorithm	CO2	K3
11	Explain the difference between a circular linked list and a singly linked list.Give the advantages and uses of a circular linked list.	CO2	K2
12	Write an algorithm to insert a node at first position in doubly linked list.	CO2	K5
MODULE III			
1	Define stack. List the operations can be performed in a stack.	CO3	K3
2	Given five memory partitions of 100Kb, 500Kb, 200Kb, 300Kb, 600Kb (in order), how would the first-fit and best-fit algorithms place processes of 212 Kb, 417 Kb, 112 Kb, and 426 Kb (in order)? Which algorithm makes the most efficient use of memory?	CO3	K3
3	Write an algorithm to insert and delete an element at rear end in DEQUEUE	CO3	K2
4	Write a program to implement multiple stacks.	CO3	K3
5	List the Applications of Stack	CO3	K5
6	Describe Memory management with bitmaps.	CO3	K3
7	Free memory blocks of size 60K, 25K, 12K, 20K, 35K, 45K and 40K are available in this order. Show the memory allocation for a sequence of job requests of size 22K, 10K, 42K, and 31K (in this order) in First Fit, Best Fit and Worst Fit allocation strategies.	CO3	K2
8	Explain how a stack can be implemented using linked list	CO3	K5
9	Define circular queue. Explain how it is different from normal queue.	CO3	K5
10	Define Stack Frames. Give example.	CO3	K2

11	Discuss the basic features of Queue.	CO3	K5
12	Write an algorithm to perform linked list implementation of Queue.	CO3	K2
13	Write an algorithm to convert an infix expression to postfix.	CO3	K1
14	Write an algorithm for evaluating a postfix expression and evaluate the following postfix expression using the algorithm AB+CD/AD-EA^+*where A=2, B=7, C=9, D=3, E=5	CO3	K2
	MODULE IV		
1	Write an algorithm to perform concatenation of two strings.	CO4	K2
2	List the properties of binary search tree. Write an algorithm to search an element from a binary search tree.	CO4	K1
3	Define tree traversal. List different ways to traverse a tree.	CO4	K2
4	Show the structure of the binary search tree after adding each of the following values in that order: 10, 1, 3, 5, 15, 12, 16. What is the height of the created binary search tree?	CO4	K3
5	Define Binary Search Tree. Develop an algorithm to add an element into a binary search tree.	CO4	K1
6	Compare the approaches for traversing a tree.	CO4	K2
7	Explain how we can initialize strings using arrays.	CO4	K3
8	Write a recursive algorithm to perform preorder traversal. Explain with example.	CO4	K3
9	Write an algorithm to insert an element in a binary search tree. Explain with example.	CO4	K2
10	Develop an algorithm to add an element into a binary search tree.	CO4	K5
11	What are the applications of trees?	CO4	K3
	MODULE V		
1	Write an algorithm/ C program to perform merge sort. Given the following list of numbers: [21, 1, 26, 45, 29, 28, 2] find the output obtained after each recursive call of merge sort algorithm.	CO5	K4

2	Write C program/algorithm to perform linear search. Find the time complexity for best, worst and average case for a linear search in an array of n elements.	CO5	K2
3	Write algorithm to perform Breadth First Search.Write one possible order of visiting the nodes of the following graph starting at vertex A.	CO5	K3
4	Give any two representations of graph. Give algorithm for DFS. Demonstrate DFS using suitable example.	CO5	K2
5	List the properties of binary search tree. Write an algorithm to search an element from a binary search tree.	CO5	K3
6	Write the non recursive preorder traversal algorithm.	CO5	K3
7	List the properties of binary search tree. Write an algorithm to search an element from a	CO5	K3
MODULE VI			
1	Give an algorithm to perform binary search. Using the algorithm, search for elements 23 and 47 in the given set of elements[12 23 27 35 39 42 50].	CO6	K3
2	What is max heap?Write an algorithm to perform heap sort. Give example.	CO6	K3
3	Write C program/algorithm to perform selection sort. Perform selection sort on an array [5,3,1,7,9].	CO6	K3
4	Write algorithm for (i) Insertion sort (ii) Bubble sort [30,20,10,60,70,40]	CO6	K3
5	Define hashing. What are the properties of a good hash function? With necessary examples explain four different hashing techniques.	CO6	K2
6	Write an algorithm for merge sort technique. Illustrate with an example. Give its complexity.	CO6	K4
7	Define collision. What is linear probing? The following keys 10, 16, 11, 1, 3, 4,23 and 15	CO6	K4

MODULE I

Page No: 1
Date:

MODULE I

> Introduction to programming methodologies
  - Structured Approach
  - stepwise refinement techniques
  - programming style & documentation
> Analysis of Algorithms
  - frequency count
  - Big O notation
  - asymptotic analysis of simple algms
> Recursive & iterative algorithms.

Question Bank:

1. Write an algm for swapping 2 values
2. Write an algm to find the larger of 2 nos
3. Write an algm to find whether a no is even or odd.
4. Write an algm to print the grade obtained by a student.
5. Write an algm to find sum of first N natural nos.
6. Show that $n = O(n \log n)$
7. Show that $10n^3 + 20n \neq O(n^2)$
8. Show that $n^2 + n = O(n^3)$

Page No:
Date:

>Introduction to programming Methodologies:

Programming methodology deals with the analysis, design & implementation of programs.

Algorithm:

Algorithm is a step by step finite sequence of instruction, to solve a well defined computational problem.

-To solve any complex real life problems, first define the problems & ~~ser~~ second design the algm to solve the problems. Writing & executing pgms & then optimizing them may be effective for small pgms. But for large pgm, each part of the pgm must be well organized before writing the pgm. There are few steps of refinement involved when a pblm is converted to pgm; this method is called stepwise refinement method. There are 2 approaches for algorithm design;

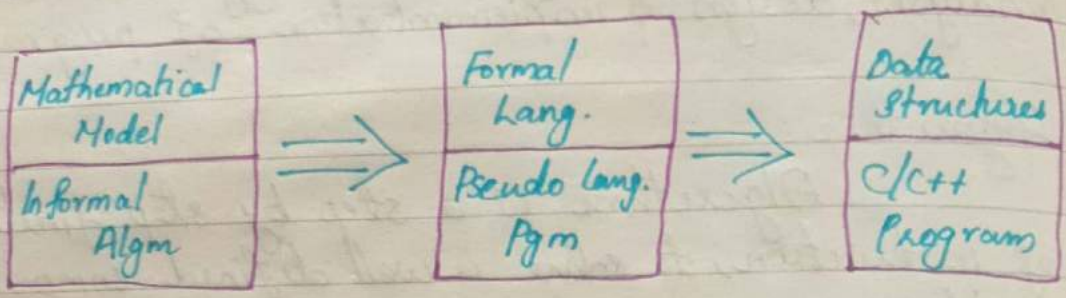
they are 1. top_down
2. bottom-up

Stepwise Refinement Techniques:

We can write an informal algm, if we have an appropriate mathematical model for a pblm. The initial version of the algm will contain general statements, i.e; informal instructions. Then convert this informal algm to formal algm, i.e; more definite inst's by applying any pgmming lang. syntax & semantics partially. Finally a pgm can be developed by converting the formal algm by a pgmming. lang. manual. There are several steps to reach a pgm from a

mathematical model. In every step there is a refinement (or conversion).

There are 3 steps in refinement process,



1. In the first stage, modelling, represent the pblm using an appropriate mathematical model such as a graph, tree etc. At this stage, the sol^n to the pblm is an algm expressed very informally.
2. The algm is written in pseudo-lang. (formal algm) i.e; mixture of any pgmming lang. constructs & less formal English stmts. The op^{n}s to be performed on the various types of data become fixed.
3. In the final stage, choose an implementation for each abstract data type & write the procedures for the various op^{n}s on that type. The remaining informal stmts in the pseudo-lang. algm are replaced by C/C++ code.

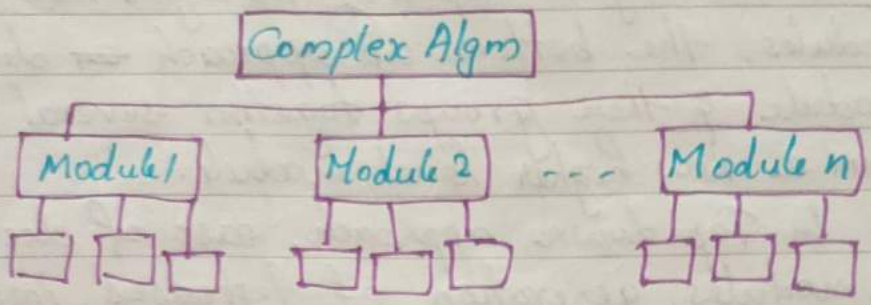
Different Approaches to design an Algorithm:

Algorithms are used to manipulate the data contained in data structures. A complex algm is often divided into smaller units called modules. This process of dividing an algm into modules is called modularization. The key advs:

1. It makes the complex algm simpler to design & implement.
2. Each module can be designed independently

There are 2 main approaches to design an algm -

1. top-down approach
2. bottom-up approach



Top-down Approach:

A top-down design approach starts by deviding the complex algm into one or more modules. These modules can be decomposed into one or more sub-modules & this process of decomposition is iterated until the desired level of module complexity is achieved. Top down design method is a form of step wise refinement where we begin with the topmost module & incrementally add modules that it calls.

Bottom-up Approach:

is just the reverse of top-down approach. In the bottom up design, we start with designing the most basic or concrete modules & then proceed towards designing higher level modules. The higher level modules are implemented by using the op^{n}s performed by lower level modules. Thus, in this approach sub-modules are grouped together to form a higher level module. All the higher level modules are clubbed together to form even higher level modules. This process is repeated until the design of the complete algm is obtained.

Top-down vs Bottom-up Approach

→ while top-down approach follows a stepwise refinement by decomposing the algm into manageable modules, the bottom-up approach ~~or~~ defines a module & then groups together several modules to form a new higher level module.

→ In Top down approach, ease of documenting the modules, generation of testcases, implementation of code & debugging.

→ Bottom-up approach allows inf° hiding

> Programming Style & Documentation

Different programming methodologies:

1. Procedural Programming
2. Modular "
3. Structured "
4. Object Oriented "

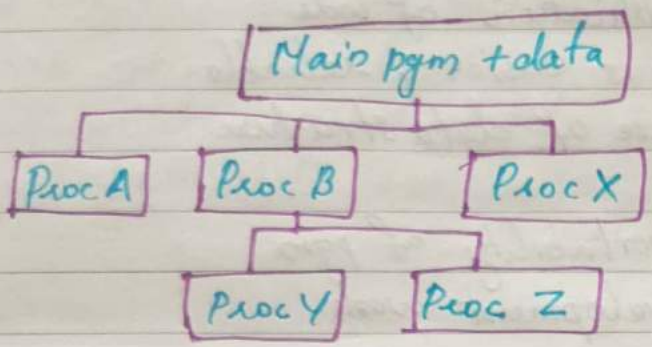
Procedural Programming:

—based on the concept of using procedures. Procedure is a sequence of commands to be executed. Any procedure can be called from any point within the general program, including other procedures or even itself.

Benefits:

1. Reusability of codes
2. ease of following the logic of pgm
3. Maintainability of code
4. Most of the fns share global data.

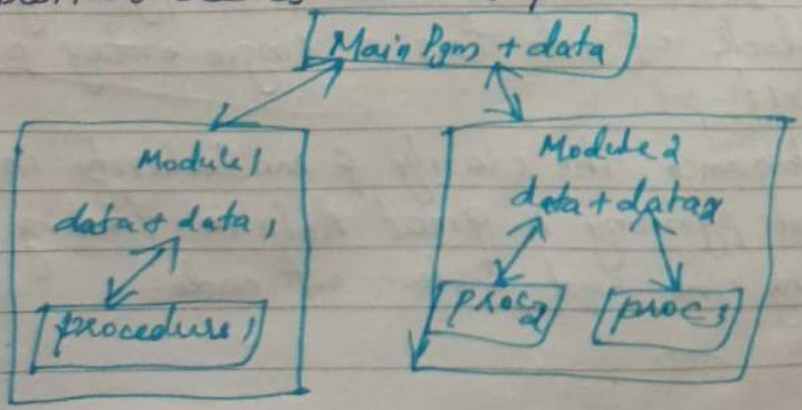
5. Data move openly around the s/m from fn to fn.
6. Fns. transform data from one form to another.
Each step of computation is described explicitly, even if by the means of defining procedures.



Modular Programming

The pgm is progressively decomposed into smaller partition called modules. The pgm can be written in modular form, thus an overall ~~pgm~~ pblm to be decomposed into a seq. of individual sub pgms.
A module decomposed into successively sub modules.
The following are the steps needed to develop a modular pgm.

1. Define the pblm
2. Outline the steps to achieve the goal
3. Decompose the pblm into subtasks.
4. Prototype a subpgm for each sub task.
5. Repeat steps 3 & 4 for each subpgm until further decomposition seems counter productive.



Two methods for modular pgmming
1. top-down 2. bottom-up

Adv: of Modular programming:
1. Reduce the complexity of the entire pblm
2. Avoid the duplication of code
3. debugging pgm is easier & reliable.
4. hides the use of data structure
5. Recusability
6. Improves the portability of pgm
7. reduces development work.

Structured Programming:

It uses following types of code structures to write pgms:
1. Seq. of sequentially executed stmts
2. Conditional ex^n of stmts (if stmts)
3. Looping or iteration (ex; for, do..while, ~~gathlestmts~~)
4. Structured subroutine calls (fns).

The follo. lang. usage is forbidden
- ~~got~~ go to stmts - Break or continue
- multiple exit pts. - multiple entry pts.

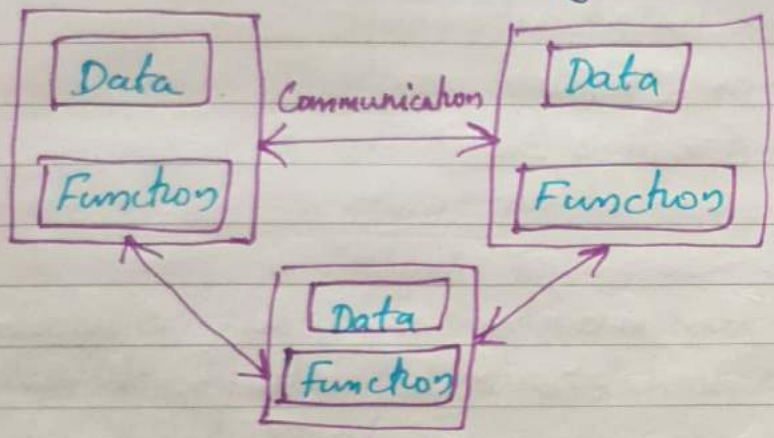
Adv: of Structured Programming:

* clarity: clarity & logical pattern to their ctrl structure & due to this increase in pgmming pdtivity.
* Each block of code has a single entry pt & single exit pt.
* Maintenance: the clarity & modularity inherent in structured pgmming is great help in finding error & redesigning the rqd section of code.

Page No :
Date :

Object Oriented Programming:

OOP treats data as a critical elmt in the pgm devpt & doesn't allow it to flow freely around the s/m. It ties data more closely to the fn that operate on it. & protects it from accidental modification from outside fn. OOP allows decomposition of a pblm into a no. of entities called objects & then builds data & fn around these objts. ~~The orgn of data & fn~~



Features:

→ Emphasis on data rather than procedure
→ Pgms are devided into objts.
→ Fns that operate on the data of an objt ~~data~~ ties together in the data structure.
→ Data is hidden & cannot be accessed by ext. fn.
→ Objts may communicate with each other through fn
→ New data & fns can be easily added whenever necessary
→ Follows bottom up approach in pgm design.

Page No :
Date :

Analysis of Algorithms

Control Structures used in Algorithms:
- Sequence
- Decision
- Repetition.

Sequence: Each step of an algm is executed in a specified order. Eg: Algm to add 2 no-s.

Step 1: Input first no. as A
step 2: Input 2nd no. as B
step 3: set sum = A+B
step 4: print Sum
step 5: End.

Decision: Decision stmts are used when the ex" of a process depends on the outcome of some condition. for eg:, if x=y, then print equal.

IF condition then process

also be in the follo. manner

IF condition
Then process1
Else process 2.
} IF-ELSE construct.

Eg: step 1: Input first Numbers as A
step 2: Input second Number as B
step 3: IF A = B
print "Equal"
ELSE
print "not equal"
Step 4: END.

Repetition

- executing one or more steps for a no. of times, can be implemented using constructs such as while, do-while & for loops. These loops execute one or more steps until some condition is true.

Eg: Algm to print first 10 natural no.s.

step1: [initialize] set I=1, N=10
step2: Repeat step3 & 4 while I<=N
step3: print I
step4: set I = I+1
[END of Loop]
step5: END

After designing an algm, it has to be checked & its correctness needs to be predicted; this is done by analyzing the algm. The algm can be analyzed by tracing all step by step inst's, reading the algm for logical correctness, & testing it on some data using mathematical techniques to prove it correct. Another type of analysis is to analyze the simplicity of the algm. The choice of a particular algm depends on follo. performance analysis & measurements:

1. Space Complexity
2. Time Complexity.

Analyzing an algm means determining the amount of resources (time & memory) needed to execute it. Algms are generally designed to work with an arbitrary no. of i/ps, so the efficiency or complexity of an algm is stated in terms of time & space complexity.

Page No :
Date :

Space Complexity:

Space complexity of an algm is the amnt of comp. memory that is rqd during the pgm exn as a fn of the i/p size.

The space needed by a pgm depends on the follo. 2 parts:

* Fixed part: It includes the space needed for storing inst^n's, consts., variables & structured variables.
* Variable part: It ~~varies~~ includes the space needed for recursion stack & for structured variables that are allocated space dynamically during the runtime of a pgm.
* Environment stack space: This space is needed to store the inf^n to resume the suspended (partially completed) functions. Each time a fn is invoked the follo. data is saved on the envt. stack.

(a) Return address: i.e; from where it has to resume after completion of the called fn.

(b) Values of all lead variables & the values of formal parameters in the fn being invoked.

Time Complexity:

Time Complexity of an algm is basically the running time of a pgm as a fn of the i/p size. The no. of m/c inst^n's which a pgm executes is called its time complexity. This no. is primarily dependent on the size of the pgms's i/p & algm used.

The exact time will depend on the implementation of the algm, pgmming lang., compiler used, the CPU speed used other h/w specific's. To measure the time complexity, count all sorts of op^n's performed in an algm.

Worst-case, Average-case, Best-case & Amortized Time Complexity.

Worst-case running time: denotes the behaviour of an algm w.r.to the worst possible case of the i/p instance. The worst-case running time of an algm is an upper bound on the running time for any i/p. Having the knowledge of worst-case running time gives us an assurance that the algm will never go beyond this time limit.

Average-case running time: is an estimate of the running time for an 'average' i/p. It specifies the expected behaviour of the algm when the i/p is randomly drawn from a given destribution.

Best-case running time: The term 'best case performance' is used to analyze an algm under optimal conditions. For eg; the best case for a simple linear search on an array occurs the desired elmt is the first in the list. While developing & choosing an algm to solve a pblm, we hardly base our decision on the best-case performance.

Amortized running time: refers to the time required to perform sequence of (related) op^{n}s averaged over all the op^{n}s performed. It guearantees avg performance of each op^n in the worst case.

Time-Space Trade off

The best algm to solve a particular pblm is one that requires less memory space & takes less time to complete its ex^n. There can be more than one algm to solve a particular pblm. One may require less mem. space, while the other

may require less CPU time to execute. There exists a time-space trade-off among algms.

If space is a big constraint, then one might choose a pgm that takes less space at the cost of more CPU time. If time is the major constraint, then one might choose a pgm that takes minimum time to execute at the cost of more space.

Expressing Time & Space Complexity:

The time & space complexity can be expressed using a fn $f(n)$ where n is the i/p size for a given instance of the pblm being solved. Expressing the complexity is reqd when

* we want to predict the rate of growth of complexity as the i/p size of the pblm increases.
* There are multiple algms that find a sol^n to a given pblm & we need to find the algm that is most efficient.

Frequency Count

Frequency Count method can be used to analyze a pgm. Here assume that every stmt takes the same constant amount of time for its ex^n. Hence the determination of time complexity of a given pgm is the matter of summing the frequency counts of all the stmts of that pgm.

Consider the follow. examples:

```
....          for(i=0; i<n; i++)        for(i=0; i<n; i++)
....          {                         for(j=0; j<n; j++)
x++;          x++;                      x++;
}             ....
(a)           (b)                       (c)
```

In the eg (a) the stmt x++ is not contained within any loop either explicit or implicit. Then its freq. count is just one.
In the eg (b) same stmt will be executed n times.
In the eg (c) it is executed by n^2.

Growth of Functions & Asymptotic Notation

→ characterizing algms, acc. to their efficiency.

→ in the order of growth of the running time of an algm, not in the exact running time. This is referred to as the asymptotic running time. Asymptotic notation gives us a method for classifying fns acc. to their rate of growth.

Big O Notation

The no. of statements executed in the pgm for n elmts of the data is a fn of the no. of elmts, expressed as f(n). Even if the exp^n derived for a fn is complex, a dominant factor in the exp. is sufficient to determine the order of magnitude of the result & hence the efficiency of the algm. This factor is the Big O & is expressed as O(n).

'O' stands for 'order of', is concerned with what happens for very large values of n. For eg: if a sorting algm performs n^2 op^ns to sort just n elmts then that algm would be described as an $O(n^2)$ algm.

In the eg (a) the elmt x++ is not contained within any loop either explicit or implicit. Then its freq count is just one.
In the eg (b) same elmt will be executed n times.
In the eg (c) it is executed by n^2.

Growth of functions & Asymptotic Notation

⇒ characterizing algms, acc. to their efficiency.

⇒ in the order of growth of the running time of an algm, not in the exact running time. This is referred to as the asymptotic running time. Asymptotic notation gives us a method for classifying fns acc. to their rate of growth.

Big O Notation

The no. of statements executed in the pgm for n elmts of the data is a fn of the no. of elmts, expressed as f(n). Even if the exact expn derived for a fn is complex, a dominant factor in the exp. is sufficient to determine the order of magnitude of the result & hence the efficiency of the algm. This factor is the Big O & is expressed as O(n).

'O' stands for 'order of', is concerned with what happens for very large values of n. For eg: if a sorting algm performs n^2 oprs to sort just n elmts then that algm would be described as an $O(n^2)$ algm.

When expressing complexity using the Big O notation, constant multipliers are ignored. So an $O(4n)$ is eqt. to $O(n)$;
If $f(n)$ & $g(n)$ are the fns defined on a +ve integer no. n, then $f(n) = O(g(n))$

— Best case O describes an upper bound for all combinations of i/p. It is possibly lower than the worst case. For eg:, when sorting an array the best case is when the array is already sorted.

— Worst case Ω describes a lower bound for worst case input combinations. It is possibly greater than the best case. For eg; when sorting an array the worst case is when the array is sorted in reverse order.

$g(n)$	$f(n) = O(g(n))$
10	$O(1)$
$2n^3+1$	$O(n^3)$
$3n^2+5$	$O(n^2)$
$2n^3+3n^2+5n-10$	$O(n^3)$

n	$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^3)$
1	1	1	1	1	1	1
2	1	1	2	2	4	8
4	1	2	4	8	16	64
8	1	3	8	24	64	512
16	1	4	16	64	256	4096

Eg1. Show that $4n^2 = O(n^3)$.

By def^n $0 \leq h(n) \leq cg(n)$

Substituting $4n^2$ as $h(n)$ & n^3 as $g(n)$, we get

$$0 \leq 4n^2 \leq cn^3$$

÷ing by n^3 $\quad 0/n^3 \leq \frac{4n^2}{n^3} \leq cn^3/n^3$

Now to determine the value of c, we see that $4/n$ is max. when $n=1$. $\therefore c=4$

To determine the value of n_0,

$0 \leq 4/n_0 \leq 4$

$0 \leq 4/4 \leq n_0$

$0 \leq 1 \leq n_0$

Thus means $n_0=1$. Therefore, $0 \leq 4n^2 \leq 4n^3 \; \forall \; n \geq n_0=1$.

Eg 2. Show that $400n^3 + 20n^2 = O(n^3)$

Solution: By definition, we have

$0 \leq h(n) \leq cg(n)$

substituting $400n^3 + 20n^2$ as $h(n)$ and n^3 as $g(n)$

$0 \leq 400n^3 + 20n^2 \leq cn^3$

Dividing by n^3

$0/n^3 \leq 400n^3/n^3 + 20n^2/n^3 \leq cn^3/n^3$

$0 \leq 400 + 20/n \leq c$

Note that $20/n \to 0$ as $n \to \infty$ & $20/n$ is max. when $n=1$

$\therefore 0 \leq 400 + 20/1 \leq c$

Thus means $c = 420$

To determine value of n_0

$0 \leq 400 + 20/n_0 \leq 420$

~~$-400 \leq 400 + 20/n_0 - 400 \leq 420 - 400$~~

This implies $n_0 = 1$

Hence $0 \leq 400n^3 + 20n^2 \leq 420n^3 \; \forall \; n \geq n_0 = 1$

MODULE II

Applications

→ used to implement mathematical vectors, matrices & rectangular tables.
→ Many databases include one-dimensional arrays whose elmts are records.
→ used to implement other data structures, such as strings, stacks, queues, heaps & hash tables.
→ used for sorting elmts in asce. or desc. order.

LINKED LIST

If the memory is allocated before the exn of the pgm, it is fixed & cannot be changed. There is a special data structure called linked list that provides a more flexible storage s/m & it doesn't require the use of array.

— special list of some data elmts linked to one another. The logical ordering is represented by having each elmt pointing to the next elmt. Each elmt is called a node, which has 2 parts, INFO part which stores the infn & LINK which points to the next elmt.

Advantages:

- ~~Lis~~ are dynamic data structure: grow or shrink during the exn of a pgm.
- Efficient mem. utilization: mem. is not pre-allocated. Mem. is allocated whenever it is required. And when its is no longer needed.
- Insertion & deletion are easier & effiwent.
- Many complex appl"s can be easily carried out with linked lists.

Disadvantages:

* More memory
* Access to an arbitrary data item is difficult & time consuming.

Types of Linked Lists:

- Singly linked list
- Doubly ,,
- Circular ,,

Basic Op^{n}s:

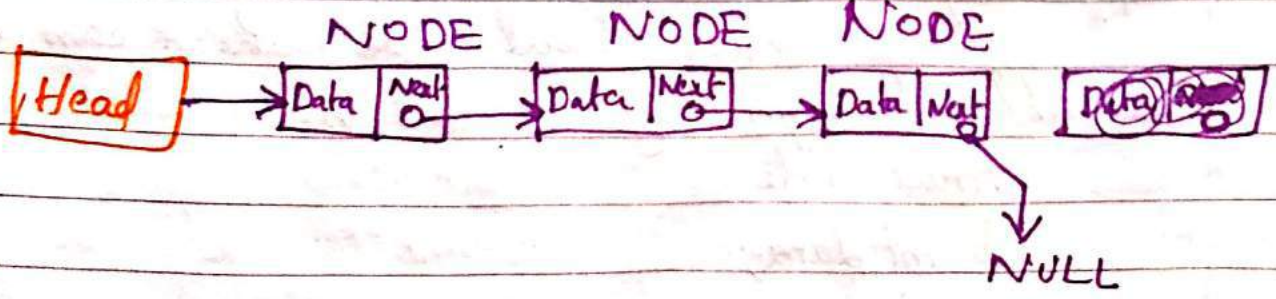
- Insertion - deletion - Search - Delete.

1. Singly Linked List:

is a non sequential coll^n of data items called nodes. Each node has 2 fields.

1. DATA field
2. NEXT field

The data field contains an actual value to be stored & processed. NEXT field contains addr. of the next data item in the linked list. The address used to access a particular node is known as a pointer. The link field of the last node contains NULL rather than a valid address. External pointer (HEAD) is a ptr to the very first node in the linked list, it enables to access the entire linked list.



Basic Operations:

* Insertion - A new node may be inserted.
  - > At the begining of the linked list.
  - > At the end of the linked list
  - > At the specific position of the linked list
* Deletion - delete an elmt from
  - > Begining of the linked list
  - > End of the linked list
  - > Specific postion of the linked list.
* Display - to print each & every node's info.
* Traversing - process of going through all the nodes of a linked list from one end to the other end.

Steps to create a linked list:

Step 1: Include alloc.h header file

```
#include <alloc.h>
```

- allocate mem. using dynamic Mem. alloc^n functions malloc.
- Dynamic memory allocation fns are included in alloc.h.

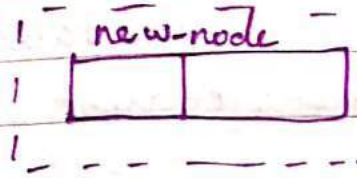
Step 2: Define Node structure

- new global node which can be accessible through any of the fn.:

```
struct node
{ int data;
  struct node *next;
} *start = NULL;
```

Step 3: Create Node using Dynamic Memory Allocation.
- creating node dynamically using malloc fn.

```
new-node = (struct node *) malloc (sizeof (struct node));
```

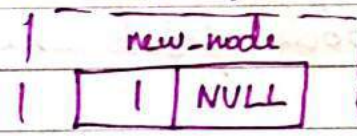


created New Node using Dynamic memory allocation.

Step 4: Fill Information in newly Created Node
- accept value from the user using scanf.

Accepted value is stored in the data field. Make its Next field as NULL.

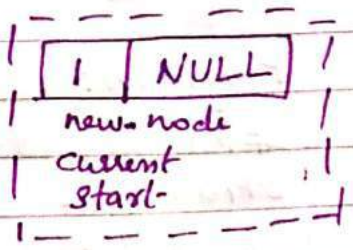
```
printf ("Efs data");
scanf ("%d", &new_node->data);
new_node -> next = NULL;
```



fill Node with the data provided by user.

Step 5: Creating Very First Node
~~If node~~ - assign it as starting node. If start is equal to null then we can identify node as first node.

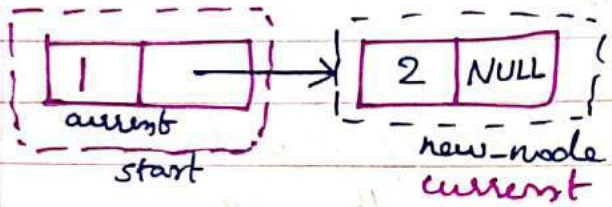
```
if start = NULL
    new_node, current, start
if (start == NULL)
  { start = new_node;
    curr = new_node;
  }
```



Step 6: Creating Second or nth node
- First node can be referred as "new_node", "curr", "start"

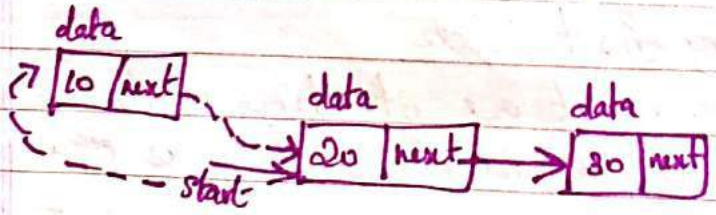
- create() fn again.

else
{ current → next = new_node; // link b/w new_node & current node
current = new_node; // move current ptr. to next node.
}



Insert node at start/first pos^n in Singly Linked list

1. Allocate a memory for the new node
2. Insert Data into the "Data Field" of new_node
3. If(start = NULL) list is empty then go to step 4 otherwise goto step 5.
4. start point to the new_node & set linked field of new_node to NULL.
5. Linked field of the new_node pointed to start, then set start to new_node.



Programm:

```
#include<stdio.h>
struct node
{ int data;
  struct node *ptr;
};
int main()
{ typedef struct node NODE;
  NODE *start=NULL, *temp;
  int item;
  printf("Single Linked List");
  temp=(NODE *)malloc((sizeof(NODE)));
```

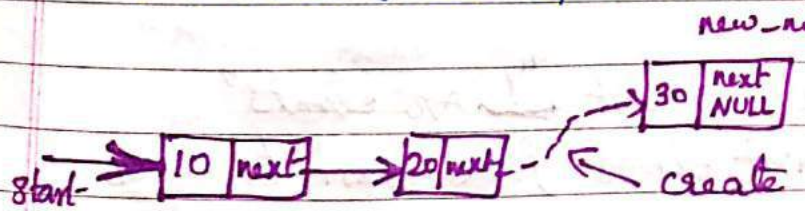
```
printf("Ets the data");
scanf("%d", &temp->data);
if(start==NULL)
 {
   temp->ptr=NULL;
   start=temp;
 }
else
{ temp->ptr = start;
  start = temp;
}
```

Last

Insert node at start/first position in Singly Linked List.

Algorithm:

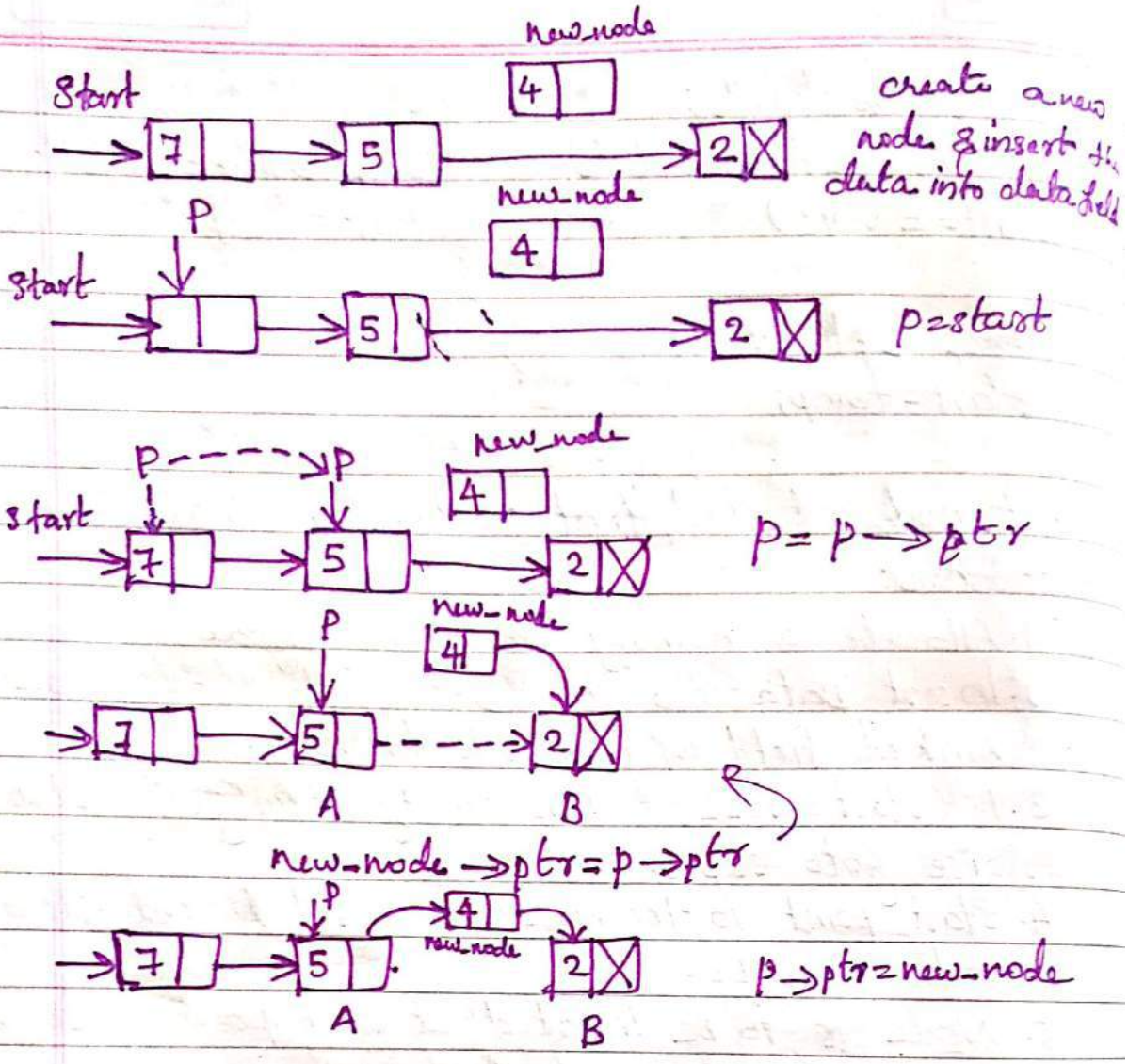
1. Allocate a memory for the new-node
2. Insert Data into the "Data field" of new_node & set "Linked field of new_node to NULL.
3. If (start = NULL) list is empty then go to step 4 otherwise goto step 5.
4. Start point to the new-node & set linked field of new-node to NULL.
5. Node is to be inserted at Last position so we need to traverse SLL upto last Node.
6. Linked field of last node pointed to the new node.



Insert node at Particular Position in Singly Linked List:

Algorithm:

1. Allocate memory for the new node.
2. Assign value to the data field of the new node.
3. Find the pos^n to insert
4. Suppose new node insert b/w node A & B, make the link field of the new node point to the node B & make the link field of node A point to the new node.



Display the elmts:

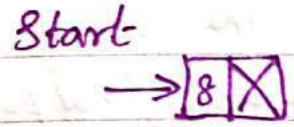
1. If start = NULL, print "Empty No elmts"
2. Otherwise print elmts from start to NULL

```
if(start == NULL)
  printf("Empty");
else
{ printf("Elmts are");
  for(p = start; p != NULL; p = p->ptr)
    printf("%d", p->data);
}
```

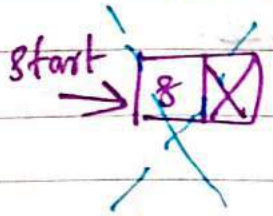
Delete node at Start/first Position in Singly Linked List

Algorithm

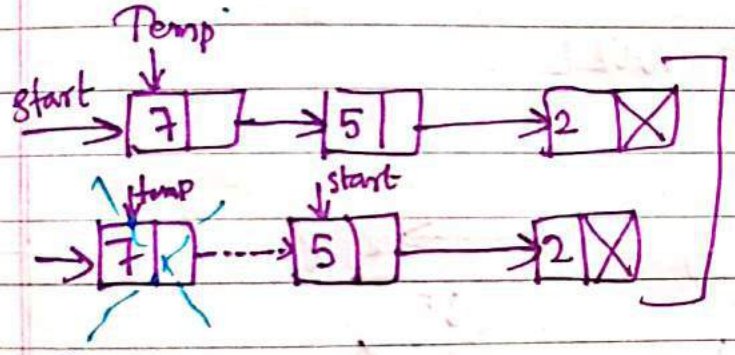
1. If list is empty, deletion is not possible.
2. If the list contain only one elmt, set external ptr to NULL. Otherwise move the ext- ptr point to the second node & delete the 1st node.



start->ptr=NULL // only one elmt.



start=NULL.



More than one elmt.

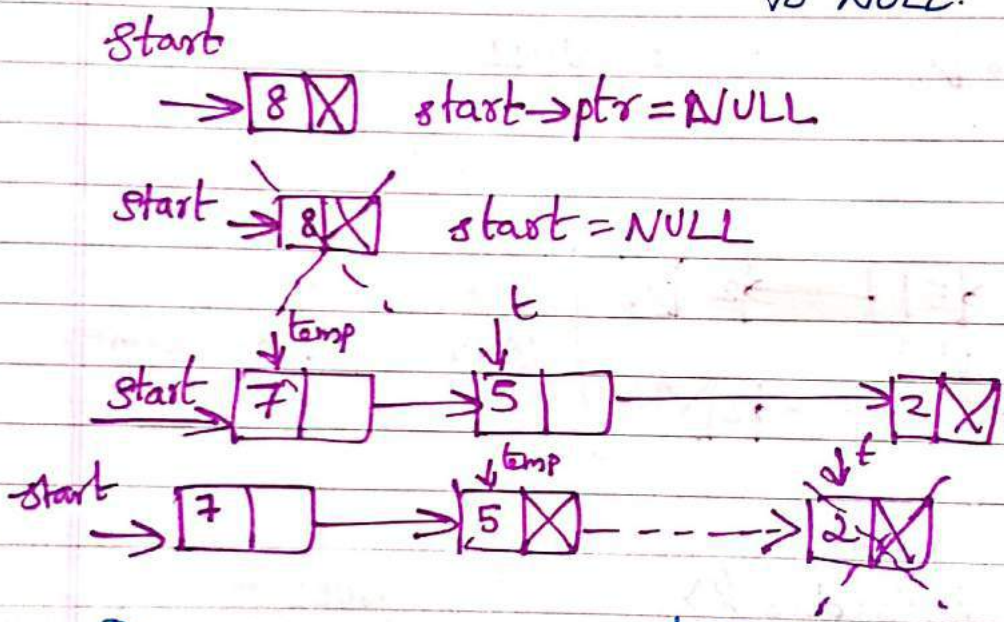
Programs:

```
#include <stdio.h>
 struct node
 { int data;
   struct node *ptr;
 };
void main()
{typedef struct node NODE;
NODE *start=NULL, *temp;
int item;
if (start==NULL)
   printf("Deletion is not possible");
else if (start->ptr==NULL)
  { temp=start;
    start=NULL;
    printf("Deleted item %d ",temp->data);
    free(temp);
  }
else
  { temp=start;
    start=start->ptr;
    printf("Deleted item %d", temp->data);
    free(temp);
  }
}
```

Delete node at End/Last position

Algorithm:

1. If the list is empty, deletion is not possible.
2. If the list contain only one elmt, set external ptr to NULL
3. Otherwise go on traversing the second last node & set the link field point to NULL.



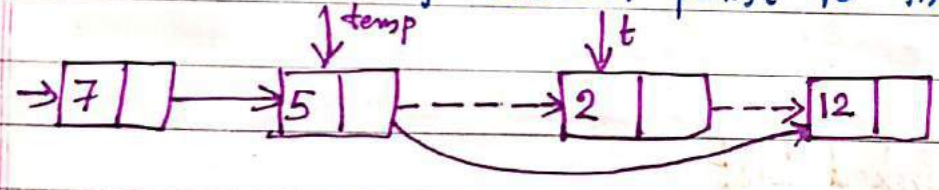
Programs:

```
#include<stdio.h>
struct node
{ int data;
  struct node *ptr;
};
void main()
{typedef struct node NODE;
NODE *start=NULL, *temp, *p, *t;
int ch, item, pos, i;
.
.
if(start==NULL)
  printf("Deletion not possible");
else if (start->ptr==NULL)
  { temp=start;
    start=NULL;
    printf("Deleted item %d",
                temp->data);
  }
else
  temp=start;
  t=start->ptr;
while(t->ptr!=NULL)
```

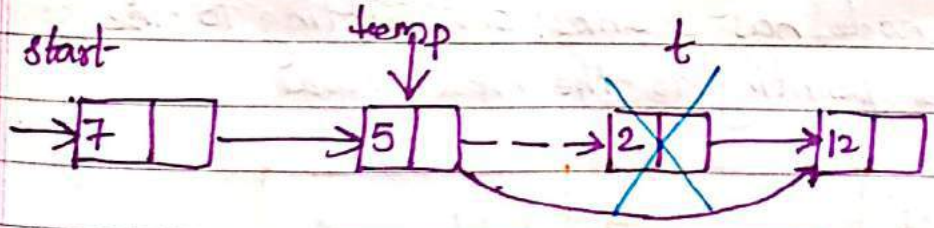
```
{ t=t->ptr;
  temp=temp->ptr;
}
temp->ptr=NULL;
printf("Deleted item %d", t->data);
free(t);
}
}
```

Delete node at particular position in Singly Linked List

1. Find the position to delete.
2. Suppose we delete node b/w node A & node B, set the link field of node A point to the node B.



temp->ptr = t->ptr



temp->ptr = t->ptr
free(t);

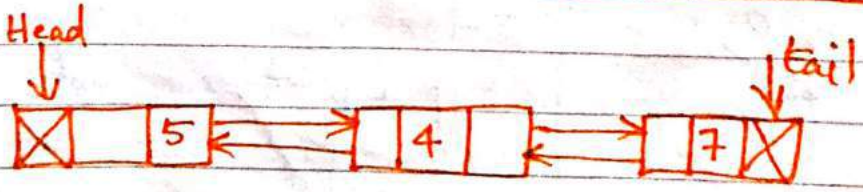
```
#include<stdio.h>
#include<stdlib.h>
struct node
{ int data;
  struct node *ptr;
};
```

```
void main()
{ typedef struct node NODE;
NODE *start=NULL, *temp, *p, *t;
int ch, item, pos, i;
 printf("Ete the position");
 scanf("%d", &pos);
 temp=start;
```

```
if (pos==1)
{
  temp=start;
  start=start->ptr;
  printf("Deleted item is %d", temp->data);
  free(temp);
}
else
  {
    for(i=1;i<pos-1;i++)
      temp=temp->ptr;
      temp->ptr=t->ptr;
      printf("Deleted item is %d", t->data);
      free(t);
  }
}
```

Doubly Linked List

— a 2 way linked list. In a doubly linked list, each node has 2 links: one pointing to the previous node & one pointing to the next node.



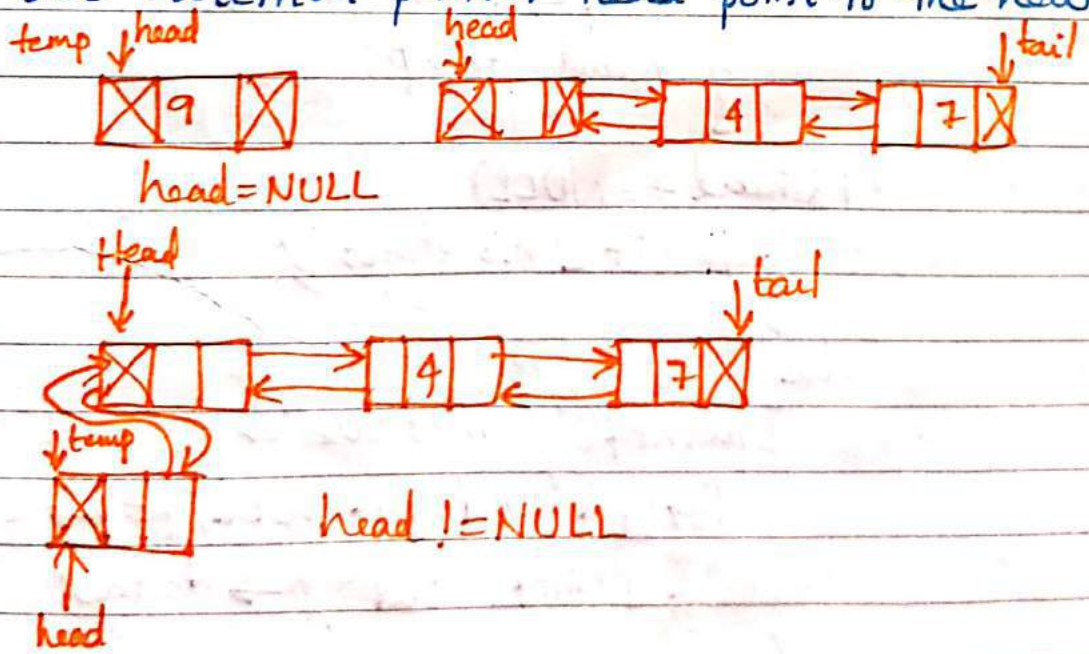
head is the external pointer, used to point starting of the doubly linked list. Tail is used to denote end of the doubly linked list.

Representation of doubly linked list

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

Insert a node at start/first Position in Doubly Linked List

1. Allocate memory for new node
2. Assign value to the data field of the new node.
3. If head = NULL then,
   set head & tail pointer point to the new node, set previous node is NULL and next node is null.
4. Otherwise
   set temp→next = head and head→prev = temp and temp→prev = NULL.
   Set external pointer head point to the new node.

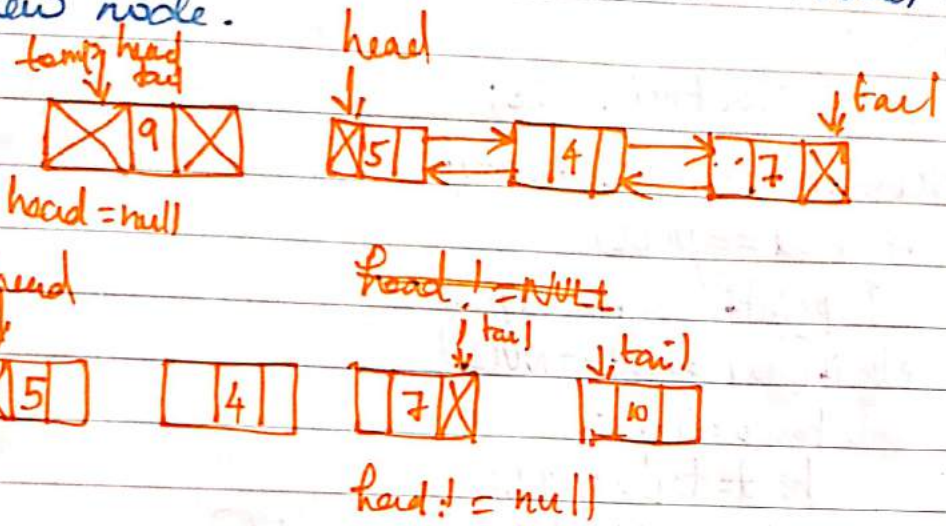



```
#include<stdio.h>
#include<process.h>
struct node
{ int data;
  struct node *prev, *next;
};
```

```
void main()
{ typedef struct node NODE;
  NODE *head=NULL, tail=NAIL, *temp;
  int no;
  temp=(NODE*) malloc(size of(NODE));
  printf("Ete no");
  scanf("%od", &no);
  temp->data=no;
      if(start==NULL)
        { temp->prev=NULL;
          temp->next=NULL;
          tail head=tail=temp;
        }
      else{ temp->next=head;
            head->prev=temp;
            temp->prev=NULL;
            head=temp;
          }
    if(head==NULL)
      { printf("No elmts")
      }
    else
      {printf("Elmts are");
       for(p=head; p!=NULL; p=p->next)
         { printf("%od", p->data);
         }
      }
}
```

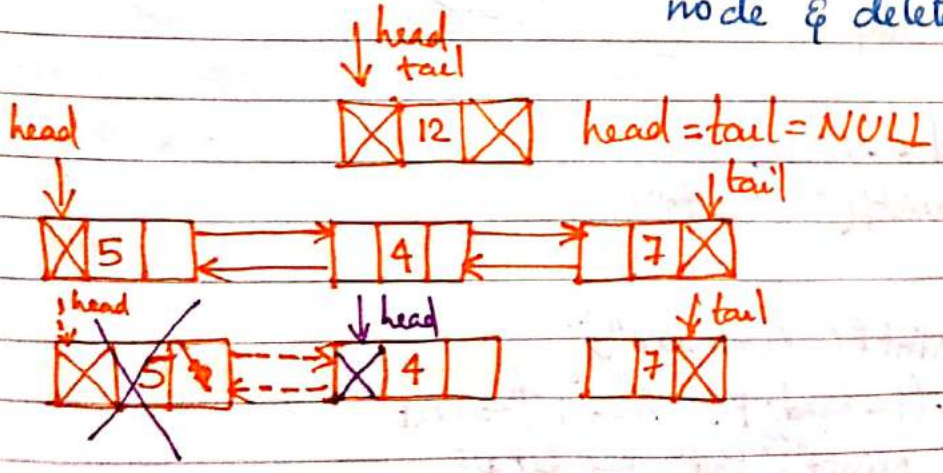
Insert node at End/last position in Doubly Linked List.

1. Allocate memory for new node
2. Assign value to the new node
3. If head = NULL then, set external pointer to the new node, set previous node is NULL & next node is NULL.
4. Otherwise, tail→next=temp, temp→prev=tail & then next pointer of new node set to NULL, tail point to the new node.



Delete a node at start/first position in Doubly Linked List

1. If start = NULL then,
   print deletion is not possible.
2. If the list contain only one element, set external ptr to NULL
3. Otherwise move the external pointer to the second node & delete first node.

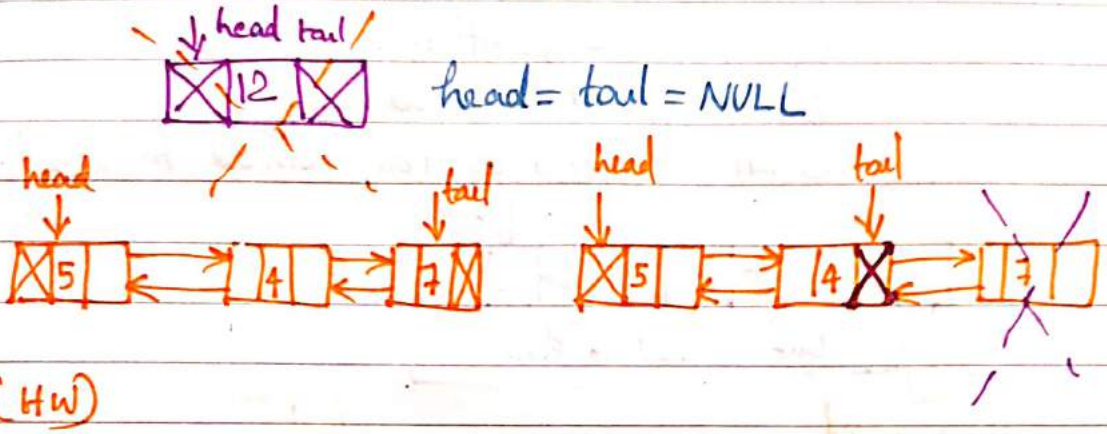


Program

```
#include<stdio.h>
struct node
{ int data;
  struct node *prev,*next;
};
void main()
{ int no;
  typedef struct node NODE;
  NODE *head=NULL, *temp;
    if(head==NULL)
      { printf("Deletion not possible");
    else if(head->next==NULL)
      { temp=start;
        head=tail=NULL;
        printf("Deleted elmt %d", temp->data);
        free(temp);
      }
    else
    { temp=head;
      head=temp->next;
      head->prev=NULL;
    printf("Deltd elmt %d", temp->data);
    free(temp);
  }
 if(head==NULL)
     { printf("No elmts");
     }
else{ printf("Elmts are");
    for(p=head; p!=NULL; p=p->next)
     { printf("%d", p->data);
} }
```

Delete a node at End/Last Position in Doubly Linked List.

1. If the list is empty, deletion is not possible
2. If the list contain only one element, set external pointer to NULL
3. Otherwise go on traversing the last & set next field of second last node to NULL.



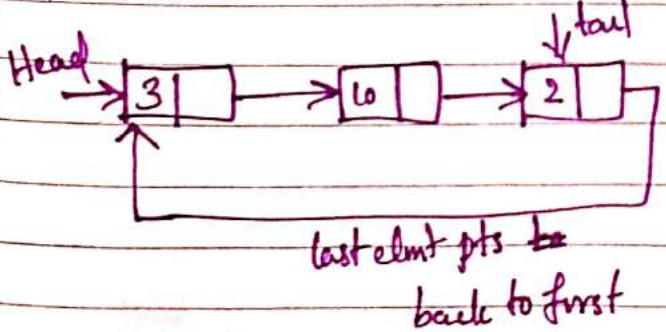
Pgm (HW)

Write a pgm to implement doubly linked list

1. Insert beginning
2. Insert at end
3. Delete from beginning
4. Delete from end
5. Display.

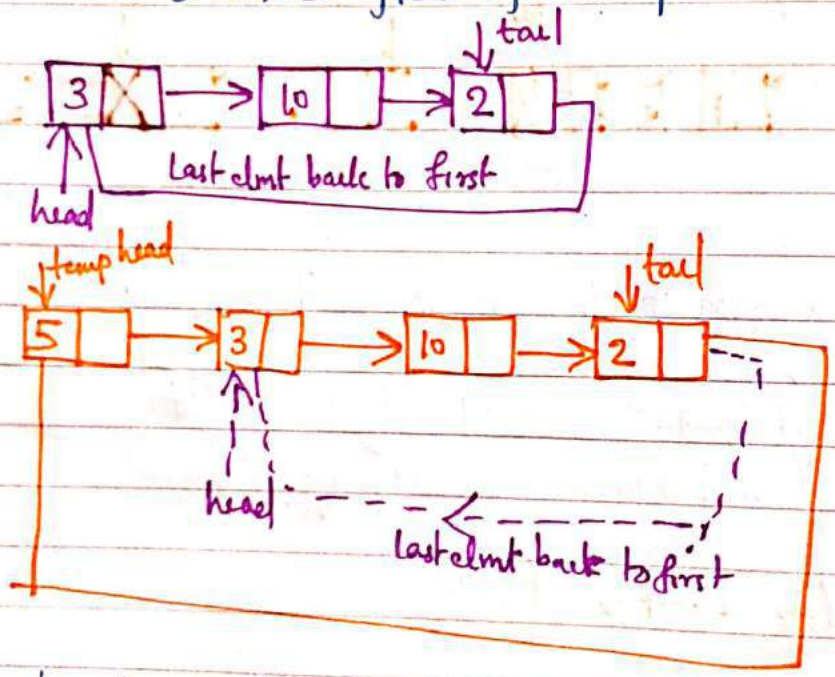
Circular Linked List

In the circular linked list, the previous element stores the address of the next element and the last element stores the address of the starting element. The circular linked list has a dynamic size which means the memory can be allocated when it is required. A circular linked list has no end.



Inserting at the begining

1. Allocate a node at the beginning
2. If list is empty then head & tail point to the new node And linked field of tail pointed to head.
3. If the list is not empty then
   a) New node pointed to head
   b) Head point to new node
   c) Linked field of tail pointed to head.



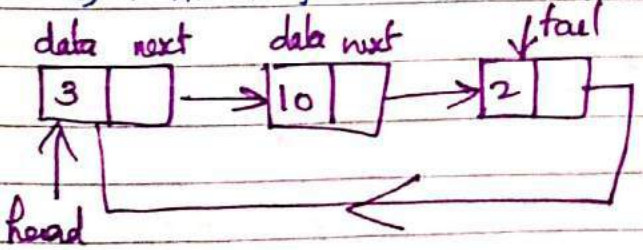
Structure for circular linked list in the same way as for the linear linked list.

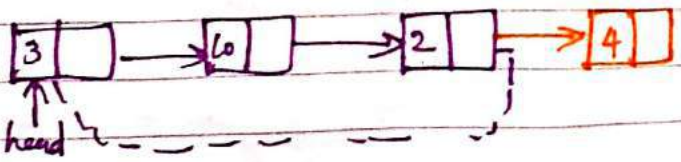
```
struct node
{ int data;
  struct node *next;
};
typedef struct node NODE;
NODE *head=NULL;
NODE *tail=NULL;
```

```
NODE *temp;
temp=(struct NODE*)malloc(sizeof(NODE));
printf("Ets the elmt to be inserted");
scanf("%od",&item);
temp->data=item;
if(head==NULL)
 { head=tail=temp};
   tail->next=head;
 }
else
   { temp->next=head;
     head=temp;
     tail->next=head;
   }
```

Inserting a node at the end

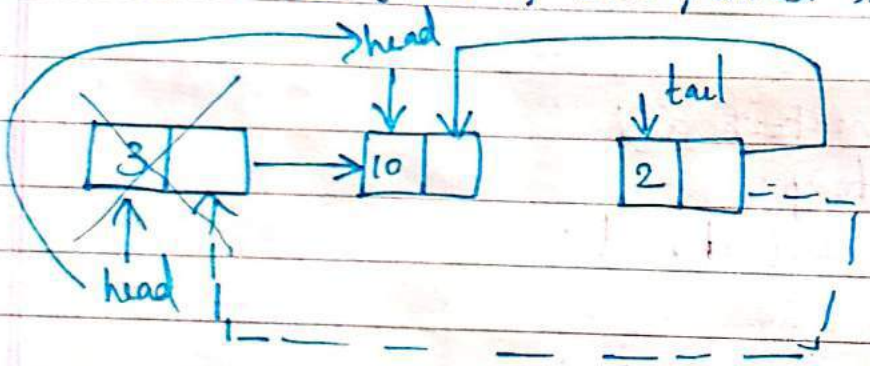
1. Allocate a memory for new node
2. If list is empty, then head & tail point to the new node. And linked field of tail pointed to head.
3. If the list is not empty then
   a) Linked field of tail point to the new node
   b) tail point to new node
   c) Linked field of tail pointed to head





Delete an node from the begening

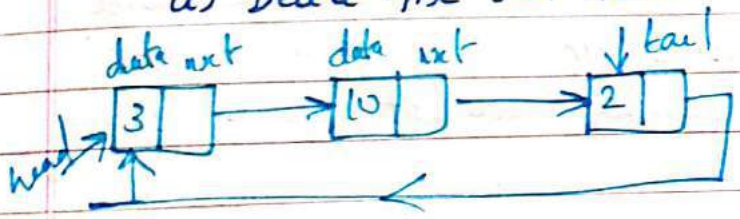
1. If list is empty then print deletion is not possible
2. If the list contain only on element the set head & tail to NULL
3. If the list contain more than one elmt then
   a) temp ptr. point to the head node
   b) head move to the next node
   c) Linked field of tail pointed to head

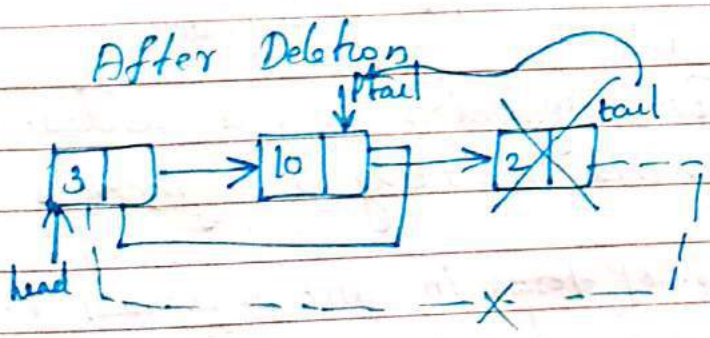



```
if(head == NULL) // List is empty
  { printf(Deletion is not possible");
  }
else if (head == tail)
  { free (head);
    head = tail = NULL;
  }
else
  { temp = head;
    head = head -> next;
    tail -> next = head;
    free (temp);
  }
```

Delete a node from the end:

1. If list is empty then print deletion is not possible.
2. If the list contain only on element the set head & tail to NULL.
3. If the list contain more than one elmt then
   a) 'p' pointer to the second last node & temp pointer points to the last node.
   b) tail move to the p
   c) Linked field of tail pointed to head
   d) Delete the last node.






```
if (head == NULL)
{ printf ("Deletion is not possible");
}
else if (head == tail)
{ free (head);
  head = tail = NULL;
}
else
{ p = head;
  while (p -> next != tail)
  { p = p -> next;
  }
  temp = p -> next;
  tail = p;
  tail -> next = head;
  free (temp);
}
```

Application of Linked list - Polynomial.

- Linked list widely used to represent & manipulate polynomials. Polynomials are the expressions containing no. of terms with nonzero coefficient & exponents.
A node contains 3 fields.
* Coefficient field * Exponent field * Link field

The coefficient field holds the value of the coefficient of a term & the exponent field contains the exponent value of the term. And the link field contains the address of the next term in the polynomial. The polynomial node structure is

Coefficient (coeff)	Exponent (expo)	Addr of the next node (next)

Algorithm: Two polynomials can be added. And the steps involved in adding 2 polynomials are given below.

1. Read the no. of terms in first polynomial P.
2. Read the coefficient & exponent of the 1st polynomial
3. Read the no. of terms in 2nd polynomial Q
4. Read the coefficient & exp. of the 2nd polynomial
5. Set the temporary pointers P & Q to traverse the 2 polynomials respectively.
6. Compare the exponents of 2 polynomials starting from the 1st nodes
   a) If both exponents are equal then add the coefficient & store it in the resultant linked list
   b) If the exponent of the current term in the 1st polynomial P is less than the exp. of the current term

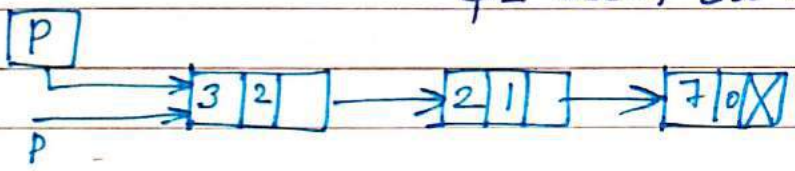
of the 2nd polynomial then added the 2nd term to the resultant linked list. And, move the pointer q to point to the next node in the 2nd polynomial Q.

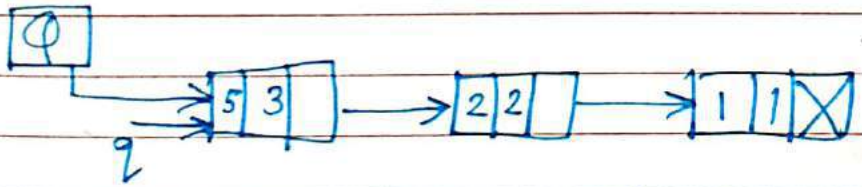
c) If the exp. of the current term in the 1st polynomial P is greater than the exp. of the current term in the 2nd polynomial Q, then the current term of the 1st polynomial is added to the resultant linked list. And move the pointer p to the next node.

d) Append the remaining nodes of either of the polynomials to the resultant linked list.

Illustration: $P = 3x^2 + 2x + 7$

$Q = 5x^3 + 2x^2 + x$





Step1: Compare the exponent of p & the corres. exp. of q.

Here expo(p) < expo(q)

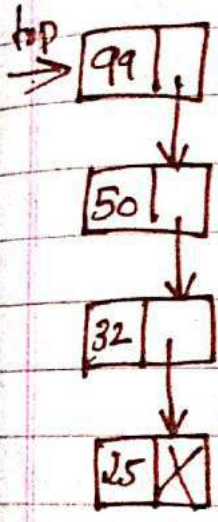
So add the terms pointed to by q to the resultant list. And now advance the q pointer. Step 2.

Stack Using Linked List

The major problem with the stack implemented using array is, it works only for fixed no. of data values. That means the amount of data must be specified at the begining of the implementation itself. Stack implemented using array is not suitable, when we don't know the size of data which we are going to use.

The stack implemented using linked list can coork for unlimited no. of values. ie; works for variable size of data. So no need to fix the size at the begining of the implementation. It can organize as many data values as we want.

In linked list implementation of a stack, every new elmt is inserted as top element. That means every newly inserted elmt is pointed by 'top'. Whenever we want to remove an elmt from the stack, simply remove the node which is pointed by 'top' by moving 'top' to its next node in the list. The next field of the first elmt must be always NULL.



In the eg:, the last inserted node is 99, & the first inserted node is 25. The order of elmts inserted is 25, 32, 50 & 99.

push (value) – Inserting an elmt into the stack

step 1: Create a new Node with a given value.
2: Check whether stack is Empty (top == NULL)
3: If it is empty, then set newNode → next = NULL.
4: If it is not empty, then set newNode → next = top.
5: Finally, set top = newNode.

Pop () – Deleting an element from a stack.

step 1: Check whether stack is Empty (top == NULL)
2: If it is Empty, then display "stack is Empty" "Del" not possible";
3: If it is not Empty, then define a Node pointer 'temp' & set it to 'top'.
4. Then set top = top → next.
5. delete temp (free (temp))

```
#include<stdio.h>
struct node
{ int data;
  struct node *ptr;
};
void main()
{typedef struct node NODE;
NODE *top=NULL,*temp,*t;
int ch, item;
while(1)
{ printf("MENU: 1-Push
          2. Pop 3.Display
          4.Exit \n Enter ur
          choice");
 scanf("%d",&ch);
 switch(ch)
 {
 case 1:
 temp=(NODE *)malloc(sizeof(NODE));
printf("Enter item");
scanf("%d",&item);
temp->data=item;
if(top==NULL)
{ temp->ptr=NULL;
  top=temp;
}
else
 { temp->ptr=top;
   top=temp;
 }
break;
case 2:
 if(top==NULL)
printf("Del^n not possible");
else if(top->ptr==NULL)
 { temp=top;
  top=NULL;
 printf("poped item is %d",
        temp->data);
 free(temp);
 }
else
 {temp=top;
  top=top->ptr;
  printf("poped item is %d",
        temp->data);
  free(temp);
 }
 break;

case 3:
 if(start==NULL)
  printf("stack is empty");
 else
  {printf("Elmnts are");
  for(t=top;t!=NULL;
                t=t->ptr)
  printf("%d",t->data);
 }
 break;
```

```
case 4:
    exit(0);
default:
    printf("wrong choice");
    break;
}
}
```

Linked Representation of Queues:

→ Array must be declared to have some fixed size. If we allocate space for 50 elmts in the queue & it hardly uses 20-25 locations, then half of the space will be wasted. And in case we a less mem. locations for a queue that might end growing large & large, then a lot of re allocato will have to be done, thereby creating a lot of overhead & consuming a lot of time.

If the array size cannot be determined i advance, the linked reps is used.

In a linked queue, every elmt has 2 parts, one that stores the data & another that stores the addres of the next elmt. The start ptr of the linked list i used as front. Another ptr called rear, which will s the address of the last elmt in the queue.

Algorithm to insert an element in a linked queue.

step1: Allocate mem. for the new node & name it as pt

step2: set ptr → Data = val

step3: 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.

Step 1, the mem. is allocated for the new node. step 2, the Data part of the new node is initialised with the value to be stored in the node. Step 3, check if the new node is first node of the linked queue. This is done by checking if FRONT = NULL.; then the new node is tagged as front as well as Rear. If the new node is not the first node in the list, then it is added at the rear end of the linked queue.

Algorithm to delete an elmt from a linked queue.

Step 1: If front = NULL
write "underflow"
goto to step 5
[End of if]
2. set ptr = front
3. set front = front → next
4. free ptr
5. End.

```
)
#include <stdio.h>
struct node
{ int data;
  struct node *ptr;
};
int main()
 {typedef struct node NODE;
NODE *front=NULL,*rear=NULL,
                *temp,*p;
int i, ch,pos,item;
while (1)
{printf("1. Insert 2.Delete 3.Display
                4. Exit Etr choice");
scanf("%d",&ch);
switch(ch)
{ case 1:
temp=(NODE*) malloc(sizeof
                (NODE));
printf("Etr data to be inserted");
scanf("%d",&temp->data);
if(rear==NULL)
 {temp->ptr=NULL;
 front=rear=temp;
 }
else
   { p=front;
 while(p!=rear)
 { p=p->ptr;
 }
```

```
p->ptr = temp;
temp->ptr = NULL;
rear = temp;
} break;
case 2:
  if(front=NULL)
  {printf("No elmts to delete");
}
else
  { if(front->ptr==NULL)
     { temp= front;
       item = temp->data;
printf("Deleted elmt is %d",
                    temp->data);
  free(temp);
  front = rear = NULL;
}
```

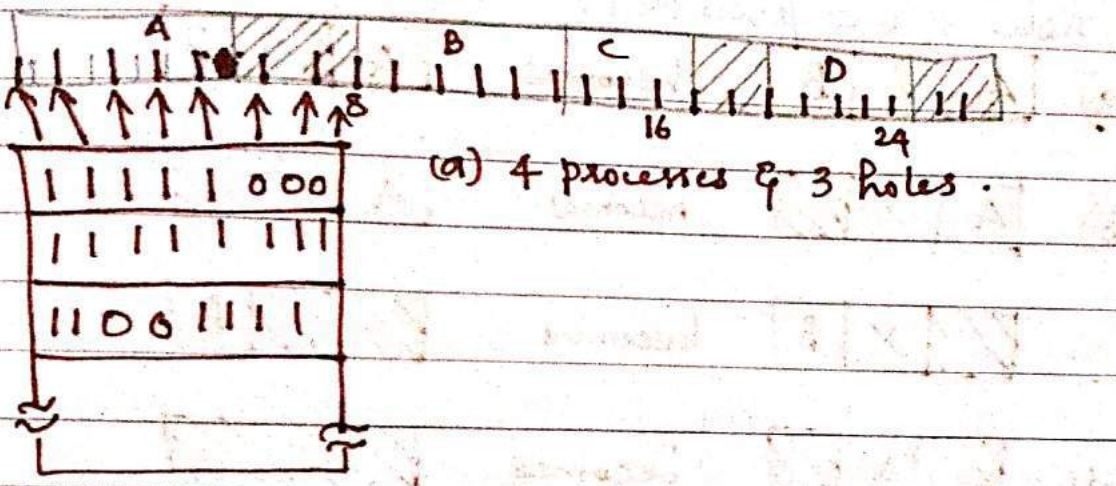
```
else
  { temp = front;
    item= temp->data;
  front=front->ptr;
  free(temp);
  printf("Deleted elmt%d",
}}
  break;
case 3:
  if(front == NULL)
  { printf("no elmts");
  }
  else
  {printf("Elmts are");
  p= front;
  while(p!=rear)
  { printf("%d", p->data);
    p=p->ptr;
  } printf("%d", p->data);
} break;
case 4:
      exit(0);
}
}
```

Applications of Queue

* Memory Management

Memory Management with Bitmaps:

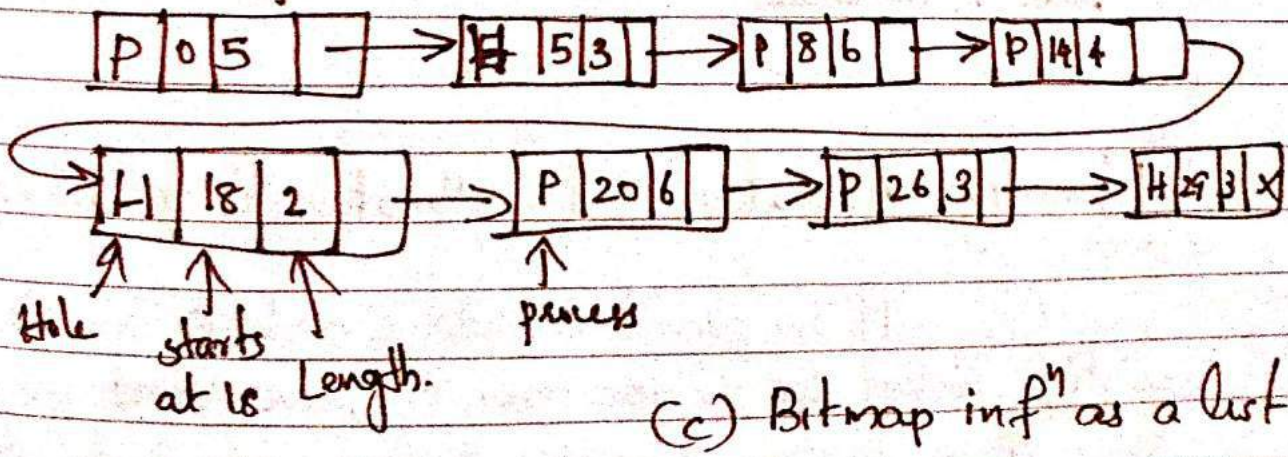
With a bitmap, memory is devided up into allocation units, as small as a few words or several kilobytes. Each allocation unit is a bit in the bitmap, which is 0 if the unit is free & 1 if it is occupied



(a) 4 processes & 3 holes.

(b) ~~part of mem.~~

(b) bitmap.



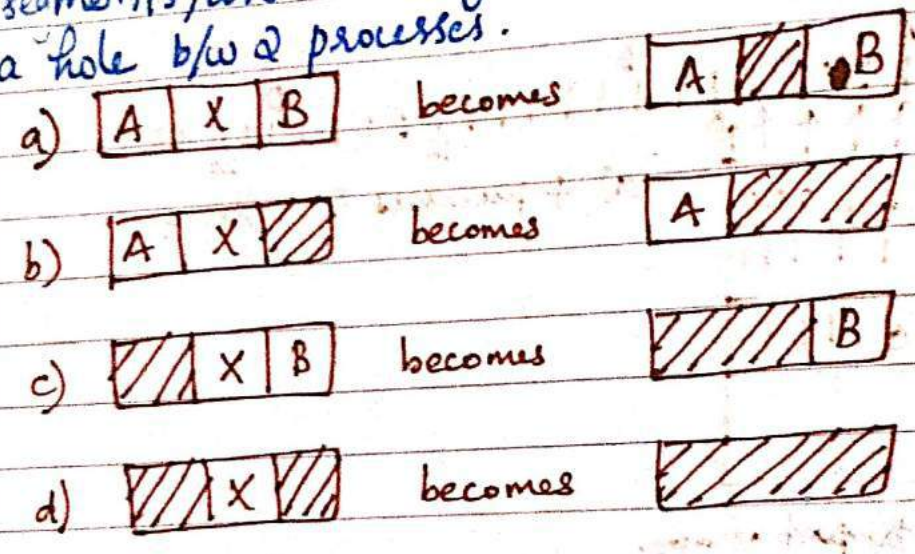
(c) Bitmap inf^{n} as a list.

The smaller the allocation unit, the larger the bitmap. A bitmap provides a simple way to keep track of memory words in a fixed amnt of mem. b/c the size of the bitmap depends only on the size of memory & the size of the allocation unit.

Drawback:

—when it has been decided to bring a k
process into memory, the mem. mngr must search
the bitmap to find a run of k consecutive 0 bi
in the map. Searching a bitmap for a run of a giv
length is a slow operation.

Another way of keeping track of memory is
to maintain a linked list of allocated & free mem
segments, where a segment is either a process or
a hole b/w 2 processes.



Memory Allocation & De-allocation for a Linked list

If we want to add a node to an already existing linked list in the memory, we first find free space in the memory & then use it to store the info. The computer maintains a list of all free memory cells. This list of available space is called the free pool.

O.S adds the freed memory to the free pool. The O.S will perform this opn whenever it finds

the CPU idle or whenever the pgms are falling short of memory space. The O-S scans through all the memory cells that are being used by some pgms. Then it collects all the cells not being used & adds their address to the free pool, so-that these cells can be reused by other pgms. This process is called garbage collection.

First fit, best fit, worst fit allocation schemes.

When the processes & holes are kept on a list sorted by address, several algms can be used to allocate memory for newly created process. We assume that the memory mngr knows how much memory to allocate.

First fit: The proces mngr scans along the list of segments until it finds a hole that is big enough. The hole is broken up into 2 pieces, one for the process & one for the unused memory except in the statistically case of exact fit.

Next fit: It keeps track of where it is whenever it finds a suitable hole. The next time it is called to find a hole, it starts searching the list from the place where it left off last time, instead of always at the begining, as first fit does.

Best Fit: It searches entire list & takes the smallest hole that is adequate. Best fit tries to find a hole that is close to the actual size needed.

Worst Fit: Always take the largest available hole, so that the hole broken off will be big enough to be useful.

MODULE III

STACK

A stack is a list of elmts in which an elmt may be inserted or deleted only at one end, called the top of the stack. Stacks are sometimes known as LIFO lists.

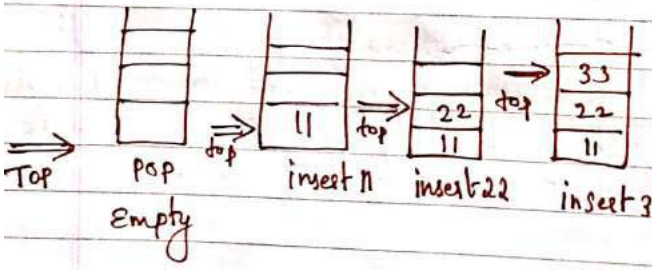
As the items can be added or removed only from the top. The 2 basic op^s associated with stacks are

Push: is the term used to insert an elmt into a stack

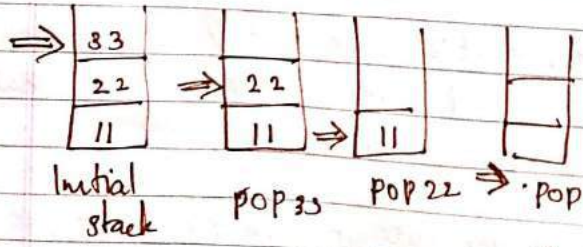
Pop: is the term used to delete an elmt from the stack

Representation of Stack:

Let us consider a stack with 5 elmts capacity. This is called the size of the stack. The no. of elmts to be added should not exceed the max. size of the stack. If we attempt to add new elmt beyond the max. size, we will encounter a stack overflow condition. If we cannot remove elmts beyond the base of the stack, we will reach a stack underflow condition.



Push op^n on stack.



Algorithm

1. Start
2. set top=-1
3. Read choice ch
4. If ch=1 (PUSH), then
   goto 4.a else 4.b
   a. if top<4 then
      Read the item, set top=top+1
      set stack[top] = item
   b. Else print stack overflow.
5. If ch=2 (POP), then goto 5.a else 5.b
   a. if top ≥ 0 then
      Set item = stack[top]
      top = top-1
      print no. deleted is item.
   b. Else print stack underflow.
6. If ch=3 then
   Display the elmts in the stack.
7. If ch=4, then
   Exit
8. Stop.

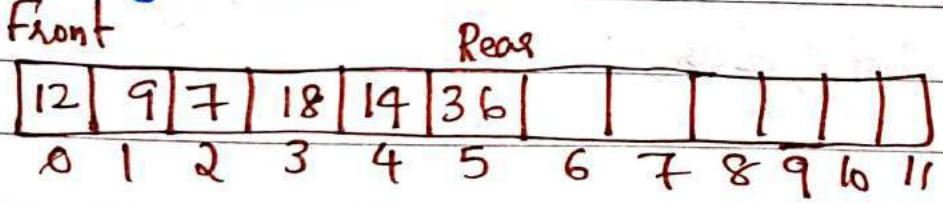
Programs:

```
#include<stdio.h>
void main()
{ int stack[10], ch, i, item, top=-1;
while(1)
{ printf("Menu: 1. Push 2. Pop
        3. traverse 4. Exit
        Etr your choice");
 scanf("%d", &ch);
 switch(ch)
  { case 1:
    if (top==9)
      { printf("stack overflow");
      }
    else
      { printf("Etr the item");
        scanf("%d", &item);
        top=top+1;
        stack[top] = item;
      }
      break;
  case 2:
    if(top==-1)
      printf("stack underflow");
    else
      { item=stack[top];
        top=top-1;
        printf("Deleted item is %d", item);
      } break;
```

```
case 3:
    if(top>=0)
    if(top==-1)
      printf("No elmts");
    else
    { printf("Elmts are");
      for(i=top; i>=0; i--)
        printf("%d", stack[i]);
    }
  break;
 case 4:
      exit(0);
  }
 }
}
```

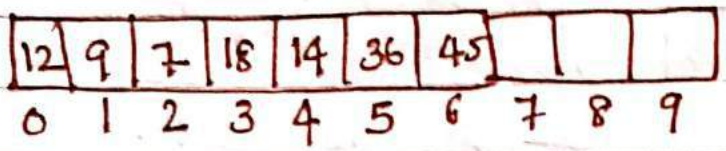
Queues

A queue is a FIFO data structure in whi
the elmt that is inserted first is first one to be taken ou
The elmts in a queue are added at one end called the
rear and removed from the other end called the front.
Queues can be implemented by using either arrays or linke
lists. Every queue has front & rear variables that point
to the position from where deletions & insertions can be do
respectively.

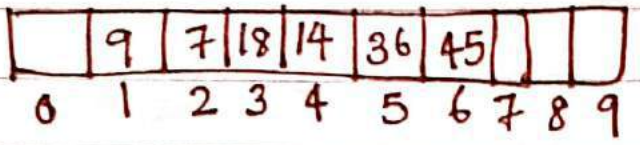


Array rep of queue.

Here, front=0 & rear=5. If we want to add anothe
elmt with the value 45, then the rear would be incremented
by 1 & the value would be stored at the pos" pointed by the rear



After insertion front = 0 rear = 6.



Queue after deletion front = 1 rear = 6.

Insertion of an element

step 1: If Rear = Max-1
Write overflow
go to step 4
(End of IF)
step 2: If front = -1 and Rear = -1
Set front = Rear = 0
Else
Set rear = rear + 1
(End of IF)
step 3: Set queue [Rear] = NUM
step 4: Exit

Deletion of an element

step 1: If front = -1 or front > Rear
Write under flow.
else
set Val = queue [front]
set front = front + 1
[End of if]
step 2: Exit.

```
int main()
{ int ch, front = -1, rear = -1, q[10], item, i;
  while(1)
  { printf("Queue 1. Insert 2. Delete 3. Display 4. exit");
    scanf("%d", &ch);
    switch(ch)
    { case 1:
        if(rear == 9)
          { printf("Queue is full");
          }
```

```
else
  { if(front==-1 && rear==-1)
    { front=0; rear=0;
    }
    else
      rear++;
      printf("Ets the inserted item:");
      scanf("%d", &item);
      q[rear]=item;
  }
case 2:
  if((front==-1 && rear==-1) || front==rear+1)
    { printf("queue is empty");
    }
    else
      { item=q[front];
        front++;
        printf("Deleted item is %d", item);
      }
      break;
case 3:
  if((front==-1 && rear==-1) || front==rear+1)
    { printf("queue is empty");
    }
  else
    { printf("elmts are");
      for(i=front ; i<=rear; i++)
        printf("%d", q[i]);
    }
      break;
```

3. Converting INFIX to POSTFIX

The rules to be remembered during infix to postfix conversion are:

1. Paranthesize the expn starting from left to right.
2. During paranthesizing the expn, the operands associated with operator having higher precedence are first paranthesized.
3. The sub-expression which has been converted to postfix is to be treated as single operand.
4. Once the expn is converted to postfix, remove paranthesis.

Eg: Convert the A*B+C/D to postfix form.

A*B + C/D

(A*B) + (C/D)

AB* + CD/

AB* CD/+

Suppose Q is an arithmetic expn written in notation. This algm finds the equivalence postfix expn P.

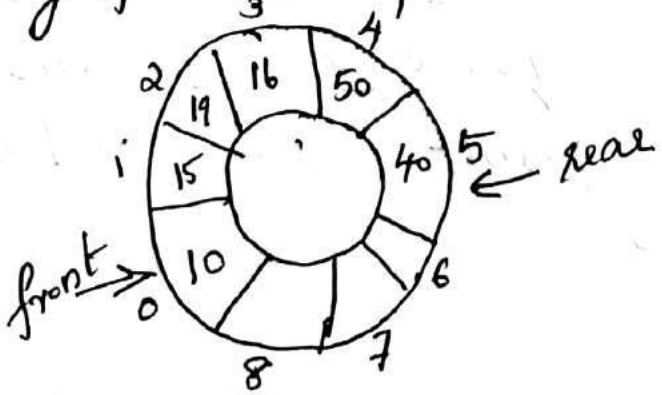
1. Scan Q from left to right and repeat step 2 to 4 for each elmt of Q until the STACK is empty.
2. If an operand is encountered, add it to P.
3. If a left paranthesis encountered, push it into STACK.
4. If an ~~operator~~ operator is encountered, then
   a) Repeatedly pop from the stack & add to P each operator (on the top of the stack) which has the same precedence or higher precedence than operator.

Circular Queue

— A circular queue is one in which the insertion of new elmt is done at the very first location of the queue, if the last location of the queue is full in circular fashion.

- Circular queue has 2 ends
  - front end – deletion
  - rear end – insertion
- traversal in one direction from front to rear.
- has starting pt. but no end point.

1. Front end will always be pointing to the first elmt of the queue.
2. If front == rear, the queue will be empty.
3. Insertion : Rear = rear + 1.
4. Deletion : front = front + 1
5. Initially front = 0 & rear = -1.



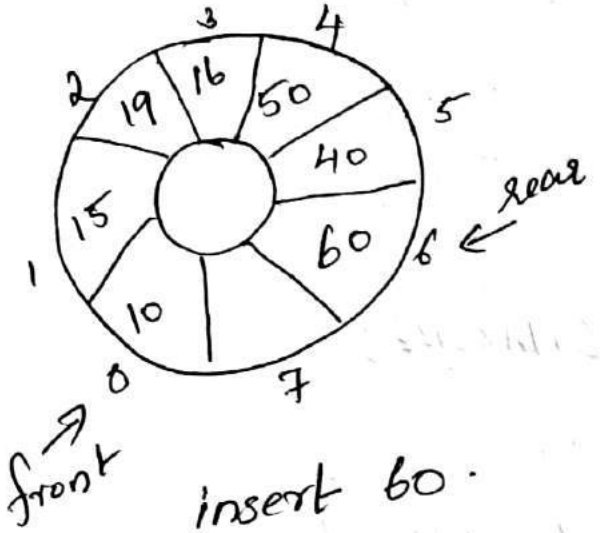
Algorithm for Insertion

1. Check whether queue is full or not. If it is full, insertion is not possible.
2. If queue is not full, check whether front=-1. If so set rear=0. front=0 otherwise
   a) The posn to insert element is calculated by rear = (rear+1) % MAXSIZE
3. Insert the element in rear position.

Program:

```
void insert()
{ if(front==(rear+1)%MAXSIZE)
   printf("Queue is full");
  else
   { printf("Etr Item");
     scanf("%d", &item);
     if(front==-1)
       front=rear=0;
     else
      rear=(rear+1)%MAXSIZE;
      queue[rear]=item;
      printf("Item inserted %d", item);
   }
}
```

Example:



insert 60.

Deletion

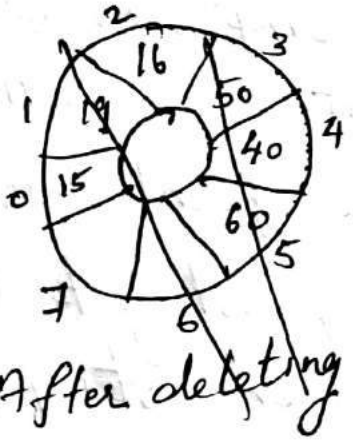
Algorithm:

1. Check whether queue is empty or not. If the queue is empty then quit, else continue.
2. If not empty, Delete the 'front' element.
3. If the front & rear is pointing to the last position of the queue then step 4 else step 5.
4. Set front = rear = -1
5. Increment the "front" position by one.

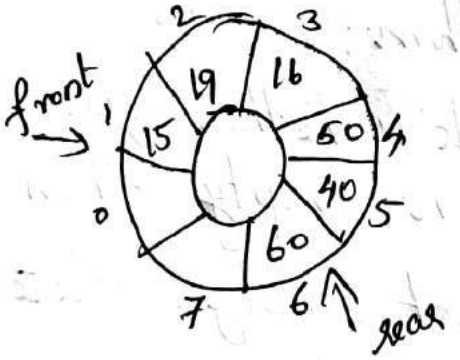
Program

```
void delete()
{ if (front == -1)
  printf("Queue is empty");
  else
     { item = queue[front];
```

```
if(front == rear)
  {
    front = -1;
    rear = -1;
  }
else
    front = (front+1) % MAXSIZE;
  }
}
  printf("\nItem deleted : %d", item);
}
```



After deleting



After deletion of 10.

Priority Queue

In Priority Queue the items are ordered by key value so that item with the lowest value of key is at front & item with the highest value of key at the rear or vice versa. So we are assigned priority to item based on its key value. Lower the value, higher the priority.

Basic Operations:

Insert/enqueue - add an item to the rear of the queue.

remove/dequeue - remove an item from the front of the queue.

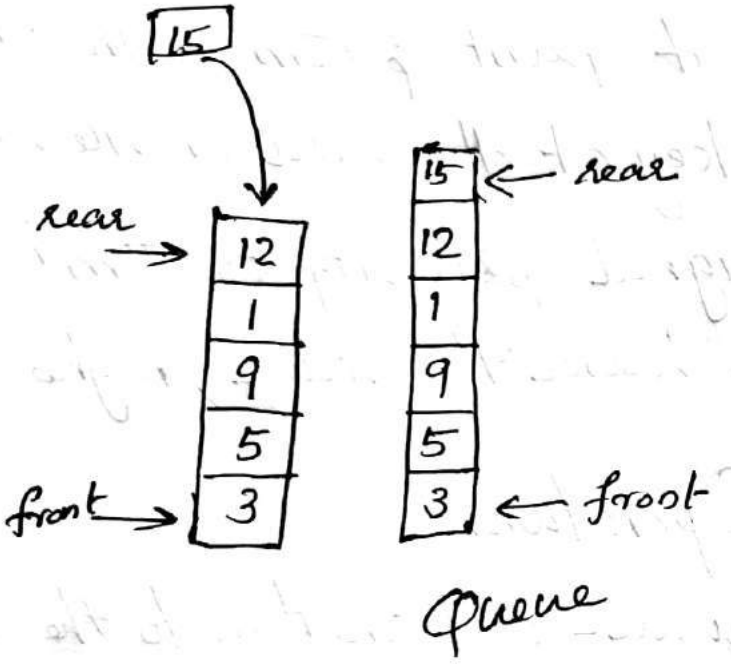
Peek - get the elmt at front of the queue

isFull - check if queue is full.

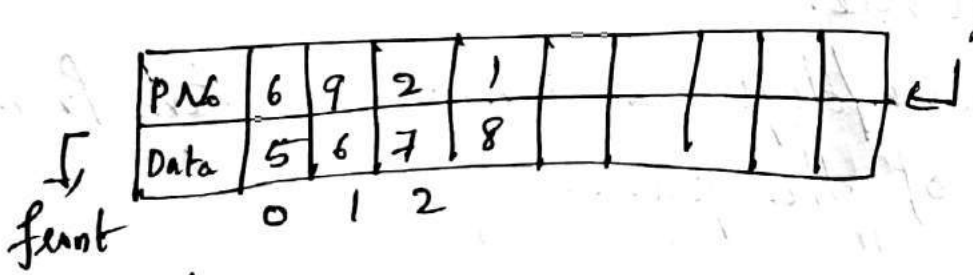
isEmpty - check if queue is empty.

Insert/Enqueue Operation.

Whenever an elmt is inserted into queue priority queue inserts the item acc-to its order.



Queue



Insertion:

```
void insertion()
{ int n; pn;
   if (rear == Max size - 1)
     { printf("Overflow").
     }
else { printf("Ete data");
       scanf("%d", n);
       printf("Ete priority");
```

```
scanf("%d", &pn);
if(rear == -1)
    { rear++;
      front++;
    }
  else
    {
      rear++;
    }
pqueue[rear].data = n;
    pqueue[rear].pno = pn;
  }
}
```

Algorithm

1 if (rear == Max size - 1)
    print("overflow")
    exit
  endif
2 read num & pn
3 if (rear == -1)
    set front ← front + 1
    set rear ← rear + 1
4 else
    set rear ← rear + 1
5   set pqueue[rear].data ← num
6   set pqueue[rear].pno ← pn

all algorithms. It is simple to understand and implement. Linear search, also known as Sequential search means, starting at the beginning of the data and checking each item in turn until either the desired item is found or the end of the data is reached.

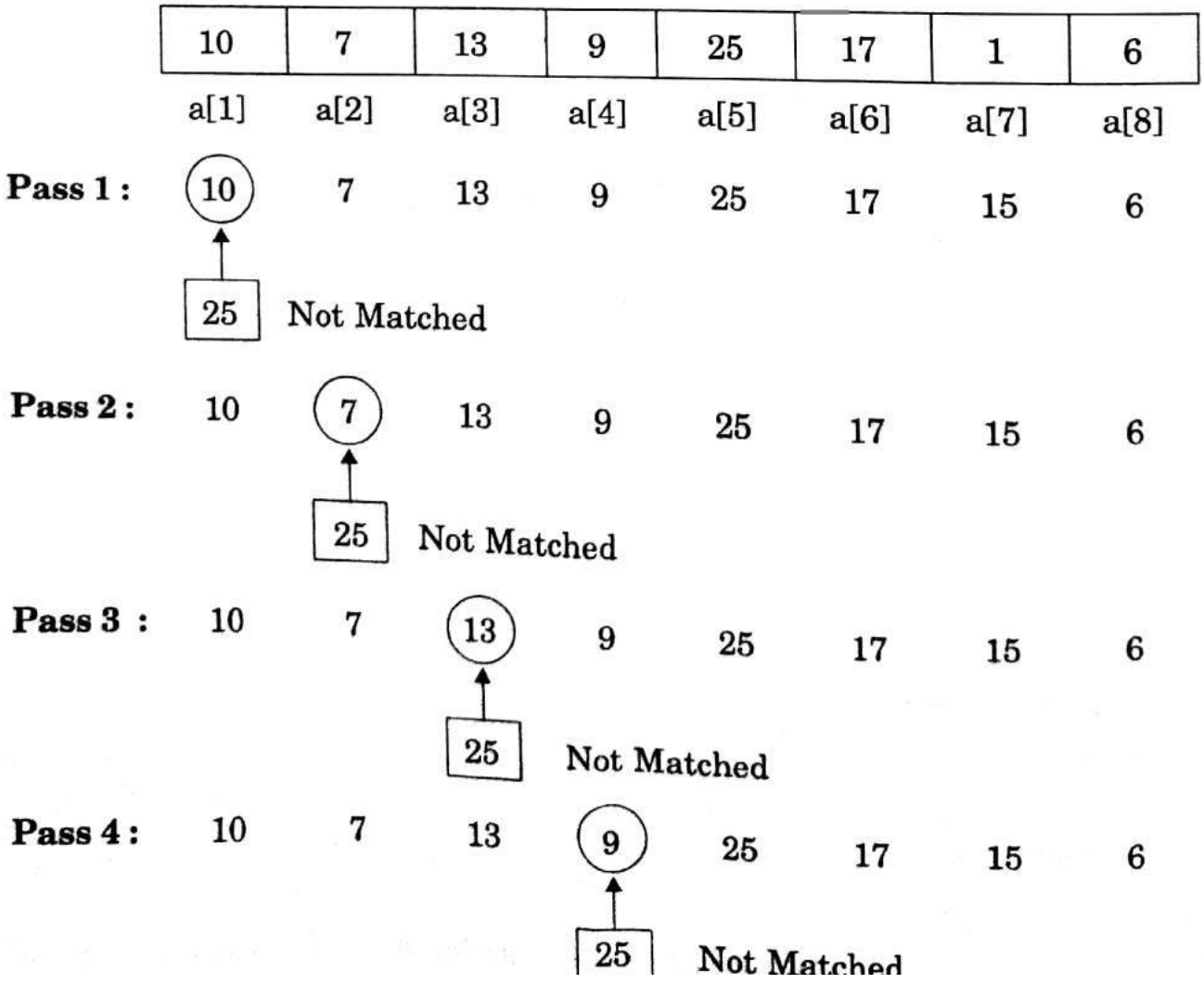
Let a be the array of n data and let x be the data to be searched. Our aim is to find whether x is present in the array. If x is in the array, print "search success". It not print "search failed". The steps to be followed are

- compare x with a[1] if equals print search success, or else compare x with a[2] if equals print success and so on. The process is repeated till last array element i.e.; a[n].
- if no array value matches print search failed.

Example

Consider an array having 8 elements. Using the array, find whether 25 is present or not.

Search data : 25



Pass 5 : 10 7 13 9 (25) 17 15 6

25 **Matched**

Procedure

```
Search (a,n,x)          //x – search data
{
for i=1 to n do
if (x= = a[i]) then
{
print "search success";
return;
}
Print "search failed";
}
```

Complexity

Let n be the maximum number of comparisons required to search an element in an array using linear search.

Then f(n) = n

f(n) = O(n)

9.11.2 Binary Search

Let 'a' be the **sorted array** of n data with lower bound(lb) and upper bound(ub).The following steps are used to find whether the search item 'X' is present or not.

i. Let 'a' be an array of sorted data.
ii. Let 'X' be the given data for searching.
iii. Find the mid position of the given array as mid = (lb+ub)/2.
iv. Compare 'X' with mid position value. If it is equal, searching is over. Return mid.
v. If 'X' is less than mid value, repeat steps (iii) and (iv) for the sub array to mid – 1.
vi. If 'X' is greater than mid value, repeat steps (iii) and (iv) for the sub array mid + 1 to n.
vii. If no values matches return 0;

Procedure:

```
BinarySearch(a,n,X,lb,ub)
{ low = 0;
```

```
high = n-1;
while (lb <= ub)
{
mid = (lb+ ub) /2
if (a[mid] > X)
ub = mid -1
else if (A[mid] < X)
lb = mid +1
else
return mid // found
}
return 0;          // not found
}
```

Example:

Find whether 33 is present in the given sorted array or not.

Sorted array:

5	13	18	25	33	48	54	60	80	85
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

Solution:

Search item = 33

lb=0 and ub=9

Pass 1 :

Compute mid=(lb + ub)/2 = (0 + 9)/2 = 5

Index = 5, so a[5] = 48 is mid.

Compare mid with search item 33.

5	13	18	25	33	48	54	60	80	85
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

33 search item less than mid value.

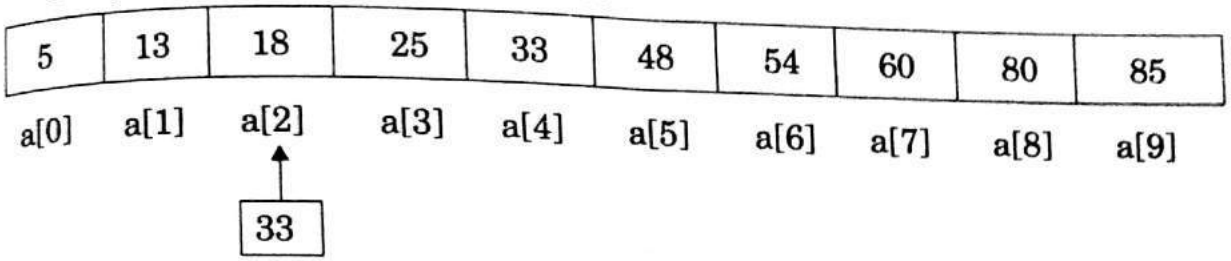
Change ub = mid – 1= 5 – 1 = 4

Pass 2 :

Compute mid=(lb + ub)/2 = (0 + 4)/2 = 2

Index = 2 , so a[2] = 18 is mid.

Compare mid with search item 33.



search item greater than mid value.Between 18 & 48

Change lb = mid +1 = 2 + 1

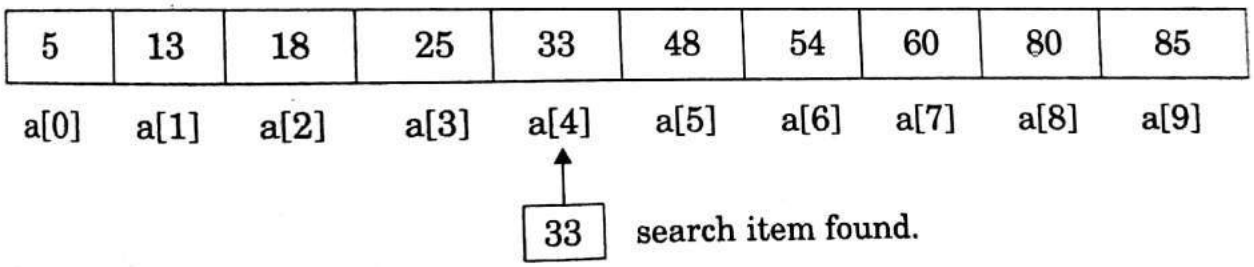
= 3

Pass 3:

Compute mid=(lb + ub)/2 = (3 + 4)/2 = 4

Index = 4 , so a[4] = 33 is mid.

Compare mid with search item 33.



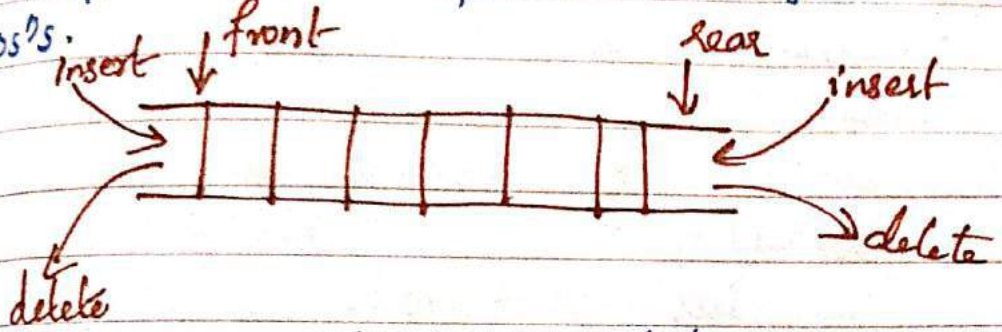
Complexity

In this search, each pass reduces the array size to half. that is, in the first pass the array size is 'n' and in the next pass we only consider only the n/2 array size elements.

Therefore, $f(n) = [\log_2 n] + 1$

Double Ended Queue (Deque)

Double Ended Queue is a queue data structure in which the insertion & deletion op's are performed at both the ends (front & rear). We can insert at both front & rear positions & can delete from both front & rear pos's.

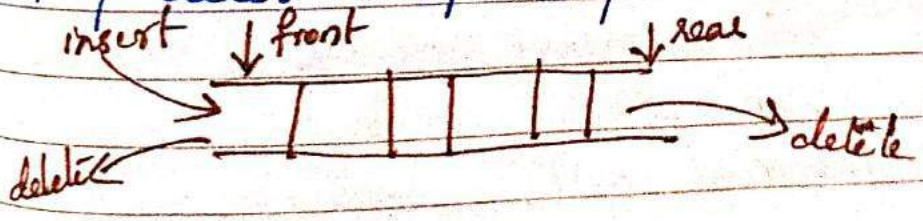


DEQUEUE can be represented in 2 ways,

1. Input restricted DEQUEUE
2. Output restricted DEQUEUE

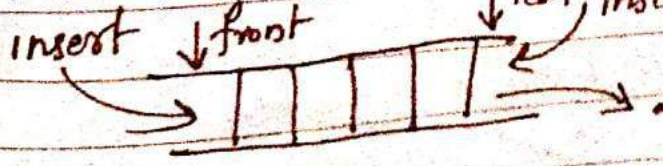
Input Restricted DEQUEUE:

The insertion op^n is performed at only one end & deletion op^n is performed at both ends.



Output Restricted Dequeue:

Deletion is performed at only one end & insertion is performed at both the ends.



Algorithm for insertion at rear end:

step 1 : Check for overflow
if (rear == MAX)
print("Queue is Overflow");
return;
step 2: Insert element
else
rear = rear + 1;
q[rear] = no;
set rear & front pointer
if (rear = -1
rear = front = 0;
step -3 : return

Algorithm for insertion at front end
step-1 : Check for front position
if (front == 0)
print("cannot add item at front end");
return;
step-2 : [insert at front]
else
front = front - 1;
q[front] = no;
step-3 : Return;

Algm for Deletion from front end:
step-1 [Check for front pointer]
if front = -1
print("Queue underflow");
return;

step 2 : [perform deletion]
else
no = q[front];
print("Deleted element is", no);
[set front & rear pointer]
if front = rear
front = rear = -1;
else
front = front + 1;
Step 3: return;

Algorithm for Deletion from rear end
step-1 : [check for the rear pointer]
if rear = 0
print("cannot delete");
return;
step-2 : [perform deletion]
else
no = q[rear];
[check for the front & rear pointer]
if front = rear
front = rear = -1;
else
rear = rear - 1;
print("Deleted elmt is", no);
step-3 : Return;

Polynomial Representation using Arrays.

Eg: $P_1(x) = 8x^3 + 3x^2 + 2x + 6$

$P_2(x) = 23x^4 + 18x - 3$

$P_1(x)$

6	2	3	8
0	1	2	3

$P_2(x)$

-3	18	0	0	23
0	1	2	3	4

Index represents the exponents.

Disadvantage:

$P_3(x) = 16x^{21} - 3x^5 + 2x + 6$

6	2	0	0	-3	0	.....	0	16
0								21

waste of space

Advantage good for non-sparse polynomials.

Poly repn using array

```
main()
{
  float poly[10];
    int d, i;
  printf("Enter the degree");
  scanf("%d", &d);
  if(d<1)
    { printf("Degree should be 1");
  else
    printf("Enter Coeff");
    for(i=d; i>=0; i--)
  printf("The polynomial:")
  for(i=d; i>=0; i--)
   { if(pol[i]!=0)
     { if(i==d)
      printf("%f x^%d", pol[i], i);
      else if(i==1)
          printf("+%f x", pol[i]);
      else if(i==0)
       printf("+%f", pol[i]);
       else
        printf("+%f x^%d", pol[i], i);
     }
```

Sparse Matrix Representation

* Row - index of row, where non zero elmt is located
* Column - index of column, where non zero elmt is located.
* Value - value of the non zero elmt located at index.

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix} \Longrightarrow$$

Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

```
main()
{ int a[3][3], i, j, k, c=0;
  printf("Enter Matrix elmts");
   for(i=0; i<3; i++)
    { for(j=0; j<3; j++)
      {scanf("%d", &a[i][j]);
      }
    }
   for (i=0; i<3; i++)
    { for(j=0; j<3; j++)
      { if(a[i][j] != 0)
        {c++;
        }
      }
    }
  int b[i][j];
 for(i=0; i<3; i++)
  { for(j=0; j<3; j++)
     if(a[i][j] != 0)
      { b[0][k] = i;
        b[1][k] = j;
        b[2][k] = a[i][j];
        k++;
      }
  }
  for(i=0; i<3; i++)
    for(j=0; j<c; j++)
     { printf("%d", b[i][j]);
     }
}
```

MODULE IV

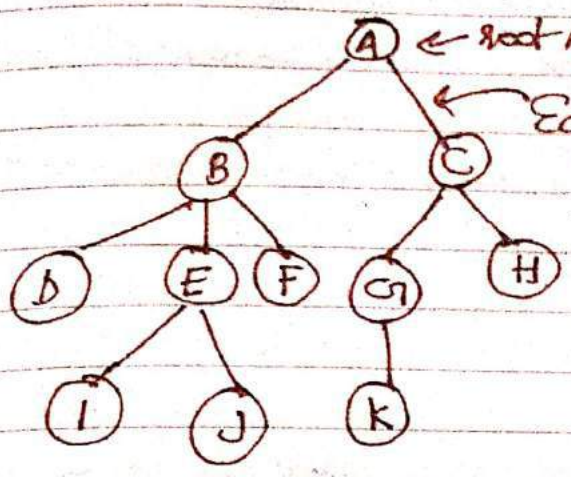
Tree

Tree data structure is a collection of data (Node) which is organized in hierarchical structure & this is a recursive definition.

- every individual elmt is called Node. Node in a tree data structure, stores the actual data of that particular elmt & link to next elmt in hierarchical structure.

Tree terminologies:

1. Root: The first node is called as Root Node. Every tree must have root node. It is the origin of tree data structure.



Parent Nodes: A, B, C, E, & G.

B & C, D, E, F, G ← child

Edge: Connecting link b/w any 2 nodes is called as EDGE. In a tree with N nodes, there will be a max. of N-1 no. of edges.

Parent: the node which is predecessor of any node is called as PARENT NODE.

Child: which is descendant of any node.

Siblings: nodes which belong to same parent.

Leaf / Terminal Nodes: which doesn't have a child. The degree of a node is zero, then it is a leaf.

→ External nodes.

Internal nodes / Non terminal nodes: the node which has atleast one child is called as Internal Nodes. Root node is also Internal.

Degree: Total no. of children. The highest degree of a node among all the nodes in a tree is called as Degree of Tree.

Level: the root node said to be at level 0 & children of Root node at level 1. & so on.. In a tree each step from top to bottom is called as a Level. & level count starts from 0 & incremented by 1.

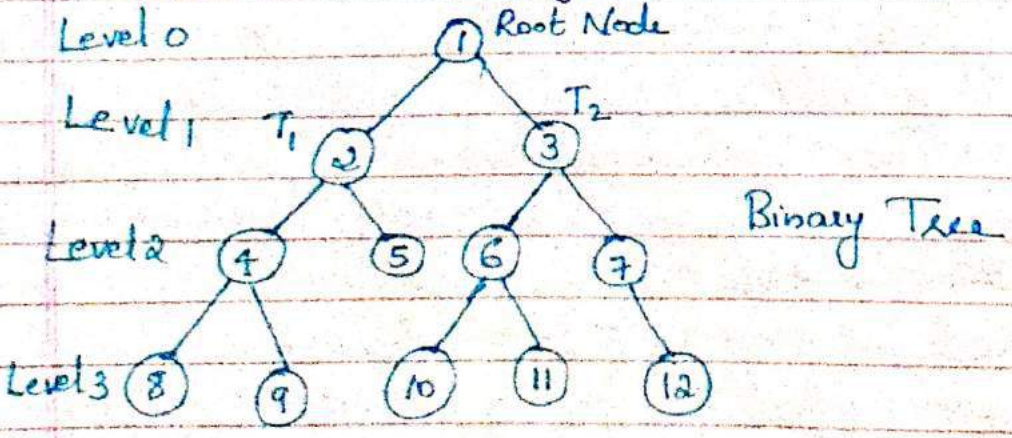
Height: the total no. of edges from leaf node to a particular node in the longest path is called as Height of that node.

Depth: total no. of edges from root node to a particular node is called as depth of that node.

Path: In a tree data structure, the sequence of Nodes from one node to another node is called as path blw that 2 nodes.
Subtree: Each child from a node forms a subtree recursively.

Binary Tree

A binary tree is a data structure that is defined as a collection of elements called nodes. In a binary tree, the topmost element is called the root node & each node has 0, 1 or at the most 2 children. A node that has zero children is called a leaf node or a terminal node. Every node contains a data elmt, a left ptr which points to left ~~elmt~~ child, & right ptr which points to the right child. The root elmt is pointed by a 'root' pointer. If root = NULL, then it means the tree is empty.



Binary Tree

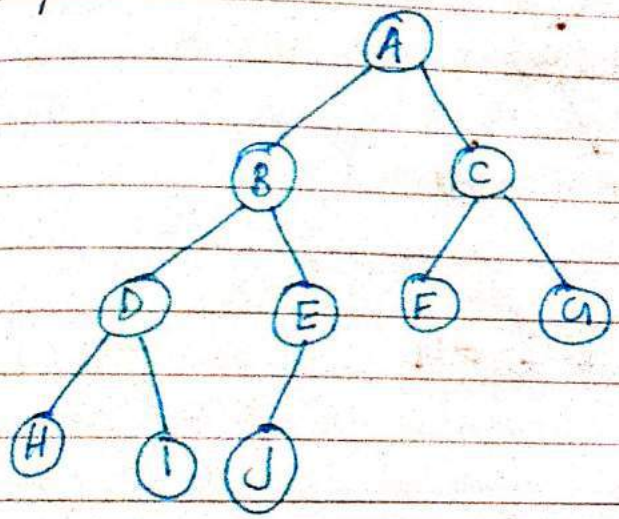
Complete Binary Tree

- 2 properties:

1. In a complete binary tree, every level, except possibly the last, is completely filled.

2. All nodes appear as far left as possible. Or The bottom level is filled from left to right.

A complete binary tree has all the leaf nodes. In the complete binary tree, all the nodes have left & right child nodes except the bottom level. At the bottom level, the nodes from left to right. The bottom level may not be completely filled, depicting that the tree is not a perfectly complete one:



A complete binary tree of height h has b/w 2^h to $2^{h+1}-1$ nodes.

Full Binary Tree

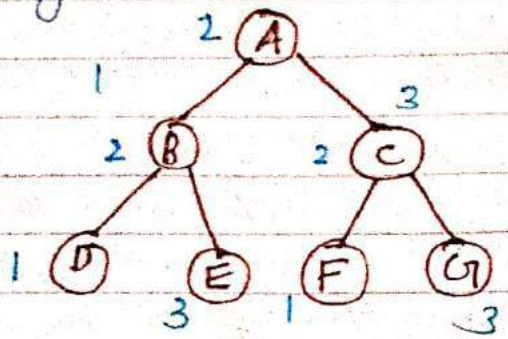
* Strictly Binary Tree either has 0 or 2 subtrees.
* A strictly Binary Tree with n leaves always contains '2n-1' nodes.

Tree Traversal

Traversal is a process to visit all the nodes of a tree & may print their values too. Because, all nodes are connected via edges (links), we always start from the root node. We cannot random access a node in a tree. There are 3 ways: — In- order Traversal
— Pre- order "
— Post- order "

In-order Traversal:

-the left subtree is visited first, then the root and later the right sub-tree. Every node may represent a subtree itself.
If a binary tree is traversed in-order, the o/p will produce sorted key values in an ascending order.



D→B→E→A→F→C→G

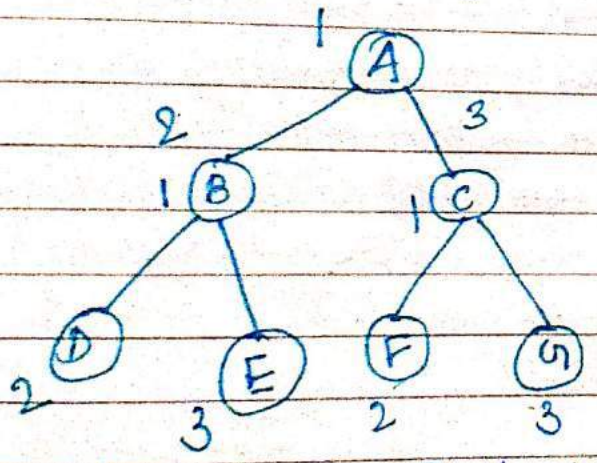
Algorithm:
Until all nodes are traversed-
step 1: Traverse the left subtree in Inorder
step 2: Visit root node
step 3: Traverse the right subtree in Inorder.

Non recursive Algorithm
-using stack: Initially push NULL onto stack and then set P= root and ~~the~~ repeat the following steps until NULL is popped from stack.
step 1: ~~proceed~~ proceed down the leftmost path rooted at P, pushing each node N with no leftchild is pushed onto stack.
Step 2:- Pop and process the nodes on stack. If NULL is popped then exit. If a node N with a right child P→right is processed set P= P→right and return to step 1.

```
void intrav(nodeptr tree)
{ if(tree != NULL)
    { intrav(tree->left);
      printf("->%d", tree->data);
      intrav(tree->right);
    }
}
```

Preorder Traversal

→ the root node is visited first, then the left subtree & finally the right subtree.



A → B → D → E → C → F → G

Left subtree Right subtree

Algorithm

Until all nodes are traversed -
step 1 - Visit root node
step 2 - traverse the left subtree in pre order.
step 3 - traverse the right subtree in pre order.

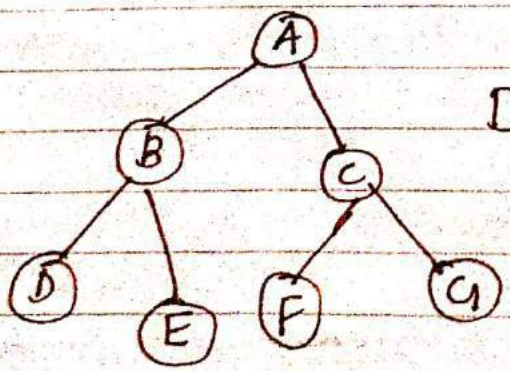
Non recussive Algorithm

This algm traverses the tree non-recursively in preorder using stack. Initially push NULL onto stack & then set P = root & repeat the follo. steps ~~if~~ if P != NULL.

Step 1- Proceed down the leftmost path rooted at P, processing each node N on the path & pushing each right child, P→right. If any, onto stack. The traversing ends after a node N with no left child L(N) is processed. Therefore the pointer P is updated using the assignment P = P→left & the traversing stops when P→left = NULL.
Step 2- Pop the elmt from stack & assign to P. If P≠NULL then return step 1 else exit.

```
void pretrav(nodeptr tree)
  { if(tree != NULL)
      { printf("→%d", tree→data);
        pretrav(tree→left);
        pretrav(tree→right);
  } }
```

Post-order Traversal



D→E→B→F→G→C→A

→The root node is visited last, hence the name. First traverse the left subtree, then the right subtree & finally root node.

Algorithm:
Until all nodes are traversed - Step 1- Traverse the left subtree in Postorder
Step 2- Traverse the right " "
Step 3- Visit root node.

Non-Recursive Algorithm

This algm traverses the tree non-recursively in Inorder using STACK. Initially push NULL onto stack & then set P = root & then repeat the following steps until NULL is poped from stack.
step 1 - proceed down the leftmost path rooted at P. At each node N of the path & push N of the path, push N onto stack & if N has a right child P→right, push P→right onto
step 2 - Pop and process nodes on stack. If Null is popped, then exit. Return to step 1.

```
void posttrav(nodeptr tree)
{ if(tree != NULL)
    { posttrav(tree→left);
      posttrav(tree→right);
      printf("→ %od", tree→data);
    }
}
```

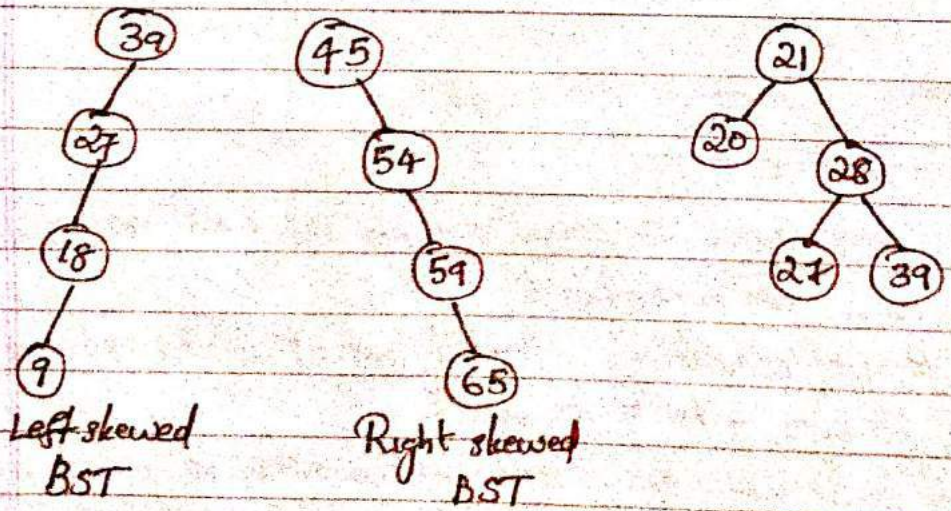
Applications of Trees

→ Trees are used to store simple as well as complex data. ie; structure or a record.
→ used for implementing other types of data structures like hash tables, sets & maps.
→ A self balancing tree, Red-black tree is used in kernel scheduling, to preempt massively multiprocessor computer operating sln use.
→ B trees are used to store tree structures on disc.
→ B trees used for secondary indexes in databases, where the index facilitates a select op^n to answer some range criteria.

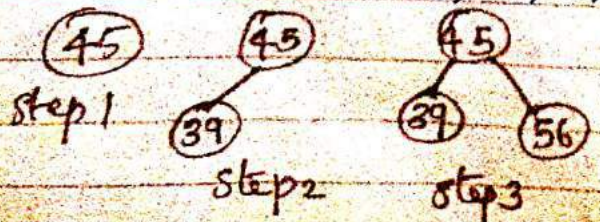
→ Trees are an important data structure used for compiler construction.
→ Trees are also used in db design.
→ Trees are used in file s/m directories.
→ used for inf^n storage & retrieval in symbol tables.

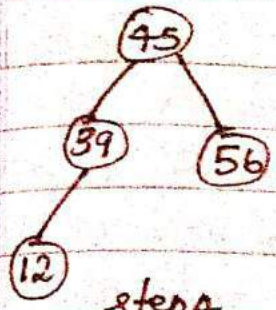
Binary Search Trees

Binary Search Tree is a binary tree with the follow. properties:
→ The left sub-tree of a node N contains values that are less than N's value.
→ The right subtree of a node N contains values that are greater than N's value.
→ Both the left & the right binary trees also satisfy these properties & thus, are binary search trees.

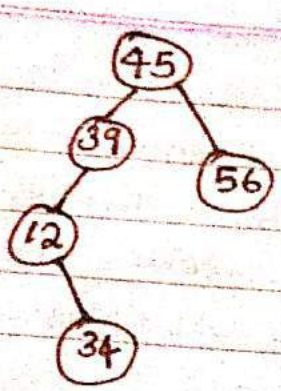


Eg: Create BST using the follo. data elmts:
45, 39, 56, 12, 34, 78, 32, 10, [illegible]

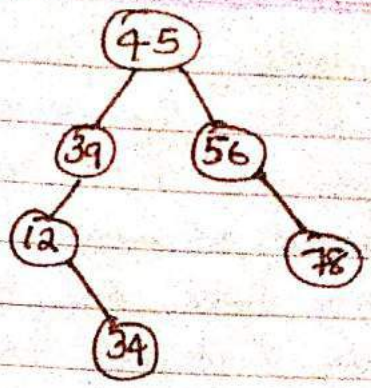




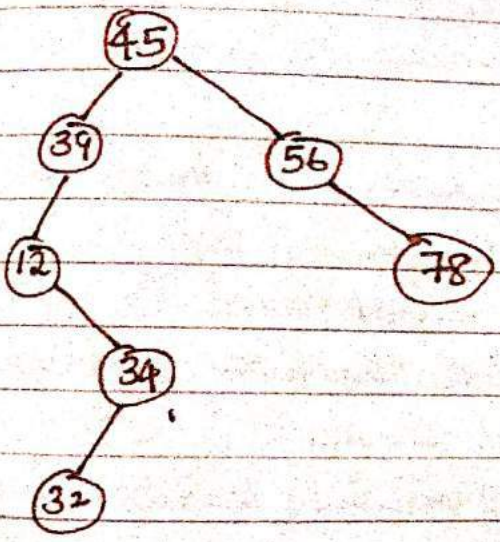
step 4



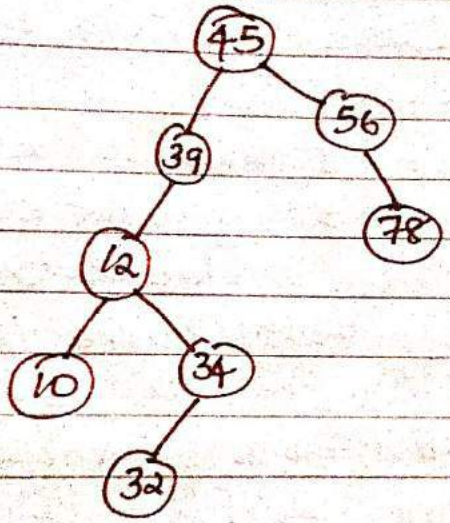
Step 5



step 6

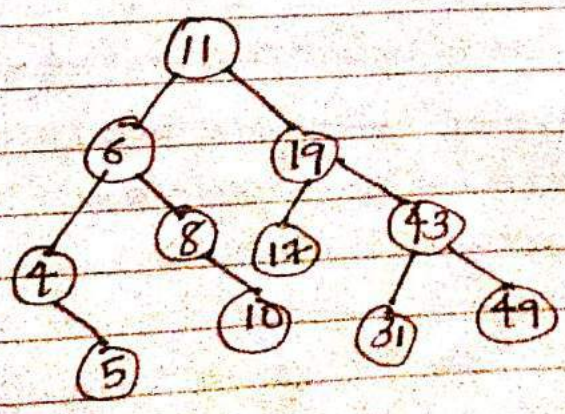


Step 7



Step 8

Eg: Given a sequence of nos: 11, 6, 8, 19, 4, 10, 5, 17, 43, 49, 31
Draw a binary search tree by inserting the above nos.



Searching

To search a given key in Binary Search Tree, we first compare it with root, if the key is present at root, we return root. If key is greater than root's key, we recur for right subtree of root node. Otherwise we recur for left subtree.

Steps: starting from the root,

1. check, whether value in current node & searched value are equal. If so, value is found.

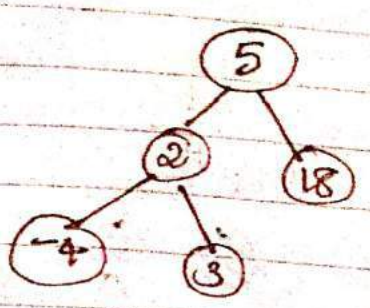
Otherwise,

2. If searched value is less, than the node's value:
   → If current node has no left child, searched value doesn't exist.
   → Otherwise, handle the left child with the same algm.
3. If a new value is greater, than the node's value:
   → If current node has no right child, searched value doesn't exist in BST.
   → otherwise, handle the right child with the same algm.

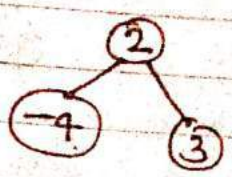
Algorithm:

1. void search(node *tree, int digit)
2. if(tree==NULL) then step3 otherwise go to step 4.
3. printf("The no. doesn't exist").
4. else if(digit==tree->num) then go to step 5 else go to step 6.
5. printf("%d" is found", digit);
6. else if(digit<tree->num) then go to step 7 otherwise go to step 8.
7. search(tree->left, digit);
8. else search(tree->right, digit)
9. End.

Eg: Search for 3 in the tree,



→ 3 is less than 5 go to left child.



3 is greater than 2 then right child (3) found.

Insertion

Steps for Insertion:

→ Check whether root node is present or not (tree available or not). If root is NULL, create root node.

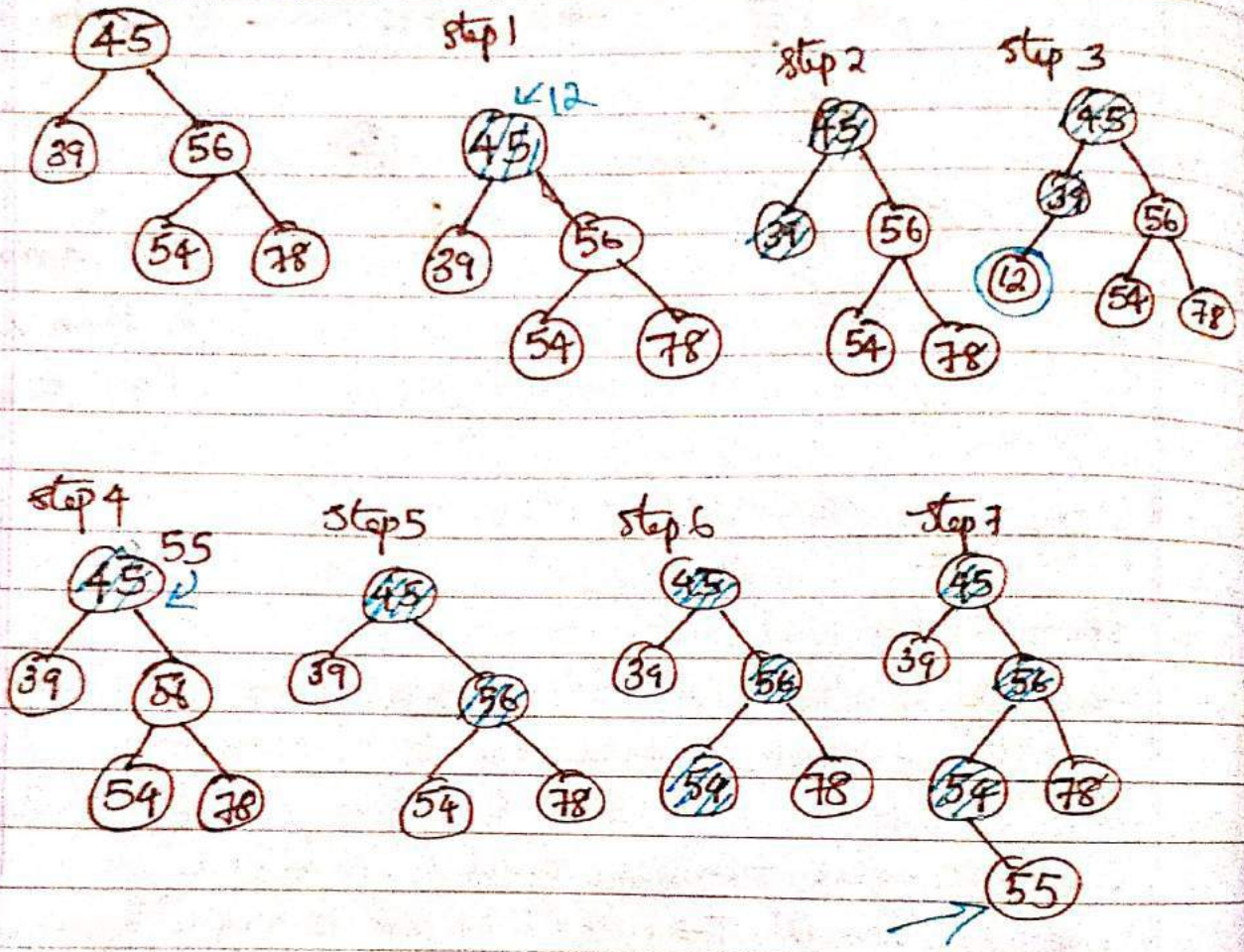
→ If elmt to be inserted is less than the elmt present in the root node, traverse the left sub-tree recursively until we reach T→left / T→right is NULL & place the new node at T→left (key in new node < key in T) / T→right (key in new node > key in T).

→ If the elmt to be inserted is greater than the elmt present in root node, traverse the right sub tree recursively until we reach T→left / T→right is NULL & the place the new node at T→left / T→right.

Algorithm:

```
Insert (TREE, VAL)
Step 1: If Tree = NULL
            Allocate memory for tree
        set tree→DATA = VAL
        set tree→Left = Tree→right = NULL
    else
        If val < Tree→Data
            Insert (Tree→left, val)
        else
            Insert (Tree→Right, val)
        [End of If]
    [End of If]
    Step 2: End.
```

Eg: Inserting nodes with values 12 & 55 in the given binary search tree.



Deletion

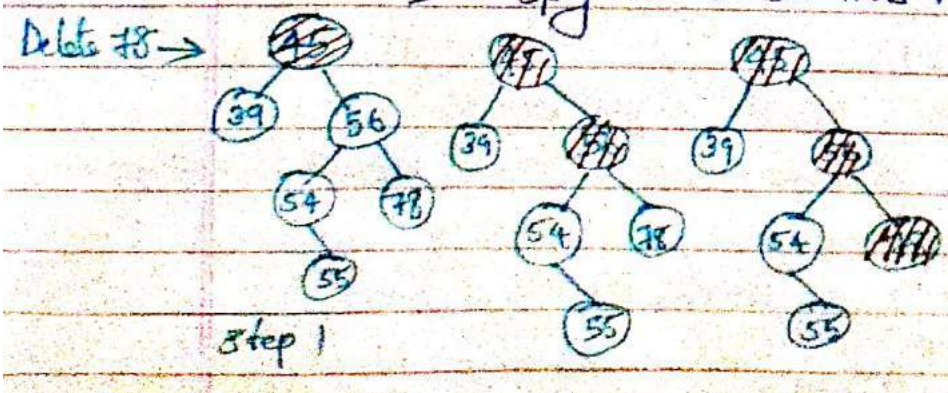
Remove op^n on BST is more complicated than add & search. It can be devided into 2 stages:

→ search for a node to remove,

→ If the node is found, run remove algm.

Case 1: Deleting a Node that has no children.

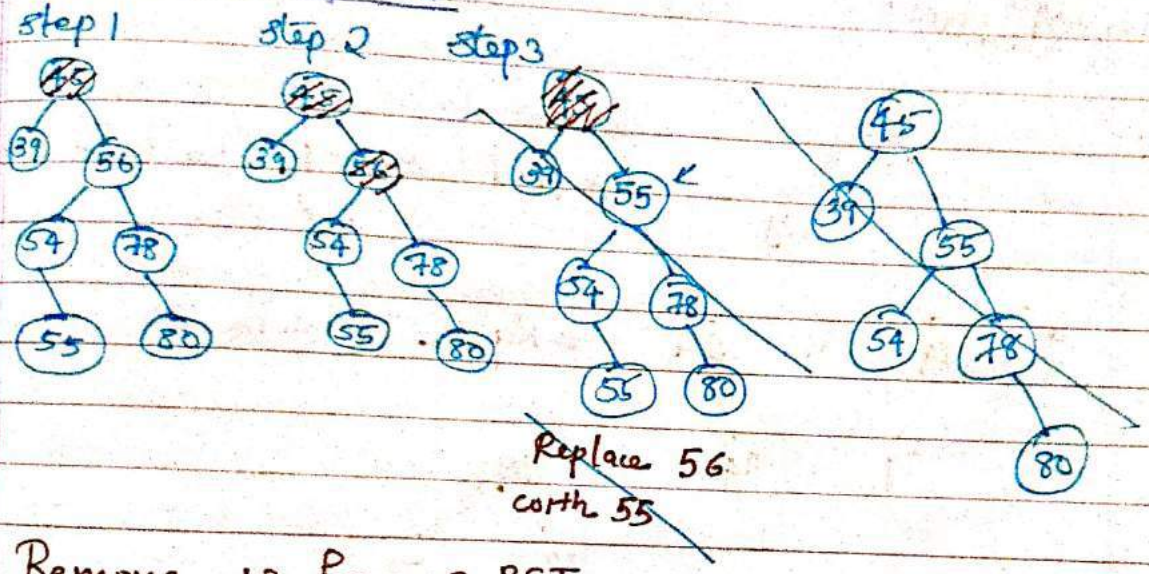
→ simply remove this node without any issue.



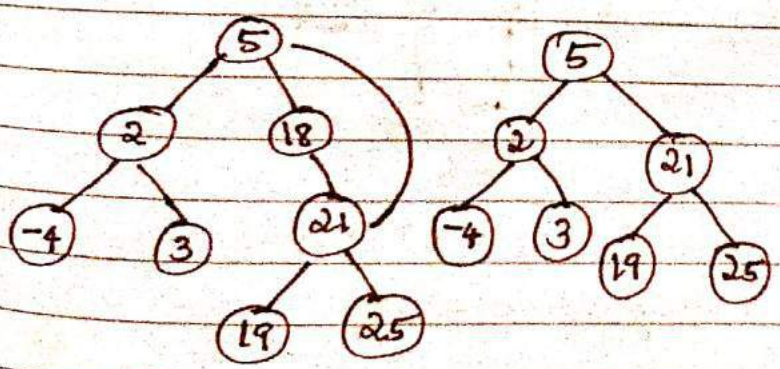
Case 2: Deleting a Node with One Child:

→ Replace the node with its child. If the node is the left child of its parent, the node's child becomes the left child of the node's parent. If the node is the right child of its parent, the node's child becomes the right child of the node's parent.

Deletion of 56.



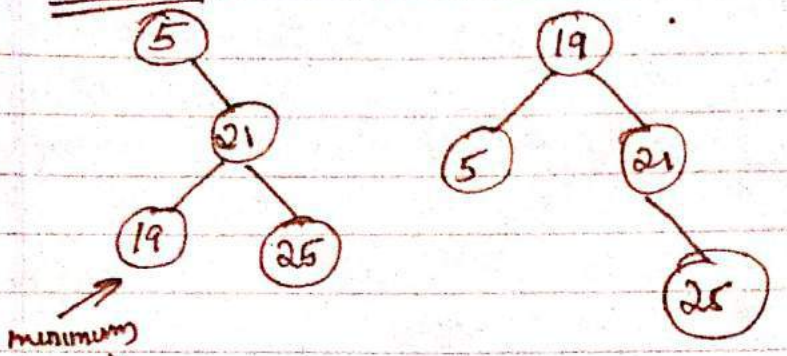
Remove 18 from a BST.



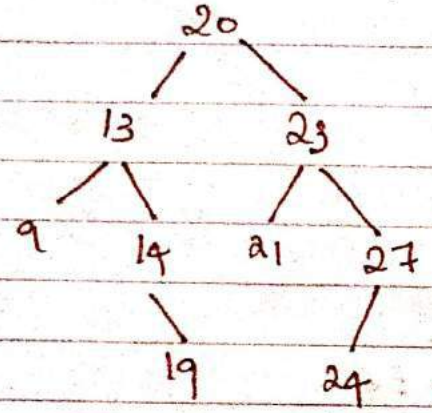
Case 3: Deleting a Node with Two Children:

→ Replace the node's value with its in-order predecessor (largest value in the left sub-tree) or in-order successor (smallest value in the right sub-tree). The in-order predecessor or the successor can be deleted using any of the above cases.

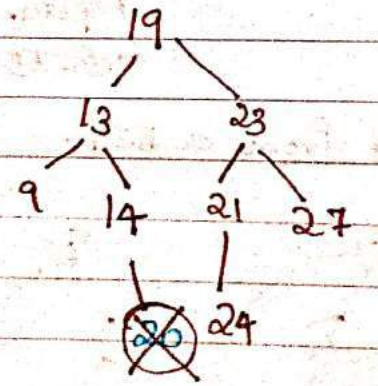
Delete 5



Eg:



Delete 20
→ find largest in the left subtree of 20.
→ Replace 19 with 20.



Algorithm:

```
void deletenode(node *tree, int digit)
{ struct node *r, *q;
  if (tree == NULL)
  { print "tree is empty"
    exit(0)
  }
  if(digit < tree->num)
```

```
deletenode(tree->left, digit);
else if (digit > tree->num)
deletenode (tree->right, digit)
q=tree;
if ((q->right==NULL) && (q->left ==NULL)
  q=NULL;
else if (q->right==NULL)
tree = q->left;
else if (q->left== NULL)
   tree = q->right
else
   delnum(q->left, digit);
   free(q);
}
delnum (int struct node *r, int digit)
{ struct node *q
   if (r->right != NULL)
   delnum (r->right, digit)
   Else
     q->num = r->num;
     q=r;
     r= r->left;
  }
```

Construct binary tree from inorder & preorder traversal:

The follo. procedure demonstrates on how to rebuild tree from given inorder & preorder traversals of a binary tree:

→ Preorder traversal visits Node, left subtree, right subtree recursively.

→ Inorder traversal visits left subtree, node, right subtree recursively.

→ Since we know that the first node in preorder is its root, we can easily locate the root node in the ex. inorder traversal & hence we can obtain left subtree & right subtree from the inorder traversal recursively.

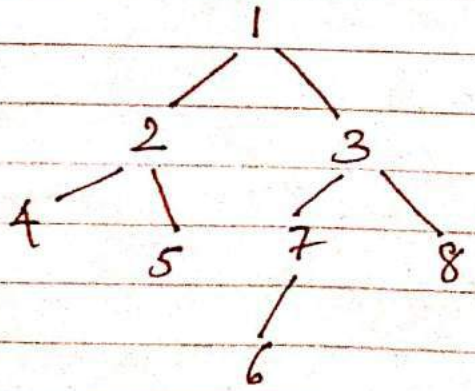
Eg: 1 : in-order : 4 2 5 (1) 6 7 3 8

pre-order : (1) 2 4 5 3 7 6 8

Construct binary tree

Sol^n: From the pre-order array, we know that first elmt is the root. We can find the root in in-order array. Then we can find the left & right sub-trees of the root from in-order array.

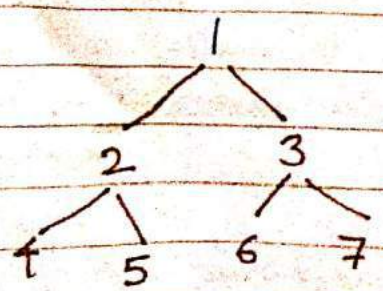
Using the length of left sub-tree, we can identify left & right sub-trees in pre-order array. Recursively we can build up the tree.



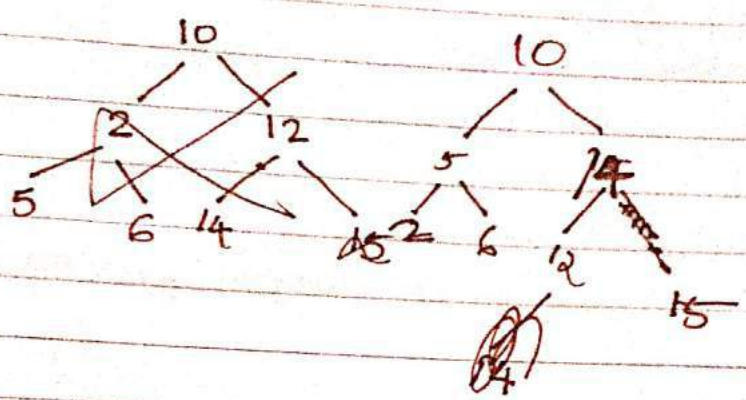
Eg: 2 inorder : {4, 2, 5, 1, 6, 3, 7}

postorder = {4, 5, 2, 6, 7, 3, 1} Construct binary tree

Sol^n:



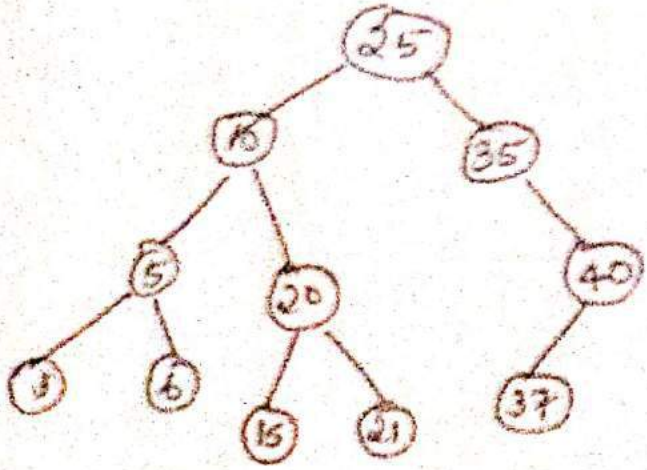
Eg: 3 : inorder = {2, 5, 6, 10, 12, 14, 15}
preorder = {10, 5, 2, 6, 14, 12, 15} Construct binary tree.



Eg: pre order traversal : 1 2 4 8 9 10 11 5 3 6 7
Inorder traversal : 8 4 10 9 11 2 5 1 6 3 7
Construct binary tree:

Deletion in BST

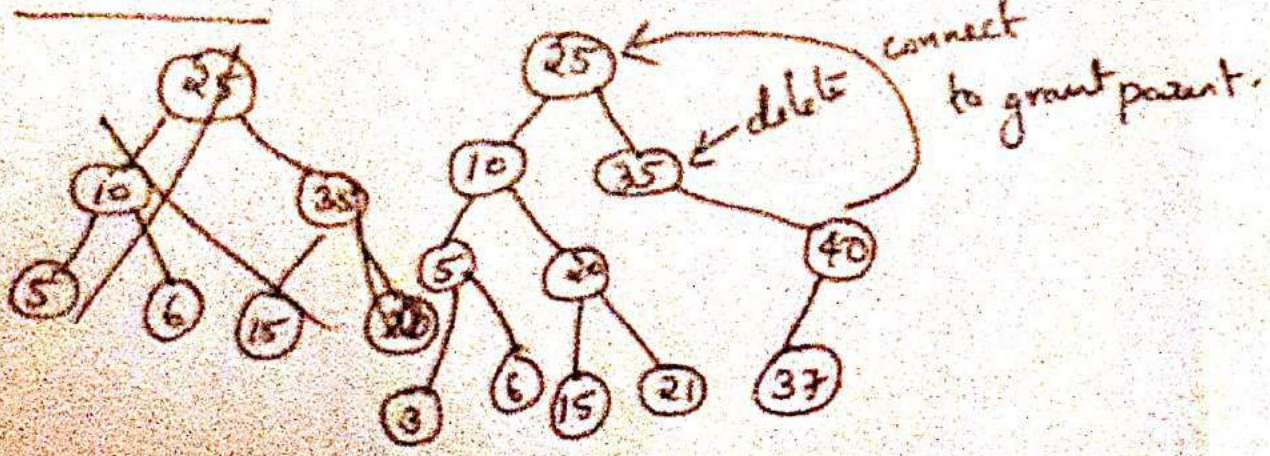
→ Deleting leaf Node



→ simply delete.

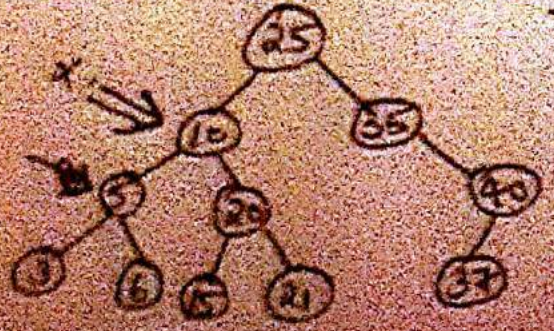
→ Deleting Non leaf
- → 1 child
- → 2 child

* 1 child



2 children

→ exchange → left subtree biggest number



→ exchange subtree smallest number

MODULE V

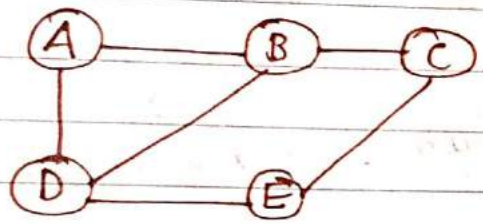
Page No :
Date :

GRAPHS

A graph is an abstract data structure that is used to implement the mathematical concept of graphs. It is a collection of vertices & edges that connect these vertices. Complex relationship can exist.

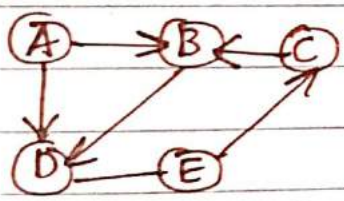
Definition: A graph G is defined as an ordered set(V,E), where V(G) represents the set of vertices & E(G) represents the edges that connect these vertices.

Eg:



undirected graph.

Fig shows a graph with $V(G) = \{A, B, C, D \& E\}$ and $E(G) = \{(A,B), (B,C), (A,D), (B,D), (D,E), (C,E)\}$.



Directed Graph.

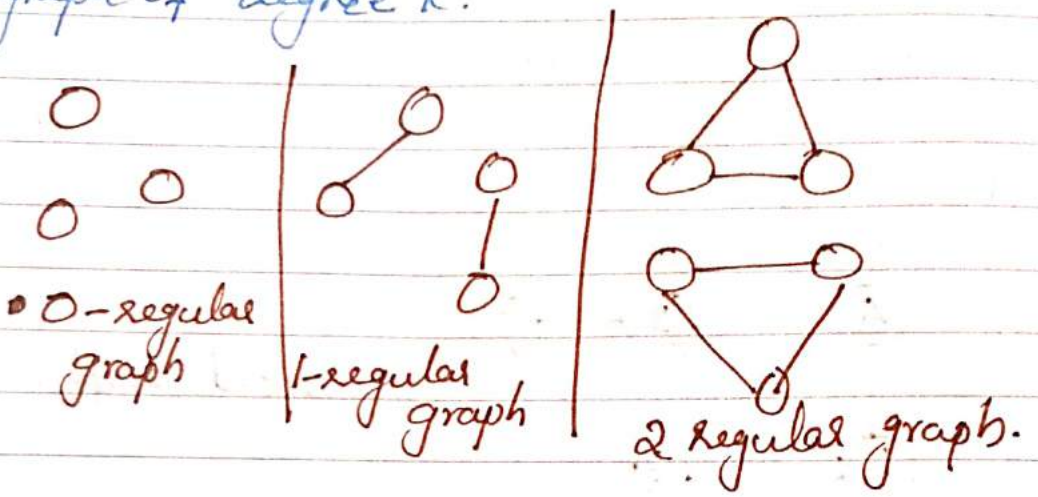
A graph can be directed or undirected graph. In an undirected graph, edges do not have any direction associated with them. In a directed graph, edges form an ordered pair.

Graph Terminology:

Adjacent nodes or neighbours: For every edge, e=(u,v) that connects nodes u & v, the nodes u & v are the end pts & are said to be the adjacent nodes or neighbours.

Degree of node: is the total no. of edges containing the node u. If deg(u)=0, it means that u doesn't belong to any edge & such a node is known as an isolated node.

Regular Graph: It is a graph where each vertex has the same no. of neighbours. ie; every node has the same degree. A regular graph with vertices of degree k is called a k-regular graph or a regular graph of degree k.



Path: A path p written as $P = \{v_0, v_1, v_2, \ldots v_n\}$, of length n from a node u to v is defined as a sequence of (n+1) nodes. Here, $u = v_0$, $v = v_n$ $\{v_{i-1}$ is adjacent to v_i for $i = 1, 2, 3, \ldots n$.

Closed path: A path p is known as a closed path if the edge has the same end-pts. ie; $v_0 = v_n$.

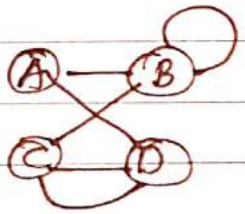
Simple path: A path p is known a simple path if all the nodes in the path are distinct with an exception that v_0 may be equal to v_n. If $v_0 = v_n$ then the path is called a closed simple path.

Cycle: A path in which the first & the last vertices are same.

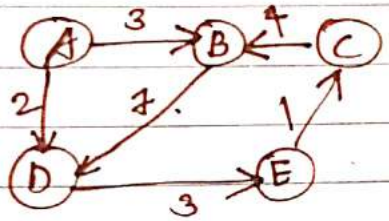
Connected Graph: A graph is said to be connected if for any 2 vertices (u, v) in v - there is a path from u to v.

Complete Graph: A graph is said to be complete if all its nodes are fully connected. ie; there is a

path from one node to every other node in the graph.
~~A graph have~~ A complete graph has $n(n-1)/2$ edges, where n is the no. of nodes in G.
Labelled/weighted graph: A graph is said to be labelled if every edge in the graph is assigned some data. The weight of an edge denoted by w(e) is a positive value which indicates the cost of traversing the edge.
Multi graph: A graph with multiple edges or loops is called a multi graph.



multi-graph



Directed Graphs

An edge of a directed graph is given as an ordered pair (u, v) of nodes in G. For an edge (u, v),
* The edge begins at u & terminates at v.
* u is known as origin or initial pt. of e. v is the destination or terminal point.
* u is the predecessor of v. v is the successor of u.
* Nodes u & v are adjacent to each other.

Out degree of a node: outdeg(u) is the no. of edges that ~~terminate at u.~~ originate at u.
In-degree of a node: indeg(u), is the no. of edges that terminate at u.
Degree of a node: deg(u): sum of in-degree & out-degree of that node. deg(u) = indeg(u) + outdeg(u).

Isolated vertex: A vertex with degree zero. Such a vertex is not an end-pt. of any edge.
Pendant vertex: A vertex with degree as one.
Cut vertex: A vertex which when deleted would disconnect the remaining graph.
Source: A node u is known as a source if it has a +ve out-degree but zero in-degree.
Sink: A node u is known as a sink if it has a +ve in-degree but a zero out-degree.
Reachability: A node v is said to be reachable from node u, iff there exists a path from node u to node v.
Strongly connected directed graph: A graph is said to be strongly connected iff there exists a path b/w every pair of nodes in G.

Representation of Graphs

3 common ways of storing graphs:

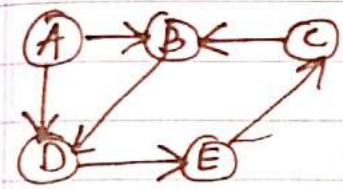
* Sequential repn by using adjacency matrix.
* Linked repn by using adjacency list
* Adjacency multi list which is an extension of linked repn.

Adjacency Matrix Representation:

- Adjacency Matrix is used to represent which nodes are adjacent to one another. 2 nodes are said to be adjacent if there is an edge connecting them.

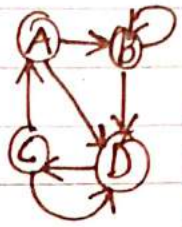
For any graph G, having n nodes, the adjacency matrix will have the dimension n x n.

In an adjacency matrix, the rows & columns are labelled by graph vertices. An entry a_{ij} in the adjacency matrix will contain 1, if vertices v_i & v_j are adjacent to each other. If the nodes are not adjacent, a_{ij} will be set to zero.

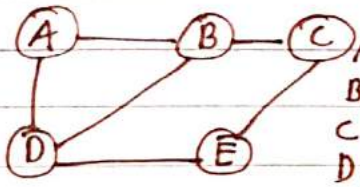


Directed graph

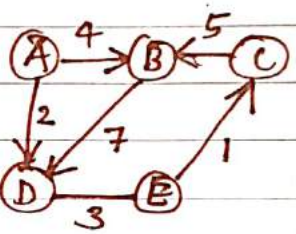
	A	B	C	D	E
A	0	1	0	1	0
B	0	0	0	1	0
C	0	1	0	0	0
D	0	0	0	0	1
E	0	0	1	0	0



	A	B	C	D
A	0	1	0	1
B	0	1	1	1
C	1	0	0	1
D	0	0	1	0



	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	1	0	0	1
D	1	1	0	0	1
E	0	0	1	1	0



	A	B	C	D	E
A	0	4	0	2	0
B	0	0	0	7	0
C	0	5	0	0	0
D	0	0	0	0	3
E	0	0	1	0	0

Features:

→ For a simple graph, the adjacency matrix has 0s on the diagonal.

→ The adjacency matrix of an undirected graph is symmetric.

→ The memory use of an adjacency matrix is $O(n^2)$, where n is the no. of nodes in the graph.

→ No. of 1s in an adjacency matrix is equal to the no. of edges in the graph.

→ The adjacency matrix for a weighted graph contains the weights of the edges connecting the nodes.

Adjacency Multi-list Representation

Graphs can also be represented using multi-lists

Adjacency list Representation

This structure consists of a list of all nodes in G. Every node is in turn linked to its own list that contains the names of all other nodes that are adjacent to it.

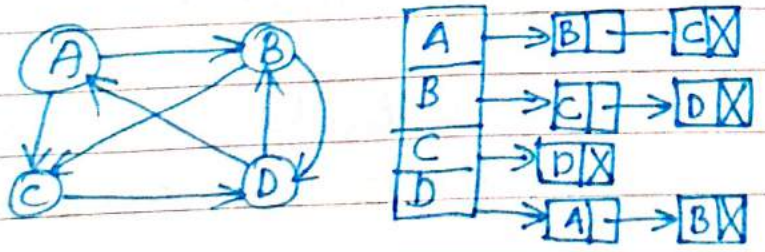
Advantages:

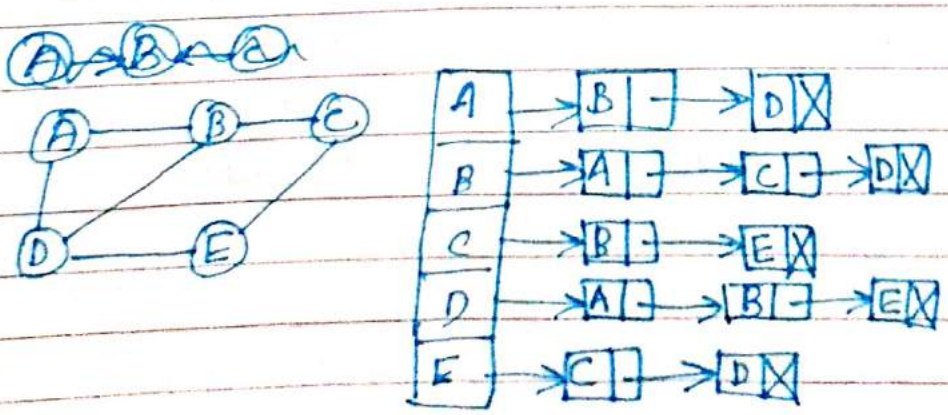
→ Easy to follow & clearly shows adjacent nodes of a particular node

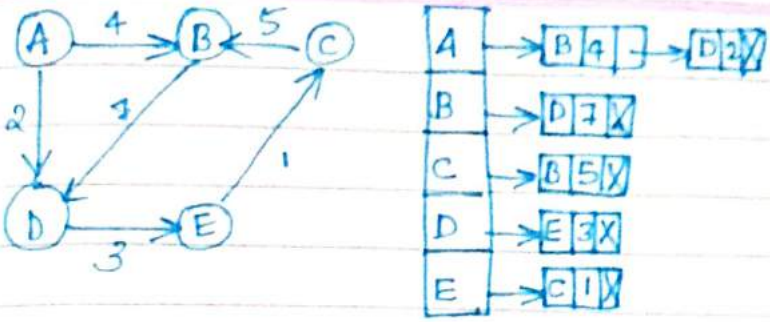
→ used for storing graphs that have a small to moderate no. of edges.

→ Adding new nodes is easy & straight forward.

Graph G & its adjacency list:







For a directed graph, the sum of the lengths of all adjacency lists is equal to the no. of edges in G.

For an undirected graph, the sum of the lengths of all adjacency lists is equal to twice the no. of edges in G b/c an edge (u, v) means an edge from u to v as well as an edge from v to u.

Adjacency lists can also be modified to store weighted graphs.

Adjacency Multi-list Representation:

→ modified version of adjacency lists.

→ Adjacency multilist is an edge based rather than a vertex based repn of graphs.

2 parts → a directory nodes info
→ a set of linked list storing info about edges.

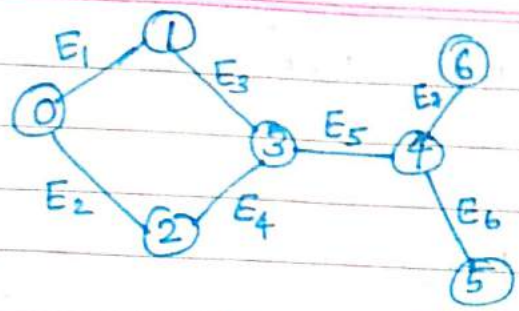
While there is a single entry for each node in the node directory, every node.

V_i : A vertex in the graph that is connected to vertex V_j by an edge.

V_j : A vertex in the graph that is connected to vertex V_i by an edge.

Link i for V_i : A link that points to another node that has an edge incident on V_i

Link j for V_i : A link that points to another node that has an edge incident on V_j.



Edge 1 | 0 | 1 | Edge2 | Edge3

Edge 2 | 0 | 2 | NULL | Edge4

Edge 3 | 1 | 3 | NULL | Edge4

Edge 4 | 2 | 3 | NULL | Edge5

Edge 5 | 3 | 4 | NULL | Edge6

Edge 6 | 4 | 5 | Edge7 | NULL

Edge 7 | 4 | 6 | NULL | NULL

Using adjacency multi-list given above, the adjacency list for vertices :—

Vertex	List of Edges
0	Edge 1, Edge 2
1	Edge 1, Edge 3
2	Edge 2, Edge 4
3	Edge 3, Edge 4, Edge 5
4	Edge 5, Edge 6, Edge 7
5	Edge 6
6	Edge 7

Graph Traversal Algorithms.

→ method of examining the nodes & edges of the graph. 2 methods

- Breadth - first Search
- Depth - first Search

While BFS uses a queue as an auxiliary data structure to store nodes for further processing, the DFS scheme uses stack. Both these algm make use of a variable status. During the exn of the algm every node in the graph will have the variable status

set to 1 or 2, depending on its current state.

status	state of the node	Description
1	Ready	Initial state of the node N
2	waiting	Node N is placed on queue or stack & waiting
3	Processed	Node N has been completely processed.

Breadth First Search Algorithm

⇒ begins at the root node & explores all the neighbouring nodes.

We start examining the node A & then all the neighbours of A are examined. We examine the neighbours of neighbours of A, so on & so forth. This means that we need to track the neighbours of the node & guarantee that every node in the graph is processed & no node is processed more than once. This is accomplished using a queue that will hold the nodes that are waiting for further processing & a variable status to represent the current state of the node.

Step 1: Set status = 1 (ready state)
for each node in Gs
Step 2: Enqueue the starting node A &
set its status = 2 (waiting state)
Step 3: Repeat steps 4 & 5 until Queue is empty
Step 4: Dequeue a node N. Process it & set its status = 3
processed state.
Step 5: Enqueue all the neighbours of N that are in the
ready state (whose status = 1) & set their status = 2
(waiting state)
End of loop
Step 6: exit.

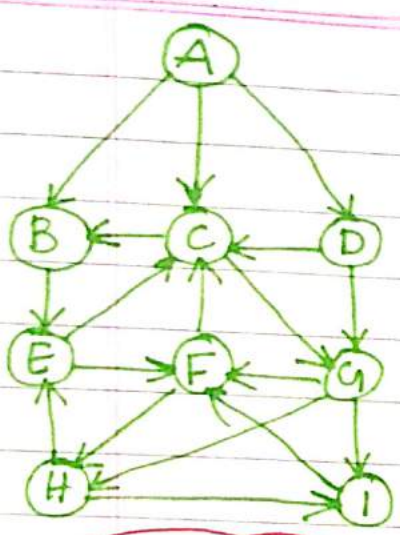


Fig 6.1

Adjacency lists

A : B, C, D
B : E
C : B, G
D : C, G
E : C, F
F : C, H
G : F, H, I
H : E, I
I : F

path P begins at city A & ends where I is encountered.

Algorithm uses 2 arrays: — QUEUE & ORIG. While QUEUE is used to keep track of the origin of each edge. Initially front = rear = -1. The algm as follows:

(a) Add A to QUEUE & add NULL to ORIG.

Front = 0	QUEUE = A
Rear = 0	ORIG = \0

(b) Dequeue a node by setting front = front + 1 (remove front elmt of queue & Enqueue the neighbours of A. Also, add A as the ORIG of its neighbours.

Front = 1	QUEUE = A B C D
rear = 3	ORIG = \0 A A A

c) Dequeue a node by setting Front = front + 1 & enqueue the neighbours of B. Also, add B as the ORIG of its neighbours.

front = 2	QUEUE = A B C D E
rear = 4	ORIG = \0 A A A B

d) Dequeue a node by setting front = front + 1 & enqueue the neighbours of C. Also add C as the ORIG of its neighbours. Note that C has 2 neighbours B & G.

Since B has been already added to the queue & it is not in the ready state, we will not add B & only add G.

front = 3	QUEUE = A B C D E G
rear = 5	ORIG = \0 A A A B C

e) Dequeue a node by setting front = front+1 & enqueue the neighbours of D. Also add D as the ORIG of its neighbours. D has 2 neighbours C & G. Since both of them have already been added to the queue & they are not in the ready state, we will not add them again.

front = 4	Queue = A B C D E G
Rear = 5	ORIG = \0 A A A B C

f) Dequeue a node by setting front = front+1 & enqueue the neighbours of E. Also, add E as the ORIG of its neighbours. Note that E has 2 neighbours C & F. Since C has already been added to the queue & it is not in the Ready state, we will not add C & add only F.

front = 5	QUEUE = A B C D E G F
rear = 6	ORIG = \0 A A A B C E

g) Dequeue a node by setting front = front+1 & enqueue the neighbours of G. Also add G as the ORIG of its neighbours. Note that G has 3 neighbours ~~of~~ F, H & I.

front = 6	QUEUE = A B C D E G F H I
Rear = 9	ORIG = \0 A A A B C E G G

Since F has already been added to the queue, we will add ~~on~~ H & I. As I is our final destination, we stop the exn of this algm as soon as it is encountered & added to the QUEUE. Now backtrack from I using ORIG to find the min. path P. Thus we have path P as A → C → G → I.

Features of BFS Algorithm

→ Space Complexity: All the nodes at a particular level must be saved until their child nodes in the next level have been generated. The space complexity is therefore proportional to the no. of nodes at the depth level of the graph.

$\Rightarrow O(b^d)$ // $b \rightarrow$ branching factor, $d \rightarrow$ depth

→ Time Complexity: In the worst case, BFS has to traverse through all paths to all possible nodes, thus the time complexity of this algm $O(b^d)$.

- can also be expressed as $O(|E| + |V|)$, since every vertex & every edge will be explored in the worst case.

→ Completeness: BFS will find a solution regardless of the kind of graph.

→ Optimality: optimal for a graph that has edges of equal length, since it always returns the result with the fewest edges b/w the start node & the goal node.

Applications of BFS:

→ Finding all connected components of a graph Gr.
→ Finding all nodes within an individual connected component.
→ Finding the shortest path b/w 2 nodes, u & v, of an unweighted graph.
→ Finding the shortest path b/w 2 nodes, u & v, of a weighted graph.

Depth First Search Algorithm

→ by expanding the starting node of G and then going deeper & deeper until the goal node is found or until a node that has no children is encountered.

DFS begins at a starting node A which becomes the current node. Then, it examines each node N along a path P which begins at A. i.e; we process a neighbour of A, then a neighbour of A & so on. During the exn of the algm, if we reach a path that has a node N that has already processed, then we back track to the current node. Otherwise, the unvisited node becomes the current node.

Algorithm:

Step 1: Set status = 1 (ready state) for each node in G.

Step 2: Push the starting node A on the stack & set its status = 2 (waiting state).

Step 3: Repeat steps 4 & 5 until STACK is empty.

Step 4: Pop the top Node N. process it & set its status = 3 (processed state)

Step 5: Push on the stack all the neighbours of N that are in the ready state (whose status = 1) and set their status = 2 (waiting state)

[End of loop]

Step 6: Exit.

Ref Fig 6.1 print all nodes that can be reached from the node H. DFS starting at node H.

a. Push H onto stack

STACK: H

b) Pop & print the top elmt of the stack, that is H. Push all the neighbours of H onto the stack that are in ready state. The stack becomes

print: H STACK: E, I

c) Pop & print top elmt of the stack, i.e; I. Push all the neighbours of F onto the stack.

print I STACK: E, F

d) Pop & print top of the stack i.e; F. Push all the neighbours of F onto the stack.

print: F STACK: E, C

e) Pop & print the top of the stack, i.e; C. Push all the neighbours of C onto the stack that are in the ready state. The stack becomes

print: C Stack: E, B, G

f) Pop & print the top elmt of the stack, i.e; G. Push all the neighbours of G on to the stack that are in ready state. Since there are no neighbours of G that are in the ready state no push opn is performed.

print: G Stack: E, B

g) Pop & print top elmt, B. Push all the neighbours of B onto the stack. Since there are no neighbours of B that are in the ready state no push opn.

print: B Stack: E

h) pop & print top elmt of the stack, E. Push all the neighbours of E onto the stack.

print: E Stack:

Page No :
Date :

Since the stack is empty, DFS of G1 starting at node H is complete & the nodes which were printed are:

H, I, F, C, G1, B, E.

Features of DFS:

Space Complexity: lower than BFS.

Time Complexity: proportional to the no. of edges + no. of vertices in the graphs that are traversed.

$O(|V|+|E|)$

Completeness: Complete algm.

Applications:

* Finding a path b/w 2 specified nodes. of un weighted & weighted node graph.
* Finding whether graph connected or not
* Computing the spanning tree of a connected graph.

Bubble Sort:

Input- an array a[size], n is the no. of elmt currently present in array.
Output- Sorted array
DS - Array.
Algorithm:

1. start
2. i=0
3. while i<n-1
   1. j=0
   2. while j<n-1-i
      1. If a[j]>a[j+1]
         1. temp=a[j]
         2. a[j]=a[j+1]
         3. a[j+1]=temp
      2. Endif
      3. j=j+1
   3 end while
4. i=i+1
4 end while.

Scan I

a[1]	a[2]	a[3]	a[4]	a[5]	j	j+1	i
43	72	10	23	1	1	2	1

1st is small no interchange.

43	72	10	23	1	2	3	1

1st is big interchange

43	10	72	23	1	3	4	1

interchange

43	10	23	72	1	4	5	1

interchange

43	10	23	1	72

Scan II

43	10	23	1	72	1	2	2

interchange

10	43	23	1	72	2	3	2

interchange

10	23	43	1	72	3	4	2

interchange

10	23	1	43	72

Scan III

10	23	1	43	72	1	2	3

no interchange

10	23	1	43	72	2	3	3

interchange

10	1	23	43	72

EXPT. No. Page No.

Scan - 4

10 1 23 43 72 1 2 4

interchange its

1 10 23 43 72

sorted list: 1 10 23 43 72.

Complexity:

$f(n)$ be the no. of comparisons required to sort the given n no. using bubble sort.

$f(n)$ = sum of comparisons required in each scan.

for $i=1$, we required $n-1$ comparison.

" $i=2$ " $n-2$ "

" $i=n-1$ " 1 "

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

$$= \frac{(n-1)\,n}{2} = \frac{n^2-n}{2} = O(n^2)$$

~~Bubble Sort~~: Selection Sort:

Let A be an array of n elmts A[1], A[2], ---- A[n], steps:

1) Find the smallest elmt position in the list A[1], A[2], -- A[n] say k_1 and interchange the values A[1] & $A[k_1]$

2) Find the smallest elmt position in the list A[2], A[3] -- A[n] say k_2 & interchange the values A[2] & $A[k_2]$.

3) Find the smallest elmt posn in the list A[n-1] & A[n] say k_{n-1} & then interchange the values A[n] & $A[k_{n-1}]$

EXPT. No. Page No.

Input: An unsorted array a[], n is the no. of elmts.
Output: a sorted array.
DS: Array

Algorithm:

1: Start
2. i=0
3 while i<n-1 do
1. j=i+1
2. small=i.
3. while(j<n do
1. if a[small]>a[j]
1. small=j
2 end if.
3 j=j+1
4 end while
5. if i!=small
1. temp=a[i]
2. a[i]=a[small]
3. a[small]=temp
6. end if
7. i=i+1
4 end while
5. stop

Step	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	Pos
1.	43	72	10	23	80	1	75	6
	interchange A[1] & A[6]							
	1	72	10	23	80	43	75	
2	1	72	10	23	80	43	75	3
	interchange A[2] & A[3]							
	1	10	72	23	80	43	75	
3.	1	10	72	23	80	43	75	4
	interchange A[3] & A[4]							
	1	10	23	72	80	43	75.	
4	1	10	23	72	80	43	75	6
	interchange A[4] & A[6]							
	1	10	23	43	80	72	75	
5.	1	10	23	43	80	72	75	6
	Interchange A[5] & A[6]							
	1	10	23	43	72	80	75	
6.	1	10	23	43	72	80	75	7
	interchange A[6] & A[7]							
	1	10	23	43	72	75	80	

Sorted List.

Complexity: Let f(n) be the no. of comparison required.

Then $f(n) = (n-1)+(n-2)+\ldots+1$

$= \frac{n(n-1)}{2} = \frac{(n^2-n)}{2} = O(n^2)$

EXPT. No.

Page No.

Insertion Sort:

Scan the array a from a[1] to a[n] & find the smallest elmts a[r], r=1, 2 ... r & insert into its proper position in the previously sorted sub array a[1], a[2] ... a[r].

steps: 1. For i=1 the previously sorted such array is empty ∴ a[1] by itself sorted.

2. For r=2, a[2] is inserted into the proper posⁿ in the previously sorted sub array a[1] ie; a[2] is inserted after or before a[1].

3. For r=3, a[3] is inserted into the proper posⁿ in the previously sorted sub array a[1], a[2]. ie; a[3] is inserted before a[1] or after a[2] or b/w a[1] & a[2].

The above steps are repeated until the last elmt a[n] is inserted into its proper posⁿ in the sorted sub array.

Example:

a[1]= 43 a[2]=72 a[3]=10 a[4]=23 a[5]=80 a[6]=1 a[7]=75

a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	r
[43]	72	10	23	80	1	75	1
↳ sorted sub array							
[43]	72	10	23	80	1	75.	2
	↳ insert elmt in the sorted sub array [43]						
[43	72]	10	23	80	1	75	3
↳ sorted sub array.							
[10	43	72]	23	80	1	75	4
	↳ sorted sub array						
[10	23	43	72]	80	1	75	5
[10	23	43	72	80]	1	75	6
[1	10	23	43	72	80]	75	7
[1	10	23	43	72	75	80]	.

Sorted Array.

EXPT. No. Page No.

Insertion sort (a[], n)
Input: An unsorted array a[], n is the no. of elmts.
Output: a sorted array.
DS: Array
Algorithm:
1. Start
2 i=1
3 while i<n
    1. j=i
    2 while a[j] < a[j-1] and j>0
        1. t=a[j]
        2. a[j]=a[j-1]
        3. a[j-1]=t
        4. j=j-1
    3 end while
    4. i=i+1
4. end while
5. stop

Merge Sort:

a process of combining 2 or more sorted arrays into a single sorted array.

Suppose A[m] and B[n] are 2 sorted arrays. Merging is a process of combining these 2 arrays into a single sorted array C[m+n].

Steps: i) Scan both arrays left to right.

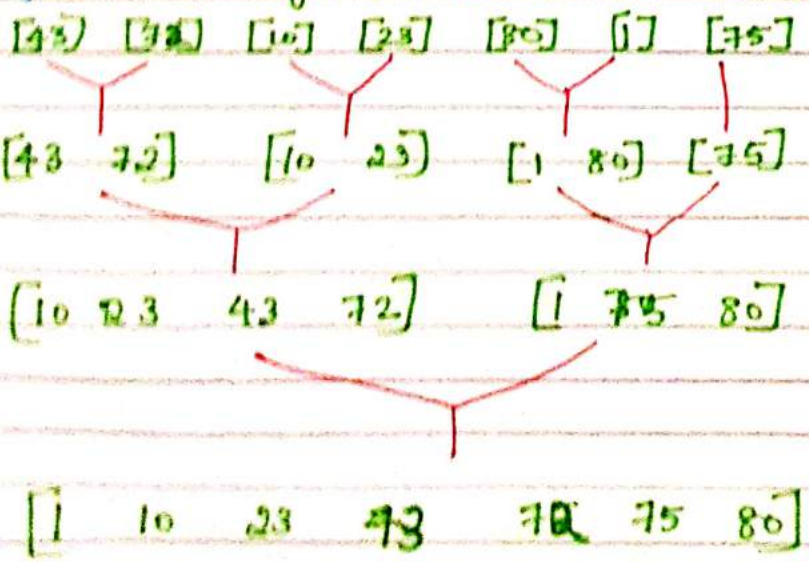
ii) Compare A[1] with B[1]. If A[1] is less than B[1] put A[1] as C[1] and compare A[2] with B[1] and so on, else put B[1] as C[1] and compare A[1] with B[2] & so on.

iii) Repeat step ii until any one array becomes empty.
iv) If A array becomes empty, store all the remaining B array elmts in the same order in the C array. Besides store all the remaining A array elmts in the same order in the B array.

Merge Sort

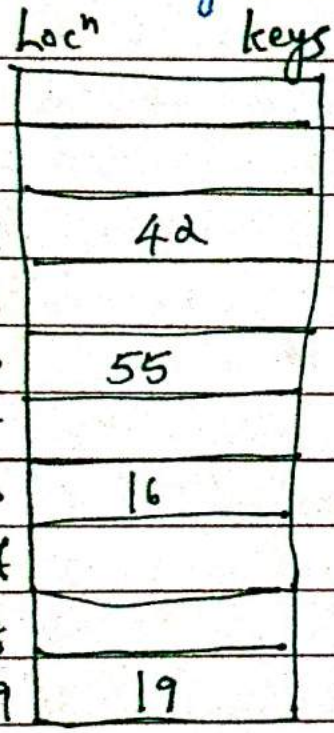
Let a be an array of n no.s. Our aim is to sort this array in ascending order using merge sort. The steps to be followed are:

(i) Divide the array into n sub arrays with size 1.
(ii) Merge adjacent pair of sub arrays [a[1]] and [a[2]], [a[3]] and [a[4]], ... [a[n-1]] and [a[n]]. Now we will get an n/2 sorted sub arrays of size 2 or less.
(iii) Merge adjacent pairs of sub arrays of [a[1]] and [a[2]], [a[3]] and [a[4]], ... [a[n-3]], [a[n-2]] and [a[n-1]], [a[n]]. Now we will get a n/4 sorted sub arrays of size 4 or less.
(iv) Repeat this process until there is only one sub array of size n. Eg: 43 72 10 23 80 1 75



HASHING

→ used for performing insertion, deletion & finds a constant avg time. The implementⁿ of hash table is called hashing. Hash table is an array of some fixed size containing the keys.



Hash Function: A fn that is used to compute the address of the record in hash table is known as hash fn. A hash fn transforms an identifier 'X' into the addr. in the hash table. The address computed is known as the hash address.

Properties of hash fn are:

1. It should be easily computable.
2. minimize the no. of collision.

Various kinds of hash fn:

1. Mid-square method
2. Division Method
3. Folding Method.

EXPT No.

Page No.

1. Mid-Square Method:

Hash address is computed, by squaring the identifies 'x' & then using an appropriate no. of bits from the middle of the square. The no. of bits to be used to obtain the address depends on table size. If the size of hash table is 2^r, then the no. of bits to be selected from the middle of the square will be r.

Mid-square hash address $(x) = r$, where 'r' is the no. of middle digits of x^2. The hash fn for this method is given below.

$h(x) = [M/w(x^2 \bmod w)]$

$M = 2^k$ for $k >= 1$

$W = 2^w$ (where w is the word size)

Eg:

X	X^2	Binary (X^2)	Mid-square (X^2)
1	1	00 000 01	000 (0)
2	4	00 001 00	001 (1)
3	9	00. 010 01	010 (2)
4	16	00 100 00	100 (4)
5	25	00 110 01	110 (6)
6	36	01 001 00	001 (1)
7	49	01 100 01	100 (4)
8	64	10 000 00	000 (0)

There occurs collision for the keys 1 & 8, 7 & 4, 2 & 6.

Division Method:

The hash fn is obtained by using the modulo operation. The identifies X is devided by some prime no. M & the remainder is used as hash addr. of 'x', ie; $f(x) = x \bmod M$

Eg: M = 11 starting at zero & ending at 10.

$f(x) = x \pmod M$, given $M = 11$, $x = 131130$.

$f(x) = 131130 \% 11 = 10$.

EXPT No. Page No.

The key 131130 is inserted into the table at addr. 10.

Folding Method:

Identifier X is partitioned into several parts. These parts are then added together to obtain the hash addr. for X.

$f(x) = (P_1 + P_2 + \ldots + P_n) \bmod$ table size.

Given $x = 123456789$, table size = 10

Break X into 3 parts evenly. Thus $P_1 = 123$, $P_2 = 456$, $P_3 = 789$.

$f(x) = (P_1 + P_2 + P_3) \bmod M.$

$= (123 + 456 + 789) \bmod 10$

$= 8.$

Load factor: $\lambda = n/m$, where 'n' is the no. of identifier

m represents table size.

Collision:

If more than one record points

EXPT. No. Page No.

Quick Sort

i) Choose any no. in the array & name it as partition elmt P. For simplicity we take first elmt a[1] as the partition elmt.

ii) W.r.to the value of P (ie; a[1]) devide the array into 2 partitions such a way that the no.s which are less than P are placed in the left side of P & the no.s which are hughes than P are placed in the right side of P.

iii) The above steps are repeated in the left partition a[1] through a[J-1] and right partition a[j+1] through a[n].

Eg: Arrange the given array of no.s in ascending order.

43, 72, 10, 23, 80, 1, 75

Let the partition elmt P = a[1] = 43

a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
43	72	10	23	80	1	75

↑ ——— ↑

- left scanning over 72 > 43

43	72	10	23	80	1	75

↑ ——————————————— ↑

75 > 43 compare next elmt.

43	72	10	23	80	1	75

1 < 43

right scanning over.

43	1	10	23	80	72	75

1 < 43 so compare next elmt.

Quick Sort Algorithm (Divide & Conquer Algm)

```
QuickSort(A, P, r)
{ if P < r
    q = partition(A, P, r)
    Quicksort(A, P, q-1)
    Quicksort(A, q+1, r)
}

Partition(A, P, r)
{ x = A[r]
  i = P-1
for (j = p to r-1)
  { if A[j] ≤ x)
    { i = i+1
      exchange A[i] with A[j]
    }
  }
  Exchange A[i+1] with A[r]
    Return i+1
}
```

A → Array name
P → lower bound
r → upper bound
x → pivot elmt.
q → position of fixed sorted elmt.

EXPT. No.....

Page No.....

A = | 5 | 7 | 6 | 1 | 3 | 2 | 4 | pivot

1 2 3 4 5

exact pos

left lesser | right greater.

Sublist A_1 Sublist A_2

After 1st partition

| 1 | 3 | 2 | 4 | 7 | 6 | 5 |

1 2 3 4 5 6 7

Linked list

P = 1 r = 7

x = A[r] = 4

i = P − 1 = 1 − 1 = 0

j = 1 to 6

if (A[j] ≤ x) 5 ≤ 4

{ ~~i = i+1~~

~~exchange A[i] with~~

7 ≤ 4

6 ≤ 4

1 ≤ 4 ✓

i = i+1 = 1

exchange A[1] & A[4]

3 ≤ 4 ✓

i = 1+1 = 2

exchange A[2] with A[5]

2 ≤ 4

i = 2+1 = 3

exchange A[3] with A[6]

exit from loop exchange A[4] with A[r]

EXPT. No

Page No

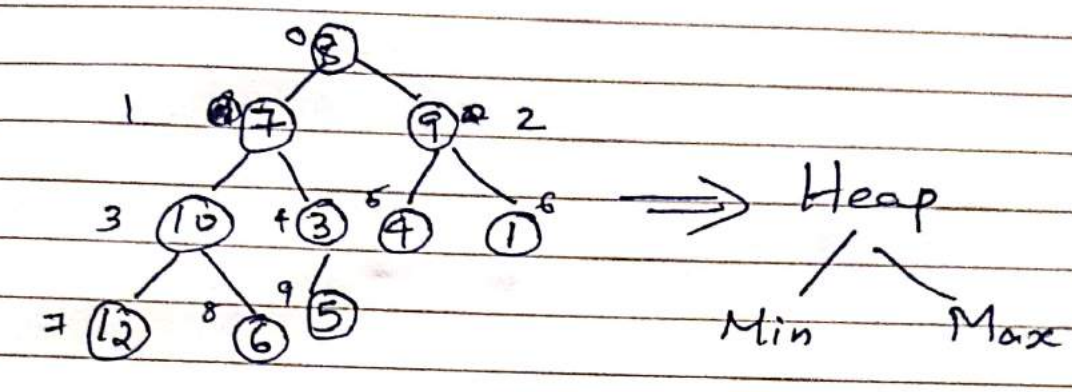
Heap Sort

→ Complete Binary tree

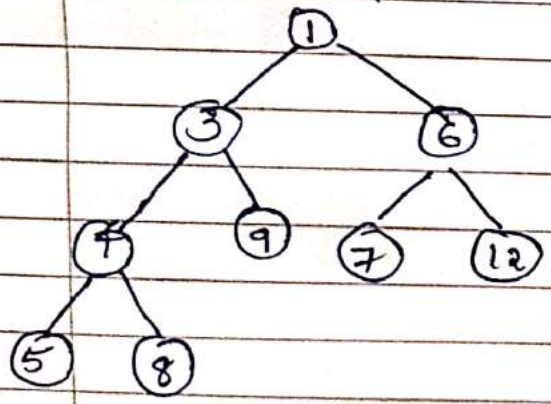
Eg:

8	7	9	10	3	4	1	12	6	5
0	1	2	3	4	5	6	7	8	9

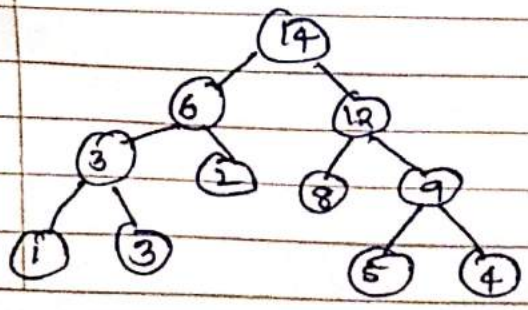
→ Binary tree



Min Heap: value in each node <= children.

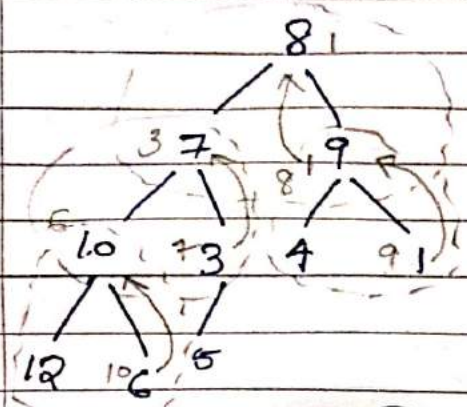


Max heap: >= children

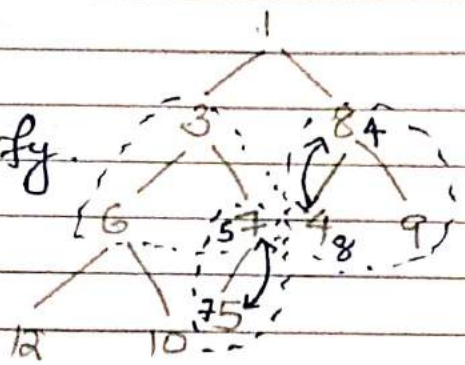


EXPT. No. Page No.

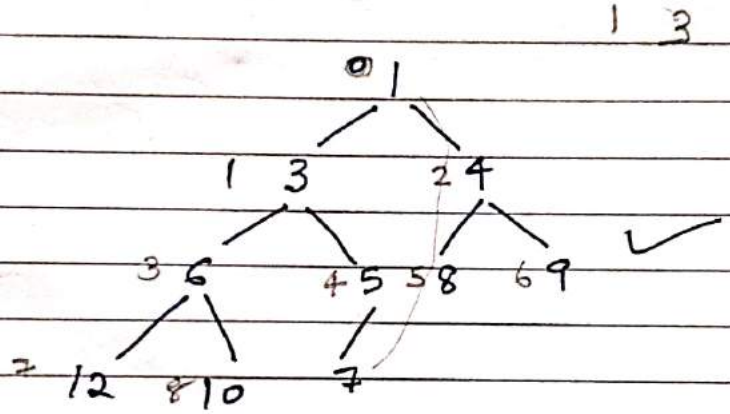
Heapify:

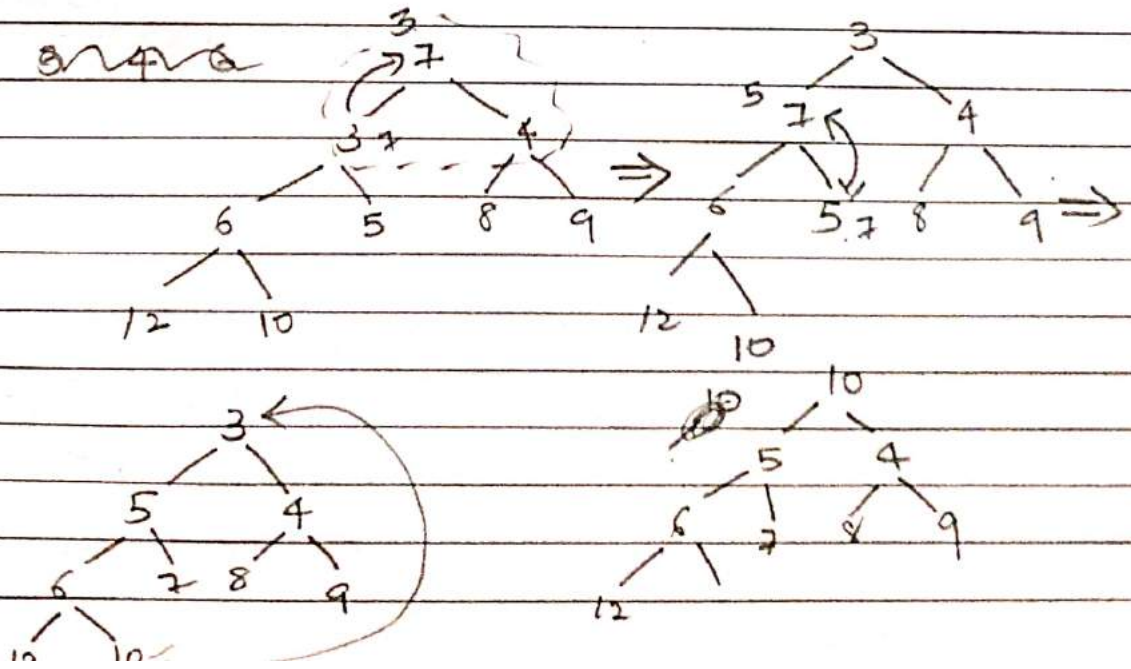


Heapify.



Repeat upto geting min heap.





MODULE VI

Linear search

Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.

Algorithm

Linear Search (Array A, Value x)

Step 1: Set i to 1

Step 2: if i > n then go to step 7

Step 3: if A[i] = x then go to step 6

Step 4: Set i to i + 1

Step 5: Go to Step 2

Step 6: Print Element x Found at index i and go to step 8

Step 7: Print element not found

Step 8: Exit

Pseudocode

```
procedure linear_search (list, value)
  for each item in the list
    if match item == value
      return the item's location
    end if
  end for
end procedure
```

Binary search

Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

```
Procedure binary_search

  A ← sorted array

  n ← size of array

  x ← value to be searched


  Set lowerBound = 1

  Set upperBound = n


  while x not found

    if upperBound < lowerBound

      EXIT: x does not exists.


    set midPoint = lowerBound + ( upperBound - lowerBound ) / 2
```

```
   if A[midPoint] < x
      set lowerBound = midPoint + 1
         if A[midPoint] > x
      set upperBound = midPoint - 1
   if A[midPoint] = x
      EXIT: x found at location midPoint
end while
end procedure
```

Hashing

Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.

Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.

Hashing

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key,value) format.

Hash Function

(1,20)

(2,70)

(42,80)

(4,25)

(12,44)

(14,32)

(17,11)

(13,78)

(37,98)

Sr.No.	Key	Hash	Array Index
1	1	1 % 20 = 1	1
2	2	2 % 20 = 2	2
3	42	42 % 20 = 2	2
4	4	4 % 20 = 4	4
5	12	12 % 20 = 12	12
6	14	14 % 20 = 14	14
7	17	17 % 20 = 17	17
8	13	13 % 20 = 13	13
9	37	37 % 20 = 17	17

Linear Probing

As we can see, it may happen that the hashing technique is used to create an already used index of the array. In such a case, we can search the next empty location in the array by looking into the next cell until we find an empty cell. This technique is called linear probing.

Sr.No.	Key	Hash	Array Index	After Linear Probing, Array Index
1	1	1 % 20 = 1	1	1
2	2	2 % 20 = 2	2	2
3	42	42 % 20 = 2	2	3
4	4	4 % 20 = 4	4	4
5	12	12 % 20 = 12	12	12
6	14	14 % 20 = 14	14	14
7	17	17 % 20 = 17	17	17
8	13	13 % 20 = 13	13	13
9	37	37 % 20 = 17	17	18

Collision Resolution Techniques:

When one or more hash values compete with a single hash table slot, collisions occur. To resolve this, the next available empty slot is assigned to the current hash value. The most common methods are open addressing, chaining, probabilistic hashing, perfect hashing and coalesced hashing technique.

Let's understand them in more detail:

a) Chaining:

This technique implements a linked list and is the most popular collision resolution techniques. Below is an example of a chaining process.

Here, since one slot has 3 elements – {50, 85, 92}, a linked list is assigned to include the other 2 items {85, 92}. When you use the chaining technique, inserting or deleting of items with the hash table is fairly simple and high performing. Likewise, a chain hash table inherits the pros and cons of a linked list. Alternatively, chaining can use dynamic arrays instead of linked lists.

b) Open Addressing:

This technique depends on space usage and can be done with linear or quadratic probing techniques. As the name says, this technique tries to find an available slot to store the record. It can be done in one of the 3 ways –

- **Linear probing** – Here, the next probe interval is fixed to 1. It supports best caching but miserably fails at clustering.
- **Quadratic probing** – the probe distance is calculated based on the quadratic equation. This is considerably a better option as it balances clustering and caching.
- **Double hashing** – Here, the probing interval is fixed for each record by a second hashing function. This technique has poor cache performance although it does not have any clustering issues.

Below are some of the hashing techniques that can help in resolving collision.

c) Probabilistic hashing:

This is memory based hashing that implements caching. When collision occurs, either the old record is replaced by the new or the new record may be dropped. Although this scenario has a risk of losing data, it is still preferred due to its ease of implementation and high performance.

d) Perfect hashing:

When the slots are uniquely mapped, there is very less chances of collision. However, it can be done where there is a lot of spare memory.

e) Coalesced hashing:

This technique is a combo of open address and chaining methods. A chain of items are stored in the table when there is a collision. The next available table space is used to store the items to prevent collision.

CONTENT BEYOND SYLLABUS

Self-organizing list

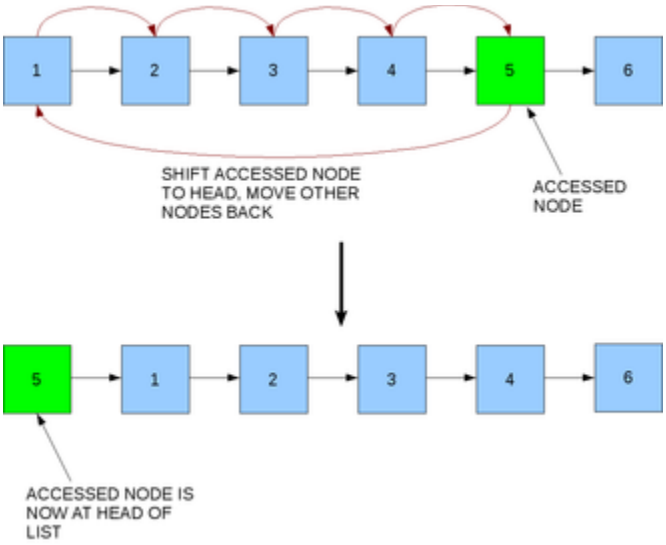
A **self-organizing list** is a list that reorders its elements based on some self-organizing heuristic to improve average access time. The aim of a self-organizing list is to improve efficiency of linear search by moving more frequently accessed items towards the head of the list. A self-organizing list achieves near constant time for element access in the best case. A self-organizing list uses a reorganizing algorithm to adapt to various query distributions at runtime.

Techniques for rearranging nodes

While ordering the elements in the list, the access probabilities of the elements are not generally known in advance. This has led to the development of various heuristics to approximate optimal behavior. The basic heuristics used to reorder the elements in the list are:

Move to front method (MTF)

This technique moves the element which is accessed to the head of the list. This has the advantage of being easily implemented and requiring no extra memory. This heuristic also adapts quickly to rapid changes in the query distribution. On the other hand, this method may prioritize infrequently accessed nodes—for example, if an uncommon node is accessed even once, it is moved to the head of the list and given maximum priority even if it is not going to be accessed frequently in the future. These 'over rewarded' nodes destroy the optimal ordering of the list and lead to slower access times for commonly accessed elements. Another disadvantage is that this method may become too flexible leading to access patterns that change too rapidly. This means that due to the very short memories of access patterns even an optimal arrangement of the list can be disturbed immediately by accessing an infrequent node in the list.

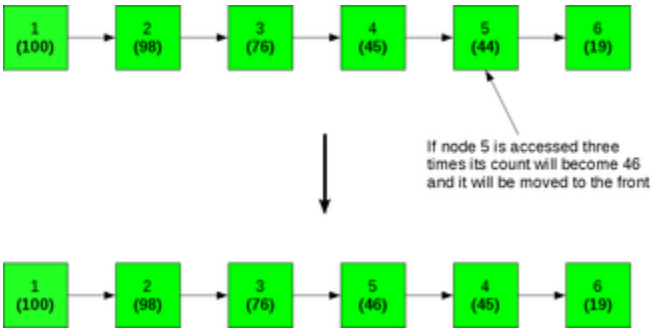


If the 5th node is selected, it is moved to the front

At the t-th item selection:
  if item i is selected:
    move item i to head of the list

Count method

In this technique, the number of times each node was searched for is counted i.e. every node keeps a separate counter variable which is incremented every time it is called. The nodes are then rearranged according to decreasing count. Thus, the nodes of highest count i.e. most frequently accessed are kept at the head of the list. The primary advantage of this technique is that it generally is more realistic in representing the actual access pattern. However, there is an added memory requirement, that of maintaining a counter variable for each node in the list. Also, this technique does not adapt quickly to rapid changes in the access patterns. For example: if the count of the head element say A is 100 and for any node after it say B is 40, then even if B becomes the new most commonly accessed element, it must still be accessed at least (100 - 40 = 60) times before it can become the head element and thus make the list ordering optimal.



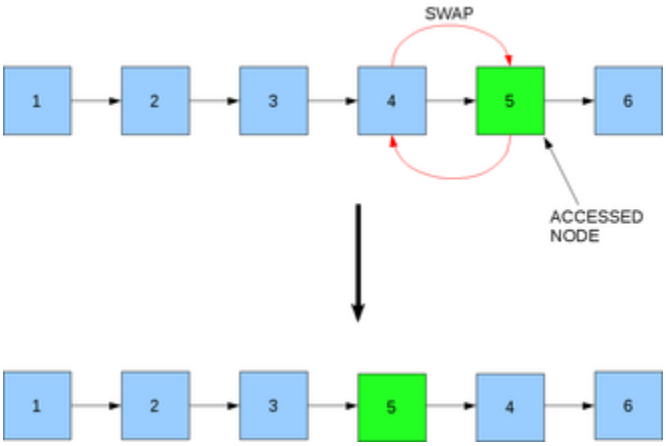
If the 5th node in the list is searched for twice, it will be swapped with the 4th

init: count(i) = 0 for each item i
At t-th item selection:
  if item i is searched:
    count(i) = count(i) + 1
    rearrange items based on count

Transpose method

This technique involves swapping an accessed node with its predecessor. Therefore, if any node is accessed, it is swapped with the node in front unless it is the head node, thereby increasing its priority. This algorithm is

again easy to implement and space efficient and is more likely to keep frequently accessed nodes at the front of the list. However, the transpose method is more cautious. i.e. it will take many accesses to move the element to the head of the list. This method also does not allow for rapid response to changes in the query distributions on the nodes in the list.



If the 5th node in the list is selected, it will be swapped with the 4th

At the t-th item selection:

if item i is selected:

if i is not the head of list:

swap item i with item (i - 1)

Segment tree

In computer science, a **segment tree,** also known as a statistic tree, is a tree data structure used for storing information about intervals, or segments. It allows querying which of the stored segments contain a given point. It is, in principle, a static structure; that is, it's a structure that cannot be modified once it's built. A similar data structure is the interval tree.

A segment tree for a set I of n intervals uses $O(n \log n)$ storage and can be built in $O(n \log n)$ time. Segment trees support searching for all the intervals that contain a query point in $O(\log n + k)$, k being the number of retrieved intervals or segments.[1]

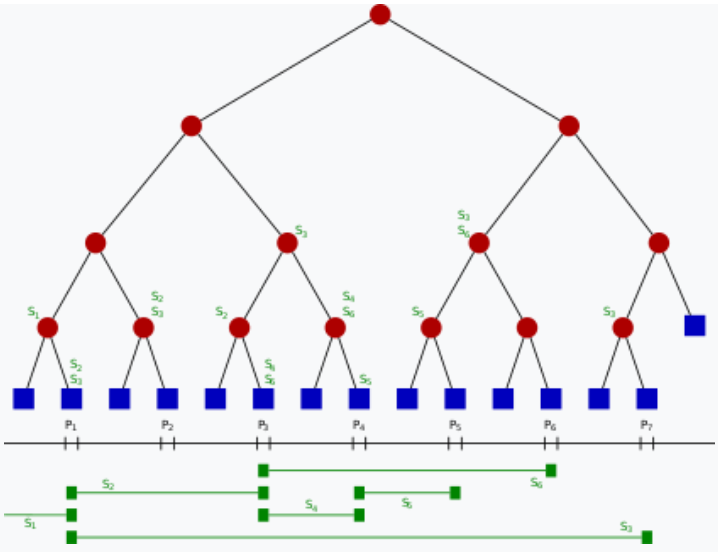
Applications of the segment tree are in the areas of computational geometry, and geographic information systems.

The segment tree can be generalized to higher dimension spaces.

Structure description

Let S be a set of intervals, or segments. Let p_1, p_2, ..., p_m be the list of distinct interval endpoints, sorted from left to right. Consider the partitioning of the real line induced by those points. The regions of this partitioning are called *elementary intervals*. Thus, the elementary intervals are, from left to right:

> That is, the list of elementary intervals consists of open intervals between two consecutive endpoints p_i and p_{i+1}, alternated with closed intervals consisting of a single endpoint. Single points are treated themselves as intervals because the answer to a query is not necessarily the same at the interior of an elementary interval and its endpoints.



Graphic example of the structure of the segment tree. This instance is built for the segments shown at the bottom.

Given a set I of intervals, or segments, a segment tree T for I is structured as follows:

- T is a binary tree.
- Its leaves correspond to the elementary intervals induced by the endpoints in I, in an ordered way: the leftmost leaf corresponds to the leftmost interval, and so on. The elementary interval corresponding to a leaf v is denoted Int(v).
- The internal nodes of T correspond to intervals that are the union of elementary intervals: the interval Int(N) corresponding to node N is the union of the intervals corresponding to the leaves of the tree rooted at N. That implies that Int(N) is the union of the intervals of its two children.
- Each node or leaf v in T stores the interval Int(v) and a set of intervals, in some data structure. This canonical subset of node v contains the intervals $[x, x']$ from I such that $[x, x']$ contains Int(v) and does not contain Int(parent(v)). That is, each node in T stores the segments that span through its interval, but do not span through the interval of its parent.

Multigraph

In graph theory, a **multigraph** is a graph which is permitted to have multiple edges (also called *parallel edges*), that is, edges that have the same end nodes. Thus two vertices may be connected by more than one edge.

There are two distinct notions of multiple edges:

- *Edges without own identity*: The identity of an edge is defined solely by the two nodes it connects. In this case, the term "multiple edges" means that the same edge can occur several times between these two nodes.
- *Edges with own identity*: Edges are primitive entities just like nodes. When multiple edges connect two nodes, these are different edges.

A multigraph is different from a hypergraph, which is a graph in which an edge can connect any number of nodes, not just two.

Undirected multigraph (edges without own identity)[edit]

A multigraph G is an ordered pair $G := (V, E)$ with

- V a set of *vertices* or *nodes*,
- E a multiset of unordered pairs of vertices, called *edges* or *lines*.

Undirected multigraph (edges with own identity)[edit]

A multigraph G is an ordered triple $G := (V, E, r)$ with

- V a set of *vertices* or *nodes*,
- E a set of *edges* or *lines*,
- $r : E \to \{\{x,y\} : x, y \in V\}$, assigning to each edge an unordered pair of endpoint nodes.

Some authors allow multigraphs to have loops, that is, an edge that connects a vertex to itself,[2] while others call these **pseudographs**, reserving the term multigraph for the case with no loops.[3]

Directed multigraph (edges without own identity)[edit]

A **multidigraph** is a directed graph which is permitted to have *multiple arcs*, i.e., arcs with the same source and target nodes. A multidigraph G is an ordered pair $G := (V, A)$ with

- V a set of *vertices* or *nodes*,

- *A* a multiset of ordered pairs of vertices called *directed edges*, *arcs* or *arrows*.

A **mixed multigraph** $G := (V, E, A)$ may be defined in the same way as a mixed graph.

Directed multigraph (edges with own identity)[edit]

A multidigraph or quiver G is an ordered 4-tuple $G := (V, A, s, t)$ with

- V a set of *vertices* or *nodes*,
- A a set of *edges* or *lines*,
- , assigning to each edge its source node,
- , assigning to each edge its target node.

This notion might be used to model the possible flight connections offered by an airline. In this case the multigraph would be a directed graph with pairs of directed parallel edges connecting cities to show that it is possible to fly both *to* and *from* these locations.

In category theory a small category can be defined as a multidigraph (with edges having their own identity) equipped with an associative composition law and a distinguished self-loop at each vertex serving as the left and right identity for composition. For this reason, in category theory the term *graph* is standardly taken to mean "multidigraph", and the underlying multidigraph of a category is called its **underlying digraph**.

Labeling[edit]

Multigraphs and multidigraphs also support the notion of graph labeling, in a similar way. However there is no unity in terminology in this case.

The definitions of **labeled multigraphs** and **labeled multidigraphs** are similar, and we define only the latter ones here.

Definition 1: A labeled multidigraph is a labeled graph with *labeled* arcs.

Formally: A labeled multidigraph G is a multigraph with *labeled* vertices and arcs. Formally it is an 8-tuple where

- V is a set of vertices and A is a set of arcs.
- and are finite alphabets of the available vertex and arc labels,

- and are two maps indicating the *source* and *target* vertex of an arc,

- and are two maps describing the labeling of the vertices and arcs.

Definition 2: A labeled multidigraph is a labeled graph with multiple *labeled* arcs, i.e. arcs with the same end vertices and the same arc label (note that this notion of a labeled graph is different from the notion given by the article graph labeling).