


DATA STRUCTURE

Prof. Durgesh Pant

Title	Data Structure
Formatting and Typesetting	
Unit I, Unit II, Unit III, Unit IV, Unit V, Unit VI, Unit VII, Unit VIII, Unit IX, Unit X, Unit XI, Unit XII, Unit XIII, Unit XIV, Unit XV, Unit XVI	Prof. Durgesh Pant, Director School of CS & IT, Uttarakhand Open University, Dehradun
 Uttarakhand Open University, 2016 © Uttarakhand Open University, 2016. This work by Uttarakhand Open University is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 4.0 International License. It is attributed to the sources marked in the References, Article Sources and Contributors section.	

Unit I: Introduction to Data Structure	5
1.1 What is Data Structure	5
1.2 Methods of Interpreting bit setting	5
1.3 Types of Data Structure	8
1.4 Problems	11
Unit II: Introduction to Algorithms	13
2.1 Introduction to algorithms	13
2.2 Time Complexity	15
2.3 Recurrence	17
2.4 Problems	19
Unit III: Linear Data Structure	20
3.1 Linear Data Structure	20
3.2 Introduction to Stack	20
3.3 Introduction to Queue	25
3.4 Problems	32
Unit IV: Linked List	34
4.1 Linked Lists	34
4.2 Inserting and Removing Nodes from a list	34
4.3 Linked Implemented of Stacks	37
4.4 Getnode and Freenode Operation	38
4.5 Linked Implemented of Queue	39
4.6 List Implementation of Priority Queue	40
4.7 Header Nodes	41
4.8 Circular Lists	42
4.9 Doubly linked list	43
4.10 Problems	44
Unit V: Sorting	45
5.1 Introduction to Sorting	45
5.2 Sink Sort	45
5.3 Selection Sort	46
5.4 Merge Sort	47

5.5 Quick Sort	49
5.6 Radix Sort	50
5.7 Problems	51
Unit VI: Searching	52
6.1 Introduction to Searching	52
6.2 Linear Search	52
6.3 Binary Search	54
6.4 Problems	56
Unit VII: Representation and Traversal	58
7.1 Representation and Traversal	58
7.2 Königsberg Bridge Problem	59
7.3 Problems	64
Unit VIII: Basic Algorithms	65
8.1 Basic Algorithms	65
8.2 Minimum Spanning Tree	66
8.3 Single Source Shortest Path	68
8.4 Problems	71
Unit IX: Binary Tree	72
9.1 Binary Tree	72
9.2 Array Representation of Binary Tree	75
9.3 Linked Representation of Binary Tree	77
Unit X: Heap Sort	81
10.1 Heap Sort	81
10.2 Problems	87
Unit XI: Search Tree	89
11.1 AVL-Tree	89
11.2 B-Tree	96
11.3 Problems	104
Unit XII: Tables	105
12.1 Hashing Techniques	105
11.2 Problems	116

Unit XIII: Sets	117
13.1 Introduction	117
13.2 Problems	121
Unit XIV: String Algorithm	122
14.1 String Algorithm	122
14.2 String Copy	123
14.3 Pattern Matching	124
14.4 Problems	129
Unit XV: Program Development	131
15.1 Life Cycle	131
15.2 Code Designing	132
15.3 Coding	133
15.4 Programming Style	133
15.5 Problems	134
Unit XVI: Program Testing and Verification	126
16.1 Testing Method	135
16.2 Verification Procedure	135
16.3 Problems	140

UNIT I: INTRODUCTION TO DATA STRUCTURE

1.1 What is Data Structure

INTRODUCTION

It is important for every Computer Science student to understand the concept of *Information* and how it is organized or how it can be utilized.

What is Information?

If we arrange some data in an appropriate sequence, then it forms a Structure and gives us a meaning. This meaning is called *Information*. The basic unit of Information in Computer Science is a bit, Binary Digit.

So, we found two things in Information: One is *Data* and the other is *Structure*.

What is Data Structure?

1. A *data structure* is a systematic way of organizing and accessing data.
2. A *data structure* tries to structure data!
 - Usually more than one piece of data
 - Should define legal operations on the data
 - The data might be grouped together (e.g. in an linked list)
3. When we define a data structure we are in fact creating a new data type of our own.
 - I.e. using predefined types or previously user defined types.

Such new types are then used to reference variables type within a program

1.2 Methods of Interpreting bit setting

Why Data Structures?

1. Data structures study how data are stored in a computer so that operations can be implemented efficiently
2. Data structures are especially important when you have a large amount of information
3. Conceptual and concrete ways to organize data for efficient storage and manipulation.

Methods of Interpreting bit Setting

1. Binary Number System
 - Non Negative
 - Negative

- Ones Complement Notation
 - Twos Complement Notation
2. Binary Coded Decimal
 3. Real Number
 4. Character String

Non-Negative Binary System

In this System each bit position represents a power of 2. The right most bit position represents 2^0 which equal 1. The next position to the left represents $2^1 = 2$ and so on. An Integer is represented as a sum of powers of 2. A string of all 0s represents the number 0. If a 1 appears in a particular bit position, the power of 2 represented by that bit position is included in the Sum. But if a 0 appears, the power of 2 is not included in the Sum. For example 10011, the sum is

Ones Complement Notation

Negative binary number is represented by ones Complement Notation. In this notation we represent a negative number by changing each bit in its absolute value to the opposite bit setting. For example, since 001001100 represent 38, 11011001 is used to represent -38. The left most number is reserved for the sign of the number. A bit String Starting with a 0 represents a positive number, where a bit string starting with a 1 represents a negative number.

Twos Complement Notation

In Twos Complement Notation is also used to represent a negative number. In this notation 1 is added to the Ones Complement Notation of a negative number. For example, since 11011001 represents -38 in Ones Complement Notation 11011010 used represent -38 in Twos Complement Notation

Binary Coded Decimal

In this System a string of bits may be used to represent integers in the Decimal Number System. Four bits can be used to represent a Decimal digit between 0 and 9 in the binary notation. A string of bits of arbitrary length may be divided into consecutive sets of four bits. With each set representing a decimal digit. The string then represents the number that is formed by those

decimal digits in conventional decimal notation. For example, in this system the bit string 00110101 is separated into two strings of four bits each: 0011 and 0101. The first of these represents the decimal digit 3 and the second represents the decimal 5, so that the entire string represents the integer 35.

In the binary coded decimal system we use 4 bits, so these four bits represent sixteen possible states. But only 10 of those sixteen possibilities are used. That means, whose binary values are 10 or larger, are invalid in Binary Coded Decimal System.

Real Number

The Floating Point Notation use to represent Real Numbers. The key concept of floating-point notation is Mantissa, Base and Exponent. The base is usually fixed and the Mantissa and the Exponent vary to represent different Real Number. For Example, if the base is fixed at 10, the number -235.47 could be represented as $-23547 * 10^{-2}$. The Mantissa is 23547 and the exponent is -2. Other possible representations are $-.23547 * 10^3$ and $-235.47 * 10^0$

In the floating-point notation a real number is represented by a 32-bit string. Including in 32-bit, 24-bit use for representation of Mantissa and remaining 8-bit use for representation of Exponent. Both the mantissa and the exponent are twos complement binary Integers. For example, the 24-bit twos complement binary representation of -23547 is 11111111010010000000101, and the 8-bit twos complement binary representation of -2 is 11111110. So the representation of 235.47 is 111111110100100000001011111110. It can be used to represent extremely large or extremely small absolute values.

Character Strings

In computer science, information is not always interpreted numerically. Item such as names, address and job title must also be represented in some fashion with in computer. To enable the representation of such nonnumeric objects, still another method of interpreting bit strings is necessary. Such information is usually represented in character string form. For example, in some computers, the eight bits 11000000 is used to represent the character "A" and 11000001 for character "B" and another for each character that has a representation in a particular machine. So, the character string "AB" would be represented by the bit string 1100000011000001.

1.3 Type of Data Structure

The assignment of bit string to character may be entirely arbitrary, but it must be adhered to consistently. It may be that some convenient rule is used in assigning bit string to character. The number of bits varies computer wise used to represent a character.

Some computers are use 7-bit (therefore allow up to 128 possible characters), some computers are use 8-bits (up to 256 character), and some use 10-bits (up to 1024 possible characters). The number of bits necessary to represent a character in a particular computer is called the **byte size** and a group of bits that number is called a **byte**.

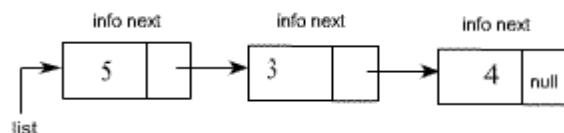
Array

In computer programming, a group of homogeneous elements of a specific data type is known as an array , one of the simplest data structures. Arrays hold a series of data elements, usually of the same size and data type. Individual elements are accessed by their position in the array. The position is given by an index, which is also called a subscript. The index usually uses a consecutive range of integers, (as opposed to an associative array) but the index can have any ordinal set of values.

Some arrays are multi-dimensional, meaning they are indexed by a fixed number of integers, for example by a tuple of four integers. Generally, one- and two-dimensional arrays are the most common. Most programming languages have a built-in array data type.

Link List

In computer science, a linked list is one of the fundamental data structures used in computer programming. It consists of a sequence of nodes, each containing arbitrary data fields and one or two references ("links") pointing to the next and/or previous nodes. A linked list is a self-referential data type because it contains a link to another data of the same type. Linked lists permit insertion and removal of nodes at any point in the list in constant time, but do not allow random access.



Types of Link List

1. Linearly-linked List
 - Singly-linked list
 - Doubly-linked list
2. Circularly-linked list
 - Singly-circularly-linked list
 - Doubly-circularly-linked list
3. Sentinel nodes

Stack

A stack is a linear Structure in which item may be added or removed only at one end. There are certain frequent situations in computer science when one wants to restrict insertions and deletions so that they can take place only at the beginning or the end of the end of the list, not in the middle. Two of the Data Structures that are useful in such situations are Stacks and queues. A stack is a list of elements in which an element may be inserted or deleted only at one end, called the Top. This means, in particular, the elements are removed from a stack in the reverse order of that which they are inserted in to the stack. The stack also called "last-in first -out (LIFO)" list.

Special terminology is used for two basic operation associated with stack:

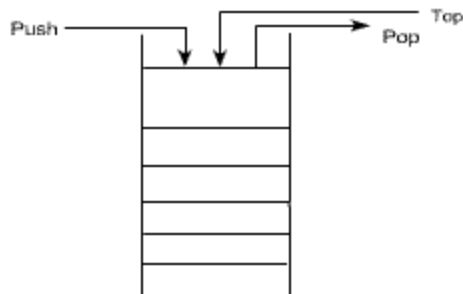


Fig: Stack

"Push" is the term used to insert an element into a stack.

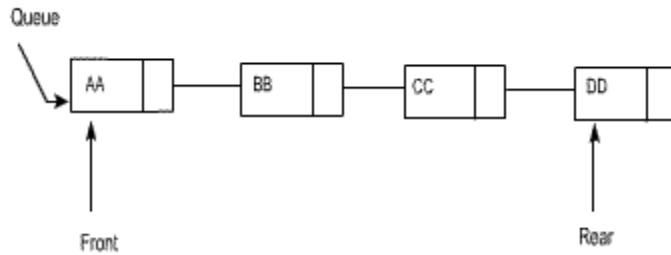
"Pop" is the term used to delete an element from a stack.

Queue

A queue is a linear list of elements in which deletions can take place only at one end, called the "front" and insertion can take place only at the other end, called "rear ". The term "front " and " rear " are used in describing a linear list only when it is implemented as a queue.

Queues are also called "first-in first-out " (FIFO) list. Since the first element in a queue will be the first element out of the queue. In other words, the order in which elements enter in a queue is

the order in which they leave. The real life example: the people waiting in a line at Railway ticket Counter form a queue, where the first person in a line is the first person to be waited on. An important example of a queue in computer science occurs in timesharing system, in which programs with the same priority form a queue while waiting to be executed.



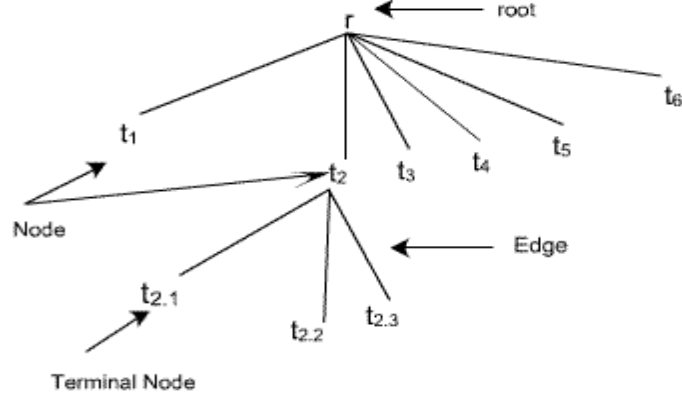
Tree

Data frequently contain a hierarchical relationship between various elements. This non-linear Data structure which reflects this relationship is called a rooted tree graph or, tree.

This structure is mainly used to represent data containing a hierarchical relationship between elements, e.g. record, family tree and table of contents.

A tree consist of a distinguished node r , called the root and zero or more (sub) tree $t_1 , t_2 , \dots t_n$, each of whose roots are connected by a directed edge to r .

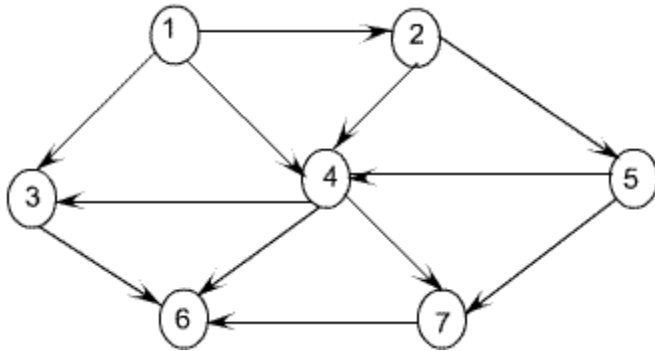
In the tree of figure, the root is A, Node t_2 has r as a parent and $t_{2.1}$, $t_{2.2}$ and $t_{2.3}$ as children. Each node may have arbitrary number of children, possibly zero. Nodes with no children are known as leaves.



Graph

A graph consists of a set of nodes (or Vertices) and a set of arc (or edge). Each arc in a graph is specified by a pair of nodes. A node n is incident to an arc x if n is one of the two nodes in the ordered pair of nodes that constitute x . The degree of a node is the number of arcs incident to it. The indegree of a node n is the number of arcs that have n as the head, and the outdegree of n is the number of arcs that have n as the tail.

The graph is the nonlinear data structure. The graph shown in the figure represents 7 vertices and 12 edges. The Vertices are $\{ 1, 2, 3, 4, 5, 6, 7 \}$ and the arcs are $\{(1,2), (1,3), (1,4), (2,4), (2,5), (3,4), (3,6), (4,5), (4,6), (4,7), (5,7), (6,7)\}$. Node (4) in figure has indegree 3, outdegree 3 and degree 6.



Abstract Data Type

Abstract Data Types (ADT's) are a model used to understand the design of a data structure 'Abstract' implies that we give an implementation-independent view of the data structure ADTs specify the type of data stored and the operations that support the data viewing a data structure as an ADT allows a programmer to focus on an idealized model of the data and its operations

1.4 Problems-Introduction to Data Structure

Problems:

- 1.What is Information in Computer Science?
- 2.What are methods for representing negative binary number? The following numbers convert to ones complement and twos complement notation.

00110111

01100110

01111101

10001001

3. Write a C program where following numbers are stored in an array :

2 12 17 24 5 78 35 18 16

4. Write a C program using linked list where following numbers are stored :

2 12 17 24 5 78 35 18 16

5. Consider the stack NAME in fig 1.01, which is stored alphabetically.

- Suppose Nirmal is to be inserted into the stack. How many names must be moved to the new location?
- Suppose Sourav is to be deleted from the stack. How many names must be removed to the new location?

NAME	
1	Abhisek
2	Anupam
3	Charls
4	Debasis
5	Pranay
6	Ritwik
7	Sourav
8	Suman

fig 1.01

6. The following is a tree structure given by means of level numbers as discussed below:

01 Employee 02 Name 02 Emp. Code 02 Designation 03 Project Leader 03 Project Manager 02 Address

Draw the corresponding tree diagram.

UNIT II: INTRODUCTION TO ALGORITHMS

2.1 Introduction to Algorithms

1. The word algorithm, came from the 9th century Persian mathematician "Abu Abdullah Muhammad bin Musa al-Khwarizmi", which means the method of doing arithmetic using Indo-Arabic decimal system. It is also the root of the word "algorithm".
2. An algorithm is a well defined computational method that takes some value(s) as input and produces some **value(s)** as output. In other words, an algorithm is a sequence of computational steps that transforms input(s) into output(s).
3. An algorithm is correct if for every input, it halts with correct output. A correct algorithm solves the given problem, **where** as an incorrect algorithm might not halt at all on some input instance, or it might halt with other than **designed** answer.
4. Each algorithm must have
 - Specification: Description of the computational procedure.
 - Pre-conditions: The condition(s) on input.
 - Body of the Algorithm: A sequence of clear and unambiguous instructions.
 - Post-conditions: The condition(s) on output.

2.1.1 Example 1- Introduction to Algorithms

- Algorithms are written in pseudo code that resembles programming languages like C and Pascal.
- Consider a simple algorithm for finding the factorial of n .

Algorithm Factorial (n)

Step 1:	FACT = 1
Step 2:	for i = 1 to n do
Step 3:	FACT = FACT * i
Step 4:	print FACT

Specification: Computes $n!$.

Pre-condition: $n \geq 0$

Post-condition: FACT = $n!$

- For better understanding conditions can also be defined after any statement, to specify values in particular variables.
- Pre-condition and post-condition can also be defined for loop, to define conditions satisfied before starting and after completion of loop respectively.
- What is remain true before execution of the i^{th} iteration of a loop is called "loop invariant".
- These conditions are useful during debugging proces of algorithms implementation.
- Moreover, these conditions can also be used for giving correctness proof.

2.1.2 Example 2- Introduction to Algorithms: A Sorting Algorithm

- Now, we take a more complex problem called sorting.
- Problem Definition: Sort given n numbers by non-descending order.
- There are many sorting algorithm. Insertion sort is a simple algorithm.
- Insertion Sort: We can assume up to first number is sorted. Then sort up to two numbers. Next, sort up to three numbers. This process continues till we sort all n numbers.
- Consider the following example of five integers:

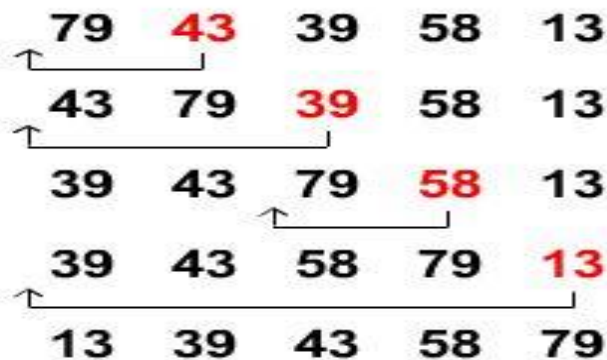
79 43 39 58 13: Up to first number, 79, is sorted.

43 79 39 58 13: Sorted up to two numbers.

39 43 79 58 13: Sorted up to three numbers.

39 43 58 79 13: Sorted up to four numbers.

13 39 43 58 79: Sorted all numbers.



- That is, if first $(i-1)$ numbers are sorted then insert i^{th} number into its correct position. This can be done by shifting numbers right one number at a time till a position for i^{th} number is found.
- That is, shift number at $(i-1)^{\text{th}}$ position to i^{th} position, number in $(i-2)^{\text{th}}$ position to $(i-1)^{\text{th}}$ position, and so on, till we find a correct position for the number in i^{th} position. This method is depicted in the figure on right side.

The algorithmic description of insertion sort is given below.

Algorithm Insertion_Sort ($a[n]$)

```

Step 1:                for i = 2 to n do
Step 2:                current_num = a[i]
Step 3:                j = i
Step 4:                while ((j > 1) and (a[j-1] > current_num)) do
Step 5:                    a[j] = a[j-1]
Step 6:                    j = j-1
Step 7:                a[j] = current_num

```

More about sorting algorithms are discussed in the Unit 5 and 10.

2.2 Time Complexity

- Execution time of an algorithm depends on numbers of instruction executed.
- Consider the following algorithm fragment:


```

for i = 1 to n do
  sum = sum + i ;

```
- The for loop executed $n+1$ times for i values 1, 2,....., n , $n+1$. Each instruction in the body of the loop is executed once for each value of $i = 1, 2, \dots, n$. So number of steps executed is $2n+1$.
- Consider another algorithm fragment:


```

for i = 1 to n do
  for j = 1 to n do

```


$$k = k + 1$$

- From previous example, number of instructions executed in the inner loop is which the body of outer loop is.

- Total number of instructions executed is

$$n + 1 + n(2n + 1)$$

$$2n^2 + 2n + 1$$

- To measure the time complexity in absolute time unit has the following problems

1. The time required for an algorithm depends on number of instructions executed, which is a complex polynomial.

2. The execution time of an instruction depends on computer's power. Since, different computers take different amount of time for the same instruction.

3. Different types of instructions take different amount of time on same computer.

- Complexity analysis technique abstracts away these machine dependent factors. In this approach, we assume all instructions take constant amount of time for execution.

- Asymptotic bounds as polynomials are used as a measure of the estimation of the number of instructions to be executed by the algorithm. Three main types of asymptotic order notations are used in practice:

1. Θ - **notation** : For a given function $g(n)$, $\Theta(g(n))$ is defined as

$$\Theta(g(n)) = \left\{ \begin{array}{l} f(n) : \text{there exist } c_1 > 0, c_2 > 0 \text{ and } n_0 \in \mathbb{N} \\ \text{such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \text{ for all } n \geq n_0 \end{array} \right\}$$

2. O - **notation** : For a given function $g(n)$, $O(g(n))$ is defined as

$$O(g(n)) = \left\{ \begin{array}{l} f(n) : \text{there exists } c > 0, \text{ and } n_0 \in \mathbb{N} \\ \text{such that } 0 \leq f(n) \leq c g(n), \text{ for all } n \geq n_0 \end{array} \right\}$$

3. Ω - **notation**: This notation provides asymptotic lower bound. For a given $g(n)$, $\Omega(g(n))$ is defined as

$$\Omega(g(n)) = \left\{ \begin{array}{l} f(n) : \text{there exists } c > 0, \text{ and } n_0 \in \mathbb{N} \\ \text{such that } 0 \leq c g(n) \leq f(n), \text{ for all } n \geq n_0 \end{array} \right\}$$

For example, $n^3 \in \Omega(n^2)$ because $n^3 \geq n^2$, for all $n \geq 0$

- Step 1: The for loop executed n times, for i values from 2, 3, $n+1$

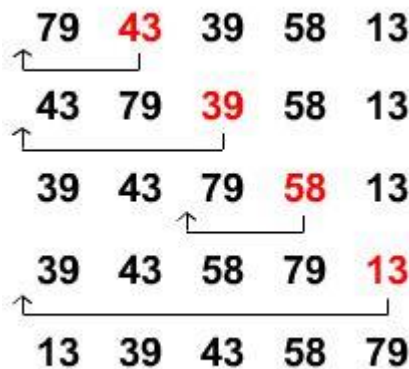
- Step 2, 3, and 7: Each executed once for each iteration of the for loop. That is, each $n-1$ times.
- Step 4: The while loop is executed at most i times, for $i = 2, 3, \dots, n$. That is, $n \times (n+1) / 2 - 1$.
- step 5 and 6: Each instruction executed $i-1$ times, for each i . Hence, $(n-1) \times n / 2$ for each instruction.
- Total number of instruction executed in the worst case is $n^2 + 4n - 4$. Hence the time complexity of the algorithm is in $\Theta(n^2)$.

Similarly, we can analyze space required for the algorithm also. The Insertion sort algorithm has to store all n inputs and using three more variables. So, the space complexity of the algorithm is in $\Theta(n)$.

2.3 Recurrence

Another way of finding number of instruction executed is by recursive equation. Let $T(n)$ be the time required to sort n numbers. $T(n)$ can be expressed as a sum of $T(n-1)$ and the time required to insert n^{th} element in the sorted array of $n-1$ element.

The time required to insert an element in sorted array of $n-1$ elements takes cn steps, where c is a positive constant. This is because to insert n^{th} element, in worst case, we have to shift all $n-1$ elements one after other.



See the following figure.

- Hence, the recurrence relation for Insertion sort is

$$(n) \quad = T(n-1) + cn, \text{ if } n > 1$$

$$= 1 \text{ if } n = 1$$

- There are three main approach to solve recurrence relations. They are substitution, iterative and master theorem methods. In this notes we discuss iterative approach only. For other methods see Algorithms by Cormen et al.

- Iterative Method:

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

$$= T(n-2) + c(n-1) + cn$$

$$= T(n-3) + c(n-2) + c(n-1) + cn$$

$$\vdots$$

$$\vdots$$

$$= T(1) + 2c + 3c + \dots + c(n-2) + c(n-1) + cn$$

$$= 1 + 2c + 3c + \dots + c(n-2) + c(n-1) + cn$$

$$< c(1 + 2 + 3 + \dots + n-2 + n-1 + n)$$

$$= c \left(\frac{n(n+1)}{2} \right)$$

$$= c \frac{n^2}{2} + c \frac{n}{2}$$

$$= O(n^2)$$

- Consider another recurrence:

$$T(n) = O(1) \quad \text{if } n = 1$$

$$= 2T(n/2) + O(n) \quad \text{if } n > 1$$

- In the above recurrence relation $O(1)$ means a constant. So we can replace with some constant c_1 .
- Similarly, $O(n)$ means a function of order n . So we can replace with C_2n . Hence, the recurrence can be rewritten as

$$T(n) \leq C_1 \text{ if } n=1$$

$$\leq 2T(n/2) + C_2n \text{ if } n > 1$$

- Solution by Iterative method:

$$T(n) \leq 2T(n/2) + C_2n$$

$$\leq 2(2T(n/4) + C_2n/2) + C_2n$$

$$\leq 2^2 T(n/2^2) + C_2n + C_2n$$

$$\leq 2^2 (2 T(n/2^3) + C_2 n/2^2) + C_2n + C_2n$$

$$\leq 2^3 T(n/2^3) + C_2 n + C_2 n + C_2 n$$

•
•
•

$$\leq 2^i T(n/2^i) + C_2 n i$$

- Assume $n = 2^i$ for some value of i . That is, $i = \log n$

$$T(n) = nT(1) + C_2 n \log(n)$$

$$= C_1 n + C_2 n \log(n)$$

$$= O(n \log(n))$$

- If $n \neq 2^i$ for some i

$$T(n) < 2nT(1) + c_2 n \log(n)$$

$$= 2c_1 n + c_2 n \log(n)$$

$$= O(n \log(n))$$

2.4 Problems – Introduction to algorithms

1. Given an array of n integers, write an algorithm to find the smallest element. Find number of instruction executed by your algorithm. What are the time and space complexities?
2. Write a algorithm to find the median of n numbers. Find number of instruction executed by your algorithm. What are the time and space complexities?
3. Write a C program for the problem 1.
4. Write a C program for the problem 2.
5. Solve the following recurrence relations. Assume $T(1) = O(1)$

1. $T(n) = T(n-2) + cn$

2. $T(n) = 2T(n/2) + c n^2$

3. $T(n) = 2T(n/2) + c n^3$

4. $T(n) = 4 T(n/4) + O(n)$

5. $T(n) = 4 T(n/2) + c n$

6. $T(n) = T(n/2) + O(1)$

7. $T(n) = 2T(n/2) + n \log n$

8. $T(n) = T(n-1) + 1/n$

9. $T(n) = T(n/2) + \log n$

10. $T(n) = T(\quad) + 1$

UNIT III: LINEAR DATA STRUCTURES

3.1 linear Data Structures

Definition

A **data structure** is said to be *linear* if its elements form a sequence or a **linear** list.

Examples:

- Array
- Linked List
- Stacks
- Queues

Operations on linear Data Structures

- *Traversal* : Visit every part of the **data structure**
- *Search* : Traversal through the data structure for a given element
- *Insertion* : Adding new elements to the **data structure**
- *Deletion*: Removing an element from the **data structure**.
- *Sorting* : Rearranging the elements in some type of order(e.g Increasing or Decreasing)
- *Merging* : Combining two similar **data** structures into one

3.2 Introduction to Stack

Introduction

1. Stack is basically a data object
2. The operational semantic (meaning) of stack is LIFO i.e. last in first out

Definition: It is an ordered list of elements n , such that $n > 0$ in which all insertions and deletions are made at one end called the top.

Primary operations defined on a stack:

1. PUSH: add an element at the top of the list.
2. POP : remove at the top of the list.

3. Also "IsEmpty()" and IsFull" function, which tests whether a stack is empty or full respectively.

Example:

1. **Practical daily life :** a pile of heavy books kept in a vertical box,dishes kept one on top of another

In computer world: In processing of subroutine calls and returns; there is an explicit use of stack of return addresses.

Also in evaluation of arithmetic expressions, stack is used.

Large number of stacks can be expressed using a single one dimensional stack only. Such an array is called a multiple stack array.

Push and Pop :

Algorithms

Push (item,array , n, top)

```
{
If ( n>= top)
Then print "Stack is full" ;
Else
{
top = top + 1;
array[top] = item ;
}
}
```

Pop (item,array,top)

```
{
if ( top<= 0)
Then print "stack is empty".
Else
{
item = array[top];
top = top - 1;
}
}
```

Arithmetic Expressions :

Arithmetic expressions are expressed as combinations of:

1. Operands
2. Operators (arithmetic, Boolean, relational operators)

Various rules have been formulated to specify the order of evaluation of combination of operators in any expression.

The arithmetic expressions are expressed in 3 different notations:

1. Infix

- In this if the operator is binary; the operator is between the 2 operands.
- And if the operator is unary, it precedes the operand.

2. Prefix:

- In this notation for the case of binary operators, the operator precedes both the operands.
- Simple algorithm using stack can be used to evaluate the final answer.

3. Postfix:

- In this notation for the case of binary operators, the operator is after both the corresponding operands.
- Simple algorithm using stack can be used to evaluate the final answer.

Always remember that the order of appearance of operands does not change in any Notation. What changes is the position of operators working on those operands.

Rules Expressions:

RULES FOR EVALUATION OF ANY EXPRESSION:

An expression can be interpreted in many different ways if parentheses are not mentioned in the expression.

- For example the below given expression can be interpreted in many different ways:
- Hence we specify some basic rules for evaluation of any expression :

A priority table is specified for the various type of operators being used:

PRIORITY LEVEL	OPERATORS
6	** ; unary - ; unary +
5	* ; /
4	+ ; -
3	< ; > ; <= ; >= ; !> ; !< ; !=
2	Logical and operation
1	Logical or operation

Arithmetic Expressions:

Algorithm for evaluation of an expression E

which is in prefix notation :

- We assume that the given prefix notation starts with IsEmpty ().
- If number of symbols = n in any infix expression then number of operations performed = some constant times n.
- Here *next token* function gives us the next occurring element in the expression in a left to right scan.
- The *PUSH* function adds element x to stack Q which is of maximum length n

Evaluate (E)

```

{
    Else
    {
Top = 0;
While (1)
    {
    If (x == operand)
        PUSH (Q, top, n, x);
    If (x == operator)
    {
    x= next token (E)
    If (x == infinity)
    {
    Pop correct number of operands according to the
    operator (unary/binary) and then perform the
    operation and store result onto the stack
    Print value of stack [top] as the output of the
    expression
    }
    }
    }
}
}
}

```


Multiple stacks:

- Here only one single one-dimensional array (Q) is used to store multiple stacks.
- B (i) denotes one position less than the position in Q for bottommost element of stacks i.
- T (i) denotes the top most element of stack i.
- m denotes the maximum size of the array being used.
- n denotes the number of stacks.

We also assume that equal segments of array will be used for each stack.

Initially let $B(i) = T(i) = [m/n] * (i-1)$ where $1 \leq i \leq n$.

Again we can have *push* or *pop* operations, which can be performed on each of these stacks.

Algorithms:

Push (i, x)

```
{
    If (((i < n) && (T(i) == B(i+1))) ||
        ((i >= n) && (T(i) == m)));

        Then call STACK_FULL;
    Else
    {
        T(i) = T(i) + 1;
        Q[T(i)] = x;
    }
}
```

Pop (i,x)

```
{
    if ( T(i) == B(i) )
        Then print that the stack is empty.
    Else
    {
        x = Q[T(i)];
        T[i] = T[i] - 1;
    }
}
```

ALGORITHM TO BE APPLIED WHEN $T[i] = B[i]$ CONDITION IS ENCOUNTERED WHILE DOING PUSH OPERATION.

{

1. Find j such that $i < j \leq n$ and there is a free space between stack j and stack $(j + 1)$.
if such a j exist , then move stack $i+1$, $i+2$,.....till j one position to the right and hence create space for element between stack i and $i + 1$.

2. Else

Find j such that $1 \leq j < i$ and there is a free space between stack j and stack $(j + 1)$.
if such a j exist , then move stack $j+1$, $j+2$,.....till i one position to the left and hence create space for element between stack i and $i + 1$.

3. Else

If none of above is possible then there is no space left out in the one-dimensional array used hence print no space for push operation.

}

3.3 Introduction to Queue

Introduction:

1. It is basically a data object
2. The operational semantic of queue is FIFO i.e. first in first out

Definition :

It is an ordered list of elements n , such that $n > 0$ in which all deletions are made at one end called the front end and all insertions at the other end called the rear end .

Primary operations defined on a Queue:

1. EnQueue : This is used to add elements into the queue at the back end.
2. DeQueue : This is used to delete elements from a queue from the front end.
3. Also "IsEmpty()" and "IsFull()" can be defined to test whether the queue is Empty or full.

Example :

1. **PRACTICAL EXAMPLE:** A line at a ticket counter for buying tickets operates on above rules
2. **IN COMPUTER WORLD:** In a batch processing system, jobs are queued up for processing.

Circular queue:

In a queue if the array elements can be accessed in a circular fashion the queue is a circular queue.

Priority queue:

Often the items added to a queue have a priority associated with them: this priority determines the order in which they exit the queue - highest priority items are removed first.

CIRCULAR QUEUE

Primary operations defined for a circular queue are :

1. **add_circular** - It is used for addition of elements to the circular queue.

2. **delete_circular** - It is used for deletion of elements from the queue.

We will see that in a circular queue, unlike static linear array implementation of the queue; the memory is utilized more efficient in case of circular queue's.

The shortcoming of static linear that once rear points to n which is the max size of our array we cannot insert any more elements even if there is space in the queue is removed efficiently using a circular queue.

As in case of linear queue, we'll see that condition for zero elements still remains the same i.e..
rear=front

ALGORITHM FOR ADDITION AND DELETION OF ELEMENTS

Data structures required for circular queue :

1. **front** counter which points to one position anticlockwise to the 1st element
2. **rear** counter which points to the last element in the queue
3. an **array** to represent the queue

```
add _ circular ( item,queue,rear,front)
    {
        rear=(rear+1)mod n;
        if (front == rear )
            then print " queue is full "
        else
```

```

        {
            queue [rear]=item;
        }
    }
delete operation :
delete_circular (item,queue,rear,front)
    {
        if (front == rear)
            print ("queue is empty");
        else
            {
                front= front+1;
                item= queue[fromt];
            }
    }

```

ALGORITHM FOR ADDITION AND DELETION OF ITEMS IN A QUEUE

note : addition is done only at the rear end of a queue like in a ticket counter line

add (item ,queue , n ,rear)

```

    {
        if (rear==n)
            then print " queue is full "
        else
            {
                rear=rear+1;
                queue [rear]=item;
            }
    }

```

ALGORITHM FOR ADDITION AND DELETION OF ITEMS IN A QUEUE

note : deletion is allowed only at the front end of the queue

delete (item , queue , rear , front)

```
{
    if (rear==front)
        then print "queue is empty";
    else
        {
            item = queue [front] ;
            front=front+1 ;
        }
}
```

Priority queue:

Queues are dynamic collections which have some concept of order. This can be either based on order of entry into the queue - giving us First-In-First-Out (FIFO) or Last-In-First-Out (LIFO) queues. Both of these can be built with linked lists: the simplest "add-to-head" implementation of a linked list gives LIFO behavior. A minor modification - adding a tail pointer and adjusting the addition method implementation - will produce a FIFO queue.

Performance

A straightforward analysis shows that for both these cases, the time needed to add or delete an item is constant and *independent of the number of items in the queue*. Thus we class both addition and deletion as an $O(1)$ operation. For any given real machine + operating system + language combination, addition may take c_1 seconds and deletion c_2 seconds, but we aren't interested in the value of the constant, it will vary from machine to machine, language to language, *etc* . The key point is that the time is not dependent on n - producing $O(1)$ algorithms.

Once we have written an $O(1)$ method, there is generally little more that we can do from an algorithmic point of view. Occasionally, a better approach may produce a lower constant time. Often, enhancing our compiler, run-time system, machine, *etc* will produce some significant improvement. However $O(1)$ methods are already very fast, and it's unlikely that effort expended in improving such a method will produce much real gain!

PRIORITY QUEUE:

Often the items added to a queue have a **priority** associated with them: this priority determines the order in which they exit the queue - highest priority items are removed first.

This situation arises often in process control systems. Imagine the operator's console in a large automated factory. It receives many routine messages from all parts of the system: they are assigned a low priority because they just report the normal functioning of the system - they update various parts of the operator's console display simply so that there is some confirmation that there are no problems. It will make little difference if they are delayed or lost.

However, occasionally something breaks or fails and alarm messages are sent. These have high priority because some action is required to fix the problem (even if it is mass evacuation because nothing can stop the imminent explosion!).

Typically such a system will be composed of many small units, one of which will be a buffer for messages received by the operator's console. The communications system places messages in the buffer so that communications links can be freed for further messages while the console software is processing the message. The console software extracts messages from the buffer and updates appropriate parts of the display system. Obviously we want to sort messages on their priority so that we can ensure that the alarms are processed immediately and not delayed behind a few thousand routine messages while the plant is about to explode.

As we have seen, we could use a tree structure - which generally provides $O(\log n)$ performance for both insertion and deletion. Unfortunately, if the tree becomes unbalanced, performance will degrade to $O(n)$ in pathological cases. This will probably not be acceptable when dealing with dangerous industrial processes, nuclear reactors, flight control systems and other *life-critical* systems.

C ++ IMPLEMENTATION OF QUEUE USING CLASSES

```

#include <iostream.h>
#include <conio.h>

#define MAX 5 // MAXIMUM CONTENTS IN QUEUE

class queue
{
private:
    int t[MAX];
    int al; // Addition End
    int dl; // Deletion End
public:
    queue()
    {
        dl=-1;
        al=-1;
    }

    void del()
    {
        int tmp;
        if(dl!=-1)
        {

        cout<<"Queue is Empty";
        }
        else
        {
            for(int j=0;j<=al;j++)
            {
                if((j+1)<=al)
                {
                    tmp=t[j+1];
                    t[j]=tmp;
                }
                else
                {
                    al--;
                }

                if(al==-1)
                    dl=-1;
                else
                    dl=0;
            }
        }
    }
}

```

```

void add(int item)
{
    if(dl==-1 && al==-1)
    {
        dl++;
        al++;
    }
    else
    {
        al++;
        if(al==MAX)
        {
            cout<<"Queue is Full\n";
            al--;
            return;
        }
    }
    t[al]=item;
}

void display()
{
    if(dl!=-1)
    {
        for(int i=0;i<=al;i++)
            cout<<t[i]<<" ";
    }
    else
        cout<<"EMPTY";
}
};

```

```

void main()
{
queue a;
int data[5]={32,23,45,99,24};

cout<<"Queue before adding Elements: ";
a.display();
cout<<endl<<endl;

for(int i=0;i<5;i++)
{
a.add(data[i]);
cout<<"Addition Number : "<<(i+1)<<" : ";
a.display();
cout<<endl;
}
cout<<endl;
cout<<"Queue after adding Elements: ";
a.display();

cout<<endl<<endl;

for(int i=0;i<5;i++)
{
a.del();
cout<<"Deletion Number : "<<(i+1)<<" : ";
a.display();
cout<<endl;
}
getch();
}

```

OUTPUT:

```

Queue before adding Elements: EMPTY
Addition Number : 1 : 32
Addition Number : 2 : 32 23
Addition Number : 3 : 32 23 45
Addition Number : 4 : 32 23 45 99
Addition Number : 5 : 32 23 45 99 24
Queue after adding Elements: 32 23 45 99 24
Deletion Number : 1 : 23 45 99 24
Deletion Number : 2 : 45 99 24
Deletion Number : 3 : 99 24
Deletion Number : 4 : 24
Deletion Number : 5 : EMPTY

```

As you can clearly see through the output of this program that addition is always done at the end of the queue while deletion is done from the front end of the queue.

3.4 Problems-Linear Data Structure

Tower of Hanoi Problem

Tower of Hanoi is a historical problem, which can be easily expressed using recursion. There are N disks of decreasing size stacked on one needle, and two other empty needles. It is required to stack all the disks onto a second needle in the decreasing order of size. The third needle can be used as a temporary storage. The movement of the disks must confirm to the following rules,

1. Only one disk may be moved at a time
2. A disk can be moved from any needle to any other.
3. The larger disk should not rest upon a smaller one.

Question: write a c program to implement tower of Hanoi using stack ?

Solution:

```
#include <stdio.h>
#include <conio.h>
void move ( int, char, char, char );
void main( )
{
    int n = 3 ;
    clrscr( ) ;
    move ( n, 'A', 'B', 'C' ) ;
    getch( ) ;
}
void move ( int n, char sp, char ap, char ep )
{
    if ( n == 1 )
        printf ( "\nMove from %c to %c ", sp, ep ) ;
    else
    {
        move ( n - 1, sp, ep, ap ) ;
        move ( 1, sp, ' ', ep ) ;
        move ( n - 1, ap, sp, ep ) ;
    }
}
```

Function Calls and Stack

A stack is used by programming languages for implementing function calls. write a program to check how function calls are made using stack.

SOLUTION

```
/* To show the use of stack in function calls */

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <dos.h>

unsigned int far *ptr ;
void (*p)(void) ;

void f1() ;
void f2() ;

void main()
{
    f1() ;
    f2() ;
    printf ( "\nback to main..." ) ;
    exit ( 1 ) ;
}

void f1()
{
    ptr = ( unsigned int far * ) MK_FP ( _SS, _SP + 2 ) ;
    printf ( "n%d", *ptr ) ;
    p = ( void ( * )() ) MK_FP ( _CS, *ptr ) ;
    (*p)() ;
    printf ( "\nI am f1() function " ) ;
}

void f2()
{
    printf ( "\nI am f2() function" ) ;
}
```

QUEUE QUESTIONNAIRE:

PROBLEM 1 :

The Queue has operations create, append, front, remove and is Empty.

A Queue contains a sequence of integers. Design an algorithm to construct another queue containing the same integers but in reverse order. The only queue operations available to you are those listed above.

PROBLEM 2 :

Repeat the previous question, only this time you have to leave the first queue in its original state as well.

PROBLEM 3 :

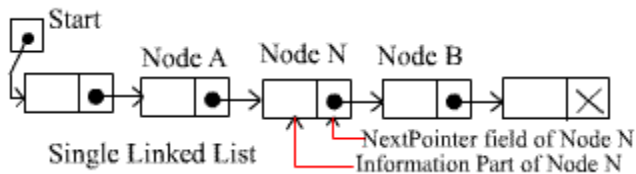
A queue contains a sequence of alphabetic characters. Design an algorithm to test whether the contents of the queue is a palindrome. As before you should assume that the only queue operations available to you are those listed in question 1.

UNIT IV: LINKED LIST

4.1 linked list

What are Linked Lists?

A linked list is simply a chain of structures which contain a pointer to the next element. It is dynamic in nature. Items may be added to it or deleted from it at will.



This definition applies only to Singly Linked Lists - Doubly Linked Lists and Circular Lists are different.

A list item has a pointer to the next element or to 0 if the current element is the tail (end of the list). This pointer points to a structure of the same type as itself. This structure that contains elements and pointers to the next structure is called a Node.

Each node of the list has two elements:

- the item being stored in the list *and*
- a pointer to the next item in the list

Some common examples of a linked list:

- Hash tables use linked lists for collision resolution
- Any "File Requester" dialog uses a linked list
- Binary Trees
- Stacks and Queues can be implemented with a doubly linked list
- Relational Databases (e.g. Microsoft Access)

4.2 Inserting and Removing Nodes from a list

Algorithm for inserting a node to the List

- allocate space for a new node,
- copy the item into it,
- make the new node's next pointer point to the current head of the list and
- Make the head of the list point to the newly allocated node.

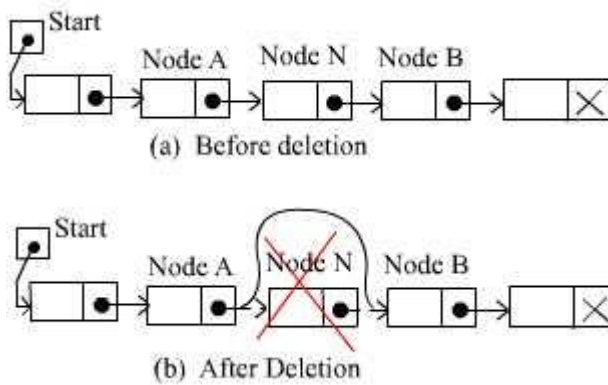
This strategy is fast and efficient, but each item is added to the head of the list. Below is given C code for inserting a node after a given node.

C Implementation

```
/* inserts an item x into a list after a node pointed to by p */
```

```
void insafter(int p, int x)
{
int q;
if(p == -1)
{
printf("void insertion\n");
return;
}
q = getnode(); /* getnode() returns a pointer to newly allocated node */
node[q].info = x;
node[q].next = node[p].next;
node[p].next = q;
return;
} /* end insafter
```

Algorithm for deleting a node from the List



Step-1: Take the value in the 'nodevalue' field of the TARGET node in any intermediate variable.
Here node N.

Step-2: Make the previous node of TARGET to point to where TARGET is currently pointing

Step-3: The nextpointer field of Node N now points to Node B, Where Node N previously pointed.

Step-4: Return the value in that intermediate variable

There are also two special cases. If the deleted node N is the first node in the list, then the Start will point to node B; and if the deleted node N is the last node in the list, then Node A will contain the NULL pointer.

Below is a C code for deleting a node after a given node.

C Implementation

```
/* routine delafter(p,px), called by the statement delafter(p,&x), deletes the node following node(p) and stores its contents in x */
```

```
void delafter(int p, int *px)
```

```
{
    int q;
    if ((p == -1) || (node[p].next == -1))
    {
        printf("void deletion\n");
        return;
    }
    q=node[p].next;
    *px = node[q].info;
    node[p].next = node[q].next;
    freenode(q);
    return;
}
```

```
/* end delafter */
```

4.3 linked implemented of stacks

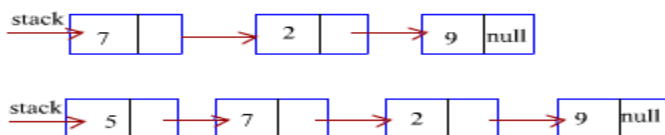
The operation of adding an element to the front of a linked list is quite similar to that of pushing an element on to a stack. A stack can be accessed only through its top element, and a list can be accessed only from the pointer to its first element. Similarly, removing the first element from a linked list is analogous to popping from a stack.

A linked-list is somewhat of a dynamic array that grows and shrinks as values are added to it and removed from it respectively. Rather than being stored in a continuous block of memory, the values in the dynamic array are linked together with pointers. Each element of a linked list is a structure that contains a value and a link to its neighbor. The link is basically a pointer to another structure that contains a value and another pointer to another structure, and so on. If an external pointer p points to such a linked list, the operation $\text{push}(p, t)$ may be implemented by

```
f = getnode();
    info(f) = t;
    next(f) = p;
p = f;
```

The operation $t = \text{pop}(p)$ removes the first node from a nonempty list and signals underflow if the list is empty

```
if(empty(p))
{
printf('stack underflow');
exit(1);
}
else {
f = p;
p = next(f);
t = info(f);
freenode(f);
}
```



4.4 Getnode and freenode operation

GETNODE AND FREENODE OPERATIONS

The getnode operation may be regarded as a machine that manufactures nodes. Initially there exist a finite pool of empty nodes and it is impossible to use more than that number at a given instant. If it is desired to use more than that number over a given period of time, some nodes must be reused. The function of freenode is to make a node that is no longer being used in its current context available for reuse in a different context.

The list of available nodes is called the available list. When the available list is empty that is all nodes are currently in use and it is impossible to allocate any more, overflow occurs.

Assume that an external pointer *avail* points to a list of available nodes. Then the operation getnode and freenode are implemented as follows :

```
int getnode(void)
{
int p;

if (avail == -1)
{
printf("overflow\n");
exit(1);
}
p = avail;

avail = node[avail].next;

return(p);

} /* end getnode */
void freenode (int p)
{
node[p].next = avail;
```

```

avail = p;

return;

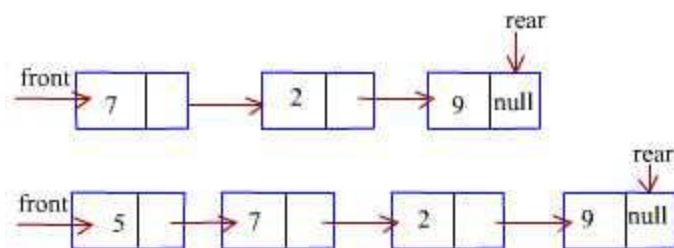
} /* end freenode */

```

4.5 linked implemented of queue

Queues can be implemented as linked lists. Linked list implementations of queues often require two pointers or references to links at the beginning and end of the list. Using a pair of pointers or references opens the code up to a variety of bugs especially when the last item on the queue is dequeued or when the first item is enqueued.

In a circular linked list representation of queues, ordinary 'for loops' and 'do while loops' do not suffice to traverse a loop because the link that starts the traversal is also the link that terminates the traversal. The empty queue has no links and this is not a circularly linked list. This is also a problem for the two pointers or references approach. If one link in the circularly linked queue is kept empty then traversal is simplified. The one empty link simplifies traversal since the traversal starts on the first link and ends on the empty one. Because there will always be at least one link on the queue (the empty one) the queue will always be a circularly linked list and no bugs will arise from the queue being intermittently circular. Let a pointer to the first element of a list represent the front of the queue. Another pointer to the last element of the list represents the rear of the queue as shown in fig. illustrates the same queue after a new item has been inserted.



Under the list representation, a queue q consists of a list and two pointers, $q.front$ and $q.rear$. The operations are insertion and deletion. Special attention is required when the last element is removed from a queue. In that case, $q.rear$ must also be set to null, Since in an empty queue both $r.front$ and $q.rear$ must be null.

The pseudo code for deletion is below:

```
        if (empty(q))
    {
printf("Queue is Underflow");
exit(1);
    }
f = q.front;
t = info(f);
q.front = next(f);
    if (q.front == null)
q.rear = null;
freenode(f);
return(t);
```

The operation insert algorithm is implemented

```
        f = getnode();
info(f) = x;
next(f) = null;
    if (q.rear == null)
        q.front = f;
    else
        next(q.rear) = f;
    q.rear = f;
```

4.6 list implementation of priority queue

We can use a list to represent a priority queue in ordered list or unordered list. For an ascending Priority queue, insertion is implemented by the place operation, which keeps the list ordered, and deletion of the minimum element is implemented by the delete operation, which removes the first element from the list. A Descending priority queue can be implemented by keeping the list in descending order rather than ascending, or by using remove to delete the minimum element. In

an ordered list, if you want to insert an element to the priority queue, it will require examining an average of approximately $n/2$ nodes but only one search for deletion.

An unordered list may also be used as a priority queue. If you want to insert an element to the list always requires examining only one node but always requires examining n elements for removal of an element.

The advantage of a list over an array for implementing a priority queue is that an element can be inserted into a list without moving any other elements, where as this is impossible for an array unless extra space is left empty.

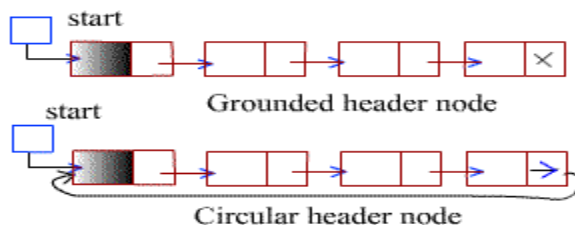
4.7 Header Nodes

A header linked list is a linked list which always contains a special node called the header node at the beginning of the list. It is an extra node kept at the front of a list. Such a node does not represent an item in the list. The information portion might be unused. There are two types of header list

- Grounded header list : is a header list where the last node contain the null pointer.
- Circular header list : is a header list where the last node points back to the header node.

More often, the information portion of such a node could be used to keep global information about the entire list such as:

- number of nodes (not including the header) in the list
 - count in the header node must be adjusted after adding or deleting the item from the list
- pointer to the last node in the list
 - it simplifies the representation of a queue
- pointer to the current node in the list
 - eliminates the need of a external pointer during traversal



4.8 Circular lists

CIRCULAR LIST

Circular lists are like singly linked lists, except that the last node contains a pointer back to the first node rather than the null pointer. From any point in such a list, it is possible to reach any other point in the list. If we begin at a given node and travel the entire list, we ultimately end up at the starting point.

Note that a circular list does not have a natural "first or "last" node. We must therefore, establish a first and last node by convention - let external pointer point to the last node, and the following node be the first node.

Stack as a Circular List

A circular list can be used to represent a stack.

The following is a C function to push an integer x onto the stack. It is called by push (&stack, x), where stack is a pointer to a circular list acting as a stack.

```
void push(NODEPTR *pstack, int x)
{
NODEPTR p;

p = getnode();
p->info = x;
if (*pstack == NULL)
*pstack = p;
else
p->next = (*pstack) -> next;
(*pstack) -> next = p;
} /*end push*/
```

Queue as a circular list

It is easier to represent a queue as a circular list than as a linear list. If considered as a linear list, a queue is specified by two pointers, one to the front of the list and other to its rear. However, by using a circular list, a queue may be specified by a single pointer q to that list. Node (q) is the rear of the queue and the following node is its front.

Queue as a circular list

The following is a C function to insert an integer x into the queue and is called by insert (&q, x)

```
Void insert (NODEPTR *pq, int x)
```

```
{  
NODEPTR p;  
    p = getnode();  
    p->info = x;  
    if (*pq == NULL)  
        *pq = p;  
    else  
        p->next = (*pq) -> next;  
    (*pq) -> next = p;  
    *pq = p;  
return;  
} /*end insert*/
```

Queue as a circular list

To insert an element into the rear of a circular queue, the element is inserted into the front of the queue and the circular list pointer is then advanced one element, so that the new element becomes the rear.

Exercise: Write an algorithm and a C routine to concatenate two circular lists.

4.9 Doubly Linked list

Doubly Linked Lists

Doubly linked lists are like singly linked lists, in which for each node there are two pointers -- one to the next node, and one to the previous node. This makes life nice in many ways:

- You can traverse lists forward and backward.

- You can insert anywhere in a list easily. This includes inserting before a node, after a node, at the front of the list, and at the end of the list and
- You can delete nodes very easily.

Doubly linked lists may be either linear or circular and may or may not contain a header node

4.10 Problems-Linked list

What are the differences between a linked list and an array?

1. Write a small program to delete the last node of a given linked list.
2. Using the same code, convert the existing list into a circular linked list.
3. Using the data given in question 2, Write a small function to traverse the linkedlist only once and find out the middle element.
4. Given a singly linked list, write a small subroutine getNth() to get nth node of the linked list.
5. Write a small subroutine insertNth() to insert nth node in a singly linked list.
6. What is a queue and how can you implement queue using a linked list?
7. What is a stack and how can you implement stack using a linked list?
8. Given below is the subroutine for deletion of a node in a queue.

```

Delete (struct node *list, struct node *first, struct node *last){
Struct node *temp;
temp=first;
first=first->next;
first->prev=NULL;
free(first);
}

```

9. There is an error in this code. Modify the code so that it deletes exactly the first node of the queue.
10. Given two singly linked lists, how do you append them?
11. What is the minimum number of queues needed to implement a stack? Write an algorithm to do so.

UNIT V: SORTING

5.1 Introduction to Sorting

Introduction:

- In many applications it is necessary to order give objects as per an attribute. For example, arranging a list of student information in increasing order of their roll numbers or arranging a set of words in alphabetical order.
- The attribute by which objects are arranged or sorted is called key. In the first example roll number is the key.
- One class of sorting methods is based on comparisons. Most important methods in this class are Insertion, Sink, Selection, Quick, Merge, and Heap sort algorithms. We have already discussed Insertion sort algorithms in the module 2. We discuss some of these algorithms in this module.

5.2 Sink Sort

Sink Sort

- Main idea in this method is to compare two adjacent elements and put them in proper order if they are not.
- Do this by scanning the element from first to last.
- After first scan, largest element is placed at last position. This is like a largest object sinking to bottom first.
- After second, scan second largest is placed at second last position.
- After n passes all elements are placed in their correct positions, hence sorted.

Input	Pass 1	Pass 2	Pass 3	Pass 4	Pass 5	Pass 6	Pass 7	Pass 8
12	12	12	12	12	12	12	12	12
32	18	18	18	18	18	14	14	14
18	24	24	19	19	14	18	18	18
24	30	19	24	14	19	19	19	19
30	19	28	14	24	24	24	24	24
19	28	14	28	28	28	28	28	28
28	14	30	30	30	30	30	30	30
14	32	32	32	32	32	32	32	32

The algorithmic description of the Sink sort is given below:

Algorithm Sink-Sort (a[n])

Step 1: for i = 0 to n-2 do

Step 2: for j = 0 to n-i do

Step 3: if ($a[j] > a[j+1]$) then

Step 4 : swap($a[j], a[j+1]$);

The algorithm can be modified, such that, after each pass next smallest element reach to the top position. This is like a bubble coming to top. Hence called Bubble-Sort.

5.3 Selection Sort

Selection Sort:

- As we have discussed earlier, purpose of sorting is to arrange a given set of records in order by specified key. In the case of student records, key can be roll number. In the case of employee's records, key can be employ identification number. That is, sorting a set of records based on specific key.
- Insertion and Sink sorts compare keys of two records and swap them if the keys are not in order. This is not just swapping keys; we have to swap whole records associated with those keys.
- The time required to swap records is proportional to its size that is number of fields.
- In these methods, same record may be swapped many times, before reaching its final position in sorted order.
- This method, selection sort, minimized the number of swaps. Hence efficient if the sizes of the records are large.
- Look at keys of all records to find a record with smallest key and place the record in the first place.
- Next, find the smallest key record among remaining records and swap with the record at second position.
- Continue this procedure till all records are placed correctly.

	12	32	18	24	30	19	28	14
After	12	32	18	24	30	19	28	14
Pass 1	12	32	18	24	30	19	28	14
Pass 2	12	14	18	24	30	19	28	32
Pass 3	12	14	18	24	30	19	28	32
Pass 4	12	14	18	19	30	24	28	32
Pass 5	12	14	18	19	24	30	28	32
Pass 6	12	14	18	19	24	28	30	32
Pass 7	12	14	18	19	24	28	30	32

- Pseudo code of the algorithm is given below.

Algorithm Selection_Sort (a[n])

```

Step 1:   for i = 0 to n-2 do
Step 2:   lowest-key = i
Step 3:   for j = i+1 to n-1 do {
Step 4:   if (a[i].key > a[j].key) then
Step 5:   lowest_key = j
Step 6:   swap(a[i], a[lowest_key]); }

```

5.4 Merge Sort

Merge sort:

- The time complexity of the sorting algorithms discussed till now are in $O(n^2)$. That is, the numbers of comparisons performed by these algorithms are bound above by $c \cdot n^2$, for some constant $c > 1$.
- Can we have better sorting algorithms? Yes, merge sort method sorts given a set of number in $O(n \log n)$ time.
- Before discussing merge sort, we need to understand what is the meaning of merging?

Merge sort Method:

Definition: Given two sorted arrays, a[p] and b[q], create one sorted array of size p + q of the elements of a[p] and a[q] .

- Assume that we have to keep p+q sorted numbers in an array c[p+q].
- One way of doing this is by copying elements of a[p] and b[q] into c[p+q] and sort c[p+q] which has time complexity of at least $O(n \log n)$.
- Another efficient method is by merging which is of time complexity of $O(n)$.
- Method is simple, take one number from each array a and b, place the smallest element in the array c. Take the next elements and repeat the procedure till all p+q element are placed in the array c

Pseudocode of the merging is given below.

Algorithm Merge (a[p], b[q])

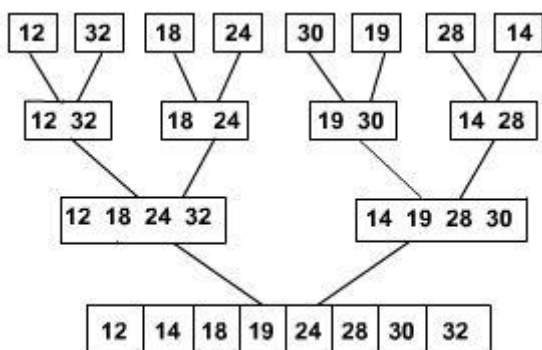
```

Step 1:   i = 0
Step 2:   j = 0
Step 3:   k = 0
Step 4:   while (i < p) && (j < q) do{
Step 5:       if (a[i] < b[j]) {
Step 6:           c[k] = a[i] ;
Step 7:           i = i + 1 ; }
Step 8:       else {
Step 9:           c[k] = b[j]
Step 10:          j = j + 1 }
Step 11:          k = k + 1 }
Step 12:  if (i == p) then
Step 13:      while ( j < q) do{
Step 14:          c[k] = b[j] ;
Step 15:          k = k + 1 ;
Step 16:          j = j + 1 ;
              }
Step 17:  else
Step 18:      while(i < p) do
Step 19:          { c[k] = a[i] ;
Step 20:            k = k + 1 ;
Step 21:            i = i + 1 ;
              }

```

We can assume each element in a given array as a sorted sub array. Take adjacent arrays and merge to obtain a sorted array of two elements. Next step, take adjacent sorted arrays, of size two, in pair and merge them to get a sorted array of four elements. Repeat the step until whole array is sorted.

Following figure illustrates the procedure.



- Assume that procedure Merge1 (a, i, j, k) takes two sorted arrays a[i..j] and a[j+1..k] as an input and puts their merged output in the array a[i..k]. That is, in the same locations. Pseudocode of the merge sort is given below.

Algorithm Merge-Sort (a, p, q)

```

Step 1:  if (p < q) {
Step 2:      mid = (p+q)/2 ;
Step 3:      Merge-Sort(a, p, mid) ;
Step 4:      Merge-Sort(a, mid+1, q) ;
Step 5:      Merge(a,p,mid, q) ;
          }

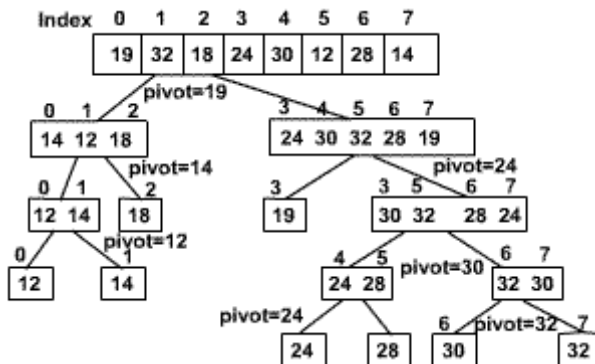
```

- The above procedure sorts the elements in the sub array a[p..q].
- To sort given any a[n] invoke the algorithm with parameters (a, 0, n -1).

5.5 Quick Sort

Quick Sort:

- Another very interesting sorting algorithm is Quick sort. Its worst time complexity is $O(n^2)$, but it is faster than many optimal algorithms for many input instances.
- It is another divide and conquer algorithm.
- Main idea of the algorithm is to partition the given elements into two sets such that the smaller numbers into one set and larger number into another. This partition is done with respect to an element called pivot. Repeat the process for both the sets until each partition contains only one element.



Pseudocode of the partition is given below.

Algorithm partition (a, p, q)

Step 1: pivot = a[p];

Step 2: $i = p-1 ; j=q+1;$
 Step 3: while ($i < j$)
 Step 5: repeat
 Step 6: $i = i + 1$
 Step 7: until $a[i] \geq \text{pivot};$
 Step 8: repeat
 Step 9: $j = j - 1$
 Step 10: until $a[j] \leq \text{pivot};$
 Step 11: if ($i < j$) then
 Step 12: swap ($a[i], a[j]$);
 Step 13: return j

Algorithm quick-sort (a, p, q)

Step 1: if ($p < q$) {
 Step 2: $\text{mid} = \text{partition}(a,p,q);$
 Step 3: quick-sort (a, p, mid);
 Step 4: quick-sort ($a, \text{mid}+1, q$);
 }

5.6 Radix Sort

Radix Sort:

- All the algorithms discussed till now are comparison based algorithms. That is, comparison is the key for sorting.
- Assume that the input numbers are three decimal digits. First we sort the numbers using least significant digit. Then by next significant digit and so on.
- See following example for illustration.

327	470	418	146
476	382	327	173
285	173	146	259
418	285	259	285
568	476	568	327
382	146	470	382
146	327	173	418
259	418	476	470
173	568	382	476
470	259	285	568
Input	Sorted by LS digit	Sorted by Middle digit	Sorted by Most Sig digit

- Next question is how to sort with respect to a least significant digit or any significant digit?
- We can use any sorting algorithm we have discussed.
- Another way by maintaining a BIN for each digit from 0 to 9. If the digit is X put the number in BIN X. Concatenate the BINs from 0 to 9 to get a sorted list of numbers of that significant digit.
- The method is given in the following pseudo code

Algorithm `radix-sort (a)`

```

Step 1:   for i = 0 to 9 do
Step 2:     empty-bin(i);
Step 3:     for position = least significant digit to most significant digit
Step 4:       for i = 1 to n do
Step 5:         x = digit in the position of a[i];

Step 6:           put a[i] in BIN x;
Step 7:           i = 1;
Step 8:           for j = 0 to 9 do
Step 9:             while (BIN(j) <> empty) do
Step 10:              a[i] = get-element(BIN(j));
Step 11:              i = i + 1;

```

- Procedure `get-element (bin)` gets the next element from the bin.
- We can use Queues for implementing Bins.

5.7 Problems- Sorting

1. Write a procedure for the merge procedure `Merge1 (a, i, j, k)`.
2. Write non-recursive procedure for merge sort.
3. Modify merge sort procedure to sort number in non-ascending order.
4. For what input, merge sort takes maximum number of comparisons.
5. Implement merge sort algorithm in c.
6. Trace the Quick sort algorithm on sorted array of elements 1 to 10.
7. Trace the Quick sort algorithm on elements 10, 9, 8, 1.
8. Write a program for quick sort method.
9. Implement the radix sort.
10. Design an algorithm to sort given names using radix sort.

UNIT VI: SEARCHING

6.1 Searching

Retrieval of information from a database by any user is very trivial aspect of all the databases.

To retrieve

Any data, first we have to locate it. This operation which finds the location of a given Element in a list is called searching.



If the element to be searched is found, then the search is successful otherwise it is unsuccessful.

We will discuss different the searching methods. The two standard ones are:

1. Linear Search
2. Binary search

6.2 Linear Search

Linear search is the most simple of all searching techniques. It is also called sequential search.

To find an element with key value='key', every element of the list is checked for key value='k'

Sequentially one by one. If such an element with key=k is found out, then the search is stopped.

But if we eventually reach the end of the list & still the required element is not found then also we terminate the search as an unsuccessful one.

The linear search can be applied for both unsorted & sorted list.

- Linear search for Unsorted list
- Linear search for sorted list

Linear Search for Unsorted List

In case of unsorted list, we have to search the entire list every time i.e. we have to keep on searching the list till we find the required element or we reach the end of the list. this is because as elements are not in any order, so any element can be found just anywhere.

Algorithm:

```
linear search(int x[],int n,int key)
{
```

```

int i,flag = 0;
for(i=0;i < n ; i++)
{
if(x[i]==key)
{
flag=1;
break;
}
}
if(flag==0)
return(-1);
else
return(1);
}

```

Complexity:

The number of comparisons in this case is $n-1$. So it is of $O(n)$. The implementation of algo is simple but the efficiency is not good. Every time we have to search the whole array (if the element with required value is not found out).

Linear Search for Sorted List

The efficiency of linear search can be increased if we take a previously sorted array say in ascending order. Now in this case, the basic algorithm remains the same as we have done in case of an unsorted array but the only difference is we do not have to search the entire array every time. Whenever we encounter an element say y greater than the key to be searched, we conclude that there is no such element which is equal to the key, in the list. This is because all the elements in the list are in ascending order and all elements to the right of y will be greater or equal to y , ie greater than the key. So there is no point in continuing the search even if the end of the list has not been reached and the required element has not been found.

Linear search(int x[], int n, int key)

```

{
int i, flag=0;
for(i=0; i < n && x[i] <= key; i++)
{
if(x[i]==key)
{

```

```

flag=1;
break;
}
}
if(flag==1) /* Unsuccessful Search*/
return(-1);
else return(1); /*Successful search*/
}

```

Illustrative Explanation:

The array to be sorted is as follows:

21 35 41 65 72

It is sorted in ascending order. Now let key = 40. At first 21 is checked as $x[0]=21$.

It is smaller than 40. So next element is checked which is 35 that is also smaller than 40. So now 41 is checked. But $41 > 40$ & all elements to the right of 41 are also greater than 40. So we terminate the search as an unsuccessful one and we may not have to search the entire list.

Complexity:

Searching is NOT more efficient when key is in present in the list in case when the search key value lies between the minimum and the maximum element in the list. The Complexity of linear search both in case of sorted and unsorted list is the same. The average complexity for linear search for sorted list is better than that in unsorted list since the search need not continue beyond an element with higher value than the search value.

6.3 Binary Search

The most efficient method of searching a sequential file is binary search. This method is applicable to elements of a sorted list only. In this method, to search an element we compare it with the center element of the list. If it matches, then the search is successful and it is terminated. But if it does not match, the list is divided into two halves. The first half consists of 0th element to the center element whereas the second list consists of the element next to the center element to the last element. Now It is obvious that all elements in first half will be $<$ or $=$ to the center element and all element elements in the second half will be $>$ than the center element. If the

element to be searched is greater than the center element then searching will be done in the second half, otherwise in the first half.

Same process of comparing the element to be searched with the center element & if not found then dividing the elements into two halves is repeated for the first or second half. This process is repeated till the required element is found or the division of half parts gives a single element.

Algorithm for Binary Search.

Illustrative Explanation:

Let the array to be sorted is as follows:

11 23 31 33 65 68 71 89 100

Now let the element to be searched ie key = 31 At first $hi=8$ $low=0$ so $mid=4$ and $x[mid]= 65$ is the center element but $65 > 31$. So now $hi = 4-1=3$. Now $mid= (0 + 3)/2 = 1$, so $x[mid]= 23 < 31$. So again $low= 1 + 1 = 2$. Now $mid = (3 + 2)/2 = 2$ & $x[mid]= 31 = key$. So the search is successful. Similarly had the key been 32 it would have been an unsuccessful search.

Complexity:

This is highly efficient than linear search. Each comparison in the binary search reduces the no. of possible candidates by a factor of 2. So the maximum no. of key comparisons is equal to $\log_2(n)$ approx. So the complexity of binary search is $O(\log n)$.

Limitations:

Binary search algorithm can only be used if the list to be searched is in array form and not linked list. This is because the algorithm uses the fact that the indices of the array elements are consecutive integers. This makes this algorithm useless for lists with many insertions and deletions which can be implemented only when the list is in the form of a linked list.

But this can be overcome using padded list.

Padding:

To use binary search in the presence of large no. of insertions & deletions we use a data structure known as padded list. Two arrays, an element array & a parallel flag array are used. The element array consists of the sorted keys in the table with empty slots initially evenly interspersed among

the keys of the table to allow for growth. The empty slot is indicated by a 0 value in the corresponding flag array element and a full slot is indicated by the value 1. Each empty slot in the element array contains a key value $>$ or $=$ to the key value in the previous full slot & $<$ than the key value in the following full slot. Thus the element array is sorted. So binary search can be done on it.

Search:

To search for an element, perform a binary search on the element array. If the argument key is not found, the element does not exist in the list. If it is found & corresponding flag is 1, then the search is successful. If the flag is 0, check if the previous full slot contains the argument key. If yes then search is successful otherwise it's an unsuccessful search.

Insertion:

To insert an element locate the position. If its empty insert the element there, reset the flag to 1 & adjust the contents of all previous contiguous empty positions to equal the contents of the previous full element & of all the following contiguous empty positions to the inserted element, leaving their flags at 0. If the position is full shift forward by 1 position all the following elements upto the first empty position to make place for new element.

Deletion:

Deleting simply involves locating a key and changing its associated flag to 0.

6.4 Problems-Searching

1. Write a program to find out a number in a given unsorted array. Input will be a random number; Output should locate its position.
2. Write a program to perform linear search in a given sorted array.
3. The maximum number of comparisons in binary search is limited to
4. Write down the case(s) when binary search can't be implemented.
5. What is the difference between internal & external sorting?
6. Arrange following sorting methods in the order of their running time, number of comparisons made & worst case complexity.....

Selection sort, Bubble sort, Quick sort, Insertion sort.

7. Implement quick sort.
8. What would be the worst case scenario for bubble sort program?
9. Write a program that sorts the elements of a two dimensional array
 - Row wise
 - Column wise
10. Suppose an array contains N elements. Given a number X that may occur several times in the array. Find
 - The number of occurrence of X in the array.
 - The position of first occurrence of X in the array.
11. WAP that determines the number of spaces " " in a given input & removes it. Input should be like STCGSS TGCCGT GTCCCTTS GTT GGSST GSGSSGGG output -> 4
STCGSSTGCCGTGTCCCTTS GTT GGSST
12. WAP that replaces all S's in the last program with H.
13. WAP that makes a dictionary type of listing from a given set of words. (Hint: Use 2-D array).
14. Take any five dates in dd\mm\yyyy format, and arrange them in ascending order. Give the outline of the work required.
15. Write a program that accepts a set of 5 records for students. Ask user to enter name, age and height of a student. Sort these records in ascending order of their names. If the names are alike then sort according to their age.

UNIT VII: GRAPHS I: REPRESENTATION AND TRAVERSAL

7.1 Graphs I: Representation and Traversal

- Graph consists of a non empty set of points called vertices and a set of edges that link vertices.
- Formal Definition :

Definition: A graph $G = (V, E)$ consists of

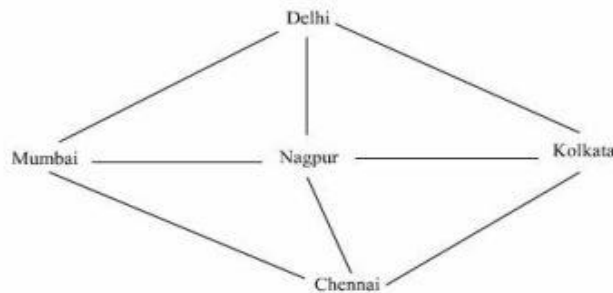
- a set $V = \{v_1, v_2, \dots, v_n\}$ of $n > 1$ vertices and
- a set of $E = \{e_1, e_2, \dots, e_m\}$ of $m > 0$ edges
- such that each edge e_k corresponds to an un ordered pair of vertices (v_i, v_j)
- where $0 < i, j \leq n$ and $0 < k \leq m$.
- A road network is a simple example of a graph, in which vertices represent cities and roads connecting them correspond to edges.

$V = \{ \text{Delhi, Chennai, Kolkata, Mumbai, Nagpur} \}$

$E = \{ (\text{Delhi, Kolkata}), (\text{Delhi, Mumbai}), (\text{Delhi, Nagpur}), (\text{Chennai, Kolkata}), (\text{Chennai, Mumbai}), (\text{Chennai, Nagpur}), (\text{Kolkata, Nagpur}), (\text{Mumbai, Nagpur}) \}$

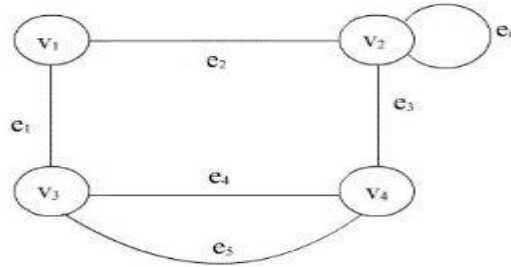
GRAPHS Representation:

- Graphs can be represented by a diagram. For example, the graph representation of the above road network is depicted in the figure below.



GRAPHS Representation: Definitions

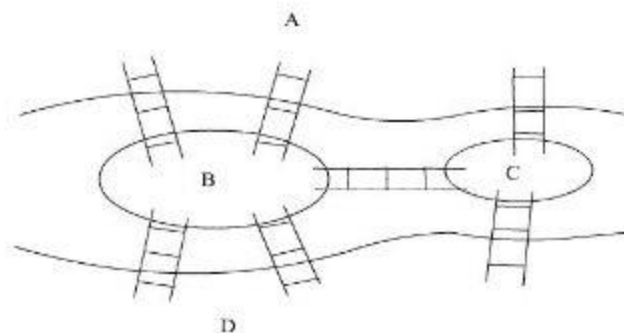
- Loop is an edge that connects a vertex to itself. Edge e_6 in the figure below is a loop.
- Edges with same end vertices are called parallel edges. Edges e_4 and e_5 are parallel edges in the below figure.



7.2 Königsberg Bridge Problem

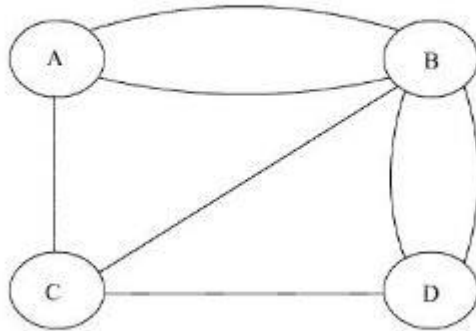
GRAPHS Representation: Königsberg Bridge Problem

- A Graph without loops and parallel edges is called a simple graph.
- A graphs with isolated vertices (no edges) is called null graph.
- Set of edges E can be empty for a graph but not set of vertices V .
- Graphs can be used in wide ranges of applications. For example consider the K Ö nigsberg Bridge Problem.
- K ö nigsberg Bridge Problem: Two river islands B and C are formed by the Pregel river in K ö nigsberg (then the capital of East Prussia, Now renamed Kaliningrad and in west Soviet Russia) were connected by seven bridges as shown in the figure below. Start from any land areas walk over each bridge exactly once and return to the starting point.



GRAPHS Representation:

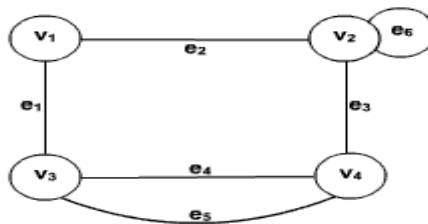
- Euler (1707-1783) in 1736 formulated the Königsberg bridge problem as a graph problem and solved.
- Represent land area by vertices and bridges connecting them by edges. We get the following graph.



- The problem is nothing but a children's game of drawing a figure without lifting pen from the paper and without retracing a line.

GRAPHS Representation: Definition

- Incidence: if a vertex v_i is an end vertex of an edge e_k , we say vertex v_i is incident on e_k and e_k is incident on v_i .
- e_1 is incident on v_1 and v_3 in the below figure.
- v_4 is incident on e_3 , e_4 , and e_5 in the figure below.
- **Degree:** Degree of a vertex is number of edges incident on it, with loops counted twice.
- $d(v_1) = 2$, $d(v_2) = 4$, $d(v_3) = 3$, and $d(v_4) = 3$ in the figure below.



- Adjacent Edges: Two non parallel edges are adjacent if they have a vertex in common.
 - e1 and e2 , e2 and e6 , e2 and e3 , e1 and e4 are adjacent edges in the above diagram.
- Adjacent vertices: Two vertices are adjacent if they are connected by an edge.
 - v1 and v3 , v1 and v2 , v2 and v4 are adjacent vertices in the above diagram.

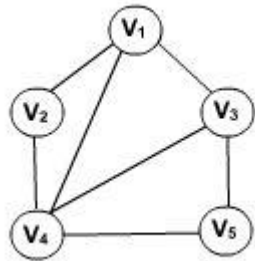
Graph Representation:

There are several different ways to represent graphs in a computer. Two main representations are Adjacency Matrix and Adjacency list.

- Adjacency Matrix Representation:
 - An adjacency matrix of a graph $G=(V,E)$ (let $V = \{ v_1 , v_2 \dots, v_n \}$) is a $n \times n$ matrix A , such that

$$A [i, j] = 1 \quad \text{if there is edge between } v_i \text{ and } v_j.$$

$$0 \quad \text{other wise}$$

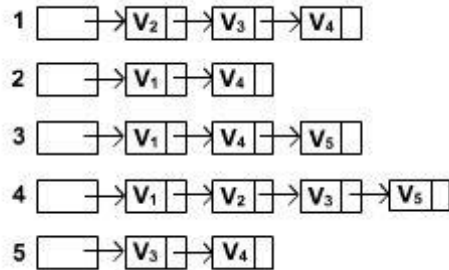
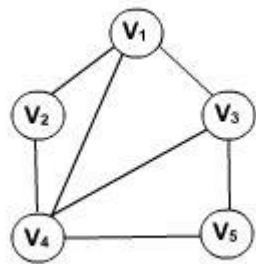


	1	2	3	4	5
1	0	1	1	1	0
2	1	0	0	1	0
3	1	0	0	1	1
4	1	1	1	0	1
5	0	0	1	1	0

- Accessing any element can be done quickly, in constant time.
- Size of the adjacency matrix is $n \times n$ for a graph of n vertices, which is independent of number of edges.
- The following representation uses memory efficiently for a sparse graph.

Adjacency List Representation:

- It consists of a list of vertices, which can be represented either by linked list or array.
 - For each vertex, adjacent vertices are represented in the form of a linked list.
- An example is given below.



- Size of the adjacency lists is proportional to number of edges and vertices.
- It is an efficient memory use for sparse graphs.
- Accessing an element takes linear time, since it involves traversal of linked list.

Graph Traversals

- Many graph problems can be solved by visiting each and every vertex in a systematic manner.
- There are two main methods to traversal graphs, Depth First Search (DFS) and Breadth First Search (BFS).
- **Depth First Search (DFS):** Initially all vertices of the graph are unvisited. Start visiting the graph from any vertex, say v . For each unvisited adjacent vertex of v , search recursively. This process stops when all vertices reachable from v are visited. If there are any more unvisited vertices are present select any unvisited vertex and repeat the same search process.
- This method is called depth first search since searching is done forward (deeper) from current node. The distance from start vertex is called depth

Algorithm DFS (G)

```

for i = 1 to n do // Initialize all vertices are unvisited
    status[i] = unvisited
    parent[i] = NULL

for i = 1 to n do

```

```
if (status[i] == unvisited) // If there exists an unvisited vertex, start traversal
    DF-Travel(i)
```

Algorithm DF-Travel (v)

```
status[v] = visited
for each vertex u adjacent to v do
    if status[u] == unvisited then
        parent[u] = v
        DF-Travel ( u )
```

- Breadth First Search (BSF): Initially all vertices of the graph are unvisited. Start visiting the graph from any vertex, say v . Visit each unvisited adjacent vertex of v . Repeat the process for each vertex visited. This process stops when all vertices reachable from v are visited. If there is any more unvisited vertices are present select any unvisited vertex and repeat the same search process.
- This method is called breadth first search, since it works outward from a center point, much like the ripples created when throwing a stone into a pond. It moves outward in all directions, one level at a time.
- BSF can be implemented using queue to maintain set vertices visited first time and their adjacent vertices are not explored for them.

■ **Algorithm BFS (G)**

```

for i = 1 to n do // Initialize all vertices are unvisited
    status[i] = unvisited
    parent[i] = NULL

for i = 1 to n do

    if (status[i] == unvisited) // If there exists an unvisited
        vertex, start traversal

```

BF-Travel(i)

■ **Algorithm DF-Travel (v)**

```

status[v] = visited
Q = {v}

while ( Q is not empty ) do {
    u = delete-queue(Q)

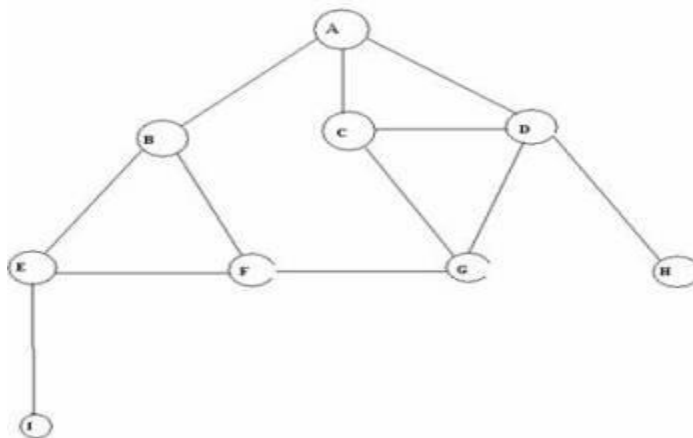
    for each vertex w adjacent to u do

        if status[w] == unvisited then {
            status[w] = visited
            insert-queue(Q, w)
            parent[w] = u
        }
    }
}

```

7.3 Problems- Graphs I: Representation and Traversal

1. Trace the DFS, BFS on the following graph



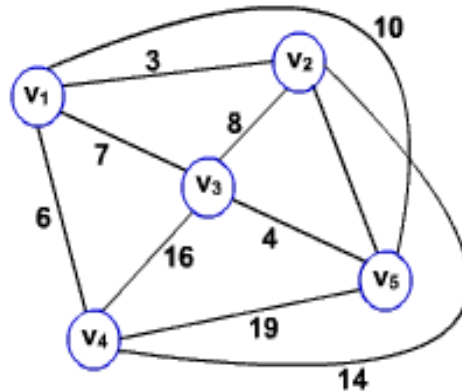
2. Implement the DFS/BFS algorithm.
3. Design an algorithm to find given graph is cyclic or not.

UNIT VIII: GRAPHS II: BASIC ALGORITHMS

8.1 Graphs II: Basic Algorithms

Introduction

- Assume, in a country, there are many oil wells. It is necessary to connect them by a pipeline. Main objective of designing should be minimization of total cost of laying pipeline.
- This problem can be modeled as a graph problem. The set of oil wells are represented by vertices. The pipeline between any pair of oil wells is corresponding to the edge between the vertices denoting these oil wells.
- The cost of laying a pipeline between a pair of oil wells is the weight of the edge joining these oil wells.
- Consider the following example of five oil wells. The weight on the edges denotes the cost of the laying pipeline between their respective oil wells.



- The best way of connecting them is shown in the following figure.

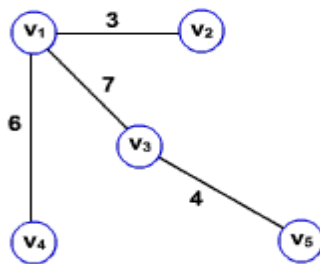


Figure 2

- The cost of the laying the above pipe line connecting these oil wells is 20. Any other way of connecting increases the cost.
- Resultant graph is a tree which spans all vertices of the graph. Hence called spanning tree.
- The spanning tree with minimum cost is called Minimum Spanning tree (MST).

8.2 Minimum Spanning Tree

Minimum Spanning Tree (MST)

- Problem: Given a weighted undirected graph G , find the minimum spanning tree.
- One of the main property of a tree is cycle freeness.
- This property is exploited during construction of a MST for a given graph. That is, to obtain a spanning tree of a given graph, starts from a null graph add edges one after the other without forming cycles.
- If the edges considered for adding is in the increasing order we get a MST.
- Consider the following weighted graph.

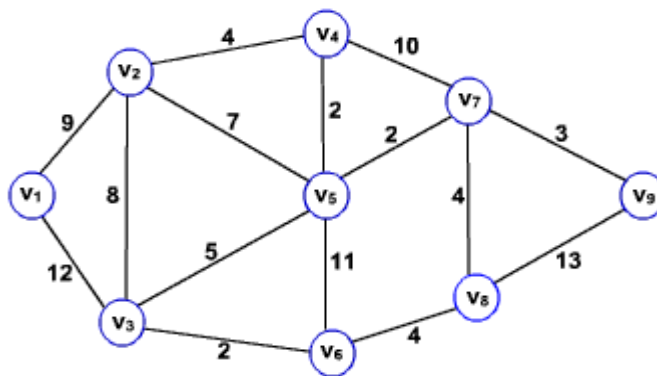


Figure 3

- In the above figure there are two least weighted edges. We can choose any one of them. Lets choose edge (v_4, v_5) .
- Next least weighted edge is between (v_3, v_6) , add this edge to MST edges, since it is not creating any cycle.
- Next edge is (v_7, v_9) . Repeat the processes of adding edges till we found $n-1$ edges without forming any cycle. The number of vertices's in the graph is n .
- This is method is Kurskal's algorithm, named after the inventor.

- The following sequence of figures shows a way of adding edges to obtain a MST.

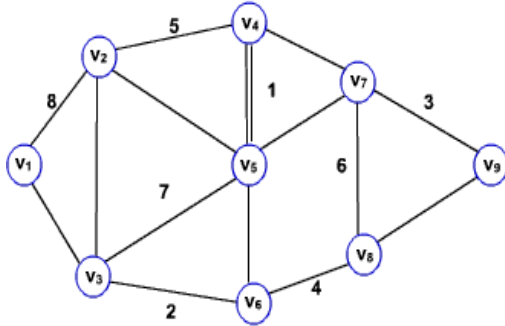


Figure 4

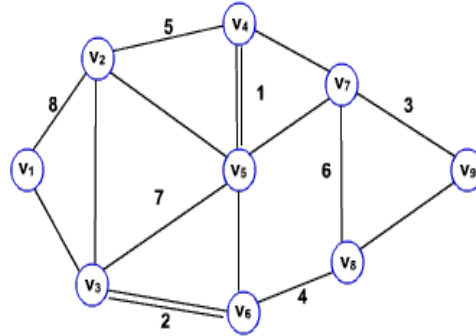


Figure 5

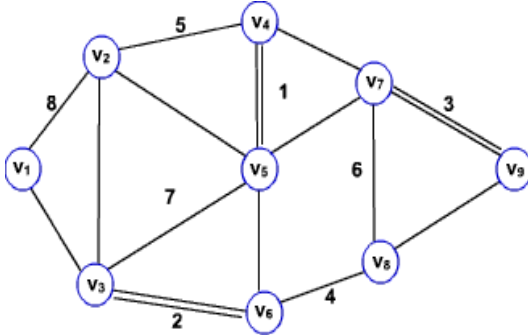


Fig:6

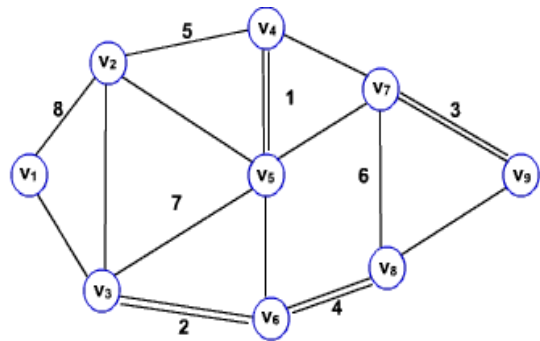


Fig:7

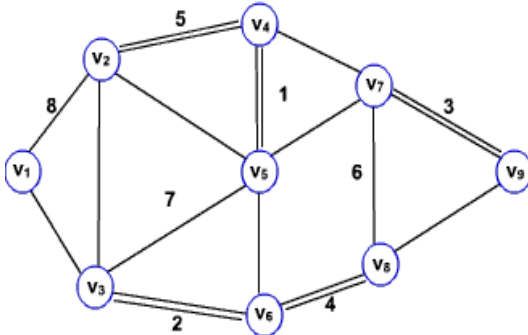


Fig : 8

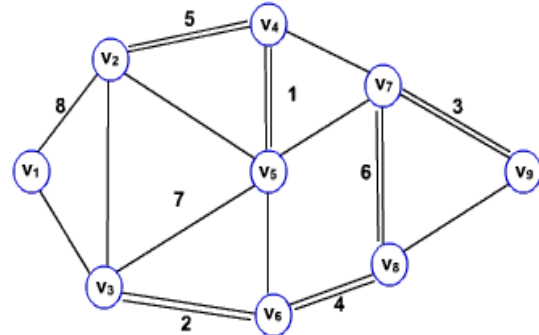


Fig : 9

- The weight of the MST is 33.

The above method is depicted in the algorithm below.

algorithm `Kruskal_MST(G)`

Step 1 : `Sort_E = Sort` the set E of edges by nondecreasing order

Step 2: `MST = NULL`

Step 3: for each edge (u, v) in E in the nondecreasing order do

Step 4: if (`MST + (u,v)` does not contain cycle) then

Step 5: `MST = MST + (u,v)`

- This algorithm can also be viewed as follows: Initially, each vertex is component. Add an edge between a pair of components to make a bigger component without any cycle. The edge chosen is a least weighted. Repeatedly add edges one after the another till we have only one component.
- An implementation of the algorithm can be done using a set data structure. Each component is represented as a set. Add next least weighted edge (u,v) , if u and v are in two different components or sets. That is, make a bigger component by taking the union of the sets containing u and v .
- The Kurskal's algorithm using set data structure is given below.

algorithm IMP_Kurskal_MST(G)

- step 1: Sort_E= Sort the set E of edges by nondecreasing order.
- step 2: For each vertex v in V do
- Step 3: Make set(v)
- Step 4: for each edge (u, v) in E in the nondecreasing order do
- step 5: if (u and v are in different components) then
- step 6: Union (u, v)

8.3 Single Source Shortest Path

Single Source Shortest Path

- A traveler wishes to find a shortest distance between New Delhi and Visakhapatnam, in a road map of India in which distance between pair of adjacent road intersections are marked in kilometers.
- One possible solution is to find all possible routes between New Delhi and Visakhapatnam, and find the distance of each route and take the shortest.
- There are many possible routes between given cities. It is difficult to enumerate all of them.
- Some of these routes are not worth considering. For example, route through Cinnai, since it is about 800KM out of the way.

- The problem can be solved efficiently, by modeling the given road map as a graph and find the shortest route between given pair of cities as suitable (single source shortest paths) graph problem.
- **Graph Representation of the road map:** The intersections between the roads are denoted as vertices and road joining them as edge in the graph representation of the given road map.
- **Single Source Shortest Path problem:** Given a graph $G = (V, E)$, we want to find the shortest path from given source to every vertex in the graph.
- The problem is solved efficiently by Dijkstra using greedy strategy. The algorithm is known as Dijkstra's algorithm.
- The method works by maintaining a set S of vertices for which shortest path from source v_0 is found. Initially, S is empty and shortest distance to each vertex from source is infinity.
- The algorithm repeatedly selects a vertex v_i in $V - S$ to which path from source is shorter than other vertices in $V - S$, adds to S , and estimates the shortest distance to other vertices from v_i .

Algorithm Dijkstra_shortest_path(G, v_0)

Step 1: for each vertex v_i in V do

Step 2: $dist[v_i] = \text{infinity}$

Step 3: $distance[v_0] = 0$;

Step 4: for each vertex v_i in V do

Step 5: $insert(Q, v_i)$

Step 6: $S = \text{empty}$

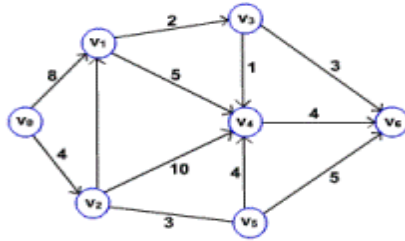
Step 7: while (Q not empty) do

Step 8: $v_i = \text{Min}(Q)$

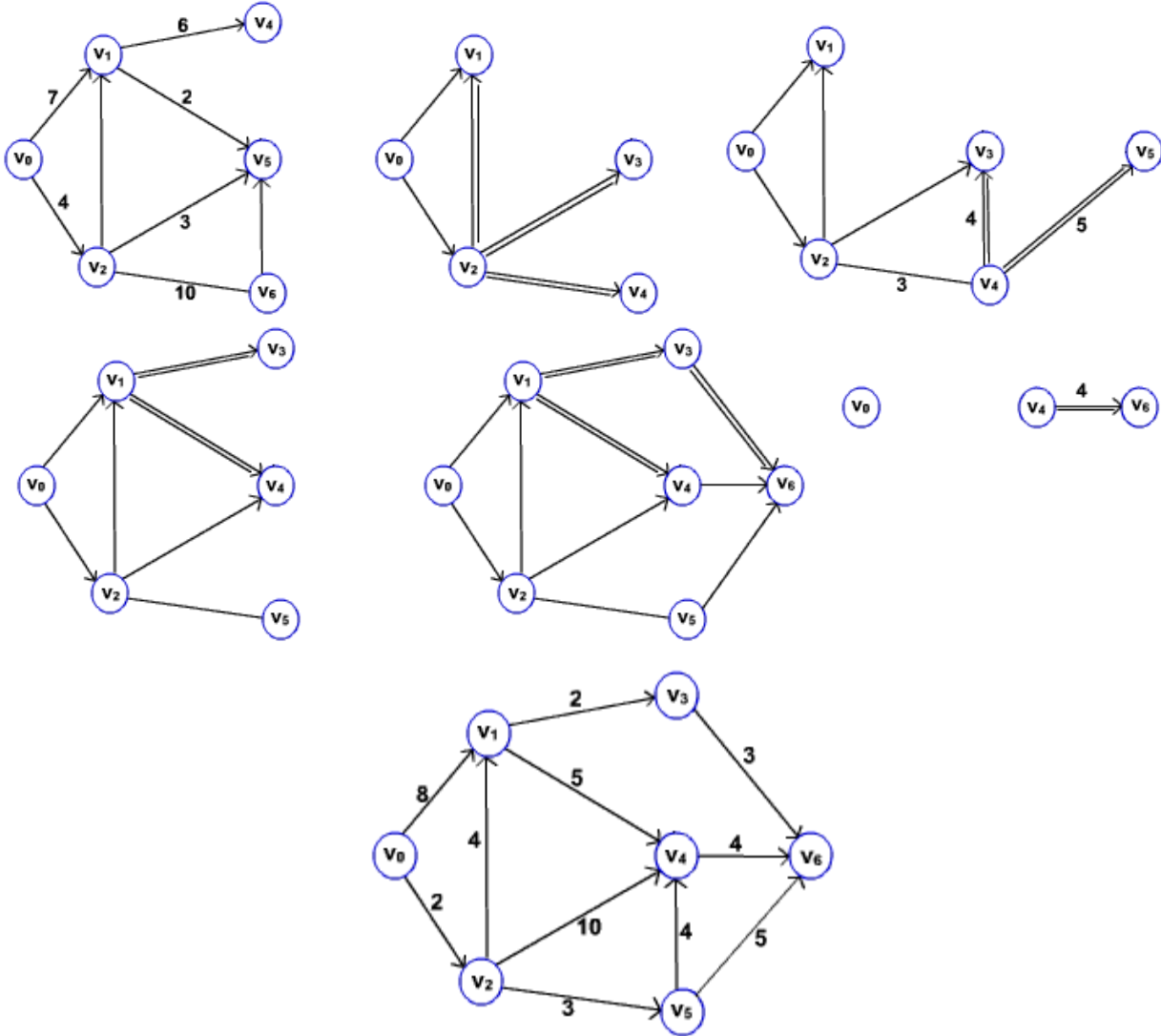
Step 9: for each vertex v_j adjacent to v_i do

Step 10: $distance[v_j] = \text{Minimum} (distance[v_j], distance[v_i] + w[v_i, v_j])$

Consider the following graph:



- The following sequence of figures shows the way of finding the shortest path from source vertex v_0 .



- The event queue in Dijkstra's algorithm can be implemented using array or heap.

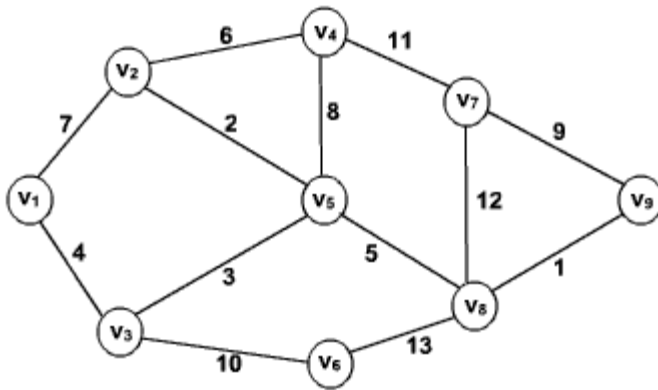
All pairs shortest path

- Problem: In a given weighted directed graph find the shortest path between every pair of vertices.

- This problem can be solved by repeatedly invoking the single source shortest path algorithm for each vertex as a source.
- Another way of solving the problem is using dynamic programming technique. See any algorithms book.

8.4 Problems- Graphs II: Basic Algorithms

1. Trace the Kruskal's algorithm on the following graph to find the MST. Show the partially constructed MST at each stage.



2. Implement the Kruskal's algorithms using set data structure.
3. Prove or disprove the statement "The MST of a graph is unique".

UNIT IX: BINARY TREES

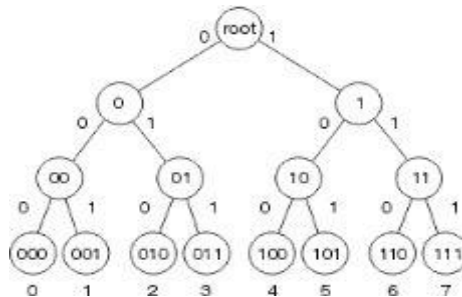
9.1 Binary Tree

A tree is a finite set of nodes having a distinct node called root.

Binary Tree is a tree which is either empty or has at most two subtrees, each of the subtrees also being a binary tree. It means each node in a binary tree can have 0, 1 or 2 subtrees. A left or right subtree can be empty.

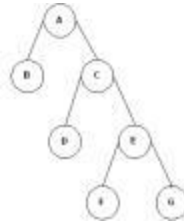
A binary tree is made of nodes, where each node contains a "left" pointer, a "right" pointer, and a data element. The "root" pointer points to the topmost node in the tree. The left and right pointers point to smaller "subtrees" on either side. A null pointer represents a binary tree with no elements -- the empty tree. The formal recursive definition is: a binary tree is either empty (represented by a null pointer), or is made of a single node, where the left and right pointers (recursive definition ahead) each point to a binary tree.

The figure shown below is a binary tree.



- It has a distinct node called root ie 2. And every node has either 0,1 or 2 children. So it is a binary tree as every node has a maximum of 2 children.
- If A is the root of a binary tree & B the root of its left or right subtree, then A is the parent or father of B and B is the left or right child of A. Those nodes having no children are leaf nodes. Any nodes say A is the ancestor of node B and B is the descendant of A if A is either the father of B or the father of some ancestor of B. Two nodes having same father are called brothers or siblings.
- Going from leaves to root is called climbing the tree & going from root to leaves is called descending the tree.

- A binary tree in which every non leaf node has non empty left & right subtrees is called a strictly binary tree. The tree shown below is a strictly binary tree.



The no. of children a node has is called its degree. The level of root is 0 & the level of any node is one more than its father. In the strictly binary tree shown above A is at level 0, B & C at level 1, D & E at level 2 & F & g at level 3.

The depth of a binary tree is the length of the longest path from the root to any leaf. In the above tree, depth is 3.

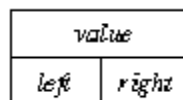
The other topics that will be covered regarding binary tree are listed below.

1. Representation of binary tree
2. Operations on a binary tree
3. Traversal of a binary tree

Representation of Binary Tree

The structure of each node of a binary tree contains one data field and two pointers, each for the right & left child. Each child being a node has also the same structure.

The structure of a node is shown below.



The structure defining a node of binary tree in C is as follows.

Struct node

```

{
struct node *lc ; /* points to the left child */
int data; /* data field */
struct node *rc; /* points to the right child */
}
  
```

There are two ways for representation of binary tree.

- Linked List representation of a Binary tree

- Array representation of a Binary tree

Array Representation of Binary Tree:

```

Struct node
{
struct node * lc;
int data;
struct node * rc;
};
struct node * buildtree(int);/* builds the tree*/
void inorder(struct node *);/* Traverses the tree in inorder*/
int
a[]={ 3,5,9,6,8,20,10,/0,/0,9,/0,/0,/0,/0,/0,/0,/0,/0};

void main( )
{
struct node * root;
root= buildtree(0);
printf("\n Inorder Traversal");
inorder(root);
getch( );
}

struct node * buildtree(int n);
{
struct node * temp=NULL;
if( a[n] != NULL)
{
temp = (struct node *) malloc(sizeof(struct node));
temp-> lc=buildtree(2n + 1);
temp-> data= a[n];
temp-> rc=buildtree(2n + 2);
}
return temp;
}

void inorder(struct node * root);
{
if(root != NULL)
{
if(root!= NULL)
{
inorder(roo-> lc);
printf("%d\t",root->data);
}
}
}

```

```
inorder(root->rc); }  
}
```

9.2 *Linked Representation of Binary Tree*

Linked Representation of Binary Tree:

Binary trees can be represented by links where each node contains the address of the left child and the right child. If any node has its left or right child empty then it will have in its respective link field, a null value. A leaf node has null value in both of its links.

The structure defining a node of binary tree in C is as follows.

Struct node

```
{  
struct node *lc ; /* points to the left child */  
int data; /* data field */  
struct node *rc; /* points to the right child */  
}
```

Binary Tree Traversal:

Traversal of a binary tree means to visit each node in the tree exactly once. The tree traversal is used in all the applications of it.



In a linear list nodes are visited from first to last, but a tree being a non linear one we need definite rules. There are a no. of ways to traverse a tree. All of them differ only in the order in which they visit the nodes.

The three main methods of traversing a tree are:

- Inorder Traversal

- Preorder Traversal
- Postorder Traversal

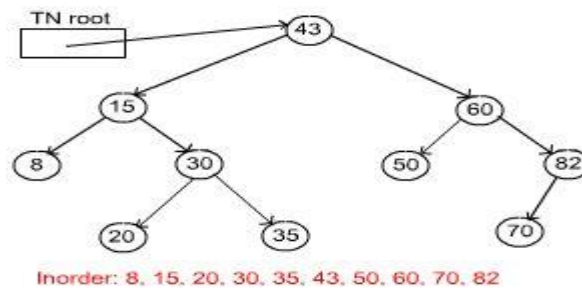
In all of them we do not require to do anything to traverse an empty tree. All the traversal methods are based on recursive functions since a binary tree is itself recursive as every child of a node in a binary tree is itself a binary tree.

Inorder Traversal:

To traverse a non empty tree in inorder the following steps are followed recursively.

- Visit the Root
- Traverse the left subtree
- Traverse the right subtree

The inorder traversal of the tree shown below is as follows.



Algorithm

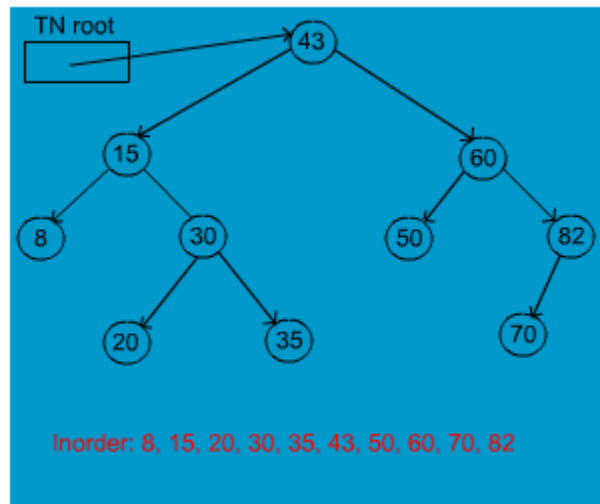
The algorithm for inorder traversal is as follows.

Struct node

```

{
struct node * lc;
int data;
struct node * rc;
};
void inorder(struct node * root);
{
if(root != NULL)
{
inorder(roo-> lc);
printf("%d\t",root->data);
inorder(root->rc);
}}
  
```

So the function calls itself recursively and carries on the traversal.



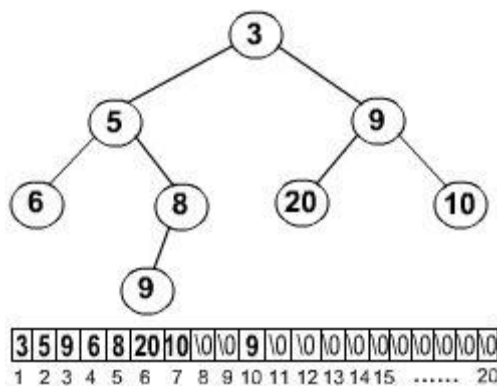
9.3 Array Representation of Binary Tree

Array Representation of Binary Tree:

A single array can be used to represent a binary tree.

For these nodes are numbered / indexed according to a scheme giving 0 to root. Then all the nodes are numbered from left to right level by level from top to bottom. Empty nodes are also numbered. Then each node having an index i is put into the array as its i th element.

In the figure shown below the nodes of binary tree are numbered according to the given scheme.



The figure shows how a binary tree is represented as an array. The root 3 is the 0th element while its left child 5 is the 1st element of the array. Node 6 does not have any child so its children i.e. 7th & 8th element of the array are shown as a Null value.

It is found that if n is the number or index of a node, then its left child occurs at $(2n + 1)$ th position & right child at $(2n + 2)$ th position of the array. If any node does not have any of its child, then null value is stored at the corresponding index of the array.

The following program implements the above binary tree in an array form. And then traverses the tree in inorder traversal.(for traversal see Traversal, Inorder traversal)

```

Struct node
{
struct node * lc;
int data;
struct node * rc;
};
struct node * buildtree(int);/* builds the tree*/
void inorder(struct node *);/* Traverses the tree in inorder*/
int
a[]={ 3,5,9,6,8,20,10,/0,/0,9,/0,/0,/0,/0,/0,/0,/0,/0};

void main( )
{
struct node * root;
root= buildtree(0);
printf("\n Inorder Traversal");
inorder(root);
getch( );
}

struct node * buildtree(int n);
{
struct node * temp=NULL;
if( a[n] != NULL)
{
temp = (struct node *) malloc(sizeof(struct node));
temp-> lc=buildtree(2n + 1);
temp-> data= a[n];
temp-> rc=buildtree(2n + 2);
}
return temp;
}

void inorder(struct node * root);
{
if(root != NULL)
{

```

```

if(root!= NULL)
{
inorder(roo->lc);
printf(“%d\t”,root->data);
inorder(root->rc);
}
}

```

Operation on Binary Tree:

Operations on Binary Tree are follows

- Searching
- Insertion
- Deletion
- Traversal
- Sort

Searching

Searching a binary tree for a specific value is a process that can be performed recursively because of the order in which values are stored. At first examining the root. If the value is equals the root, the value exists in the tree. If it is less than the root, then it must be in the left subtree, so we recursively search the left subtree in the same manner. Similarly, if it is greater than the root, then it must be in the right subtree, so we recursively search the right subtree. If we reach a leaf and have not found the value, then the item does not lie in the tree at all.

Here is the search algorithm

```

search_btree(node, key):
if node is None:
return None // key not found
if key < node.key:
return search_btree(node.left, key)
else if key > node.key:
return search_btree(node.right, key)
else : // key is equal to node key
return node.value // found key

```


Insertion

The way to insert a new node in the tree, its value is first compared with the value of the root. If its value is less than the root's, it is then compared with the value of the root's left child. If its value is greater, it is compared with the root's right child. This process continues, until the new node is compared with a leaf node, and then it is added as this node's right or left child, depending on its value.

Another way is examine the root and recursively insert the new node to the left subtree if the new value is less than or equal to the root, or the right subtree if the new value is greater than the root.

Deletion

There are several cases to be considered:

Deleting a leaf: If the key to be deleted has an empty left or right subtree, Deleting the key is easy, we can simply remove it from the tree.

Deleting a node with one child: Delete the key and fill up this place with its child.

Deleting a node with two children: Suppose the key to be deleted is called K . We replace the key K with either its in-order successor (the left-most child of the right subtree) or the in-order predecessor (the right-most child of the left subtree). We find either the in-order successor or predecessor, swap it with K , and then delete it. Since either of these nodes must have less than two children (otherwise it cannot be the in-order successor or predecessor), it can be deleted using the previous two cases.

Sort

A binary tree can be used to implement a simple but inefficient sorting algorithm. We insert all the values we wish to sort into a new ordered data structure.

UNIT X: HEAP SORT

10.1 Heap Sort

- Heap sort is an efficient sorting algorithm with average and worst case time complexities are in $O(n \log n)$.
- Heap sort is an in-place algorithm i.e. does not use any extra space, like merge sort.
- This method is based on a data structure called Heap.
- Heap data structure can also be used as a priority queue.

Heap:

- A binary heap is a complete binary tree in which each node other than root is smaller than its parent.
- Heap example:

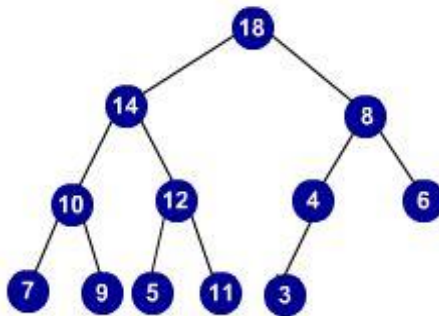


Figure 1

Heap Representation:

- A Heap can be efficiently represented as an array
- The root is stored at the first place, i.e. $a[1]$.
- The children of the node i are located at $2*i$ and $2*i + 1$.
- In other words, the parent of a node stored in i th location is at $\text{floor}\left(\frac{i}{2}\right)$.
- The array representation of a heap is given in the figure below.

1	2	3	4	5	6	7	8	9	10	11	12
18	14	8	10	12	4	6	7	9	5	11	3

Figure 2

Heapification

- Before discussing the method for building heap of an arbitrary complete binary tree, we discuss a simpler problem.
- Let us consider a binary tree in which left and right subtrees of the root satisfy the heap property, but not the root. See the following figure.
- Now the question is how to transform the above tree into a heap?
- Swap the root and left child of root, to make the root satisfy the heap property.
- Then check the subtree rooted at left child of the root is heap or not. If it is, we are done. If not, repeat the above action of swapping the root with the maximum of its children.
- That is, push down the element at root till it satisfies the heap property.
- The following sequence of figures depicts the heapification process.

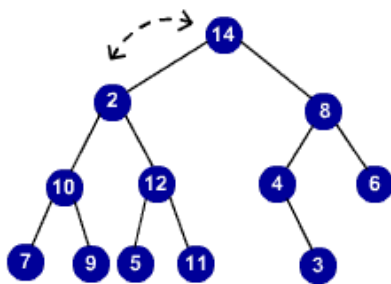


fig 4

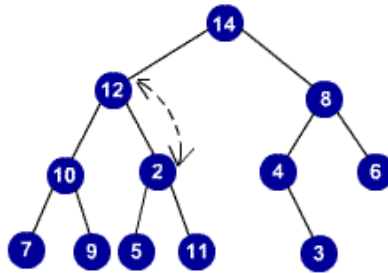


Fig : 4.1

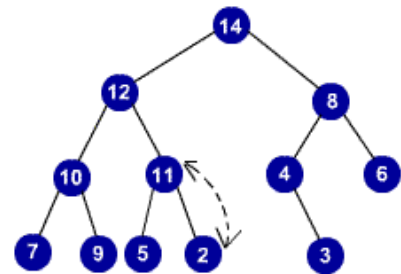


Fig : 4.2

algorithm: Heapification (a,i,n)

Step 1 : left = 2i

Step 2 : right = 2i + 1

Step 3 : if (left < n) and (a[left] > a[i]) then

Step 4 : maximum = left

Step 5 : else

Step 6 : maximum = i

Step 7 : if (right < n) and (a[right] > a[maximum]) then

Step 8 : maximum = right

Step 9 : if (maximum != i) then

Step 10 : swap(a[i],a[maximum])

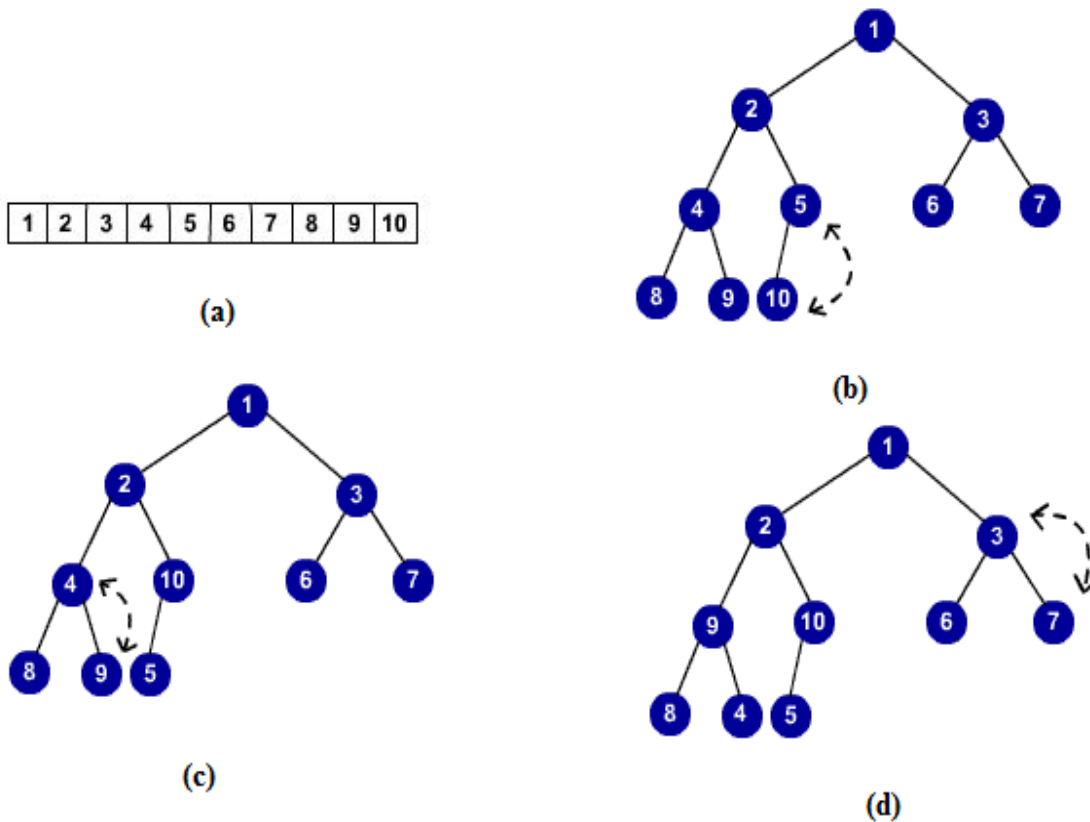
Step 11 : heapfication(a, maximum, n)

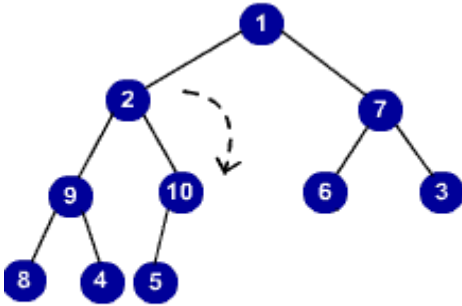
The time complexity of heapfication is $O(\log n)$.

Build Heap

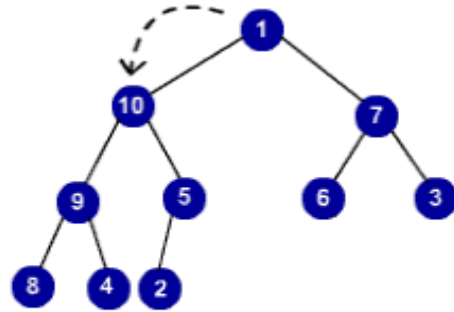
- Heap building can be done efficiently with bottom up fashion.
- Given an arbitrary complete binary tree, we can assume each leaf is a heap.
- Start building the heap from the parents of these leaves. i.e., heapify subtrees rooted at the parents of leaves.
- Then heapify subtrees rooted at their parents. Continue this process till we reach the root of the tree.

The following sequence of the figures illustrates the build heap procedure.

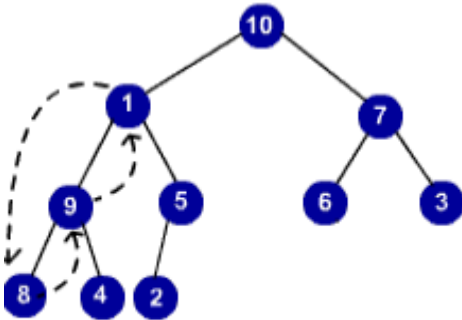




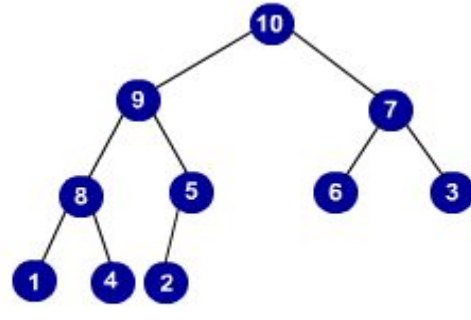
(e)



(f)



(g)



(h)

algorithm : build_heap(a,i,n)

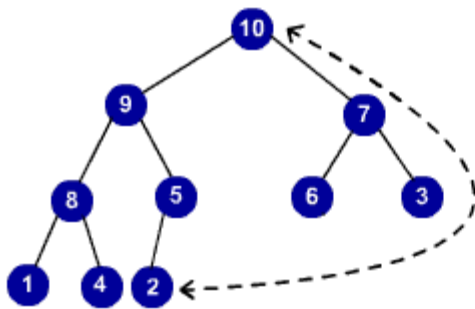
step 1 : for j = down to 1 do

Step 2 : heapification(a,j,n)

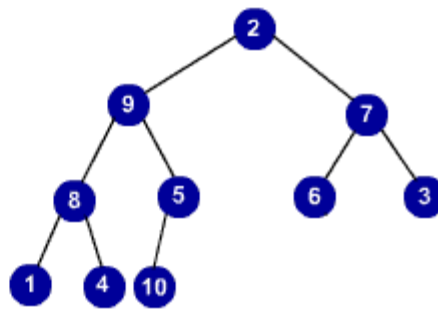
The time complexity of the build heap is in $O(n)$.

Heap sort

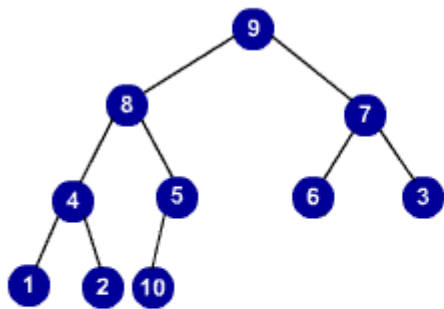
- Given an array of n element, first we build the heap.
- The largest element is at the root, but its position in sorted array should be at last. So, swap the root with the last element and heapify the tree with remaining n-1 elements.
- We have placed the highest element in its correct position. We left with an array of n-1 elements. Repeat the same of these remaining n-1 elements to place the next largest element in its correct position.
- Repeat the above step till all elements are placed in their correct positions.



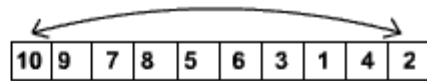
(a)



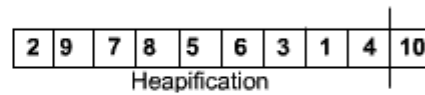
(b)



(c)



(d)



(e)

- Pseudocode of the algorithm is given below.

Algorithm Heap_Sort(a,i)

build_heap(a,i)

for j = i down to 1 do

swap (a[1],a[j])

heapification(a,1,j-1)

The time complexity of the heap sort algorithm is in $O(n \log n)$

Priority Queue

- Let consider a set S of elements, such that each element has priority.
- We want to design a data structure for these elements such that the highest priority element should be extracted/ deleted efficiently.

- We can use heap for this purpose since highest element, is always at the root, which can be extracted quickly.
- Pseudocode for extracting the maximum from the priority queue P is given below. Let the global variable size maintains the number of elements in P.

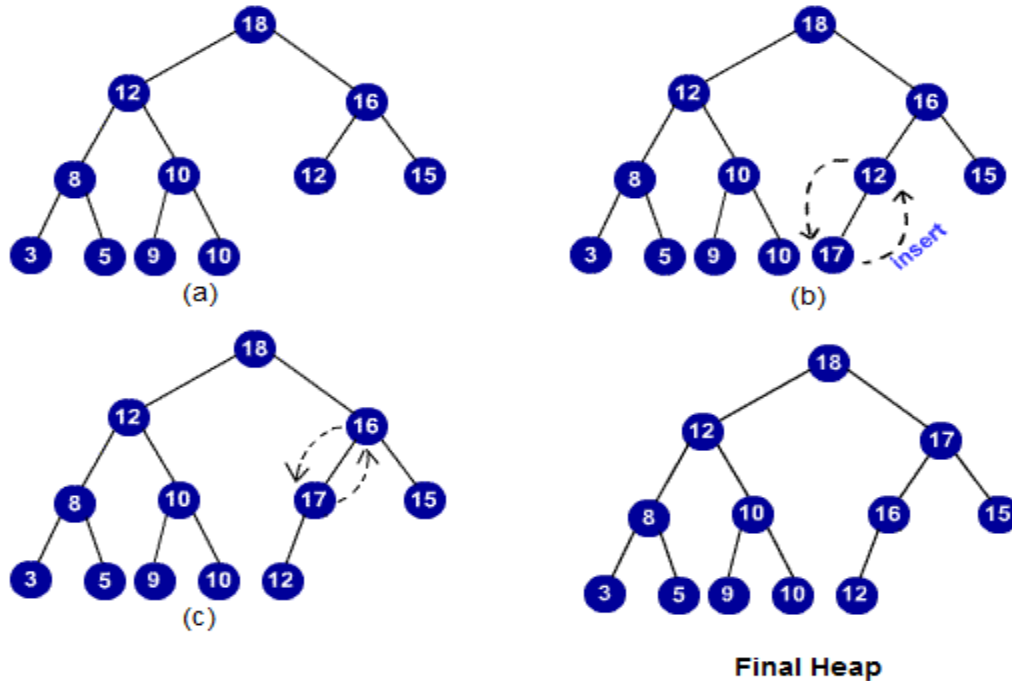
Algorithm : Max_Extract(P)

```

max = p[1]
P[1] = P[size]
size = size -1
heapification(P,1,size)
return (max)

```

- After extracting the maximum, we have to maintain remaining elements in the priority queue. So we heapify before returning the maximum.
- Other operation to be supported is to insert an element into the priority queue.
- Inserting an element into the priority queue can be done easily.
- Insert the new element as a new leaf, and push this up till it satisfies the heap property.
- The following sequence of figures illustrates the inserting procedure.



- The pseudo code is given below.

Algorithm: Insert (P, x)

size =size + 1

i = size

while (i > 1) and (x > P[i]) do

 P[i] = P[i/2]

 i = i/2

P[i] = x

10.2 Problems- Heap Sort

1. Write a program for heap sort.
2. Describe heapsort and show that its worst case performance is $O(n \log n)$.
3. Design a heap sort algorithm to sort in non-ascending order.
4. Given an array of n elements sorted in non-ascending order. Is it a heap?
5. Heapify the array {9, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10}. Show the tree structure after each operation.
6. Modify the priority queue algorithms as that the smaller the value of the priority higher the priority.
7. What is the maximum depth of a heap with n elements?
8. Will it be inefficient if we use a singly linked list, instead of using an array to represent the heap? If so, why? (Hint: Consider the insertions that must be done).
9. In terms of space, why is heap sort attractive?
10. In the first phase of heapsort, an initially random vector is rearranged to satisfy the heap structure constraints. Describe how the rearrangement is done, and prove that it can be done in $O(n)$ time, where n is the number of elements in the vector.
11. Describe the time complexity of inserting an element into a complete heap in terms of N, the number of elements in the heap, and in terms of H, the height of the tree.
12. Given the index of an element in a heap, write a function that changes the value of that element and restores the heap property. Assume those heaps functions insert, delete max, heapify up and heapify down are available.

13. How many element comparisons would heap sort use to sort the integers 1 to 8 if they were (i) initially in sorted order, and (ii) initially in reverse sorted order? Explain how you obtained your answers.
14. Would it be possible to implement a variant of heapsort based on a perfectly balanced ternary structure in which the children of node i are at positions $3i - 1$, $3i$, and $3i + 1$, and if so what would be the advantages and disadvantages of the new method?

UNIT XI: SEARCH TREES

11.1 Avl- Tree

INTRODUCTION

As we know that searching in a binary search tree is efficient if the height of the left sub-tree and right sub-tree is same for a node. But frequent insertion and deletion in the tree affects the efficiency and makes a binary search tree inefficient. The efficiency of searching will be ideal if the difference in height of left and right sub-tree with respect of a node is at most one. Such a binary search tree is called balanced binary tree (sometimes called AVL Tree).

REPRESENTATION OF AVL TREE

In order to represent a node of an AVL Tree, we need four fields: - One for data, two for storing address of left and right child and one is required to hold the balance factor. The balance factor is calculated by subtracting the right sub-tree from the height of left sub - tree.

The structure of AVL Tree can be represented by: -

```
Struct AVL
{
    struct AVL *left;
    int data;
    struct AVL *right;
    int balfact;
};
```

DETERMINATION OF BALANCE FACTOR

The value of balance factor may be -1, 0 or 1.

Any value other than these represent that the tree is not an AVL Tree

- If the value of balance factor is -1, it shows that the height of right sub-tree is one more than the height of the left sub-tree with respect to the given node.
- If the value of balance factor is 0, it shows that the height of right sub-tree is equal to the height of the left Sub-tree with respect to the given node.

- If the value of balance factor is 1, it shows that the height of right sub-tree is one less than the height of the left sub-tree with respect to the given node.

INVENTION AND DEFINITION

It was invented in the year 1962 by two Russian mathematicians named G.M. Adelson-Velskii and E.M. Landis and so named AVL Tree.

It is a binary tree in which difference of height of two sub-trees with respect to a node never differ by more than one (1).

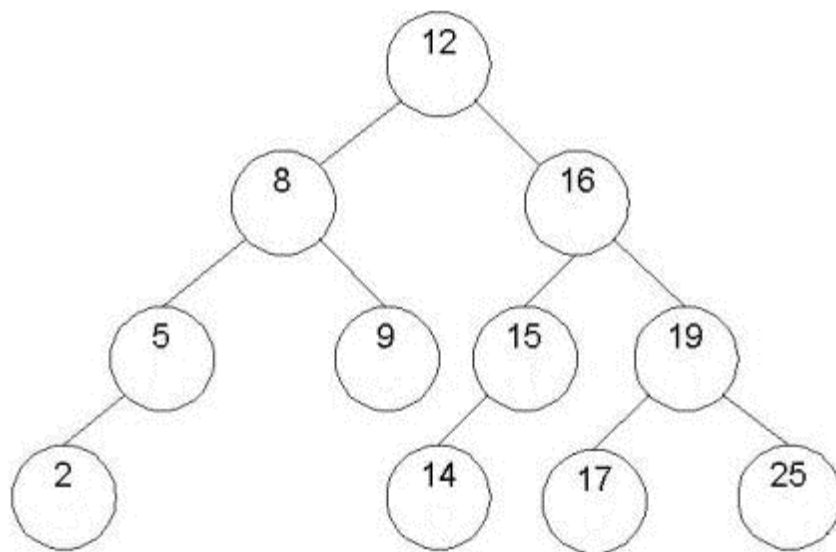


Diagram showing AVL Tree

INSERTION OF A NODE IN AVL TREE

Insertion can be done by finding an appropriate place for the node to be inserted. But this can disturb the balance of the tree if the difference of height of sub-trees with respect to a node exceeds the value one. If the insertion is done as a child of non-leaf node then it will not affect the balance, as the height doesn't increase. But if the insertion is done as a child of leaf node, then it can bring the real disturbance in the balance of the tree.

This depends on whether the node is inserted to the left sub-tree or the right sub-tree, which in turn changes the balance factor. If the node to be inserted is inserted as a node of a sub-tree of smaller height then there will be no effect. If the height of both the left and right sub-tree is same

then insertion to any of them doesn't affect the balance of AVL Tree. But if it is inserted as a node of sub-tree of larger height, then the balance will be disturbed.

To rebalance the tree, the nodes need to be properly adjusted. So, after insertion of a new node the tree is traversed starting from the new node to the node where the balance has been disturbed. The nodes are adjusted in such a way that the balance is regained.

ALGORITHM FOR INSERTION IN AVL TREE

```
int avl_insert(node *treep, value_t target)
{
/* insert the target into the tree, returning 1 on success or 0 if it
* already existed
*/
node tree = *treep;
node *path_top = treep;
while (tree && target != tree->value)
{
direction next_step = (target > tree->value);
if (!Balanced(tree)) path_top = treep;
treep = &tree->next[next_step];
tree = *treep;
}
if (tree) return 0;
tree = malloc(sizeof(*tree));
tree->next[0] = tree->next[1] = NULL;
tree->longer = NEITHER;
tree->value = target;
*treep = tree;
avl_rebalance(path_top, target);
return 1;
}
```

ALGORITHM FOR REBALANCING IN INSERTION

```
void avl_rebalance_path(node path, value_t target)
    {
/* Each node in path is currently balanced. Until we find target, mark each node as longer in the
direction of rget because we know we have inserted target there */
        while (path && target != path->value) {
            direction next_step = (target > path->value);
            path->longer = next_step;
            path = path->next[next_step];
        }
    }

void avl_rebalance(node *path_top, value_t target)
    {
        node path = *path_top;
        direction first, second, third;
        if (Balanced(path)) {
            avl_rebalance_path(path, target);
            return;
        }
        first = (target > path->value);
        if (path->longer != first) {
            /* took the shorter path */
            path->longer = NEITHER;
            avl_rebalance_path(path->next[first], target);
            return;
        }
/* took the longer path, need to rotate */
        second = (target > path->next[first]->value);
        if (first == second) {
/* just a two-point rotate */
```

```

    path = avl_rotate_2(path_top, first);
    avl_rebalance_path(path, target);
    return;
}

```

/* fine details of the 3 point rotate depend on the third step. However there may not be a third step, if the third point of the rotation is the newly inserted point. In that case we record the third step as NEITHER */

```

path = path->next[first]->next[second];
    if (target == path->value) third = NEITHER;
        else third = (target > path->value);
            path = avl_rotate_3(path_top, first, third);
                avl_rebalance_path(path, target);
    }

```

DELETION

A node in AVL Tree is deleted as it is deleted in the binary search tree. The only difference is that we have to do rebalancing which is done similar to that of insertion of a node in AVL Tree. The algorithm for deletion and rebalancing is given below:

ALGORITHM FOR DELETION IN AVL TREE

```

    int avl_delete(node *treep, value_t target)
    {
/* delete the target from the tree, returning 1 on success or 0 if it wasn't found */
        node tree = *treep;
        direction dir;
        node *targetp, targetn;
        while(tree) {
            dir = (target > value);
            if (target == value) targetp = treep;
                if (tree->next[dir] == NULL)

```

```

        break;
        if (tree->longer == NEITHER || (tree->longer == 1-dir && tree-
>next[1-dir]->longer == NEITHER))
            path_top = treep;
            treep = &tree->next[dir];
            tree = *treep;
        }
    if (targetp == NULL) return 0;
    targetp = avl_rebalance_del(path_top, target, targetp);
    avl_swap_del(targetp, treep, dir);
    return 1;
}

```

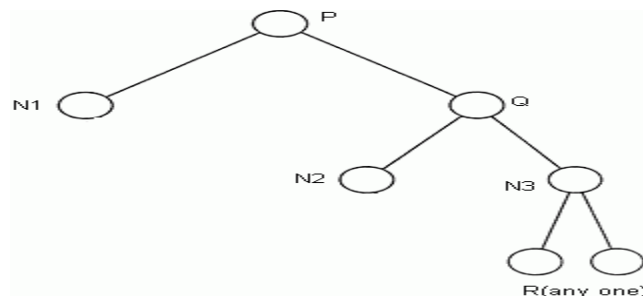
REBALANCING OF AVL TREE

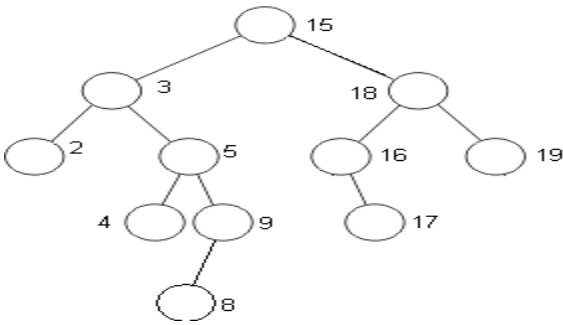
When we insert a node to the taller sub-tree, four cases arise and we have different rebalancing methods to bring it back to a balanced tree form.

- Left Rotation
- Right Rotation
- Right and Left Rotation
- Left and Right Rotation

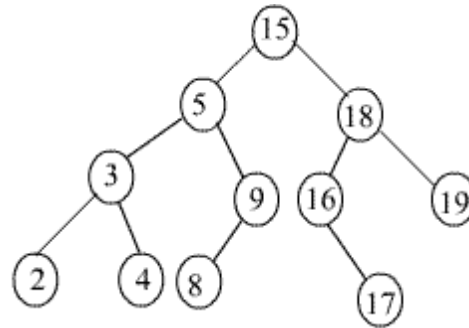
LEFT ROTATION

In general if we want to insert a node R(either as left child or right child) to N3 as shown in figure. Here, as we see the balance factor of node P becomes 2. So to rebalance it, we have a technique called left rotation.





Before Rotation



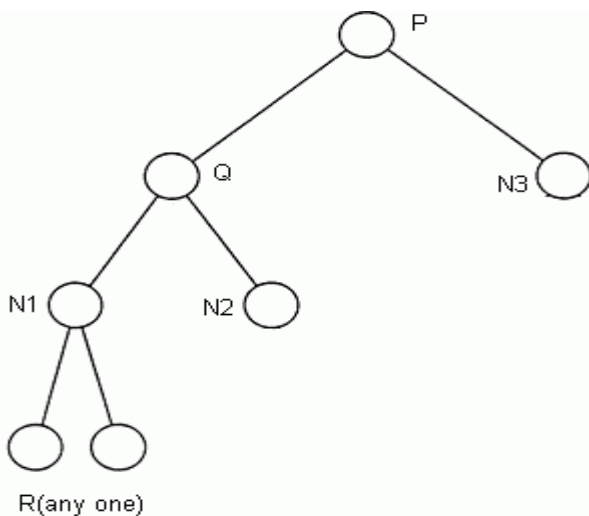
After Rotation

EXPLANATION OF EXAMPLE

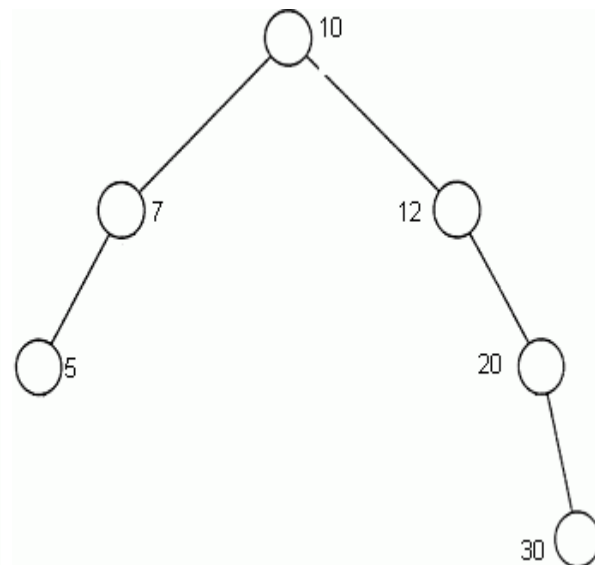
In the given AVL tree when we insert a node 8, it becomes the left child of node 9 and the balance doesn't exist, as the balance factor of node 3 becomes -2. So, we try to rebalance it. In order to do so, we do left rotation at node 3. Now node 5 becomes the left child of the root. Node 9 and node 3 becomes the right and left child of node 5 respectively. Node 2 and node 4 becomes the left and right child of node 3 respectively. Lastly, node 8 becomes the left child of node 9. Hence, the balance is once again attained and we get AVL Tree after the left rotation.

RIGHT ROTATION

In general if we want to insert a node R (either as left or right child) to N1 as shown in figure. Here, as we see the balance factor of node P becomes 2. So to rebalance it, we have a technique called right rotation.



Before Rotation



After Rotation

EXPLANATION OF EXAMPLE

In the given AVL tree when we insert a node 7, it becomes the right child of node 5 and the balance doesn't exist, as the balance factor of node 20 becomes 2. So, we try to rebalance it. In order to do so, we do right rotation at node 20. Now node 10 becomes the root. Node 12 and node 7 becomes the right and left child of root respectively. Node 20 becomes the right child of node 12. Node 30 becomes the right child of node 20. Lastly, node 5 becomes the left child of node 7. Hence, the balance is once again attained and we get AVL Tree after the right rotation

11.2 B- Tree

Tree structures support various basic dynamic set operations including Search, Insert, and Delete in time proportional to the height of the tree. Ideally, a tree will be balanced and the height will be $\log n$ where n is the number of nodes in the tree.

To ensure that the height of the tree is as small as possible and therefore provide the Best running time, a balanced tree structure like AVL tree, 2-3 Tree, Red Black Tree or B-tree must be used.

When working with large sets of data, it is often not possible or desirable to maintain the entire structure in primary storage (RAM). Instead, a relatively small portion of the data structure is maintained in primary storage, and additional data is read from secondary storage as needed. Unfortunately, a magnetic disk, the most common form of secondary storage, is significantly slower than random access memory (RAM). In fact, the system often spends more time retrieving data than actually processing data.

To reduce the time lost in retrieving data from secondary storage, we need to minimize the no. of references to the secondary memory. This is possible if a node in a tree contains more no. of values, then in a single reference to the secondary memory more nodes can be accessed. The AVL trees or red Black Trees can hold a max. of 1 value only in a node while 2-3 Trees can hold a max of 2 values per node. To improve the efficiency Multiway Search Trees are used.

Multiway Search Trees

A Multiway Search Tree of order n is a tree in which any node can have a maximum of $n-1$ values & a max. of n children. B - Trees are a special case of Multiway Search Trees.

A B Tree of order n is a Multiway Search Tree of order n with the following characteristics:

1. All the non leaf nodes have a max of n child nodes & a min of $n/2$ child nodes.
2. If a root is non leaf node, then it has a max of n non empty child nodes & a min of 2 child nodes.
3. If a root node is a leaf node, then it does not have any child node.
4. A node with n child nodes has $n-1$ values arranged in ascending order.
5. All values appearing on the left most child of any node are smaller than the left most value of that node while all values appearing on the right most child of any node are greater than the right most value of that node.
6. If x & y are two adjacent values in a node such that $x < y$, ie they are the i th & $(i+1)$ th values in the node respectively, then all values in the $(i+1)$ th child of that node are $> x$ but $< y$.

The topics to be discussed under B - Trees are as follows:

REPRESENTATION OF B - TREE

The B - Tree is represented as follows using structure:

```
Struct btnode
{
    int count;
    int value[max+1];
    Struct btnode * child[max + 1];
};
```

Count is the no. of children of any node. The values of node are in the array value.

The addresses of child nodes are in child array while the Max is a macro that defines the maximum no. of values any node can have.

OPERATIONS ON B - TREE

THE following operations can be done on a B - Tree :

- Searching
- Insertion
- Deletion

SEARCHING OF A VALUE IN A B-TREE

Searching of a value k in a B-Tree is exactly similar to searching for values in a 2-3 tree. To begin with the value k is compared with the first value key $[0]$ of the root node. If they are similar then the search is complete. If k is less than key $[0]$ then the search is done in the first child node or the sub-tree of the root node.

If k is greater than key $[0]$ then it is compared with key $[1]$. If k is greater than key $[0]$ and smaller than key $[1]$ then k is searched in the second child node or sub-tree of the root node. If k is greater than the last value key $[i]$ of the root node then searching is done in the last child node or sub-tree of the root node. If k is searched in any of the child nodes or sub-tree of the root node then the same procedure of searching is repeated for that particular node or sub-tree.

If the value k is found in the tree then the search is successful. The address of the node in which k is present and the position of the value k in that node is returned. If the value k is not found in the tree, then the search is unsuccessful.

INSERTION OF A VALUE IN A B-TREE

When inserting an item, first do a search for it in the B-tree. If the item is not already in the B-tree, this unsuccessful search will end at a leaf. If there is room in this leaf, just insert the new item here. Note that this may require that some existing keys be moved one to the right to make room for the new item. If instead this leaf node is full so that there is no room to add the new item, then the node must be "split" with about half of the keys going into a new node to the right of this one. The median (middle) key is moved up into the parent node. (Of course, if that node has no room, then it may have to be split as well.)

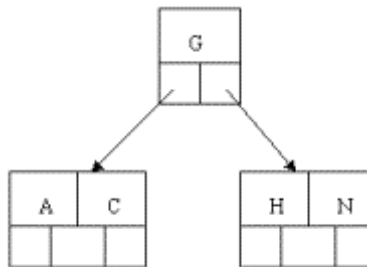
Note that when adding to an internal node, not only might we have to move some keys one position to the right, but the associated pointers have to be moved right as well. If the root node

is ever split, the median key moves up into a new root node, thus causing the tree to increase in height by one.

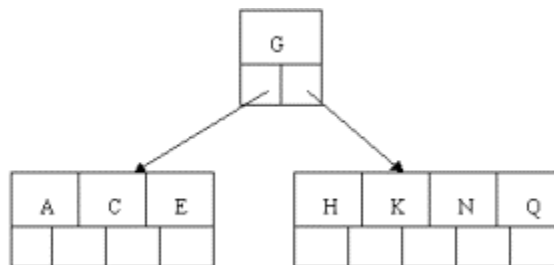
Let's take an example. Insert the following letters into what is originally an empty B-tree of order 5: C N G A H E K Q M F W L T Z D P R X Y S Order 5 means that a node can have a maximum of 5 children and 4 keys. All nodes other than the root must have a minimum of 2 keys. The first 4 letters get inserted into the same node, resulting in this picture:



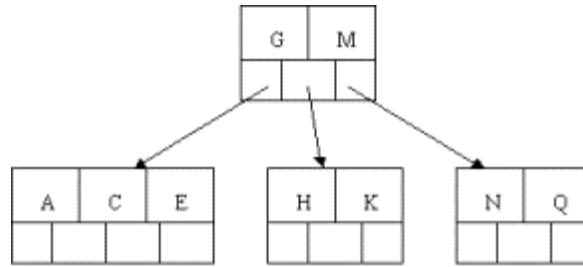
When we try to insert the H, we find no room in this node, so we split it into 2 nodes, moving the median item G up into a new root node. Note that in practice we just leave the A and C in the current node and place the H and N into a new node to the right of the old one.



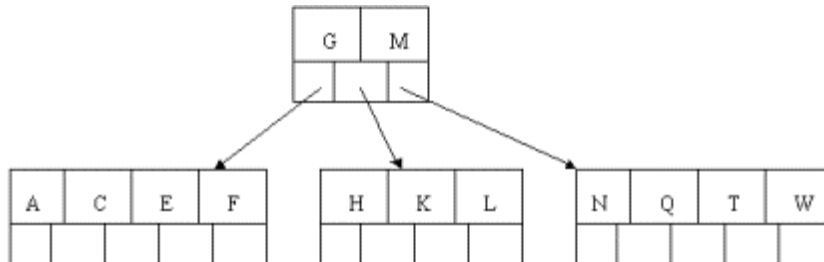
Inserting E, K, and Q proceeds without requiring any splits:



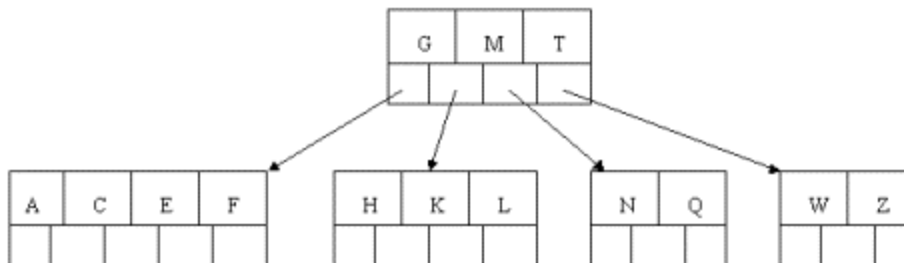
Inserting M requires a split. Note that M happens to be the median key and so is moved up into the parent node.



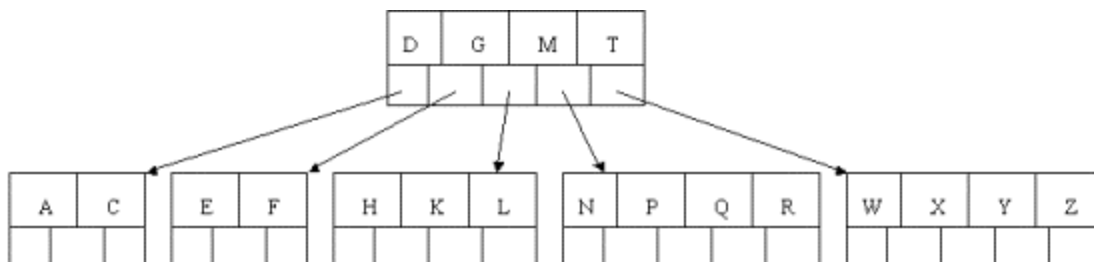
The letters F, W, L, and T are then added without needing any split.



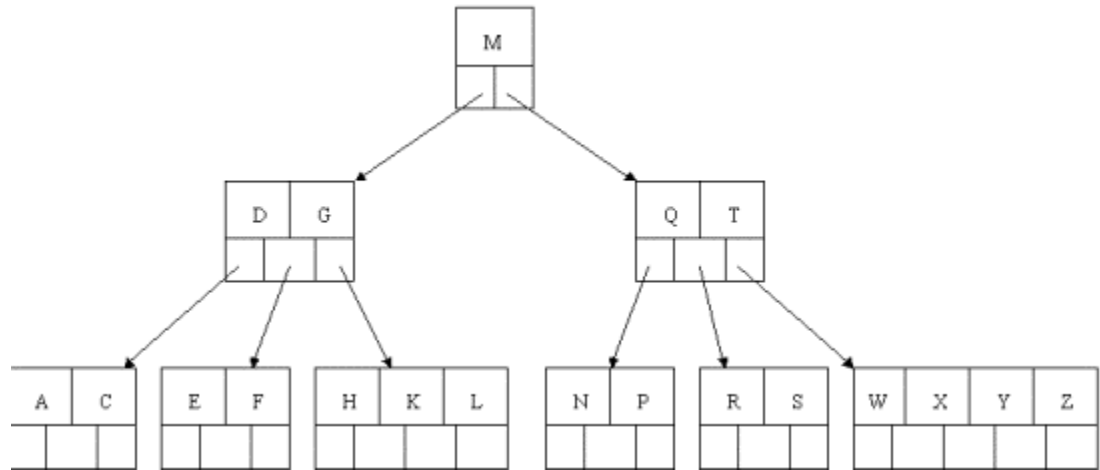
When Z is added, the rightmost leaf must be split. The median item T is moved up into the parent node. Note that by moving up the median key, the tree is kept fairly balanced, with 2 keys in each of the resulting nodes.



The insertion of D causes the leftmost leaf to be split. D happens to be the median key and so is the one moved up into the parent node. The letters P, R, X, and Y are then added without any need of splitting:



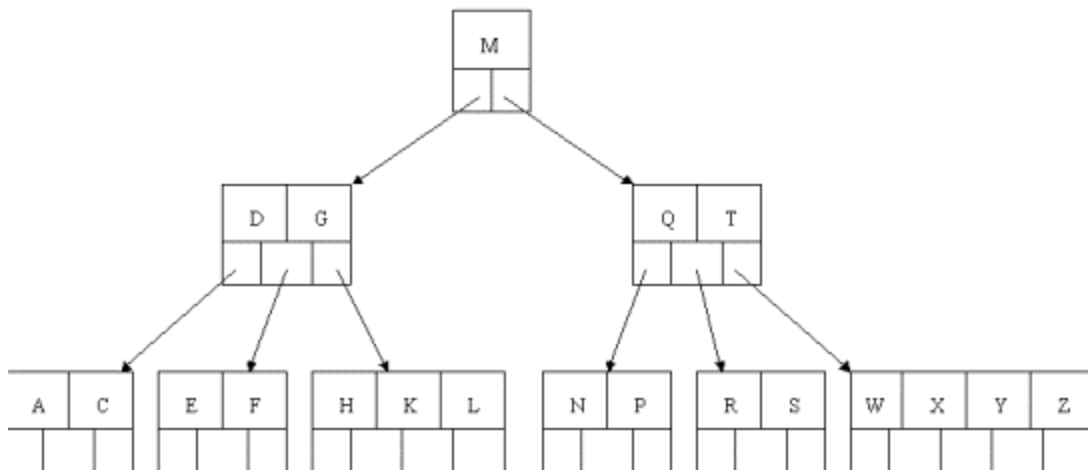
Finally, when S is added, the node with N, P, Q, and R splits, sending the median Q up to the parent. However, the parent node is full, so it splits, sending the median M up to form a new root node. Note how the 3 pointers from the old parent node stay in the revised node that contains D and G.



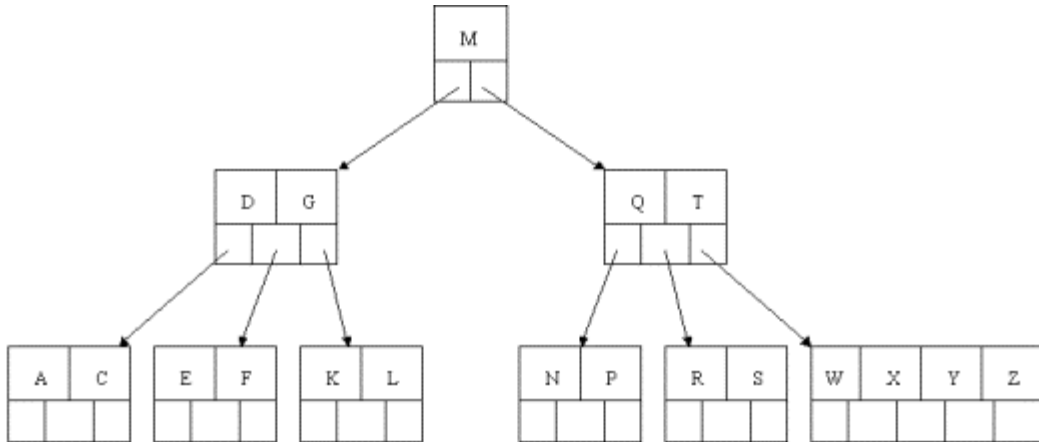
DELETION IN A B – TREE

Deletion in a B -Tree is similar to insertion. At first the node from which a value is to be deleted is searched. If found out, then the value is deleted. After deletion the tree is checked if it still follows B - Tree properties.

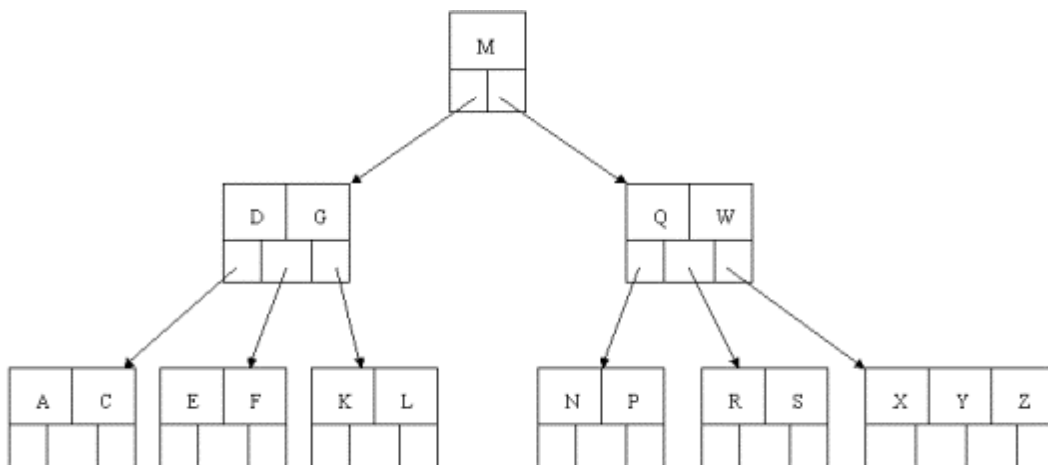
Let us take an example. The original B - Tree taken is as follows:



Delete H. first it is found out. Since H is in a leaf and the leaf has more than the minimum number of keys, this is easy. We move the K over where the H had been and the L over where the K had been. This gives:

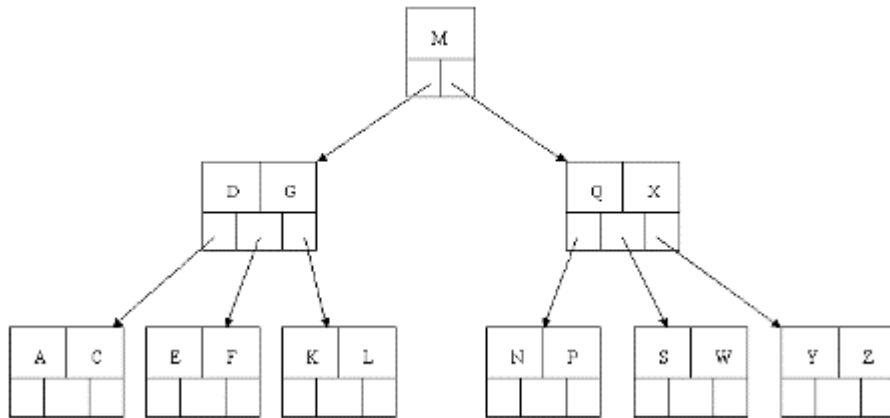


Next, delete the T. Since T is not in a leaf, we find its successor (the next item in ascending order), which happens to be W, and move W up to replace the T. That way, what we really have to do is to delete W from the leaf, which we already know how to do, since this leaf has extra keys. In ALL cases we reduce deletion to a deletion in a leaf, by using this method

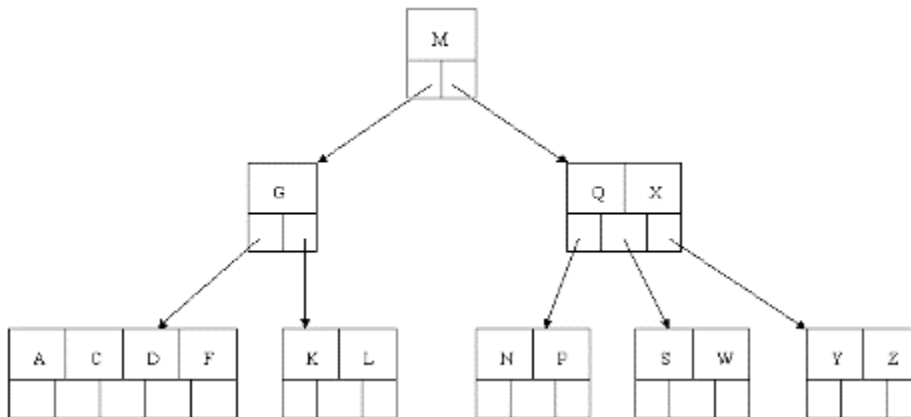


Next, delete R. Although R is in a leaf, this leaf does not have an extra key; the deletion results in a node with only one key, which is not acceptable for a B-tree of order 5. If the sibling node to the immediate left or right has an extra key, we can then borrow a key from the parent and move a key up from this sibling. In our specific case, the sibling to the right has an extra key. So, the successor W of S (the last key in the node where the deletion occurred), is moved down from the

parent, and the X is moved up. (Of course, the S is moved over so that the W can be inserted in its proper place.)

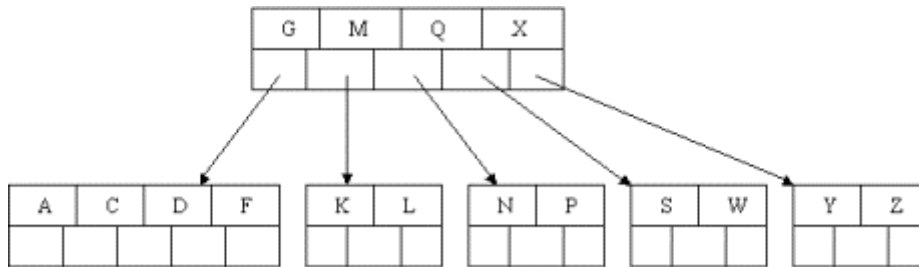


Finally, let's delete E. This one causes lots of problems. Although E is in a leaf, the leaf has no extra keys, nor do the siblings to the immediate right or left. In such a case the leaf has to be combined with one of these two siblings. This includes moving down the parent's key that was between those of these two leaves. In our example, let's combine the leaf containing F with the leaf containing A C. We also move down the D.



Of course, you immediately see that the parent node now contains only one key, G. This is not acceptable. If this problem node had a sibling to its immediate left or right that had a spare key, then we would again "borrow" a key. Suppose for the moment that the right sibling (the node with Q X) had one more key in it somewhere to the right of Q. We would then move M down to the node with too few keys and move the Q up where the M had been. However, the old left subtree of Q would then have to become the right subtree of M.

In other words, the N P node would be attached via the pointer field to the right of M's new location. Since in our example we have no way to borrow a key from a sibling, we must again combine with the sibling, and move down the M from the parent. In this case, the tree shrinks in height by one.



11.3 Problems- Search Trees

- What is AVL Tree?
- Who invented AVL Tree?
- How can we determine the balance factor?
- Insert a node 3 in the AVL Tree of the tutorial and try to rebalance the tree by any of the methods?
- How an AVL Tree or B - Tree can be better than a Binary Search Tree?
- How an AVL Tree or B - Tree can be better than a Binary Search Tree?
- What advantages does a multiway search Tree have over an AVL Tree?
- What are 5the differences between a multiway search tree & a B - Tree?

UNIT XII: TABLES

12.1 Hashing Techniques

INTRODUCTION

Hashing is a method to store data in an array so that sorting, searching, inserting and deleting data is fast. For this every record needs unique key.

The basic idea is not to search for the correct position of a record with comparisons but to compute the position within the array. The function that returns the position is called the 'hash function' and the array is called a 'hash table'.

WHY HASHING?

In the other type of searching, we have seen that the record is stored in a table and it is necessary to pass through some number of keys before finding the desired one. While we know that the efficient search technique is one which minimizes these comparisons. Thus we need a search technique in which there are no unnecessary comparisons.

If we want to access a key in a single retrieval, then the location of the record within the table must depend only on the key, not on the location of other keys(as in other type of searching i.e. tree). The most efficient way to organize such a table is an array.It was possible only with hashing.

HASH CLASH

Suppose two keys k_1 and k_2 are such that $h(k_1)$ equals $h(k_2)$.When a record with key two keys can't get the same position. such a situation is called hash collision or hash clash.

METHODS OF DEALING WITH HASH CLASH

There are three basic methods of dealing with hash clash. They are:

- Chaining
- Rehashing
- Separate chaining.

CHAINING

It builds a link list of all items whose key has the same value. During search, this sorted linked list is traversed sequentially for the desired key. It involves adding an extra link field to each table position. There are three types of chaining

1. Standard Coalsced Hashing
2. General Coalsced Hashing
3. Varied insertion coalsced Hashing

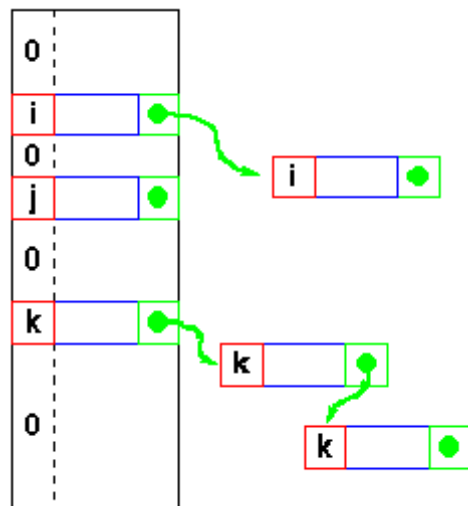
Standard Coalsced Hashing

It is the simplest of chaining methods. It reduces the average number of probes for an unsuccessful search. It efficiently does the deletion without affecting the efficiency

General Coalsced Hashing

It is the generalization of standard coalesced chaining method. In this method, we add extra positions to the hash table that can be used to list the nodes in the time of collision.

Varied insertion coalsced Hashing



It is the combination of standard and general coalesced hashing. Under this method, the colliding item is inserted to the list immediately following the hash position unless the list forming from that position containing a cellular element.

SOLVING HASH CLASHES BY LINEAR PROBING

The simplest method is that when a clash occurs; insert the record in the next available place in the table. For example in the table the next position 646 is empty. So we can insert the record with key 012345645 in this place which is still empty. Similarly if the record with key %1000 = 646 appears, it will be inserted in next empty space. This technique is called linear probing and is an example for resolving hash clashes called rehashing or open addressing.

WORKING OF LINEAR PROBING ALGORITHM

It works like this: If array location $h(\text{key})$ is already occupied by a record with a different key, rh is applied to the value of $h(\text{key})$ to find the other location where the record may be placed. If position $rh(h(\text{key}))$ is also occupied, it too is rehashed to see if $rh(rh(h(\text{key})))$ is available. This process continues until an empty location is found. Thus we can write a search and insert algorithm using hashing as follows:

ALGORITHM

```
void insert( key, r )
typekey key; dataarray r;
{
    extern int n;
    int i, last;
    i = hashfunction( key );
    last = (i+m-1) % m;
    while ( i!=last && !empty(r[i]) && !deleted(r[i]) && r[i].k!=key )
        i = (i+1) % m;
    if (empty(r[i]) || deleted(r[i]))
    {
        /*** insert here ***/
        r[i].k = key;
        n++;
    }
    else Error /*** table full, or key already in table ***/;}
}
```

DISADVANTAGES OF LINEAR PROBING

It may happen, however, that the loop executes forever. There are two possible reasons for this. First, the table may be full so that it is impossible to insert any new record. This situation can be detected by keeping an account of the number of records in the table.

When the count equals the table size, no further insertion should be done. The other reason may be that the table is not full, too. In this type, suppose all the odd positions are empty and the even positions are full and we want to insert in the even position by $rh(i)=(i+2)\%1000$ used as a hash function. Of course, it is very unlikely that all the odd positions are empty and all the even positions are full.

However the rehash function $rh(i)=(i+200)\%1000$ is used, each key can be placed in one of the five positions only. Here the loop can run infinitely, too.

SEPARATE CHAINING

As we have seen earlier, we can't insert items more than the table size. In some cases, we allocate space much more than required resulting in wastage of space. In order to tackle all these problems, we have a separate method of resolving clashes called separate chaining. It keeps a distinct link list for all records whose keys hash into a particular value. In this method, the items that end with a particular number (unit position) is placed in a particular link list as shown in the figure. The 10's, 100's not taken into account. The pointer to the node points to the next node and when there is no more nodes, the pointer points to NULL value.

ADVANTAGES OF SEPERATE CHAINING

- No worries of filling up the table whatever be the number of items.
- The list items need not be contiguous storage
- It allows traversal of items in hash key order.

SITUATION OF HASH CLASH

What would happen if we want to insert a new part number 012345645 in the table .Using he hash function $key \%1000$ we get 645. Therefore for the part belongs in position 645. However

record for the part is already being occupied by 011345645. Therefore the record with the key 012345645 must be inserted somewhere in the table resulting in hash clash. This is illustrated in the given table:

POSITION	KEY	RECORD
0	258001201	
2	698321903	
3	986453204	
.		
.		
450	256894450	
451	158965451	
.		
.		
647	214563647	
648	782154648	
649	325649649	
.		
.		
997	011239997	
998	231452998	
999	011232999	

DOUBLE HASHING

A method of open addressing for a hash table in which a collision is resolved by searching the table for an empty place at intervals given by a different hash function, thus minimizing clustering. Double Hashing is another method of collision resolution, but unlike the linear collision resolution, double hashing uses a second hashing function that normally limits multiple collisions. The idea is that if two values hash to the same spot in the table, a constant can be calculated from the initial value using the second hashing function that can then be used to change the sequence of locations in the table, but still have access to the entire table.

Algorithm for double hashing

```
void insert( key, r )
```

```
typekey key; dataarray r;
```

```
{
```

```

extern int n;
int i, inc, last;
i = hashfunction( key );
inc = increment( key );
last = (i+(m-1)*inc) % m;
while ( i!=last && !empty(r[i]) && !deleted(r[i]) && r[i].k!=key )
i = (i+inc) % m;
if ( empty(r[i]) || deleted(r[i]) )
{
/**/ insert here /***/
r[i].k = key;
n++;
}
else Error /**/ table full, or key already in table /***/;
}

```

CLUSTERING

There are mainly two types of clustering:

Primary clustering

When the entire array is empty, it is equally likely that a record is inserted at any position in the array. However, once entries have been inserted and several hash clashes have been resolved, it doesn't remain true. For, example in the given above table, it is five times as likely for the record to be inserted at the position 994 as the position 401. This is because any record whose key hashes into 990, 991, 992, 993 or 994 will be placed in 994, whereas only a record whose key hashes into 401 will be placed there. This phenomenon where two keys that hash into different values compete with each other in successive rehashes is called primary clustering.

CAUSE OF PRIMARY CLUSTERING

Any rehash function that depends solely on the index to be rehashed causes primary clustering

WAYS OF ELIMINATING PRIMARY CLUSTERING

One way of eliminating primary clustering is to allow the rehash function to depend on the number of times that the function is applied to a particular hash value. Another way is to use random permutation of the number between 1 and e , where e is (table size - 1, the largest index of the table). One more method is to allow rehash to depend on the hash value. All these methods allow key that hash into different locations to follow separate rehash paths.

SECONDARY CLUSTERING

In this type, different keys that hash to the same value follow same rehash path.

WAYS TO ELIMINATE SECONDARY CLUSTERING

All types of clustering can be eliminated by double hashing, which involves the use of two hash function $h_1(\text{key})$ and $h_2(\text{key})$. h_1 is known as primary hash function and is allowed first to get the position where the key will be inserted. If that position is occupied already, the rehash function $rh(i, \text{key}) = (i + h_2(\text{key})) \% \text{table size}$ is used successively until an empty position is found. As long as $h_2(\text{key}_1)$ doesn't equal $h_2(\text{key}_2)$, records with keys h_1 and h_2 don't compete for the same position. Therefore one should choose functions h_1 and h_2 that distributes the hashes and rehashes uniformly in the table and also minimizes clustering.

DELETING AN ITEM FROM THE HASH TABLE:

It is very difficult to delete an item from the hash table that uses rehashes for search and insertion. Suppose that a record r is placed at some specific location. We want to insert some other record r_1 on the same location. We will have to insert the record in the next empty location to the specified original location. Suppose that the record r which was there at the specified location is deleted.

Now, we want to search the record r_1 , as the location with record r is now empty, it will erroneously conclude that the record r_1 is absent from the table. One possible solution to this problem is that the deleted record must be marked "deleted" rather than "empty" and the search must continue whenever a "deleted" position is encountered. But this is possible only when there are small numbers of deletions otherwise an unsuccessful search will have to search the entire table since most of the positions will be marked "deleted" rather than "empty".

DYNAMIC AND EXTENDIBLE HASHING

One of the serious drawbacks associated with hashing of external storage is its being insufficiently flexible. The contents of the external storage structure tend to grow and shrink unpredictably. The entire hash table structuring method that we have examined has a sharp space/time trade-off. Either the table uses a large amount of space for efficient access which results in wastage of large space or it uses a small amount of space and accommodates growth very poorly and sharply increasing the access time for overflow elements. So in order to tackle the above stated problems, we would like to develop a scheme that doesn't utilize too much extra space when a file is small but permits efficient access when it grows larger. Two such schemes are dynamic hashing and Extendible hashing.

DYNAMIC HASHING

Dynamic hashing is a hash table that grows to handle more items. The associated hash function must change as the table grows. Some schemes may shrink the table to save space when items are deleted.

EXTENDIBLE HASHING

A hash table in which the hash function is the last few bits of the key and the table refer to buckets. Table entries with the same final bits may use the same bucket. If a bucket overflows, it splits, and if only one entry referred to it, the table doubles in size. If a bucket is emptied by deletion, entries using it are changed to refer to an adjoining bucket, and the table may be halved.

HASH TABLE REORDERING

When a hash table is nearly full, many items given by their hash keys are not at their specified location. Thus, we have to make a lot of key comparisons before finding such items. If an item is not in the table, entire hash table has to be searched. Then, we come to the conclusion that the key is not in the table. In order to tackle this situation, many techniques came forward.

AMBLE AND KNUTH METHOD

In this method, all the records that hash into same locations are placed in descending order (assuming that the NULLKEY is the smallest one). Suppose we want to search a key, we need

not rehash repeatedly until an empty slot is found. As soon as an item whose key is less than the search key is found in the table, we come to the conclusion that the search key is not in the table. At the time of insertion, if we want to insert a key k , if the rehash accesses a key smaller than k , the associated record with k replaces it and the insertion process continues with the replaced key.

Note

The ordered hash table method can be used only in the technique in which a rehash depends only on the index and the key not in the technique in which a rehash function depends on the no of items the item is rehashed

(Unless that number is kept in the table).

ADVANTAGES OF AMBLE AND KNUTH'S METHOD

It reduces significantly the number of key comparisons necessary to determine that a key doesn't exist in the table.

DISADVANTAGES OF AMBLE AND KNUTH'S METHOD

- It doesn't change the average number of key comparisons required to find a key that is in the table
- The unsuccessful search needs same average number of probes as the successful search.
- Average number of probes in insertion is not reduced in ordered table.

BRENT'S METHOD

This technique involves rehashing the search argument until an empty slot is found. Then each of the keys in rehash path is itself rehashed to determine if placing one of those keys in an empty slot would require fewer rehashes. If this is the case the search argument replaces the existing key in the table and the existing key is inserted in its empty rehash slot.

BINARY TREE HASHING

Another method of reordering the hash table was developed by Gonnet and Munro and is called as binary tree hashing. It is seen as an improvement to Brent's algorithm. In this method, we assume to use double hashing. Whenever a key is inserted in the hash table, an almost complete binary tree is constructed. Figure below illustrates an example of such a tree in which the nodes are arranged according to the array representation of an almost complete binary tree. node(0) is the root and node($2*i+1$) and node($2*i+2$) are the left and right children of node(i) respectively. Each node of the tree contains an index into the hash table. In the explanation, node(i) will be referred to as index(i) and the key at that position is referred to as k(-1).

HOW TO CONSTRUCT A TREE?

Firstly, we define the youngest right ancestor of node(i) or yra(i) as the node number of the father of the youngest son i.e. the right son. In the given figure, yra(12) is 1, since it is the left son of its father node(6) and its father is also a left child. So the youngest ancestor of node(12) is node(3) and its father is node(1). Similarly, yra(10) is 2 and yra(18) is 4 and yra(14) is 3. if node(i) is the right son, yra(i) is defined as the node number of its father $(i-1)/2$. Thus, yra(15) is 7 and yra(13) is 6. If node(i) has no ancestor i.e. is a right son, yra(i) is defined as (-1). Thus, yra(16) is -1. The binary tree is constructed according to the node number. The table construction continues until a NULLKEY and an empty position is found in the table.

HOW TO CALCULATE yra(i)

As we have seen above, the entire algorithm depends on the routine yra(i). Fortunately, yra(i) can be calculated very easily. It can be derived directly using this method. Find the binary representation of (i+1). Delete all the trailing zero bits along with one bit preceding them. Subtract 1 from the result and you will get the resulting binary number to get the value of yra(i).

EXAMPLES

1. yra(11):
 $11+1=12$

Binary representation: 1100.

Removing 100, we get 1, which is binary representation of 1.

Therefore, $yra(11)=0$.

2. $yra(17)$:

$17+1=18$

Binary representation: 10010.

Removing 10, we get 100, which is binary representation of 4.

Therefore, $yra(11)=3$.

3. $yra(15)$:

$15+1=16$

Binary representation: 010000.

Removing 10000, we get 0, which is binary representation of 0.

Therefore, $yra(11)=-1$.

HOW TO INSERT A KEY IN THE TABLE

Once the tree has been constructed, the keys along the path from the root to the last node are reordered in the hash table. Let, i be initialized to the last node of the tree. If $yra(i)$ is non-zero, $k(yra(i))$ and its associated record are shifted from the $table[index(yra(i))]$ to $table[index(i)]$ and I is reset to $yra(i)$. It is repeated until $yra(i)$ is -1 at which point insertion is complete.

EXAMPLE OF INSERTION

Suppose $yra(21)=10$ and $index(10)$ is j , the key and record from j is shifted to u which is the right child. Then suppose $yra(10)$ is 2, the record and key from position b is shifted to j which is the right child of $index b$. Finally since $yra(2)$ is -1, key is inserted in position b .

ADVANTAGE OF BINARY SEARCH TREE

Binary tree hashing yields results that are even closer to optimal than Brent's

12.2 Problems-Tables

- What is Direct Addressing? When is it used?
- What is the advantage of using Hash Function over Direct Addressing? Explain.
- When does a 'collision' occur? What are the methods to resolve it? Explain giving examples.
- Explain the 'division method' for creating hash functions.
- Consider a version of the division method in which $h(k) = k \bmod m$, where $m = (2^p) - 1$ and k is a character string interpreted in radix 2^p .
- Show that if string x can be derived from string y by permuting its characters, then x and y hash to the same value.
- What is 'universal hashing' and how will you design a universal class of hash functions?
- Differentiate between linear probing and quadratic probing.
- Insert the keys 10,22,31,4,15,28,17,88,59 into a hash table of length $m=11$ using open addressing with the primary hash function $h'(k) = k \bmod m$.
- Illustrate the result of inserting keys using linear probing, using quadratic probing with $c_1=1$ and $c_2=3$ and using double hashing with $h_2(k) = 1 + (k \bmod (m-1))$.
- Canadian postal codes have the format LDL DLD, where L is always a letter (between A-Z), D is always a digit (0-9), and $_$ is always a single space. For example, the postal code for the University of Waterloo is N2L 3G1. Devise a suitable hash function for this system.

UNIT XIII: SETS

13.1 Introduction

Definition:

- A set is a collection of distinct objects.
Example $S = \{ 2, 5, c, \text{ball}, \text{red}, 34.5 \}$
- In the above definition there is no restriction on objects. But, in many applications we deal with a set of objects of same type. For example, integers, characters, or strings.
- In this module, we discuss representation of sets whose element is integers in the range 1 between n .

Bit Vector Representation

A simple representation of a set is using an n -bit vector.

If i is a member of a set S then i th bit is 1 (or true), otherwise 0 (or false).

Example : $A = \{ 5, 3, 7, 9, 1 \}$

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

Fig: Representation of the set

- Inserting an integer x into a set can be done by changing the x th bit to 1.
- Similarly, deleting x is nothing but changing the x th bit to 0.
- Member checking, that is x is a member or not boils down to checking the x th bit. If x th bit is 1 then x is a member, otherwise not.
- That is, insert, delete, and member operation can be performed in constant time.
- Other operation like union, intersection, difference can be performed using logical bit operation.

- Assume, sets A, B, and C are represented using an arrays a, b, and c respectively of size n.
- Pseudocode for
Algorithm Union (a, b, c)
for i = 1 to n do
c[i] = a[i] OR b[i]
- Pseudocode for
Algorithm Intersection (a, b, c)
for i = 1 to n do
c[i] = a[i] AND b[i]
- Pseudocode for C = A - B
Algorithm Difference (a, b, c)
for i = 1 to n do
c[i] = a[i] AND (NOT b[i])
- These operations take time proportional to the n.
- If n less than the size of a word in computer, the bit vector can be stored in a word. The operation Union, intersection, and difference can be performed using one word logical operation.

Linked list representation

- More general representation of sets is using linked lists. Each item in the list is a member of the set.
- The elements in the set need not be from any universal set.
- The size of the list proportional to the size of the set, but not proportional to the universal set.
- Assume, there exists a linear ordered relation $<$, amount the elements of the universal set.
- That is, for all x_i and x_j , exactly one of $x_i < x_j$, $x_i = x_j$, or $x_j < x_i$ is true.
- The relation $<$ is transitive, that is, for all x_i , x_j and x_k , if $x_i < x_j$ and $x_j < x_k$, then $x_i < x_k$.
- We store element in a set by the above linear ordered relation. That is, if $x_i < x_j$ then x_i is placed before x_j in the linked list.

A = { 5, 3, 7, 9, 1 }

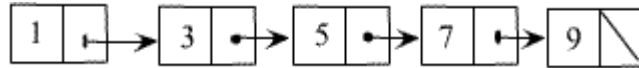


Fig 2: Set Representation using linked list

- So, operations inset, delete, and member are reduced to insert, delete, and search in the linked list of sorted elements. See module 3.
- Similarly, operations union, intersection, and difference also can reduce to linked list operations. This is left as an exercise, see problem.
- Another variation of linked lists representation of sets is using a pointer from each element to the set representing element.
- The structure of each object is shown below.

Struct Object

```
{
    Int data;
    Struct object      *nextobject;
    Struct object      *representative;
};
```

- The above structure describes that each object in the linked list contains a set member, a pointer to the object containing the next set member, and a pointer back to the representative.
- The first object in each linked list serves as its sets representative.

• Examples

Set1 = { a, b, c, d }

Set2 = { 1, 2, 3, 4 }

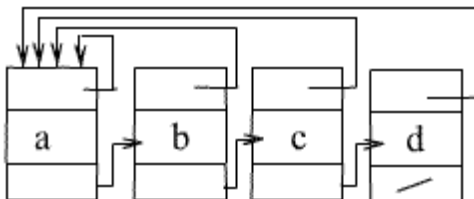


Fig 3: Set1

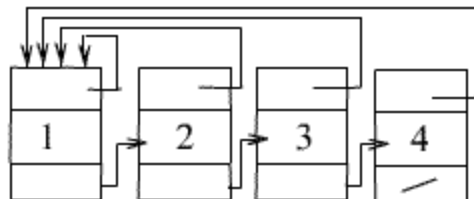


Fig 4: Set 2

- In Fig 3, 'a' is the linked list representative for Set1 and in Fig4, '1' is the linked list representative for Set2.
- The function MAKE-SET(x) creates a new set whose only member is pointed by 'x'.

Object* MAKE-SET(x)

```
{
  Struct object *temp;
  temp= (struct object*)malloc(sizeof(struct object));
  temp->data = x;
  temp->representative=temp;
  return temp;
}
```

The function MAKE-SET(x) takes $O(1)$ time

- The UNION operation using linked list representation takes significantly more time. We perform UNION (X,Y) by appending Y's list onto the end of X's list. We must update the pointer to the representative for each object originally on Y's List to X.

Void UNION (X,Y)

```
{
  For each object a?Y
  For each object b?X
    If( b->data == a->data)
    {
      Add 'a' to X;
      a->representative = X;
    }
}
```

The following example shows the union of two sets for the previously mentioned Set1 and Set2. That is we are performing the operation UNION (a,1).

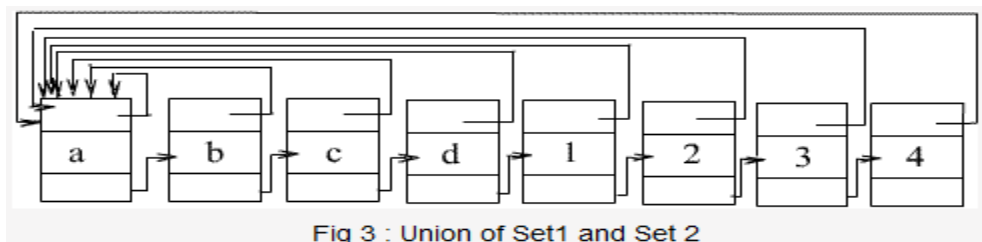


Fig 3 : Union of Set1 and Set 2

- The UNION operation takes $O(mn)$ time. Here ' m' and 'n' are the sizes of the given two lists.

13.2 Problems-Sets

- Give bit-vector and linked list

UNIT XIV: STRING ALGORITHM

14.1 Introduction

- A string is a sequence of characters. In computer science, strings are more often used than numbers. We have all used text editors for editing programs and documents. Some of the Important Operations which are used on strings are: searching for a word, find - and -replace operations, etc.
- There are many functions which can be defined on strings. Some important functions are
 - a. String length: Determines length of a given string.
 - b. String concatenation: Concatenation of two or more strings coping.
 - c. String copy: Creating another string which is a copy of the original or a copy of a part of the original.
 - d. String matching: Searching for a query string in given string.

STRING LENGTH

- Strings can have an arbitrary but finite length.
- There are two types of string data types:
 - a. Fixed length strings
 - b. Variable length strings
- Fixed length strings have a maximum length and all the strings uses same amount of space despite of their actual size.
- Variable length strings uses varying amount of memory depending on their actual size. Throughout of our discussion we assume that strings are of variable length type.
- Variable length string is an array of characters terminated by a special character.
- To find the length of a string we scan through the string from left to right until we find the special symbol and each time incrementing a counter to keep track of number of characters scanned so far.

ALGORITHM FOR StringLength

We assume that the given string STR is terminated by special symbol '\0'.

```

length = 0, i=0; //Identity starts from '0'.
while STR[i] != '\0' //In C '\0' is used as end-of-string markes.

    i++;
    length=i;
return length

```

STRING CONCATENATION

- Appending one string to the end of another string is called string concatenation
Example let STR1= "hello"
STR2= "world"
- If we concatenate STR2 with STR1, then we get the string "helloworld"

Algorithm

1. i= 0, j=0;
2. while STR1[i] != '\0'
 - i++;
3. while STR2[j] != '\0'
 - STR1[i]= STR2[j];
 - i=i+1
 - j = j+1
4. STR1[i]= '\0';
5. Return STR1;

14.2 String Copy

- By string copy, we mean copying one string to another string character by character.
- The size of the destination string should be greater than equal to the size of the source string.

Algorithm

1. Set i=0

2. while STR[i] != '\0'
3. {
 - STR2[i]=STR1[i];
 - i =i+1;
- }
4. Set STR2[i]='\0'
5. Return STR2

14.3 Pattern Matching

STRING-MATCHING

- String matching is a most important problem.
- String matching consists of searching a query string (or pattern) P in a given text T.
- Generally the size of the pattern to be searched is smaller than the given text.
- There may be more than one occurrences of the pattern P in the text T. Sometimes we have to find all the occurrences of the pattern in the text.
- There are several applications of the string matching. Some of these are
 1. Text editors
 2. Search engines
 3. Biological applications
- Since string-matching algorithms are used extensively, these should be efficient in terms of time and space.
- Let P [1..m] is the pattern to be searched and its size is m.
- T [1..n] is the given text whose size is n
- Assume that the pattern occurs in T at position (or shift) i. Then the output of the matching algorithm will be the integer i where $1 \leq i \leq n-m$. If there are multiple occurrences of the pattern in the text, then sometimes it is required to output all the shifts where the pattern occurs.

Let Pattern $P = CAT$

Text = ABABNACATMAN

Then there is a match with the shift 7 in the text T

1	2	3	4	5	6	7	8	9	10	11	12
A	B	A	B	N	A	C	A	T	M	A	N
						C	A	T			

Brute Force String Matching algorithm.

- This algorithm is a simple and obvious one, in which we compare a given pattern P with each of the sub strings of the text T , moving from left to right, until a match is found
- Let S_i is the substring of T , beginning at the i th position and whose length is same as pattern P .
- We compare P , character by character, with the first substring S_1 . If all the corresponding characters are same, then the pattern P appears in T at shift 1. If some of the characters of S_1 are not matched with the corresponding characters of P , then we try for the next substring S_2 . This procedure continues till the input text exhausts.
- In this algorithm we have to compare P with $n-m+1$ substrings of T .

Example

- Let $P = abc$
 $T = aabab$
- Compare P with 1st substring of T

a b a

a a b a b

Mismatch at the second character of T

Fig: 2(a)

- Compare P with 2nd substring of T

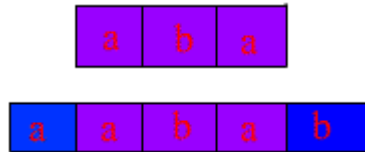


Fig: 2(b)

Since the corresponding characters are same, there is a match at shift 1.

- Compare P with 3rd substring of T

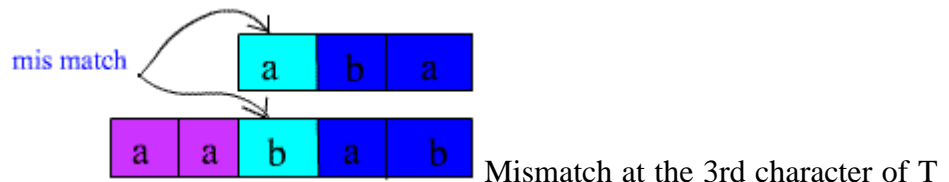


Fig: 2(c)

Algorithm For Brute-Force String matching

Let $P[0..m]$ is the given pattern and $T[0..n]$ is the text.

1. $i = 1$; [substring 1]
2. Repeat steps 3 to 5 while $i \leq n - m + 1$ do
3. for $j = 1$ to m [For each character of P]

If $P[j] \neq T[i+j-1]$ then

goto step 5

4. Print "Pattern found at shift i "
5. $i = i + 1$
6. exit

- The complexity of the brute force string matching algorithm is $O(nm)$
- On average the inner loop runs fewer than m times to know that there is a mismatch.

- The worst case situation arises when first m character are matched for all substrings S_i . If pattern is of the form $a^{m-1}b$ and text is of the form $a^{n-1}b$, where a^{n-1} denotes a repeated $n-1$ times. In this case the inner loop runs exactly for m times before knowing that there is a mismatch. In this situation there will be exactly $m \cdot (n-m+1)$ number of comparisons.

Knuth-Morris-Pratt(KMP) string matching algorithm

- In brute force method, irrespective of the structure of the pattern P , if there is a mismatch, we restart the matching process with shift $s = s+1$.
- But if we know the structure of the pattern, we can intelligently avoid some number of shifts.

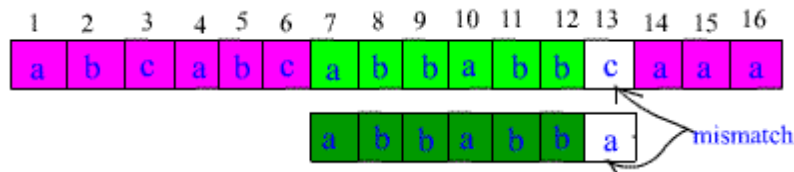


Fig: 3

- Suppose 6 characters are matched successfully and there is a mismatch at the 7th position. The shift $s = s+1$ is obviously invalid as the first pattern character 'a' would be aligned to the text character 'b', which is known to match the second pattern character.
- Also if we observe, the shift $s = s+2$ is also invalid.
- But the shift $s = s+3$ may be valid as first three characters of the pattern will be nicely aligned to the last three characters of the portion of the text which are already matched.

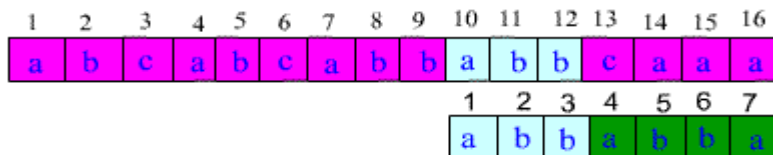


Fig: 4

- KMP (Knuth-Morris-pratt) algorithm avoids some redundant comparisons. When a mismatch occurs, the most we can shift the pattern so as to avoid redundant comparisons is the largest prefix of $P[1..j]$ that is also suffix of $P[2..j]$, where j is the number of characters that are known to be matched successfully before the mismatch.
- The amount of shift that is not necessarily invalid can be pre-computed by comparing the pattern against itself. For string matching process we need to find a "Failure Function" FF which will give the amount of shift that is not necessarily invalid. If j characters are already successfully matched and there is a mismatch position $j+1$, then the output of the FF is the largest prefix of $P[1..j]$ that is also a suffix of $P[2..j]$.

Algorithm for Failure function

FailureFunction(P)

1. $m = \text{stringlength}(P)$
2. $\text{FF}[1] = 0$
3. $k = 0$
4. for $j = 2$ to m
5. while $k > 0$ and $P[k+1] \neq P[j]$
6. do $k = \text{FF}[k]$
7. if $P[k+1] = P[j]$
8. then $k = k + 1$
9. $\text{FF}[j] = k$
10. return FF

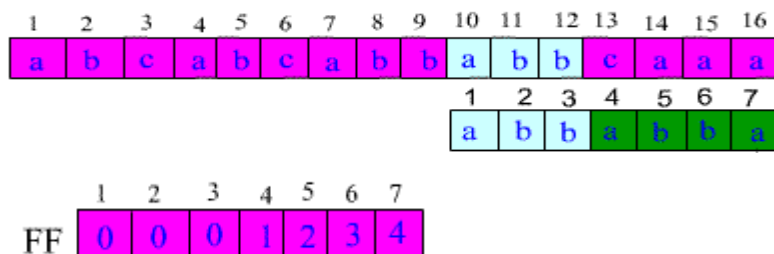


Fig: 5: failure function for the above pattern

The running time of FailureFunction is $O(m)$ where m is the length of the pattern P .

Algorithm for KMP string matcher

KMP string matching first preprocesses the given pattern P to compute Failure Function. Then the output of the Failure Function is used to skip some of the invalid comparisons. Note that the preprocessing is done once for each pattern. Since in general the size of the pattern to be searched is relatively small, it takes far less time than the string matching process.

KMP stringMatch (T, P)

1. $n = \text{stringlength}[T]$
2. $m = \text{length}[P]$
3. $FF = \text{FailureFunction}(P)$
4. $j = 0$
5. for $i = 1$ to n
6. while $j > 0$ and $P[j+1] \neq T[i]$
7. $j = FF[j]$
8. if $P[j+1] = T[i]$
9. then $j = j+1$
10. if $j = m$
11. then print "Pattern found with shift " $i - m$
12. $j = FF[j]$

KMP algorithm runs in optimal time $O(m+n)$ time.

14.4 Problems-String algorithm

- For each of the following cases find the number of comparisons to find the index (first occurrence) of the pattern P in the text T .
 - $P = \text{cat}, T = \text{bcbcbcbc}$
 - $P = \text{bbb}, T = \text{aabbaabbaabbaabbb}$
 - $P = \text{xxx}, T = \text{xyxyxyxyxyxyxyxy}$
- What is the complexity of the brute force string-matching algorithm in the best case.
- Write a procedure to count the number of the time the word 'the' appears in a given text.

- Find the output of The FailureFunction for the pattern $P = \text{aabbaababbabaa}$
- Can we find the occurrence of the pattern P in the text T by computing FailureFunction for the string PT which is the concatenation of the strings P and T . If so, then write an algorithm for that and what is the running time of your algorithm?
- Give best case inputs (both pattern and text) for KMP string matching algorithm.

UNIT XV: PROGRAM DEVELOPMENT

15.1 Life Cycle

- Software/program development is not just writing code, it is much more than that. It has a life cycle, which can be divided into following stages.
- Requirements: The description of what is required in general terms is called the requirements of the programmed.
- Program Specification: It is a complete description of what the program does.
- Code Design: Overall design of the programmed, including algorithms and data structures used, file formats, and modules definition.
- Coding: Writing code also involves standard coding practices like simplicity, clarity, commenting etc.
- Testing: Testing program for correctness.
- Debugging: Most programs fail first time. Finding the logical error and correcting the program is debugging process.
- Documentation: Writing a user manual for the program.
- Maintenance: After testing also programs are not perfect. Correcting bugs found in future is called the maintenance phase of the program.
- Updating: After using the program for some time, user may be need change program to add more futures or better algorithms and data structures.

The programmed life cycle is depicted in the below figure.

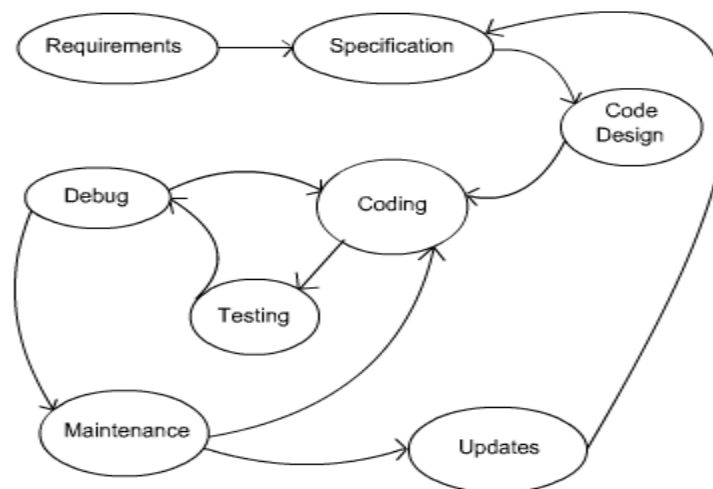


Fig: 1

We discuss these phases for the following problem.

Develop a program to count number of character, words and lines of given text files.

- **Requirements:** Count number of character, words and lines in given text files.
- **Specification:** Program take text file names as arguments and output number of characters, words, and lines in each text file.

1. After making the first draft of the specifications, it can be given to user and/or circulated to people working on this project.
2. Update the specifications as per their feedback.
3. For example, what happens if a file does not exists?
4. Program has to give an error message if file does not exist and proceed to next file. This has to be added to specifications.

- **Revised Specification:** Program take text file names as arguments and output number of characters, words, and lines in each text file. If any file does not exist, it gives an error message and proceeds to next file.

15.2 Code Designing

- If the program is large, it is desirable to divide into logical sections called modules. Each module can be further subdivided into sub modules depending on their size. Since, developing and testing smaller modules or sub modules can be done separately and easily.
- If more than one programmer involved in development, the task of developing these modules can be divided among them.
- Some programming languages, like C, allow program can be split into multiple file. In such languages, each module or group of logically related modules can be developed in a set of files by programmers and tested independently.
- In this stage, we have to decide the file formats, data structures and algorithms required.
- We name our program "line-count". Idea in coding is to read one character at time till it reaches end of a word or a line then increase corresponding counter.
- The "line-count" program is divided into three modules, character counting, word counting, and line counting.
- These entire modules can be developed in a single file, since the program is simple.

15.3 Coding

- In this "line-count" program, we can develop a partial program to count number of words only. If it works correctly then we can extend to counting number line. This is because, code for both are very similar.
- Where there are similar tasks, it is better to write code for any one task and test it. Then extend to other tasks.
- Around more than 70% of time is spend on maintaining, upgrading, and debugging the program. So it is important to writing a simple and easy to read program. A good programming style is an important aspect in the programming development.

15.4 Programming Style

The following aspects are important for a good programming style.

- **Comments:** Write comments to explain everything programmer has to know. Depending on the importance of the comment, one can format differently to get the program reader attention. For example most important comments can be kept in a comment box as shown below.

```

/ *** ----- ****
*** This program computes ***
*** number of lines, words ***
*** and character for the ***
*** input text file. ****
*** ----- **/

```

- Less important comment can be shown as follows.

```

/**-----
      comments for each source file
-----**/

```

This type of comments use for beginning of a function.

```

/*****
Comments beginning of each function
*****

```

This is the beginning of a section. It describes that how it works.

```
/***** Less Important Comments *****/
```

This type of comments use for less important message. For example, explaining how the next line work.

```
/----->> Simple comments explaining the
```

```
Next line <<-----/
```

- As per the requirements, one can have many different levels of comments.
- At the beginning of the program in main source file can contain boxed comment containing the heading, author, purpose, usage, references, file formats, limitations, and other related information.
 - Put comment on each declaration.
 - Put comments to explain everything programmer has to know.
- Indentation: Indent the program for readability. Now-a-days, many editors are coming with automatic program indentation.
- Simplicity: Many cases, simplicity in coding gives better and easy understanding of the program. Some of simplicity rules are given below.
 - Function should not be too large,
 - Avoid complex logic.
 - Keep expressions simple and short.

15.5 Problems- Program Developments

For each of the following programming assignment, follow programming life cycle to give from requirements to coding.

1. Write a program for four operator calculator.
2. Write a program to find the day of the given date.
3. Write a program to find all prime number in a given range of integers.
4. Write a program for library user administration.
5. Write a program for air-line reservation.

UNIT XVI: PROGRAM TESTING AND VERIFICATION

16.1 Testing Method

Important commands of gdb :

- `break [file :] function` : Set a breakpoint at the beginning of function (in file) .
- `break [file:] #n` : Set a breakpoint at line #n (in file) .
- `run [arglist]` : Start your program (with command line arglist , if any).
- `next`: Execute next line of the program (after stopping). That is, executes next line completely, including function calls (if any). It is called step over any function calls in the line.
- `c` : Continue executing the program (after stopping) till next breakpoint, if any.
- `print expr` : Display the value of an expression.
- `list [file :] function` : Type the text of the program in the vicinity of where it is presently stopped.
- `step`: Execute next program line (after stopping). If there is any function call present in the next program line, it goes into 1 st line of that function. It is called step into any function calls in the line.
- `quit` : Exit from GDB.
- `q` : Exit from GDB.
- `help` : Show information about GDB commands.
- `help [name]` : Show information about GDB command name .

16.2 Verification Procedures

Palindrome problem:

A palindrome is a string that reads same in either direction. A palindrome can be string, phrase, numbers, or any sequence of symbols.

We debug the following program (`palindrome.c`) which takes a string as command line argument and tests whether it is a palindrome or not.

- Compile the program `gcc -g -o palindrome palindrome.c`
- It generates an executable program named `palindrome`.

- Start the debugger with the name of executable program to be debugged. In our case, command is `gdb palindrome`
- The debugger out puts its prompt (gdb)
\$ `gdb palindrome` (gdb)
- We keep first break point at the first instruction of the main function that is the line 7, to stop the execution immediately after started.
(gdb) `break main`
Breakpoint 1 at 0x80483da: file palindrome.c, line 7.
(gdb)
- Start the execution by the command `run madam`
(gdb) `run madam`
Starting program: `palindrome madam`
Breakpoint 1, main (argc=2, argv=0xafb79394) at
`palindrome.c:7`
`if (argc != 2) {`
(gdb)
- Program is stopped at the line 7.
- Now we can keep more break points or execute the program step by step.
- We will go for keeping more break points. We have to decide where to keep break points?
- To see the part of the code around the breakpoint use the command `list`. It displays 10 lines around the current breakpoint.
 1. (gdb) `list`
 2. `#include<string.h>`
 3. `main(int argc, char *argv[])`
 4. `{`
 5. `int i,length, mid;`
 6. `if (argc != 2) {`
 7. `printf("Error in input.\n");`
 8. `exit(1);`
 9. `}`

- It may not be necessary to keep the break point in the above part of the program. So we will see the code around line 15.

```
(gdb) list 15
      }
```

```
1.     length = strlen(argv[1]);
2.     mid = length/2;
3.     for (i=0; i<=mid; ++i){
4.         if (argv[1][i] != argv[1][length-i]){
5.             printf("\'%s\' is not a palindrome.\n", argv[1]);
6.             exit(0);
7.         }
8.     }
```

(gdb)

- There could be an error in the for loop. So, keep break point at the line 15.

```
(gdb) break 15
```

```
Breakpoint 2 at 0x8048435: file palindrome.c, line 15.
```

```
(gdb)
```

- Continue execution using the command c .

```
(gdb) c
```

```
Continuing.
```

```
Breakpoint 2, main (argc=2, argv=0xafb79394) at palindrome.c:15
```

```
15     if (argv[1][i] != argv[1][length-i]){
```

- (gdb)

- Program giving wrong output, that it is executing lines 16 and 17, which is in the if part.
- We have to check why if condition is satisfied?
- Use the print command to display the values stored in variables or expression.

```
(gdb) print length
```

```
$1 = 5
```

```
(gdb)
```

- Length of the input string "madam" is correct. Check other variables.

```
(gdb) print mid
```

\$2 = 2

(gdb)

- Value stored in mid also correct. Check left and right expression of the if condition.

(gdb) print argv[1][i]

\$3 = 109 'm'

(gdb) print argv[1][length-i]

\$4 = 0 '\0'

(gdb)

- The left expression is 'm' and the right expression is '\0' which are not equal!
- Why? It is comparing 'm' and '\0'. It is supposed to compare 'm' and 'm', which are first and last character of the input string. Check indexes used to compare? That is, i and length - i .

(gdb) print i

\$5 = 0

(gdb) print length-i

\$6 = 5

(gdb)

- Index of an array of length 5 varies between 0 and 4. But we are comparing the character stored in the indexes 0 and 5, which is not correct.
- Index should be used in the right expression should be length-i-1.
- Quit the gdb.

(gdb) q

The program is running. Exit anyway? (y or n) y

\$

- Edit the program, compile, and execute.
- Now, the program works correctly.

Run time Errors:

- Correcting run time errors are easier than correcting logical errors.
- We discuss most important run time errors.
- Segmentation fault: It occurs when a program try to access a memory location that it is not allowed to access.

- 1 This location may part of the operating system, other user, or other processes.
- Divide by zero: Many operating systems report message Floating point exception, when there is expression which denominator of the division is 0.
 - Stack Over flow: Most of the cases it happen, when there is infinite recursion. In recursive function calls, program has to store various environment variables, before start executing the calling function. It can also happen when there are many big temporary arrays.
 - Whenever these errors occur, program execution stops with appropriate error message.
 - These errors also can be debugged using gdb.

16.3 Problems- Testing and Verification

- Take any of your programs and run it using a interactive debugger and examine intermediate values.