Technical and vocational Training Institute

Course Title: Introduction to Data Mining and warehousing.

Course No:

Credit Hr: 3 hrs

Contact hrs; 2 Lecture , 3 Lab

Course outline

Chapter One

Introduction

Chapter Two

Data Pre-Processing

Chapter Three

Mining Association Rules in Large Databases

Chapter Four

Classification and Prediction

Tools for Use

- Primarily **WEKA** and **Excel** as necessary

Chapter One
Introduction

- In this chapter we will cover the following issues in brief
  - Motivation: Why data mining?
  - Data mining & Data Warehousing
  - DM Models
  - CRISP-DM Model
  - DM and Business Intelligence
  - Application of Data Mining
  - Data Mining Functionalities
  - Are all the Patterns Interesting?
  - Classification of Data Mining Systems

Motivation:
"Necessity is the Mother of Invention"

- Our capacity of generating and collecting data have been increased rapidly in the last several decades.
- Huge amount of data is available at our finger tip
- It is predicted that more data will be produced in the next year than has been generated during the entire existence of humankind!
- Many factors related to ICT development made it easier to create, collect, and store all types of data.

Motivation:
"Necessity is the Mother of Invention"

- As a result it creates a problem of what is called ***Data Exposition***
- *Data Explosion* is the problem of having huge amount of data in an enterprise stored in ***transaction*** or ***relational databases***, ***data warehouses*** and other ***information repositories***
- These dataset are generated by automated data collection tools and mature database technology in large databases
- These dataset are in turn needs to be processed to make a decision.
- As the size of data gets larger and larger, analyzing the data becomes ***very difficult***

Motivation:
"Necessity is the Mother of Invention"

- Data can be managed and stored in
  - structured relational databases;
  - in semi-structured file systems, such as e-mail;
  - unstructured fixed content, like documents and graphic files.
- Companies rely on this enterprise data to improve decision-making and to gain a competitive advantage;
- Data has indeed become a highly valued business asset.
- The huge amount of data exceeds our human ability to make comprehension on the data and to put the best decision without tools
- Generating and storing of large volumes of data has reached a critical mass and appropriate ***tools*** to comprehend the data becomes vital.

Motivation:
"Necessity is the Mother of Invention"

- We are data rich, but starving for knowledge (Info or Knowledge Poor)!
- The Solution: **Data Mining**
- ***Data Mining*** can be viewed as a result of the natural evolution of Information Technology.
- This can be more explained if we look at the evolution of database technology since 19^{th} century.

Data Mining & Data warehousing

- Data mining is extraction of interesting (**non-trivial, implicit, previously unknown and potentially useful**) information or patterns from data in large databases (data warehouse) or data sources in general

- The term Data mining is a misnomer as it doesn't directly related to what it does.

- For example mining gold from rock is called Gold mining but not rock mining.
- Similarly oil mining is mining oil from the ground.

- Data mining should best describe as knowledge mining from data rather than data mining
- Any way, we will use the term with this understanding

Data Mining & Data warehousing

- Alternative names
  - Knowledge Discovery from databases (KDD),
  - Knowledge Mining
  - knowledge extraction,
  - data/pattern analysis,
  - data archeology,
  - data dredging,
  - information harvesting,
  - business intelligence, etc.

- Note that:
  - Query processing systems, Expert systems (knowledge base systems) or Information retrieval systems are not data mining tasks as they are not intended to extract knowledge from data source

Data Mining & Data warehousing

- Data can now be stored in many different types of databases.
- Special DB architecture that has recently emerged is ***data warehouse***
- **Data warehouse** is a repository of multiple heterogeneous data sources organized under a unified schema at a single site in order to facilitate management decision making processes
- **Data warehouse** is a *subject-oriented*, *integrated*, *time-variant*, and *nonvolatile* collection of data in support of management's decision-making process."

Data Mining & Data warehousing

- Data warehouse technology includes
  - Data cleansing (filing missing values, and inconsistent data, clearing noisy data)
  - Data integration (combining multiple heterogeneous data sources into one data warehouse)
  - On-Line Analytical Processing (OLAP)

Data Mining & Data warehousing

- The abundance of data, coupled with the need for powerful data analysis tools has been described as data rich but information poor situation

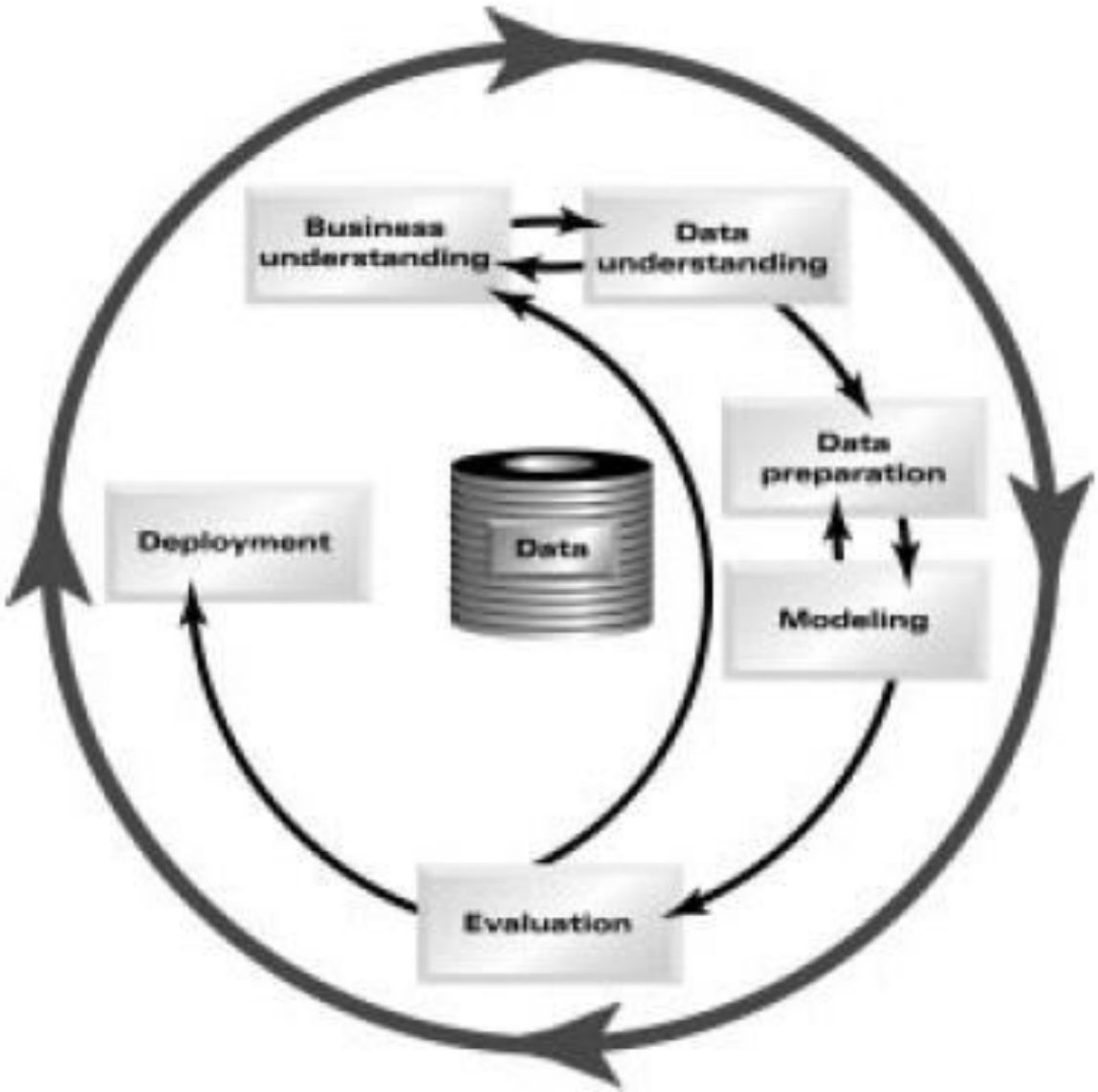
Data Mining Models

- Data mining (Knowledge Discovery in Databases) consists of iterative sequences of steps
- The steps used to systematically discover knowledge from a huge collection of data source are usually referred as DM models
- There are a number of DM models one of the most common is the CRISP-DM model (CRoss-Industry Standard Process for Data Mining)

- CRISP_DM model is the emphasis of the course DM modeling methodology

CRISP-DM Models

- Involves the following steps



CRISP-DM Models
Business/Research Understanding

- Focus on understanding the project objectives and requirements from a business perspective

- Convert this into a data mining problem definition and preliminary plan designed to achieve the objectives

- Enable us to learn relevant prior knowledge to be used in the DM process and to learn the goals of DM application

CRISP-DM Models
Data Understanding

- Start with an initial data collection and proceed with activities in order to:
  - Get familiar with the data;
  - Identify data quality problems;
  - Discover first insight into the data;
  - Detect interesting subsets ;
  - Form hypothesis for hidden information

CRISP-DM Models
Data Preparation

- Cover all activities to construct the final data set from the initial row data
- Tasks include:
  - Table, Record and Attribute Selection
    - consolidating and amalgamating (combining) records, summarizing fields,
  - Data Cleaning (may take 60% of effort!)
    - checking for data integrity, detecting irregularities and illegal attributes, filling in for missing values, trimming outliers.
  - Data Reduction and Transformation
    - Find useful features, dimensionality/variable reduction, invariant representation

CRISP-DM Models
Modeling

- Appropriate DM functionality will be selected
  - *summarization, classification, regression, association, clustering.*
- Appropriate algorithm for the selected DM function will be selected
- Select appropriate data mining tools
- Transforming the data if the tools require it
- Generating samples for training and testing the model and
- Finally using the tools to build and select a model

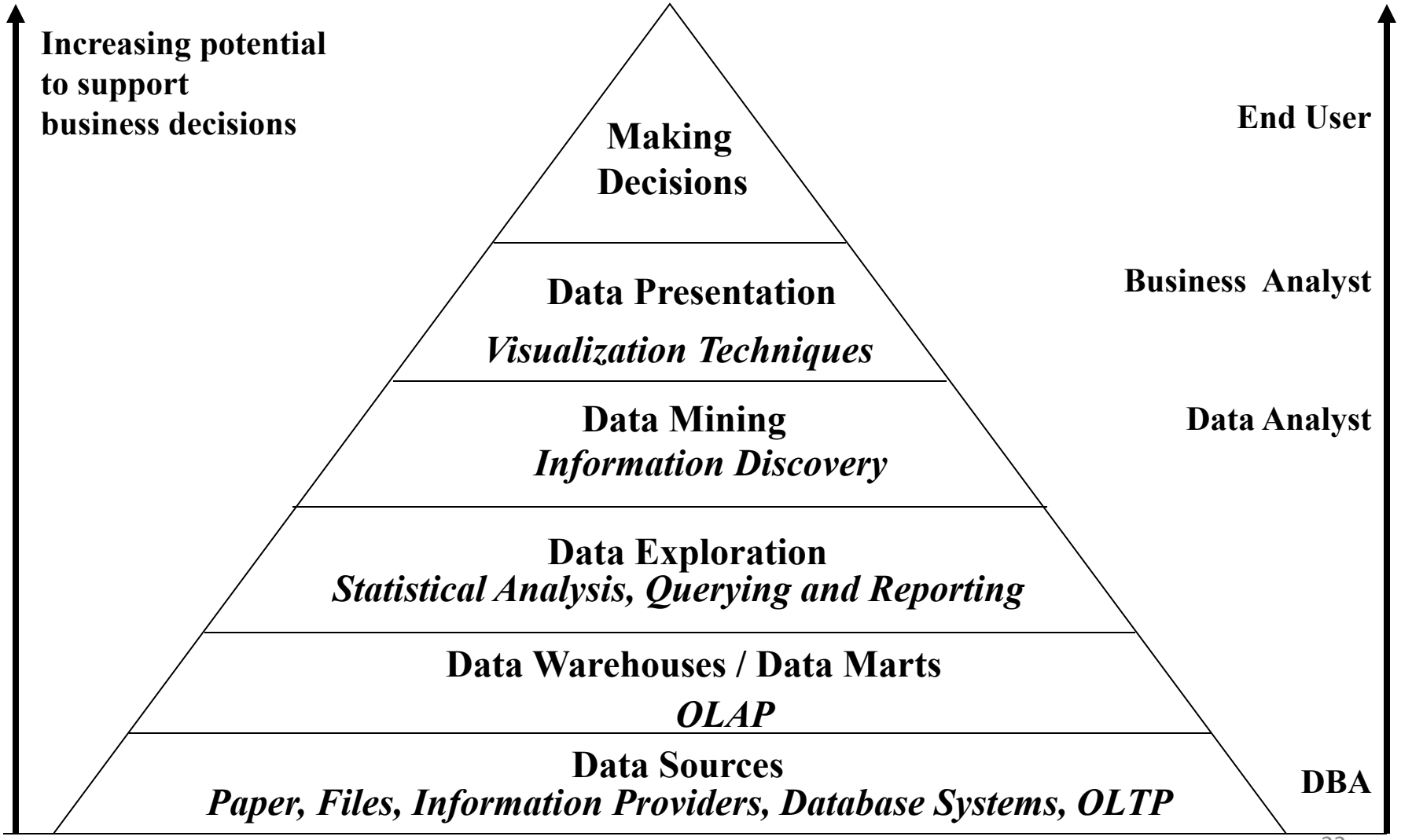
CRISP-DM Models
Evaluation

- It is a step that we need to thoroughly evaluate the model performance and
- Review the steps executed to construct the model to be certain of it to properly achieve the business objective
- A key objective is to determine if there is some important business issue that has not been sufficiently considered if not to reconsider the entire process

CRISP-DM Models
Deployment

- Refers to the process of making the implemented model for knowledge discovery usable by the customer
- Involves how to make the system live and visual to decision making

Data Mining and Business Intelligence



Potential Applications

- Data mining has many and varied fields of application some of which are listed below.

1. Marketing Analysis which can be
   - Retail Analysis (Retail analysis refers to how an individual or business that sold items maximize their profit or selling capacity)
   - Banking Analysis
2. Fraud detection and management
3. Insurance and Health Care Analysis
4. Transportation Analysis

Potential Applications

- Data mining can be applied in various application areas

1. Target Market Analysis
   - Given a product, who should be targeted so that he/she/they will buy the product
   - This need to analyze what type of customer potentially buy what type of product
   - This enable us where, when and to whom to make the promotion using various strategies.
2. Market Basket Analysis
   - Which item should be clustered on a shopping shelf together so that customers will be comfortable to find items that they need to buy in the same vicinity

Potential Applications

3. Market Segmentation
   - Clustering customer by their demographic, geographic behavior and physiological information and treat them accordingly.
4. Cross Market Analysis
   - Involves Analysis of the correlation/association between set of two or more product
   - Does the increase in amount of sell of one product affect positively or negatively the other product.
6. Fraud detection and management
   - Intrusion detection
   - Misuse detection of resources
   - Malicious activity detection
   - Credit card fraud

Potential Applications

7. Bioscience
   - Sequence-based analysis of protein
   - Protein structure and function prediction.
8. Trend analysis
   - What will happen if thing move like this in any area?

9. Predict customer spending
   - Predict how much a customer will invest if catalog is send to him to purchase items

10. Sales forecasting
    - Forecast how much retailers will sell in the future

Potential Applications

11. Identify loyal customers vs. risky customer
    - Load risk assessment
    - Abuse protection of various insurance of customers

12. Risk analysis and management
    - Forecasting, customer retention, improved underwriting, quality control, competitive analysis.

14. Science: Chemistry, Physics, Medicine
    - Biochemical analysis
    - Remote sensors on a satellite
    - Telescopes – star galaxy classification
    - Medical Image analysis

Data Mining Functionalities

- Data mining can be performed on various types of data stores and Databases
- Data mining functionalities are used to specify the kind of patterns to be found in data mining task
- The kind of pattern mined form a given data is not known for the user.
- Techniques should be implemented to extract various pattern from the available data so that user can choose what they need to use.
- There are different kinds of data mining functionalities (tasks) that can be used to extract various types of pattern from data

Data Mining Functionalities

- Generally data mining task can be broadly classified as
  - **Descriptive (unsupervised)**
  - **Predictive (Supervised)**
- Descriptive data mining task characterize the general properties of the data in a database.
- This kind of data mining is usually unsupervised
- Predictive data mining task perform inference on the current data in order to make prediction to the future reference
- This is usually supervised technique

Data Mining Functionalities

- The supervised predictive data mining functionalities includes
  - **Classification**
    - Finding class label for unknow data point
  - **Regression**
    - Predicting the value associated to a given dependent variable
  - **Time series**
    - Finding the relation between previous observation and the current on a given variable
  - **Estimation**
    - Estimate missing element which can be either classification or regression or time series

Data Mining Functionalities

- The unsupervised descriptive data mining functionalities includes
  - **Concept /class description:** Characterization (summarization) and discrimination
    - Characterization refers to describing a given set of data that belongs to a class
    - Discrimination refers to finding similarities and differences among a class with another set of class (classes)
  - **Association Analysis**
    - Finding how one or more attribute and their values are related into a cause and effect relationship
  - **Clustering analysis**
    - Finding the possible set of classes for a given set of data that maximize the intra cluster distance and minimizes the inter cluster distance
  - **Outlier analysis**
    - Find a group of data set or activities that deviate from the normal behavior
  - **Trend or Evolution analysis**
    - Try to see the evolutionary trend from the data and may be used to predict about the present or the future

Are All the "Discovered" Patterns Interesting?

- A data mining system/query may generate thousands of patterns, not all of them are interesting
- Questions
  - What makes a pattern interesting?
  - Can a data mining system generate all of the interesting patterns?
  - Can a data mining system generate only interesting patterns?
- Answers for all the three questions will be given bellow

Question 1
Are All the "Discovered" Patterns Interesting?

- **A pattern is interesting if**
- **it is <u>easily understood</u> by humans**
- **valid on new or test data with some degree of certainty,**
- **potentially useful,**
- **novel,**
- **or validates** some hypothesis that a user seeks to confirm

An interesting pattern represents knowledge

- **<u>Measure of Interestingness measures</u>**
  - **Two types (Objective vs. subjective)**
    - **<u>Objective:</u>** based on statistics and structures of patterns, e.g., support, confidence, Error (Mean Square error, absolute error, etc), Similarity measure, etc.
    - **<u>Subjective:</u>** based on user's belief in the data, e.g., unexpectedness (contradicting a user's belief), novelty, actionability, etc.

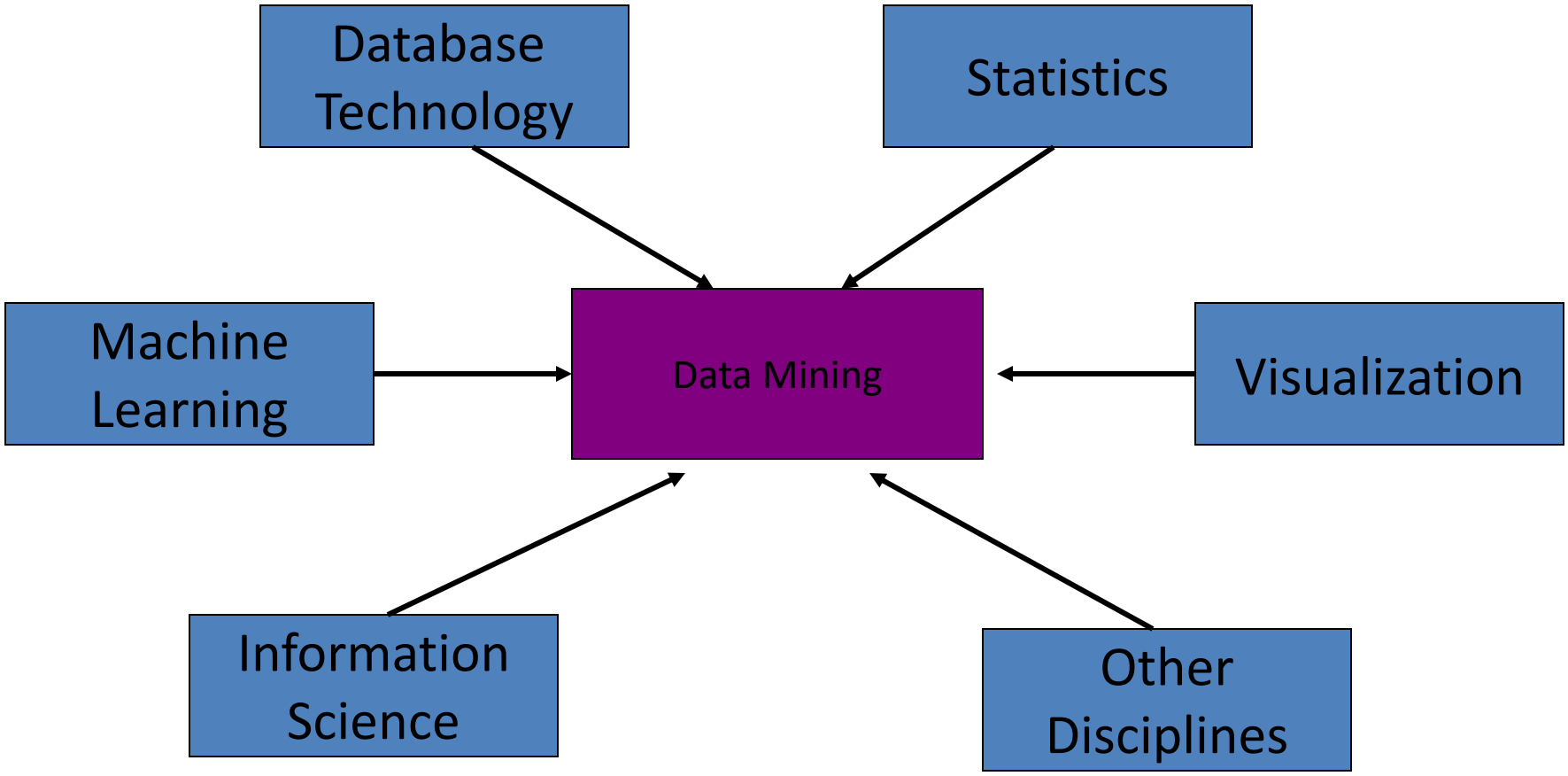
Question 2
Can We Find All Interesting Patterns?

- Referred as Completeness of the data mining algorithm
- No single data mining system is complete but users can set a constraint on the type of pattern they are looking for in which the data mining function generate all the pattern with the specified constraints
- Association algorithms don’t find classification pattern and others for example

Question 3
Can We Find Only Interesting Patterns?

- This is an Optimization problem in data mining system
- It remain an challenging issue
- Approaches in the research topic
  - First generate all the patterns and then filter out the uninteresting ones.
  - Generate only the interesting patterns—mining query optimization

Data Mining: Confluence of Multiple Disciplines



Data Mining: Classification Schemes

- General functionality
  - Descriptive data mining
  - Predictive data mining
- Different views, different classifications
  - Kinds of databases to be mined
  - Kinds of knowledge to be discovered
  - Kinds of techniques utilized
  - Kinds of applications adapted

A Multi-Dimensional View of Data Mining Classification

- **Databases to be mined**
  - Relational, transactional, object-oriented, object-relational, active, spatial, time-series, text, multi-media, heterogeneous, legacy, WWW, etc.
- **Knowledge to be mined**
  - Characterization, discrimination, association, classification, clustering, trend, deviation and outlier analysis, etc.
  - Multiple/integrated functions and mining at multiple levels
- **Techniques utilized**
  - Database-oriented, data warehouse (OLAP) oriented, machine learning, statistics, visualization, neural network, etc.
- **Applications adapted**
  - Retail, telecommunication, banking, fraud analysis, DNA mining, stock market analysis, Web mining, Weblog analysis, etc.

END

Thank You

Chapter Two

Data Preprocessing

Why Data Preprocessing?

- Quality decisions must be based on quality data
- Data in the real world is full of dirty.
  - **Incomplete**:
    - lacking attribute values that is vital for decision making so they have to be added.
    - lacking certain attributes of interest in certain dimension and should be again added with the required value.
    - containing only aggregate data so that the primary source of the aggregation should be included.
  - **Noisy**: containing *errors* or *outliers* that deviate from the expected Values
  - **Inconsistent**: containing discrepancies in codes or names of the organization or domain.

Major Tasks in Data Preprocessing

- Data pre-processing in data mining activity refers to the processing of the various data elements to prepare for the mining operation.
- Any activity performed prior to mining the data to get knowledge out of it is called **Data Pre-Processing**
- This involves:
  - Data Cleaning
  - Data Integration
  - Data Transformation
  - Data Reduction
  - Data Discretization and
  - Conceptual Hierarchy Generation

Data Cleaning

Refers to the Process of

- Filling in Missing Values,
- Smooth Noisy Data,
- Identify or remove outliers, and
- Resolve inconsistencies

Data Cleaning: Missing Data

- Data of interest is not always Available
  - E.g., many tuples have no recorded value for several attributes, such as customer income in *sales data*
- Missing data may be due to
  - Measurement Equipment Malfunction
  - Inconsistent with other recorded data and thus ignored during data entry
  - data not entered due to misunderstanding
  - certain data may not be considered important at the time of entry
- Missing data may need to be inferred.

Data Cleaning: How to Handle Missing Data?

- **Ignore the tuple:** usually done when class label is missing (assuming the tasks in classification—not effective when the percentage of missing values per attribute varies considerably.

- **Fill in the missing value manually:** tedious and infeasible
- **Use a global constant to fill in the missing value:** e.g., "unknown", a new class?! Simple but not recommended as this constant may form some interesting pattern for the data mining task which mislead decision process

Data Cleaning: How to Handle Missing Data?

- **Use the attribute *mean*** for all samples belonging to the same class to fill in the missing value

- **Use the most probable value** to fill in the missing value: inference-based such as Bayesian formula or decision tree

- Except the first two approach, the rest filled values are incorrect

- The last two approaches are the most commonly used technique to fill missing data

Data Cleaning: Noisy Data

- Noise: random error or variance in a measured variable
- Incorrect attribute values may be due to
  - Faulty data collection instruments
  - Data entry problems
  - Data transmission problems
  - Technology Limitation
  - Inconsistency in Naming Convention
  - Duplicate Records
  - Incomplete Data per Field

Data Cleaning: How to Handle Noisy Data?

- Noisy data can be handled by the techniques such as
  - Simple Discretization Methods (Binning method)
  - Clustering
  - Regression
  - Combined computer and human inspection
    - detect suspicious values and check by human

Data Cleaning:
Handling Noisy Data by Simple Discretization Methods (Binning)

- Sort data and partition into bins
- The bins can be ***equal-depth*** or ***equal-width*** bin
- Choose smoothing algorithm and apply the algorithm on each bin
- The algorithm can be
  - smooth by bin means,
  - smooth by bin median,
  - smooth by bin boundaries, etc.

Data Cleaning:

Handling Noisy Data by Simple Discretization Methods (Binning)

- Equal-width (Distance) Partitioning:
  - It divides the range into *N* intervals of equal size: uniform grid
  - if *A* and *B* are the lowest and highest values of the attribute, the width of intervals will be: $W = (B-A)/N$.
  - This approach leads into bins with non uniform distribution of data elements per bin
  - The most straightforward approach
  - But outliers may dominate presentation
  - Skewed data is not handled well.

Data Cleaning:
Handling Noisy Data by Simple Discretization Methods (Binning)

- Equal-width (distance) partitioning:
  - Given the data set (say 24, 21, 28, 8, **4**, 26, **34**, 21, 29, 15, 9, 25)
    - Determine the number of bins : N (say 3)
    - Determine the range R = Max – Min
    - Divide the range into N equal width where the ith bin is [Xi-1, Xi) where X0=Min and XN=Max and Xi = Xi-1 + R/N
    - For the above data R = 30, R/3 = 10, X1 = 14, X2 = 24, and X3 = 34
    - First sort the data as 4, 8, 9, 15, 21, 21, 24, 25, 26, 28, 29, 34
    - Therefore in our case Bin 1 = 4,8,9 Bin 2 = 15, 21, 21 Bin3 = 24, 25, 26, 28, 29, 34

Data Cleaning

Handling Noisy Data by Simple Discretization Methods (Binning)

- Equal-Depth (Frequency) partitioning:
  - It divides the range into *N* intervals, each containing approximately same number of samples
  - Given the data set (say 24, 21, 28, 8, 4, 26, 34, 21, 29, 15, 9, 25)
    - Determine the number of bins : N (say 3)
    - Determine the frequency F of the data set (12)
    - Determine the number of sample per bin F/N (12/3 = 4)
    - Sort the data as 4, 8, 9, 15, 21, 21, 24, 25, 26, 28, 29, 34 and place F/N element in order into a bin
    - Therefore in our case Bin1 = 4,8,9 ,15 Bin2 = 21, 21,24, 25 Bin3 = 26, 28, 29, 34
  - Good data scaling
  - Managing categorical attributes can be tricky.

Data Cleaning:

Handling Noisy Data by Simple Discretization Methods (Binning)

Smoothing Algorithm

- Given the data set in bins as say:
  - **Bin 1:** 4, 8, 9, 15
  - **Bin 2:** 21, 21, 24, 25
  - **Bin 3:** 26, 28, 29, 34

- Smoothing by bin means:
  - Find the mean in each bin and replace all the element by the bin mean
  - **Bin 1:** 9, 9, 9, 9
  - **Bin 2:** 23, 23, 23, 23
  - **Bin 3**: 29, 29, 29, 29

Data Cleaning:
Handling Noisy Data by Simple Discretization Methods (Binning)

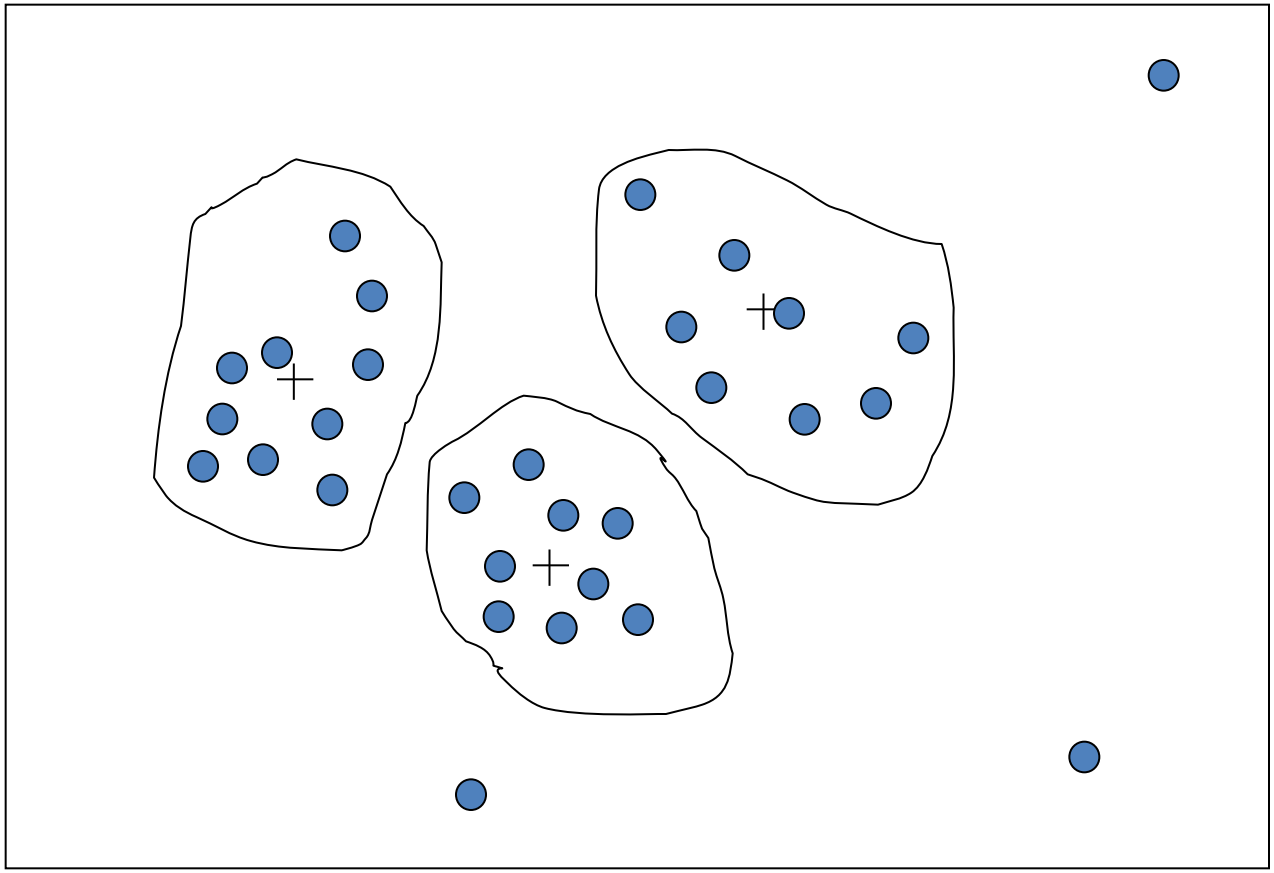
Smoothing algorithm

- Smoothing by bin median:
  - Find the median in each bin and replace all the element by the bin median
  - **Bin 1:** 8.5, 8.5, 8.5, 8.5
  - **Bin 2:** 22.5, 22.5, 22.5, 22.5
  - **Bin 3**: 28.5, 28.5, 28.5, 28.5

- Smoothing by bin boundaries:
  - Replace each element by the bin min or bin max which ever is the nearest
  - Distance is measured just as the absolute of the difference
  - **Bin 1:** 4, 4, 4, 15
  - **Bin 2:** 21, 21, 25, 25
  - **Bin 3:** 26, 26, 26, 34

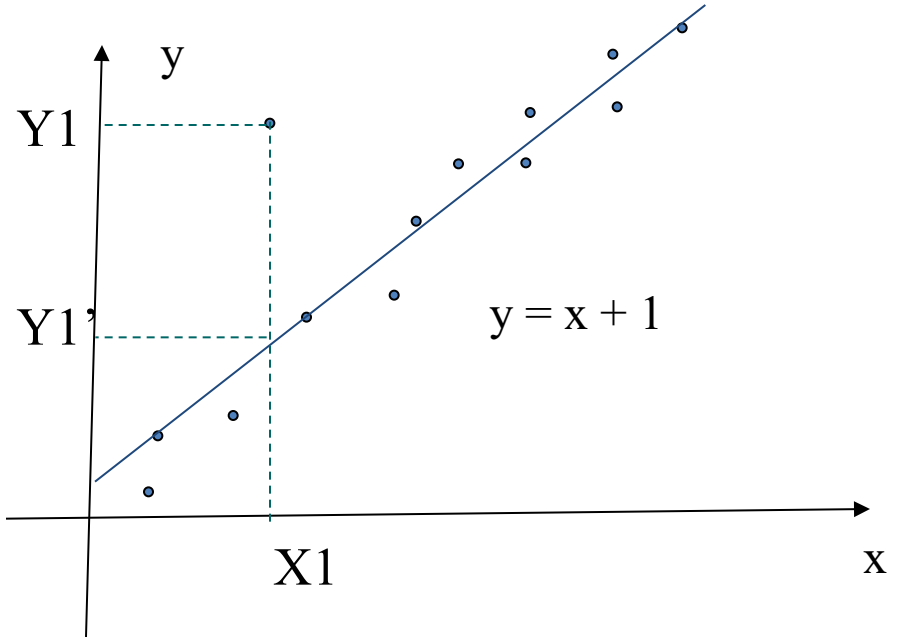
Data Cleaning:
Handling Noisy Data by Cluster Analysis

- Detect and Remove Outliers



Data Cleaning:
Handling Noisy Data by Regression

- Smooth by fitting the data into Regression Functions
- Finding a fitting function for two dimensional case so that the value of the second variable can be estimated using the value of the first variable



- It can be extended to N dimensional case in which regression finds multidimensional space so that value of one of the dimension can be predicted from the rest N-1 values

Data Cleaning:
Resolve Inconsistencies

- Involve write the same information in many ways such as Bahrdar vs Bahirdar, AAU vs A.A.U., $X vs Y Birr

- Very difficult and most important task which require manual investigation

Data Integration

- **Data integration:**
  - Combines data from multiple sources (***databases, data cubes, or files***) into a coherent store.

- There are a number of issues to consider during data integration

- Some of these are
  - Schema integration issue
  - Entity identification issue
  - Data value conflict issue
  - Avoiding redundancy issue

Data Integration

- Schema integration
  - Schema refers to the design of an entity and its relation in the data source
  - Integrate metadata from different sources

- Entity identification problem:
  - identify real world entities from multiple data sources which are identical so that they can be integrated properly
  - As data source for data mining differ, the same entity will have different representation in the different sources
  - How can equivalent real world entities from multiple sources can be matched up?
  - e.g., A.cust-id ≡ B.cust-#

Data Integration

- **Data Value Conflict Issue**
  - Involves detecting and resolving data value conflicts
  - For the same real world entity, attribute values from different sources may be different
  - Possible reasons: different representations, different scales, measurement unit used

Data Transformation

- Data transformation is the process of transforming or consolidating data into a form appropriate for mining which is more appropriate for measurement of similarity and distance

- This involves
  - Smoothing
  - Aggregation
  - Generalization
  - Normalization
  - Attribute/feature construction

Data Transformation

- **Smoothing:** concerned mainly to remove noise from data using techniques such as binning, clustering, and regression

- **Aggregation:** summarization, data cube construction

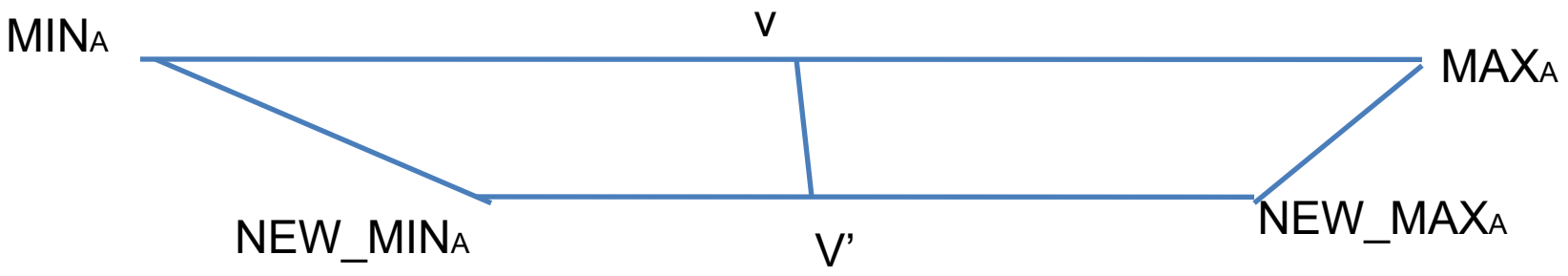
- **Generalization:** concept hierarchy climbing (from low level into higher level)

- **Normalization:** scaled to fall within a small, specified range
  - Used mainly
    - for classification algorithms such as neural network,
    - distance measurements such as clustering, nearest neighbor approach
  - Exist in various forms

1. min-max normalization
2. z-score normalization
3. normalization by decimal scaling
4. Attribute/feature construction

Data Transformation: Normalization

- **min-max normalization**
  - Perform a linear transformation on the original data into a range specified range of min and max value



$$v' = \frac{v - min_A}{max_A - min_A}(new\_max_A - new\_min_A) + new\_min_A$$

Data Transformation: Normalization

- z-score (zero mean) normalization
  - A value will be normalized based on the mean and standard deviation of the original data
  - The transformed data will have zero mean value

$$v' = \frac{v - mean_A}{stand\_dev_A}$$

Data Transformation: Normalization

- Normalization by decimal scaling
  - Normalizes by moving the decimal point of values of attribute A.
  - The resulting value ranges from -1 to +1 exclusive

$$v' = \frac{v}{10^j}$$ Where j is the smallest integer such that Max($|v'|$)<1

- For example, given the data set V= 132, -89, 756, -1560, 234, -345 and 1234
- The value of v' becomes in the range from -1 to +1 if j=4
- In that case V' = 0.0132, -0.0089, 0.0756, -0.156, 0.0234, -0.0345, and 0.1234

Data Transformation: Attribute construction

- Attribute construction is the process of driving new attributes from the existing attributes.
- Attribute construction is important to improve performance of data mining as the derived attribute will have more discriminative power than the base attributes
- Enable to discover missing information or information hidden within the data set

- For example
  - Rate of telephone charge can be derived from billable amount and call duration
  - area can be derived from width and height

Data Reduction

- Data sources may store terabytes of data
- Complex data analysis/mining may take a very long time to run on the complete data set
- Data reduction tries to obtain a reduced representation of the data set that is much smaller in volume but yet produces the same (or almost the same or better) analytical results

Data Reduction

- Data reduction strategies includes
  1. **Data cube aggregation**
     - Apply data cube aggregation to have summarized data to work with DM functionalities
  2. **Attribute subset selection**
     - Select only relevant attribute from the set of attributes that the data set has
  3. Numerousity reduction
     - **Regression and log-linear models** (once analyzed store only the parameters that determine the regression and the independent variable values
     - **Histograms:** store the frequency of each bin in the histogram rather than each element in detail
     - **Clustering:** store the cluster labels rather that the individual elements
     - **Sampling:** use representative sample data rather than the entire population of data

Data Reduction

- Data reduction strategies includes
  4. **Dimensionality reduction**
     - ***Huffman coding*** (text coding or compression algorithm)
     - ***Wavelet transforms*** (transforming usually image data into important set of coefficients called wavelets)
     - ***Principal component*** analysis (representing N sequence of M dimensional data by using small parameters having either N or M elements which ever is possible)
       - It can be used for a series image data reduction where N can be the number of pixels and M the important information on the pixel at the same location
       - If each parameter has N elements, we use it for generating every image from the parameters and the reduced image coefficients
       - If each parameter has M elements, we use it for generating every pixel of all the image from the parameters and the reduced pixel coefficient values

Data Discretization and concept hierarchy generation

- Data discritization refers to transforming the data set which is usually continous into discrete interval values
- Concept hierarchy refers to generating the concept levels so that data mining function can be applied at specific concept level
- Data discretization techniques can be used to reduce the number of values for a given continuous attribute by dividing the range of attribute into intervals
- Interval labels can be used to replace actual data values
- This leads to concise, easy to use, knowledge level representation of mining result

Data Discretization and concept hierarchy generation

- Discretization:
  - Divide the range of a continuous attribute into intervals
  - Reduce data size by discretization
  - Prepare for further analysis
  - Some classification algorithms only accept categorical attributes.
- Concept hierarchies
  - Reduce the data by collecting and replacing low level concepts (such as numeric values for the attribute age) by higher level concepts (such as young, middle-aged, or senior).
  - Concept hierarchy generation refers to finding hierarchical relation in the data set and build the concept hierarchy automatically

Discretization and concept hierarchy generation for numeric data

- Three types of attributes:
  - Nominal — values from an unordered set like location, address
  - Ordinal — values from an ordered set like age, salary
  - Continuous — real numbered values such as height, frequency

- It is difficult and laborious to specify concept hierarchies for numerical attributes because of the wide diversity of possible data ranges and frequent update of data values

- Concept hierarchies for *numerical data* can be constructed automatically based on data ***Discretization algorithms***

Discretization and concept hierarchy generation for numeric data

- The following are methods for discretization and concept hierarchy generation

1. Binning :
   - This will discretize the attributes by putting the attribute instances and into the different bins and replace their values by the bin statistics
   - Doing it recursively in a top down approach will generate concept hierarchy
2. Histogram analysis
   - This will discretize the attribute by generating equal width or depth interval and replace the attribute instance value by the depth or width of the histogram
   - Doing it recursively in a top down approach will generate concept hierarchy

Discretization and concept hierarchy generation for numeric data

3. Clustering analysis

   - This will discretize the attribute and doing it recursively in a top down or bottom up approach will generate concept hierarchy
   - The bottom up approach will start by generating larger number of class and incrementally merge some of them forming the concept hierarchy from the bottom up till the top

4. Entropy-based discretization

   - This is appropriate if the data set has class label and the task is classification
   - For a given continuous or nominal values it finds the best split point for the possible values using concept of information gain and entropy

Discretization and Concept Hierarchy Generation for Categorical Data

- Categorical data are discrete data.
- Categorical attributes have a finite (but possibly large) number of distinct values with no ordering among the values
- Examples of categorical attributes: *geographic location, job category, item type, fertilizer recommended, citizen, and color*

Discretization and concept Hierarchy Generation for Categorical data

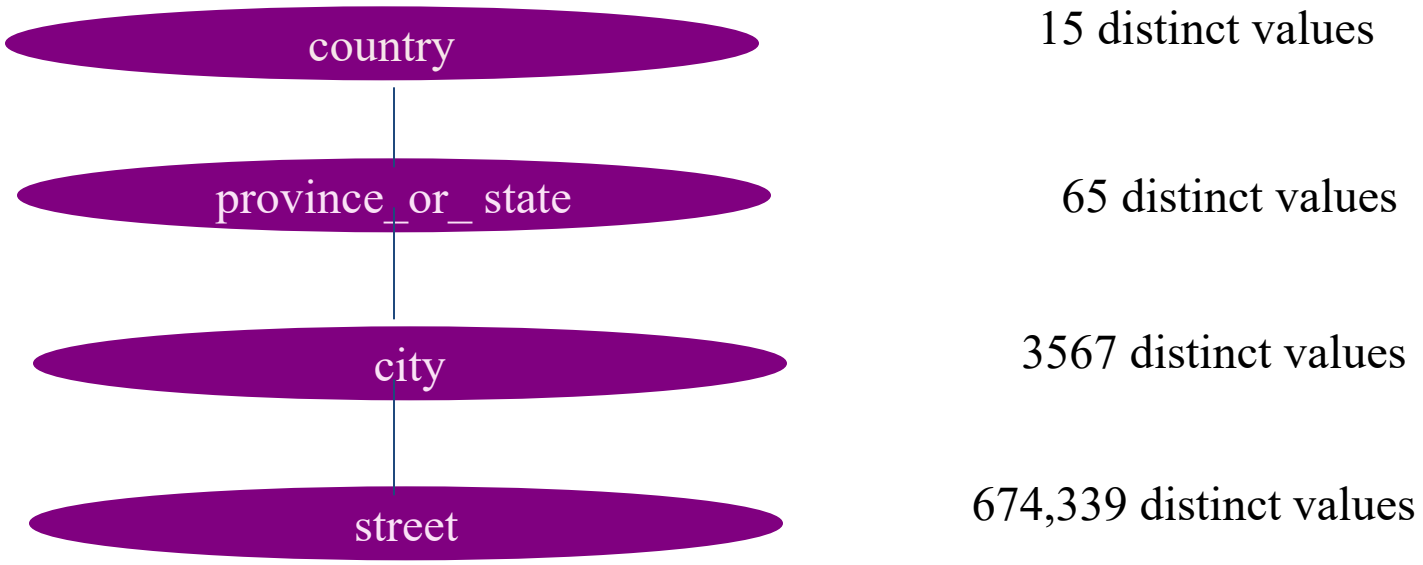
- Methods for the generation of concepts hierarchies for categorical data includes
  1. Specification of a partial or total ordering of attributes explicitly at the schema level by users or experts
     - If telephone number start with +251 it is from Ethiopia
     - Indication of telephone number type at schema level mobile/wired
     - Others

Discretization and concept hierarchy generation for categorical data

2. Specification of a portion of a hierarchy by explicit data grouping
   - This is essentially the manual definition of a portion of a concept hierarchy.
   - It helps to enrich the concept hierarchy after the initial hierarchy is explicitly stated
   - The user will in reach the concept hierarchy by saying {scanner, telephone} $\subset$ {peripherals}, {laptop, desktop, workstation} $\subset$ {computer} after giving explicit concept hierarchy as {peripherals, computer} $\subset$ {electronics} and {machine } $\subset$ {electronics}

Discretization and concept hierarchy generation for categorical data

3. Specification of a set of attributes, but not of their partial ordering
   - Concept hierarchy can be automatically generated based on the number of distinct values per attribute in the given attribute set.
   - The attribute with the most distinct values is placed at the lowest level of the hierarchy.



Discretization and concept hierarchy generation for categorical data

4. Specification of only a partial set of attributes
   - User will provide some of the concept hierarchies and system will fill the remaining hierarchical levels by analyzing the semantics of the attributes of the data set

Chapter-3
Linked list Data Structure

3.1 Introduction

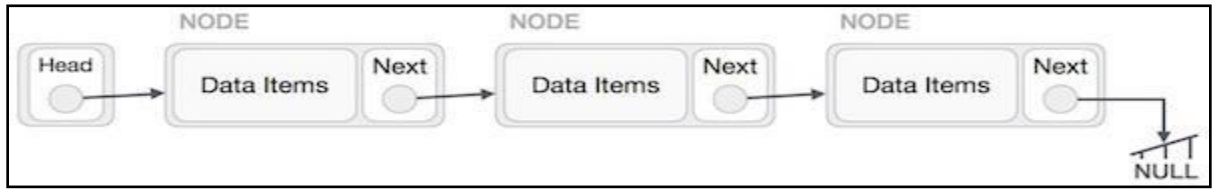
- **Linked list** is one of the fundamental data structures, and can be used to implement other data structures.
- In a linked list there are different numbers of nodes.
- Each node is consists of two fields.
  - The first field holds the value or data and the second field holds the reference to the next node or null if the linked list is empty
- **malloc()** is used to dynamically allocate a single block of memory in C++.
- **sizeof()** is used to determine size in bytes of an element in C++.

Types of LinkedList

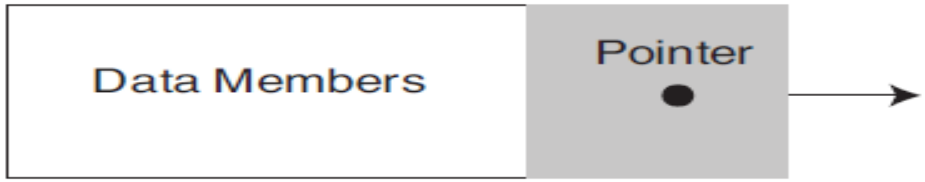
- **Single linked list:-** navigation is forward only.
  - It is made up of nodes that consists of two parts,i.e,data and link
- **Doubly linked list:-** Forward and backward navigation is possible.
- **Circular linked list :-**Last element is linked with the first-element element.

3.2 Linked List

- Dynamically allocated data structures may be linked together in memory to form a chain.
  - A linked list is a series of connected nodes where each node is a data structure .
  - A linked list can grow or shrink as program runs.
  - This is possible because the nodes in a linked list are dynamically allocated.



- A linked list is called linked because each node in the series has a pointer that points to the next node in the list.
  - This creates a chain where the first node points to the second node, the second node points to the third node, and so on.



- *List head:* simply points to the first node in the list.
  - The last one in the list, it points to the NULL

Advantages of Linked Lists over Arrays

- **A linked list can easily grow or shrink in size.**
  - In fact, the programmer doesn't need to know how many nodes will be in the list.
  - They are simply created in memory as they are needed.
  - Arrays are *simple* and *fast but we m*ust specify their size at construction time.
- **The Composition of a Linked List.**
  - Each node in a linked list contains one or more members that represent data.
  - In addition to the data, each node contains a pointer, which can point to another node.

Create the Node structed

Node: declaration

self-referential data structure is a structure which contains a pointer to the structure of the same object.

```
struct ListNode
{
        double value;// the value or stored in the node
        ListNode *next;//A reference to the next node,null for the last node
};
```

- The ListNode structure contains a pointer to an object of the same type as that being declared, it is known as a *self-referential data structure*.
- This structure makes it possible to create nodes that point to other nodes of the same type.

Linked List Declaration

- The First member of the ListNode structure is a double named value.
  - It will be used to hold the nodes data.
  - The second member is a pointer named next .
- The pointer can hold the address of any object that is a ListNode structure.
- This allows each ListNode structure to point to the next ListNode structure in the list.

Example Create Constant Node

```
#include<iostream>
#include <cstdlib>
using namespace std;
struct Node
{
  int data;
  struct Node *next;
};
int main()
{
  struct Node* head = NULL;
  struct Node* second = NULL;
  struct Node* third = NULL;
  // allocate 3 nodes in the heap
  head = (struct Node*)malloc(sizeof(struct Node));
  second = (struct Node*)malloc(sizeof(struct Node));
  third = (struct Node*)malloc(sizeof(struct Node));
```

```
head->data = 1; //assign data in first node
  head->next = second; // Link first node with second

  second->data = 2; //assign data to second node
  second->next = third;

  third->data = 3; //assign data to third node
  third->next = NULL;
```

Appending a node/Add a node

- To append a node to a linked list means to add the node to the end of the list.
- *The steps to add a new node as follow*
- *Algorithm :-*

Create a new node.

Store data in the new node.

If there are no nodes in the list(It's empty)

Make the new node the first node.(make head pointers to the new node)

Else

Traverse the list to find the last node.

Add the new node to the end of the list.

End If.

Linked List Operations

- **The basic linked list operations are :**
  - **Appending a node**
  - **Traversing the list**
  - **Inserting a node**
  - **Deleting a node, and**
  - **Destroying the list.**

appendNode: find last element

- How to find the last node in the list?
- Algorithm:
  - Make a pointer p point to the first element.
  - while (the node p points to) is not pointing to NULL or make p point to next.

```
ListNode *p = head;
while (p->next)
p = p->next;
```

Append node (code)

```
void appendNode(double num)
{
ListNode *newNode; // To point to a new node
ListNode *nodePtr; // To move through the list
 newNode = new ListNode;
 newNode->value = num;
newNode->next = NULL;
 if (!head) //No nodes
        head = newNode;
else // Otherwise, insert newNode at end.
 {
        nodePtr = head;
        while (nodePtr->next) // Find the last node in the list.
                nodePtr = nodePtr->next;
        nodePtr->next = newNode; // Insert newNode as the last node.
}
}
```

Traversing a Linked List

- **Traversing a single linked list** means visiting each node of a single linked list until the end node is reached.
  - A **while loop** that traverses, or travels through the linked list.
- The following pseudocode represents the algorithm

Assign List head to node pointer.

While node pointer is not NULL

Display the value member of the node pointed to by node pointer.

Assign node pointer to its own next member.

End While.

Class work: Write a program to count the number of nodes in the linked list?

Displaying the list of nodes

1. Set a temporary pointer to point to the same thing as the start pointer.
2. If the pointer points to NULL, display the message "End of list" and stop.
3. Otherwise, display the details of the node pointed to by the start pointer.
4. Make the temporary pointer point to the same thing as the **nxt** pointer of the node it is currently indicating.
5. Jump back to step 2.

```
temp = start_ptr;
do
{
if (temp == NULL)
   cout << "End of list" << endl;
else
{  // Display details for what temp points to
 cout << "Name : " << temp->name << endl;
 cout << "Age : " << temp->age << endl;
 cout << "Height : " << temp->height << endl;
  cout << endl;       // Blank line
    // Move to next node (if present)
  temp = temp->nxt;
}
 }
while (temp != NULL);
```

To display node value

```
nodePtr=head;
while(nodePtr)
{
        cout<<nodePtr->value<<endl;
        nodePtr=nodePtr->next;
}
```

Inserting a node

- Inserting a node in the middle of a list is complex than appending a node.
  - For example, suppose the values in a list are sorted and you wish all new values to be inserted in their proper position.

```
Create a new node.
Store data in the new node.
If there are no nodes in the list
Make the new node the first node.
Else
Find the first node whose value is greater than or equal to the new
value, or the end of the list (whichever is first).
Insert the new node before the found node, or at the end of the list
if no such node was found.
End If.
```

Code for insert node

```
ListNode *newNode; // A new node
ListNode *nodePtr; // To traverse the list
ListNode *previousNode = NULL; // The previous node
// Allocate a new node and store num there.
newNode = new ListNode;newNode->value = num;
// If there are no nodes in the list
// make ne
if (!head)
 {
head = newNode;
newNode->next = NULL;
 }
```

Insert node

```
else // Otherwise, insert newNode
 {
// Position nodePtr at the head of list.
 nodePtr = head;
 // Initialize previousNode to NULL.
 previousNod
// Skip all nodes whose value is less than num.
 while (nodePtr != NULL && nodePtr->value < num)
 {
 previousNode = nodePtr;
  nodePtr = nodePtr->next;
 }
```

…continue insert node

```
// If the new node is to be the 1st in the list,
// insert it before all other nodes.
if (previousNode == NULL)
   {
head = newNode;
newNode->next = nodePtr;
}
else // Otherwise insert after the previous node.
{
 previousNode->next = newNode;
 newNode->next = nodePtr;
 }
 }
}
```

Deleting a node from the linked list

- Deleting a node from a linked list requires two steps:
  - Remove the node from the list without breaking the links created by the next pointers.
  - Delete the node from memory.
- The delete Node member function searches for a node containing a particular value and deletes it from the list
  - Two node pointers requires, namely, **nodePtr and previousNode**, are used to traverse the list.
  - **previousNode** always points to the node whose position is just before the one pointed to by nodePtr.
  - When **nodePtr** points to the node that is to be deleted,
- Change the pointer of the previous node to point to the node after the one to be deleted.
  - **previousNode->next is made to point to nodePtr->next.**

- Delete the node containing num
  - use nodePtr to traverse the list, until it points to num or NULL
    - as **nodePtr** is advancing, make **previousNode** point to the node before it

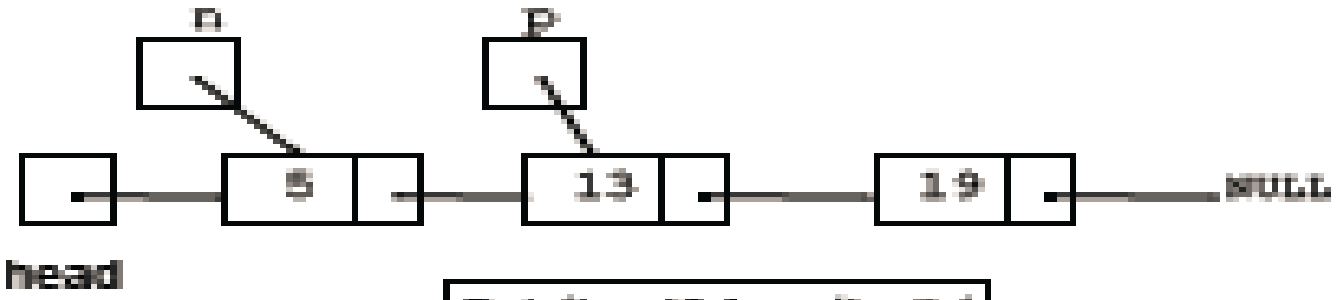
if (**nodePtr** is not NULL) //found!

if (**nodePtr** ==head) //it's the first node, and make head point to the second element
delete head node (the first node)

else

make **previousNode** node point to what nodePtr node points to delete nodePtr node

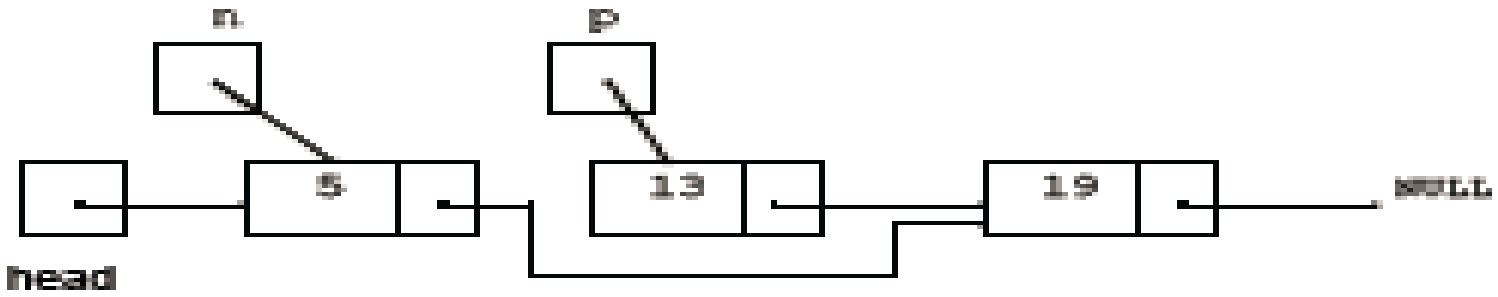
else: . . . p is NULL, not found do nothing



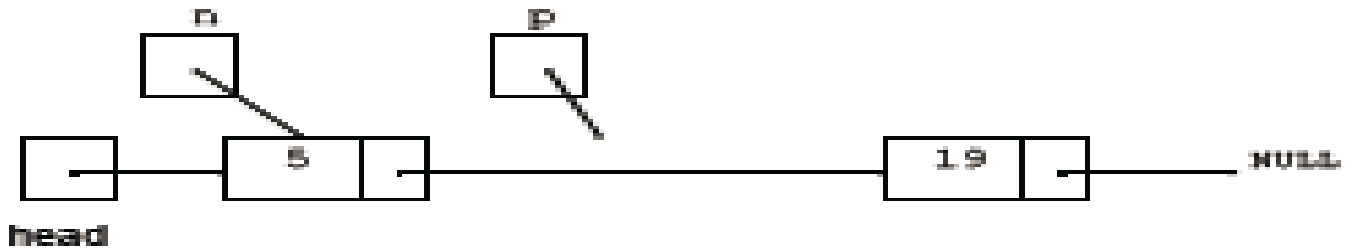
Deleting 13 from the list

16

```
n->next = p->next;
```




```
delete p;
```



Deleting a node (code)

```
void deleteNode(double num)
 {
        ListNode *nodePtr; // To traverse the list
        ListNode *previousNode; // To point to the previous node
        if (!head) // If the list is empty, do nothing
                return;
        if (head->value == num) // Determine if the first node is the one.
        {
                nodePtr = head->next;
                delete head;
                head = nodePtr;
        }
```

Deleting a node (code) II

```
    else
    {
            nodePtr = head; // Initialize nodePtr to head of list
            // Skip all nodes whose value member is not equal to num.
            while (nodePtr != NULL && nodePtr->value != num)
            {
                    previousNode = nodePtr;
                    nodePtr = nodePtr->next;
            }
            // If nodePtr is not at the end of the list,
            // link the previous node to the node after
            // nodePtr, then delete nodePtr.
            if (nodePtr)
            {
                    previousNode->next = nodePtr->next;
                    delete nodePtr;
            }
    }
}
```

Destroyng the list

```
Void deletelist()
{
        ListNode *nodePtr; // To traverse the list
        ListNode *nextNode; // To point to the next node
        nodePtr = head; // Position nodePtr at the head of the list.
        while (nodePtr != NULL) // While nodePtr is not at the end of the list...
        {
        // Save a pointer to the next node.
                nextNode = nodePtr->next;
        // Delete the current node.
                delete nodePtr;
        // Position nodePtr at the next node.
                nodePtr = nextNode;
        }
}
```

Doubly LinkedList

- each node has two pointers,
  - One to the next node (next) and one to the previous node (prev)
  - Head points to first element, tail points to last.
  - Can traverse list in reverse direction by starting at the tail and using p=p->prev

4. Stacks
5. Queue

4. Stack

- **Stack: A simple data structure, in which insertion and deletion occur at the same end, is termed (called) a stack.**
  - **a LIFO (last in, first out) data structure.**
- **Examples:**
  - **plates in a cafeteria serving area**
  - **return addresses for function calls**

Stack Operations and Functions

Operations:

- push: add a value at the top of the stack
- pop: remove a value from the top of the stack

Functions:

- isEmpty: true if the stack currently contains no elements
- isFull: true if the stack is full; only useful for static stacks

Implementation:

- **The Basic Operations:**

Push()

```
{
   if there is room {
   put an item on the top of the stack
   else
   give an error message
   }
}
```

Pop()

```
{
        if stack not empty {
                return the value of the top item
                remove the top item from the stack
                                }
        else {
                give an error message
                }
}
```

Array Implementation of Stacks: The PUSH operation

- **Step-1:** Increment the Stack TOP by 1.
  - Check whether it is always less than the Upper Limit of the stack.
  - If it is less than the Upper Limit go to step-2 else report -"Stack Overflow“
- **Step-2:** Put the new element at the position pointed by the TOP

Static Stack Implementation

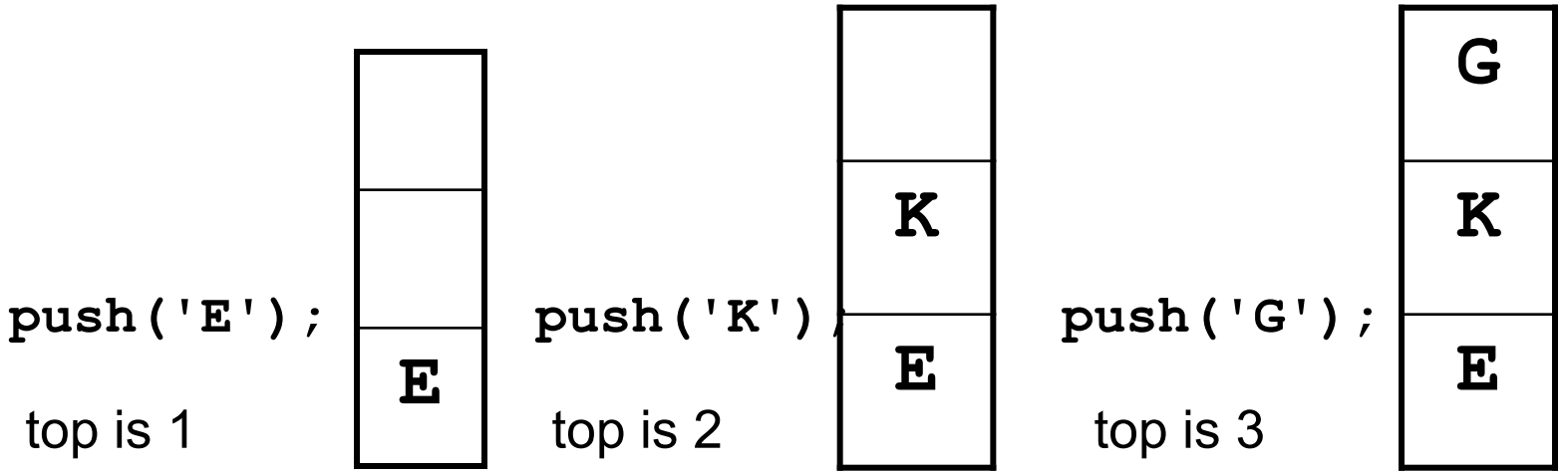
- Uses an array of a fixed size
- Bottom of stack is at index 0. A variable called top tracks the current top of the stack

```
const int STACK_SIZE = 3;
char s[STACK_SIZE];
int top = 0;
```

top is where the next item will be added

Array Implementation Example

This stack has max capacity 3, initially top = 0 and stack is empty.



Array Implementation Example

```
char s[STACK_SIZE];
 int top=0;
```

To check if stack is full:

```
 bool isFull()
 {
   if (top ==
 STACK_SIZE)
          return true;
   else return false;
 }
```

Array Implementation

To add an item to the stack

```
void push(char x)
 {
  if (isFull())
     {error(); exit(1);}
  // or could throw an exception
  s[top] = x;
  top++;
}
```

Array Implementation

To remove an item from the stack

```
void pop(char &x)
 {
  if (isEmpty())
     {error(); exit(1);}
  // or could throw an exception
 top--;
  x = s[top];
}
```

Array Implementation of Stacks: the POP operation

- **Step-1:** If the Stack is empty then give the alert "Stack underflow" and quit; or else go to step-2
  Step-2: a) Hold the value for the element pointed by the TOP
  b) Put a NULL value instead
  c) Decrement the TOP by 1

To remove an item from the stack

```
void pop(char &x)
 {
  if (isEmpty())
    {error(); exit(1);}
  // or could throw an exception
 top--;
  x = s[top];
}
```

Dynamic Stacks

- Implemented as a linked list
- Can grow and shrink as necessary
- Can't ever be full as long as memory is available

Linked List Implementation of Stacks: the PUSH operation

- In Step [1] we create the new element to be pushed to the Stack
- In Step [2] the TOP most element is made to point to our newly created element.
- In Step [3] the TOP is moved and made to point to the last element in the stack, which is our newly added element.

Algorithm

- **Step-1:** If the Stack is empty go to step-2 or else go to step-3
- **Step-2:** Create the new element and make your "stack" and "top" pointers point to it and quit.
- **Step-3:** Create the new element and make the last (top most) element of the stack to point to it
- **Step-4:** Make that new element your TOP most element by making the "top" pointer point to it.

Linked List Implementation

- Node for the linked list

```
struct LNode
{
 char value;
 LNode *next;
 LNode(char ch, LNode *p = 0)
  { value = ch; next = p;}
};
```

- Pointer to beginning of linked list, which will serve as top of stack

```
LNode *top = NULL;
```

- **Implementation:**

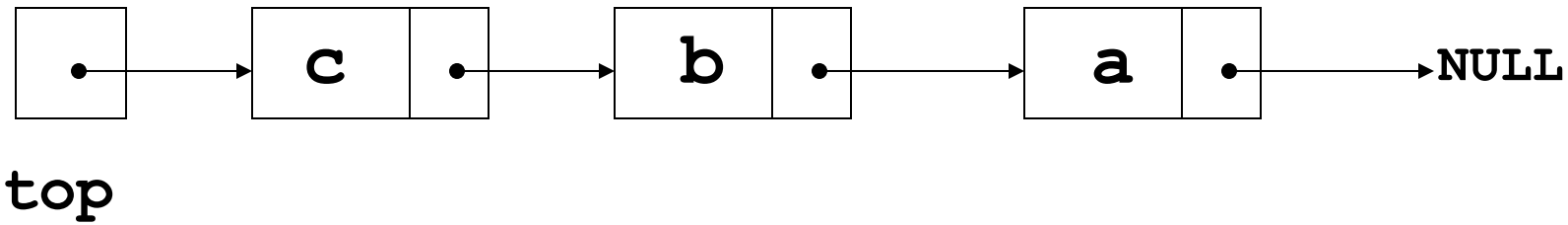
```
struct node{
     int item;
     struct node *next;
    }
struct node *stack = NULL; /*stack is initially empty*/
struct node *top = stack;
main()
{
..
..
push(item);
..
}
```

-
```
push(int item)
{
     if(stack == NULL)  /*step-1*/
        {
        newnode = new node  /*step-2*/
        newnode -> item = item;
        newnode -> next = NULL;
        stack = newnode;
        top = stack;
        }
     else
        {
        newnode = new node; /*step-3*/
        newnode -> item = item;
        newnode -> next = NULL;
        top ->next = newnode;
        top = newnode;   /*step-4*/
        }
}
```

Linked List Implementation

A linked stack after three push operations:

```
push('a'); push('b'); push('c');
```



Operations on a Linked Stack

Check if stack is empty:

```
bool isEmpty()
{
  if (top == NULL)
    return true;
  else
    return false;
}
```

Operations on a Linked Stack

Add a new item to the stack

```
void push(char x)
{
   top = new LNode(x, top);
}
```

Applications of Stacks in data structure

- Expression Handling:-
  - Infix to Postfix(455*+) or Infix to Prefix Conversion (+4*55)
  - Postfix or Prefix Evaluation
- Backtracking Procedure
- Function Calls

Algorithm

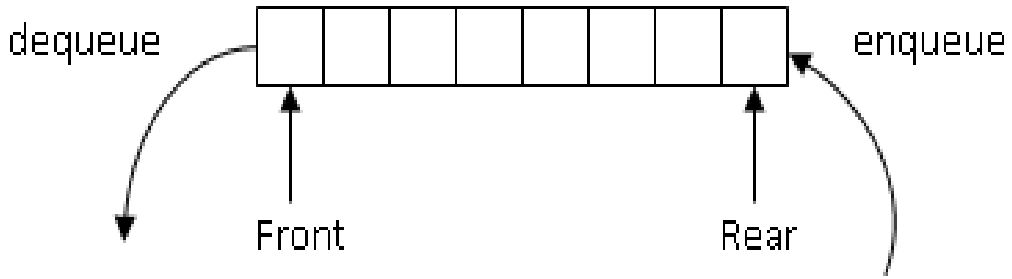
```
initialize stack to empty;
while (not end of postfix expression)
{
get next postfix item;
if(item is value)
 push it onto the stack;
else if(item is binary operator)
{
pop the stack to x;
pop the stack to y;
perform y operator x;
push the results onto the stack;
 }
else if (item is unary operator)
 {
pop the stack to x;
perform operator(x);
 push the results onto the stack
}}
```

Example

So for **6 5 2 3 + 8 * + 3 + ***
the first item is a value (6) so it is pushed onto the stack
the next item is a value (5) so it is pushed onto the stack
the next item is a value (2) so it is pushed onto the stack
the next item is a value (3) so it is pushed onto the stack and the stack becomes

Chapter 5
Queue

- a data structure that has access to its data at the front and rear.
- operates on FIFO (Fast In First Out) basis.
- uses two pointers/indices to keep tack of information/data.
- has two basic operations:
  - **enqueue -** inserting data at the rear of the queue
  - **dequeue –** removing data at the front of the queue

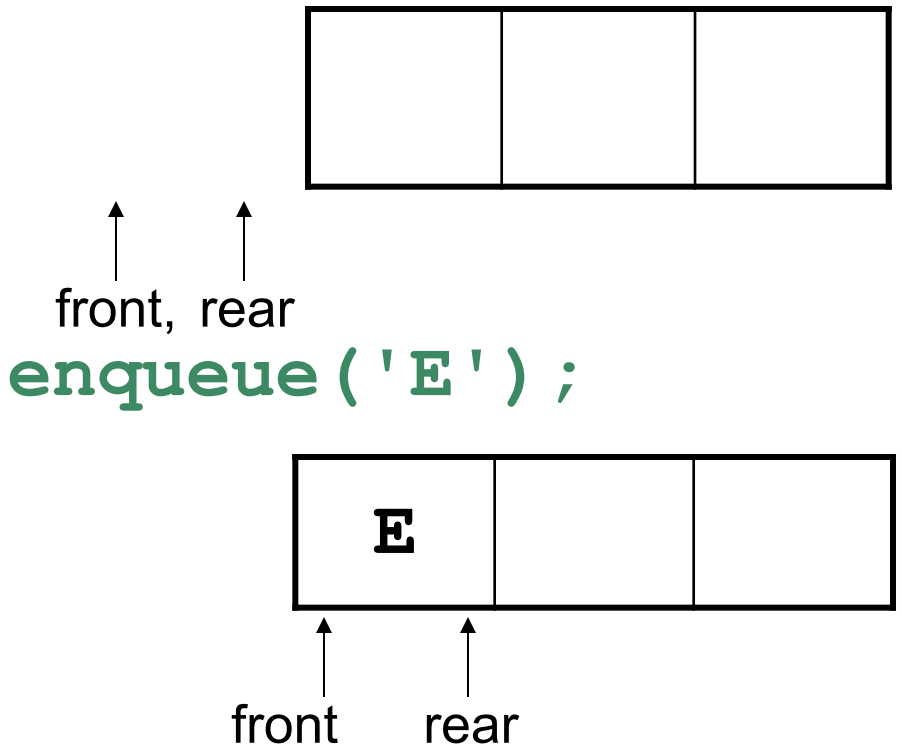


Queue Locations and Operations

- **rear:** position where elements are added
- **front:** position from which elements are removed
- **enqueue:** add an element to the rear of the queue
- **dequeue:** remove an element from the front of a queue

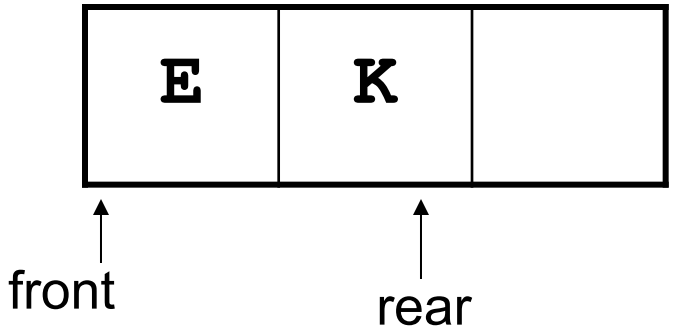
Array Implementation of Queue

An empty queue that can hold **`char`** values:

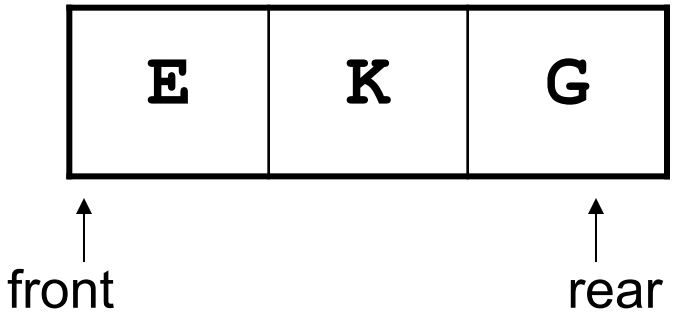


Queue Operations - Example

```
enqueue('K');
```

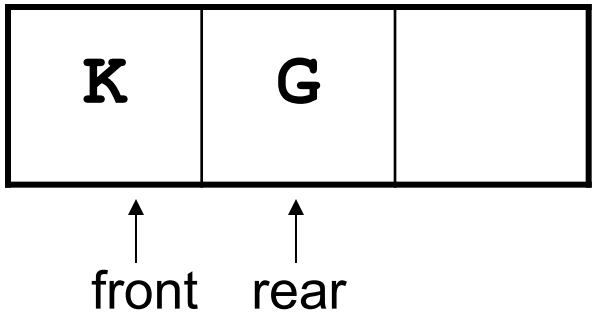



```
enqueue('G');
```

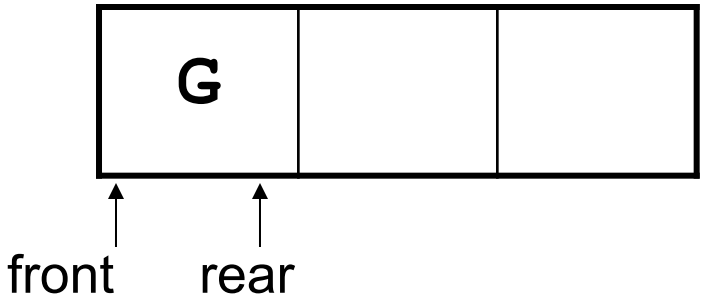


Queue Operations - Example

```
dequeue(); // remove E
```




```
dequeue(); // remove K
```



Array Implementation Issues

- In the preceding example, Front never moves.
- Whenever **`dequeue`** is called, all remaining queue entries move up one position. This takes time.
- Alternate approach:
  - Circular array: `front` and `rear` both move when items are added and removed. Both can 'wrap around' from the end of the array to the front if warranted.
- Other conventions are possible

Array Implementation Issues

- Variables needed
  - `const int QSIZE = 100;`
  - `char q[QSIZE];`
  - `int front = -1;`
  - `int rear = -1;`
  - `int number = 0; //how many in queue`
- Could make these members of a queue class, and queue operations would be member functions

`isEmpty` Member Function

Check if queue is empty

```
bool isEmpty()
{
  if (number > 0)
    return false;
  else
    return true;
}
```

`isFull` Member Function

Check if queue is full

```
bool isFull()
{
  if (number < QSIZE)
    return false;
  else
    return true;
}
```

enqueue and **dequeue**

- To enqueue, we need to add an item **x** to the rear of the queue
- Queue convention says **q[rear]** is already occupied. Execute

```
if(!isFull)
{ rear = (rear + 1) % QSIZE;
// mod operator for wrap-around
  q[rear] = x;
   number ++;
}
```

enqueue and **dequeue**

- To dequeue, we need to remove an item **x** from the front of the queue
- Queue convention says **q[front]** has already been removed. Execute

```
if(!isEmpty)
{   front = (front + 1) % QSIZE;
    x = q[front];
    number--;
}
```

`enqueue` and `dequeue`

- `enqueue` moves `rear` to the right as it fills positions in the array
- `dequeue` moves `front` to the right as it empties positions in the array
- When `enqueue` gets to the end, it wraps around to the beginning to use those positions that have been emptied
- When `dequeue` gets to the end, it wraps around to the beginning use those positions that have been filled

enqueue and **dequeue**

- Enqueue wraps around by executing

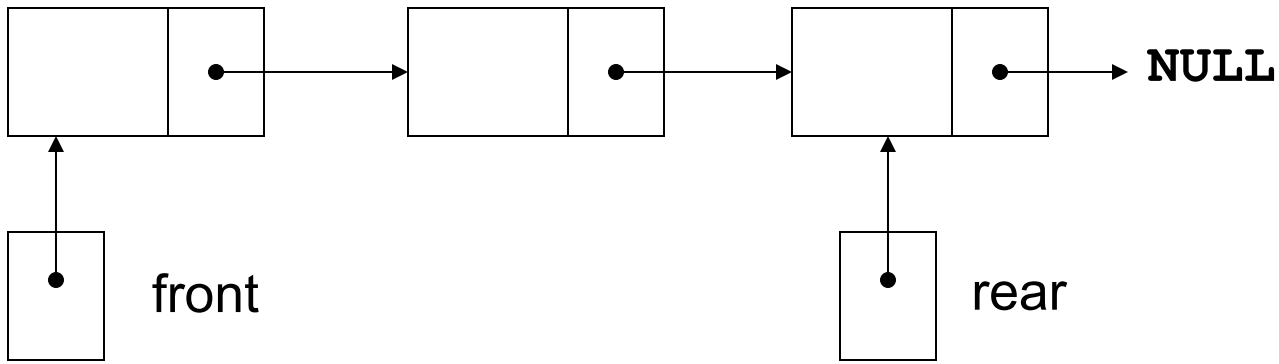
```
rear = (rear + 1) % QSIZE;
```

- Dequeue wraps around by executing

```
front = (front + 1) % QSIZE;
```

18.5 Dynamic Queues

- Like a stack, a queue can be implemented using a linked list
- Allows dynamic sizing, avoids issue of wrapping indices



Dynamic Queue Implementation Data Structures

```
struct QNode
{
    char value;
    QNode *next;
    QNode(char ch, QNode *p = NULL);
    {value = ch; next = p;}
}
QNode *front = NULL;
QNode *rear = NULL;
```

isEmpty Member Function

To check if queue is empty:

```
bool isEmpty()
{
   if (front == NULL)
      return true;
   else
      return false;
}
```

enqueue Member Function

To add item at rear of queue

```
void enqueue(char x)
{
  if (isEmpty())
   { rear = new QNode(x);
     front = rear;
     return;
   }
  rear->next = new QNode(x);
  rear = rear->next;
}
```

dequeue Member Function

To remove item from front of queue

```
void dequeue(char &x)
{
  if (isEmpty())
   { error(); exit(1);
   }
  x = front->value;
  QNode *oldfront = front;
  front = front->next;
  delete oldfront;
}
```

18.6 The STL **deque** and **queue** Containers

- **deque**: a double-ended queue. Has member functions to enqueue (**push_back**) and dequeue (**pop_front**)

- **queue**: container ADT that can be used to provide a queue based on a **vector**, **list**, or **deque**. Has member functions to enqueue (**push**) and dequeue (**pop**)

Defining a Queue

- Defining a queue of **`char`**, named cQueue, based on a **`deque`**:

```
deque<char> cQueue;
```

- Defining a **`queue`** with the default base container

```
queue<char> cQueue;
```

- Defining a queue based on a **`list`**:

```
queue<char, list<char> > cQueue;
```

- Spaces are required between consecutive **`> >`** symbols to distinguish from stream extraction

18.7 Eliminating Recursion

- Recursive solutions to problems are often elegant but inefficient
- A solution that does not use recursion is more efficient for larger sizes of inputs
- Eliminating the recursion: re-writing a recursive algorithm so that it uses other programming constructs (stacks, loops) rather than recursive calls