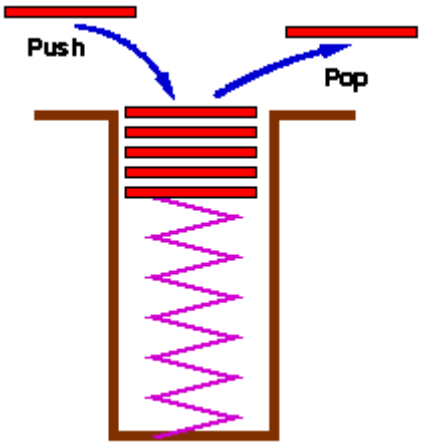
UNESCO-NIGERIA TECHNICAL & VOCATIONAL
EDUCATION REVITALISATION PROJECT-PHASE II

NATIONAL DIPLOMA IN

COMPUTER SCIENCE

DATA STRUCTURES AND ALGORITHMS

COURSE CODE: COM 124



YEAR I- SE MESTER 2

THEORY

Course contents

Table of Contents

Week 1 :

Concept of Data structures

This week Learning outcomes:

• Define data structure.
• Define data attributes: name, value range, data types.
• Define units for identifying data character, fields, sub fields , records, files.

Data Structure

The logical or mathematical model of a particular organization of data is called its data structures. A data item is a single unit of values. It is a raw fact which becomes information after processing . Data items for example , date are called group items if they can be divided into subsystems. The date for instance is represented by the day, the month andumber is called an elementary item, because it can not be sub-divided into sud-items. It is indeed treated as a single item. An entity is used to describe anything that has certain attributes or propreties, which may be assigned values. For example , the following are possible attributes and their corresponding values for an entity known as STUDENT.

ATTRIBUTES	NAME	AGE	SEX	MATRIC NO
VALUES	Paul	21	Male	800654

Entities with similar attributes for example, all the 200 level Computer science & Statistics students form an entity set.

Main functions of data Structures:

- Seek to identify and develop entities, operations and appropriate classes of problems to use them.
- Determine representations for abstract entities to implement abstract operations on concrete representations.

Algorithm. A finite sequence of instructions, each of which has a clear meaning and can be executed with a finite amount of effort in finite time.
whatever the input values, an algorithm will definitely terminate after executing a finite number of instructions.

Charasteristics of algorithm:

- Has a finite set of steps with definite instructions.
- Instructions have definite order.
- Algorithm must eventually stop.
- Actions are deterministic.

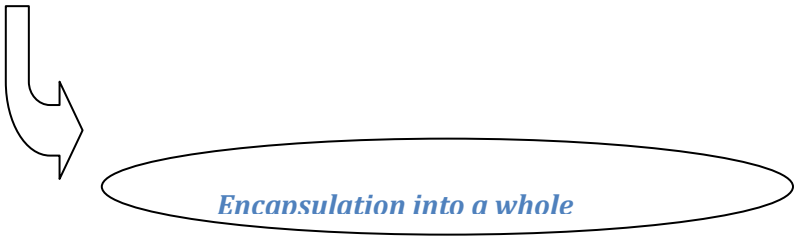
Data:

Some abstraction simplification of reality.

- Abstract Data Structure is a conceptual organization without regard to how data is organized on the machine.

Abstract Data Type:

- Abstract Data structure.
- Plus operations to perform on it.



What determines the nature of the abstraction?

- Kind of problem to solve.
- Operations to be performed.
- Machine restrictions.

Language Independent ADT Specification

1. Syntactic Specification. (Form)
2. Semantic Specification. (Meaning)
3. Restrictions.

Data Declarations:

Causes storage space to be reserved for each variable.

1. Associate identifier name with storage to allow access.

2. Contents of storage are interpreted according to language data types.
3. More efficient use of storage.
4. Better storage of management.
5. Static Type Checking.

Fundamental Data Structure

Bit – 0,1

Boolean – compare

Assignment

1 byte word – 2^8 of info to manipulate.

4 byte word – 2^{32} of info to manipulate.

Boolean – 2 Values

NewBoolean(ident)

MakeTrue(ident) ident=**true**

MakeFalse(ident) ident=**false**

IsTrue(ident) ident=**true**

IsFalse(ident) ident=**false**

Assign(I_1,I_2) I_1=I_2

And(I_1,I_2) I_1 && I_2

Or(I_1,I_2) I_1 || I_2

Not(I_1) ! I_1

char

NewChar(Id)

Assign(I_1,I_2)

AreEqual(I_1,I_2)

Encode(Id)

Decode(Id)

Precedes(I_1,I_2) $I_1<I_2$

Delete(Id)

int

word

½ word ⟶ Saves storage space!

short

int

long

MazSize()

Create(Id)

Delete(Id)

Assign(I_1,I_2)

Equality(I_1,I_2)

IsLessThan(I_1,I_2)

Negative(Id)

Sum(I_1,I_2)

Difference(I_1,I_2)

Quotient(I_1,I_2) ↘ Float!

Mod(I_1,I_2) ↗

Reals: **float**

Value range

All possible values that could be assigned to a given atttibute of an entity set is called the range of values of the attribute.

Data types

In mathematics it is customary to classify variables according to certain important characteristics. Clear distinctions are made between real, complex, and logical variables or between variables representing individual values, or sets of values, or sets of sets, or between functions, functionals, sets of functions, and so on. This notion of classification is equally if not more important in data processing. We will adhere to the principle that every constant, variable, expression, or function is of a certain *type*. This type essentially characterizes the set of values to which a constant belongs, or which can be assumed by a variable or expression, or which can be generated by a function .

Therefore, a data type is a set of values together with the operations defined on the values: {(values) (operations)}. The operations are performed on the values defined. E.g integer (-4,-1,1,3,4) are values while(+,-,*,/) are operations. Data types also allow us to associate meaning to sequence of bits in the computer memory. Eg string AA@, integer 5 etc .

Data types	operations	storage representation
1. Integer	*,+,-,/	2's complement, sign magnitude
2. Real	""""""""	
3. Boolean	AND,OR,NOT	True=0,False=1
4. Character		8 bits ASCII/EBCDIC
		length followed by sequence of characters
5. String	Concatenation.	Length, substring, eg for AABC@ we can have: /3/A/B/C/
Pattern matching	sequence of characters terminated by special symbols.	
6. Pointers	Value of "----->"	As for integers

DATA TYPES can be classified as :

Primitive (Also called built in) eg Integers, real, pointers,

booleanY They are native or local to the language. The language knows their structure. They are part of the original design .

Standard Primitive Types

Standard primitive types are those types that are available on most computers as built-in features. They include the whole numbers, the logical truth values, and a set of printable characters. On many computers fractional numbers are also incorporated, together with the standard arithmetic operations. We denote these types by the identifiers
INTEGER, REAL, BOOLEAN, CHAR, SET

Integer types

The type INTEGER comprises a subset of the whole numbers whose size may vary among individualcomputer systems.

The type REAL

The type REAL denotes a subset of the real numbers. Whereas arithmetic with operands of the types INTEGER is assumed to yield exact results, arithmetic on values of type REAL is permitted to be inaccurate within the limits of round-off errors caused by computation on a finite number of digits. This is the principal reason for the explicit distinction between the types INTEGER and REAL, as it is made in most programming languages.
The standard operators are the four basic arithmetic operations of addition (+), subtraction (-), multiplication (*), and division (/). It is an essence of data typing that different types are incompatible under assignment. An exception to this rule is made for assignment of integer values to real variables, because here the semanitcs are unambiguous. After all, integers form a subset of real numbers.

The type CHAR

The standard type CHAR comprises a set of printable characters. Unfortunately, there is no generally accepted standard character set used on all computer systems. Therefore, the use of the predicate "standard" may in this case be almost misleading; it is to be understood in the sense of "standard on the computer system on which a certain program is to be executed."
The character set defined by the International Standards Organization (ISO), and particularly its American version ASCII (American Standard Code for Information Interchange) is the most widely accepted set. The ASCII set is therefore tabulated in Appendix A. It consists of 95 printable (graphic) characters and 33 control characters, the latter mainly being used in data transmission and for the control of printing equipment.
In order to be able to design algorithms involving characters (i.e., values of type CHAR) that are system independent, we should like to be able to assume certain minimal properties of character sets, namely:

1. The type CHAR contains the 26 capital Latin letters, the 26 lower-case letters, the 10 decimal digits, and a number of other graphic characters, such as punctuation marks.
2. The subsets of letters and digits are ordered and contiguous
3. The type CHAR contains a non-printing, blank character and a line-end character that may be used as separators.

The type SET

The type SET denotes sets whose elements are integers in the range 0 to a small number, typically 31 or 63.
Given, for example, variables
VAR r, s, t: SET
possible assignments are: r := {5}; s := {x, y .. z}; t := {}

PURPOSE OF TYPE INFORMATION:

Type information has 4 purposes:

a) It allows us to associate meaning to sequence of bits in the computer memory eg string AA@ , integer 5YThis is because all data and instructions are store in the same manner as sequence of bits.

b) It is useful during program development to improve readability and debugging.

c) It helps simplify implementation, eg it is easier and more efficient for implementations to allocate storage for integers only, rather that arbitrary value

d) It allows checking for compatibility between operation and

operands before execution e.g in the scope of the PASCAL declaration VAR X:integer, the expression ANOT X@ would be an error because the type of operand "X" is not compatible with the type expected by the boolean operator .NOT.

A language is said to be STATIC type checking when it

requires type declaration that allow language translator to check data during translation.

A DYNAMIC type checking is one that checks for data type during program execution.

Some languages LISP,SNOBOLY use no declaration . This simplifies programming in these languages and allows great flexibility in creating and manipulating data structure but the cost is paid in Dynamic checking , less efficient data representation, and more complex storage management, all on which slow program execution.

FORTRAN, COBOL PASCAL require extensive declaration for all data structure and also introduce many related restrictions on the manner in which data may be created, destroyed and modified. These requirements make programming considerably more complex but program execution speed is greatly enhanced. A central problem in PL design is to find the proper balance between added execution efficiency obtainable through explicit data declaration and the added flexibility possible without them.

OPERATIONS: there are of 2 types:

Operations on the programmer defined data e.g add, subtract,*,/

Operations on system-defined data eg GOTO..

Subprogram Calls, naming of data structure, parameter transmissionY Operations on programmer defined data may be further subdivided into:

- Primitive ie operation built into the language

-Programmer-defined operations eg subprograms

Type checking:

The main objective of type checking is to determine before program execution whether a domain incompatibility can occur. If so, error messages occur for coercion or execution time testing may be generated.

Type checking can be performed statically (before execution or during compilation) or dynamically (at execution time)

For example, static type checking occur in most languages such as FORTRAN, COBOL, PASCALY

Dynamic type checking occurs in APLY

Static type checking has following advantages over dynamic type checking.

Advantages of Type checking:

a.) Efficiency: since the program is typically executed many times but needs only be type-checked once.

b.) Furthermore type information may used by implementation to improve efficiency in many other ways.

c.) Minor programming errors could be detected before actual execution; this will simplify program testing and debugging eg when another operation is to be performed on non numerical data.

d.) Type specification also improves programme readability by making explicit the data representation used by the programmer.

Disadvantages:

- Syntax of a type language are usually more complex Inflexibility: restriction are imposed on the programmer's freedom of expression.

In conclusion, static type checking should be used in languages to prevent domain incompatibility whenever the disadvantages outweigh its benefit.

Primitive Data Types

A new, primitive type is definable by enumerating the distinct values belonging to it. Such a type is called an *enumeration type*. Its definition has the form

TYPE T = (c1, c2, ... , cn)
T is the new type identifier, and the ci are the new constant identifiers.

Examples

TYPE shape = (rectangle, square, ellipse, circle)
TYPE color = (red, yellow, green)
TYPE sex = (male, female)
TYPE weekday = (Monday, Tuesday, Wednesday, Thursday, Friday,
Saturday, Sunday)
TYPE currency = (franc, mark, pound, dollar, shilling, lira, guilder,
krone, ruble, cruzeiro, yen)
TYPE destination = (hell, purgatory, heaven)
TYPE vehicle = (train, bus, automobile, boat, airplane)
TYPE rank = (private, corporal, sergeant, lieutenant, captain, major, colonel, general)
TYPE object = (constant, type, variable, procedure, module)
TYPE structure = (array, record, set, sequence)
TYPE condition = (manual, unloaded, parity, skew)

The definition of such types introduces not only a new type identifier, but at the same time the set ofidentifiers denoting the values of the new type. These identifiers may then be used as constants throughout
the program, and they enhance its understandability considerably. If, as an example, we introduce variables
s, d, r, and b.

VAR s: sex
VAR d: weekday
VAR r: rank

then the following assignment statements are possible:

s := male
d := Sunday
r := major
b := TRUE

Evidently, they are considerably more informative than their counterparts
s := 1 d := 7 r := 6 b := 2 which are based on the assumption that c, d, r, and b are defined as integers and that the constants are mapped onto the natural numbers in the order of their enumeration.

The Record Structure

The most general method to obtain structured types is to join elements of arbitrary types, that are possibly themselves structured types, into a compound. Examples from mathematics are complex numbers, composed of two real numbers, and coordinates of points, composed of two or more numbers according to the dimensionality of the space spanned by the coordinate system. An example from data processing is describing people by a few relevant characteristics, such as their first and last names, their date of birth, sex, and marital status.

In mathematics such a compound type is the Cartesian product of its constituent types. This stems from the fact that the set of values defined by this compound type consists of all possible combinations of values, taken one from each set defined by each constituent type. Thus, the number of such combinations, also called *n-tuples*, is the product of the number of elements in

each constituent set, that is, the cardinality of the compound type is the product of the cardinalities of the constituent types.
In data processing, composite types, such as descriptions of persons or objects, usually occur in files or data banks and record the relevant characteristics of a person or object. The word record has therefore become widely accepted to describe a compound of data of this nature, and we adopt this nomenclature in preference to the term Cartesian product. In general, a record type T with components of the types T1, T2,... , Tn is defined as follows:
TYPE T = RECORD s1: T1; s2: T2; ... sn: Tn END
card(T) = card(T1) * card(T2) * ... * card(Tn)
Examples
TYPE Complex = RECORD re, im: REAL END
TYPE Date = RECORD day, month, year: INTEGER END
TYPE Person = RECORD name, firstname: Name;
birthdate: Date;
sex: (male, female);
marstatus: (single, married, widowed, divorced)
END
We may visualize particular, record-structured values of, for example, the variables
z: Complex
d: Date
p: Person

The following are the units for identifying data character, fields, sub fields , records, files.

A **file** is a collection of logically related records; e.g students file, stock file.

A **record** is a collection of logically related data fields; e. g Data relating to students in students file. In a database table records are usually in rows. Therefore, the table below has three (3) records. While a **field** is consecutive storage position of values. It is a unit of data within a record e. g student's number, Name, Age. In a database concept fields are usually in columns of a given table.

Data items for example , date are called group items if they can be divided into subsystems. The date for instance is represented by the day, the month andumber is called an elementary item, because it can not be sub-divided into sud-items otherwise known as **sub fields** called . It is indeed treated as a single item.

Character is the smallest unit of information. It includes letters, digits and special symbols such as + (Plus sign), _(minus sign), \, /, $,a,b,…z, A,B,…Z etc. Every character requires one **byte** of memory unit for storage in computer system.

WEEK 2 :

Graph.

This week learning ontcomes:
• Define a graph.
• State properties of graph : routes, edge, sequence, directed and nondirected.
. Computer representation of graphs.
• Describe operations such as precede, less than points to , move to , search, change, entry.

Introduction

Graphs are a commonly used data structure because they can be used to model many real-world problems. A graph consists of a set of nodes with an arbitrary number of connections, or edges, between the nodes. These edges can be either directed or undirected and weighted or unweighted.

In this study we will examine the basics of graphs and created a Graph class. This class was similar to the BinaryTree class , the difference being that instead of only have a reference for at most two edges, the Graph class's GraphNodes could have an arbitrary number of references. This similarity is not surprising because trees are a special case of graphs

Definitions

Definition1. A *graph* $G = (V,E)$ is a finite nonempty set V of objects called *vertices* (the singular is *vertex*) together with a (possibly empty) set E of unordered pairs of distinct vertices of G called *edges.*

Graphs are a very expressive formalism for system modeling, especially when attributes are allowed. Our research is mainly focused on the use of graphs for system verification.

Up to now, there are two main different approaches of modeling (typed) attributed graphs and specifying their transformation. Here we report preliminary results of our investigation on a third approach. In our approach we couple a graph to a data signature that consists of unary operations only. Therefore, we transform arbitrary signatures into a structure comparable to what is called a graph structure signature in the literature, and arbitrary algebras into the corresponding algebra graph.

Some authors call a graph by the longer term ``undirected graph" and simply use the following definition of a directed graph as a graph. However when using Definition 1 of a graph, it is standard practice to abbreviate the phrase ``directed graph" (as done below in Definition 2) with the word digraph.

Definition 2. A *digraph* $G = (V, E)$ is a finite nonempty set V of vertices together with a (possibly empty) set E of ordered pairs of vertices of G called *arcs*.

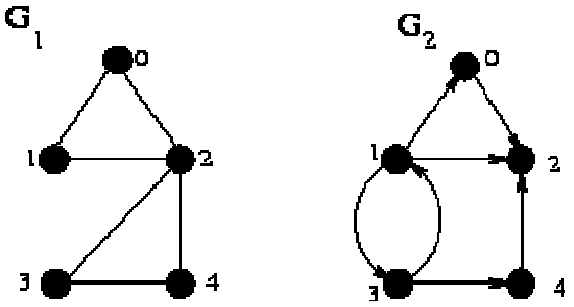
An arc that begins and ends at a same vertex u is called a *loop*. We usually (but not always) disallow loops in our digraphs. By being defined as a set, E does not contain duplicate (or multiple) edges/arcs between the same two vertices. For a given graph (or digraph) G we also denote the set of vertices by $V(G)$ and the set of edges (or arcs) by $E(G)$ to lessen any ambiguity.

Definition 3. The order of a graph (digraph) $G = (V, E)$ is $|V|$, sometimes denoted by $|G|$, and the size of this graph is $|E|$.

Sometimes we view a graph as a digraph where every unordered edge (u, v) is replaced by two directed arcs (u, v) and (v, u). In this case, the size of a graph is half the size of the corresponding digraph.

In the next example we display a graph G_1 and a digraph G_2 both of order 5. The size of the graph G_1 is 6 where $E(G_1)$ = {(0, 1), (0, 2), (1, 2), (2, 3), (2, 4), (3, 4)} while the size of the digraph G_2 is 7 where $E(G_2)$ = {(0, 2), (1, 0), (1, 2), (1, 3), (3, 1), (3, 4), (4, 2)}.

Example 4. A pictorial example of a graph G_1 and a digraph G_2 is given below.



Definition 5. A *walk* in a graph (digraph) G is a sequence of vertices $v_0v_1...v_n$ such that, for all $0 \leq i < n$, (v_i, v_{i+1}) is an edge (arc) in G. The *length* of the walk $v_0v_1...v_n$ is the number n (i.e., number of edges/arcs). A *path* is a walk in which no vertex is repeated. A *cycle* is a walk (of length at least three for graphs) in which $v_0 = v_n$ and no other vertex is repeated; sometimes, if it is understood, we omit v_n from the sequence.

Example 6. For the graph G_1 of Example 4 the following sequences of vertices are classified as being walks, paths, or cycles.

$v_0v_1...$	is walk?	is path?	is cycle?
0 1 2 3 4	yes	yes	no
0 1 2 0	yes	no	yes
0 1 2	yes	yes	yes (understood)
0 3 2	no	no	no
0 1 0	yes	no	no

Example 7. For the digraph G_2 of Example 4 the following sequences of vertices are classified as being walks, paths, or cycles.

$v_0v_1...$	is walk?	is path?	is cycle?
0 1 2 3 4	no	no	no
0 2 4	no	no	no
3 1 2	yes	yes	no
1 3 1	yes	no	yes
3 1 3 1 0	yes	no	no

Definition 8. A graph G is *connected* if there is a path between all pairs of vertices u and v of $V(G)$. A digraph G is *strongly connected* if there is a path from vertex u to vertex v for all pairs u and v in $V(G)$.

In Example 4 the graph G_1 is connected but the digraph G_2 is not strongly connected because there are no arcs leaving vertex 2. The *underlying graph* (by replacing each arc with an edge) of G_2 is connected, however.

Definition 9. In a graph, the *degree* of a vertex v, denoted by *deg(v)*, is the number of edges incident to v. For digraphs, the *out-degree* of a vertex v is the number of arcs $\{(v,x) \in E \mid x \in V\}$ incident from v (leaving v) and the *in-degree* of vertex v is the number of arcs $\{(x,v) \in E \mid x \in V\}$ incident to v (entering v).

For a graph the in-degree and out-degree's are the same as the degree. For our graph G_1, we have deg(0)=2, deg(1)=2, deg(2)=4, deg(3)=2 and deg(4)=2. We may concisely write this as a *degree sequence* (2, 2, 4, 2, 2) if there is a natural ordering (e.g., 0,1,2,3,4) of the vertices. The in-degree sequence and out-degree sequence of the digraph G_2 are (1, 1, 3, 1, 1) and (1, 3, 0, 2, 1), respectively. The degree of a vertex of a digraph is sometimes defined as the sum of its in-degree and out-degree. Using this definition, a degree sequence of G_2 would be (2, 4, 3, 3, 2).

Definition 10. The *diameter* of a connected graph (strongly connected digraph $G = (V, E)$ is the least integer D such that for all vertices u and v in G we have $d(u, v) \leq D$, where $d(u, v)$ denotes the *distance* from u to v in G, that is, the length of a shortest path between u and v.

Example 11. The diameter of graph G_1 of Example 4 is 2. We calculated d(0,1)=1, d(0,2)=1, d(0,3)=2, d(0,4)=2, d(1,2)=1, d(1,3)=2, d(1,4)=2, d(2,3)=1, d(2,4)=1 and d(3,4)=1. Note for graphs, we have d(x, y) = d(y, x) for all vertices x and y.

Since the digraph G_2 is not strongly connected the diameter is undefined. However, we can compute shortest distances between various pairs of vertices: d(0,2)=1, d(1,0)=1, d(1,2)=1, d(3,1)=1, d(3,0)=2, d(3,2)=2, d(3,4)=1 and d(4,2)=1.

Computer representations of graphs

There are two common computer representations for graphs (or digraphs), called adjacency matrices and adjacency lists. For a graph G of order n, an adjacency matrix representation is a boolean matrix (often encoded with 0's and 1's) of dimension n such that entry (i,j) is true if and only if edge/arc (i,j) is in E(G). For a graph G of order n, an adjacency lists representation is n lists such that the i-th list contains a sequence (often sorted) of out-neighbours of vertex i of G.

We can see the structure of these representations more clearly with examples.

Example 12. We give adjacency matrices for the graph G_1 and digraph G_2 of Example 4 below.

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Notice that the 1's in the rows represent how many out-neighbours and the 1's in the columns represent how many in-neighbours a vertex has.

Example 13. We give adjacency lists for the graph G_1 and digraph G_2 of Example 4 below.

```
0: 1 2        0: 2
1: 0 2        1: 0 2 3
2: 0 1 3 4    2:
3: 2 4        3: 1 4
4: 2 3        4: 2
```

Only the out-neighbours are listed in the adjacency lists representation. The numbers with colons (i:) denote the index i of the lists (and are not really necessary). An empty list can occur (e.g., list 2 of the digraph G_2).

For a graph/digraph with n vertices and m edges, the adjacency matrix representation requires $O(n^2)$ storage while the adjacency lists representation requires $O(m)$ storage. So for sparse graphs the latter is probably preferable. However, to check whether edge/arc (i, j) is in the graph the adjacency matrix representation has constant-time lookup, while the adjacency lists representation may require $O(n)$ time in the worst case.

We mention that there are also other specialized graph representations besides the two mentioned in this section. These data structures take advantage of the graph structure for improved storage or access time, often for families of graphs sharing a common property.

WEEK 3 :

Symbol, and relations

The week Learning outcomes :

- Define symbols, and relations.
- Explain equivalence relation.
- Explain composite relation.

Relations

A binary relation is determined by specifying all ordered pairs of objects in that relation; it does not matter by what property the set of these ordered pairs is described. We are led to the following definition.

Definition. A set R is a *binary relation* if all elements of R are ordered pairs, i.e., if for any $z \in R$ there exist x and y such that $z = (x, y)$.

It is customary to write xRy instead of $(x, y) \in R$. We say that *x is in relation R with y* if xRy holds.

The set of all x which are in relation R with some y is called the *domain* of R and denoted by "dom R." So dom $R = \{x \mid$ there exists y such that $xRy\}$. dom R is the set of all first coordinates of ordered pairs in R.

The set of all y such that, for some x, x is in relation R with y is called the *range* of R, denoted by "ran R." So ran $R = \{y \mid$ there exists x such that $xRy\}$.

Symbol

A **symbol** is something such as an object, picture, written word, sound, or particular mark that represents something else by association, resemblance, or convention. For example, a red octagon may stand for "STOP". On maps, crossed sabres may indicate a battlefield. Numerals are symbols for numbers.

All language consists of symbols. The word "cat" is not a cat, but represents the idea of a cat.

Language and symbols

All languages are made up of symbols. Spoken words are the symbols of mental experience, and written words are the symbols of spoken words.

The word "cat", for example, whether spoken or written, is not a literal cat but a sequence of symbols that by convention associate the word with a concept. Hence, the written or spoken word "cat" represents (or stands for) a particular concept formed in the mind. A drawing of a cat, or a stuffed cat, could also serve as a symbol for the idea of a cat.

The study or interpretation of symbols is known as symbology, and the study of signs is known as semiotics.

Symbols and Corresponding HTML Entities

If the leftmost column below shows `&equiv;`, a square or nothing instead of the actual symbol, your browser does not support HTML entities; please use the picture version of this document instead.

Relational Operators [3]

Symbol	LaTeX Command [2]	HTML Entity [1]	Comment
≡	\equiv	≡	
≈	\approx	≈	
∝	\propto	∝	
≃	\simeq		
~	\sim	∼	
≠	\neq	≠	
≥	\geq		
≫	\gg		
≪	\ll		

Logic Symbols [3]

Symbol	LaTeX Command [2]	HTML Entity [1]	Comment
¬	\neg	¬	
∧	\wedge	∧	
∨	\vee	∨	
⊕	\oplus	⊕	
⇒	\Rightarrow		
⇔	\Leftrightarrow		
∃	\exists	∃	
∀	\forall	∀	

Set Symbols [3]

Symbol	LaTeX Command [2]	HTML Entity [1]	Comment
∩	\cap	∩	
∪	\cup	∪	
⊃	\supset	⊃	
⊂	\subset	⊂	
∅	\emptyset	∅	
ℤ	\mathbb{Z}		requires the amsfonts and amssymb packages.
∈	\in	∈	
∉	\notin	∉	
⋈	\Join		requires the latexsym package (present in most LaTeX distributions).

Miscellaneous Math Symbols [3]

Symbol	LaTeX Command [2]	HTML Entity [1]	Comment
′	\prime	′	
⌋	\rfloor	⌋	
∞	\infty	∞	

Equivalence relation

Equivalence relation is a binary relation between two elements of a set which groups them together as being "equivalent" in some way. Let *a*, *b*, and *c* be arbitrary elements of some set *X*. Then "$a \sim b$" or "$a \equiv b$" denotes that *a* is equivalent to *b*.

An equivalence relation "~" is reflexive, symmetric, and transitive. In other words, the following must hold for "~" to be an equivalence relation on *X*:

An equivalence relation partitions a set into several disjoint subsets, called equivalence classes. All the elements in a given equivalence class are equivalent among themselves, and no element is equivalent with any element from a different class.

- Reflexivity: $a \sim a$
- Symmetry: if $a \sim b$ then $b \sim a$
- Transitivity: if $a \sim b$ and $b \sim c$ then $a \sim c$.

- The equivalence class of *a* under "~", denoted $[a]$, is the subset of *X* for which every element *b*, $a \sim b$. *X* together with "~" is called a setoid.

Examples of equivalence relations

A ubiquitous equivalence relation is the equality ("=") relation between elements of any set. Other examples include:

- "Has the same birthday as" on the set of all people, given naive set theory.
- "Is similar to" or "congruent to" on the set of all triangles.

- "Is congruent to modulo *n*" on the integers.
- "Has the same image under a function" on the elements of the domain of the function.
- Logical equivalence of logical sentences.
- "Is isomorphic to" on models of a set of sentences.
- In some axiomatic set theories other than the canonical ZFC (e.g., New Foundations and related theories):
    - Similarity on the universe of well-orderings gives rise to equivalence classes that are the ordinal numbers.
    - Equinumerosity on the universe of:
        - Finite sets gives rise to equivalence classes which are the natural numbers.
        - Infinite sets gives rise to equivalence classes which are the transfinite cardinal numbers.
- Let a, b, c, d be natural numbers, and let (a, b) and (c, d) be ordered pairs of such numbers. Then the equivalence classes under the relation $(a, b) \sim (c, d)$ are the:
    - Integers if $a + d = b + c$;
    - Positive rational numbers if $ad = bc$.
- Let (r_n) and (s_n) be any two Cauchy sequences of rational numbers. The real numbers are the equivalence classes of the relation $(r_n) \sim (s_n)$, if the sequence $(r_n - s_n)$ has limit 0.
- Green's relations are five equivalence relations on the elements of a semigroup.
- "Is parallel to" on the set of subspaces of an affine space.

Examples of relations that are not equivalences

- The relation "≥" between real numbers is reflexive and transitive, but not symmetric. For example, $7 \geq 5$ does not imply that $5 \geq 7$. It is, however, a partial order.
- The relation "has a common factor greater than 1 with" between natural numbers greater than 1, is reflexive and symmetric, but not transitive. (The natural numbers 2 and 6 have a common factor greater than 1, and 6 and 3 have a common factor greater than 1, but 2 and 3 do not have a common factor greater than 1).
- The empty relation R on a non-empty set X (i.e. aRb is never true) is vacuously symmetric and transitive, but not reflexive. (If X is also empty then R *is* reflexive.)
- The relation "is approximately equal to" between real numbers, even if more precisely defined, is not an equivalence relation, because although reflexive and symmetric, it is not transitive, since multiple small changes can accumulate to become a big change. However, if the approximation is defined asymptotically, for example by saying that two functions f and g are approximately equal near some point if the limit of f-g is 0 at that point, then this defines an equivalence relation.
- The relation "is a sibling of" on the set of all human beings is not an equivalence relation. Although siblinghood is symmetric (if A is a sibling of B, then B is a sibling of A) it is neither reflexive (no one is a sibling of himself), nor transitive (since if A is a sibling of B, then B is a sibling of A, but A is not a sibling of A). Instead of being transitive, siblinghood is "almost transitive", meaning that if $A \sim B$, and $B \sim C$, and $A \neq C$, then $A \sim C$. However, the relation "A is a sibling of B or A is B" is an equivalence relation. (This applies only to *full* siblings. A and B could have the same mother, and B and C the same father, without A and C having a common parent.)
- The concept of parallelism in ordered geometry is not symmetric and is, therefore, not an equivalence relation.

- An equivalence relation on a set is never an equivalence relation on a proper superset of that set. For example $R = \{(1,1), (1,2), (1,3), (2,1), (2,2), (2,3), (3,1), (3,2), (3,3)\}$ is an equivalence relation on {1,2,3} but not on {1,2,3,4} or on the natural number. The problem is that reflexivity fails because (4,4) is not a member.

Connection to other relations

- A congruence relation is an equivalence relation whose domain X is also the underlying set for an algebraic structure, and which respects the additional structure. In general, congruence relations play the role of kernels of homomorphisms, and the quotient of a structure by a congruence relation can be formed. In many important cases congruence relations have an alternative representation as substructures of the structure on which they are defined. E.g. the congruence relations on groups correspond to the normal subgroups.
- A partial order replaces symmetry with antisymmetry and is thus reflexive, antisymmetric, and transitive. Equality is the only relation that is both an equivalence relation and a partial order.
- A strict partial order is irreflexive, transitive, and asymmetric.
- A partial equivalence relation is transitive and symmetric. Transitive and symmetric imply reflexive iff for all $a \in X$ exists $b \in X$ such that $a \sim b$.
- A dependency relation is reflexive and symmetric.
- A preorder is reflexive and transitive.

Equivalence class, quotient set, partition

Let X be a nonempty set with typical elements a and b. Some definitions:

- The set of all a and b for which $a \sim b$ holds make up an **equivalence class** of X by ~. Let $[a] =: \{x \in X : x \sim a\}$ denote the equivalence class to which a belongs. Then all elements of X equivalent to each other are also elements of the same equivalence class: $\forall a, b \in X\ (a \sim b \leftrightarrow [a] = [b])$.
- The set of all possible equivalence classes of X by ~, denoted $X/\sim =: \{[x] : x \in X\}$, is the **quotient set** of X by ~. If X is a topological space, there is a natural way of transforming $X/\sim$ into a topological space; see quotient space for the details.
- The **projection** of ~ is the function $\pi : X \to X/\sim$, defined by $\pi(x) = [x]$, mapping elements of X into their respective equivalence classes by ~.

Theorem on projections (Birkhoff and Mac Lane 1999: 35, Th. 19): Let the function $f: X \to B$ be such that $a \sim b \to f(a) = f(b)$. Then there is a unique function $g : X/\sim \to B$, such that $f = g\pi$. If f is a surjection and $a \sim b \leftrightarrow f(a) = f(b)$, then g is a bijection.

- The **equivalence kernel** of a function f is the equivalence relation, denoted Ef, such that $xEfy \leftrightarrow f(x) = f(y)$. The equivalence kernel of an injection is the identity relation.
- A **partition** of X is a set P of subsets of X, such that every element of X is an element of a single element of P. Each element of P is a **cell** of the partition. Moreover, the elements of P are pairwise disjoint and their union is X.

Theorem ("Fundamental Theorem of Equivalence Relations": Wallace 1998: 31, Th. 8; Dummit and Foote 2004: 3, Prop. 2):

- An equivalence relation ~ partitions X.
- Conversely, corresponding to any partition of X, there exists an equivalence relation ~ on X.

In both cases, the cells of the partition of X are the equivalence classes of X by ~. Since each element of X belongs to a unique cell of any partition of X, and since each cell of the partition is identical to an equivalence class of X by ~, each element of X belongs to a unique equivalence class of X by ~. Thus there is a natural bijection from the set of all possible equivalence relations on X and the set of all partitions of X.

Counting possible partitions. Let X be a finite set with n elements. Since every equivalence relation over X corresponds to a partition of X, and vice versa, the number of possible equivalence relations on X equals the number of distinct partitions of X, which is the *nth* Bell number B_n:

Generating equivalence relations

- Given any set X, there is an equivalence relation over the set of all possible functions $X \to X$. Two such functions are deemed equivalent when their respective sets of fixpoints have the same cardinality, corresponding to cycles of length one in a permutation. Functions equivalent in this manner form an equivalence class on X^2, and these equivalence classes partition X^2.

- An equivalence relation ~ on X is the equivalence kernel of its surjective projection $\pi : X \to X/\sim$. (Birkhoff and Mac Lane 1999: 33 Th. 18). Conversely, any surjection between sets determines a partition on its domain, the set of preimages of singleton [*disambiguation needed*]s in the codomain. Thus an equivalence relation over X, a partition of X, and a projection whose domain is X, are three equivalent ways of specifying the same thing.

- The intersection of any collection of equivalence relations over X (viewed as a subset of $X \times X$) is also an equivalence relation. This yields a convenient way of generating an equivalence relation: given any binary relation R on X, the equivalence relation *generated by* R is the smallest equivalence relation containing R. Concretely, R generates the equivalence relation $a \sim b$ iff there exist elements $x_1, x_2, \ldots, x_n$ in X such that $a = x_1$, $b = x_n$, and $(x_i, x_{i+1}) \in R$ or $(x_{i+1}, x_i) \in R$, $i = 1, \ldots, n\text{-}1$.

  Note that the equivalence relation generated in this manner can be trivial. For instance, the equivalence relation ~ generated by:

- The binary relation ≤ has exactly one equivalence class, *X* itself, because *x* ~ *y* for all *x* and *y*;
- An antisymmetric relation has equivalence classes that are the singletons[*disambiguation needed*] of *X*.

- Let *r* be any sort of relation on *X*. Then $r \cup r^{-1}$ is a symmetric relation. The transitive closure *s* of $r \cup r^{-1}$ assures that *s* is transitive and reflexive. Moreover, *s* is the "smallest" equivalence relation containing *r*, and *r*/*s* partially orders *X*/*s*.

- Equivalence relations can construct new spaces by "gluing things together." Let *X* be the unit Cartesian square [0,1] × [0,1], and let ~ be the equivalence relation on *X* defined by $\forall a, b \in [0,1]$ $((a, 0) \sim (a, 1) \wedge (0, b) \sim (1, b))$. Then the quotient space *X*/~ can be naturally identified with a torus: take a square piece of paper, bend and glue together the upper and lower edge to form a cylinder, then bend the resulting cylinder so as to glue together its two open ends, resulting in a torus.

Composite Relations

If the elements of a set ***A*** are related to those of a set ***B***, and those of ***B*** are in turn related to the elements of a set ***C***, then one can expect a relation between ***A*** and ***C***. For example, if Tom is my father(parent-child relation) and Sarah is a sister of Tom (sister relation), then Sarah is my aunt (aunt-nephew/niece relation). Composite relations give that kind of relations.

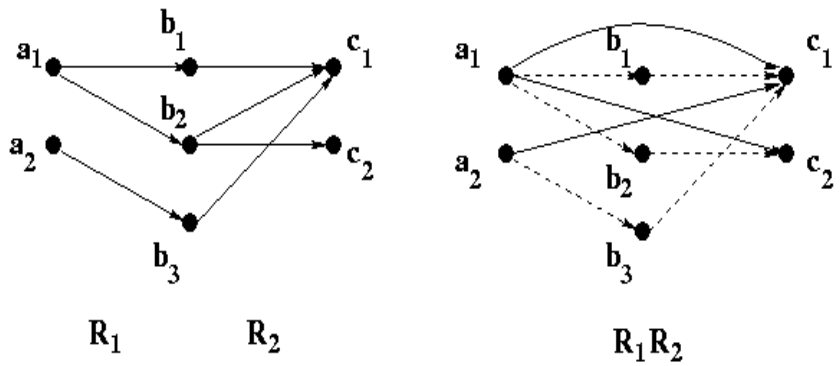
Definition(composite relation): Let R_1 be a binary relation from a set ***A*** to a set ***B***, R_2 a binary relation from ***B*** to a set ***C***. Then the **composite relation** from ***A*** to ***C*** denoted by R_1R_2(also denoted by $R_1 \bigcirc R_2$ is defined as

$$R_1R_2 = \{<a, c> \mid a \in A \wedge c \in C \wedge \exists b\, [b \in B \wedge <a, b> \in R_1 \wedge <b, c> \in R_2\,]\,\}\,.$$

In English, this means that an element ***a*** in ***A*** is related to an element ***c*** in ***C*** if there is an element ***b*** in ***B*** such that ***a*** is related to ***b*** by R_1 and ***b*** is related to ***c*** by R_2 . Thus R_1R_2 is a relation from ***A*** to ***C*** via ***B*** in a sense. If R_1 is a parent-child relation and R_2 is a sister relation, then R_1R_2 is an aunt-nephew/niece relation.

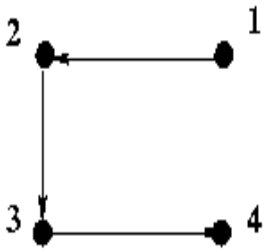
Example 1: Let $A = \{a_1 , a_2\}$, $B = \{b_1 , b_2 , b_3\}$, and $C = \{c_1 , c_2\}$. Also let $R_1 = \{<a_1 , b_1> , <a_1 , b_2> , <a_2 , b_3> \}$, and $R_2 = \{<b_1 , c_1> , <b_2 , c_1> , <b_2 , c_2> , <b_3 , c_1> \}$. Then $R_1R_2 = \{<a_1 , c_1> , <a_1 , c_2> , <a_2 , c_1> \}$.

This is illustrated in the following figure. The dashed lines in the figure of R_1R_2 indicate the ordered pairs in R_1R_2, and dotted lines show ordered pairs that produce the dashed lines. (The lines in the left figure are all supposed to be solid lines.)

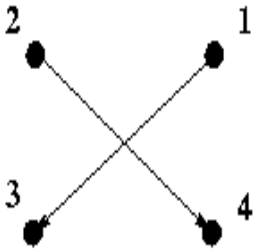


Example 2: If R is the parent-child relation on a set of people A, then RR, also denoted by R^2, is the grandparent-grandchild relation on A.

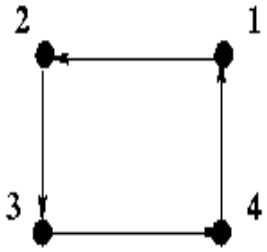
More examples:
The digraphs of R^2 for several simple relations R are shown below:



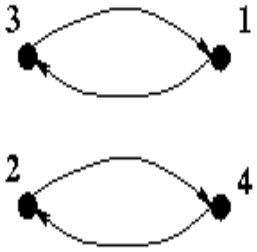
R_1



R_1^2



R_2



R_2^2

R_3

R_3^2

Basis Clause: $R^0 = E$, where E is the equality relation on A.

Properties of Composite Relations

Composite relations defined above have the following properties. Let R_1 be a relation from A to B, and R_2 and R_3 be relations from B to C. Then

1. $R_1(R_2R_3) = (R_1R_2)R_3$
2. $R_1(R_2 \cup R_3) = R_1R_2 \cup R_1R_3$
3. $R_1(R_2 \cap R_3) \subseteq R_1R_2 \cap R_1R_3$

Proofs for these properties are not necessary

Powers of Relation

Let R be a binary relation on A. Then R^n for all positive integers n is defined recursively as follows:

Definition(power of relation):

Inductive Clause: For an arbitrary natural number n , $R^{n+1} = R^n R$.
Note that there is no need for extremal clause here.

Thus for example $R^1 = R$, $R^2 = RR$, and $R^3 = R^2R = (RR)R = R(RR) = RRR$.

The powers of binary relation R on a set A defined above have the following properties.

1. $R^{m+n} = R^m R^n$,
2. $(R^m)^n = R^{mn}$.

Using composite identity relationships

An identity relationship establishes an association between business objects or other data on a *one-to-one basis*. A composite identity relationship relates two business objects through a composite key attribute.

Creating composite identity relationship definitions

Identity relationship definitions differ from lookup relationship definitions in that the participant types are business objects, *not* of the type Data (the first selection in the participant types list). As with a simple identity relationship:

- The composite identity relationship consists of the generic business object and at least one application-specific business object.
- The participant type is a business object for *all* participants.

However, for a composite identity relationship, the participant attribute for every participant is a composite key. This composite key usually consists of a unique key from a parent business object and a nonunique key from a child business object.

Steps for creating composite identity relationship definitions

To create a relationship definition for a composite identity relationship, perform the following steps:

1. Create a participant definition whose participant type is the parent business object.
2. Set the first participant attribute to the key of the parent business object.

   Tip: Expand the parent business object and select the key attribute.

3. Set the second participant attribute to the key of the child attribute.

   Tip: Expand the parent business object, then expand the child attribute within the parent. Select the key attribute from this child object.

4. Repeat steps 1-3 for each of the participants. As with all composite identity relationships, this relationship contains one participant for the generic business object and at least one participant for a application-specific business object. Each participant consists of two attributes: the key of the parent business object and the key of the child business object (from the attribute within the parent business object).

Restriction: To manage composite relationships, the server creates internal tables. A table is created for each role in the relationship. A unique *index* is then created on these tables across all *key attributes* of the relationship. (In other words, the columns which correspond to the key attributes of the relationship are the participants of the index.) The column sizes of the internal tables have a direct relation to the attributes of the relationship and are determined by the value of the MaxLength attribute for the relationship.

Databases typically have restrictions on the size of the indexes that can be created. For instance, DB2 has an index limitation of 1024 bytes with the default page size. Thus, depending on the MaxLength attribute of a relationship and the number of attributes in a relationship, you could run into an index size restriction while creating composite relationships.

Important:

- You must ensure that appropriate MaxLength values are set in the repository file for all *key attributes* of a relationship, such that the total index would never exceed the index size limitations of the underlying DBMS.

  If the MaxLength attribute for type String is not specified, the default is nvarchar(255) in the SQLServer. Thus, if a relationship has *N* Keys, all of type String and the default MaxLength attribute of 255 bytes, the index size would be ((N*255)*2) + 16 bytes. You can see that you would exceed the SQLServer 7 limit of 900 bytes quite easily when N takes values of >=2 for the default MaxLength value of 255 bytes for type String.

- Remember, too, that even when some DBMS'es support large indexes, it comes at the cost of performance; hence, it is always a good idea to keep index sizes to the minimum.

Determining the relationship action

Table 100 shows the activity function blocks that the Mapping API provides to maintain a composite identity relationship from the child attribute of the parent source business object. The actions that these methods take depends on the source object's verb and the calling context.

Table 100. Maintaining a composite identity relationship from the child attribute

Function block	Description
General/APIs/Identity Relationship/ Maintain Child Verb	Set source child verb correctly
General/APIs/Identity Relationship/ Maintain Composite Relationship	Perform appropriate action on the relationship tables

Actions of General/APIs/Identity Relationship/Maintain Composite Relationship

The Maintain Composite Relationship function block will generate Java code that calls the mapping API maintainCompositeRelationship(), which will manage relationship tables for a composite identity relationship. This method ensures that the relationship instances contain the associated application-specific key values for each relationship instance ID. This method automatically handles all of the basic adding and deleting of participants and relationship instances for a composite identity relationship.

The actions that maintainCompositeRelationship() takes are based on the value of the business object's verb and the calling context. The method iterates through the child objects of a specified participant, calling the maintainSimpleIdentityRelationship() on each one to correctly set the child key value. As with maintainSimpleIdentityRelationship(), the action that maintainCompositeRelationship() takes is based on the following information:

- The calling context: EVENT_DELIVERY, ACCESS_REQUEST, SERVICE_CALL_REQUEST, SERVICE_CALL_RESPONSE, SERVICE_CALL_FAILURE, and ACCESS_RESPONSE
- The verb of the source business object: Create, Update, Delete, or Retrieve

The maintainCompositeRelationship() method deals *only* with composite keys that extend to only two nested levels. In other words, the method cannot handle the case where the child object's composite key depends on values in its grandparent objects.

Example: If A is the top-level business object, B is the child of A, and C is the child of B, the two methods will *not* support the participant definitions for the child object C that are as follows:

- The participant type is A and the attributes are:
  - key attribute of A: ID
  - key attribute of B: B[0].ID
  - key attribute of C: B[0].C[0].ID
- The participant type is A and the attributes are:
  - key attribute of A: ID
  - key attribute of C: B[0].C[0].ID

To access a grandchild object, these methods only support the participant definitions that are as follows:

- The participant type is B and the attributes are:
  - key attribute of B: ID
  - key attribute of C: C[0].ID
- The participant type is B and the attributes are:
  - key attribute of B: ID
  - first key attribute of C: C[0].ID1
  - second key attribute of C: C[0].ID2

Actions of General/APIs/Identity Relationship/Maintain Child Verb

The Maintain Child Verb function block will generate Java code that calls the mapping API maintainChildVerb(), which will maintain the verb of the child objects in the destination business object. It can handle child objects whose key attributes are part of a composite identity relationship. When you call maintainChildVerb() as part of a composite relationship, make sure that its last parameter has a value of true. This method ensures that the verb settings are appropriate given the verb in the parent source object and the calling context.

Customizing map rules for a composite identity relationship

Once you have created the relationship definition and participant definitions for the composite identity relationship, you can customize the map to maintain the composite identity relationship.

A composite identity relationship manages a composite key. Therefore, managing this kind of relationship involves managing both parts of the composite key. To code a composite identity relationship, you need to customize the mapping transformation rules for both the parent and child business objects, as Table 3.1 shows.

Table 3.1 Activity function blocks for a composite identity relationship

Map involved	Business object involved	Attribute	Activity function blocks
Main	Parent business object	Top-level business object	Use a Cross-Reference transformation rule
		Child attribute (child business object)	General/APIs/Identity Relationship/Maintain Composite Relationship General/APIs/Identity Relationship/Maintain Child Verb General/APIs/Identity Relationship/Update My Children (optional)
Submap	Child business object	Key attribute (nonunique key)	Define a Move or Set Value transformation for the verb.

If child business objects have a nonunique key attribute, you can relate these child business objects in a composite identity relationship.

The following sections describe the steps for customizing this composite identity relationship:

- Steps for customizing the main map
- Customizing the submap
- Managing child instances

Steps for customizing the main map

In the map for the parent business object (the main map), add the mapping code to the parent attributes:

1. Map the verb of the top-level business object by defining a Move or Set Value transformation rule.
2. Define a Cross-Reference transformation between the top-level business objects.
3. Define a Custom transformation for the child attribute and use the General/APIs/Identity Relationship/Maintain Composite Relationship function block in Activity Editor.

Steps for coding the child attribute

The child attribute of the parent object contains the child business object. This child object is usually a multiple cardinality business object. It contains a key attribute whose value identifies the child. However, this key value is not required to be unique. Therefore, it does not uniquely identify one child object among those for the same parent nor is it sufficient to identify the child object among child objects for all instances of the parent object.

To identify such a child object uniquely, the relationship uses a composite key. In the composite key, the parent key uniquely identifies the parent object. The combination of parent key and child key uniquely identifies the child object. In the map for the parent business object (the main map), add the mapping code to the attribute that contains the child business object. In Activity Editor for this attribute, perform the following steps to code a composite identity relationship:

1. Define a Submap transformation for the child business object attribute of the main map. Usually mapping transformations for a child object are done within a submap, especially if the child object has multiple cardinality.
2. In the main map, define a Custom transformation rule for the child verb and use the General/APIs/Identity Relationship/Maintain Child Verb function block to maintain the child business object's verb.

   The last input parameter of the General/APIs/Identity Relationship/Maintain Child Verb function block is a boolean flag to indicate whether the child objects are participating in a composite relationship. Make sure you pass a value of true as the last argument to maintainChildVerb() because this child object participates in a composite, not a simple identity relationship. Make sure you call maintainChildVerb() *before* the code that calls the submap.

3. To maintain this composite key for the parent source object, customize the mapping rule to use the General/APIs/Identity Relationship/Maintain Composite Relationship function block.
4. To maintain the relationship tables in the case where a parent object has an Update verb caused by child objects being deleted, customize the mapping rule to use the General/APIs/Identity Relationship/Update My Children function block.

   Tip: Make sure the transformation rule that contains the Update My Children function block has an execution order after the transformation rule that contains the Maintain Composite Relationship function block.

Example of customizing the map for a Composite Identity Relationship

The following example describes how the map can be customized for a Composite Identity Relationship.

1. In the main map, define a Custom transformation rule between the child business object's verbs. Use the General/APIs/Identity Relationship/Maintain Child Verb function block in the customized activity to maintain the verb for the child business objects.

   The goal of this custom activity is to use the maintainChildVerb() API to set the child business object verb based on the map execution context and the verb of the parent business object. Figure3.2 shows this custom activity.

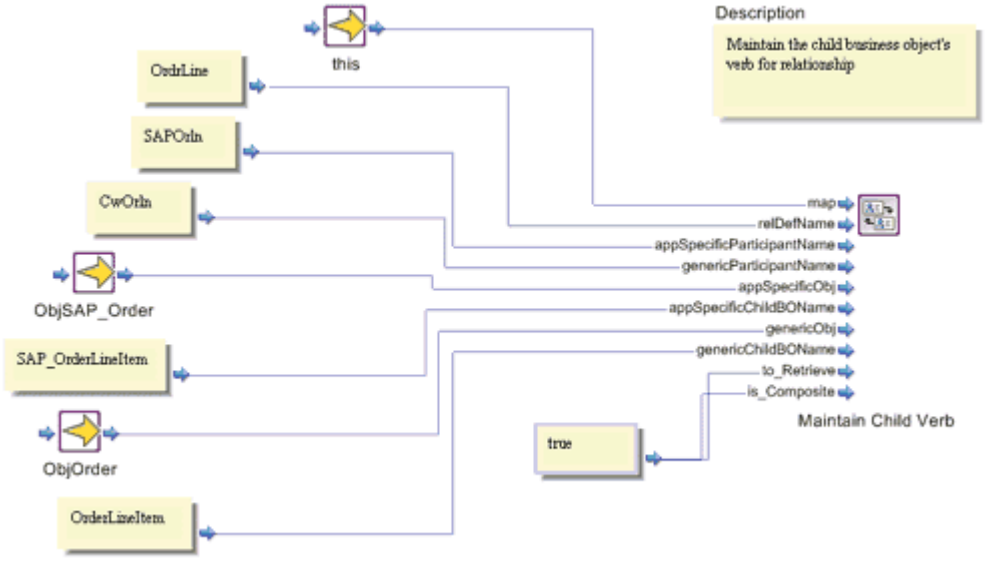


Figure 3.2. Using the Maintain Child Verb function block

2. If necessary, define a Submap transformation rule between the child business object to perform any mapping necessary in the child level.
3. Define a Custom transformation rule between the top-level business objects. Use the General/APIs/Identity Relationship/Maintain Composite Relationship function block in the customized activity to maintain the composite identity relationship for this map.

   The goal of this custom activity is to use the maintainComposite Relationship() API to maintain a composite identity relationship within the map. Figure 3.3 shows this custom activity.

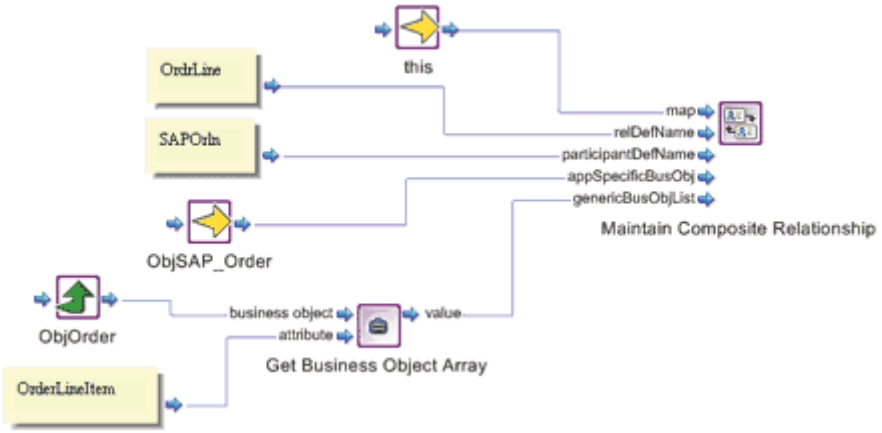


Figure 3.3. Using the Maintain Composite Relationship function block

4. Define a Custom transformation rule mapping from the source top-level business object to the destination child business object attribute. Use the General/APIs/Identity Relationship/Update My Children function block in the customized activity to maintain the child instances in the relationship.

   The goal of this custom activity is to use the updateMyChildren() API to add or delete child instances in the specified parent/child relationship of the identity relationship.

Figure 3.4 shows this custom activity.

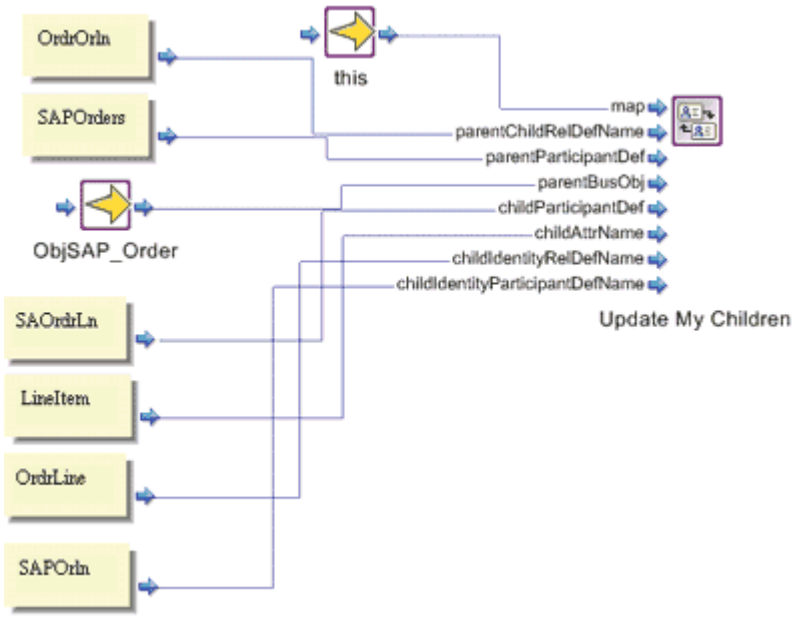


Figure 3.4. Using the Update My Children function block

Example of coding the child attribute

Here is a sample of how the code in the child attribute of the parent map might look. This code fragment would exist in the Order Line Item attribute of an SAP Order business object. It uses maintainChildVerb() to set the child object verbs, then calls a submap (Sub_SaOrderLieItem_to_CwOrderLineItem) in a for loop to handle mapping of the Order line items child object:

```
{
BusObjArray srcCollection_For_ObjSAP_Order_SAP_OrderLineItem =
   ObjSAP_Order.getBusObjArray("SAP_OrderLineItem");

//
// LOOP ONLY ON NON-EMPTY ARRAYS
// ----------------------------
//
// Perform the loop only if the source array is non-empty.
//
if ((srcCollection_For_ObjSAP_Order_SAP_OrderLineItem != null) &&
      (srcCollection_For_ObjSAP_Order_SAP_OrderLineItem.size() > 0))
   {
```

```
  int currentBusObjIndex_For_ObjSAP_Order_SAP_OrderLineItem;
  int lastInputIndex_For_ObjSAP_Order_SAP_OrderLineItem =
  srcCollection_For_ObjSAP_Order_SAP_OrderLineItem.getLastIndex();

  // ----
  IdentityRelationship.maintainChildVerb(
    "OrdrLine",
    "SAPOrln",
    "CWOrln",
    ObjSAP_Order,
    "SAP_OrderLineItem",
    ObjOrder,
    "OrderLineItem",
    cwExecCtx,
    true,
    true);

  // ----
  for (currentBusObjIndex_For_ObjSAP_Order_SAP_
       OrderLineItem = 0;
       currentBusObjIndex_For_ObjSAP_Order_SAP_OrderLineItem <=
         lastInputIndex_For_ObjSAP_Order_SAP_OrderLineItem;
       currentBusObjIndex_For_ObjSAP_Order_SAP_OrderLineItem++)
    {
    BusObj currentBusObj_For_ObjSAP_Order_SAP_OrderLineItem =
(BusObj) (srcCollection_For_ObjSAP_Order_SAP_OrderLineItem.elementAt(
          currentBusObjIndex_For_ObjSAP_Order_SAP_OrderLineItem));

    //
    // INVOKE MAP ON VALID OBJECTS
    // --------------------------
    //
    // Invoke the map only on those children objects that meet
    // certain criteria.
    //
    if (currentBusObj_For_ObjSAP_Order_SAP_OrderLineItem != null)
      {
      BusObj[] _cw_inObjs = new BusObj[2];
      _cw_inObjs[0] =
           currentBusObj_For_ObjSAP_Order_SAP_OrderLineItem;
      _cw_inObjs[1] = ObjSAP_Order;
      logInfo ("*** Inside SAPCW header, verb is: " +
        (_cw_inObjs[0].getVerb()));

      try
        {
```

```
        BusObj[] _cw_outObjs = DtpMapService.runMap(
          "Sub_SaOrderLineItem_to_CwOrderLineItem",
          "CwMap",
          _cw_inObjs,
          cwExecCtx);
        _cw_outObjs[0].setVerb(_cw_inObjs[0].getVerb());
        ObjOrder.setWithCreate("OrderLineItem", _cw_outObjs[0]);
        }

      catch (MapNotFoundException me)
        {
        logError(5502,
          " Sub_SaOrderLineItem_to_CwOrderLineItem ");
        throw new MapFailureException ("Submap not found");
        }
      }
    }

// Start of the child relationship code
BusObjArray temp = (BusObjArray)ObjOrder.get("OrderLineItem");
try
  {
  IdentityRelationship.maintainCompositeRelationship(
    "OrdrLine",
    "SAPOrln",
    ObjSAP_Order,
    temp,
    cwExecCtx);
  }

catch RelationshipRuntimeException re
  {
  logError(re.toString());
  }

// This call to updateMyChildren() assumes the existence of the
// OrdrOrln parent/child relationship between the SAP_Order
// (parent) and SAP_OrderItem (child)
IdentityRelationship.updateMyChildren(
  "OrdrOrln",
  "SAOrders",
  ObjSAP_Order,
  "SAOrdrLn",
  "LineItem",
  "OrdrLine",
  "SAPOrln",
```

```
    cwExecCtx);

  // End of the child relationship code
  }
}
```

Customizing the submap

In the map for the child business object (the submap), add the mapping code to the the key attribute of the child object. The only code you need to add is a call to the setVerb() method to set the child object's verb to the parent object's verb.

Note:

When the child object primary key requires the maintainCompositeRelationship() method, make the call in the parent map, right after the end of the for loop for calling the submap. In the submap, the code for the destination object's primary key should contain the following line:

```
// maintainCompositeRelationship()
is called in the parent map.
```

WEEK 4 :

SETS

This week Learning outcomes:
• Define sets .
• Define set operations.
• Define the elements of set , subsets, super sets, universal sets and null set.
• Explain Storage set representation.

The type SET

The objects of study of Set Theory are *sets*. As sets are fundamental objects that can be used to define all other concepts in mathematics, they are not defined in terms of more fundamental concepts. Rather, sets are introduced either informally, and are understood as something self-evident, or, as is now standard in modern mathematics, axiomatically, and their properties are postulated by the appropriate formal axioms.

The language of set theory is based on a single fundamental relation, called *membership*. We say that *A* is a member of *B* (in symbols $A \in B$), or that the set *B* contains *A* as its element. The understanding is that a set is determined by its elements; in other words, two sets are deemed equal if they have exactly the same elements. In practice, one considers sets of numbers, sets of points, sets of functions, sets of some other sets and so on. In theory, it is not necessary to distinguish between objects that are members and objects that contain members -- the only objects one needs for the theory are sets. See the supplement

The type SET denotes sets whose elements are integers in the range 0 to a small number, typically 31 or 63.
Given, for example, variables
VAR r, s, t: SET
possible assignments are
r := {5}; s := {x, y .. z}; t := {}
Here, the value assigned to r is the singleton set consisting of the single element 5; to t is assigned the
empty set, and to s the elements x, y, y+1, … , z-1, z.

Set Operations

A relation is a set. It is a set of ordered pairs if it is a binary relation, and it is a set of ordered ***n***-tuples if it is an ***n***-ary relation. Thus all the set operations apply to relations such as ∪, ∩, and complementing.

For example, the union of the "less than" and "equality" relations on the set of integers is the "less than or equal to" relation on the set of integers. The intersection of the "less than" and "less than or equal to" relations on the set of integers is the "less than" relation on the same set. The complement of the "less than" relation on the set of integers is the "greater than or equal to" relation on the same set.

Therefore, the following are the elementary operators are defined on variables of type SET:
* set intersection
+ set union
- set difference
/ symmetric set difference
IN set membership
Constructing the intersection or the union of two sets is often called set multiplication or set addition, respectively; the priorities of the set operators are defined accordingly, with the intersection operator having priority over the union and difference operators, which in turn have priority over the membership operator, which is classified as a relational operator. Following are examples of set expressions and their fully parenthesized equivalents:
r * s + t = (r*s) + t
r - s * t = r - (s*t)
r - s + t = (r-s) + t
THIS IS A TEXT
18
r + s / t = r + (s/t)
x IN s + t = x IN (s+t)

Sets, Elements, and Subsets

One dictionary has, among the many definitions for set, the following: **a number of things naturally connected by location, formation, or order in time**.

Although **set** holds the record for words with the most dictionary definitions, there are terms mathematicians choose to leave undefined, or actually, defined by usage. Set, element, member, and subset are four such terms which will be discussed in today's lesson. Today's activity will also explore the concept.

Each item inside a set is termed an **element**.

The brace symbols { and } are used to enclose the elements in a set.

Each **element** is a **member** of the set (or belongs to the set).

The symbol for membership is $\in$. It can be read "is an element of" and looks quite similar to the Greek letter epsilon (ϵ).

A **subset** is a portion of a set.

The symbol for subset is $\subset$. Some books will allow and use it reversed—we will not.

A **superset** is a set that includes other sets.

For example: If A$\subset$ B, then A is a subset of B and B is a superset of A.

A subset might have no members, in which case it is termed the **null set** or **empty set**.

The empty set is denoted either by {} or by $\emptyset$, a Norwegian letter. The null set is a subset of every set.

Note: a common mistake is to use {$\emptyset$} to denote the null set. This is actually a set with one element and that element is the null set. Since some people slash their zeroes, it is safest when handwriting to **always use** the notation {} to denote the empty or null set.

A **singleton** is a set with only one element.

A subset might contain every member of the original set.
In this case it is termed an **improper subset**.

A **proper subset** does not contain every member of the original set.

Sets may be **finite**, {1, 2, 3,..., 10}, or **infinite**, {1, 2, 3,...}. The **cardinality** of a set A, n(A), is how many elements are in the set. The symbol ... called **ellipses** means to continue in the indicated pattern. There are 2^n subsets of any set, where n is the set's cardinality—check it out for n=3!

The **power set** of a set is the complete set of subsets of the set.

In this class we will consider only safe sets, that is, any set we consider should be **well-defined**. There should be no ambiguity as to whether or not an element belongs to a set. That is why we will avoid things like the village barber who shaves everyone in the village that does not shave himself. This results in a contradiction as to whether or not he shaves himself. Also consider Russell's Paradox: Form the set of sets that are not members of themselves. It is both true and false that this set must contain itself. These are examples of **ill-defined** sets.

Sometimes, instead of listing elements in a set, we use **set builder** notation: {x| x is a letter in the word "mathematics"}. The symbol | can be read as "such that." Sometimes the symbol $\subset$is reserved to mean proper subset and the symbol $\subseteq$is used to allow the inclusion of the improper subset. Compare this with the use of < and $\leq$to exclude or include an endpoint. We will make no such distinction. A set may contain the same elements as another set. Such sets are **equal** or **identical** sets— element order is unimportant. A = B where A = {m,o,r,e} and B = {r,o,m,e}, in general A=B if A$\subset$ B and B$\subset$ A. Sets may be termed **equivalent** if they have the same cardinality. If they are equivalent, a **one-to-one correspondence** can be established between their elements.

The **universal set** is chosen arbitrarily, but must be large enough to include all elements of all sets under discussion.

Complementary set, **A'**, is a set that contains all the elements of the universal set that are not included in A. The symbol ' can be read "prime."

For example: if U={0, 1, 2, 3, 4, 5, 6,...} and A={0, 2, 4, 6, ...}, then A'={1, 3, 5, ...}.

Such paradoxes as those mentioned above, particularily involving infinities (discussed in the next lesson), were well known by the ancient Greeks. During the 19th century, mathematicians were able to tame such paradoxes and about the turn of the 20th century Whitehead and Russell started an ambitious project to carefully codify mathematics. Set theory was developed about this time and serves to unify the many branches of mathematics. Although in 1931 Kurt Gödel showed this approach to be fatally flawed, it is still a good way to explore areas of mathematics such as: arithmetic, number theory, [abstract] algebra, geometry, probability, *etc.*

Geometry has a long history of such systematic study. The ancient Greek Euclid similarily codified the mathematics of his time into 13 books called *The Elements*. Although these books were not limited to Geometry, that is what they are best known for. In fact, up until about my grandfather's day, *The Elements* was the textbook of choice for the study of Geometry! *The Elements* carefully separated the assumptions and definitions from what was to be proved. The concept of proof dates back another couple hundred years to the ancient Greek Pythagoras and his school, the Pythagorean School.

Intersection and Union

Once we have created the concept of a set, we can manipulate sets in useful ways termed **set operations**. Consider the following sets: animals, birds, and white things. Some animals are white: polar bears, mountain goats, big horn sheep, for example. Some birds are white: dove, stork, sea gulls. Some white things are not birds or animal (but birds are animals!): snow, milk, wedding gowns (usually).

The **intersection** of sets are those elements which belong to all intersected sets.

Although we usually intersect only two sets, the definition above is general. The symbol for intersection is ∩.

The **union** of sets are those elements which belong to any set in the union.

Again, although we usually form the union of only two sets, the definition above is general. The symbol for union is ∪.

For the example given above, we can see that:

{white things}∩{birds} = white birds

{white animals}∪{birds} = white animals and all birds

{white birds}⊂{white animals}⊂{animals}

Another name for intersection is **conjuction**. This comes from the fact that an element must be a member of set A **and** set B to be a member of A∩B. Another name for union is **disjunction**. This comes from the fact that an element must be a member of set A **or** set B to be a member of A∪B. Conjunction and disjunction are grammar terms and date back to when Latin was widely used.

I should note the very mathematical use of the word **or** in the sentence above. Common usage now of the word or means one or the other, but not both (excludes both). Mathematicians and computer scientists on the other hand mean one or the other, possibly both (including both). This ambiguity can cause all kinds of problems! Mathematicians term the former exclusive or (EOR or XOR) and the latter inclusive or. We will see ands & ors again in numbers lesson 6 on truth tables.

Representation of Sets

A set *s* is conveniently represented in a computer store by its characteristic function C(s). This is an array of logical values whose ith component has the meaning “i is present in s”. As an example, the set of small
integers s = {2, 3, 5, 7, 11, 13} is represented by the sequence of bits, by a bitstring:
C(s) = (… 0010100010101100)
The representation of sets by their characteristic function has the advantage that the operations of computing the union, intersection, and difference of two sets may be implemented as elementary logical operations. The following equivalences, which hold for all elements i of the base type of the sets x and y, relate logical operations with operations on sets:
i IN (x+y) = (i IN x) OR (i IN y)
i IN (x*y) = (i IN x) & (i IN y)
i IN (x-y) = (i IN x) & ~(i IN y)
These logical operations are available on all digital computers, and moreover they operate concurrently on
all corresponding elements (bits) of a word. It therefore appears that in order to be able to implement the
basic set operations in an efficient manner, sets must be represented in a small, fixed number of words upon
which not only the basic logical operations, but also those of shifting are available. Testing for membership

is then implemented by a single shift and a subsequent (sign) bit test operation. As a consequence, a test of
the form x IN {c1, c2, ... , cn} can be implemented considerably more efficiently than the equivalent
Boolean expression
(x = c1) OR (x = c2) OR ... OR (x = cn)
A corollary is that the set structure should be used only for small integers as elements, the largest one being
the wordlength of the underlying computer (minus 1).

WEEK 5:

String structure.

This week Learning outcomes :
• Define Define string .
•.Explain basic operations of strings.

String Processing

A finite sequence S of zero or more characters is called a string. The number of character in a string is called its length. The string with zero character is called the empty string or the null string. The following are strings of length 9,18, 14 and 0 respectively:

I.) "ND1 CLASS"
II.) "COMPUTER DEPARTMENT"
III.) "CAMPUS SHUTTLE" ,and
IV.) " "

Note that the blank is regarded as a character only when it appears swith other characters .

Concatenation of Strings

Let S1 and S2 be strings, the string consisting of the characters of S1 followed by the characters of S1 is called concatenation of S1 and S1.
This is donated by S1//S2. E.g. "STARLETS"//"//"DEFEAT"//"EA..GLET"
A string Y is called a substring of a string S if there exist strings X and Z such that S=X//Y//Z.
If X is an empty string, then Y is called an intial substing of S, and if Z is an empty string then Y is called a terminal substring of S.
E.g.'BE OR NOT' is a substring of 'TO BE OR NOT TO BE'.
'THE' is an initial substring of 'THE END'.

Length

The general form is LENGTH(string) and this will return the number of character(s) in a giving string.

i.) LENGTH('student')=7
ii.) LENGTH(' ')=0

Insertion

Suppose we want to insert a strings in a given text T so that S starts in position K. We denote this operation as INSERT(text, position, string)
E.g. INSERT('ABCDEFG', 3,'XYZ')='ABXYZCDEFG'
The INSERT function can be implemented by using the string operation as follows:
INSERT(T,K,S)=SUBSTRING(T,1,K-1)//S//SUBSTRING(T,K,LENGTH(T)-K+1).

That is, the initial substring of T before the position K , which has length K-1, is concatenated with the string, and the result is concatenated with the remaining part of T, which begins in position K and has length, LENGTH(T)-(K-1)=LENGTH(T)-K+1.

Deletion

The general form is DELETE(text,position,length).
E.g. DELETE('ABCDEFG',4,2)='ABCFG'
We assume that nothing is deleted if position K=0.
Thus DELETE('ABCDEFG',0,2)='ABCDEFG'
The DELETE function can be implemented using the string operations given as follows:

DELETE(T,K,L)=SUBSTRING(T,1,K-1)//SUBSTRING(T,K+L,LENTGH(T)-K-L+1)

That is the initial substring of T before position K is concatenated with the terminal substring of T beginning in position K+L, and the length of the terminal substring is :
LENGTH(T)-(K+L-1)=LENGTH(T)-K-L+1

When K=0, we assume that DELETE(T,K,L)=T
Suppose that text T and pattern P are given and it is required to delete from T the first occurrence of the pattern P. We can use the following DELETE function

DELETE(T,INDEX(T,P),LENGTH(P))
E.g. = 'ABCDEFG', P='CD', then
DELETE(T,INDEX(T,P),LENGTH(P))=DELETE('ABCDEFG',INDEX('ABCDEFG','CD')
,2)='ABEFG'
Suppose that we want to delete every occurrence of the pattern P in the text T, then we can do this by repeatedly applying DELETE(T,INDEX(T,P),LENGTH(P))
Until INDEX(T,P)=0 that is, until P does not appear in T.
The following algorithm is used to accomplish this:

Algorithm:

A text T and a pattern P are in computer memory. This algorithm deletes every occurrence of P in T.

i.) Find index of P in T. Set K=INDEX(T,P)
ii.) Repeat while K not equal to 0
    a. [Delete P from T.]
       Set T:= DELETE(T,INDEX(T,P),LENGTH(P))

    b. [Update index]
       Set K:= INDEX(T,P)
    [End of loop]
iii.) Write: T.
iv.) Exit.

REPLACEMENT

Suppose in a given T we want to replace the first occurrence of a patter P1 by a pattern P2, we will denote this operation by REPLACE(test,pattern1,pattern2)
E.g. REPLACE('XABYABZ,'AB','C')='XCYABZ'
REPLACE('XABYABZ,'BA','C')='XABYABZ'
In the second case, the pattern BA does not occur, and hence there is no change.
Suppose a text T and patterns P and Q are in the memory of a computer. Suppose we want to replace every occurrence of the pattern P in T by the Pattern Q. This might be accomplished by repeatedly applying REPLACE(T,P.Q), UNTIL(T,P)=0

This could be done using the following algorithm:

1. [Find index of P] Set K :=INDEX(T,P)
2. Repeat while K>0:
   a.) [Replace P by Q] Set T:=REPLACE(T,P,Q)
   b.) [Updarte] Set K:= INDEX(T,P)
   [Eend loop]
3. Write: T
4. Exit.

Exercise

a.) T=XABYABZ
P=AB
Q=C
REPLACE(T,P,Q)

b.) If T=XAB
P=A
Q=AB
REPLACE(T,P,Q)

Use the algorithm above to solve the problems.

WEEK 6:

Queues and Stacks.

This week learning outcomes:
• Queue data structure .
• Stack data structure..

Queues

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 behaviour. A minor modification - adding a tail pointer and adjusting the addition method implementation - will produce a FIFO queue.

Representation of queues:
Two pointer variable namely FRONT and REAR are used in quiues, FRONT contains the location of the front element of the queue, and REAR contains the location of the rear element of the queue. The condition FRONT=NULL will indicate that the queue is empty.
Whenever an item is deleted from the queue, the value of FRONT is increased by 1 , that is , FRONT=FRONT + 1.
Whenever, an item is added to the queue, the value of REAR is increased by 1 that is, REAR=REAR +1.

A B C D …… …. N-1 N

FRONT=1 , and REAR = 4
Suppose that we want to insert an element ITEM into a queue at the time the queue does occupy the last part of the array that is , when REAR=N and the queue is not yet filled. To do this, we can assume that the queue is circular, that is, that QUEUE(1) comes after QUEUE(N) in the array. With this assumption, insert ITEM into the queue by assigning ITEM to QUEUE(1). Instead of increasing REAR to N+1, we reset REAR=1 and then assign QUEUE[REAR]=ITEM.
Similarly, if FRONT=N and element of queue is deleted, we reset FRONT=1 instead of increasing FRONT to N+1.
If the queue is empty, we assign FRONT=REAR=NULL.

EXAMPLE

The following example shows how a queue may be maintained by a circular array QUEUE with N=5 memory locations

a.) Initially empty:FRONT=0

1 2 3 4 5

REAR=0

b.) A and B and C inserted: FRONT= 1

A B C

1 2 3 4 5

REAR=3

c.) A deleted FRONT =2

B C

1 2 3 4 5

REAR=3

Stacks

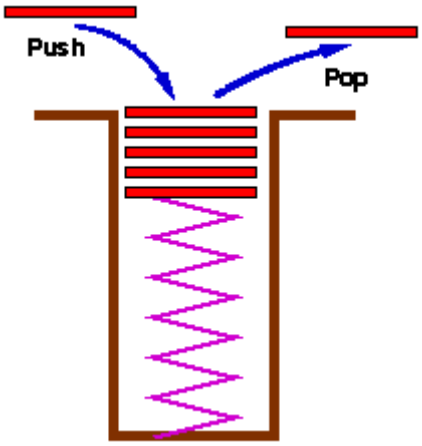
Another way of storing data is in a stack. A stack is a linear structure in which items may be added or removed only at one end for example, a stack of dishes, stack of pennies . Only two (2) operations can be carried out on a stack. A stack is generally implemented with only two principle operations (apart from a constructor and destructor methods):

`Push`	adds an item to a stack
`Pop`	extracts the most recently pushed item from the stack

Other methods such as

`Top`	returns the item at the top *without removing it*
`isempty`	determines whether the stack has anything in it

are sometimes added.



A common model of a stack is a plate or coin stacker. Plates are "pushed" onto to the top and "popped" off the top.

Stacks form Last-In-First-Out (LIFO) queues and have many applications from the parsing of algebraic expressions to ...

A formal specification of a stack class would look like:

```
typedef struct t_stack *stack;

stack ConsStack( int max_items, int item_size );
/* Construct a new stack
   Pre-condition: (max_items > 0) && (item_size > 0)
   Post-condition: returns a pointer to an empty stack
*/

void Push( stack s, void *item );
/* Push an item onto a stack
   Pre-condition: (s is a stack created by a call to ConsStack) &&
                  (existing item count < max_items) &&
                  (item != NULL)
   Post-condition: item has been added to the top of s
*/

void *Pop( stack s );
/* Pop an item of a stack
   Pre-condition: (s is a stack created by a call to
                  ConsStack) &&
                  (existing item count >= 1)
   Post-condition: top item has been removed from s
*/
```

Points to note:

a. A stack is simply another collection of data items and thus it would be possible to use exactly the same specification as the one used for our general collection. However, collections with the LIFO semantics of stacks are so important in computer science that it is appropriate to set up a limited specification appropriate to stacks only.
b. Although a linked list implementation of a stack is possible (adding and deleting from the head of a linked list produces exactly the LIFO semantics of a stack), the most common applications for stacks have a space restraint so that using an array implementation is a

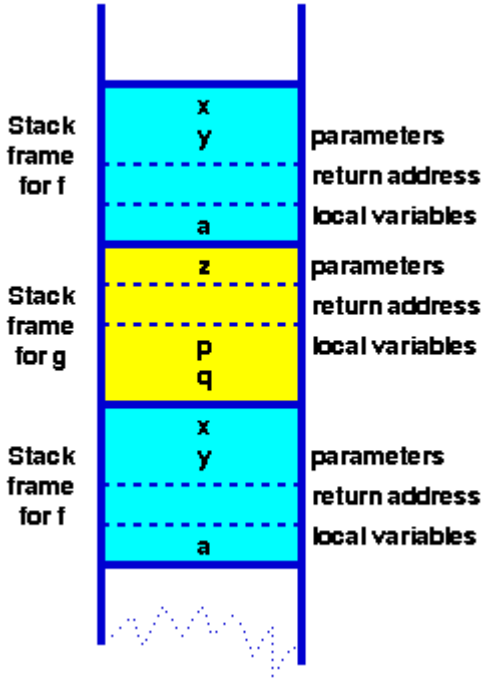
natural and efficient one (In most operating systems, allocation and de-allocation of memory is a relatively expensive operation, there is a penalty for the flexibility of linked tlist implementations.).

Stack Frames

The data structure containing all the data (arguments, local variables, return address, *etc*) needed each time a procedure or function is called.

Almost invariably, programs compiled from modern high level languages (even C!) make use of a stack frame for the working memory of each procedure or function invocation. When any procedure or function is called, a number of words - the stack frame - is pushed onto a program stack. When the procedure or function returns, this frame of data is popped off the stack.

As a function calls another function, first its arguments, then the return address and finally space for local variables is pushed onto the stack. Since each function runs in its own "environment" or **context**, it becomes possible for a function to call itself - a technique known as *recursion*. This capability is extremely useful and extensively used - because many problems are elegantly specified or solved in a recursive way.



Program stack after executing a pair of mutually recursive functions:

```
function f(int x, int y) {
    int a;
    if ( term_cond ) return ...;
    a = .....;
    return g(a);
    }

function g(int z) {
    int p,q;
    p = ...; q = ...;
    return f(p,q);
    }
```

Note how all of function `f` and `g`'s environment (their parameters and local variables) are found in the stack frame. When `f` is called a second time from `g`, a new frame for the second invocation of `f` is created.

Key terms

push, pop

Generic terms for adding something to, or removing something from a stack

context

The environment in which a function executes: includes argument values, local variables and global variables. All the context except the global variables is stored in a stack frame.

A number of programming languages are stack-oriented, meaning they define most basic operations (adding two numbers, printing a character) as taking their arguments from the stack, and placing any return values back on the stack. For example, PostScript has a return stack and an operand stack, and also has a graphics state stack and a dictionary stack.

Forth uses two stacks, one for argument passing and one for subroutine return addresses. The use of a return stack is extremely commonplace, but the somewhat unusual use of an argument stack for a human-readable programming language is the reason Forth is referred to as a *stack-based* language.

Many virtual machines are also stack-oriented, including the p-code machine and the Java virtual machine..

Almost all computer runtime memory environments use a special stack (the "call stack") to hold information about procedure/function calling and nesting in order to switch to the context of the called function and restore to the caller function when the calling finishes. They follow a runtime protocol between caller and callee to save arguments and return value on the stack. Stacks are an important way of supporting nested or recursive function calls. This type of stack is used implicitly by the compiler to support CALL and RETURN statements (or their equivalents) and is not manipulated directly by the programmer.

Some programming languages use the stack to store data that is local to a procedure. Space for local data items is allocated from the stack when the procedure is entered, and is deallocated when the procedure exits. The C programming language is typically implemented in this way. Using the same In computer science, a **stack** is an abstract data type and data structure based on the principle of *Last In First Out (LIFO).* Stacks are used extensively at every level of a modern computer system. For example, a modern PC uses stacks at the architecture level, which are used in the basic design of an operating system for interrupt handling and operating system function calls. Among other uses, stacks are used to run a Java Virtual Machine, and the Java language itself has a class called "Stack", which can be used by the programmer. The stack is ubiquitous.

As an abstract data type, the stack is a container of nodes and has two basic operations: *push* and *pop*. *Push* adds a given node to the top of the stack leaving previous nodes below. *Pop* removes and returns the current top node of the stack. A frequently used metaphor is the idea of a stack of plates in a spring loaded cafeteria stack. In such a stack, only the top plate is visible and accessible to the user, all other plates remain hidden. As new plates are added, each new plate becomes the top of the stack, hiding each plate below, *pushing* the stack of plates down. As the top plate is removed from the stack, they can be used, the plates *pop* back up, and the second plate becomes the top of the stack. Two important principles are illustrated by this metaphor: the Last In First Out principle is one; the second is that the contents of the stack are hidden. Only the top plate is visible, so to see what is on the third plate, the first and second plates will have to be removed. This can also be written as FILO-First In Last Out, i.e. the record inserted first will be popped out at last.

Operations

In modern computer languages, the stack is usually implemented with more operations than just "push" and "pop". The length of a stack can often be returned as a parameter. Another helper operation *top* (also known as *peek* or *peak*) can return the current top element of the stack without removing it from the stack.

This section gives pseudocode for adding or removing nodes from a stack, as well as the length and top functions. Throughout we will use null to refer to an end-of-list marker or sentinel value, which may be implemented in a number of ways using pointers.

```
record Node {
   data // The data being stored in the node
   next // A reference to the next node; null for last node
}
record Stack {
    Node stackPointer   // points to the 'top' node; null for an empty stack
}
function push(Stack stack, Element element) { // push element onto stack
    new(newNode)             // Allocate memory to hold new node
    newNode.data   := element
    newNode.next   := stack.stackPointer
    stack.stackPointer := newNode
}
function pop(Stack stack) { // increase the stack pointer and return 'top'
node data
```

```
    // You could check if stack.stackPointer is null here.
    // If so, you may wish to error, citing the stack underflow.
    node := stack.stackPointer
    stack.stackPointer := node.next
    element := node.data
    return element
}
function top(Stack stack) { // return 'top' node
    return stack.stackPointer.data
}
function length(Stack stack) { // return the amount of nodes in the stack
    length := 0
    node := stack.stackPointer
    while node not null {
        length := length + 1
        node := node.next
    }
    return length
}
```

As you can see, these functions pass the stack and the data elements as parameters and return values, not the data nodes that, in this implementation, include pointers. A stack may also be implemented as a linear section of memory (i.e. an array), in which case the function headers would not change, just the internals of the functions.

Implementation

A typical storage requirement for a stack of *n* elements is *O(n)*. The typical time requirement of O(1) operations is also easy to satisfy with a dynamic array or (singly) linked list implementation.

C++'s Standard Template Library provides a "`stack`" templated class which is restricted to only push/pop operations. Java's library contains a `Stack` class that is a specialization of `Vector`. This could be considered a design flaw because the inherited get() method from `Vector` ignores the LIFO constraint of the `Stack`.

Here is a simple example of a stack with the operations described above (but no error checking) in Python.

```
class Stack(object):
    def __init__(self):
        self.stack_pointer = None

    def push(self, element):
```

```
        self.stack_pointer = Node(element, self.stack_pointer)

    def pop(self):
        e = self.stack_pointer.element
        self.stack_pointer = self.stack_pointer.next
        return e

    def peek(self):
        return self.stack_pointer.element

    def __len__(self):
        i = 0
        sp = self.stack_pointer
        while sp:
            i += 1
            sp = sp.next
        return i

class Node(object):
    def __init__(self, element=None, next=None):
        self.element = element
        self.next = next

if __name__ == '__main__':
    # small use example
    s = Stack()
    [s.push(i) for i in xrange(10)]
    print [s.pop() for i in xrange(len(s))]
```

The above is admittedly redundant as Python supports the 'pop' and 'append' functions to lists.

Applications

Stacks are ubiquitous in the computing world.

Expression evaluation and syntax parsing

Calculators employing reverse Polish notation use a stack structure to hold values. Expressions can be represented in prefix, postfix or infix notations. Conversion from one form of the expression to another form needs a stack. Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code. Most of the programming languages are context-free languages allowing them to be parsed with stack based machines.

For example, The calculation: ((1 + 2) * 4) + 3 can be written down like this in postfix notation with the advantage of no precedence rules and parentheses needed:

```
1 2 + 4 * 3 +
```

The expression is evaluated from the left to right using a stack:

- push when encountering an operand and
- pop two operands and evaluate the value when encountering an operation.
- push the result

Like the following way (the *Stack* is displayed after *Operation* has taken place):

Input	Operation	Stack
1	Push operand	1
2	Push operand	1, 2
+	Add	3
4	Push operand	3, 4
*	Multiply	12
3	Push operand	12, 3
+	Add	15

The final result, 15, lies on the top of the stack at the end of the calculation.

example : implementation in pascal. using marked sequential file as data archives.

```
{
programmer : clx321
file  : stack.pas
unit  : Pstack.tpu
}
program TestStack;
{this program use ADT of Stack, i will assume that the unit of ADT of Stack
has already existed}

uses
   PStack;   {ADT of STACK}
```

```
{dictionary}
const
   mark = '.';
var
   data : stack;
   f : text;
   cc : char;
   ccInt, cc1, cc2 : integer;

  {functions}
  IsOperand (cc : char) : boolean;    {JUST  Prototype}
    {return TRUE if cc is operand}
  ChrToInt (cc : char) : integer;     {JUST Prototype}
    {change char to integer}
  Operator (cc1, cc2 : integer) : integer;     {JUST Prototype}
    {operate two operands}

{algorithms}
begin
  assign (f, cc);
  reset (f);
  read (f, cc);  {first elmt}
  if (cc = mark) then
     begin
        writeln ('empty archives !');
     end
  else
     begin
        repeat
          if (IsOperand (cc)) then
             begin
               ccInt := ChrToInt (cc);
               push (ccInt, data);
             end
          else
             begin
               pop (cc1, data);
               pop (cc2, data);
               push (data, Operator (cc2, cc1));
             end;
           read (f, cc);   {next elmt}
        until (cc = mark);
     end;
  close (f);
end.
```

Runtime stack for both data and procedure calls has important security implications (see below) of which a programmer must be aware in order to avoid introducing serious security bugs into a program.

Security

Some computing environments use stacks in ways that may make them vulnerable to security breaches and attacks. Programmers working in such environments must take special care to avoid the pitfalls of these implementations.

For example, some programming languages use a common stack to store both data local to a called procedure and the linking information that allows the procedure to return to its caller. This means that the program moves data into and out of the same stack that contains critical return addresses for the procedure calls. If data is moved to the wrong location on the stack, or an oversized data item is moved to a stack location that is not large enough to contain it, return information for procedure calls may be corrupted, causing the program to fail.

Malicious parties may attempt to take advantage of this type of implementation by providing oversized data input to a program that does not check the length of input. Such a program may copy the data in its entirety to a location on the stack, and in so doing it may change the return addresses for procedures that have called it. An attacker can experiment to find a specific type of data that can be provided to such a program such that the return address of the current procedure is reset to point to an area within the stack itself (and within the data provided by the attacker), which in turn contains instructions that carry out unauthorized operations.

This type of attack is a variation on the buffer overflow attack and is an extremely frequent source of security breaches in software, mainly because some of the most popular programming languages (such as C) use a shared stack for both data and procedure calls, and do not verify the length of data items. Frequently programmers do not write code to verify the size of data items, either, and when an oversized or undersized data item is copied to the stack, a security breach may occur.

WEEK 7 :

Properties linear Array.

These weeks Learning outcomes :
•Define linear Array.
•Discuss various operations that can be performed on ordered list.

Linear Arrays

Data structures are classified as either linear or nonlinear. It is said to be linear if its elements form a sequence , that is, a linear list, otherwise nonlinear for example, trees and graphs , records .They are mainly used to represent data containing a hierarchical relationship between elements.. Strings, array lists, and queues are linear types of data structure.The operations normally performed on linear lists include:

a.) Traversal: processing each element in the list.
b.) Search: finding the location of the element with a given value or the record with a given key.
c.) Insertion: adding a new element to the list.
d.) Deletion: removing an element from the list.
e.) Sorting: arranging the elements in some type of order.
f.) Merging: combining two lists into a single list.

A linear array is a list of a finite number, n, of homogeneous data elements, where the number n of elements is called the length or size of the array. The elements array A , may be denoted as follws:
A1, A2,……..An or A(1), A(2),……A(n) or A[1],A[2],……A[n]
Where 1 is the the lower bound, LB of the array and n, the upperbound, UB of the array.

Example:
Let DATA be a 5-element linear array of integers such that DATA[1]=24, DATA[2]=56. DATA[3]=405. DATA[4]=35, DATA[5]=87

The array DATA is fryquently pictured as either of the following:
DATA
DATA[1] 24

DATA[2] 56

DATA[3] 405

DATA[4] 35

DATA[5] 87

24	56	405	35	87
1	2	3	4	5

DATA

MULTIDIMENTIONAL ARRAYS

The linear arrays earlier discussed are also one-dimensional arrays, since each element in the array is referenced by a single subscript. Most programming languages allow 2-dimentional and 3-dimentional arrays, some allow the number of dimensions for an array to be as high as 7.

Two-dimensionnal arrays are called Matrices in mathematics and tables in business applications.

A 2-dimensional m*n array is a collection of m.n data elements such that each Element is specified by a pair of integers (such as J,K), called subscripts, with the property that :

$$1 \leq J \leq m \text{ ,and } 1 \leq K \leq n$$

e.g. the element of A with first subscript J and second subscript K will be denoted by A(j.k) or A[j, k]

Example
Suppose each student in a class of 10 students in a given 3 tests. Assuming the students are numbered according, the test scores can be assigned to a 10 *3 matrix array SCORE. Thus, SCORE[K,L] contains the Kth student's score on Lth test.

This can be represented as follows:

Student	Test 1	Test 2	Test 3
1	56	46	90
2	78	90	98
3	.	.	.
4	.	.	.
5	.	.	.
6	.	.	.

7	.	.	.
8	.	.	.
9	.	.	.
10	78	82	85

The Array Structure

The array is probably the most widely used data structure; in some languages it is even the only one
available. An array consists of components which are all of the same type, called its *base type*; it is
therefore called a *homogeneous* structure. The array is a *random-access* structure, because all components
can be selected at random and are equally quickly accessible. In order to denote an individual component,
the name of the entire structure is augmented by the *index* selecting the component. This index is to be an
integer between 0 and n-1, where n is the number of elements, the *size*, of the array.

TYPE T = ARRAY n OF T0

Examples

TYPE Row = ARRAY 4 OF REAL
TYPE Card = ARRAY 80 OF CHAR
TYPE Name = ARRAY 32 OF CHAR

A particular value of a variable

VAR x: Row

with all components satisfying the equation xi = 2-i, may be visualized as shown in Fig. 1.2.

Fig. 1.2 Array of type Row with xi = 2-i

An individual component of an array can be selected by an *index*. Given an array variable x, we denote an array selector by the array name followed by the respective component's index i, and we write xi or x[i].

Because of the first, conventional notation, a component of an array component is therefore also called a *subscripted* variable.

The common way of operating with arrays, particularly with large arrays, is to selectively update single components rather than to construct entirely new structured values. This is expressed by considering an
array variable as an array of component variables and by permitting assignments to selected components, such as for example x[i] := 0.125. Although selective updating causes only a single component value to change, from a conceptual point of view we must regard the entire composite value as having changed too.

The fact that array indices, i.e., names of array components, are integers, has a most important

consequence: indices may be computed. A general index expression may be substituted in place of an index constant; this expression is to be evaluated, and the result identifies the selected component. This
generality not only provides a most significant and powerful programming facility, but at the same time it also gives rise to one of the most frequently encountered programming mistakes: The resulting value may be outside the interval specified as the range of indices of the array. We will assume that decent computing systems provide a warning in the case of such a mistaken access to a non-existent array component.
The cardinality of a structured type, i. e. the number of values belonging to this type, is the product of the cardinality of its components. Since all components of an array type T are of the same base type T0, we obtain
card(T) = card(T0)n
x0 1.0
x1 0.5
x2 0.25
x3 0.125
19
Constituents of array types may themselves be structured. An array variable whose components are again
arrays is called a *matrix*. For example,
M: ARRAY 10 OF Row
is an array consisting of ten components (rows), each constisting of four components of type REAL, and is
called a 10 × 4 matrix with real components. Selectors may be concatenated accordingly, such that Mij and M[i][j] denote the j th component of row Mi, which is the i th component of M. This is usually abbreviated as M[i, j] and in the same spirit the declaration
M: ARRAY 10 OF ARRAY 4 OF REAL
can be written more concisely as M: ARRAY 10, 4 OF REAL.
If a certain operation has to be performed on all components of an array or on adjacent components of a section of the array, then this fact may conveniently be emphasized by using the FOR satement, as shown in the following examples for computing the sum and for finding the maximal element of an array declared as

VAR a: ARRAY N OF INTEGER
sum := 0;
FOR i := 0 TO N-1 DO sum := a[i] + sum END
k := 0; max := a[0];
FOR i := 1 TO N-1 DO
IF max < a[i] THEN k := i; max := a[k] END
END.
In a further example, assume that a fraction f is represented in its decimal form with k-1 digits, i.e., by an
array d such that

f = **S** i : 0 ≤i < k: di * 10-i or

f = d0 + 10*d1 + 100*d2 + … + dk-1*10k-1

Now assume that we wish to divide f by 2. This is done by repeating the familiar division operation for all
k-1 digits di, starting with i=1. It consists of dividing each digit by 2 taking into account a possible carry
from the previous position, and of retaining a possible remainder r for the next position:
r := 10*r +d[i]; d[i] := r DIV 2; r := r MOD 2
This algorithm is used to compute a table of negative powers of 2. The repetition of halving to compute 2-1,
2-2, ... , 2-N is again appropriately expressed by a FOR statement, thus leading to a nesting of two FOR
statements.

```
PROCEDURE Power(VAR W: Texts.Writer; N: INTEGER);
(*compute decimal representation of negative powers of 2*)
VAR i, k, r: INTEGER;
d: ARRAY N OF INTEGER;
BEGIN
FOR k := 0 TO N-1 DO
Texts.Write(W, "."); r := 0;
FOR i := 0 TO k-1 DO
r := 10*r + d[i]; d[i] := r DIV 2; r := r MOD 2;
Texts.Write(W, CHR(d[i] + ORD("0")))
END ;
d[k] := 5; Texts.Write(W, "5"); Texts.WriteLn(W)
END
END Power.
```

The resulting output text for N = 10 is

```
20
.5
.25
.125
.0625
.03125
.015625
.0078125
.00390625
.001953125
.0009765625
```

A representation of an array structure is a mapping of the (abstract) array with components of type T onto the store which is an array with components of type BYTE. The array should be mapped in such a way that the computation of addresses of array components is as simple (and therefore as efficient) as possible. The address i of the j-th array component is computed by the linear mapping function i = i0 + j*s
where i0 is the address of the first component, and s is the number of words that a component occupies.

Assuming that the word is the smallest individually transferable unit of store, it is evidently highly desirable that s be a whole number, the simplest case being s = 1. If s is not a whole number (and this is the normal case), then s is usually rounded up to the next larger integer S. Each array component then occupies
S words, whereby S-s words are left unused (see Figs. 1.5 and 1.6). Rounding up of the number of words
needed to the next whole number is called *padding*. The storage utilization factor u is the quotient of the
minimal amounts of storage needed to represent a structure and of the amount actually used:
u = s / (s rounded up to nearest integer)

WEEK 8 :

Linked list.

- Define linked list .
- Define linked list and compare it with linear list.
- Discuss the advantages and disadvantages of linked list.

Lists

Linked list is an algorithm for storing a list of items. It is made of any number of pieces of memory (nodes) and each node contains whatever data you are storing along with a pointer (a link) to another node. By locating the node referenced by that pointer and then doing the same with the pointer in that new node and so on, you can traverse the entire list.

Because a linked list stores a list of items, it has some similarities to an array. But the two are implemented quite differently. An array is a single piece of memory while a linked list contains as many pieces of memory as there are items in the list. Obviously, if your links get messed up, you not only lose part of the list, but you ill lose any reference to those items no longer included in the list (unless you store another pointer to those items somewhere).

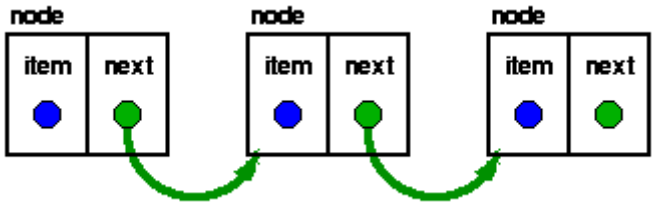
Some advantages that a linked list has over an array are that you can quickly insert and delete items in a linked list. Inserting and deleting items in an array requires you to either make room for new items or fill the "hole" left by deleting an item. With a linked list, you imply rearrange those pointers that are affected by the change. linked lists also allow you to have different-sized nodes in the list. Some disadvantages to linked lists include that hey are quite difficult to sort. Also, you cannot immediately locate, say, the hundredth element in a linked list the way you can in an array. Instead, you must traverse the list until you've found the hundredth element.

Again, the array implementation of our collection has one serious drawback: you must know the maximum number of items in your collection when you create it. This presents problems in programs in which this maximum number cannot be predicted accurately when the program starts up. Fortunately, we can use a structure called a linked list to overcome this limitation.

Linked lists

The linked list is a very flexible **dynamic data structure that is** structure which grows or shrinks as the data they hold changes. Lists, stacks and trees are all dynamic structures.: items may be added to it or deleted from it at will. A programmer need not worry about how many items a program will have to accommodate: this allows us to write robust programs which require much less maintenance. A very common source of problems in program maintenance is the need to increase the capacity of a program to handle larger collections: even the most generous allowance for growth tends to prove inadequate over time!

In a linked list, each item is allocated space as it is added to the list. A link is kept with each item to the next item in the list.



Each node of the list has two elements

1. the item being stored in the list *and*
2. a pointer to the next item in the list

The last node in the list contains a NULL pointer to indicate that it is the end or *tail* of the list.

As items are added to a list, memory for a node is dynamically allocated. Thus the number of items that may be added to a list is limited only by the amount of memory available.

Handle for the list

The variable (or handle) which represents the list is simply a pointer to the node at the *head* of the list.

Adding to a list

The simplest strategy for adding an item to a list is to:

a. allocate space for a new node,
b. copy the item into it,
c. make the new node's `next` pointer point to the current head of the list *and*
d. 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.

An alternative is to create a structure for the list which contains both head and tail pointers:

```
struct fifo_list {
        struct node *head;
        struct node *tail;
        };
```

The code for `AddToCollection` is now trivially modified to make a list in which the item most recently added to the list is the list's tail.

The specification remains identical to that used for the array implementation: the `max_item` parameter to `ConsCollection` is simply ignored .

Thus we only need to change the implementation. As a consequence, applications which use this object will need no changes. The ramifications for the cost of software maintenance are significant.

The data structure is changed, but since the details (the attributes of the object or the elements of the structure) are hidden from the user, there is no impact on the user's program.

Points to note:

a. This implementation of our collection can be substituted for the first one with no changes to a client's program. With the exception of the added flexibility that any number of items may be added to our collection, this implementation provides exactly the same high level behaviour as the previous one.
b. The linked list implementation has exchanged flexibility for efficiency - on most systems, the system call to allocate memory is relatively expensive. Pre-allocation in the array-based implementation is generally more efficient. More examples of such trade-offs will be found later.

The study of data structures and algorithms will enable you to make the implementation decision which most closely matches your users' specifications.

Advantages of Linked List over its array counterpart

Some advantages that a linked list has over an array are as follows:

i.) that you can quickly insert and delete items in a linked list.

ii.) Inserting and deleting items in an array requires you to either make room for new items or fill the "hole" left by deleting an item.

iii.) With a linked list, you imply rearrange those pointers that are affected by the change.

iv.) linked lists also allow you to have different-sized nodes in the list.

Some disadvantages of linked lists include that :

i.) They are quite difficult to sort.

ii.) Also, you cannot immediately locate, say, the hundredth element in a linked list the way you can in an array. Instead, you must traverse the list until you've found the hundredth element.

WEEK 9 :

Properties of linked list.

This week learning outcomes:

• Explain types of linked list.

•Applications of linked lists.

• Implementation of different operations of linked lists.

Types of linked lists

Linearly linked list

Singly-linked list

The simplest kind of linked list is a **singly-linked list** (or **slist** for short), which has one link per node. This link points to the next node in the list, or to a null value or empty list if it is the final node.

A singly-linked list containing two values: the value of the current node and a link to the next node

A singly linked list's node is divided into two parts. The first part holds or points to information about the node, and second part holds the address of next node. A singly linked list travels one way.

Doubly-linked list

A more sophisticated kind of linked list is a **doubly-linked list** or **two-way linked list**. Each node has two links: one points to the previous node, or points to a null value or empty list if it is the first node; and one points to the next, or points to a null value or empty list if it is the final node. *A doubly-linked list containing three integer values: the value, the link forward to the next node, and the link backward to the previous node*

In some very low level languages, XOR-linking offers a way to implement doubly-linked lists using a single word for both links, although the use of this technique is usually discouraged.

Circularly-linked list

In a **circularly-linked list**, the first and final nodes are linked together. This can be done for both singly and doubly linked lists. To traverse a circular linked list, you begin at any node and follow the list in either direction until you return to the original node. Viewed another way, circularly-linked lists can be seen as having no beginning or end. This type of list is most useful for managing buffers for data ingest, and in cases where you have one object in a list and wish to iterate through all other objects in the list in no particular order.

The pointer pointing to the whole list may be called the access pointer.*A circularly-linked list containing three integer values*

Sentinel nodes

Linked lists sometimes have a special *dummy* or sentinel node at the beginning and/or at the end of the list, which is not used to store data. Its purpose is to simplify or speed up some operations, by ensuring that every data node always has a previous and/or next node, and that every list (even one that contains no data elements) always has a "first" and "last" node. Lisp has such a design - the special value nil is used to mark the end of a 'proper' singly-linked list, or chain of cons cells as they are called. A list does not have to end in nil, but a list that did not would be termed 'improper'.

Applications of linked lists

Linked lists are used as a building block for many other data structures, such as stacks, queues and their variations.

The "data" field of a node can be another linked list. By this device, one can construct many linked data structures with lists; this practice originated in the Lisp programming language, where linked lists are a primary (though by no means the only) data structure, and is now a common feature of the functional programming style.

Sometimes, linked lists are used to implement associative arrays, and are in this context called **association lists**. There is very little good to be said about this use of linked lists; they are easily outperformed by other data structures such as self-balancing binary search trees even on small data sets (see the discussion in associative array). However, sometimes a linked list is dynamically created out of a subset of nodes in such a tree, and used to more efficiently traverse that set.

Linked lists vs. arrays

	Array	Linked list
Indexing	O(1)	O(*n*)
Inserting / Deleting at end	O(1)	O(1) or O(*n*)[2]
Inserting / Deleting in middle (with iterator)	O(*n*)	O(1)
Persistent	No	Singly yes
Locality	Great	Bad

Linked lists have several advantages over arrays. Elements can be inserted into linked lists indefinitely, while an array will eventually either fill up or need to be resized, an expensive operation that may not even be possible if memory is fragmented. Similarly, an array from which many elements are removed may become wastefully empty or need to be made smaller.

Further memory savings can be achieved, in certain cases, by sharing the same "tail" of elements among two or more lists — that is, the lists end in the same sequence of elements. In this way, one can add new elements to the front of the list while keeping a reference to both the new and the old versions — a simple example of a persistent data structure.

On the other hand, arrays allow random access, while linked lists allow only sequential access to elements. Singly-linked lists, in fact, can only be traversed in one direction. This makes linked lists unsuitable for applications where it's useful to look up an element by its index quickly, such as heapsort. Sequential access on arrays is also faster than on linked lists on many machines due to locality of reference and data caches. Linked lists receive almost no benefit from the cache.

Another disadvantage of linked lists is the extra storage needed for references, which often makes them impractical for lists of small data items such as characters or boolean values. It can also be slow, and with a naïve allocator, wasteful, to allocate memory separately for each new element, a problem generally solved using memory pools.

A number of linked list variants exist that aim to ameliorate some of the above problems. Unrolled linked lists store several elements in each list node, increasing cache performance while decreasing memory overhead for references. CDR coding does both these as well, by replacing references with the actual data referenced, which extends off the end of the referencing record.

A good example that highlights the pros and cons of using arrays vs. linked lists is by implementing a program that resolves the Josephus problem. The Josephus problem is an election method that works by having a group of people stand in a circle. Starting at a predetermined person, you count around the circle *n* times. Once you reach the *n*th person, take them out of the circle and have the members close the circle. Then count around the circle the same *n* times and repeat the process, until only one person is left. That person wins the election. This shows the strengths and weaknesses of a linked list vs. an array, because if you view the people as connected nodes in a circular linked list then it shows how easily the linked list is able to delete nodes (as it only has to rearrange the links to the different nodes). However, the linked list will be poor at finding the next person to remove and will need to recurse through the list until it finds that person. An array, on the other hand, will be poor at deleting nodes (or elements) as it cannot remove one node without individually shifting all the elements up the list by one. However, it is exceptionally easy to find the *n*th person in the circle by directly referencing them by their position in the array.

The list ranking problem concerns the efficient conversion of a linked list representation into an array. Although trivial for a conventional computer, solving this problem by a parallel algorithm is complicated and has been the subject of much research.

Doubly-linked vs. singly-linked

Double-linked lists require more space per node (unless one uses xor-linking), and their elementary operations are more expensive; but they are often easier to manipulate because they allow sequential access to the list in both directions. In particular, one can insert or delete a node in a constant number of operations given only that node's address. Comparing with singly-linked lists, it requires the *previous* node's address in order to correctly insert or delete. Some algorithms require access in both directions. On the other hand, they do not allow tail-sharing, and cannot be used as persistent data structures.

Circularly-linked vs. linearly-linked

Circular linked lists are most useful for describing naturally circular structures, and have the advantage of regular structure and being able to traverse the list starting at any point. They also allow quick access to the first and last records through a single pointer (the address of the last element). Their main disadvantage is the complexity of iteration, which has subtle special cases.

Sentinel nodes (header nodes)

Doubly linked lists can be structured without using a front and NULL pointer to the ends of the list. Instead, a node of object type T set with specified default values is used to indicate the "beginning" of the list. This node is known as a Sentinel node and is commonly referred to as a "header" node. Common searching and sorting algorithms are made less complicated through the use of a header node, as every element now points to another element, and never to NULL. The header node, like any other, contains a "next" pointer that points to what is considered by the linked list to be the first element. It also contains a "previous" pointer which points to the last element in the linked list. In this way, a doubly linked list structured around a Sentinel Node is circular.

The Sentinel node is defined as another node in a doubly linked list would be, but the allocation of a front pointer is unnecessary as the next and previous pointers of the Sentinel node will point to itself. This is defined in the default constructor of the list.

next == this; prev == this;

If the previous and next pointers point to the Sentinel node, the list is considered empty. Otherwise, if one or more elements is added, both pointers will point to another node, and the list will contain those elements. [3]

Sentinel node may simplify certain list operations, by ensuring that the next and/or previous nodes exist for every element. However sentinel nodes use up extra space (especially in applications that use many short lists), and they may complicate other operations. To avoid the extra space requirement the sentinel nodes can often be reused as references to the first and/or last node of the list.

The Sentinel node eliminates the need to keep track of a pointer to the beginning of the list, and also eliminates any errors that could result in the deletion of the first pointer, or any accidental relocation.

Linked list operations

When manipulating linked lists in-place, care must be taken to not use values that you have invalidated in previous assignments. This makes algorithms for inserting or deleting linked list nodes somewhat subtle. This section gives pseudocode for adding or removing nodes from singly, doubly, and circularly linked lists in-place. Throughout we will use *null* to refer to an end-of-list marker or sentinel, which may be implemented in a number of ways.

Linearly-linked lists

Singly-linked lists

Our node data structure will have two fields. We also keep a variable *firstNode* which always points to the first node in the list, or is *null* for an empty list.

```
record Node {
   data // The data being stored in the node
   next // A reference to the next node, null for last node
}
record List {
    Node firstNode   // points to first node of list; null for empty list
}
```

Traversal of a singly-linked list is simple, beginning at the first node and following each *next* link until we come to the end:

```
node := list.firstNode
while node not null {
    (do something with node.data)
    node := node.next
}
```

The following code inserts a node after an existing node in a singly linked list. The diagram shows how it works. Inserting a node before an existing one cannot be done; instead, you have to locate it while keeping track of the previous node.

```
function insertAfter(Node node, Node newNode) { // insert newNode after node
    newNode.next := node.next
    node.next    := newNode
}
```

Inserting at the beginning of the list requires a separate function. This requires updating *firstNode*.

```
function insertBeginning(List list, Node newNode) { // insert node before
current first node
    newNode.next   := list.firstNode
    list.firstNode := newNode
}
```

Similarly, we have functions for removing the node *after* a given node, and for removing a node from the beginning of the list. The diagram demonstrates the former. To find and remove a particular node, one must again keep track of the previous element.

```
function removeAfter(node node) { // remove node past this one
    obsoleteNode := node.next
    node.next := node.next.next
    destroy obsoleteNode
}
function removeBeginning(List list) { // remove first node
    obsoleteNode := list.firstNode
```

```
    list.firstNode := list.firstNode.next          // point past deleted
node
    destroy obsoleteNode
 }
```

Notice that removeBeginning() sets *list.firstNode* to *null* when removing the last node in the list.

Since we can't iterate backwards, efficient "insertBefore" or "removeBefore" operations are not possible.

Appending one linked list to another can be inefficient unless a reference to the tail is kept as part of the List structure, because we must traverse the entire first list in order to find the tail, and then append the second list to this. Thus, if two linearly-linked lists are each of length *n*, list appending has asymptotic time complexity of $O(n)$. In the Lisp family of languages, list appending is provided by the `append` procedure.

Many of the special cases of linked list operations can be eliminated by including a dummy element at the front of the list. This ensures that there are no special cases for the beginning of the list and renders both insertBeginning() and removeBeginning() unnecessary. In this case, the first useful data in the list will be found at list.firstNode.next.

Doubly-linked lists

With doubly-linked lists there are even more pointers to update, but also less information is needed, since we can use backwards pointers to observe preceding elements in the list. This enables new operations, and eliminates special-case functions. We will add a *prev* field to our nodes, pointing to the previous element, and a *lastNode* field to our list structure which always points to the last node in the list. Both *list.firstNode* and *list.lastNode* are *null* for an empty list.

```
record Node {
   data // The data being stored in the node
   next // A reference to the next node; null for last node
   prev // A reference to the previous node; null for first node
}
record List {
    Node firstNode   // points to first node of list; null for empty list
    Node lastNode    // points to last node of list; null for empty list
}
```

Iterating through a doubly linked list can be done in either direction. In fact, direction can change many times, if desired.

Forwards

```
node := list.firstNode
while node ≠ null
    <do something with node.data>
    node := node.next
```

Backwards

```
node := list.lastNode
while node ≠ null
    <do something with node.data>
    node := node.prev
```

These symmetric functions add a node either after or before a given node, with the diagram demonstrating after:

```
function insertAfter(List list, Node node, Node newNode)
    newNode.prev := node
    newNode.next := node.next
    if node.next = null
        list.lastNode := newNode
    else
        node.next.prev := newNode
    node.next := newNode
function insertBefore(List list, Node node, Node newNode)
    newNode.prev := node.prev
    newNode.next := node
    if node.prev is null
        list.firstNode := newNode
    else
        node.prev.next := newNode
    node.prev    := newNode
```

We also need a function to insert a node at the beginning of a possibly-empty list:

```
function insertBeginning(List list, Node newNode)
    if list.firstNode = null
        list.firstNode := newNode
        list.lastNode  := newNode
        newNode.prev := null
        newNode.next := null
    else
        insertBefore(list, list.firstNode, newNode)
```

A symmetric function inserts at the end:

```
function insertEnd(List list, Node newNode)
    if list.lastNode = null
        insertBeginning(list, newNode)
    else
        insertAfter(list, list.lastNode, newNode)
```

Removing a node is easier, only requiring care with the *firstNode* and *lastNode*:

```
function remove(List list, Node node)
  if node.prev = null
      list.firstNode := node.next
      if node.next != null
          node.next.prev := null
  else
      node.prev.next := node.next
  if node.next = null
      list.lastNode := node.prev
      if node.prev != null
          node.prev.next := null
  else
      node.next.prev := node.prev
  destroy node
```

One subtle consequence of this procedure is that deleting the last element of a list sets both *firstNode* and *lastNode* to *null*, and so it handles removing the last node from a one-element list correctly. Notice that we also don't need separate "removeBefore" or "removeAfter" methods, because in a doubly-linked list we can just use "remove(node.prev)" or "remove(node.next)" where these are valid.

Circularly-linked list

Circularly-linked lists can be either singly or doubly linked. In a circularly linked list, all nodes are linked in a continuous circle, without using *null.* For lists with a front and a back (such as a queue), one stores a reference to the last node in the list. The *next* node after the last node is the first node. Elements can be added to the back of the list and removed from the front in constant time.

Both types of circularly-linked lists benefit from the ability to traverse the full list beginning at any given node. This often allows us to avoid storing *firstNode* and *lastNode*, although if the list may be empty we need a special representation for the empty list, such as a *lastNode* variable which points to some node in the list or is *null* if it's empty; we use such a *lastNode* here. This

representation significantly simplifies adding and removing nodes with a non-empty list, but empty lists are then a special case.

Doubly-circularly-linked lists

Assuming that *someNode* is some node in a non-empty list, this code iterates through that list starting with *someNode* (any node will do):

Forwards

```
node := someNode
do
    do something with node.value
    node := node.next
while node ≠ someNode
```

Backwards

```
node := someNode
do
    do something with node.value
    node := node.prev
while node ≠ someNode
```

Notice the postponing of the test to the end of the loop. This is important for the case where the list contains only the single node *someNode*.

This simple function inserts a node into a doubly-linked circularly-linked list after a given element:

```
function insertAfter(Node node, Node newNode)
    newNode.next := node.next
    newNode.prev := node
    node.next.prev := newNode
    node.next      := newNode
```

To do an "insertBefore", we can simply "insertAfter(node.prev, newNode)". Inserting an element in a possibly empty list requires a special function:

```
function insertEnd(List list, Node node)
    if list.lastNode = null
        node.prev := node
        node.next := node
    else
```

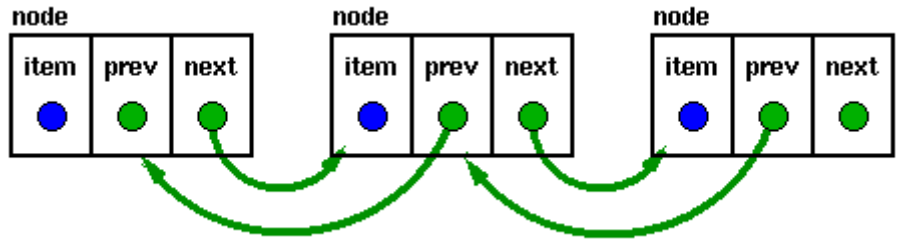
```
        insertAfter(list.lastNode, node)
    list.lastNode := node
```

To insert at the beginning we simply "insertAfter(list.lastNode, node)". Finally, removing a node must deal with the case where the list empties:

```
function remove(List list, Node node)
    if node.next = node
        list.lastNode := null
    else
        node.next.prev := node.prev
        node.prev.next := node.next
        if node = list.lastNode
            list.lastNode := node.prev;
    destroy node
```

As in doubly-linked lists, "removeAfter" and "removeBefore" can be implemented with "remove(list, node.prev)" and "remove(list, node.next)".

Doubly Linked Lists



Doubly linked lists have a pointer to the preceding item as well as one to the next.

They permit scanning or searching of the list in both directions. (To go backwards in a simple list, it is necessary to go back to the start and scan forwards.) Many applications require searching backwards and forwards through sections of a list: for example, searching for a common name like "Kim" in a Korean telephone directory would probably need much scanning backwards and forwards through a small region of the whole list, so the backward links become very useful. In this case, the node structure is altered to have two links:

```
struct t_node {
     void *item;
     struct t_node *previous;
     struct t_node *next;
     } node;
```

WEEK 10:

Non- Linear structures.

This week learning outcomes :
• Define a tree .
• State properties of tree.
• Describe different types of tree.(General tree , binary tree)
• Explain binary tree reprentation.

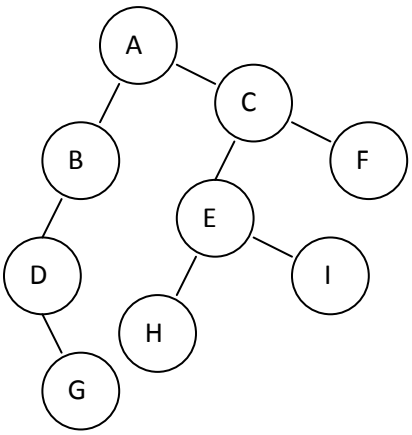
Tree Structures

Basic Concepts and Definitions

Srings, arrays, and queues are linear types of data structure. However, tree is a nonlinear data structure is mainly used to represent data containing a hierarchical relationship between elements . Examples of this feature include records, family tree and table of contents.

Trees

```
void BST :: clearhelp (NodePtr *P)

{

        if (P == NULL) return;

        clearhelp (P → Left)

        clearhelp (P → Right);

        delete P;

}
```



balance ≅ {left – right}

Almost perfect ≅ balanced

In databases, you use this type of Array based implementation.

Height-balanced trees = AVL trees

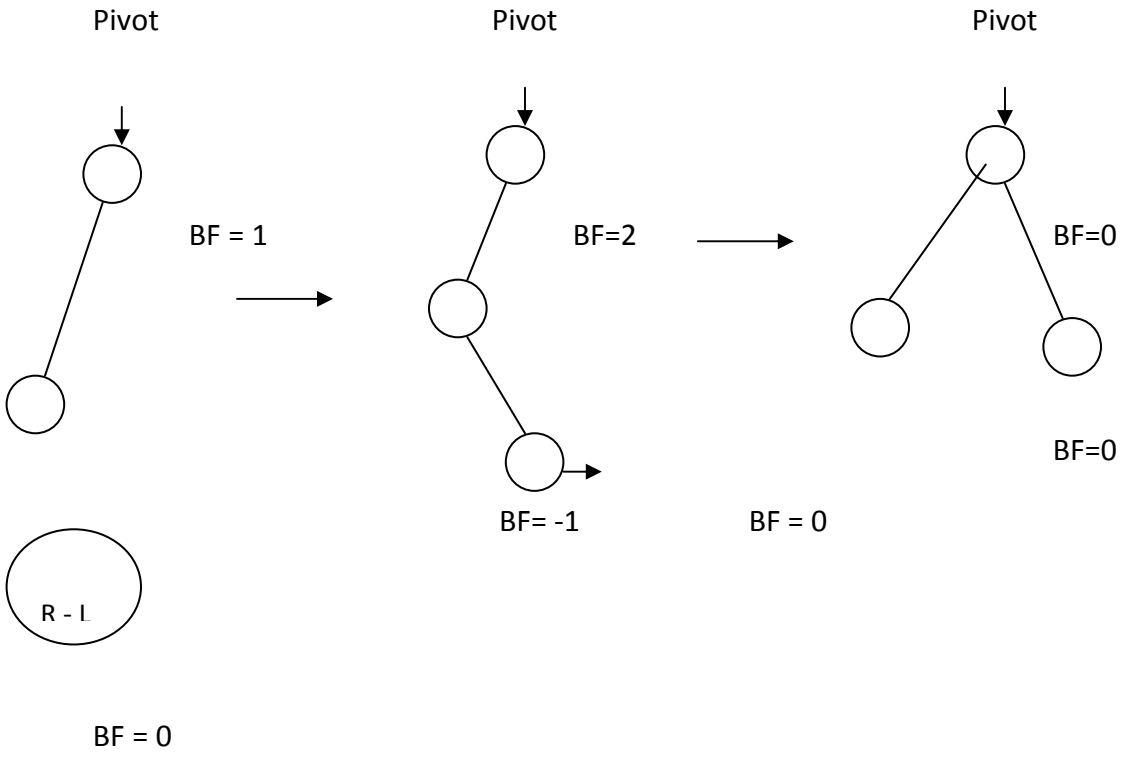
To insert and maintain balance:

1. Travel down the appropriate branch and keep track of balance. Left deepest node balance +1 or –1. This is called the *pivot node*.
2. From pivot-down, recompute all balance factors along insertion path.
3. Determine whether newly computed balance changes |1| → |2|
4. There was a change-manipulated pointers centered at pivot node to restore balance. (AVL Rotation)

4 cases – – discussing unbalancing

- Insert into left subtree of a left child of pivot node.
- Insert into right tree of right child of pivot node.
- Insert into right subtree of left child.
- Insert into left subtree of right child.

AVL Trees:



Y

```
X = Node [Pivot].Left;

Y = Node [X].Right;

Node {Pivot}.Left = Node [Y].Right;

Node [X].Right = Node [Y].Left;

Node [Y].Left = X;

Node [Y].Right = Pivot;

Pivot = Y;
```

Subcase # 1

```
if Node [Pivot].BF = 0 then

{

        Node [Node [Pivot].Left].BF = 0
        Node [Node [Pivot].Right].BF = 0

}
```

Subcase # 2

```
else if Node [Pivot].BF = 1 then

{

        Node [Pivot].BF = 0

        Node [Node [Pivot].Left].BF = 1
        Node [Node [Pivot].Right].BF = -1

}
```

Subcase # 3

```
else Node [Pivot].BF = 0

{
```

Node [Pivot].BF = 0

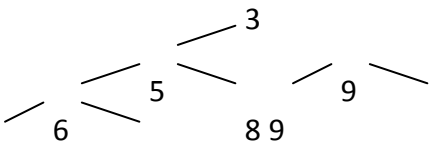
Node [Node [Pivot].Left].BF = +1
Node [Node [Pivot].Right].BF = 0

Priority Ques:

1. Linked List
   Insert by priority order.
2. Array
   Sort it and rearrange the elements.
3. Queue – Search
4. Heap Sort – Partial Ordered tree.
5. Array of Queues – Small & Priorities.

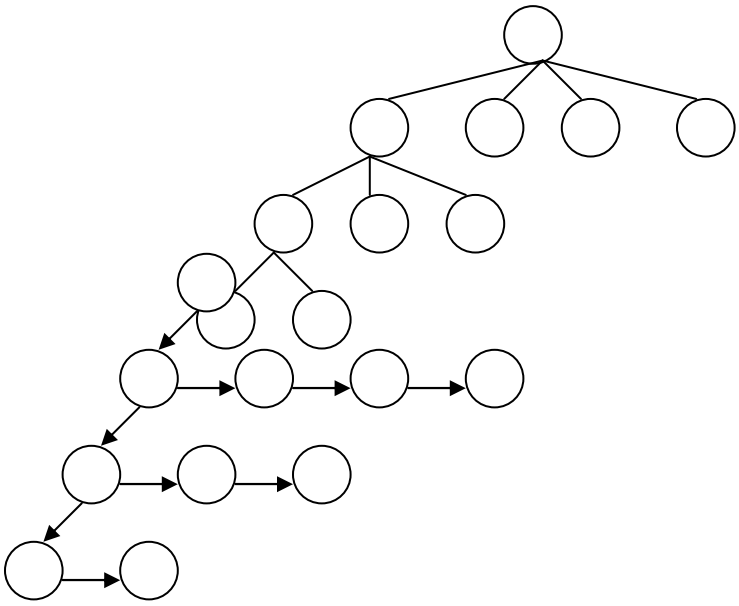
The priority of Node V is not greater than its children.

Trees that have lots of dependents (general trees)



Divide & Conquer
O (log(n))

10 18



Left pointer points to a list of children.

Right pointer points to a list of siblings.

Other way to implement this is Array of Pointers.

In-Order Traversal:

Traverse (forest) of first tree inorder.

- Visit root of first tree.
- Visit forest of remaining trees in order.

Algorithm:

```
void intra (Ptr K)
{
        if ( R == NULL ) then
        return
        else
        {
                intra ( R → child );
                R → Info >> cout;
                intra ( R → sib );
        }
}
```

WEEK 11:

Non- Linear structures.

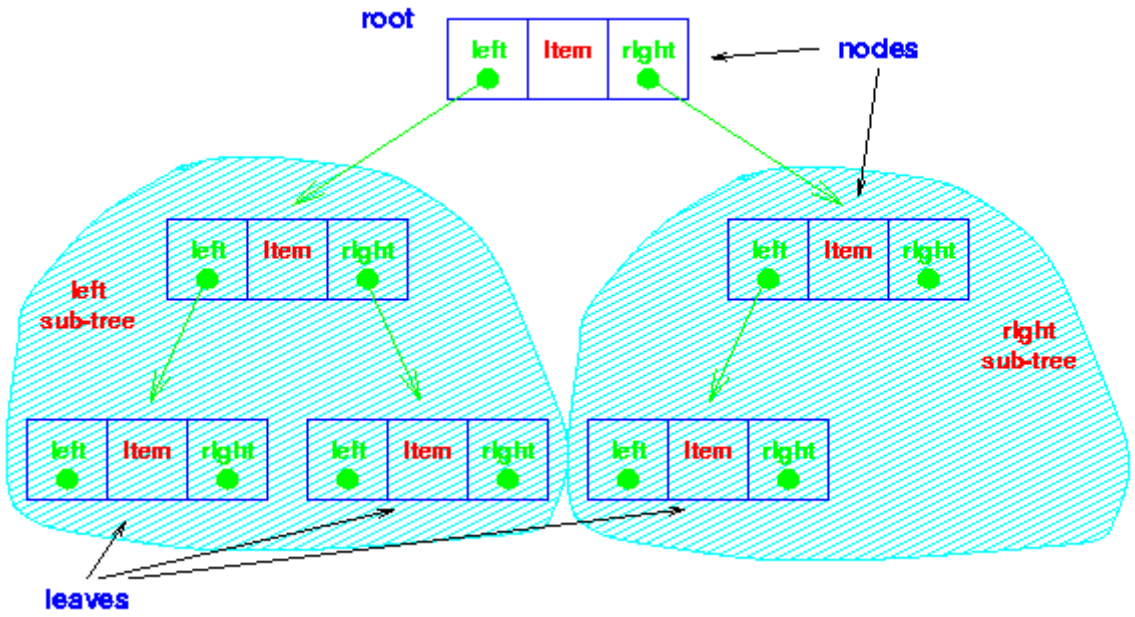
This week learning outcomes :
• Describe binary tree.
• Analysis of a complete tree .

Binary Trees

The simplest form of tree is a **binary tree**. A binary tree consists of

a. a *node* (called the **root** node) and
b. left and right sub-trees.
Both the sub-trees are themselves binary trees.

You now have a *recursively defined data structure*. (It is also possible to define a list recursively: can you see how?)



A binary tree

The nodes at the lowest levels of the tree (the ones with no sub-trees) are called **leaves**.

In an *ordered binary tree*,

1. the keys of all the nodes in the left sub-tree are less than that of the root,
2. the keys of all the nodes in the right sub-tree are greater than that of the root,
3. the left and right sub-trees are themselves ordered binary trees.

Data Structure

The data structure for the tree implementation simply adds left and right pointers in place of the next pointer of the linked list implementation. [Load the tree struct.]

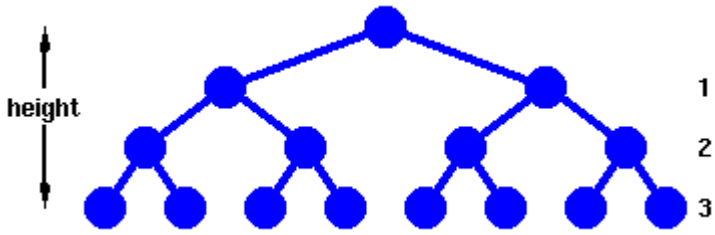
The `AddToCollection` method is, naturally, recursive. [Load the `AddToCollection` method.]

Similarly, the `FindInCollection` method is recursive. [Load the `FindInCollection` method.]

Analysis

Complete Trees

Before we look at more general cases, let's make the optimistic assumption that we've managed to fill our tree neatly, *ie* that each leaf is the same 'distance' from the root.



This forms a *complete tree*, whose *height* is defined as the number of links from the root to the deepest leaf.

A complete tree

First, we need to work out how many nodes, ***n***, we have in such a tree of height, ***h***.

Now,

$n = 1 + 2^1 + 2^2 + ... + 2^h$

From which we have,

$n = 2^{h+1} - 1$

and

h = **floor($\log_2 n$)**

Examination of the `Find` method shows that in the worst case, ***h***+1 or **ceiling($\log_2 n$)** comparisons are needed to find an item. This is the same as for binary search.

However, `Add` also requires **ceiling($\log_2 n$)** comparisons to determine where to add an item. Actually adding the item takes a constant number of operations, so we say that a binary tree requires **O(log*n*)** operations for *both* adding and finding an item - a considerable improvement over binary search for a *dynamic* structure which often requires addition of new items.

Deletion is also an **O(log*n*)** operation.

General binary trees

However, in general addition of items to an ordered tree will not produce a complete tree. The worst case occurs if we add an ordered list of items to a tree.

What will happen? Think before you click here!

This problem is readily overcome: we use a structure known as a heap. However, before looking at heaps, we should formalise our ideas about the complexity of algorithms by defining carefully what **O(f(*n*))** means.

Key terms

Root Node

Node at the "top" of a tree - the one from which all operations on the tree commence. The root node may not exist (a NULL tree with no nodes in it) or have 0, 1 or 2 children in a binary tree.

Leaf Node

Node at the "bottom" of a tree - farthest from the root. Leaf nodes have no children.

Complete Tree

Tree in which each leaf is at the same distance from the root. A more precise and formal definition of a complete tree is set out later.

Height

Number of nodes which must be traversed from the root to reach a leaf of a tree.

WEEK 12:

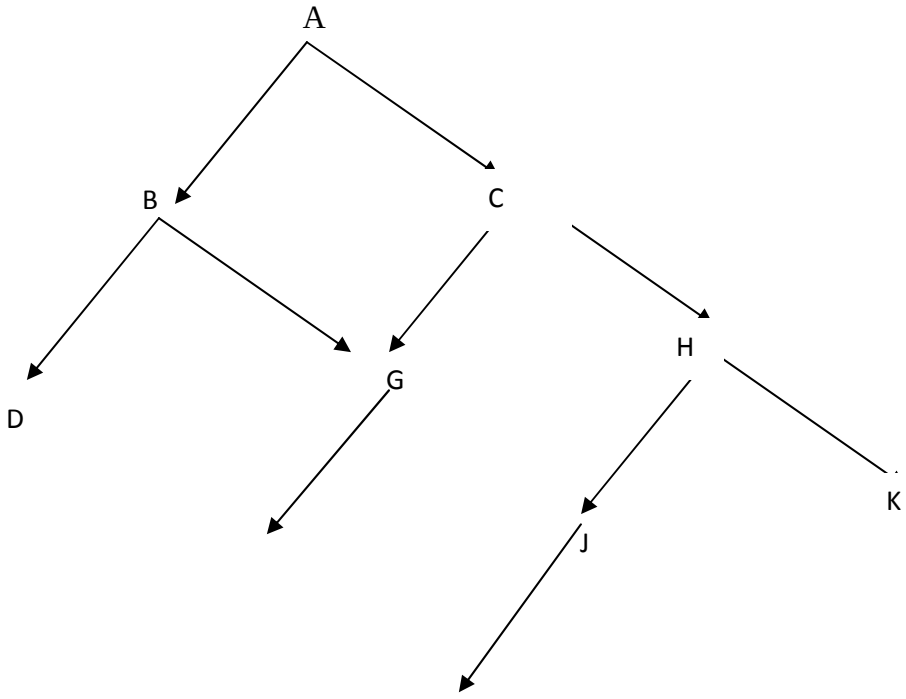
Non- Linear structures.

This week learning outcomes :
• Binary tree structure revisited .
• State properties of tree.
• **Basic Operations on Binary Trees.**

Binary Tree

A binary tree T , is defined as a finite set of elements called nodes, such that
a.) T is empty (called the null tree or empty tree) or
b.) T contains a distinguished node R, called the root of T, and the remaining nodes of T form an order pair of disjoint binary trees T1 and T2.

If T contains a root R, then the two trees T1 and T2 are called respectively, the left and right sub-tree of R , if T1 is nonempty, then its root is called the left successor of R , similarly if T2 is nonempty, then its root is called the right successor of R.



A left-downward slanted line from a node N indicates a left successor of N, and a right-downward slanted line from N indicates a right successor of N . Observe that :
i.) B is a left successor and C is a right successor of the node A.
ii.) The left sub-tree of the root A consists of the nodes B,D,E and F .
iii.)The right sub-tree of A consists of the nodes C,G,H,J,K ,,and L.

Any node N in a binary tree T has either 0,1, or 2 successors. The nodes A,B,C and H have 2 successors, the nodes E and J have only one successor, and the nodes D,F,G,L, and K have no successors. The nodes with zero successors are called terminal nodes.

Terminology

Suppose N is a node in T with left successor S1 and right successor S2 then N is called the parent (or father) of S1 and S2 . Analogously, S1 is called the left child (or son) of N and S2 is called the right child (or son). Furthermore, S1 and S2 are said to be siblings (or brothers). Every node N in a binary tree T, except the root has a unique parent, called the predecessor of N.

The terms, descendant and ancestor have their usual meaning. That is , a node L is called a descendant of node N (and N is called an ancestor of L) if there is a succession of children from N to L. In particular, L is called a left or right descendant of N according to whether L belongs to the left or right sub-tree of N . The line drawn from a node N of T to a successor is called an edge, and a sequence of consecutive edges is called a path. A terminal node is called a leaf, and path ending in a leaf is called a branch. Each node in a binary tree T is assigned a level number as follows:

The root R of the tree T is assigned the level number 0 , and every other node is assigned a level number which is 1 more than the level number of its parent . Those nodes with the same level are said to belong to the same generation .

The depth (or height) of a tree T is the maximum number of nodes in a branch of T. This turns ont to be 1 more than the largest level number of T. Binary tree T and T1 are said to be similar if they have the same structure or in other words, if they have the same shape . The trees are said to be copies if they are similar and if they have the same contents at corresponding nodes.

TRAVERSING BINARY TREES

Basic Operations on Binary Trees

There are many tasks that may have to be perfomed on a tree structure; a common one is that of executing a given operation P on each element of the tree. P is then understood to be a parameter of the more general task of visting all nodes or, as it is usually called, of tree traversal. If we consider the task as a single sequential process, then the individual nodes are visited in some specific order and may be considered as being laid out in a linear arrangement. In fact, the description of many algorithms is considerably facilitated if we can talk about processing the next element in the tree based in an underlying order. There are three principal orderings that emerge naturally from the structure of trees. Like the treestructure itself, they are conveniently expressed in recursive terms. Referring to the binary tree .

Let R denote the root and A and B denote the left and right subtrees, the three orderings are

1. Preorder: R, A, B (visit root before the subtrees)
2. Inorder: A, R, B
3. Postorder: A, B, R (visit root after the subtrees)

In other words, there are three (3) standard ways of travasing a binary tree T with root R. These three algorithms are called preorder, inorder, and postorder.

Preorder:

a.) Process the root R.
b.) Traverse the left sub-tree of R in preorder.
c.) Traverse the right sub-tree of R in Preorder.

Inorder:

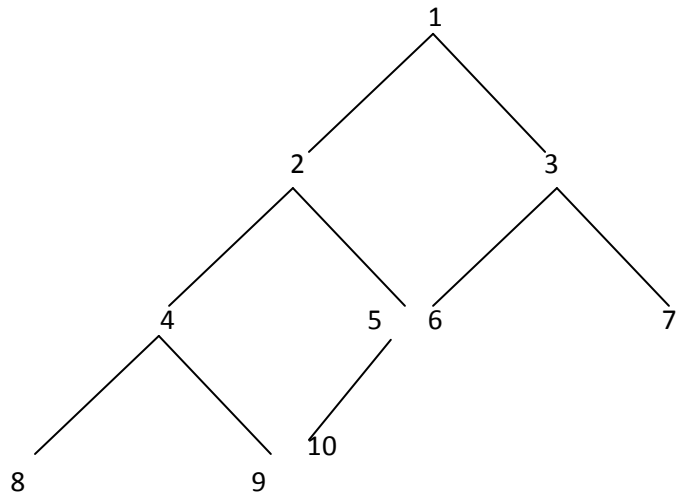
a.) Traverse the left sub-tree of R inorder.
b.) Process the root R .
c.) Traverse the right sub-tree of R inorder .

Postorder:

a.) Traverse the left sub-tree of R in preorder.
b.) Traverse the right sub-tree of R in Postorder.
c.) Process the root R.

Traverse the following binary tree using:

i.) **Preorder algorithm**
ii.) **Inorder algorithm**
iii.)**Postorder algorithm.**



Solution:

i.) **Preorder algorithm: 12489510367**

ii.) **Inorder algorithm: ?**

iii.)**Postorder algorithm: ?**

WEEK 13:

Sorting

This week Learning outcomes :
• Define sorting.
• Explain various categories of sorting.

Sorting

Sorting is one of the most important operations performed by computers. In the days of magnetic tape storage before modern data-bases, it was almost certainly the *most* common operation performed by computers as most "database" updating was done by sorting transactions and merging them with a master file. It's still important for presentation of data extracted from databases: most people prefer to get reports sorted into some relevant order before wading through pages of data!

Sorting is generally understood to be the process of rearranging a given set of objects in a specific order. The purpose of sorting is to facilitate the later search for members of the sorted set. As such it is an almost universally performed, fundamental activity. Objects are sorted in telephone books, in income tax files, in tables of contents, in libraries, in dictionaries, in warehouses, and almost everywhere that stored objects have to be searched and retrieved. Even small children are taught to put their things "in order", and they are confronted with some sort of sorting long before they learn anything about arithmetic.
Hence, sorting is a relevant and essential activity, particularly in data processing. What else would be easier to sort than data! Nevertheless, our primary interest in sorting is devoted to the even more fundamental techniques used in the construction of algorithms. There are not many techniques that do not occur somehere in connection with sorting algorithms. In particular, sorting is an ideal subject to demonstrate a great diversity of algorithms, all having the same purpose, many of them being optimal in some sense, and most of them having advantages over others. It is therefore an ideal subject to demonstrate the necessity of performance analysis of algorithms. The example of sorting is moreover well suited for showing how a very significant gain in performance may be obtained by the development of sophisticated algorithms when obvious methods are readily available.

Different categories of sorting

The dependence of the choice of an algorithm on the structure of the data to be processed -- an ubiquitous phenomenon -- is so profound in the case of sorting that sorting methods are generally classified into two categories, namely:

i.) sorting of arrays, and
ii.) sorting of (sequential) files.

The two classes are often called *internal* and *external sorting* because arrays are stored in the fast, high-speed, random-access "internal" store of computers and files are appropriate on the slower, but more spacious "external" stores based on mechanically moving devices (disks and tapes).

WEEK 14:

Different types of sorting techniques.

This week Learning outcomes :
• Explain Sorting by insertion.
• Explain Sorting by insertion.
• Explain Sorting by exchange.

Different types of sorting techniques.

Sorting methods that sort items in situ can be classified into three principal categories according to their underlying method:
Sorting by insertion
Sorting by selection
Sorting by exchange
These three pinciples will now be examined and compared. The procedures operate on a global variable *a* whose components are to be sorted in situ, i.e. without requiring additional, temporary storage. The components are the keys themselves. We discard other data represented by the record type *Item*, thereby simplifying matters. In all algorithms to be developed in this chapter, we will assume the presence of an array *a* and a constant *n*, the number of elements of *a*:

```
TYPE Item = INTEGER;
VAR a: ARRAY n OF Item
```

Sorting by Straight Insertion

This method is widely used by card players. The items (cards) are conceptually divided into a destination sequence a1 ... ai-1 and a source sequence ai ... an. In each step, starting with i = 2 and incrementing i by unity, the i th element of the source sequence is picked and transferred into the destination sequence by inserting it at the appropriate place.
47
Initial Keys: 44 55 12 42 94 18 06 67
i=1 44 55 12 42 94 18 06 67
i=2 12 44 55 42 94 18 06 67
i=3 12 42 44 55 94 18 06 67
i=4 12 42 44 55 94 18 06 67
i=5 12 18 42 44 55 94 06 67
i=6 06 12 18 42 44 55 94 67
i=7 06 12 18 42 44 55 67 94
A Sample Process of Straight Insertion Sorting.
The process of sorting by insertion is shown in an example of eight numbers chosen at random . The algorithm of straight insertion is as follows:

```
FOR i := 1 TO n-1 DO
x := a[i];
```

insert x at the appropriate place in a0 ... ai
END

In the process of actually finding the appropriate place, it is convenient to alternate between comparisons and moves, i.e., to let x sift down by comparing x with the next item aj, and either inserting x or moving aj to the right and proceeding to the left. We note that there are two distinct conditions that may cause the termination of the sifting down process:
1. An item aj is found with a key less than the key of x.
2. The left end of the destination sequence is reached.

```
PROCEDURE StraightInsertion;
VAR i, j: INTEGER; x: Item;
BEGIN
FOR i := 1 TO n-1 DO
x := a[i]; j := i;
WHILE (j > 0) & (x < a[j-1] DO a[j] := a[j-1]; DEC(j) END ;
a[j] := x
END
END StraightInsertion
```

Analysis of straight insertion. The number Ci of key comparisons in the i-th sift is at most i-1, at least 1, and-- assuming that all permutations of the n keys are equally probable -- i/2 in the average. The number Mi of moves (assignments of items) is Ci + 2 (including the sentinel). Therefore, the total numbers of comparisons
and moves are

Cmin = n-1 Mmin = 3*(n-1)
Cave = (n2 + n - 2)/4 Mave = (n2 + 9n - 10)/4
Cmax = (n2 + n - 4)/4 Mmax = (n2 + 3n - 4)/2

The minimal numbers occur if the items are initially in order; the worst case occurs if the items are initially in reverse order. In this sense, sorting by insertion exhibits a truly natural behavior. It is plain that the given algorithm also describes a stable sorting process: it leaves the order of items with equal keys unchanged.

The algorithm of straight insertion is easily improved by noting that the destination sequence a0 ... ai-1, in which the new item has to be inserted, is already ordered. Therefore, a faster method of determining the insertion point can be used. The obvious choice is a binary search that samples the destination sequence in the middle and continues bisecting until the insertion point is found. The modified sorting algorithm is called *binary insertion*.

```
PROCEDURE BinaryInsertion(VAR a: ARRAY OF Item; n: INTEGER);
VAR i, j, m, L, R: INTEGER; x: Item;
BEGIN
FOR i := 1 TO n-1 DO
```



```
x := a[i]; L := 1; R := i;
WHILE L < R DO
m := (L+R) DIV 2;
IF a[m] <= x THEN L := m+1 ELSE R := m END
END ;
FOR j := i TO R+1 BY -1 DO a[j] := a[j-1] END ;
```

```
a[R] := x
END
END BinaryInsertion
```

Sorting by Straight Selection

This method is based on the following principle:

1. Select the item with the least key.
2. Exchange it with the first item a0.
3. Then repeat these operations with the remaining n-1 items, then with n-2 items, until only one item -- the largest -- is left.

This method is shown on the same eight keys as given above.

Initial keys 44 55 12 42 94 18 06 67
06 55 12 42 94 18 44 67
06 12 55 42 94 18 44 67
06 12 18 42 94 55 44 67
06 12 18 42 94 55 44 67
49
06 12 18 42 44 55 94 67
06 12 18 42 44 55 94 67
06 12 18 42 44 55 67 94

A Sample Process of Straight Selection Sorting.

The algorithm is formulated as follows:

```
FOR i := 0 TO n-1 DO
assign the index of the least item of ai ... an-1 to k;
exchange ai with ak
END
```

This method, called *straight selection*, is in some sense the opposite of straight insertion: Straight insertion
considers in each step only the one next item of the source sequence and all items of the destination array to
find the insertion point; straight selection considers all items of the source array to find the one with the least
key and to be deposited as the one next item of the destination sequence..

```
PROCEDURE StraightSelection;
VAR i, j, k: INTEGER; x: Item;
BEGIN
FOR i := 0 TO n-2 DO
k := i; x := a[i];
FOR j := i+1 TO n-1 DO
IF a[j] < x THEN k := j; x := a[k] END
END ;
a[k] := a[i]; a[i] := x
END
END StraightSelection
```

Sorting by Straight Exchange

The classification of a sorting method is seldom entirely clear-cut. Both previously discussed methods can also be viewed as exchange sorts. In this section, however, we present a method in which the exchange of two items is the dominant characteristic of the process. The subsequent algorithm of straight exchanging is based on the principle of comparing and exchanging pairs of adjacent items until all items are sorted.

As in the previous methods of straight selection, we make repeated passes over the array, each time sifting the least item of the remaining set to the left end of the array. If, for a change, we view the array to be in a vertical instead of a horizontal position, and -- with the help of some imagination -- the items as bubbles in a water tank with weights according to their keys, then each pass over the array results in the ascension of a
bubble to its appropriate level of weight (see Table below). This method is widely known as the *Bubblesort*.

```
I = 1 2 3 4 5 6 7 8
44 06 06 06 06 06 06 06
55 44 12 12 12 12 12 12
12 55 44 18 18 18 18 18
42 12 55 44 42 42 42 42
94 42 18 55 44 44 44 44
18 94 42 42 55 55 55 55
06 18 94 67 67 67 67 67
67 67 67 94 94 94 94 94
```

A Sample of Bubblesorting.

```
PROCEDURE BubbleSort;
VAR i, j: INTEGER; x: Item;
BEGIN
FOR i := 1 TO n-1 DO
FOR j := n-1 TO i BY -1 DO
IF a[j-1] > a[j] THEN
x := a[j-1]; a[j-1] := a[j]; a[j] := x
END
END
END
END BubbleSort
```

Insertion Sort by Diminishing Increment

A refinement of the straight insertion sort was proposed by D. L. Shell in l959. The method is explained and demonstrated on our standard example of eight items. First, all items that are four positions apart are grouped and sorted separately. This process is called a 4-sort. In this example of eight items, each group contains exactly two items. After this first pass, the items are regrouped into groups with items two positions apart and then sorted anew. This process is called a 2-sort. Finally, in a third pass, all items are sorted in an ordinary sort or 1-sort.

One may at first wonder if the necessity of several sorting passes, each of which involves all items, does not introduce more work than it saves. However, each sorting step over a chain either involves relatively few items or the items are already quite well ordered and comparatively few rearrangements are required.
It is obvious that the method results in an ordered array, and it is fairly obvious that each pass profits from previous passes (since each i-sort combines two groups sorted in the preceding 2i-sort). It is also obvious that any sequence of increments is acceptable, as long as the last one is unity, because in the worst case the last pass does all the work. It is, however, much less obvious that the method of diminishing increments yields
even better results with increments other than powers of 2.
44 55 12 42 94 18 06 67
4-sort yields 44 18 06 42 94 55 12 67
2-sort yield 06 18 12 42 44 55 94 67
1-sort yields 06 12 18 42 44 55 67 94
Table 2.5 An Insertion Sort with Diminishing Increments.
The procedure is therefore developed without relying on a specific sequence of increments. The T increments are denoted by h0, h1, ... , hT-1 with the conditions
ht-1 = 1, hi+1 < hi

The algorithm is described by the procedure *Shellsort* [2.11] for t = 4:

```
PROCEDURE ShellSort;
CONST T = 4;
VAR i, j, k, m, s: INTEGER;
x: Item;
h: ARRAY T OF INTEGER;
BEGIN h[0] := 9; h[1] := 5; h[2] := 3; h[3] := 1;
FOR m := 0 TO T-1 DO
k := h[m];
FOR i := k+1 TO n-1 DO
x := a[i]; j := i-k;
WHILE (j >= k) & (x < a[j]) DO a[j+k] := a[j]; j := j-k END ;
a[j+k] := x
END
END
END ShellSort
```

Analysis of Shellsort. The analysis of this algorithm poses some very difficult mathematical problems, many of which have not yet been solved. In particular, it is not known which choice of increments yields the best results. One surprising fact, however, is that they should not be multiples of each other. This will avoid the phenomenon evident from the example given above in which each sorting pass combines two chains that
before had no interaction whatsoever. It is indeed desirable that interaction between various chains takes place as often as possible, and the following theorem holds: If a k-sorted sequence is i-sorted, then it remains k-sorted. Knuth [2.8] indicates evidence that a reasonable choice of increments is the sequence (written in reverse order)
1, 4, 13, 40, 121, ...

where hk-1 = 3hk+1, ht = 1, and t = k×log3(n) - 1. He also recommends the sequence
1, 3, 7, 15, 31, ...
where hk-1 = 2hk+1, ht = 1, and t = k×log2(n) - 1. For the latter choice, mathematical analysis yields an effort proportional to n2 required for sorting n items with the Shellsort algorithm. Although this is a significant improvement over n2, we will not expound further on this method, since even better algorithms are known.

WEEK 15:

Different sorting and searching .

This week Learning outcomes :
• Explain various sorting techniques.
• Explain linear and binary search algorithm.

Different types of sorting revisited:

Bubble, Selection, Insertion Sorts

There are a large number of variations of one basic strategy for sorting. It's the same strategy that you use for sorting your bridge hand. You pick up a card, start at the beginning of your hand and find the place to insert the new card, insert it and move all the others up one place.

```
/* Insertion sort for integers */

void insertion( int a[], int n ) {
/* Pre-condition: a contains n items to be sorted */
    int i, j, v;
    /* Initially, the first item is considered 'sorted' */
    /* i divides a into a sorted region, x<i, and an
       unsorted one, x >= i */
    for(i=1;i<n;i++) {
        /* Select the item at the beginning of the
           as yet unsorted section */
        v = a[i];
        /* Work backwards through the array, finding where v
           should go */
        j = i;
        /* If this element is greater than v,
              move it up one */
        while ( a[j-1] > v ) {
          a[j] = a[j-1]; j = j-1;
          if ( j <= 0 ) break;
          }
        /* Stopped when a[j-1] <= v, so put v at position j */
        a[j] = v;
        }
    }
```

Bubble Sort

Another variant of this procedure, called bubble sort, is commonly taught:

```
/* Bubble sort for integers */
#define SWAP(a,b)   { int t; t=a; a=b; b=t; }

void bubble( int a[], int n )
/* Pre-condition: a contains n items to be sorted */
```

```
{
int i, j;
/* Make n passes through the array */
for(i=0;i<n;i++)
    {
    /* From the first element to the end
       of the unsorted section */
    for(j=1;j<(n-i);j++)
       {
       /* If adjacent items are out of order, swap them */
       if( a[j-1]>a[j] ) SWAP(a[j-1],a[j]);
       }
    }
```

Analysis

Each of these algorithms requires ***n***-1 passes: each pass places one item in its correct place. (The n^{th} is then in the correct place also.) The i^{th} pass makes either ***i***or ***n - i*** comparisons and moves. So:

$$\begin{aligned} T(n) &= 1+2+3+\ldots+(n-1) \\ &= \sum_{i=1}^{n-1} i \\ &= \tfrac{n}{2}(n-1) \end{aligned}$$

or **O(*n*^{2})** - but we already know we can use heaps to get an **O(*n* log*n*)** algorithm. Thus these algorithms are only suitable for small problems where their simple code makes them faster than the more complex code of the **O(*n* log*n*)** algorithm. As a rule of thumb, expect to find an **O(*n* log*n*)** algorithm faster for *n*>10 - *but the exact value depends very much on individual machines!*.

They can be used to squeeze a little bit more performance out of fast sort algorithms

Quick Sort

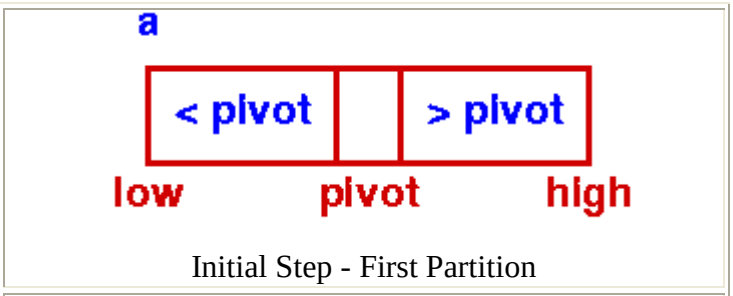
Quicksort is a very efficient sorting algorithm invented by C.A.R. Hoare. It has two phases:

- the partition phase and
- the sort phase.

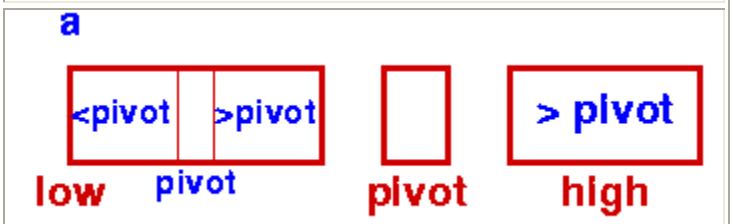
As we will see, most of the work is done in the partition phase - it works out where to divide the work. The sort phase simply sorts the two smaller problems that are generated in the partition phase.

This makes Quicksort a good example of the **divide and conquer** strategy for solving problems. (You've already seen an example of this approach in the binary search procedure.) In quicksort, we divide the array of items to be sorted into two partitions and then call the quicksort procedure recursively to sort the two partitions, *ie* we *divide* the problem into two smaller ones and *conquer* by solving the smaller ones. Thus the conquer part of the quicksort routine looks like this:

```
quicksort( void *a, int low, int
high )
  {
  int pivot;
  /* Termination condition! */
  if ( high > low )
    {
    pivot = partition( a, low,
high );
    quicksort( a, low, pivot-1
);
    quicksort( a, pivot+1, high
);
    }
  }
```



Initial Step - First Partition



Sort Left Partition in the same way

For the strategy to be effective, the *partition* phase must ensure that all the items in one part (the lower part) and less than all those in the other (upper) part.

To do this, we choose a *pivot* element and arrange that all the items in the lower part are less than the pivot and all those in the upper part greater than it. In the most general case, we don't know anything about the items to be sorted, so that any choice of the pivot element will do - the first element is a convenient one.

As an illustration of this idea, you can view this animation, which shows a partition algorithm in which items to be sorted are copied from the original array to a new one: items *smaller* than the pivot are placed to the left of the new array and items *greater* than the pivot are placed on the right. In the final step, the pivot is dropped into the remaining slot in the middle.

Searching

Computer systems are often used to store large amounts of data from which individual records must be retrieved according to some search criterion. Thus the efficient storage of data to facilitate fast searching is an important issue. In this section, we shall investigate the performance of some searching algorithms and the data structures which they use.

Linear Search

The simplest type of search is linear search, where every item in the item in the table is searched in sequence. If the the table is sorted on the field being searched (the key field) then the search can be be abandoned as soon as the search value exeeds the field.

The pseudocode for a procedure to search the table for a given course code is as follows:

```
Procedure SEACH_Table
Begin
        Subscript=0
        Code_found = false
        Repeat
                Subscript =subscript + 1
                If course[substript].cource_code=Item_sought
                        Then code_found = true
                Endif
Until code_found=true or subscript=no_of_elements
                Or courses[subscript].course_code > item_sought
End procedure
```

The procedure sets a boolean variable code_founf to true if the course code is found.

Binary Search

However, if we place our items in an array and sort them in either ascending or descending order on the key first, then we can obtain much better performance with an algorithm called **binary search**.

Binary search is a technique for searching an ordered list in which we first check the middle item and - based on that comparison - "discard" half the data. The same procedure is then applied to the remaining half until a match is found or there are no more items left.

That is In binary search, we first compare the key with the item in the middle position of the array. If there's a match, we can return immediately. If the key is less than the middle key, then the item sought must lie in the lower half of the array; if it's greater then the item sought must lie in the upper half of the array. So we repeat the procedure on the lower (or upper) half of the array.

Our `FindInCollection` function can now be implemented:

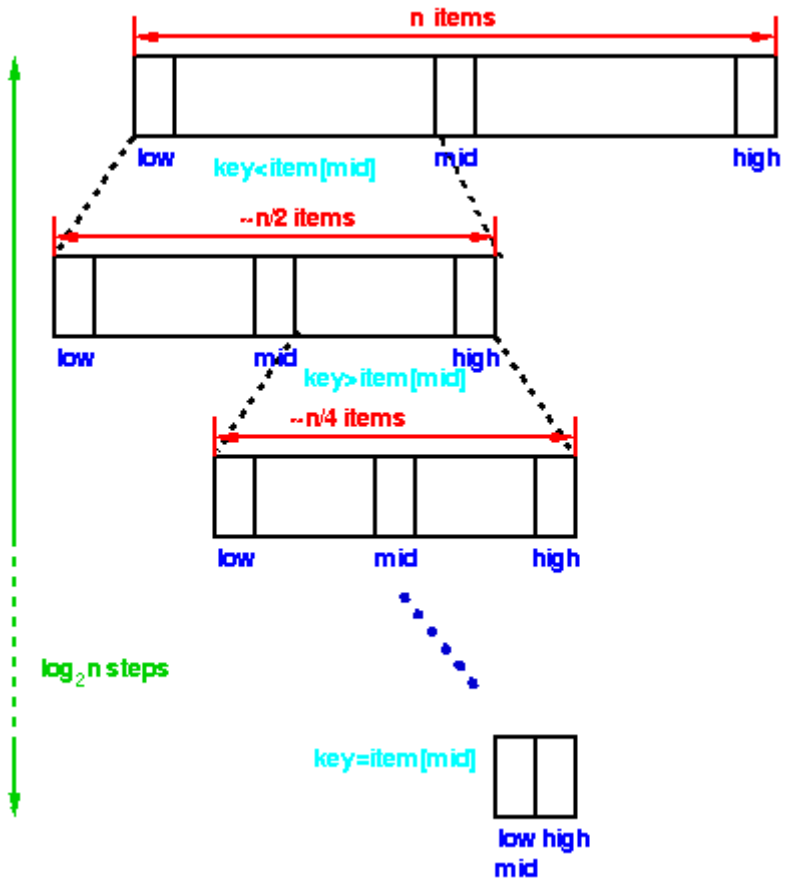
```
static void *bin_search( collection c, int low, int high, void *key ) {
        int mid;
        /* Termination check */
        if (low > high) return NULL;
        mid = (high+low)/2;
        switch (memcmp(ItemKey(c->items[mid]),key,c->size)) {
                /* Match, return item found */
                case 0: return c->items[mid];
                /* key is less than mid, search lower half */
                case -1: return bin_search( c, low, mid-1, key);
                /* key is greater than mid, search upper half */
                case 1: return bin_search( c, mid+1, high, key );
                default : return NULL;
                }


void *FindInCollection( collection c, void *key ) {
/* Find an item in a collection
   Pre-condition:
        c is a collection created by ConsCollection
        c is sorted in ascending order of the key
        key != NULL
   Post-condition: returns an item identified by key if
   one exists, otherwise returns NULL
*/
        int low, high;
        low = 0; high = c->item_cnt-1;
        return bin_search( c, low, high, key );
}
```

Points to note:

a. `bin_search` is recursive: it determines whether the search key lies in the lower or upper half of the array, then calls itself on the appropriate half.
b. There is a termination condition (two of them in fact!)
    i. If `low > high` then the partition to be searched has no elements in it *and*
    ii. If there is a match with the element in the middle of the current partition, then we can return immediately.
c. `AddToCollection` will need to be modified to ensure that each item added is placed in its correct place in the array. The procedure is simple:
    i. Search the array until the correct spot to insert the new item is found,
    ii. Move all the following items up one position *and*
    iii. Insert the new item into the empty position thus created.
d. `bin_search` is declared `static`. It is a local function and is not used outside this class: if it were not declared static, it would be exported and be available to all parts of the program. The static declaration also allows other classes to use the same name internally.

`static` reduces the visibility of a function an should be used wherever possible to control access to functions!

Analysis



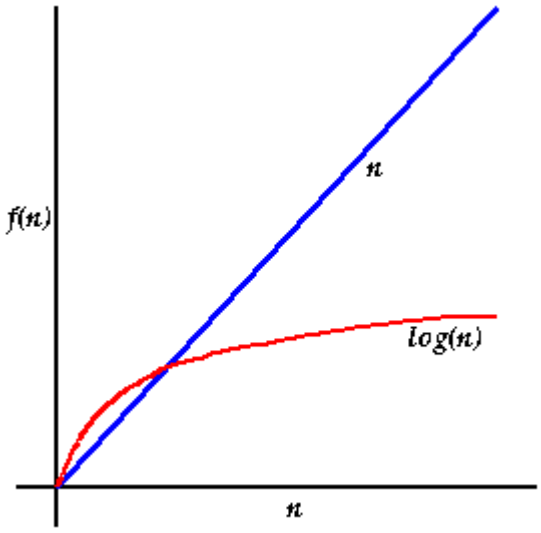
Each step of the algorithm divides the block of items being searched in half. We can divide a set of ***n*** items in half at most $\log_2 n$ times.

Thus the running time of a binary search is proportional to $\log n$ and we say this is a $O(\log n)$ algorithm.

Binary search requires a more complex program than our original search and thus for *small* ***n*** it may run slower than the simple linear search. However, for large ***n***,

$$\lim_{n \to \infty} \frac{\log n}{n} = 0$$

Thus at large ***n***, **log *n*** is *much* smaller than ***n***, consequently an ***O*(log *n*)** algorithm is *much* faster than an ***O(n)*** one.



Plot of ***n*** and **log *n*** *vs* ***n*** .

We will examine this behaviour more formally in a later section. First, let's see what we can do about the insertion (`AddToCollection`) operation.

In the worst case, insertion may require ***n*** operations to insert into a sorted list.

1. We can find the place in the list where the new item belongs using binary search in ***O*(log *n*)** operations.
2. However, we have to shuffle all the following items up one place to make way for the new one. In the worst case, the new item is the first in the list, requiring ***n*** move operations for the shuffle!

A similar analysis will show that deletion is also an ***O(n)*** operation.

If our collection is static, *ie* it doesn't change very often - if at all - then we may not be concerned with the time required to change its contents: we may be prepared for the initial build of the collection and the occasional insertion and deletion to take some time. In return, we will be able to use a simple data structure (an array) which has little memory overhead.

However, if our collection is large and dynamic, *ie* items are being added and deleted continually, then we can obtain considerably better performance using a data structure called a tree.

Note that Big Oh Is a notation formally describing the set of all functions which are bounded above by a nominated function.