CSE & IT

- Analysis,Design And Algorithsms(ADA)
- Operating System
- Lexical Analysis
- Database Management System

Analysis,Design And Algorithm Study Material

- Numerical algorithm

- Review of searching algorithm
- Review of sorting algorithm
- Recursion v/s iteration
- Introduction to graph theory
- Matrix representation
- Trees
- Divide & Conquer : Binary search
- Max- Min Search & Merge sort
- Integer Moltiplication
- Cassette filling
- Knapsack problem
- Job schedoling
- Backtracking

- Branch & Bound
- Shortest path
- Minimal spanning trees
- Techniques for Graphs

Concept of Algorithm:

A common man's belief is that a computer can do anything and everything that he imagines. It is very difficult to make people realize that it is not really the computer but the man behind computer who does everything.

In the modern internet world man feels that just by entering what he wants to search into the computers he can get information as desired by him. He believes that, this is done by computer. A common man seldom understands that a man made procedure called search has done the entire job and the only support provided by the computer is the executional speed and organized storage of information.

In the above instance, a designer of the information system should know what one frequently searches for. He should make a structured organization of all those details to store in memory of the computer. Based on the requirement, the right information is brought out. This is accomplished through a set of instructions created by the designer of the information system to search the right information matching the requirement of the user. This set of instructions is termed as program. It should be evident by now that it is not the computer, which generates automatically the program but it is the designer of the information system who has created this.

Thus, the program is the one, which through the medium of the computer executes to perform all the activities as desired by a user. This implies that programming a computer is more important than the computer itself while solving a problem using a computer and this part of programming has got to be done by the man behind the computer. Even at this stage, one should not quickly jump to a conclusion that coding is programming. Coding is perhaps the last stage in the process of programming. Programming involves various activities form the stage of conceiving the problem upto the stage of creating a model to solve the problem. The formal representation of this model as a sequence of instructions is called an algorithm and coded algorithm in a specific computer language is called a program.

One can now experience that the focus is shifted from computer to computer programming and then to creating an algorithm. This is algorithm design, heart of problem solving.

Characteristic of Algorithm:

Let us try to present the scenario of a man brushing his own teeth(natural denture) as an algorithm as follows. Step 1. Take the brush Step 2. Apply the paste Step 3. Start brushing Step 4. Rinse Step 5. Wash Step 6. Stop

If one goes through these 6 steps without being aware of the statement of the problem, he could possibly feel that this is the algorithm for cleaning a toilet. This is because of several ambiguities while comprehending every step. The step 1 may imply tooth brush, paint brush, toilet brush etc. Such an ambiguity doesn't an instruction an algorithmic step. Thus every step should be made unambiguous. An unambiguous step is called definite instruction. Even if the step 2 is rewritten as apply the tooth paste, to eliminate ambiguities yet the conflicts such as, where to apply the tooth paste and where is the source of the tooth paste, need to be resolved. Hence, the act of applying the toothpaste is not mentioned. Although unambiguous, such unrealizable steps can't be included as algorithmic instruction as they are not effective. The definiteness and effectiveness of an instruction implies the successful termination of that instruction. However the above two may not be sufficient to guarantee the termination of the algorithm. Therefore, while designing an algorithm care should be taken to provide a proper termination for algorithm.
Thus, every algorithm should have the following five characteristic feature
Input
Output
Definiteness
Effectiveness
Termination

Therefore, an algorithm can be defined as a sequence of definite and effective instructions, which terminates with the production of correct output from the given input. In other words, viewed little more formally, an algorithm is a step by step formalization of a mapping function to map input set onto an output set. The problem of writing down the correct algorithm for the above problem of brushing the teeth is left to the reader. For the purpose of clarity in understanding, let us consider the following examples. Example 1: Problem : finding the largest value among n>=1 numbers. Input : the value of n and n numbers Output : the largest value Steps : Let the value of the first be the largest value denoted by BIG
Let R denote the number of remaining numbers. R=n-1
If R != 0 then it is implied that the list is still not exhausted. Therefore look the next number called NEW.
Now R becomes R-1
If NEW is greater than BIG then replace BIG by the value of NEW
Repeat steps 3 to 5 until R becomes zero.
Print BIG
Stop
End of algorithm Example 2: quadratic equation Example 3: listing all prime numbers between two limits n1 and n2. **1.2.1 Algorithmic N otations** In this section we present the pseudocode that we use through out the book to describe algorithms. The pseudo code used resembles PASCAL and C language control structures. Hence, it is expected that the reader be aware of PASCAL/C. Even otherwise atleast now it is required that the reader should know preferably C to practically test the algorithm in this course work. However, for the sake of completion we present the commonly employed control constructs present in the algorithms. A conditional statement has the following form **If < condition> then Block 1 Else Block 2 If end.** This pseudocode executes block1 if the condition is true otherwise block2 is executed. The two types of loop structures are counter based and conditional based and they are as follows **For variable = value1 to value2 do Block For end** Here the block is executed for all the values of the variable from value 1 to value 2. There are two types of conditional looping, while type and repeat type. **While (condition) do Block While end.** Here block gets executed as long as the condition is true. **Repeat Block Until<condition>** Here block is executed as long as condition is false. It may be observed that the block is executed atleast once in repeat type. **Exercise 1;** Devise the algorithm for the following and verify whether they satisfy all the features. An algorithm that inputs three numbers and outputs them in ascending order. To test whether the three numbers represent the sides of a right angle triangle. To test whether a given

point p(x,y) lies on x-axis or y-axis or in I/II/III/IV quadrant. To compute the area of a circle of a given circumference To locate a specific word in a dictionary.

Numerical Algorithm:

If there are more then one possible way of solving a problem, then one may think of more than one algorithm for the same problem. Hence, it is necessary to know in what domains these algorithms are applicable. Data domain is an important aspect to be known in the field of algorithms. Once we have more than one algorithm for a given problem, how do we choose the best among them? The solution is to devise some data sets and determine a performance profile for each of the algorithms.

A best case data set can be obtained by having all distinct data in the set. But, it is always complex to determine a data set, which exhibits some average behavior. The following sections give a brief idea of the well-known accepted algorithms.

2.1 Numerical Algorithms

Numerical analysis is the theory of constructive methods in mathematical analysis. Constructive method is a procedure used to obtain the solution for a mathematical problem in finite number of steps and to some desired accuracy.

2.1.1 Numerical Iterative Algorithm

An iterative process can be illustrated with the flow chart given in fig 2.1. There are four main blocks in the process viz., initialization, decision, computation, and update. The functions of these four blocks are as follows:

1. Initialization: all parameters are set to their initial values.
2. Decision: decision parameter is used to determine when to exit from the loop.
3. Computation: required computation is performed.
4. Update: decision parameter is updated and is transformed for next iteration.

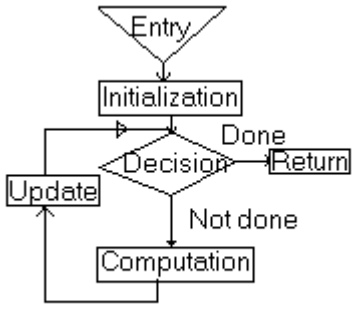


Fig 2.1 Flowchart for iterative process

Many problems in engineering or science need the solution of simultaneous linear algebraic equations. Every iterative algorithm is infinite step algorithm. One of the iterative algorithms to solve system of simultaneous equations is Guass Siedel. This iteration method requires generally a few iteration. Iterative techniques have less round-off error. For large system of equations, the iteration required may be quite large. But, there is a guarantee of getting the convergent result.

For example: consider the following set of equations,

$10x_1+2x_2+x_3= 9$
$2x_1+20x_2-2x_3= -44$
$-2x_1+3x_2+10x_3= 22.$
To solve the above set of equations using Guass Siedel iteration scheme, start with $(x_1^{(1)},x_2^{(1)},x_3^{(1)})=(0,0,0)$ as initial values and compute the values of we write the system of x_1, x_2, x_3 using the equations given below

$$x_1^{(k+1)}=(b_1-a_{12}x_2^{(k+1)}-a_{13}x_3^{(k)})/a_{11}$$
$$x_2^{(k+1)}=(b_2-a_{21}x_1^{(k+1)}-a_{23}x_3^{(k)})/a_{22}$$
$$x_3^{(k+1)}=(b_3-a_{31}x_1^{(k+1)}-a_{32}x_3^{(k+1)})/a_{33}$$
for k=1,2,3,…

This process is continued upto some desired accuracy. Numerical iterative methods are also applicable for obtaining the roots of the equation of the form f(x)=0. The various iterative methods used for this purpose are,

1. Bisection method: $x_{i+2}=(x_i+x_{i+1})/2$
2. Regula- Falsi method: $x_2=(x_0f(x_1)+ x_1f(x_0))/ (f(x_1)-f(x_0))$
3. Newton Raphson method: $x_2= x_1-f(x_1)/f^1(x_1)$

Review of Searching Algorithm:

Let us assume that we have a sequential file and we wish to retrieve an element matching with key 'k', then, we have to search the entire file from the beginning till the end to check whether the element matching k is present in the file or not.
There are a number of complex searching algorithms to serve the purpose of searching. The linear search and binary search methods are relatively straight forward methods of searching.

2.2.1 Sequential search

In this method, we start to search from the beginning of the list and examine each element till the end of the list. If the desired element is found we stop the search and return the index of that element. If the item is not found and the list is exhausted the search returns a zero value.
In the worst case the item is not found or the search item is the last (n^{th}) element. For both situations we must examine all n elements of the array so the order of magnitude or complexity of the sequential search is n. i.e., O(n). The execution time for this algorithm is proportional to n that is the algorithm executes in linear time.
The algorithm for sequential search is as follows,

Algorithm : sequential search
Input : A, vector of n elements
K, search element
Output : j –index of k
Method : i=1

```
While(i<=n)
{
if(A[i]=k)
{
write("search successful")
write(k is at location i)
exit();
}
else
i++
if end
while end
write (search unsuccessful);
algorithm ends.
```

2.2.2 Binary search

Binary search method is also relatively simple method. For this method it is necessary to have the vector in an alphabetical or numerically increasing order. A search for a particular item with X resembles the search

for a word in the dictionary. The approximate mid entry is located and its key value is examined. If the mid value is greater than X, then the list is chopped off at the $(mid-1)^{th}$ location. Now the list gets reduced to half the original list. The middle entry of the left-reduced list is examined in a similar manner. This procedure is repeated until the item is found or the list has no more elements. On the other hand, if the mid value is lesser than X, then the list is chopped off at $(mid+1)^{th}$ location. The middle entry of the right-reduced list is examined and the procedure is continued until desired key is found or the search interval is exhausted.
The algorithm for binary search is as follows,

Algorithm : binary search
Input : A, vector of n elements
K, search element
Output : low –index of k
Method : low=1,high=n

```
While(low<=high-1)
{
mid=(low+high)/2
if(k<a[mid])
high=mid
else
low=mid
if end
}
while end
if(k=A[low])
{
write("search successful")
write(k is at location low)
exit();
}
else
write (search unsuccessful);
if end;
algorithm ends.
```

Sorting:

One of the major applications in computer science is the sorting of information in a table. Sorting algorithms arrange items in a set according to a predefined ordering relation. The most common types of data are string information and numerical information. The ordering relation for numeric data simply involves arranging items in sequence from smallest to largest and from largest to smallest, which is called ascending and descending order respectively.

The items in a set arranged in non-decreasing order are {7,11,13,16,16,19,23}. The items in a set arranged in descending order is of the form {23,19,16,16,13,11,7}
Similarly for string information, {a, abacus, above, be, become, beyond}is in ascending order and { beyond, become, be, above, abacus, a}is in descending order.
There are numerous methods available for sorting information. But, not even one of them is best for all applications. Performance of the methods depends on parameters like, size of the data set, degree of relative order already present in the data etc.

2.3.1 Selection sort

The idea in selection sort is to find the smallest value and place it in an order, then find the next smallest and place in the right order. This process is continued till the entire table is sorted.
Consider the unsorted array,
a[1] a[2] a[8]

20 35 18 8 14 41 3 39

The resulting array should be
a[1] a[2] a[8]

3 8 14 18 20 35 39 41

One way to sort the unsorted array would be to perform the following steps:

- Find the smallest element in the unsorted array
- Place the smallest element in position of a[1]

  i.e., the smallest element in the unsorted array is 3 so exchange the values of a[1] and a[7]. The array now becomes,

a[1] a[2] a[8]

3 35 18 8 14 41 20 39

Now find the smallest from a[2] to a[8] , i.e., 8 so exchange the values of a[2] and a[4] which results with the array shown below,

a[1] a[2] a[8]

3 8 18 35 14 41 20 39

Repeat this process until the entire array is sorted. The changes undergone by the array is shown in fig 2.2.The number of moves with this technique is always of the order O(n).

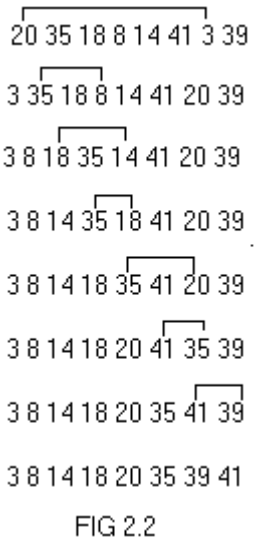


FIG 2.2

2.3.2 Insertion sort

Insertion sort is a straight forward method that is useful for small collection of data. The idea here is to obtain the complete solution by inserting an element from the unordered part into the partially ordered solution extending it by one element. Selecting an element from the unordered list could be simple if the first element of that list is selected.

a[1] a[2] a[8]

20 35 18 8 14 41 3 39

Initially the whole array is unordered. So select the minimum and put it in place of a[1] to act as sentinel. Now the array is of the form,

a[1] a[2] a[8]

3 35 18 8 14 41 20 39

Now we have one element in the sorted list and the remaining elements are in the unordered set. Select the next element to be inserted. If the selected element is less than the preceding element move the preceding element by one position and insert the smaller element.
In the above array the next element to be inserted is x=35, but the preceding element is 3 which is less than x. Hence, take the next element for insertion i.e., 18. 18 is less than 35, so move 35 one position ahead and place 18 at that place. The resulting array will be,
a[1] a[2] a[8]

3 18 35 8 14 41 20 39

Now the element to be inserted is 8. 8 is less than 35 and 8 is also less than 18 so move 35 and 18 one position right and place 8 at a[2]. This process is carried till the sorted array is obtained.
The changes undergone are shown in fig 2.3.

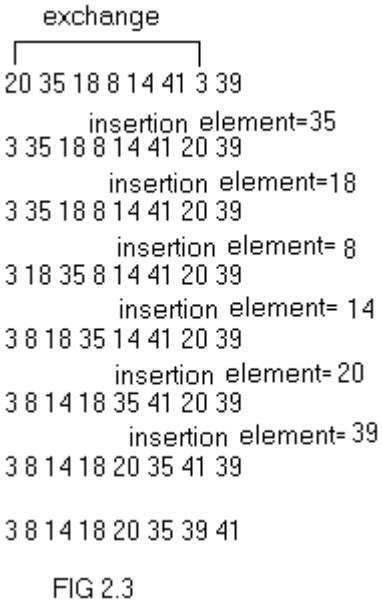


FIG 2.3

One of the disadvantages of the insertion sort method is the amount of movement of data. In the worst case, the number of moves is of the order $O(n^2)$. For lengthy records it is quite time consuming.

2.3.3 Merge sort

Merge sort begins by interpreting the inputs as n **sorted files** each of

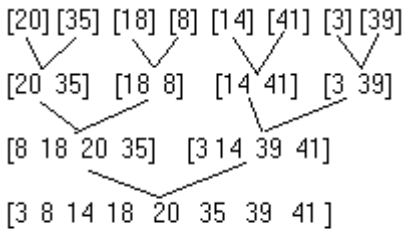


Fig 2.4 example for merge sort

length one. These are merged pair wise to obtain n/2 files of size two. If n is odd one file is of size one. These n/2 files are then merged pair wise and so on until we are left with only one file. The example in fig 2.4 illustrates the process of merge sort.
As illustrated in the example merge sort consists of several passes over the records being sorted. In the first pass files of size one are merged. In the second pass the size of the files being merged is two. In the i^{th} pass the files being merged will be of size 2^{i-1}. A total of $\log_2 n$ passes are made over the data. Since, two files can

be merged in linear time, each pass of merge sort takes O(n) time. As there are $\log_2 n$ passes the total time complexity is O(n $\log_2 n$).

Recursion:

Recursion may have the following definitions:
-The nested repetition of identical algorithm is recursion.
-It is a technique of defining an object/process by itself.
-Recursion is a process by which a function calls itself repeatedly until some specified condition has been satisfied.

2.4.1 When to use recursion

Recursion can be used for repetitive computations in which each action is stated in terms of previous result. There are two conditions that must be satisfied by any recursive procedure.

1. Each time a function calls itself it should get nearer to the solution.
2. There must be a decision criterion for stopping the process.

In making the decision about whether to write an algorithm in recursive or non-recursive form, it is always advisable to consider a tree structure for the problem. If the structure is simple then use non-recursive form. If the tree appears quite bushy, with little duplication of tasks, then recursion is suitable.
The recursion algorithm for finding the factorial of a number is given below,
Algorithm : factorial-recursion
Input : n, the number whose factorial is to be found.
Output : f, the factorial of n
Method : if(n=0)
f=1
else
f=factorial(n-1) * n
if end
algorithm ends.
The general procedure for any recursive algorithm is as follows,

1. Save the parameters, local variables and return addresses.
2. If the termination criterion is reached perform final computation and goto step 3 otherwise perform final computations and goto step 1
3. Restore the most recently saved parameters, local variable and return address and goto the latest return address.

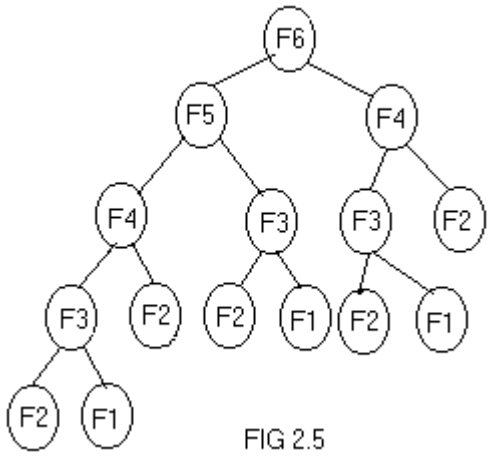


FIG 2.5

2.4.2 Iteration v/s Recursion

Demerits of recursive algorithms

1. Many programming languages do not support recursion, hence recursive mathematical function is implemented using iterative methods.
2. Even though mathematical functions can be easily implemented using recursion it is always at the cost of execution time and memory space. For example, the recursion tree for generating 6 numbers in a fibonacci series generation is given in fig 2.5. A fibonacci series is of the form 0,1,1,2,3,5,8,13,…etc, where the third number is the sum of preceding two numbers and so on. It can be noticed from the fig 2.5 that, f(n-2) is computed twice, f(n-3) is computed thrice, f(n-4) is computed 5 times.
3. A recursive procedure can be called from within or outside itself and to ensure its proper functioning it has to save in some order the return addresses so that, a return to the proper location will result when the return to a calling statement is made.
4. The recursive programs needs considerably more storage and will take more time.

Demerits of iterative methods

1. Mathematical functions such as factorial and fibonacci series generation can be easily implemented using recursion than iteration.
2. In iterative techniques looping of statement is very much necessary.

Recursion is a top down approach to problem solving. It divides the problem into pieces or selects out one key step, postponing the rest.
Iteration is more of a bottom up approach. It begins with what is known and from this constructs the solution step by step. The iterative function obviously uses time that is O(n) where as recursive function has an exponential time complexity.
It is always true that recursion can be replaced by iteration and stacks. It is also true that stack can be replaced by a recursive program with no stack.

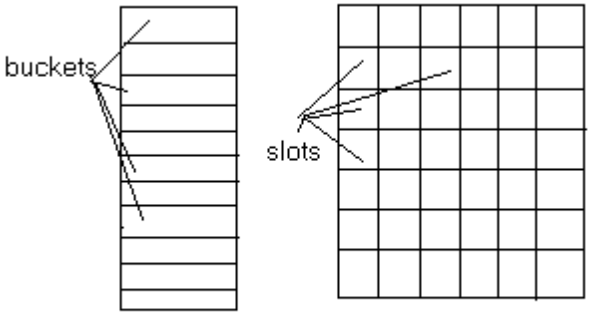


Fig 2.6

2.5 Hashing

Hashing is a practical technique of maintaining a symbol table. A symbol table is a data structure which allows to easily determine whether an arbitrary element is present or not.
Consider a sequential memory shown in fig 2.6. In hashing technique the address X of a variable x is obtained by computing an arithmetic function (hashing function) f(x). Thus f(x) points to the address where x should be placed in the table. This address is known as the hash address.
The memory used to store the variable using hashing technique is assumed to be sequential. The memory is known as hash table. The hash table is partitioned into several storing spaces called buckets and each bucket is divided into slots (fig 2.6).
If there are b buckets in the table, each bucket is capable of holding s variables, where each variable occupies one slot. The function f(x) maps the possible variable onto the integers 0 through b-1. The size of the space from where the variables are drawn is called the identifier space. Let T be the identifier space, n be the number of variables/identifiers in the hash table. Then, the ratio n/T is called the identifier density and a = n/sb is the loading density or loading factor.

Fig 2.7

If $f(x_1)=f(x_2)$, where $x_1$and x_2 are any two variables, then $x_1$and x_2 are called synonyms. Synonyms are mapped onto the same bucket. If a new identifier is hashed into a already complete bucket, collision occurs.

A hashing table with single slot is as given below. Let there be 26 buckets with single slot. The identifier to be stored are GA, D, A, G, L, A2, A1, A3, A4, Z, ZA, E. Let f(x) be the function which maps on to a address equal to the position of the first character of the identifier in the set of English alphabet. The hashing table generated is as shown in fig 2.7.
Time taken to retrieve the identifiers is as follows,

Search element (x)	Search time (t)
GA	1
D	1
A	1
G	2
L	1
A2	2
A1	3
A3	5
A4	6
Z	1
ZA	10
E	6
	∑t =39

Average retrieval time =(Σt)/n.
The average retrieval time entirely depends on the hashing function.

Exercise 2:

1. What are the serious short comings of the binary search method and sequential search method.
2. Know more searching techniques involving hashing functions
3. Implement the algorithms for searching and calculate the complexities
4. Write an algorithm for the above method of selection sort and implement the same.
5. Write the algorithm for merge sort method
6. Take 5 data set of length 10 and hand simulate for each method given above.
7. Try to know more sorting techniques and make a comparative study of them.
8. Write an iterative algorithm to find the factorial of a number
9. Write a recursive and iterative program for reversing a number
10. Write recursive and iterative program to find maximum and minimum in a list of numbers.
11. Write an algorithm to implement the hashing technique and implement the same
12. Hand simulate all algorithms for a 5 datasets.

Introduction to Graph Theory:

3.1

3.1.1 What is graph?

A graph G = (V, E) consists of a set of objects V = {v_1, v_2, …} called vertices, and another set E = {e_1, e_2, …} whose elements are called edges. Each edge e_k in E is identified with an unordered pair (v_i, v_j) of vertices. The vertices v_i, v_j associated with edge e_k are called the end vertices of e_k.

The most common representation of graph is by means of a diagram, in which the vertices are represented as points and each edge as a line segment joining its end vertices. Often this diagram itself is referred to as a graph.

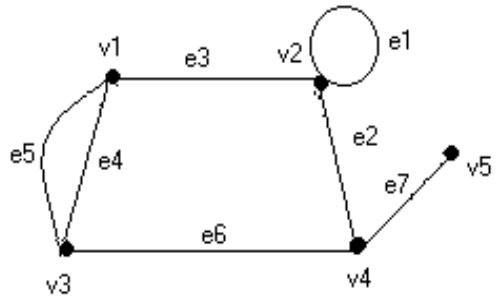


Fig 3-1.

In the Fig. 3-1 edge e_1 having same vertex as both its end vertices is called a self-loop. There may be more than one edge associated with a given pair of vertices, for example e4 and e5 in Fig. 3-1. Such edges are referred to as parallel edges.

A graph that has neither self-loop nor parallel edges are called a ***simple graph***, otherwise it is called ***general graph***. It should also be noted that, in drawing a graph, it is immaterial

whether the lines are drawn straight or curved, long or short: what is important is the incidence between the edges and vertices.
A graph is also called a *linear complex*, a *1-complex*, or a *one-dimensional complex*. A vertex is also referred to as a *node*, a *junction*, a *point*, *0-cell*, or an *0-simplex*. Other terms used for an edge are a *branch*, a *line*, an *element*, a *1-cell*, an *arc*, and a *1-simplex*.
Because of its inherent simplicity, graph theory has a very wide range of applications in engineering, physical, social, and biological sciences, linguistics, and in numerous other areas. A graph can be used to represent almost any physical situation involving discrete objects and a relationship among them.

3.1.2 Finite and Infinite Graphs

Although in the definition of a graph neither the vertex set V nor the edge set E need be finite, in most of the theory and almost all applications these sets are finite. A graph with a finite number of vertices as well as a finite number of edges is called a *finite graph*; otherwise, it is an *infinite graph*.

3.1.3 Incidence and Degree

When a vertex v_i is an end vertex of some edge e_j, v_i and e_j are said to be incident with (on or to) each other. In Fig. 3-1, for example, edges e_2, e_6, and e_7 are incident with vertex v_4.
Two nonparallel edges are said to be adjacent if they are incident on a common vertex. For example, e_2 and e_7 in Fig. 3-1 are adjacent. Similarly, two vertices are said to be adjacent if they are the end vertices of the same edge. In Fig. 3-1, v_4 and v_5 are adjacent, but v_1 and v_4 are not.
The number of edges incident on a vertex v_i, with self-loops counted twice is called the degree, $d(v_i)$, of vertex v_i. In Fig. 3-1, for example, $d(v_1) = d(v_3) = d(v_4) = 3$, $d(v_2) = 4$, and $d(v_5) = 1$. The degree of a vertex is sometimes also referred to as its *valency*. Since each edge contributes two degrees, the sum of the degrees of all vertices in G is twice the number of edges in G.

3.1.4 Isolated vertex, Pendent vertex, and Null graph

A vertex having no incident edge is called an*isolated vertex*. In other words, isolated vertices are vertices with zero degree. Vertex v_4 and v_7 in Fig. 3-2, for example, are isolated vertices. A vertex of degree one is called a *pendent vertex* or an *end vertex*. Vertex v_3 in Fig. 3-2 is a pendant vertex. Two adjacent edges are said to be in*series* if their common vertex is of degree two. In Fig. 3-2, the two edges incident on v_1 are in series.

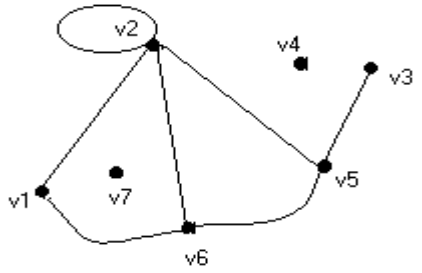
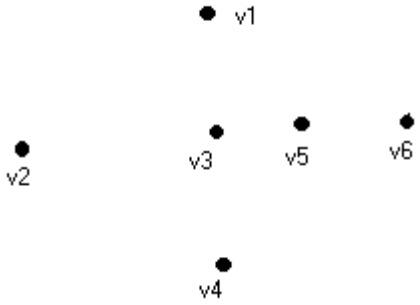


Fig. 3-2 Graph containing isolated vertices, series edges and a pendant vertex.
In the definition of a graph G = (V, E), it is possible for the edge set E to be empty. Such a graph, without any edges, is called a null graph. In other words, every vertex in a null graph is an isolated vertex. A null graph of six vertices is shown in Fig. 3-3. Although the edge set E may be empty, the vertex set V must not be empty; otherwise, there is no graph. In other words, by definition, a graph must have at least one vertex.



Matrix Representation of Graph:

Although a pictorial representation of a graph is very convenient for a visual study, other representations are better for computer processing. A matrix is a convenient and useful way of representing a graph to a computer.

Matrices lend themselves easily to mechanical manipulations. Besides, many known results of matrix algebra can be readily applied to study the structural properties of graphs from an algebraic point of view. In many applications of graph theory, such as in electrical network analysis and operation research, matrices also turn out to be the natural way of expressing the problem.

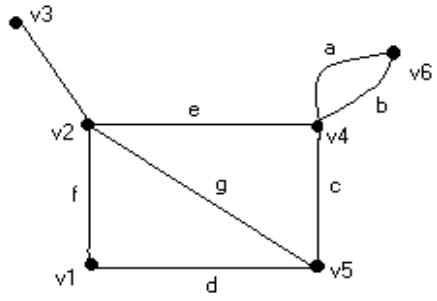
3.2.1 Incidence Matrix

Let G be a graph with n vertices, e edges, and no self-loops. Define an n by e matrix A =[a_{ij}], whose n rows correspond to the n vertices and the e columns correspond to the e edges, as follows:

The matrix element

A_{ij} = 1, if j^{th} edge e_j is incident on i^{th} vertex v_i, and

= 0, otherwise.



(a)

a b c d e f g h

v1 0 0 0 1 0 1 0 0
v2 0 0 0 0 1 1 1 1
v3 0 0 0 0 0 0 0 1
v4 1 1 1 0 1 0 0 0
v5 0 0 1 1 0 0 1 0

v6 1 1 0 0 0 0 0 0

(b)

Fig. 3-4 Graph and its incidence matrix.

Such a matrix A is called the *vertex-edge incidence matrix*, or simply *incidence matrix*. Matrix A for a graph G is sometimes also written as A(G). A graph and its incidence matrix are shown in Fig. 3-4. The incidence matrix contains only two elements, 0 and 1. Such a matrix is called a*binary matrix* or a *(0, 1)-matrix*.
The following observations about the incidence matrix A can readily be made:

1. Since every edge is incident on exactly two vertices, each column of A has exactly two 1's.
2. The number of 1's in each row equals the degree of the corresponding vertex.
3. A row with all 0's, therefore, represents an isolated vertex.
4. Parallel edges in a graph produce identical columns in its incidence matrix, for example, columns 1 and 2 in Fig. 3-4.

Concept of Trees:

The concept of a *tree* is probably the most important in graph theory, especially for those interested in applications of graphs.
A *tree* is a connected graph without any circuits. The graph in Fig 3-5 for instance, is a tree. It follows immediately from the definition that a tree has to be a simple graph, that is, having neither a self-loop nor parallel edges (because they both form circuits).

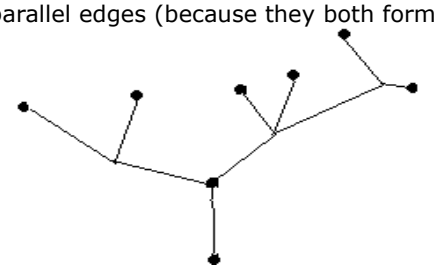


Fig. 3-5. Tree
Trees appear in numerous instances. The genealogy of a family is often represented by means of a tree. A river with its tributaries and sub-tributaries can also be represented by a tree. The sorting of mail according to zip code and the sorting of punched cards are done according to a tree (called*decision tree* or *sorting tree*).

3.3.1 Some properties of Trees

1. There is one and only one path between every pair of vertices in a tree, T.
2. A tree with n vertices has n-1 edges.
3. Any connected graph with n vertices and n-1 edges is a tree.
4. A graph is a tree if and only if it is minimally connected.

Therefore a graph with n vertices is called a tree if

1. G is connected and is circuit less, or
2. G is connected and has n-1 edges, or
3. G is circuit less and has n-1 edges, or
4. There is exactly one path between every pair of vertices in G, or
5. G is a minimally connected graph.

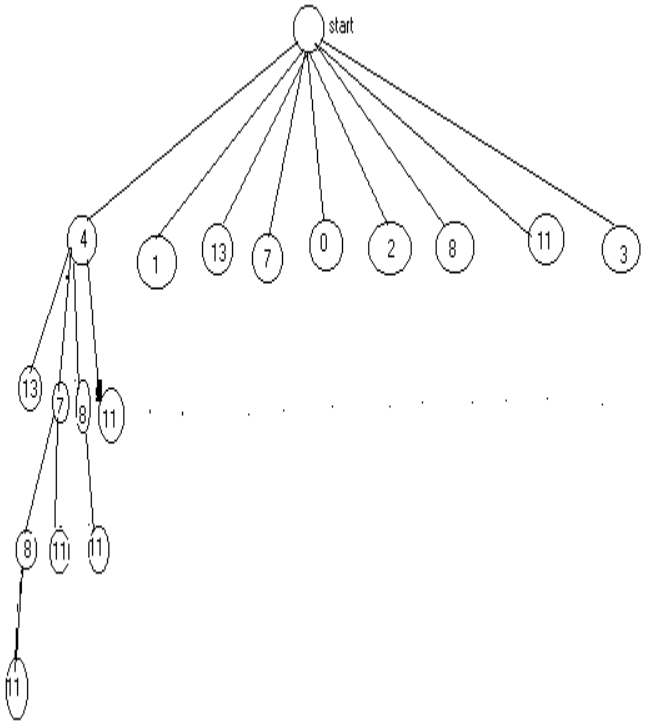


Fig. 3-6 Tree of a monotonically increasing sequences in 4,1,13,7,0,2,8,11,3

3.3.2 Pendent Vertices in a Tree

It is observed that a tree shown in the Fig. 3-5 has several pendant vertices. A pendant vertex was defined as a vertex of degree one). The reason is that in a tree of n vertices we have n-1 edges, and hence 2(n-1) degrees to be divided among n vertices. Since no vertex can be of zero degree, we must have at least two vertices of degree one in a tree. This makes sense only if $n \geq 2$.

An Application: The following problem is used in teaching computer programming. Given a sequence of integers, no two of which are the same find the largest monotonically increasing subsequence in it. Suppose that the sequence given to us is 4,1,13,7,0,2,8,11,3; it can be represented by a tree in which the vertices (except the start vertex) represent individual numbers in the sequence, and the path from the start vertex to a particular vertex *v* describes the monotonically increasing subsequence terminating in *v*.

As shown in Fig. 3-6, this sequence contains four longest monotonically increasing subsequences, that is, (4,7,8,11), (1,7,8,11), (1,2,8,11) and (0,2,8,11). Each is of length four. Computer programmers refer to such a tree used in representing data as a data tree.

3.3.3 Rooted and Binary Tree

A tree in which one vertex (called the *root*) is distinguished from all the others is called a *rooted tree*. For instance, in Fig. 3-6 vertex named *start*, is distinguished from the rest of the vertices. Hence vertex *start* can be considered the root of the tree, and so the tree is rooted. Generally, the term*tree* means trees without any root. However, for emphasis they are sometimes called *free trees* (or *non rooted trees*) to differentiate them from the rooted kind.

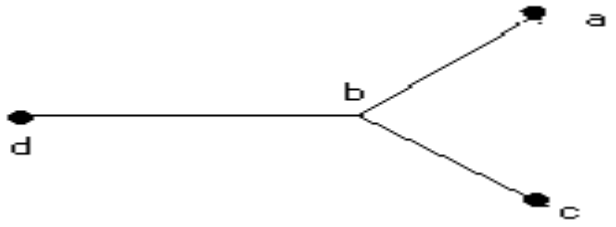


Fig. 3-6 Tree.

Binary Trees: A special class of rooted trees, called binary rooted trees, is of particular interest, since they are extensively used in the study of computer search methods, binary identification problems, and variable-length binary codes. A *binary tree* is defined as a tree in which there is exactly one vertex of degree two,

and each of the remaining vertices of degree one or three. Since the vertex of degree two is distinct from all other vertices, this vertex serves as a root. Thus every binary tree is a rooted tree.

3.3.4 Spanning Trees

So far we have discussed the trees when it occurs as a graph by itself. Now we shall study the tree as a subgraph of another graph. A given graph has numerous subgraphs, from e edges, 2^e distinct combinations are possible. Obviously, some of these subgrphs will be trees. Out of these trees we are particularly interested in certain types of trees, called *spanning trees*.

A tree T is said to be a spanning tree of a connected graph G if T is a subgraph of G and T contains all vertices of G. Since the vertices of G are barely hanging together in a spanning tree, it is a sort of skeleton of the original graph G. This is why a spanning tree is sometimes referred to as a *skeleton*or *scaffolding* of G. Since spanning trees are the largest trees among all trees in G, it is also quite appropriate to call a spanning tree a *maximal tree subgraph* or *maximal tree of*G.

Finding a spanning tree of a connected graph G is simple. If G has no circuit, it is its own spanning tree. If G has a circuit, delete an edge from the circuit. This will still leave the graph connected. If there are more circuits, repeat the operation till an edge from the last circuit is deleted, leaving a connected, circuit-free graph that contains all the vertices of G.

3.3.5 Hamiltonian Paths and Circuits

Hamiltonian circuit in a connected graph is defined as a closed walk that traverses *every vertex* of G exactly once, except of course the starting vertex, at which the walk also terminates. A circuit in a connected graph G is said to be Hamiltonian if it includes every vertex of G. Hence a Hamiltonian circuit in a graph of n vertices consists of exactly n edges.

Hamiltonian path: If we remove any one edge from a Hamiltonian circuit, we are left with a path. This path is called a *Hamiltonian path.* Clearly, a Hamiltonian path in a graph G traverses every vertex of G. Since a Hamiltonian path is a subgraph of a Hamiltonian circuit (which in turn is a subgraph of another graph), every graph that has a Hamiltonian circuit also has a Hamiltonian path. There are, however, many graphs with Hamiltonian paths that have no Hamiltonian circuits. The length of a Hamiltonian path in a connected graph of n vertices is n-1.

3.3.5 Traveling-Salesman Problem

A problem closely related to the question of Hamiltonian circuits is the *Traveling-salesman problem*, stated as follows: A salesman is required to visit a number of cities during a trip. Given the distances between the cities, in what order should he travel so as to visit every city precisely once and return home, with the minimum mileage traveled?

Representing the cities by vertices and the roads between them by edges, we get a graph. In this graph, with every edge e_i there is associated a real number (the distance in miles, say), $w(e_i)$. Such a graph is called a *weighted graph;*$w(e_i)$ being the weight of edge e_i.

In our problem, if each of the cities has a road to every other city, we have a *complete weighted graph*. This graph has numerous Hamiltonian circuits, and we are to pick the one that has the smallest sum of distances (or weights).

The total number of different (not edge disjoint, of course) Hamiltonian circuits in a complete graph of n vertices can be shown to be (n-1)! / 2. This follows from the fact that starting from any vertex we have n-1 edges to choose from the first vertex, n-2 from the second, n-3 from the third, and so on. These being independent, results with (n-1)! choices. This number is, however, divided by 2, because each Hamiltonian circuit has been counted twice.

Theoretically, the problem of the traveling salesman can always be solved by enumerating all (n-1)!/2 Hamiltonian circuits, calculating the distance traveled in each, and then picking the shortest one. However, for a large value of n, the labor involved is too great even for a digital computer.

The problem is to prescribe a manageable algorithm for finding the shortest route. No efficient algorithm for problems of arbitrary size has yet been found, although many attempts have been made. Since this problem has applications in operations research, some specific large-scale examples have been worked out. There are also available several heuristic methods of solution that give a route very close to the shortest one, but do not guarantee the shortest.

Exercise 3

1. Draw all simple graphs of one, two, three and four vertices
2. Name 10 situations that can be represented by means of graphs. Explain what each vertex and edge represent
3. Draw a connected graph that becomes disconnected when any edge is removed from it
4. Draw all trees of n labeled vertices for n=1,2,3,4 and 5
5. Sketch all binary trees with six pendent edges

6. Sketch all spanning trees of given graphs in this chapter
7. Write incidence matrix for all the graphs developed
8. Find the spanning trees for all the graphs developed
9. Draw a graph which has Hamiltonian path but does not have Hamiltonian circuit
10. List different paths from vertex1 to vertex n in each graph developed.

Divide and conquer:

There are a number of general and powerful computational strategies that are repeatedly used in computer science. It is often possible to phrase any problem in terms of these general strategies. These general strategies are Divide and Conquer, Dynamic Programming. The techniques of Greedy Search, Backtracking and Branch and Bound evaluation are variations of dynamic programming idea. All these strategies and techniques are discussed in the subsequent chapters.

The most widely known and often used of these is the divide and conquer strategy.
The basic idea of divide and conquer is to divide the original problem into two or more sub-problems which can be solved by the same technique. If it is possible to split the problem further into smaller and smaller sub-problems, a stage is reached where the sub-problems are small enough to be solved without further splitting. Combining the solutions of the individuals we get the final conquering. Combining need not mean, simply the union of individual solutions.
Divide and Conquer involves four steps

1. Divide
2. Conquer [Initial Conquer occurred due to solving]
3. Combine
4. Conquer [Final Conquer].

In precise, forward journey is divide and backward journey is Conquer. A general binary divide and conquer algorithm is :

```
Procedure D&C (P,Q) //the data size is from p to q
{
If size(P,Q) is small Then
Solve(P,Q)
Else
M ← divide(P,Q)
Combine (D&C(P,M), D&C(M+1,Q))
}
```

Sometimes, this type of algorithm is known as control abstract algorithms as they give an abstract flow. This way of breaking down the problem has found wide application in sorting, selection and searching algorithm.

4.1 Binary Search:

Algorithm:

```
m← (p+q)/2
If (p ≤ m ≤ q) Then do the following Else Stop
If (A(m) = Key Then 'successful' stop
Else
If (A(m) < key Then
q=m-1;
Else
```

p← m+1
End Algorithm.

Illustration :

Consider the data set with elements {12,18,22,32,46,52,59,62,68}. First let us consider the simulation for successful cases.

Successful cases:

Key=12 P Q m Search
1 9 5 x
1 4 2 x
1 1 1 successful
To search 12, 3 units of time is required

Key=18 P Q m Search
1 9 5 x
1 4 2 successful
To search 18, 2 units of time is required
Key=22 P Q m Search
1 9 5 x
1 4 2 x
3 4 3 successful
To search 22, 3 units of time is required
Key=32 P Q m Search
1 9 5 x
1 4 2 x
3 4 3 x
4 4 4 successful
To search 32, 4 units of time is required
Key=46 P Q m Search
1 9 5 successful
To search 46, 1 unit of time is required
Key=52 P Q m Search
1 9 5 x
6 9 7 x
6 6 6 successful
To search 52, 3 units of time is required
Key=59 P Q m Search
1 9 5 x
6 9 7 successful
To search 59, 2 units of time is required
Key=62 P Q m Search
1 9 5 x
6 9 7 x
8 9 8 successful
To search 62, 3 units of time is required
Key=68 P Q m Search
1 9 5 x
6 9 7 x
8 9 8 x
9 9 9 successful
To search 68, 4 units of time is required

3+2+3+4+1+3+2+4
Successful average search time= ----------

9
unsuccessful cases
Key=25 P Q m Search
1 9 5 x
1 4 2 x
3 4 3 x
4 4 4 x
To search 25, 4 units of time is required

Key=65 P Q m Search
1 9 5 x
6 9 7 x
8 9 8 x
9 9 9 x
To search 65, 4 units of time is required
4+4
Unsuccessful search time =----------
2
average (sum of unsuccessful search time
search = + sum of Successful search time)/(n+(n+1))
time

Study Notes Home | Next Section>>

4.1 Divide and Conquer

There are a number of general and powerful computational strategies that are repeatedly used in computer science. It is often possible to phrase any problem in terms of these general strategies. These general strategies are Divide and Conquer, Dynamic Programming. The techniques of Greedy Search, Backtracking and Branch and Bound evaluation are variations of dynamic programming idea. All these strategies and techniques are discussed in the subsequent chapters.

The most widely known and often used of these is the divide and conquer strategy.
The basic idea of divide and conquer is to divide the original problem into two or more sub-problems which can be solved by the same technique. If it is possible to split the problem further into smaller and smaller sub-problems, a stage is reached where the sub-problems are small enough to be solved without further splitting. Combining the solutions of the individuals we get the final conquering. Combining need not mean, simply the union of individual solutions.
Divide and Conquer involves four steps

1. Divide
2. Conquer [Initial Conquer occurred due to solving]
3. Combine
4. Conquer [Final Conquer].

In precise, forward journey is divide and backward journey is Conquer. A general binary divide and conquer algorithm is :
Procedure D&C (P,Q) //the data size is from p to q

{
If size(P,Q) is small Then
Solve(P,Q)
Else
M ← divide(P,Q)
Combine (D&C(P,M), D&C(M+1,Q))
}
Sometimes, this type of algorithm is known as control abstract algorithms as they give an abstract flow. This way of breaking down the problem has found wide application in sorting, selection and searching algorithm.

4.1 Binary Search:

Algorithm:

m← (p+q)/2
If (p ≤ m ≤ q) Then do the following Else Stop
If (A(m) = Key Then 'successful' stop
Else
If (A(m) < key Then
q=m-1;
Else
p← m+1
End Algorithm.

Illustration :

Consider the data set with elements {12,18,22,32,46,52,59,62,68}. First let us consider the simulation for successful cases.

Successful cases:

Key=12 P Q m Search
1 9 5 x
1 4 2 x
1 1 1 successful
To search 12, 3 units of time is required

Key=18 P Q m Search
1 9 5 x
1 4 2 successful
To search 18, 2 units of time is required
Key=22 P Q m Search
1 9 5 x
1 4 2 x
3 4 3 successful
To search 22, 3 units of time is required
Key=32 P Q m Search
1 9 5 x
1 4 2 x
3 4 3 x
4 4 4 successful
To search 32, 4 units of time is required
Key=46 P Q m Search
1 9 5 successful
To search 46, 1 unit of time is required
Key=52 P Q m Search
1 9 5 x
6 9 7 x
6 6 6 successful

To search 52, 3 units of time is required
Key=59 P Q m Search
1 9 5 x
6 9 7 successful
To search 59, 2 units of time is required
Key=62 P Q m Search
1 9 5 x
6 9 7 x
8 9 8 successful
To search 62, 3 units of time is required
Key=68 P Q m Search
1 9 5 x
6 9 7 x
8 9 8 x
9 9 9 successful
To search 68, 4 units of time is required

3+2+3+4+1+3+2+4
Successful average search time= ----------
9

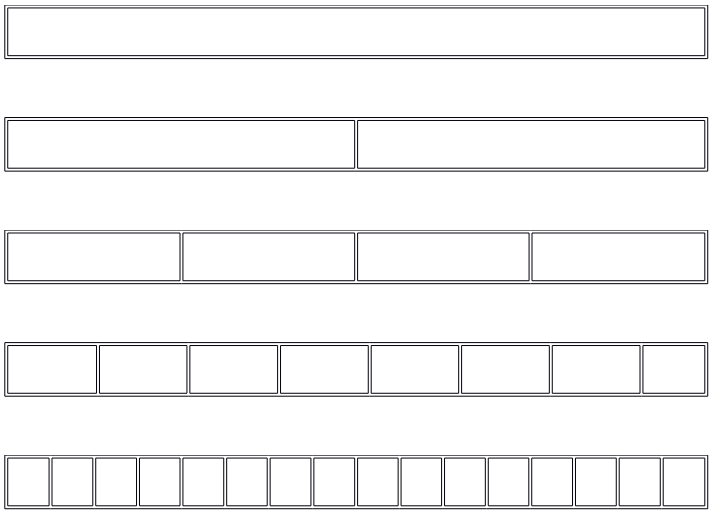
unsuccessful cases

Key=25 P Q m Search
1 9 5 x
1 4 2 x
3 4 3 x
4 4 4 x
To search 25, 4 units of time is required

Key=65 P Q m Search
1 9 5 x
6 9 7 x
8 9 8 x
9 9 9 x
To search 65, 4 units of time is required
4+4
Unsuccessful search time =----------
2
average (sum of unsuccessful search time
search = + sum of Successful search time)/(n+(n+1))
time

Max-Min Search:

Max-Min search problem aims at finding the smallest as well as the biggest element in a vector A of n elements.
Following the steps of Divide and Conquer the vector can be divided into sub-problem as shown below.



The search has now reduced to comparison of 2 numbers. The time is spent in conquering and comparing which is the major step in the algorithm.

Algorithm: Max-Min (p, q, max, min)
{
If (p = q) Then
max = a(p)
min = a(q)
Else
If (p – q-1) Then
If a(p) > a(q) Then
max = a(p)
min = a(q)
Else
max = a(q)
min = a(p)
If End
Else
m ← (p+q)/2
max-min(p,m,max1,min1)
max-min(m+1,q,max2,min2)
max ← large(max1,max2)
min ← small(min1,min2)
If End
If End
Algorithm End.
Illustration

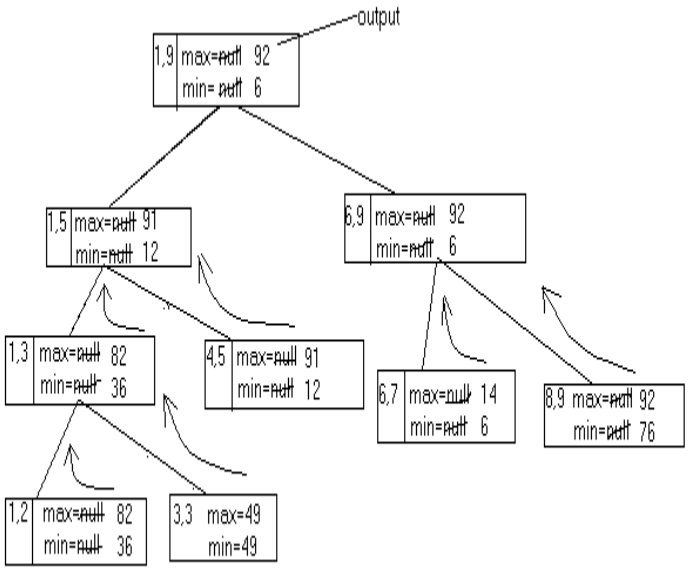


Fig 4.1

Consider a data set with elements {82,36,49,91,12,14,06,76,92}. Initially the max and min variables have null values. In the first call, the list is broken into two equal halves.. The list is again broken down into two. This process is continued till the length of the list is either two or one. Then the maximum and minimum values are chosen from the smallest list and these values are returned to the preceding step where the length of the list is slightly big. This process is continued till the entire list is searched. The detail description is shown in fig 4.1

Integer Multiplication:

There are various methods of obtaining the product of two numbers. The repeated addition method is left as an assignment for the reader. The reader is expected to find the product of some bigger numbers using the repeated addition method.
Another way of finding the product is the one we generally use i.e., the left shift method.

4.3.1 left shift method

981*1234
3924
2943*
1962**
981***

1210554

In this method, a=981 is the multiplicand and b=1234 is the multiplier. A is multiplied by every digit of b starting from right to left. On each multiplication the subsequent products are shifted one place left. Finally the products obtained by multiplying a by each digit of b is summed up to obtain the final product.

The above product can also be obtained by a right shift method, which can be illustrated as follows,

4.3.2 right shift method

981*1234

981
1962
*2943
**3924

1210554

In the above method, a is multiplied by each digit of b from leftmost digit to rightmost digit. On every multiplication the product is shifted one place to the right and finally all the products obtained by multiplying 'a' by each digit of 'b' is added to obtain the final result. The product of two numbers can also be obtained by dividing 'a' and multiplying 'b' by 2 repeatedly until a<=1.

4.3.3 halving and doubling method

Let a=981 and b=1234

The steps to be followed are

1. If a is odd store b
2. A=a/2 and b=b*2
3. Repeat step 2 and step 1 till a<=1

a	b	result
981	1234	1234
490	2468	----------
245	4936	4936
122	9872	---------
61	19744	19744
30	39488	----------
15	78976	78976
7	157952	157952
3	315904	315904
1	631808	631808

Sum=1210554

The above method is called the halving and doubling method.

4.3.4 Speed up algorithm:

In this method we split the number till it is easier to multiply. i.e., we split 0981 into 09 and 81 and 1234 into 12 and 34. 09 is then multiplied by both 12 and 34 but, the products are shifted 'n' places left before adding. The number of shifts 'n' is decided as follows

Multiplication sequence	shifts	
09*12	4	108****
09*34	2	306**
81*12	2	972**
81*34	0	2754

Sum=1210554
For 0981*1234, multiplication of 34 and 81 takes zero shifts, 34*09 takes 2 shifts, 12 and 81 takes 2 shifts and so on.

Exercise 4

1. Write the algorithm to find the product of two numbers for all the methods explained.
2. Hand simulate the algorithm for atleast 10 different numbers.
3. Implement the same for verification.
4. Write a program to find the maximum and minimum of the list of n element with and without using recursion.

Greedy Method:

Greedy method is a method of choosing a subset of the dataset as the solution set that results in some profit. Consider a problem having n inputs, we are required to obtain the solution which is a series of subsets that satisfy some constraints or conditions. Any subset, which satisfies these constraints, is called a feasible solution. It is required to obtain the feasible solution that maximizes or minimizes the objective function. This feasible solution finally obtained is called optimal solution.

If one can devise an algorithm that works in stages, considering one input at a time and at each stage, a decision is taken on whether the data chosen results with an optimal solution or not. If the inclusion of a particular data results with an optimal solution, then the data is added into the partial solution set. On the other hand, if the inclusion of that data results with infeasible solution then the data is eliminated from the solution set.
The general algorithm for the greedy method is

1. Choose an element e belonging to dataset D.
2. Check whether e can be included into the solution set S if Yes solution set is s $\leftarrow$ s U e.
3. Continue until s is filled up or D is exhausted whichever is earlier.

5.1 Cassette Filling

Consider n programs that are to be stored on a tape of length L. Each program I is of length l_i where i lies between 1 and n. All programs can be stored on the tape iff the sum of the lengths of the programs is at most L. It is assumed that, whenever a program is to be retrieved the tape is initially positioned at the start end.
Let t_j be the time required retrieving program $i_j$where programs are stored in the order $I = i_1, i_2, i_3, \ldots, i_n$.
The time taken to access a program on the tape is called the mean retrieval time (MRT)
i.e., $t_j = \Sigma\ l_{ik}$ k=1,2,…,j

Now the problem is to store the programs on the tape so that MRT is minimized. From the above discussion one can observe that the MRT can be minimized if the programs are stored in an increasing order i.e., $l1 \leq l2 \leq l3, \ldots \leq ln$.
Hence the ordering defined minimizes the retrieval time. The solution set obtained need not be a subset of data but may be the data set itself in a different sequence.

Illustration
Assume that 3 sorted files are given. Let the length of files A, B and C be 7, 3 and 5 units respectively. All these three files are to be stored on to a tape S in some sequence that reduces the average retrieval time. The table shows the retrieval time for all possible orders.

Order of recording	Retrieval time	MRT
ABC	7+(7+3)+(7+3+5)=32	32/3
ACB	7+(7+5)+(7+5+3)=34	34/3
BAC	3+(3+7)+(3+7+5)=28	28/3
BCA	3+(3+5)+(3+5+7)=26	26/3
CAB	5+(5+7)+(5+7+3)=32	32/3
CBA	5+(5+3)+(5+3+7)=28	28/3

General knapsack Problem:

Greedy method is best suited to solve more complex problems such as a knapsack problem. In a knapsack problem there is a knapsack or a container of capacity M n items where, each item i is of weight w_i and is associated with a profit p_i.

The problem of knapsack is to fill the available items into the knapsack so that the knapsack gets filled up and yields a maximum profit. If a fraction x_i of object i is placed into the knapsack, then a profit $p_i x_i$ is earned. The constrain is that all chosen objects should sum up to M
Illustration
Consider a knapsack problem of finding the optimal solution where, M=15, (p1,p2,p3…p7) = (10, 5, 15, 7, 6, 18, 3) and (w1, w2, …., w7) = (2, 3, 5, 7, 1, 4, 1).
In order to find the solution, one can follow three different srategies.
Strategy 1 : **non-increasing profit values**
Let (a,b,c,d,e,f,g) represent the items with profit (10,5,15,7,6,18,3) then the sequence of objects with non-increasing profit is (f,c,a,d,e,b,g).
Item chosen for inclusion
Quantity of item included
Remaining space in M
P_iX_i
f
1 full unit
15-4=11
18*1=18

C
1 full unit
11-5=6

15*1=15

A
1 full unit
6-2=4

10*1=10

d
4/7 unit
4-4=0

4/7*7=04

Profit= 47 units
The solution set is (1,0,1,4/7,0,1,0).

Strategy 2: non-decreasing weights

The sequence of objects with non-decreasing weights is (e,g,a,b,f,c,d).
Item chosen for inclusion
Quantity of item included
Remaining space in M
P_iX_I
E
1 full unit
15-1=14

6*1=6

G
1 full unit
14-1=13

3*1=3

A
1 full unit
13-2=11

10*1=10

b
1 full unit
11-3=8

5*1=05

f
1 full unit
8-4=4

18*1=18

c
4/5 unit
4-4=0

4/5*15=12

Profit= 54 units
The solution set is (1,1,4/5,0,1,1,1).

Strategy 2: maximum profit per unit of capacity used

(This means that the objects are considered in decreasing order of the ratio P_i/w_I)

a: P_1/w_1 =10/2 = 5 b: P_2/w_2 =5/3=1.66 c: P_3/w_3=15/5 = 3

d: P_4/w_4 =7/7=1 e: P_5/w_5 =6/1=6 f: P_6/w_6 =18/4 = 4.5

g: P_7/w_7 =3/1=3

Hence, the sequence is (e,a,f,c,g,b,d)

Item chosen for inclusion
Quantity of item included
Remaining space in M
P_iX_I
E
1 full unit
15-1=14

6*1=6

A
1 full unit
14-2=12

10*1=10

F
1 full unit
12-4=8

18*1=18

C
1 full unit
8-5=3

15*1=15

g
1 full unit
3-1=2

3*1=3

b
2/3 unit
2-2=0

2/3*5=3.33

Profit= 55.33 units

The solution set is (1,2/3,1,0,1,1,1).

In the above problem it can be observed that, if the sum of all the weights is ≤ M then all $x_i = 1$, is an optimal solution. If we assume that the sum of all weights exceeds M, all x_i's cannot be one. Sometimes it becomes necessary to take a fraction of some items to completely fill the knapsack. This type of knapsack problems is a general knapsack problem.

Concept of back Tracking:

Problems, which deal with searching a set of solutions, or which ask for an optimal solution satisfying some constraints can be solved using the backtracking formulation. The backtracking algorithm yields the proper solution in fewer trials.
The basic idea of backtracking is to build up a vector one component at a time and to test whether the vector being formed has any chance of success. The major advantage of this algorithm is that if it is realized that the partial vector generated does not lead to an optimal solution then that vector may be ignored.
Backtracking algorithm determine the solution by systematically searching the solution space for the given problem. This search is accomplished by using a free organization.

Backtracking is a depth first search with some bounding function. All solutions using backtracking are required to satisfy a complex set of constraints. The constraints may be explicit or implicit.
Explicit constraints are rules, which restrict each vector element to be chosen from the given set. Implicit constraints are rules, which determine which of the tuples in the solution space, actually satisfy the criterion function.

6.1.1 Cassette filling problem:

There are n programs that are to be stored on a tape of length L. Every program 'i' is of length l_i. All programs can be stored on the tape if and only if the sum of the lengths of the programs is at most L. In this problem, it is assumed that whenever a program is to be retrieved, the tape is positioned at the start end. Hence, the time t_j needed to retrieve program $i_j$from a tape having the programs in the order $i_1,i_2, \ldots,i_n$ is called mean retrieval time(MRT) and is given by

$t_j = \Sigma\ l_{ik}$ k=1,2,…,j

In the optimal storage on tape problem, we are required to find a permutation for the n programs so that when they are stored on the tape, the MRT is minimized.
Let n=3 and $(l_1,l_2,l_3)=(5,10,3)$,there are n!=6 possible orderings. These orderings and their respective MRT is given in the fig 6.1. Hence, the best order of recording is 3,1,2.

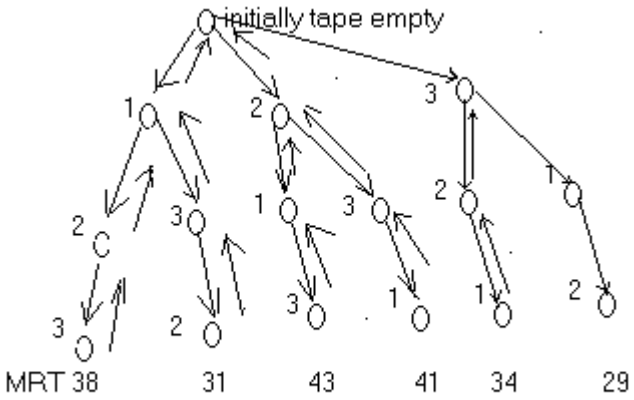


FIG 6.1

6.1.2 Subset problem:

There are n positive numbers given in a set. The desire is to find all possible subsets of this set, the contents of which add onto a predefined value M.
Let there be n elements in the main set. W=w[1..n] represent the elements of the set. i.e., w = (w1,w2,w3,…,wn) vector x = x[1..n] assumes either 0 or 1 value. If element w(i) is included in the subset then x(i) =1.
Consider n=6 m=30 and w[1..6]={5,10,12,13,15,18}. The partial backtracking tree is shown in fig 6.2. The label to the left of a node represents the item number chosen for insertion and the label to the right represents the space occupied in M. S represents a solution to the given problem and B represents a bounding criteria if no solution can be reached. For the above problem the solution could be (1,1,0,0,1,0), (1,0,1,1,0,0) and (0,0,1,0,0,1). Completion of the tree structure is left as an assignment for the reader.

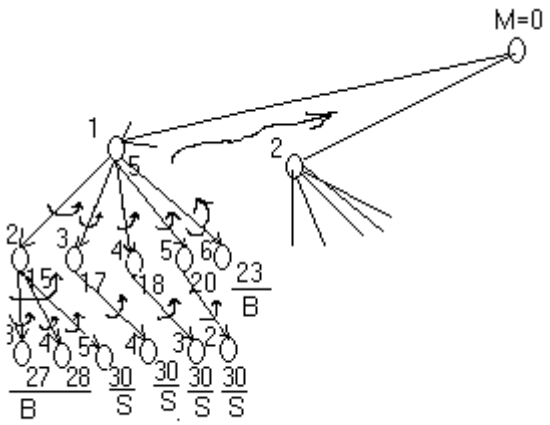


fig 6.2

6.3.3 8 queen problem:

The 8 queen problem can be stated as follows. Consider a chessboard of order 8X8. The problem is to place 8 queens on this board such that no two queens are attack can attack each other.

Illustration.

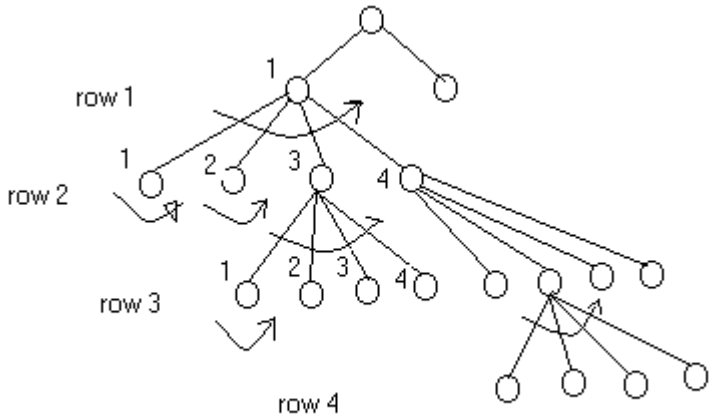


fig 6.3

Consider the problem of 4 queens, backtracking solution for this is as shown in the fig 6.3. The figure shows a partial backtracking tree. Completion of the tree is left as an assignment for the reader.

Concept of Branch and Bound:

The term branch and bound refer to all state space search methods in which all possible branches are derived before any other node can become the E-node. In other words the exploration of a new node cannot begin until the current node is completely explored.

6.2.1 Tape filling:

The branch and bound tree for the records of length (5,10,3) is as shown in fig 6.4

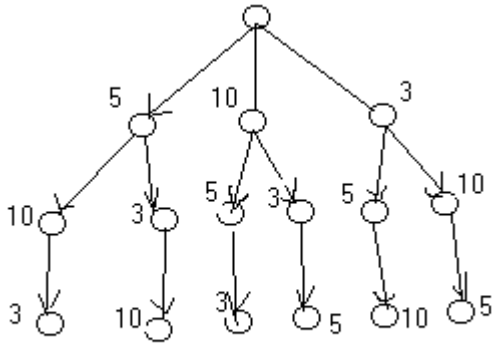


fig 6.4

Single Source Shortest Path:

Graphs can be used to represent the highway structure of a state or country with vertices representing cities and edges representing sections of highway. The edges can then be assigned weights which may be either the distance between the two cities connected by the edge or the average time to drive along that section of highway. A motorist wishing to drive from city A to B would be interested in answers to the following questions:

1. Is there a path from A to B?
2. If there is more than one path from A to B? Which is the shortest path?

The problems defined by these questions are

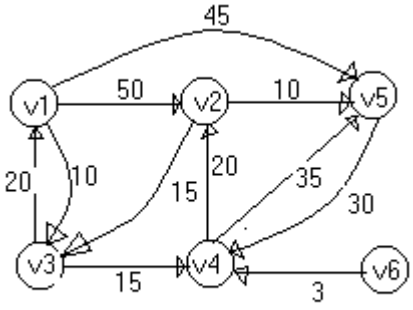


Fig 7.1

special case of the path problem we study in this section. The length of a path is now defined to be the sum of the weights of the edges on that path. The starting vertex of the path is referred to as the source and the last vertex the destination. The graphs are digraphs representing streets. Consider a digraph G=(V,E), with the distance to be traveled as weights on the edges. The problem is to determine the shortest path from v0 to all the remaining vertices of G. It is assumed that all the weights associated with the edges are positive. The shortest path between v0 and some other node v is an ordering among a subset of the edges. Hence this problem fits the ordering paradigm.

Example:

Consider the digraph of fig 7-1. Let the numbers on the edges be the costs of travelling along that route. If a person is interested travel from v1 to v2, then he encounters many paths. Some of them are

1. v1→ v2 = 50 units
2. v1→ v3→ v4→ v2 = 10+15+20=45 units
3. v1→ v5→ v4→ v2 = 45+30+20= 95 units
4. v1→ v3→ v4→ v5→ v4→ v2 = 10+15+35+30+20=110 units

The cheapest path among these is the path along v1→ v3→ v4→ v2. The cost of the path is 10+15+20 = 45 units. Even though there are three edges on this path, it is cheaper than travelling along the path connecting v1 and v2 directly i.e., the path v1→ v2 that costs 50 units. One can also notice that, it is not possible to travel to v6 from any other node.
To formulate a greedy based algorithm to generate the cheapest paths, we must conceive a multistage solution to the problem and also of an optimization measure. One possibility is to build the shortest paths one by one. As an optimization measure we can use the sum of the lengths of all paths so far generated. For this measure to be minimized, each individual path must be of minimum length. If we have already constructed i shortest paths, then using this optimization measure, the next path to be constructed should be the next shortest minimum length path. The greedy way to generate these paths in non-decreasing order of path length. First, a shortest path to the nearest vertex is generated. Then a shortest path to the second nearest vertex is generated, and so on.

A much simpler method would be to solve it using matrix representation. The steps that should be followed is as follows,

Step 1: find the adjacency matrix for the given graph. The adjacency matrix for fig 7.1 is given below

	V1	V2	V3	V4	V5	V6
V1	-	50	10	Inf	45	Inf
V2	Inf	-	15	Inf	10	Inf
V3	20	Inf	-	15	inf	Inf
V4	Inf	20	Inf	-	35	Inf
V5	Inf	Inf	Inf	30	-	Inf
V6	Inf	Inf	Inf	3	Inf	-

Step 2: consider v1 to be the source and choose the minimum entry in the row v1. In the above table the minimum in row v1 is 10.
Step 3: find out the column in which the minimum is present, for the above example it is column v3. Hence, this is the node that has to be next visited.
Step 4: compute a matrix by eliminating v1 and v3 columns. Initially retain only row v1. The second row is computed by adding 10 to all values of row v3.
The resulting matrix is

V2	V4	V5	V6

V1→ Vw	50	Inf	45	Inf
V1→ V3→ Vw	10+inf	10+15	10+inf	10+inf
Minimum	50	25	45	inf

Step 5: find the minimum in each column. Now select the minimum from the resulting row. In the above example the minimum is 25. Repeat step 3 followed by step 4 till all vertices are covered or single column is left.
The solution for the fig 7.1 can be continued as follows

	V2	V5	V6
V1→ Vw	50	45	Inf
V1→ V3→ V4→ Vw	25+20	25+35	25+inf
Minimum	45	45	inf

	V5	V6
V1→ Vw	45	Inf
V1→ V3→ V4→ V2→ Vw	45+10	45+inf
Minimum	45	inf

	V6
V1→ Vw	Inf
V1→ V3→ V4→ V2→ V5→ Vw	45+inf
Minimum	inf

Finally the cheapest path from v1 to all other vertices is given by V1→ V3→ V4→ V2→ V5.

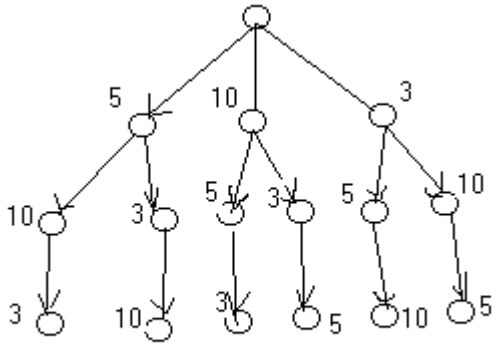


fig 6.4

Minimum Cost Spanning Tree:

Let G=(V,E) be an undirected connected graph. A sub-graph t = (V,E^1) of G is a spanning tree of G if and only if t is a tree.

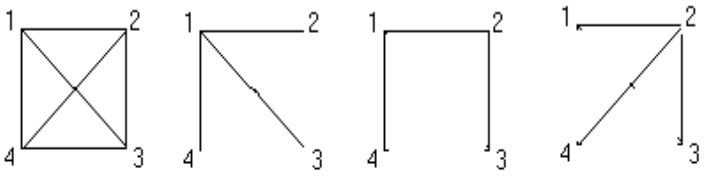


Fig 7.2

Above figure shows the complete graph on four nodes together with three of its spanning tree.

Spanning trees have many applications. For example, they can be used to obtain an independent set of circuit equations for an electric network. First, a spanning tree for the electric network is obtained. Let B be the set of network edges not in the spanning tree. Adding an edge from B to the spanning tree creates a cycle. Kirchoff's second law is used on each cycle to obtain a circuit equation.

Another application of spanning trees arises from the property that a spanning tree is a minimal sub-graph G' of G such that V(G') = V(G) and G' is connected. A minimal sub-graph with n vertices must have at least n-1 edges and all connected graphs with n-1 edges are trees. If the nodes of G represent cities and the edges represent possible communication links connecting two cities, then the minimum number of links needed to connect the n cities is n-1. the spanning trees of G represent all feasible choice.

In practical situations, the edges have weights assigned to them. Thse weights may represent the cost of construction, the length of the link, and so on. Given such a weighted graph, one would then wish to select cities to have minimum total cost or minimum total length. In either case the links selected have to form a tree. If this is not so, then the selection of links contains a cycle. Removal of any one of the links on this cycle results in a link selection of less const connecting all cities. We are therefore interested in finding a spanning tree of G. with minimum cost since the identification of a minimum-cost spanning tree involves the selection of a subset of the edges, this problem fits the subset paradigm.

7.2.1 Prim's Algorithm

A greedy method to obtain a minimum-cost spanning tree builds this tree edge by edge. The next edge to include is chosen according to some optimization criterion. The simplest such criterion is to choose an edge that results in a minimum increase in the sum of the costs of the edges so far included. There are two possible ways to interpret this criterion. In the first, the set of edges so far selected form a tree. Thus, if A is the set of edges selected so far, then A forms a tree. The next edge(u,v) to be included in A is a minimum-cost edge not in A with the property that A U {(u,v)} is also a tree. The corresponding algorithm is known as prim's algorithm.

For Prim's algorithm draw n isolated vertices and label them v1, v2, v3,…vn. Tabulate the given weights of the edges of g in an n by n table. Set the non existent edges as very large. Start from vertex v1 and connect it to its nearest neighbor (i.e., to the vertex, which has the smallest entry in row1 of table) say Vk. Now consider v1 and vk as one subgraph and connect this subgraph to its closest neighbor. Let this new vertex be vi. Next regard the tree with v1 vk and vi as one subgraph and continue the process until all n vertices have been connected by n-1 edges.

Consider the graph shown in fig 7.3. There are 6 vertices and 12 edges. The weights are tabulated in table given below.

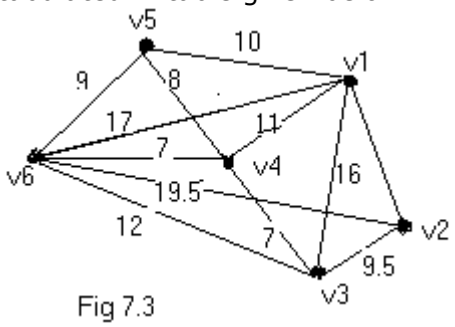


Fig 7.3

	V1	V2	V3	V4	V5	V6
V1	-	10	16	11	10	17
V2	10	-	9.5	Inf	Inf	19.5
V3	16	9.5	-	7	Inf	12
V4	11	Inf	7	-	8	7
V5	10	Inf	Inf	8	-	9
V6	17	19.5	12	7	9	-

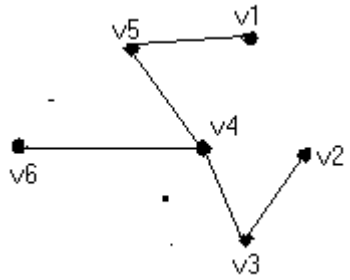


Fig 7.4

Start with v1 and pick the smallest entry in row1, which is either (v1,v2) or (v1,v5). Let us pick (v1, v5). The closest neighbor of the subgraph (v1,v5) is v4 as it is the smallest in the rows v1 and v5. The three remaining edges selected following the above procedure turn out to be (v4,v6) (v4,v3) and (v3, v2) in that sequence. The resulting shortest spanning tree is shown in fig 7.4. The weight of this tree is 41.5.

7.2.3 Kruskal's Algorithm:

There is a second possible interpretation of the optimization criteria mentioned earlier in which the edges of the graph are considered in non-decreasing order of cost. This interpretation is that the set t of edges so far selected for the spanning tree be such that it is possible to complete t into a tree. Thus t may not be a tree at all stages in the algorithm. In fact, it will generally only be a forest since the set of edges t can be completed into a tree if and only if there are no cycles in t. this method is due to kruskal.

The Kruskal algorithm can be illustrated as folows, list out all edges of graph G in order of non-decreasing weight. Next select a smallest edge that makes no circuit with previously

selected edges. Continue this process until (n-1) edges have been selected and these edges will constitute the desired shortest spanning tree.
For fig 7.3 kruskal solution is as follows,
V1 to v2 =10
V1 to v3 = 16
V1 to v4 = 11
V1 to v5 = 10
V1 to v6 = 17
V2 to v3 = 9.5
V2 to v6 = 19.5
V3 to v4 = 7
V3 to v6 =12
V4 to v5 = 8
V4 to v6 = 7
V5 to v6 = 9
The above path in ascending order is
V3 to v4 = 7
V4 to v6 = 7
V4 to v5 = 8
V5 to v6 = 9
V2 to v3 = 9.5
V1 to v5 = 10
V1 to v2 =10
V1 to v4 = 11
V3 to v6 =12
V1 to v3 = 16
V1 to v6 = 17
V2 to v6 = 19.5
Select the minimum, i.e., v3 to v4 connect them, now select v4 to v6 and then v4 to v5, now if we select v5 to v6 then it forms a circuit so drop it and go for the next. Connect v2 and v3 and finally connect v1 and v5. Thus, we have a minimum spanning tree, which is similar to the figure 7.4.

Technique for Graphs:

A fundamental problem concerning graphs is the reachability problem. In its simplest form it requires us to determine whether there exists a path in the given graph G=(V,E) such that this path starts at vertex v and ends at vertex u. A more general form is to determine for a given starting
Vertex v belonging to V all vertices u such that there is a path from v to u. This latter problem can be solved by starting at vertex v and systematically searching the graph G for vertices that can be reached from v. The 2 search methods for this are :

1. Breadth first search.
2. Depth first search.

7.3.1 Breadth first search:

In Breadth first search we start at vertex v and mark it as having been reached. The vertex v at this time is said to be unexplored. A vertex is said to have been explored by an algorithm when the algorithm has visited all vertices adjacent from it. All unvisited vertices adjacent from v are visited next. There are new unexplored vertices. Vertex v has now been explored. The newly visited vertices have not been explored and are put onto the end of the list of unexplored vertices. The first vertex on this list is the next to be explored. Exploration continues until no unexplored vertex is left. The list of unexplored vertices acts as a queue and can be represented using any of the standard queue representations.

7.3.2 Depth first search:

A depth first search of a graph differs from a breadth first search in that the exploration of a vertex v is suspended as soon as a new vertex is reached. At this time the exploration of the new vertex u begins. When this new vertex has been explored, the exploration of u continues. The search terminates when all reached vertices have been fully explored. This search process is best-described recursively.

```
Algorithm DFS(v)
{
visited[v]=1
for each vertex w adjacent from v do
{
If (visited[w]=0)then
DFS(w);
}
}
```

Operating System

Overview and History:

What is an operating system?

Hard to define precisely, because operating systems arose historically as people needed to solve problems associated with using computers.

☐ Much of operating system history driven by relative cost factors of hardware and people. Hardware started out fantastically expensive relative to people and the relative cost has been decreasing ever since. Relative costs drive the goals of the operating system.

- In the beginning: **Expensive Hardware, Cheap People** Goal: maximize hardware utilization.
- Now: **Cheap Hardware, Expensive People**Goal: make it easy for people to use computer.

☐ In the early days of computer use, computers were huge machines that are expensive to buy, run and maintain. Computer used in single user, interactive mode. Programmers interact with the machine at a very low level - flick console switches, dump cards into card reader, etc. The interface is basically the raw hardware.

- Problem: Code to manipulate external I/O devices. Is very complex, and is a major source of programming difficulty.
- Solution: Build a subroutine library (device drivers) to manage the interaction with the I/O devices. The library is loaded into the top of memory and stays there. This is the first example of something that would grow into an operating system.

☐ Because the machine is so expensive, it is important to keep it busy.

- Problem: computer idles while programmer sets things up. Poor utilization of huge investment.
- Solution: Hire a specialized person to do setup. Faster than programmer, but still a lot slower than the machine.
- Solution: Build a batch monitor. Store jobs on a disk (spooling), have computer read them in one at a time and execute them. Big change in computer usage: debugging now done offline from print outs and memory dumps. No more instant feedback.
- Problem: At any given time, job is actively using either the CPU or an I/O device, and the rest of the machine is idle and therefore unutilized.
- Solution: Allow the job to overlap computation and I/O. Buffering and interrupt handling added to subroutine library.
- Problem: one job can't keep both CPU and I/O devices busy. (Have compute-bound jobs that tend to use only the CPU and I/O-bound jobs that tend to use only the I/O devices.) Get poor utilization either of CPU or I/O devices.
- Solution: multiprogramming - several jobs share system. Dynamically switch from one job to another when the running job does I/O. Big issue: protection. Don't want one job to affect the results of another. Memory protection and relocation added to hardware, OS must manage new hardware functionality. OS starts to become a significant software system. OS also starts to take up significant resources on its own.

☐ Phase shift: Computers become much cheaper. People costs become significant.

- Issue: It becomes important to make computers easier to use and to improve the productivity of the people. One big productivity sink: having to wait for batch output (but is this really true?). So, it is important to run interactively. But computers are still so expensive that you can't buy one for every person. Solution: interactive timesharing.
- Problem: Old batch schedulers were designed to run a job for as long as it was utilizing the CPU effectively (in practice, until it tried to do some I/O). But now, people need reasonable response time from the computer.
- Solution: Preemptive scheduling.
- Problem: People need to have their data and programs around while they use the computer.
- Solution: Add file systems for quick access to data. Computer becomes a repository for data, and people don't have to use card decks or tapes to store their data.
- Problem: The boss logs in and gets terrible response time because the machine is overloaded.

- Solution: Prioritized scheduling. The boss gets more of the machine than the peons. But, CPU scheduling is just an example of resource allocation problems. The timeshared machine was full of limited resources (CPU time, disk space, physical memory space, etc.) and it became the responsibility of the OS to mediate the allocation of the resources. So, developed things like disk and physical memory quotas, etc.

☐ Overall, time sharing was a success. However, it was a limited success. In practical terms, every timeshared computer became overloaded and the response time dropped to annoying or unacceptable levels. Hard-core hackers compensated by working at night, and we developed a generation of pasty-looking, unhealthy insomniacs addicted to caffeine.
☐ Computers become even cheaper. It becomes practical to give one computer to each user. Initial cost is very important in market. Minimal hardware (no networking or hard disk, very slow microprocessors and almost no memory) shipped with minimal OS (MS-DOS). Protection, security less of an issue. OS resource consumption becomes a big issue (computer only has 640K of memory). OS back to a shared subroutine library.
☐ Hardware becomes cheaper and users more sophisticated. People need to share data and information with other people. Computers become more information transfer, manipulation and storage devices rather than machines that perform arithmetic operations. Networking becomes very important, and as sharing becomes an important part of the experience so does security. Operating systems become more sophisticated. Start putting back features present in the old time sharing systems (OS/2, Windows NT, even Unix).
☐ Rise of network. Internet is a huge popular phenomenon and drives new ways of thinking about computing. Operating system is no longer interface to the lower level machine - people structure systems to contain layers of middleware. So, a Java API or something similar may be the primary thing people need, not a set of system calls. In fact, what the operating system is may become irrelevant as long as it supports the right set of middleware.
☐ Network computer. Concept of a box that gets all of its resources over the network. No local file system, just network interfaces to acquire all outside data. So have a slimmer version of OS.
☐ In the future, computers will become physically small and portable. Operating systems will have to deal with issues like disconnected operation and mobility. People will also start using information with a psuedo-real time component like voice and video. Operating systems will have to adjust to deliver acceptable performance for these new forms of data.
☐ What does a modern operating system do?

- **Provides Abstractions** Hardware has low-level physical resources with complicated, idiosyncratic interfaces. OS provides abstractions that present clean interfaces. Goal: make computer easier to use. Examples: Processes, Unbounded Memory, Files, Synchronization and Communication Mechanisms.
- **Provides Standard Interface** Goal: portability. Unix runs on many very different computer systems. To a first approximation can port programs across systems with little effort.
- **Mediates Resource Usage** Goal: allow multiple users to share resources fairly, efficiently, safely and securely. Examples:
  - Multiple processes share one processor. (preemptable resource)
  - Multiple programs share one physical memory (preemptable resource).
  - Multiple users and files share one disk. (non-preemptable resource)
  - Multiple programs share a given amount of disk and network bandwidth (preemptable resource).
- **Consumes Resources** Solaris takes up about 8Mbytes physical memory (or about $400).

☐ Abstractions often work well - for example, timesharing, virtual memory and hierarchical and networked file systems. But, may break down if stressed. Timesharing gives poor performance if too many users run compute-intensive jobs. Virtual memory breaks down if working set is too large (thrashing), or if there are too many large processes (machine runs out of swap space). Abstractions often fail for performance reasons.
☐ Abstractions also fail because they prevent programmer from controlling machine at desired level. Example: database systems often want to control movement of information between disk and physical memory, and the paging system can get in the way. More recently, existing OS schedulers fail to adequately support multimedia and parallel processing needs, causing poor performance.

☐ Concurrency and asynchrony make operating systems very complicated pieces of software. Operating systems are fundamentally non-deterministic and event driven. Can be difficult to construct (hundreds of person-years of effort) and impossible to completely debug. Examples of concurrency and asynchrony:

- I/O devices run concurrently with CPU, interrupting CPU when done.
- On a multiprocessor multiple user processes execute in parallel.
- Multiple workstations execute concurrently and communicate by sending messages over a network. Protocol processing takes place asynchronously.

Operating systems are so large no one person understands whole system. Outlives any of its original builders.

- ☐ The major problem facing computer science today is how to build large, reliable software systems. Operating systems are one of very few examples of existing large software systems, and by studying operating systems we may learn lessons applicable to the construction of larger systems.

Overview of Process:

☐ A process is an execution stream in the context of a particular process state.

- An execution stream is a sequence of instructions.
- Process state determines the effect of the instructions. It usually includes (but is not restricted to):
- Registers
- Stack
- Memory (global variables and dynamically allocated memory)
- Open file tables
- Signal management information

Key concept: processes are separated: no process can directly affect the state of another process.
☐ Process is a key OS abstraction that users see - the environment you interact with when you use a computer is built up out of processes.

- The shell you type stuff into is a process.
- When you execute a program you have just compiled, the OS generates a process to run the program.
- Your WWW browser is a process.

☐ Organizing system activities around processes has proved to be a useful way of separating out different activities into coherent units.
☐ Two concepts: uniprogramming and multiprogramming.

- Uniprogramming: only one process at a time. Typical example: DOS. Problem: users often wish to perform more than one activity at a time (load a remote file while editing a program, for example), and uniprogramming does not allow this. So DOS and other uniprogrammed systems put in things like memory-resident programs that invoked asynchronously, but still have separation problems. One key problem with DOS is that there is no memory protection - one program may write the memory of another program, causing weird bugs.
- Multiprogramming: multiple processes at a time. Typical of Unix plus all currently envisioned new operating systems. Allows system to separate out activities cleanly.

☐ Multiprogramming introduces the resource sharing problem - which processes get to use the physical resources of the machine when? One crucial resource: CPU. Standard solution is to use preemptive multitasking - OS runs one process for a while, then takes the CPU away from that process and lets another process run. Must save and restore process state. Key issue: fairness. Must ensure that all processes get their fair share of the CPU.

☐ How does the OS implement the process abstraction? Uses a context switch to switch from running one process to running another process.
☐ How does machine implement context switch? A processor has a limited amount of physical resources. For example, it has only one register set. But every process on the machine has its own set of registers. Solution: save and restore hardware state on a context switch. Save the state in Process Control Block (PCB). What is in PCB? Depends on the hardware.

- Registers - almost all machines save registers in PCB.
- Processor Status Word.
- What about memory? Most machines allow memory from multiple processes to coexist in the physical memory of the machine. Some may require Memory Management Unit (MMU) changes on a context switch. But, some early personal computers switched all of process's memory out to disk (!!!).

☐ Operating Systems are fundamentally event-driven systems - they wait for an event to happen, respond appropriately to the event, then wait for the next event. Examples:

- User hits a key. The keystroke is echoed on the screen.
- A user program issues a system call to read a file. The operating system figures out which disk blocks to bring in, and generates a request to the disk controller to read the disk blocks into memory.
- The disk controller finishes reading in the disk block and generates and interrupt. The OS moves the read data into the user program and restarts the user program.
- A Mosaic or Netscape user asks for a URL to be retrieved. This eventually generates requests to the OS to send request packets out over the network to a remote WWW server. The OS sends the packets.
- The response packets come back from the WWW server, interrupting the processor. The OS figures out which process should get the packets, then routes the packets to that process.
- Time-slice timer goes off. The OS must save the state of the current process, choose another process to run, the give the CPU to that process.

☐ When build an event-driven system with several distinct serial activities, threads are a key structuring mechanism of the OS.
☐ A thread is again an execution stream in the context of a thread state. Key difference between processes and threads is that multiple threads share parts of their state. Typically, allow multiple threads to read and write same memory. (Recall that no processes could directly access memory of another process). But, each thread still has its own registers. Also has its own stack, but other threads can read and write the stack memory.
☐ What is in a thread control block? Typically just registers. Don't need to do anything to the MMU when switch threads, because all threads can access same memory.
☐ Typically, an OS will have a separate thread for each distinct activity. In particular, the OS will have a separate thread for each process, and that thread will perform OS activities on behalf of the process. In this case we say that each user process is backed by a kernel thread.

- When process issues a system call to read a file, the process's thread will take over, figure out which disk accesses to generate, and issue the low level instructions required to start the transfer. It then suspends until the disk finishes reading in the data.
- When process starts up a remote TCP connection, its thread handles the low-level details of sending out network packets.

☐ Having a separate thread for each activity allows the programmer to program the actions associated with that activity as a single serial stream of actions and events. Programmer does not have to deal with the complexity of interleaving multiple activities on the same thread.
☐ Why allow threads to access same memory? Because inside OS, threads must coordinate their activities very closely.

- If two processes issue read file system calls at close to the same time, must make sure that the OS serializes the disk requests appropriately.

- When one process allocates memory, its thread must find some free memory and give it to the process. Must ensure that multiple threads allocate disjoint pieces of memory.

Having threads share the same address space makes it much easier to coordinate activities - can build data structures that represent system state and have threads read and write data structures to figure out what to do when they need to process a request.

☐ One complication that threads must deal with: asynchrony. Asynchronous events happen arbitrarily as the thread is executing, and may interfere with the thread's activities unless the programmer does something to limit the asynchrony. Examples:

- An interrupt occurs, transferring control away from one thread to an interrupt handler.
- A time-slice switch occurs, transferring control from one thread to another.
- Two threads running on different processors read and write the same memory.

☐ Asynchronous events, if not properly controlled, can lead to incorrect behavior. Examples:

- Two threads need to issue disk requests. First thread starts to program disk controller (assume it is memory-mapped, and must issue multiple writes to specify a disk operation). In the meantime, the second thread runs on a different processor and also issues the memory-mapped writes to program the disk controller. The disk controller gets horribly confused and reads the wrong disk block.
- Two threads need to write to the display. The first thread starts to build its request, but before it finishes a time-slice switch occurs and the second thread starts its request. The combination of the two threads issues a forbidden request sequence, and smoke starts pouring out of the display.
- For accounting reasons the operating system keeps track of how much time is spent in each user program. It also keeps a running sum of the total amount of time spent in all user programs. Two threads increment their local counters for their processes, then concurrently increment the global counter. Their increments interfere, and the recorded total time spent in all user processes is less than the sum of the local times.

☐ So, programmers need to coordinate the activities of the multiple threads so that these bad things don't happen. Key mechanism: synchronization operations. These operations allow threads to control the timing of their events relative to events in other threads. Appropriate use allows programmers to avoid problems like the ones outlined above.

Overview Of Thread:

☐ We first must postulate a thread creation and manipulation interface. Will use the one in Nachos:

```
class Thread {
public:
Thread(char* debugName);
~Thread();
void Fork(void (*func)(int), int arg);
void Yield();
void Finish();
}
```

☐ The `Thread` constructor creates a new thread. It allocates a data structure with space for the TCB.

☐ To actually start the thread running, must tell it what function to start running when it runs. The `Fork` method gives it the function and a parameter to the function.

☐ What does `Fork` do? It first allocates a stack for the thread. It then sets up the TCB so that when the thread starts running, it will invoke the function and pass it the correct parameter. It then puts the thread on a run queue someplace. `Fork` then returns, and the thread that called `Fork` continues.

☐ How does OS set up TCB so that the thread starts running at the function? First, it sets the stack pointer in the TCB to the stack. Then, it sets the PC in the TCB to be the first instruction in the function. Then, it sets the register in the TCB holding the first parameter to the parameter. When the thread system restores the state from the TCB, the function will magically start to run.

☐ The system maintains a queue of runnable threads. Whenever a processor becomes idle, the thread scheduler grabs a thread off of the run queue and runs the thread.

☐ Conceptually, threads execute concurrently. This is the best way to reason about the behavior of threads. But in practice, the OS only has a finite number of processors, and it can't run all of the runnable threads at once. So, must multiplex the runnable threads on the finite number of processors.
☐ Let's do a few thread examples. First example: two threads that increment a variable. int a = 0;

```
void sum(int p) {
a++;
printf("%d : a = %d\n", p, a);
} void main() {
Thread *t = new Thread("child");
t->Fork(sum, 1);
sum(0);
}
```

☐ The two calls to `sum` run concurrently. What are the possible results of the program? To understand this fully, we must break the `sum` subroutine up into its primitive components.
☐ `sum` first reads the value of `a` into a register. It then increments the register, then stores the contents of the register back into `a`. It then reads the values of of the control string, `p` and `a` into the registers that it uses to pass arguments to the `printf` routine. It then calls`printf`, which prints out the data.
☐ The best way to understand the instruction sequence is to look at the generated assembly language (cleaned up just a bit). You can have the compiler generate assembly code instead of object code by giving it the -S flag. It will put the generated assembly in the same file name as the .c or .cc file, but with a .s suffix.

```
la a, %r0
ld [%r0],%r1
add %r1,1,%r1
st %r1,[%r0]
ld [%r0], %o3
!parameters are passed starting with %o0
mov %o0, %o1
la .L17, %o0
call printf
```

☐ So when execute concurrently, the result depends on how the instructions interleave. What are possible results?

```
0 : 1                   0 : 1
1 : 2                   1 : 1

1 : 2                   1 : 1
0 : 1                   0 : 1

1 : 1                   0 : 2
0 : 2                   1 : 2

0 : 2                   1 : 2
1 : 1                   0 : 2
```

So the results are nondeterministic - you may get different results when you run the program more than once. So, it can be very difficult to reproduce bugs. Nondeterministic execution is one of the things that makes writing parallel programs much more difficult than writing serial programs.
☐ Chances are, the programmer is not happy with all of the possible results listed above. Probably wanted the value of `a`to be 2 after both threads finish. To achieve this, must make the increment operation atomic. That is, must prevent the interleaving of the instructions in a way that would interfere with the additions.
☐ Concept of atomic operation. An atomic operation is one that executes without any interference from other operations - in other words, it executes as one unit. Typically build complex atomic operations up out of sequences of primitive operations. In our case the primitive operations are the individual machine instructions.
☐ More formally, if several atomic operations execute, the final result is guaranteed to be the same as if the operations executed in some serial order.
☐ In our case above, build an increment operation up out of loads, stores and add machine instructions. Want the increment operation to be atomic.

☐ Use synchronization operations to make code sequences atomic. First synchronization abstraction: semaphores. A semaphore is, conceptually, a counter that supports two atomic operations, P and V. Here is the Semaphore interface from Nachos:

```
class Semaphore {
 public:
Semaphore(char* debugName, int initialValue);
~Semaphore();
void P();
void V();
}
```

☐ Here is what the operations do:

- Semphore(name, count) : creates a semaphore and initializes the counter to count.
- P() : Atomically waits until the counter is greater than 0, then decrements the counter and returns.
- V() : Atomically increments the counter.

☐ Here is how we can use the semaphore to make the `sum`example work:

```
int a = 0;
Semaphore *s;
void sum(int p) {
int t;
s->P();
a++;
t = a;
s->V();
printf("%d : a = %d\n", p, t);
}
void main() {
Thread *t = new Thread("child");
s = new Semaphore("s", 1);
t->Fork(sum, 1);
sum(0);
}
```

☐ We are using semaphores here to implement a mutual exclusion mechanism. The idea behind mutual exclusion is that only one thread at a time should be allowed to do something. In this case, only one thread should access `a`. Use mutual exclusion to make operations atomic. The code that performs the atomic operation is called a critical section.

☐ Semaphores do much more than mutual exclusion. They can also be used to synchronize producer/consumer programs. The idea is that the producer is generating data and the consumer is consuming data. So a Unix pipe has a producer and a consumer. You can also think of a person typing at a keyboard as a producer and the shell program reading the characters as a consumer.

☐ Here is the synchronization problem: make sure that the consumer does not get ahead of the producer. But, we would like the producer to be able to produce without waiting for the consumer to consume. Can use semaphores to do this. Here is how it works:

```
Semaphore *s;
void consumer(int dummy) {
while (1) {
  s->P();
  consume the next unit of data
}
}
void producer(int dummy) {
while (1) {
  produce the next unit of data
  s->V();
```

```
}
}
void main() {
s = new Semaphore("s", 0);
Thread *t = new Thread("consumer");
t->Fork(consumer, 1);
t = new Thread("producer");
t->Fork(producer, 1);
}
```

In some sense the semaphore is an abstraction of the collection of data.

☐ In the real world, pragmatics intrude. If we let the producer run forever and never run the consumer, we have to store all of the produced data somewhere. But no machine has an infinite amount of storage. So, we want to let the producer to get ahead of the consumer if it can, but only a given amount ahead. We need to implement a bounded buffer which can hold only N items. If the bounded buffer is full, the producer must wait before it can put any more data in.

```
Semaphore *full;
Semaphore *empty;
void consumer(int dummy) {
while (1) {
  full->P();
  consume the next unit of data
  empty->V();
}
}
void producer(int dummy) {
while (1) {
  empty->P();
  produce the next unit of data
  full->V();
}
}
void main() {
empty = new Semaphore("empty", N);
full = new Semaphore("full", 0);
Thread *t = new Thread("consumer");
t->Fork(consumer, 1);
t = new Thread("producer");
t->Fork(producer, 1);
}
```

An example of where you might use a producer and consumer in an operating system is the console (a device that reads and writes characters from and to the system console). You would probably use semaphores to make sure you don't try to read a character before it is typed.

☐ Semaphores are one synchronization abstraction. There is another called locks and condition variables.

☐ Locks are an abstraction specifically for mutual exclusion only. Here is the Nachos lock interface:

```
class Lock {
public:
Lock(char* debugName);
// initialize lock to be FREE
~Lock();
// deallocate lock
void Acquire();
// these are the only operations on a lock
void Release();
// they are both *atomic*
}
```

☐

A lock can be in one of two states: locked and unlocked. Semantics of lock operations:

- Lock(name) : creates a lock that starts out in the unlocked state.
- Acquire() : Atomically waits until the lock state is unlocked, then sets the lock state to locked.
- Release() : Atomically changes the lock state to unlocked from locked.

In assignment 1 you will implement locks in Nachos on top of semaphores.

☐ What are requirements for a locking implementation?

- Only one thread can acquire lock at a time. (safety)
- If multiple threads try to acquire an unlocked lock, one of the threads will get it. (liveness)
- All unlocks complete in finite time. (liveness)

☐ What are desirable properties for a locking implementation?

- Efficiency: take up as little resources as possible.
- Fairness: threads acquire lock in the order they ask for it. Are also weaker forms of fairness.
- Simple to use.

☐ When use locks, typically associate a lock with pieces of data that multiple threads access. When one thread wants to access a piece of data, it first acquires the lock. It then performs the access, then unlocks the lock. So, the lock allows threads to perform complicated atomic operations on each piece of data.

☐ Can you implement unbounded buffer only using locks? There is a problem - if the consumer wants to consume a piece of data before the producer produces the data, it must wait. But locks do not allow the consumer to wait until the producer produces the data. So, consumer must loop until the data is ready. This is bad because it wastes CPU resources.

☐ There is another synchronization abstraction called condition variables just for this kind of situation. Here is the Nachos interface:

```
class Condition {
public:
  Condition(char* debugName);
  ~Condition();
  void Wait(Lock *conditionLock);
  void Signal(Lock *conditionLock);
  void Broadcast(Lock *conditionLock);
}
```

☐ Semantics of condition variable operations:

- Condition(name) : creates a condition variable.
- Wait(Lock *l) : Atomically releases the lock and waits. When Wait returns the lock will have been reacquired.
- Signal(Lock *l) : Enables one of the waiting threads to run. When Signal returns the lock is still acquired.
- Broadcast(Lock *l) : Enables all of the waiting threads to run. When Broadcast returns the lock is still acquired.

All locks must be the same. In assignment 1 you will implement condition variables in Nachos on top of semaphores.

☐ Typically, you associate a lock and a condition variable with a data structure. Before the program performs an operation on the data structure, it acquires the lock. If it has to wait before it can perform the operation, it uses the condition variable to wait for another operation to bring the data structure into a state where it can perform the operation. In some cases you need more than one condition variable.

☐ Let's say that we want to implement an unbounded buffer using locks and condition variables. In this case we have 2 consumers.

```
Lock *l;
Condition *c;
int avail = 0;
void consumer(int dummy) {
while (1) {
  l->Acquire();
```

```
  if (avail == 0) {
        c->Wait(l);
  }
  consume the next unit of data
  avail--;
  l->Release();
}
}
void producer(int dummy) {
while (1) {
  l->Acquire();
  produce the next unit of data
  avail++;
  c->Signal(l);
  l->Release();
}
}
void main() {
l = new Lock("l");
c = new Condition("c");
Thread *t = new Thread("consumer");
t->Fork(consumer, 1);
Thread *t = new Thread("consumer");
t->Fork(consumer, 2);
t = new Thread("producer");
t->Fork(producer, 1);
}
```

☐ There are two variants of condition variables: Hoare condition variables and Mesa condition variables. For Hoare condition variables, when one thread performs a `Signal`, the very next thread to run is the waiting thread.

For Mesa condition variables, there are no guarantees when the signalled thread will run. Other threads that acquire the lock can execute between the signaller and the waiter. The example above will work with Hoare condition variables but not with Mesa condition variables.

☐ What is the problem with Mesa condition variables? Consider the following scenario: Three threads, thread 1 one producing data, threads 2 and 3 consuming data.

- Thread 2 calls `consumer`, and suspends.
- Thread 1 calls `producer`, and signals thread 2.
- Instead of thread 2 running next, thread 3 runs next, calls `consumer`, and consumes the element. (Note: with Hoare monitors, thread 2 would always run next, so this would not happen.)
- Thread 2 runs, and tries to consume an item that is not there. Depending on the data structure used to store produced items, may get some kind of illegal access error.

☐ How can we fix this problem? Replace the `if` with a`while`.

```
void consumer(int dummy) {
while (1) {
  l->Acquire();
  while (avail == 0) {
        c->Wait(l);
  }
  consume the next unit of data
  avail--;
  l->Release();
}
}
```

In general, this is a crucial point. Always put `while`'s around your condition variable code. If you don't, you can get really obscure bugs that show up very infrequently.

☐ In this example, what is the data that the lock and condition variable are associated with?
The `avail` variable.

☐ People have developed a programming abstraction that automatically associates locks and condition variables with data. This abstraction is called a monitor. A monitor is a data structure plus a set of operations (sort of like an abstract data type). The monitor also has a lock and, optionally, one or more condition variables. See notes for Lecture 14.

☐ The compiler for the monitor language automatically inserts a lock operation at the beginning of each routine and an unlock operation at the end of the routine. So, programmer does not have to put in the lock operations.

☐ Monitor languages were popular in the middle 80's - they are in some sense safer because they eliminate one possible programming error. But more recent languages have tended not to support monitors explicitly, and expose the locking operations to the programmer. So the programmer has to insert the lock and unlock operations by hand. Java takes a middle ground - it supports monitors, but also allows programmers to exert finer grain control over the locked sections by supporting synchronized blocks within methods. But synchronized blocks still present a structured model of synchronization, so it is not possible to mismatch the lock acquire and release.

☐ Laundromat Example: A local laudromat has switched to a computerized machine allocation scheme. There are N machines, numbered 1 to N. By the front door there are P allocation stations. When you want to wash your clothes, you go to an allocation station and put in your coins. The allocation station gives you a number, and you use that machine. There are also P deallocation stations. When your clothes finish, you give the number back to one of the deallocation stations, and someone else can use the machine. Here is the alpha release of the machine allocation software:

```
allocate(int dummy) {
while (1) {
wait for coins from user
n = get();
give number n to user
}
}
deallocate(int dummy) {
while (1) {
wait for number n from user
put(i);
}
}
main() {
for (i = 0; i < P; i++) {
t = new Thread("allocate");
t->Fork(allocate, 0);
t = new Thread("deallocate");
t->Fork(deallocate, 0);
}
}
```

☐ The key parts of the scheduling are done in the two routines `get` and `put`, which use an array data structure `a` to keep track of which machines are in use and which are free.

```
int a[N];
int get() {
for (i = 0; i < N; i++) {
if (a[i] == 0) {
a[i] = 1;
return(i+1);
}
}
}
void put(int i) {
a[i-1] = 0;
```

```
}
```

☐ It seems that the alpha software isn't doing all that well. Just looking at the software, you can see that there are several synchronization problems.
☐ The first problem is that sometimes two people are assigned to the same machine. Why does this happen? We can fix this with a lock:

```
int a[N];
Lock *l;
int get() {
l->Acquire();
for (i = 0; i < N; i++) {
  if (a[i] == 0) {
        a[i] = 1;
        l->Release();
        return(i+1);
  }
}
l->Release();
}
void put(int i) {
l->Acquire();
a[i-1] = 0;
l->Release();
}
```

So now, have fixed the multiple assignment problem. But what happens if someone comes in to the laundry when all of the machines are already taken? What does the machine return? Must fix it so that the system waits until there is a machine free before it returns a number. The situation calls for condition variables.

```
int a[N];
Lock *l;
Condition *c;
int get() {
l->Acquire();
while (1) {
for (i = 0; i < N; i++) {
if (a[i] == 0) {
  a[i] = 1;
  l->Release();
  return(i+1);
}
}
c->Wait(l);
}
}
void put(int i) {
l->Acquire();
a[i-1] = 0;
c->Signal();
l->Release();
}
```

☐ What data is the lock protecting? The `a` array.
☐ When would you use a broadcast operation? Whenever want to wake up all waiting threads, not just one. For an event that happens only once. For example, a bunch of threads may wait until a file is deleted. The thread that actually deleted the file could use a broadcast to wake up all of the threads.
☐ Also use a broadcast for allocation/deallocation of variable sized units. Example: concurrent malloc/free.

```
Lock *l;
Condition *c;
char *malloc(int s) {
```

```
l->Acquire();
while (cannot allocate a chunk of size s) {
  c->Wait(l);
}
allocate chunk of size s;
l->Release();
return pointer to allocated chunk
}
void free(char *m) {
l->Acquire();
deallocate m.
c->Broadcast(l);
l->Release();
}
```

☐ Example with malloc/free. Initially start out with 10 bytes free.

Time	Process 1	Process 2	Process 3
	malloc(10) succeeds	malloc(5) suspends lock	malloc(5) suspends lock
1		gets lock - waits	
2			gets lock waits
3	free(10) broadcast		
4		resume malloc(5) succeeds	
5			resume malloc(5) succeeds
6	malloc(7) waits		
7			malloc(3) waits
8		free(5) broadcast	
9	resume malloc(7) waits		
10			resume malloc(3) succeeds

What would happen if changed `c->Broadcast(l)` to `c->Signal(l)`? At step 10, process 3 would not wake up, and it would not get the chance to allocate available memory. What would happen if changed `while` loop to an `if`?

☐ You will be asked to implement condition variables as part of assignment 1. The following implementation is INCORRECT. Please do not turn this implementation in.

```
class Condition {
private:
  int waiting;
  Semaphore *sema;
}
void Condition::Wait(Lock* l)
{
waiting++;
l->Release();
sema->P();
```

```
l->Acquire();
}
void Condition::Signal(Lock* l)
{
if (waiting > 0) {
  sema->V();
  waiting--;
}
}
```

Why is this solution incorrect? Because in some cases the signalling thread may wake up a waiting thread that called Wait after the signalling thread called Signal.

Overview of Dead Lock:

You may need to write code that acquires more than one lock. This opens up the possibility of deadlock. Consider the following piece of code:

Lock *l1, *l2; void p() { l1->Acquire(); l2->Acquire(); code that manipulates data that l1 and l2 protect l2->Release(); l1->Release(); } void q() { l2->Acquire(); l1->Acquire(); code that manipulates data that l1 and l2 protect l1->Release(); l2->Release(); }

If p and q execute concurrently, consider what may happen. First, p acquires l1 and q acquires l2. Then, p waits to acquire l2 and q waits to acquire l1. How long will they wait? Forever. This case is called deadlock. What are conditions for deadlock? Mutual Exclusion: Only one thread can hold lock at a time. Hold and Wait: At least one thread holds a lock and is waiting for another process to release a lock. No preemption: Only the process holding the lock can release it. Circular Wait: There is a set t_1, ..., t_n such that t_1 is waiting for a lock held by t_2, ..., t_n is waiting for a lock held by t_1.

How can p and q avoid deadlock? Order the locks, and always acquire the locks in that order. Eliminates the circular wait condition. Occasionally you may need to write code that needs to acquire locks in different orders. Here is what to do in this situation. First, most locking abstractions offer an operation that tries to acquire the lock but returns if it cannot. We will call this operation `TryAcquire`. Use this operation to try to acquire the lock that you need to acquire out of order. If the operation succeeds, fine. Once you've got the lock, there is no problem. If the operation fails, your code will need to release all locks that come after the lock you are trying to acquire. Make sure the associated data structures are in a state where you can release the locks without crashing the system. Release all of the locks that come after the lock you are trying to acquire, then reacquire all of the locks in the right order. When the code resumes, bear in mind that the data structures might have changed between the time when you released and reacquired the lock.

Here is an example. int d1, d2; // The standard acquisition order for these two locks // is l1, l2. Lock *l1, // protects d1 *l2; // protects d2 // Decrements d2, and if the result is 0, increments d1 void increment() { l2->Acquire(); int t = d2; t--; if (t == 0) { if (l1->TryAcquire()) { d1++; } else { // Any modifications to d2 go here - in this case none l2->Release(); l1->Acquire(); l2->Acquire(); t = d2; t--; // some other thread may have changed d2 - must recheck it if (t == 0) { d1++; } } l1->Release(); } d2 = t; l2->Release(); }

This example is somewhat contrived, but you will recognize the situation when it occurs. There is a generalization of the deadlock problem to situations in which processes need multiple resources, and the hardware may have multiple kinds of each resource - two printers, etc. Seems kind of like a batch model - processes request resources, then system schedules process to run when resources are available. In this model, processes issue requests to OS for resources, and OS decides who gets which resource when. A lot of theory built up to handle this situation. Process first requests a resource, the OS issues it and the process uses the resource, then the process releases the resource back to the OS. Reason about resource allocation using resource allocation graph. Each resource is represented with a box, each process with a circle, and the individual instances of the resources with dots in the boxes. Arrows go from processes to resource boxes if the process is waiting for the resource. Arrows go from dots in resource box to processes if the process holds that instance of the resource. See Fig. 7.1. If graph contains no cycles, is no deadlock. If has a cycle, may or may not have deadlock. See Fig. 7.2, 7.3. System can either Restrict the way in which processes will

request resources to prevent deadlock. Require processes to give advance information about which resources they will require, then use algorithms that schedule the processes in a way that avoids deadlock. Detect and eliminate deadlock when it occurs. First consider prevention. Look at the deadlock conditions listed above. Mutual Exclusion - To eliminate mutual exclusion, allow everybody to use the resource immediately if they want to. Unrealistic in general - do you want your printer output interleaved with someone elses? Hold and Wait. To prevent hold and wait, ensure that when a process requests resources, does not hold any other resources. Either asks for all resources before executes, or dynamically asks for resources in chunks as needed, then releases all resources before asking for more. Two problems - processes may hold but not use resources for a long time because they will eventually hold them. Also, may have starvation. If a process asks for lots of resources, may never run because other processes always hold some subset of the resources. Circular Wait. To prevent circular wait, order resources and require processes to request resources in that order. Deadlock avoidance. Simplest algorithm - each process tells max number of resources it will ever need. As process runs, it requests resources but never exceeds max number of resources. System schedules processes and allocates resoures in a way that ensures that no deadlock results. Example: system has 12 tape drives. System currently running P0 needs max 10 has 5, P1 needs max 4 has 2, P2 needs max 9 has

2. Can system prevent deadlock even if all processes request the max? Well, right now system has 3 free tape drives. If P1 runs first and completes, it will have 5 free tape drives. P0 can run to completion with those 5 free tape drives even if it requests max. Then P2 can complete. So, this schedule will execute without deadlock. If P2 requests two more tape drives, can system give it the drives? No, because cannot be sure it can run all jobs to completion with only 1 free drive. So, system must not give P2 2 more tape drives until P1 finishes. If P2 asks for 2 tape drives, system suspends P2 until P1 finishes. Concept: Safe Sequence. Is an ordering of processes such that all processes can execute to completion in that order even if all request maximum resources. Concept: Safe State - a state in which there exists a safe sequence. Deadlock avoidance algorithms always ensure that system stays in a safe state. How can you figure out if a system is in a safe state? Given the current and maximum allocation, find a safe sequence. System must maintain some information about the resources and

Avail[j] = number of resource j available Max[i,j] = max number of resource j that process i will use Alloc[i,j] = number of resource j that process i currently has Need[i,j] = Max[i,j] - Alloc[i,j] Notation: `A<=B` if for all processes `i`, `A[i]<=B[i]`. Safety Algorithm: will try to find a safe sequence. Simulate evolution of system over time under worst case assumptions of resource demands. 1: Work = Avail; Finish[i] = False for all i; 2: Find i such that Finish[i] = False and Need[i] <= Work If no such i exists, goto 4 3: Work = Work + Alloc[i]; Finish[i] = True; goto 2 4: If Finish[i] = True for all i, system is in a safe state

Now, can use safety algorithm to determine if we can satisfy a given resource demand. When a process demands additional resources, see if can give them to process and remain in a safe state. If not, suspend process until system can allocate resources and remain in a safe state. Need an additional data structure: Request[i,j] = number of j resources that process i requests Here is algorithm. Assume process `i` has just requested additional resources.

1: If Request[i] <= Need[i] goto 2. Otherwise, process has violated its maximum resource claim. 2: If Request[i] <= Avail goto 3. Otherwise, i must wait because resources are not available. 3: Pretend to allocate resources as follows: Avail = Avail - Request[i] Alloc[i] = Alloc[i] + Request[i] Need[i] = Need[i] - Request[i] If this is a safe state, give the process the resources. Otherwise, suspend the process and restore the old state.

When to check if a suspended process should be given the resources and resumed? Obvious choice - when some other process relinquishes its resources. Obvious problem - process starves because other processes with lower resource requirements are always taking freed resources. See Example in Section 7.5.3.3. Third alternative: deadlock detection and elimination. Just let deadlock happen. Detect when it does, and eliminate the deadlock by preempting resources. Here is deadlock detection algorithm. Is very similar to safe state detection algorithm.

1: Work = Avail; Finish[i] = False for all i; 2: Find i such that Finish[i] = False and Request[i] <= Work If no such i exists, goto 4 3: Work = Work + Alloc[i]; Finish[i] = True; goto 2 4: If Finish[i] = False for some i, system is deadlocked.

When to run deadlock detection algorithm? Obvious time: whenever a process requests more resources and suspends. If deadlock detection takes too much time, maybe run it less frequently. OK, now you've found a deadlock. What do you do? Must free up some resources so that some processes can run. So, preempt resources - take them away from processes. Several different preemption cases: Can preempt some resources without killing job - for example, main memory. Can just swap out to disk and resume job later. If

job provides rollback points, can roll job back to point before acquired resources. At a later time, restart job from rollback point. Default rollback point - start of job. For some resources must just kill job. All resources are then free. Can either kill processes one by one until your system is no longer deadlocked. Or, just go ahead and kill all deadlocked processes. In a real system, typically use different deadlock strategies for different situations based on resource characteristics. This whole topic has a sort of 60's and 70's batch mainframe feel to it. How come these topics never seem to arise in modern Unix systems?

Overview of Synchronization:

☐
How do we implement synchronization operations like locks? Can build synchronization operations out of atomic reads and writes. There is a lot of literature on how to do this, one algorithm is called the bakery algorithm. But, this is slow and cumbersome to use. So, most machines have hardware support for synchronization - they provide synchronization instructions.

☐ On a uniprocessor, the only thing that will make multiple instruction sequences not atomic is interrupts. So, if want to do a critical section, turn off interrupts before the critical section and turn on interrupts after the critical section. Guaranteed atomicity. It is also fairly efficient. Early versions of Unix did this.
☐ Why not just use turning off interrupts? Two main disadvantages: can't use in a multiprocessor, and can't use directly from user program for synchronization.
☐ Test-And-Set. The test and set instruction atomically checks if a memory location is zero, and if so, sets the memory location to 1. If the memory location is 1, it does nothing. It returns the old value of the memory location. You can use test and set to implement locks as follows:

- The lock state is implemented by a memory location. The location is 0 if the lock is unlocked and 1 if the lock is locked.
- The lock operation is implemented as:
- ```
  while (test-and-set(l) == 1);
  ```
- The unlock operation is implemented as: *l = 0;

The problem with this implementation is busy-waiting. What if one thread already has the lock, and another thread wants to acquire the lock? The acquiring thread will spin until the thread that already has the lock unlocks it.
☐ What if the threads are running on a uniprocessor? How long will the acquiring thread spin? Until it expires its quantum and thread that will unlock the lock runs. So on a uniprocessor, if can't get the thread the first time, should just suspend. So, lock acquisition looks like this:

```
while (test-and-set(l) == 1) {
  currentThread->Yield();
}
```

Can make it even better by having a queue lock that queues up the waiting threads and gives the lock to the first thread in the queue. So, threads never try to acquire lock more than once.
☐ On a multiprocessor, it is less clear. Process that will unlock the lock may be running on another processor. Maybe should spin just a little while, in hopes that other process will release lock. To evaluate spinning and suspending strategies, need to come up with a cost for each suspension algorithm. The cost is the amount of CPU time the algorithm uses to acquire a lock.
☐ There are three components of the cost: spinning, suspending and resuming. What is the cost of spinning? Waste the CPU for the spin time. What is cost of suspending and resuming? Amount of CPU time it takes to suspend the thread and restart it when the thread acquires the lock.
☐ Each lock acquisition algorithm spins for a while, then suspends if it didn't get the lock. The optimal algorithm is as follows:

- If the lock will be free in less than the suspend and resume time, spin until acquire the lock.
- If the lock will be free in more than the suspend and resume time, suspend immediately.

Obviously, cannot implement this algorithm - it requires knowledge of the future, which we do not in general have.
☐ How do we evaluate practical algorithms - algorithms that spin for a while, then suspend. Well, we compare them with the optimal algorithm in the worst case for the practical algorithm. What is the worst
```

case for any practical algorithm relative to the optimal algorithm? When the lock become free just after the practical algorithm stops spinning.

☐ What is worst-case cost of algorithm that spins for the suspend and resume time, then suspends? (Will call this the SR algorithm). Two times the suspend and resume time. The worst case is when the lock is unlocked just after the thread starts the suspend. The optimal algorithm just spins until the lock is unlocked, taking the suspend and resume time to acquire the lock. The SR algorithm costs twice the suspend and resume time -it first spins for the suspend and resume time, then suspends, then gets the lock, then resumes.

☐ What about other algorithms that spin for a different fixed amount of time then block? Are all worse than the SR algorithm.

- If spin for less than suspend and resume time then suspend (call this the LT-SR algorithm), worst case is when lock becomes free just after start the suspend. In this case the the algorithm will cost spinning time plus suspend and resume time. The SR algorithm will just cost the spinning time.
- If spin for greater than suspend and resume time then suspend (call this the GR-SR algorithm), worst case is again when lock becomes free just after start the suspend. In this case the SR algorithm will also suspend and resume, but it will spin for less time than the GT-SR algorithm

Of course, in practice locks may not exhibit worst case behavior, so best algorithm depends on locking and unlocking patterns actually observed.

☐ Here is the SR algorithm. Again, can be improved with use of queueing locks.

```
notDone = test-and-set(l);
if (!notDone) return;
start = readClock();
while (notDone) {
  stop = readClock();
  if (stop - start >= suspendAndResumeTime) {
    currentThread->Yield();
    start = readClock();
  }
  notDone = test-and-set(l);
}
```

☐ There is an orthogonal issue. test-and-set instruction typically consumes bus resources every time. But a load instruction caches the data. Subsequent loads come out of cache and never hit the bus. So, can do something like this for inital algorithm:

```
while (1) {
  if !test-and-set(l) break;
  while (*l == 1);
}
```

☐ Are other instructions that can be used to implement spin locks - swap instruction, for example.

☐ On modern RISC machines, test-and-set and swap may cause implementation headaches. Would rather do something that fits into load/store nature of architecture. So, have a non-blocking abstraction: Load Linked(LL)/Store Conditional(SC).

☐ Semantics of LL: Load memory location into register and mark it as loaded by this processor. A memory location can be marked as loaded by more than one processor.

☐ Semantics of SC: if the memory location is marked as loaded by this processor, store the new value and remove all marks from the memory location. Otherwise, don't perform the store. Return whether or not the store succeeded.

☐ Here is how to use LL/SC to implement the lock operation:

```
while (1) {
  LL  r1, lock
  if (r1 == 0) {
    LI r2, 1
    if (SC r2, lock) break;
  }
}
```

Unlock operation is the same as before.

☐ Can also use LL/SC to implement some operations (like increment) directly. People have built up a whole bunch of theory dealing with the difference in power between stuff like LL/SC and test-and-set.

```
while (1) {
  LL   r1, lock
  ADDI r1, 1, r1
  if (SC r2, lock) break;
}
```

☐ Note that the increment operation is non-blocking. If two threads start to perform the increment at the same time, neither will block - both will complete the add and only one will successfully perform the SC. The other will retry. So, it eliminates problems with locking like: one thread acquires locks and dies, or one thread acquires locks and is suspended for a long time, preventing other threads that need to acquire the lock from proceeding.

Overview of CPU Scheduling:

☐
What is CPU scheduling? Determining which processes run when there are multiple runnable processes. Why is it important? Because it can can have a big effect on resource utilization and the overall performance of the system.
☐ By the way, the world went through a long period (late 80's, early 90's) in which the most popular operating systems (DOS, Mac) had NO sophisticated CPU scheduling algorithms. They were single threaded and ran one process at a time until the user directs them to run another process. Why was this true? More recent systems (Windows NT) are back to having sophisticated CPU scheduling algorithms. What drove the change, and what will happen in the future?
☐ Basic assumptions behind most scheduling algorithms:

- There is a pool of runnable processes contending for the CPU.
- The processes are independent and compete for resources.
- The job of the scheduler is to distribute the scarce resource of the CPU to the different processes ``fairly'' (according to some definition of fairness) and in a way that optimizes some performance criteria.

In general, these assumptions are starting to break down. First of all, CPUs are not really that scarce - almost everybody has several, and pretty soon people will be able to afford lots. Second, many applications are starting to be structured as multiple cooperating processes. So, a view of the scheduler as mediating between competing entities may be partially obsolete.
☐ How do processes behave? First, CPU/IO burst cycle. A process will run for a while (the CPU burst), perform some IO (the IO burst), then run for a while more (the next CPU burst). How long between IO operations? Depends on the process.

- IO Bound processes: processes that perform lots of IO operations. Each IO operation is followed by a short CPU burst to process the IO, then more IO happens.
- CPU bound processes: processes that perform lots of computation and do little IO. Tend to have a few long CPU bursts.

One of the things a scheduler will typically do is switch the CPU to another process when one process does IO. Why? The IO will take a long time, and don't want to leave the CPU idle while wait for the IO to finish.
☐ When look at CPU burst times across the whole system, have the exponential or hyperexponential distribution in Fig. 5.2.
☐ What are possible process states?

- Running - process is running on CPU.
- Ready - ready to run, but not actually running on the CPU.
- Waiting - waiting for some event like IO to happen.

☐ When do scheduling decisions take place? When does CPU choose which process to run? Are a variety of possibilities:

- When process switches from running to waiting. Could be because of IO request, because wait for child to terminate, or wait for synchronization operation (like lock acquisition) to complete.
- When process switches from running to ready - on completion of interrupt handler, for example. Common example of interrupt handler - timer interrupt in interactive systems. If scheduler switches processes in this case, it has preempted the running process. Another common case interrupt handler is the IO completion handler.
- When process switches from waiting to ready state (on completion of IO or acquisition of a lock, for example).
- When a process terminates.

☐ How to evaluate scheduling algorithm? There are many possible criteria:

- CPU Utilization: Keep CPU utilization as high as possible. (What is utilization, by the way?).
- Throughput: number of processes completed per unit time.
- Turnaround Time: mean time from submission to completion of process.
- Waiting Time: Amount of time spent ready to run but not running.
- Response Time: Time between submission of requests and first response to the request.
- Scheduler Efficiency: The scheduler doesn't perform any useful work, so any time it takes is pure overhead. So, need to make the scheduler very efficient.

☐ Big difference: Batch and Interactive systems. In batch systems, typically want good throughput or turnaround time. In interactive systems, both of these are still usually important (after all, want some computation to happen), but response time is usually a primary consideration. And, for some systems, throughput or turnaround time is not really relevant - some processes conceptually run forever.
☐ Difference between long and short term scheduling. Long term scheduler is given a set of processes and decides which ones should start to run. Once they start running, they may suspend because of IO or because of preemption. Short term scheduler decides which of the available jobs that long term scheduler has decided are runnable to actually run.
☐ Let's start looking at several vanilla scheduling algorithms.
☐ First-Come, First-Served. One ready queue, OS runs the process at head of queue, new processes come in at the end of the queue. A process does not give up CPU until it either terminates or performs IO.
☐ Consider performance of FCFS algorithm for three compute-bound processes. What if have 4 processes P1 (takes 24 seconds), P2 (takes 3 seconds) and P3 (takes 3 seconds). If arrive in order P1, P2, P3, what is

- Waiting Time? (24 + 27) / 3 = 17
- Turnaround Time? (24 + 27 + 30) = 27.
- Throughput? 30 / 3 = 10.

What about if processes come in order P2, P3, P1? What is

- Waiting Time? (3 + 3) / 2 = 6
- Turnaround Time? (3 + 6 + 30) = 13.
- Throughput? 30 / 3 = 10.

☐ Shortest-Job-First (SJF) can eliminate some of the variance in Waiting and Turnaround time. In fact, it is optimal with respect to average waiting time. Big problem: how does scheduler figure out how long will it take the process to run?
☐ For long term scheduler running on a batch system, user will give an estimate. Usually pretty good - if it is too short, system will cancel job before it finishes. If too long, system will hold off on running the process. So, users give pretty good estimates of overall running time.
☐ For short-term scheduler, must use the past to predict the future. Standard way: use a time-decayed exponentially weighted average of previous CPU bursts for each process. Let T_n be the measured burst time of the nth burst, s_n be the predicted size of next CPU burst. Then, choose a weighting factor w, where $0 <= w <= 1$ and compute $s_{n+1} = w\ T_n + (1 - w)s_n$. s_0 is defined as some default constant or system average.

☐ *w* tells how to weight the past relative to future. If choose *w* = .5, last observation has as much weight as entire rest of the history. If choose *w* = 1, only last observation has any weight. Do a quick example.
☐ Preemptive vs. Non-preemptive SJF scheduler. Preemptive scheduler reruns scheduling decision when process becomes ready. If the new process has priority over running process, the CPU preempts the running process and executes the new process. Non-preemptive scheduler only does scheduling decision when running process voluntarily gives up CPU. In effect, it allows every running process to finish its CPU burst.
☐ Consider 4 processes P1 (burst time 8), P2 (burst time 4), P3 (burst time 9) P4 (burst time 5) that arrive one time unit apart in order P1, P2, P3, P4. Assume that after burst happens, process is not reenabled for a long time (at least 100, for example). What does a preemptive SJF scheduler do? What about a non-preemptive scheduler?
☐ Priority Scheduling. Each process is given a priority, then CPU executes process with highest priority. If multiple processes with same priority are runnable, use some other criteria - typically FCFS. SJF is an example of a priority-based scheduling algorithm. With the exponential decay algorithm above, the priorities of a given process change over time.
☐ Assume we have 5 processes P1 (burst time 10, priority 3), P2 (burst time 1, priority 1), P3 (burst time 2, priority 3), P4 (burst time 1, priority 4), P5 (burst time 5, priority 2). Lower numbers represent higher priorities. What would a standard priority scheduler do?
☐ Big problem with priority scheduling algorithms: starvation or blocking of low-priority processes. Can use aging to prevent this - make the priority of a process go up the longer it stays runnable but isn't run.
☐ What about interactive systems? Cannot just let any process run on the CPU until it gives it up - must give response to users in a reasonable time. So, use an algorithm called round-robin scheduling. Similar to FCFS but with preemption. Have a time quantum or time slice. Let the first process in the queue run until it expires its quantum (i.e. runs for as long as the time quantum), then run the next process in the queue.
☐ Implementing round-robin requires timer interrupts. When schedule a process, set the timer to go off after the time quantum amount of time expires. If process does IO before timer goes off, no problem - just run next process. But if process expires its quantum, do a context switch. Save the state of the running process and run the next process.
☐ How well does RR work? Well, it gives good response time, but can give bad waiting time. Consider the waiting times under round robin for 3 processes P1 (burst time 24), P2 (burst time 3), and P3 (burst time 4) with time quantum 4. What happens, and what is average waiting time? What gives best waiting time?
☐ What happens with really a really small quantum? It looks like you've got a CPU that is 1/n as powerful as the real CPU, where n is the number of processes. Problem with a small quantum - context switch overhead.
☐ What about having a really small quantum supported in hardware? Then, you have something called multithreading. Give the CPU a bunch of registers and heavily pipeline the execution. Feed the processes into the pipe one by one. Treat memory access like IO - suspend the thread until the data comes back from the memory. In the meantime, execute other threads. Use computation to hide the latency of accessing memory.
☐ What about a really big quantum? It turns into FCFS. Rule of thumb - want 80 percent of CPU bursts to be shorter than time quantum.
☐ Multilevel Queue Scheduling - like RR, except have multiple queues. Typically, classify processes into separate categories and give a queue to each category. So, might have system, interactive and batch processes, with the priorities in that order. Could also allocate a percentage of the CPU to each queue.
☐ Multilevel Feedback Queue Scheduling - Like multilevel scheduling, except processes can move between queues as their priority changes. Can be used to give IO bound and interactive processes CPU priority over CPU bound processes. Can also prevent starvation by increasing the priority of processes that have been idle for a long time.
☐ A simple example of a multilevel feedback queue scheduling algorithm. Have 3 queues, numbered 0, 1, 2 with corresponding priority. So, for example, execute a task in queue 2 only when queues 0 and 1 are empty.
☐ A process goes into queue 0 when it becomes ready. When run a process from queue 0, give it a quantum of 8 ms. If it expires its quantum, move to queue 1. When execute a process from queue 1, give it a quantum of 16. If it expires its quantum, move to queue 2. In queue 2, run a RR scheduler with a large quantum if in an interactive system or an FCFS scheduler if in a batch system. Of course, preempt queue 2 processes when a new process becomes ready.
☐ Another example of a multilevel feedback queue scheduling algorithm: the Unix scheduler. We will go over a simplified version that does not include kernel priorities. The point of the algorithm is to fairly allocate the CPU between processes, with processes that have not recently used a lot of CPU resources given priority over processes that have.
☐ Processes are given a base priority of 60, with lower numbers representing higher priorities. The system clock generates an interrupt between 50 and 100 times a second, so we will assume a value of 60 clock

interrupts per second. The clock interrupt handler increments a CPU usage field in the PCB of the interrupted process every time it runs.
☐ The system always runs the highest priority process. If there is a tie, it runs the process that has been ready longest. Every second, it recalculates the priority and CPU usage field for every process according to the following formulas.

- CPU usage field = CPU usage field / 2
- Priority = CPU usage field / 2 + base priority

☐ So, when a process does not use much CPU recently, its priority rises. The priorities of IO bound processes and interactive processes therefore tend to be high and the priorities of CPU bound processes tend to be low (which is what you want).
☐ Unix also allows users to provide a ``nice'' value for each process. Nice values modify the priority calculation as follows:

- Priority = CPU usage field / 2 + base priority + nice value

So, you can reduce the priority of your process to be ``nice'' to other processes (which may include your own).
☐ In general, multilevel feedback queue schedulers are complex pieces of software that must be tuned to meet requirements.
☐ Anomalies and system effects associated with schedulers.
☐ Priority interacts with synchronization to create a really nasty effect called priority inversion. A priority inversion happens when a low-priority thread acquires a lock, then a high-priority thread tries to acquire the lock and blocks. Any middle-priority threads will prevent the low-priority thread from running and unlocking the lock. In effect, the middle-priority threads block the high-priority thread.
☐ How to prevent priority inversions? Use priority inheritance. Any time a thread holds a lock that other threads are waiting on, give the thread the priority of the highest-priority thread waiting to get the lock. Problem is that priority inheritance makes the scheduling algorithm less efficient and increases the overhead.
☐ Preemption can interact with synchronization in a multiprocessor context to create another nasty effect - the convoy effect. One thread acquires the lock, then suspends. Other threads come along, and need to acquire the lock to perform their operations. Everybody suspends until the lock that has the thread wakes up. At this point the threads are synchronized, and will convoy their way through the lock, serializing the computation. So, drives down the processor utilization.
☐ If have non-blocking synchronization via operations like LL/SC, don't get convoy effects caused by suspending a thread competing for access to a resource. Why not? Because threads don't hold resources and prevent other threads from accessing them.
☐ Similar effect when scheduling CPU and IO bound processes. Consider a FCFS algorithm with several IO bound and one CPU bound process. All of the IO bound processes execute their bursts quickly and queue up for access to the IO device. The CPU bound process then executes for a long time. During this time all of the IO bound processes have their IO requests satisfied and move back into the run queue. But they don't run - the CPU bound process is running instead - so the IO device idles. Finally, the CPU bound process gets off the CPU, and all of the IO bound processes run for a short time then queue up again for the IO devices. Result is poor utilization of IO device - it is busy for a time while it processes the IO requests, then idle while the IO bound processes wait in the run queues for their short CPU bursts. In this case an easy solution is to give IO bound processes priority over CPU bound processes.
☐ In general, a convoy effect happens when a set of processes need to use a resource for a short time, and one process holds the resource for a long time, blocking all of the other processes. Causes poor utilization of the other resources in the system.

Overview of File System of Operating System:

- When does a process need to access OS functionality? Here are several examples
- Reading a file. The OS must perform the file system operations required to read the data off of disk.
- Creating a child process. The OS must set stuff up for the child process.
- Sending a packet out onto the network. The OS typically handles the network interface.

Why have the OS do these things? Why doesn't the process just do them directly?

- Convenience. Implement the functionality once in the OS and encapsulate it behind an interface that everyone uses. So, processes just deal with the simple interface, and don't have to write complicated low-level code to deal with devices.
- Portability. OS exports a common interface typically available on many hardware platforms. Applications do not contain hardware-specific code.
- Protection. If give applications complete access to disk or network or whatever, they can corrupt data from other applications, either maliciously or because of bugs. Having the OS do it eliminates security problems between applications. Of course, applications still have to trust the OS.

☐ How do processes invoke OS functionality? By making a system call. Conceptually, processes call a subroutine that goes off and performs the required functionality. But OS must execute in a different protection domain than the application. Typically, OS executes in supervisor mode, which allows it to do things like manipulate the disk directly.
☐ To switch from normal user mode to supervisor mode, most machines provide a system call instruction. This instruction causes an exception to take place. The hardware switches from user mode to supervisor mode and invokes the exception handler inside the operating system. There is typically some kind of convention that the process uses to interact with the OS.
☐ Let's do an example - the `Open` system call. System calls typically start out with a normal subroutine call. In this case, when the process wants to open a file, it just calls the `Open`routine in a system library someplace. /* Open the Nachos file "name", and return an "OpenFileId" that can * be used to read and write to the file. */ OpenFileId Open(char *name);
☐ Inside the library, the `Open` subroutine executes a `syscall`instruction, which generates a system call exception. Open: addiu $2,$0,SC_Open syscall j $31 .end Open By convention, the `Open` subroutine puts a number (in this case `SC_Open`) into register 2. Inside the exception handler the OS looks at register 2 to figure out what system call it should perform.
☐ The `Open` system call also takes a parameter - the address of the character string giving the name of the file to open. By convention, the compiler puts this parameter into register 4 when it generates the code that calls the `Open` routine in the library. So, the OS looks in that register to find the address of the name of the file to open.
☐ More conventions: succeeding parameters are put into register 5, register 6, etc. Any return values from the system call are put into register 2.
☐ Inside the exception handler, the OS figures out what action to take, performs the action, then returns back to the user program.
☐ There are other kinds of exceptions. For example, if the program attempts to deference a NULL pointer, the hardware will generate an exception. The OS will have to figure out what kind of exception took place and handle it accordingly. Another kind of exception is a divide by 0 fault.
☐ Similar things happen on a interrupt. When an interrupt occurs, the hardware puts the OS into supervisor mode and invokes an interrupt handler. The difference between interrupts and exceptions is that interrupts are generated by external events (the disk IO completes, a new character is typed at the console, etc.) while exceptions are generated by a running program.
☐ Object file formats. To run a process, the OS must load in an executable file from the disk into memory. What does this file contain? The code to run, any initialized data, and a specification for how much space the uninitialized data takes up. May also be other stuff to help debuggers run, etc.
☐ The compiler, linker and OS must agree on a format for the executable file. For example, Nachos uses the following format for executables: #define NOFFMAGIC 0xbadfad /* magic number denoting Nachos * object code file */ typedef struct segment { int virtualAddr; /* location of segment in virt addr space */ int inFileAddr; /* location of segment in this file */ int size; /* size of segment */ } Segment; typedef struct noffHeader { int noffMagic; /* should be NOFFMAGIC */ Segment code; /* executable code segment */ Segment initData; /* initialized data segment */ Segment uninitData; /* uninitialized data segment -- * should be zero'ed before use */ } NoffHeader;
☐ What does the OS do when it loads an executable in?

- Reads in the header part of the executable.
- Checks to see if the magic number matches.
- Figures out how much space it needs to hold the process. This includes space for the stack, the code, the initialized data and the uninitialized data.

- If it needs to hold the entire process in physical memory, it goes off and finds the physical memory it needs to hold the process.
- It then reads the code segment in from the file to physical memory.
- It then reads the initialized data segment in from the file to physical memory.
- It zeros the stack and unintialized memory.

☐ How does the operating system do IO? First, we give an overview of how the hardware does IO.
☐ There are two basic ways to do IO - memory mapped IO and programmed IO.

- Memory mapped IO - the control registers on the IO device are mapped into the memory space of the processor. The processor controls the device by performing reads and writes to the addresses that the IO device is mapped into.
- Programmed IO - the processor has special IO instructions like IN and OUT. These control the IO device directly.

☐ Writing the low level, complex code to control devices can be a very tricky business. So, the OS encapsulates this code inside things called device drivers. There are several standard interfaces that device drivers present to the kernel. It is the job of the device driver to implement its standard interface for its device. The rest of the OS can then use this interface and doesn't have to deal with complex IO code.
☐ For example, Unix has a block device driver interface. All block device drivers support a standard set of calls like open, close, read and write. The disk device driver, for example, translates these calls into operations that read and write sectors on the disk.
☐ Typically, IO takes place asynchronously with respect to the processor. So, the processor will start an IO operation (like writing a disk sector), then go off and do some other processing. When the IO operation completes, it interrupts the processor. The processor is typically vectored off to an interrupt handler, which takes whatever action needs to take place.
☐ Here is how Nachos does IO. Each device presents an interface. For example, the disk interface is in disk.h, and has operations to start a read and write request. When the request completes, the "hardware" invokes the HandleInterrupt method.
☐ Only one thread can use each device at a time. Also, threads typically want to use devices synchronously. So, for example, a thread will perform a disk operation then wait until the disk operation completes. Nachos therefore encapsulates the device interface inside a higher level interface that provides synchronous, synchronized access to the device. For the disk device, this interface is in synchdisk.h. This provides operations to read and write sectors, for example.
☐ Each method in the synchronous interface ensures exclusive access to the IO device by acquiring a lock before it performs any operation on the device.
☐ When the synchronous method gets exclusive access to the device, it performs the operation to start the IO. It then uses a semaphore (P operation) to block until the IO operation completes. When the IO operation completes, it invokes an interrupt handler. This handler performs a V operation on the semaphore to unblock the synchronous method. The synchronous method then releases the lock and returns back to the calling thread.

Overview of Memory Management System:

☐
Point of memory management algorithms - support sharing of main memory. We will focus on having multiple processes sharing the same physical memory. Key issues:

- Protection. Must allow one process to protect its memory from access by other processes.
- Naming. How do processes identify shared pieces of memory.
- Transparency. How transparent is sharing. Does user program have to manage anything explicitly?
- Efficiency. Any memory management strategy should not impose too much of a performance burden.

☐ Why share memory between processes? Because want to multiprogram the processor. To time share system, to overlap computation and I/O. So, must provide for multiple processes to be resident in physical memory at the same time. Processes must share the physical memory.
☐ Historical Development.

- For first computers, loaded one program onto machine and it executed to completion. No sharing required. OS was just a subroutine library, and there was no protection. What addresses does program generate?
- Desire to increase processor utilization in the face of long I/O delays drove the adoptation of multiprogramming. So, one process runs until it does I/O, then OS lets another process run. How do processes share memory? Alternatives:
  - Load both processes into memory, then switch between them under OS control. Must relocate program when load it. Big Problem: Protection. A bug in one process can kill the other process. MS-DOS, MS-Windows use this strategy.
  - Copy entire memory of process to disk when it does I/O, then copy back when it restarts. No need to relocate when load. Obvious performance problems. Early version of Unix did this.
  - Do access checking on each memory reference. Give each program a piece of memory that it can access, and on every memory reference check that it stays within its address space. Typical mechanism: base and bounds registers. Where is check done? Answer: in hardware for speed. When OS runs process, loads the base and bounds registers for that process. Cray-1 did this. Note: there is now a translation process. Program generates virtual addresses that get translated into physical addresses. But, no longer have a protection problem: one process cannot access another's memory, because it is outside its address space. If it tries to access it, the hardware will generate an exception.

☐ End up with a model where physical memory of machine is dynamically allocated to processes as they enter and exit the system. Variety of allocation strategies: best fit, first fit, etc. All suffer from external fragmentation. In worst case, may have enough memory free to load a process, but can't use it because it is fragmented into little pieces.
☐ What if cannot find a space big enough to run a process? Either because of fragmentation or because physical memory is too small to hold all address spaces. Can compact and relocate processes (easy with base and bounds hardware, not so easy for direct physical address machines). Or, can swap a process out to disk then restore when space becomes available. In both cases incur copying overhead. When move process within memory, must copy between memory locations. When move to disk, must copy back and forth to disk.
☐ One way to avoid external fragmentation: allocate physical memory to processes in fixed size chunks called page frames. Present abstraction to application of a single linear address space. Inside machine, break address space of application up into fixed size chunks called pages. Pages and page frames are same size. Store pages in page frames. When process generates an address, dynamically translate to the physical page frame which holds data for that page.
☐ So, a virtual address now consists of two pieces: a page number and an offset within that page. Page sizes are typically powers of 2; this simplifies extraction of page numbers and offsets. To access a piece of data at a given address, system automatically does the following:

- Extracts page number.
- Extracts offset.
- Translate page number to physical page frame id.
- Accesses data at offset in physical page frame.

☐ How does system perform translation? Simplest solution: use a page table. Page table is a linear array indexed by virtual page number that gives the physical page frame that contains that page. What is lookup process?

- Extract page number.
- Extract offset.
- Check that page number is within address space of process.
- Look up page number in page table.

- Add offset to resulting physical page number
- Access memory location.

☐ With paging, still have protection. One process cannot access a piece of physical memory unless its page table points to that physical page. So, if the page tables of two processes point to different physical pages, the processes cannot access each other's physical memory.
☐ Fixed size allocation of physical memory in page frames dramatically simplifies allocation algorithm. OS can just keep track of free and used pages and allocate free pages when a process needs memory. There is no fragmentation of physical memory into smaller and smaller allocatable chunks.
☐ But, are still pieces of memory that are unused. What happens if a program's address space does not end on a page boundary? Rest of page goes unused. This kind of memory loss is called internal fragmentation.

Introduction of Paging in Operating System:

☐
Basic idea: allocate physical memory to processes in fixed size chunks called page frames. Present abstraction to application of a single linear address space. Inside machine, break address space of application up into fixed size chunks called pages.

Pages and page frames are same size. Store pages in page frames. When process generates an address, dynamically translate to the physical page frame which holds data for that page.
☐ So, a virtual address now consists of two pieces: a page number and an offset within that page. Page sizes are typically powers of 2; this simplifies extraction of page numbers and offsets. To access a piece of data at a given address, system automatically does the following:

- Extracts page number.
- Extracts offset.
- Translate page number to physical page frame id.
- Accesses data at offset in physical page frame.

☐ How does system perform translation? Simplest solution: use a page table. Page table is a linear array indexed by virtual page number that gives the physical page frame that contains that page. What is lookup process?

- Extract page number.
- Extract offset.
- Check that page number is within address space of process.
- Look up page number in page table.
- Add offset to resulting physical page number
- Access memory location.

Problem: for each memory access that processor generates, must now generate two physical memory accesses.
☐ Speed up the lookup problem with a cache. Store most recent page lookup values in TLB. TLB design options: fully associative, direct mapped, set associative, etc. Can make direct mapped larger for a given amount of circuit space.
☐ How does lookup work now?

- Extract page number.

- Extract offset.
- Look up page number in TLB.
- If there, add offset to physical page number and access memory location.
- Otherwise, trap to OS. OS performs check, looks up physical page number, and loads translation into TLB. Restarts the instruction.

☐ Like any cache, TLB can work well, or it can work poorly. What is a good and bad case for a direct mapped TLB? What about fully associative TLBs, or set associative TLB?
☐ Fixed size allocation of physical memory in page frames dramatically simplifies allocation algorithm. OS can just keep track of free and used pages and allocate free pages when a process needs memory. There is no fragmentation of physical memory into smaller and smaller allocatable chunks.
☐ But, are still pieces of memory that are unused. What happens if a program's address space does not end on a page boundary? Rest of page goes unused. Book calls this internal fragmentation.
☐ How do processes share memory? The OS makes their page tables point to the same physical page frames. Useful for fast interprocess communication mechanisms. This is very nice because it allows transparent sharing at speed.
☐ What about protection? There are a variety of protections:

- Preventing one process from reading or writing another process' memory.
- Preventing one process from reading another process' memory.
- Preventing a process from reading or writing some of its own memory.
- Preventing a process from reading some of its own memory.

How is this protection integrated into the above scheme?
☐ Preventing a process from reading or writing memory: OS refuses to establish a mapping from virtual address space to physical page frame containing the protected memory. When program attempts to access this memory, OS will typically generate a fault. If user process catches the fault, can take action to fix things up.
☐ Preventing a process from writing memory, but allowing a process to read memory. OS sets a write protect bit in the TLB entry. If process attempts to write the memory, OS generates a fault. But, reads go through just fine.
☐ Virtual Memory Introduction.
☐ When a segmented system needed more memory, it swapped segments out to disk and then swapped them back in again when necessary. Page based systems can do something similar on a page basis.
☐ Basic idea: when OS needs to a physical page frame to store a page, and there are none free, it can select one page and store it out to disk. It can then use the newly free page frame for the new page. Some pragmatic considerations:

- In practice, it makes sense to keep a few free page frames. When number of free pages drops below this threshold, choose a page and store it out. This way, can overlap I/O required to store out a page with computation that uses the newly allocated page frame.
- In practice the page frame size usually equals the disk block size. Why?
- Do you need to allocate disk space for a virtual page before you swap it out? (Not if always keep one page frame free) Why did BSD do this? At some point OS must refuse to allocate a process more memory because has no swap space. When can this happen? (malloc, stack extension, new process creation).

☐ When process tries to access paged out memory, OS must run off to the disk, find a free page frame, then read page back off of disk into the page frame and restart process.
☐ What is advantage of virtual memory/paging?

- Can run programs whose virtual address space is larger than physical memory. In effect, one process shares physical memory with itself.
- Can also flexibly share machine between processes whose total address space sizes exceed the physical memory size.
- Supports a wide range of user-level stuff - See Li and Appel paper.

☐ Disadvantages of VM/paging: extra resource consumption.

- Memory overhead for storing page tables. In extreme cases, page table may take up a significant portion of virtual memory. One Solution: page the page table. Others: go to a more complicated data structure for storing virtual to physical translations.
- Translation overhead.

Lexical Analysis

- Lexical Analyser
- Regular Expressions and Strings
- Languages
- Regolar Expressions

- Example: Grammer Format
- Working Example
- Automata
- Converting NFA to DFA
- Converting Regolar Expression to NFA

Lexical Analyser:

Lexical Analyzer

The lexical analyzer is responsible for:

- Reading in a stream of input characters
- Produce as output a sequence of tokens
- Upon get-next-token request from the parser, the analyzer reads in a string of characters (lexeme) to generate a token.

For the remainder of the lecture, we will go over some examples and an in class discussion of how the above works, and how we can implement it.

- How do we check for lexemes. If stmt, case/switch stmt, search an array (see example code)
- How do we remove whitespace?
- What is the format of returned tokens?

Introduction of Regular Expressions and Strings:

Regular Expressions and Strings

String

- *prefix* of s : A target string derived from another source string by removing 0 or more symbols from the end of the source string
- *suffix* of s : A target string derived from another source string by removing 0 or more symbols from the beginning of the source string
- *substring* of s : A target string derived from another source string by removing 0 or more symbols from the beginning and/or 0 or more symbols from the end of the source string
- *proper* prefix, suffix or substring of s : A target string that is not equal to the original string.
- *subsequence* of s : A target string derived from another source string by removing 0 or more (not necessarily contiguous) symbols from the source string.

Introduction of Language in Lexical Analysis:

Languages

We define a language as a set of strings over a fixed alphabet (which is a set of symbols) and may also include the empty string. Lets look at the following examples for an understanding of operations on languages. Assuming that L and D are languages or sets of symbols (letters and digits respectively)

- $L \cup D$ (union) is the language consisting of strings of letters and digits. ($L \cup D = \{s \mid s \in L \vee s \in D\}$
- LD is the set of strings consisting of one letter following by one digit ($LD = \{st \mid s \in L \wedge t \in D\}$)
- L^4 is the set of all four-letter strings
- L^+ is the set of all finite strings of letters in L
- $L*$ is L^+ and also the empty string

Introduction to Regular Expressions:

Regular Expressions

What is a regular expression? And why do we care?

- A regular expression is a simple expression that denotes a language. (A collection of strings that can be recognized).
- It has several features that promote easy writing of complex strings. For example:
  - a, b, c - we use to denote terminal symbols of the language
  - r, s, t - we use to denote regular expressions
  - $a \mid b$ - denotes choice (can use [abc] to denote choice – one of the list)
  - ab - denotes concatenation
  - r^* - denotes 0 or more occurrences
  - r^+ - denotes 1 or more occurrences
  - $r^?$ - denotes optional occurrence
- Examples
  - $(aa)^+$ – all non-empty strings of even numbers of a's
  - $(a-z, A-Z)(a-z, A-Z, 0-9)^*$ – langauge of identifiers
  - $0 \mid (1-9)^?(0-9)^*$ – possible integer literals

Introduction to Regular Grammar:

Example: Grammar Fragment

$stmt \longrightarrow$ if $expr$ then $stmt$ |
if $expr$ then $stmt$ else $stmt$ | ϵ

$expr \longrightarrow$ $term$ ⟨relop⟩ $term$ | $term$

$term \longrightarrow$ id | num

Regular Expression	*Token*	*Attribute Value*
ws hline if	**if**	
then	**then**	
else	**else**	
id	**id**	pointer to table entry
num	**num**	pointer to table entry
$<$	**relop**	LT
$<=$	**relop**	LE
$==$	**relop**	EQ
$!=$	**relop**	NE
$>$	**relop**	GT
$>=$	**relop**	GE

Examples:

Working Example

Consider the following grammar:

$program$	::=	program id is $block$
$block$	::=	$stmt^+$
$stmt$	::=	$assn \mid if$
$assn$	::=	id := $numExpr$;
if	::=	if $boolExpr$ then $stmt$ else $stmt$
$numExpr$	::=	$numExpr$ + $term \mid$
		$numExpr$ - $term \mid term$
$term$	::=	$term$ * $factor \mid$
		$term$ / $factor \mid factor$
$factor$	::=	$number \mid$'(' $numExpr$ ')'
$boolExpr$	::=	$numExpr\ relop\ numExpr$
$relop$	::=	$<\mid<=\mid==\mid!=\mid>=\mid>$
id	::=	$letter\ letDigUnd^*$
$letDigUnd$	::=	$letter \mid digit \mid\_$
$number$	::=	$zero \mid nonZero\ digit^*$
$digit$	::=	$zero \mid nonZero$
$zero$	::=	0
$nonZero$	::=	[1..9]
$letter$	::=	[$a..zA..Z$]

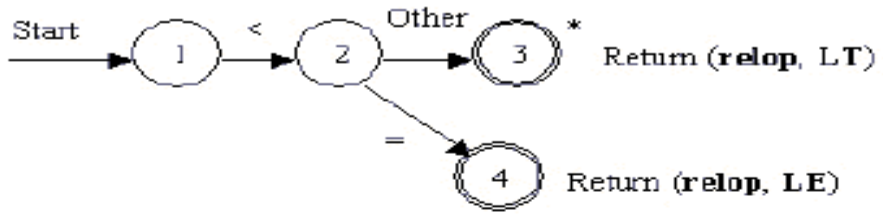
The lexemes for the tokens of this grammar are:

Keywords	Operators	Other Values
program	+	letter = [a..zA..Z]
is	−	digit = [0..9]
if	*	
then	/	
else	relop = [<, <=, ==, ! =, >=, >]	
	:=	

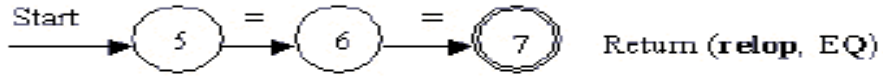
How can we recognize these? One way is to generate a transition diagram for each lexeme.

We will make the assumption that keywords exists in a symbol table and thus are recognized by the identifier transition diagram. Here are ALL of the transition diagrams for the above grammar.

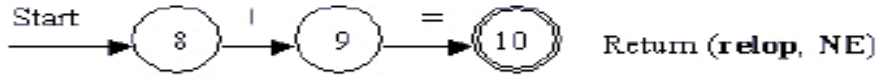
To recognize < or <=



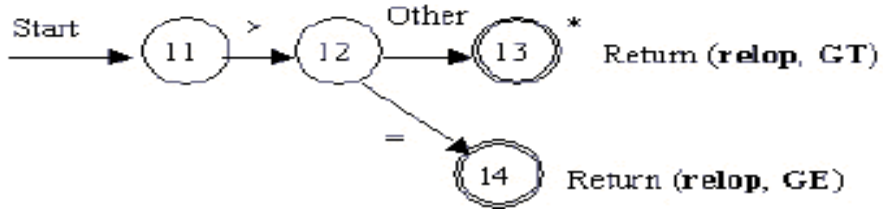
To recognize ==



To recognize ! =

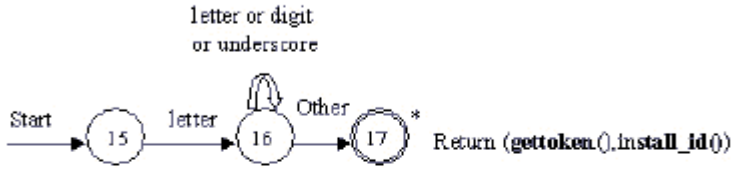


To recognize > or >=



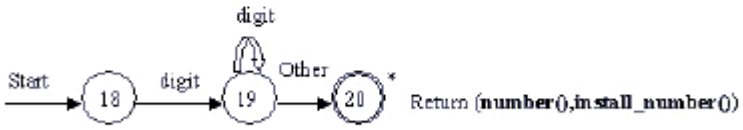
To recognize identifiers.

We use the function gettoken() to determine if the current identifier is a reserved word. If it is, we return the token corresponding to that reserved word. We use the function install_id() to return a symbol table entry for any other identifier – it will return a default value for reserved words.

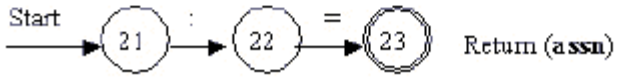


To recognize numbers

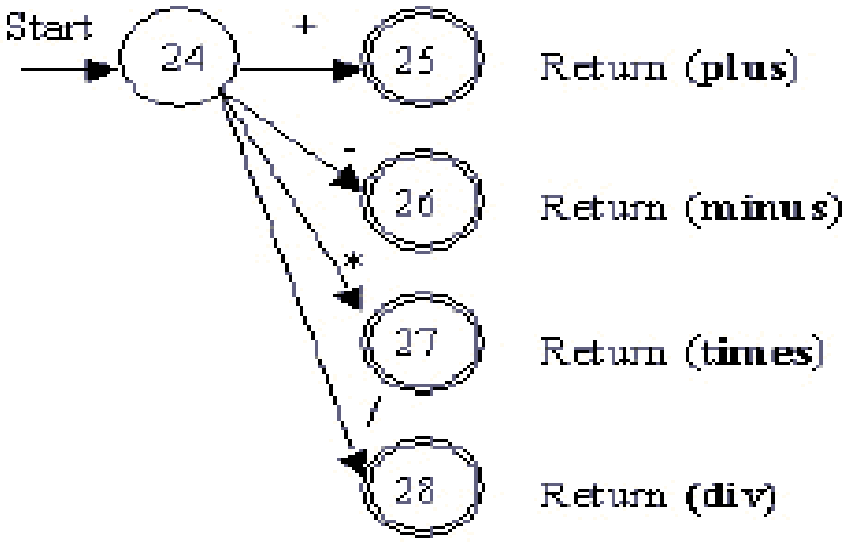
We insert ALL constants into a constant table using install_number()



To recognize assignment operator :=



To recognize math operators (+, −, *, /)



Introduction To Finite Automata:

Automata

NFA

An NFA (Non-Deterministic Finite-State Automata) has the following characteristics

- set of states
- set of input symbols/characters
- a transition function *move* that maps the current state and input symbol to a new state
- an initial state
- a set of *final* states

Consider an NFA that accepts $(a \mid b) * abb$. We can represent this diagrammatically or in a table.

State	INPUT SYMBOL	
	a	b
0	$\{0,1\}$	$\{0\}$
1	-	$\{2\}$
2	-	$\{3\}$

We can simulate an NFA by keeping track of all possible states that we are in and processing the inputs for each of those states. Or we can convert the NFA to a DFA and then simulate the DFA.

DFA

A DFA (deterministic finite state automata) is the same as an NFA with the following additional constraints

- There are no epsilon-transitions from any state
- For any single input symbol there can be at most one transition out of any single state. (The same symbol can transit out of different states, just not the same state more than once.)

To simulate a DFA, we can execute the following code

```
s := s0;
c := next_char();
while c != eof do
   s := move(s,c);
   c := next_char();
end;
if s in in F then
   return "yes";
else return "no";
```

Conversion of Automata:

Converting NFA to DFA

To convert an NFA to a DFA we need to create a collection of DFA states that represent a subset of NFA states (this subset representing a set of possible states that we are in for the NFA). We will define Dstate as this set of states and Dtran as the transition functions for the resulting DFA.

The algorithm requires the following operations:

Operation	Description
epsilon-closure(s)	This is the set of all NFA states reachable from the single NFA state s without using any input characters. (This is s itself and any states reachable through one or more epsilon-transitions
epsilon-closure(T)	T represents a set of NFA states, and such this is the union of all epsilon-closure(s) operations for all s in T.
move(T,a)	Set of all NFA states that we can reach from any state in T upon reading the input symbol

Now we need to calculate the Dstates with the following algorithm (this is java-like, see the text for a generic description

```
Dstates = epsilon_closure(s0);
while (T = Dstates.get_unmarked_state())
         != NULL do {
 Dstates.mark(T);
 for (a=first_character();
      a != last_symbol();
      a=get_next_symbol()) {
   U := epsilon_closure(move(T,a));
   if (!Dstates.member_of(U)) then {
     Dstates.add(U);
     Dtran[T,a] = U
   }
 }
}
```

Where we calculate the epsilon-closure of T using:

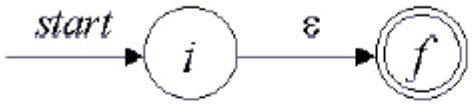
```
stack.push_all(T);
result = T;
while (!stack.empty()) do {
  t = stack.top();
  for(u = first_state();
      u != last_state();
      u = next_state()) {
    if(!result.member_of(u)) then {
      result.add(u);
      stack.push(u);
    }
  }
}
```

Regular Expression to NFA Conversion:

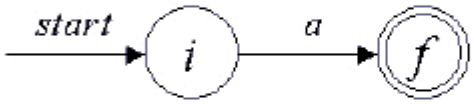
Converting Regular Expression to NFA

Now that we have looked at converting an NFA to a DFA; which we know how to simulate, how do we create the original NFA's to begin with? This section discusses a process to do just that.

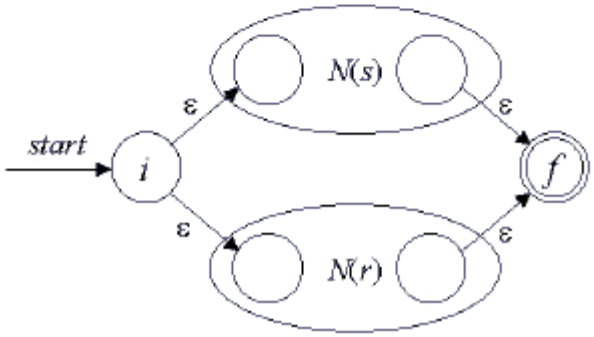
1. For an epsilon regular expression, construct:



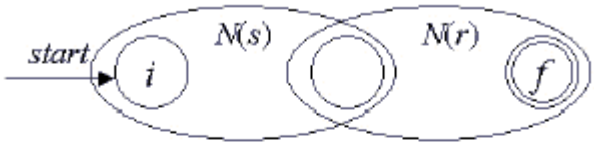
2. For a single symbol a construct:



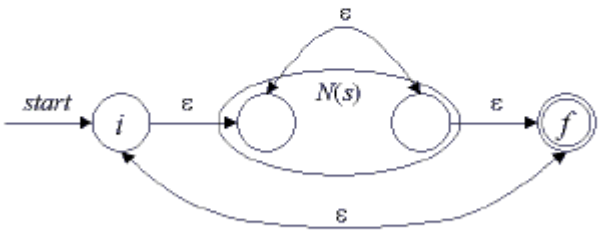
3. For composition of regular expressions
   (a) For the regular expression $(s \mid r)$, assuming $N(s)$ is the NFA for s and $N(r)$ is the NFA for r, construct:



(b) For the regular expression (sr), assuming $N(s)$ is the NFA for s and $N(r)$ is the NFA for r, construct:



(c) For the regular expression (s)* construct:



This technique builds NFA's. We can now convert these to DFA's. As an example, use this technique to "build" the NFA for $((a \mid b)^* abb)$.

Data Base Management Systems

- Database system Concept and Architecture
- Entity Relationship and Enhanced E-R
- Relational Data Model and Relational Algebra
- Relational Database Design
- Query Language-SQL

Normalization

Database System Concept:

The term database originated within the computing discipline. Although its meaning has been broadened by popular use, even to include non-electronic databases, this article is about computer databases.

A computer database is astructured collection of records or data that is stored in a computer system so that a computer program or person using a query language can consult it to answer queries
href="http://en.wikipedia.org/wiki/Database#_note-0">[1]. The records retrieved in answer to queries are information that can be used to make decisions.

The computer program used to manage and query a database is known as a database management system(DBMS). The properties and design of database systems are included in the study of information science.

A typical query could be a question such as, How many hamburgers with two or more beef patties were sold in the month of March in New Jersey?". To answer such a question, the database would have to store information about hamburgers sold, including number of patties, sales date, and the region.

The central concept of a database is that of a collection of records, or pieces of information. Typically, for a given database, there is a structural description of the type of facts held in that database: this description is known as a schema. The schema describes the objects that are represented in the database, and the relationships among them. There are a number of different ways of organizing a schema, that is, of modeling the database structure: these are known as database models(or data models). The model in most common use today is the relational model, which in layman's terms represents all information in the form of multiple related tables each consisting of rows and columns (the formal definition uses mathematical terminology). This model represents relationships by the use of values common to more than one table. Other models such as the hierarchical model and the network model use a more explicit representation of relationships.

The term database refers to the collection of related records, and the software should be referred to as the database management system or DBMS. When the

context is ambiguous, however, many database administrators and programmers use the term database to cover both meanings.

Many professionals consider a collection of data to constitute a database only if it has certain properties: for example, if the data is managed to ensure its integrity and quality, if it allows shared access by a community of users, if it has a schema, or if it supports a query language. However, there is no definition of these properties that is universally agreed upon.

Database management systems are usually categorized according to the data model that they support: relational, object-relational, network, and so on. The data model will tend to determine the query languages that are available to access the database. A great deal of the internal engineering of a DBMS, however, is independent of the data model, and is concerned with managing factors such as performance, concurrency, integrity, and recovery from hardware failures. In these areas there are large differences between products.

Purpose of Database System:

- To see why database management systems are necessary, let's look at a typical "file-processing system'' supported by a conventional operating system.

  The application is a savings bank:
  - Savings account and customer records are kept in permanent system files.
  - Application programs are written to manipulate files to perform the following tasks:
    - Debit or credit an account.
    - Add a new account.
    - Find an account balance.
    - Generate monthly statements.
- Development of the system proceeds as follows:
  - New application programs must be written as the need arises.
  - New permanent files are created as required.

  - **but** over a long period of time files may be in different formats, and
  - Application programs may be in different languages.

-
- So we can see there are problems with the straight file-processing approach:
  - Data redundancy and inconsistency
    - Same information may be duplicated in several places.
    - All copies may not be updated properly.
  - Difficulty in accessing data
    - May have to write a new application program to satisfy an unusual request.
    - E.g. find all customers with the same postal code.
    - Could generate this data manually, but a long job...
  - Data isolation
    - Data in different files.
    - Data in different formats.
    - Difficult to write new application programs.
  - Multiple users
    - Want concurrency for faster response time.
    - Need protection for concurrent updates.
    - E.g. two customers withdrawing funds from the same account at the same time - account has $500 in it, and they withdraw $100 and $50. The result could be $350, $400 or $450 if no protection.
  - Security problems
    - Every user of the system should be able to access only the data they are permitted to see.
    - E.g. payroll people only handle employee records, and cannot see customer accounts; tellers only access account data and cannot see payroll data.
    - Difficult to enforce this with application programs.
  - Integrity problems
    - Data may be required to satisfy constraints.
    - E.g. no account balance below $25.00.
    - Again, difficult to enforce or to change constraints with the file-processing approach.

These problems and others led to the development of **database management systems**.

Database System Architecture:

Database-centric architecture or **data-centric architecture** has several distinct meanings, generally relating to software architectures in which databases play a crucial role. Often this description is meant to contrast the design to an alternative approach. For example, the characterization of an architecture as "database-centric" may mean any combination of the following:

- using a standard, general-purpose relational database management system, as opposed to customized in-memory or file-based data structures and access methods. With the evolution of sophisticated DBMS software, much of which is either free or included with the operating system, application developers have become increasingly reliant on standard database tools, especially for the sake of rapid application development.
- using dynamic, table-driven logic, as opposed to logic embodied in previously The use of table-driven logic, i.e. behavior that is heavily dictated by the contents of a database, allows programs to be simpler and more flexible. This capability is a central feature of dynamic programming languages.
- using stored procedures that run on database servers, as opposed to greater reliance on logic running in middle-tier application servers in a multi-tier architecture. The extent to which business logic should be placed at the back-end versus another tier is a subject of ongoing debate. For example, Oracle presents a detailed analysis of alternative architectures that vary in the placement of business logic, concluding that a database-centric approach has practical advantages from the standpoint of ease of development and maintainability.
- using a shared database as the basis for communicating between parallel processes in distributed computing applications, as opposed to direct inter-process communication via message passing functions and message-oriented middleware. A potential benefit of database-centric architecture in distributed applications is that it simplifies the design by utilizing DBMS-provided transaction processing and indexing to

achieve a high degree of reliability, performance, and capacity. For example, Base One describes a database-centric distributed computing architecture for grid and cluster computing, and explains how this design provides enhanced security, fault-tolerance, and scalability

The E-R Model:

- The entity-relationship model is based on a perception of the world as consisting of a collection of basic **objects**(entities) and **relationships** among these objects.

  - An **entity** is a distinguishable object that exists.
  - Each entity has associated with it a set of**attributes** describing it.
  - E.g. *number* and *balance* for an account entity.
  - A **relationship** is an association among several entities.
  - e.g. A *cust_acct* relationship associates a customer with each account he or she has.
  - The set of all entities or relationships of the same type is called the **entity set** or **relationship set**.
  - Another essential element of the E-R diagram is the **mapping cardinalities**, which express the number of entities to which another entity can be associated via a relationship set.

  We'll see later how well this model works to describe real world situations.

- The overall logical structure of a database can be expressed graphically by an **E-R diagram**:
  - **rectangles**: represent entity sets.
  - **ellipses**: represent attributes.
  - **diamonds**: represent relationships among entity sets.
  - **lines**: link attributes to entity sets and entity sets to relationships.

  See figure 1.2 for an example.

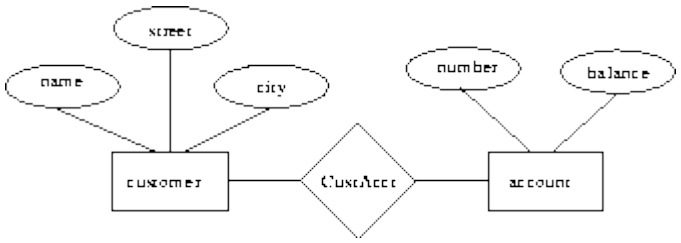


Figure 1.2: A sample E-R diagram.

Entity and Entity Set:

- An entity is an object that exists and is distinguishable from other objects. For instance, John Harris with S.I.N. 890-12-3456 is an entity, as he can be uniquely identified as one particular person in the universe.

- An entity may be concrete (a person or a book, for example) or abstract (like a holiday or a concept).

- An entity set is a set of entities of the same type (e.g., all persons having an account at a bank).

- Entity sets need not be disjoint. For example, the entity set employee (all employees of a bank) and the entity set >customer (all customers of the bank) may have members in common.

- An entity is represented by a set of attributes.
  - E.g. name, S.I.N., street, city for ``customer'' entity.
  - The >domainof the attribute is the set of permitted values (e.g. the telephone number must be seven positive integers).

- Formally, an attribute is a function which maps an entity set into a domain.
  - Every entity is described by a set of (attribute, data value) pairs.
  - There is one pair for each attribute of the entity set.
  - E.g. a particular customer entity is described by the set {(name, Harris), (S.I.N., 890-123-456), (street, North), (city, Georgetown)}.

An analogy can be made with the programming language notion of type definition.

- The concept of an entity set corresponds to the programming language type definition.
- A variable of a given type has a particular value at a point in time.

- Thus, a programming language variable corresponds to anentity in the E-R model.

  Figure 2-1 shows two entity sets.

  We will be dealing with five entity sets in this section:

- branch, the set of all branches of a particular bank. Each branch is described by the attributes branch-name, branch-city and assets.

- customer, the set of all people having an account at the bank. Attributes arecustomer-name,S.I.N., street and customer-city

- employee, with attributes employee-name and phone-number

- account, the set of all accounts created and maintained in the bank. Attributes areaccount-number and balance.

- transaction>, the set of all account transactions executed in the bank. Attributes are transaction-number, date and amount

Relationship and Relationship sets:

A **relationship** is an association between several entities.

A **relationship set** is a set of relationships of the same type.

Formally it is a mathematical relation on $n \geq 2$ (possibly non-distinct) sets.

If $E_1, E_2, \ldots, E_n$ are entity sets, then a relationship set R is a **subset**of

$$\{(e_1, e_2, \ldots, e_n) \mid e_1 \in E_1, e_2 \in E_2, \ldots, e_n \in En\}$$

where $(e_1, e_2, \ldots, e_n)$ is a relationship.

For example, consider the two entity sets *customer* and *account*. (Fig. 2.1 in the text). We define the relationship *CustAcct* to denote the association between customers and their accounts. This is a **binary**relationship set (see Figure 2.2 in the text).

Going back to our formal definition, the relationship set *CustAcct* is a subset of all the possible customer and account pairings.

This is a binary relationship. Occasionally there are relationships involving more than two entity sets.

The **role** of an entity is the function it plays in a relationship. For example, the relationship *works-for* could be ordered pairs of *employee*entities. The first employee takes the role of manager, and the second one will take the role of worker.

A relationship may also have **descriptive** attributes. For example, *date*(last date of account access) could be an attribute of the *CustAcct*relationship set.

Attributes:

It is possible to define a set of entities and the relationships among them in a number of different ways. The main difference is in how we deal with attributes.

Consider the entity set *employee* with attributes*employee-name* and *phone-number*.

- We could argue that the phone be treated as an entity itself, with attributes *phone-number* and*location*.
- Then we have two entity sets, and the relationship set *EmpPhn* defining the association between employees and their phones.
- This new definition allows employees to have several (or zero) phones.
- New definition may more accurately reflect the real world.
- We cannot extend this argument easily to making *employee-name* an entity.

The question of what constitutes an entity and what constitutes an attribute depends mainly on the structure of the real world situation being modeled, and the semantics associated with the attribute in question.

Mapping Constraint:

An E-R scheme may define certain constraints to which the contents of a database must conform.

Mapping Cardinalities: express the number of entities to which another entity can be associated via a relationship. For binary relationship sets between entity sets A and B, the mapping cardinality must be one of:

1. **One-to-one**: An entity in A is associated with at most one entity in B, and an entity in B is associated with at most one entity in A. (Figure 2.3)

2. **One-to-many**: An entity in A is associated with any number in B. An entity in B is associated with at most one entity in A. (Figure 2.4)

3. **Many-to-one**: An entity in A is associated with at most one entity in B. An entity in B is associated with any number in A. (Figure 2.5)

4. **Many-to-many**: Entities in A and B are associated with any number from each other. (Figure 2.6)

The appropriate mapping cardinality for a particular relationship set depends on the real world being modeled. (Think about the *CustAcct* relationship...)

- **Existence Dependencies:** if the existence of entity X depends on the existence of entity Y, then X is said to be **existence dependent** on Y. (Or we say that Y is the **dominant** entity and X is the **subordinate** entity.)

  For example,

  - Consider *account* and *transaction* entity sets, and a relationship *log* between them.
  - This is one-to-many from account to transaction.
  - If an *account* entity is deleted, its associated *transaction* entities must also be deleted.

- Thus *account* is dominant and*transaction* is subordinate.

Keys:

Differences between entities must be expressed in terms of attributes.

- A **superkey** is a set of one or more attributes which, taken collectively, allow us to identify uniquely an entity in the entity set.
- For example, in the entity set customer, customer-name and S.I.N. is a superkey.
- Note that customer-name alone is not, as two customers could have the same name.
- A superkey may contain extraneous attributes, and we are often interested in the smallest superkey. A superkey for which no subset is a superkey is called a **candidate key**.
- In the example above, S.I.N. is a candidate key, as it is minimal, and uniquely identifies a customer entity.
- A **primary key** is a candidate key (there may be more than one) chosen by the DB designer to identify entities in an entity set.

An entity set that does not possess sufficient attributes to form a primary key is called a **weak entity set.**One that does have a primary key is called a **strong entity set**.

For example,

- The entity set transaction has attributes >.
- Different transactions on different accounts could share the same number.
- These are not sufficient to form a primary key (uniquely identify a transaction).
- Thus transaction is a weak entity set.

For a weak entity set to be meaningful, it must be part of a one-to-many relationship set. This relationship set should have no descriptive attributes. (Why?)

The idea of strong and weak entity sets is related to the existence dependencies seen earlier.

- Member of a strong entity set is a dominant entity.
- Member of a weak entity set is a subordinate entity.

A weak entity set does not have a primary key, but we need a means of distinguishing among the entities.

The **discriminator** of a weak entity set is a set of attributes that allows this distinction to be made.

The **primary key of a weak entity set** is formed by taking the primary key of the strong entity set on which its existence depends (see Mapping Constraints) plus its **discriminator**.

To illustrate:

- transaction is a weak entity. It is existence-dependent on account.
- The primary key of account> is account-number.
- transaction-number distinguishes transaction entities within the same account (and is thus the discriminator).
- So the primary key for transaction would be (account-number, transaction-number).

Just Remember: The primary key of a weak entity is found by taking the primary key of the strong entity on which it is existence-dependent, plus the discriminator of the weak entity set.

Primary Keys For Relationship Sets:

The attributes of a relationship set are the attributes that comprise the primary keys of the entity sets involved in the relationship set.

For example:

- S.I.N. is the primary key of customer, and
- account-number is the primary key of account.
- The attributes of the relationship set custacct are then (account-number, S.I.N.).

This is enough information to enable us to relate an account to a person.

If the relationship has descriptive attributes, those are also included in its attribute set. For example, we might add the attribute date> to the above

relationship set, signifying the date of last access to an account by a particular customer.

Note that this attribute cannot instead be placed in either entity set as it relates to both a customer and an account, and the relationship is many-to-many.

The primary key of a relationship set R depends on the mapping cardinality and the presence of descriptive attributes.

With no descriptive attributes:

- **many-to-many:** all attributes in R.
- **one-to-many:** primary key for the "many" entity.

Descriptive attributes may be added, depending on the mapping cardinality and the semantics involved (see text).

The Entity Relationship Daigram:

We can express the overall logical structure of a database**graphically** with an E-R diagram.

Its components are:

- **rectangles** representing entity sets.
- **ellipses** representing attributes.
- **diamonds** representing relationship sets.
- **lines** linking attributes to entity sets and entity sets to relationship sets.

In the text, lines may be directed (have an arrow on the end) to signify mapping cardinalities for relationship sets. Figures 2.8 to 2.10 show some examples.

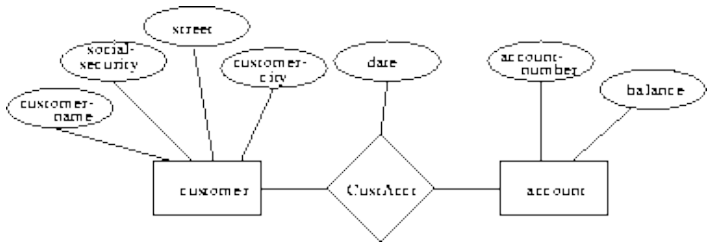


Figure 2.7: An E-R diagram

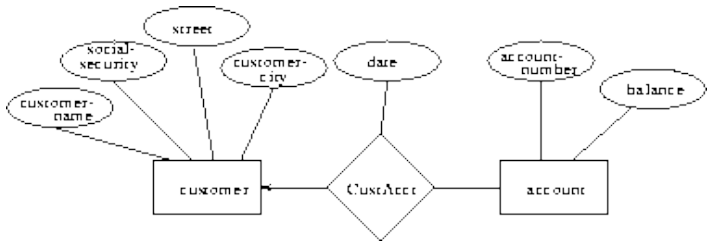


Figure 2.8: One-to-many from *customer* to *account*

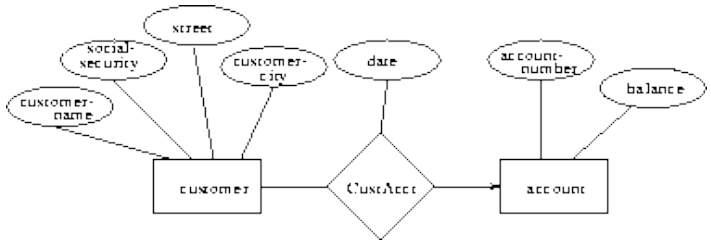


Figure 2.9: Many-to-one from *customer* to *account*

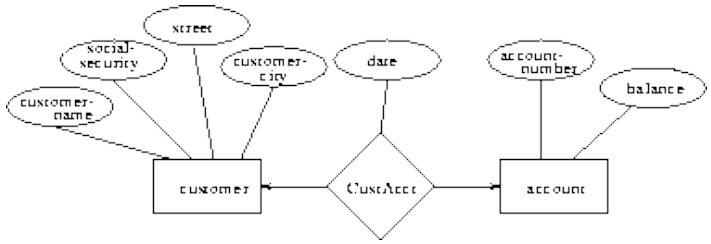


Figure 2.10: One-to-one from *customer* to *account*

Go back and review mapping cardinalities. They express the number of entities to which an entity can be associated via a relationship.

The arrow positioning is simple once you get it straight in your mind, so do some examples. Think of the arrow head as pointing to the entity that ``one'' refers to.

Other Style of E-R Diagram:

The text uses one particular style of diagram. Many variations exist.

Some of the variations you will see are:

- Diamonds being omitted - a link between entities indicates a relationship.
  - Less symbols, clearer picture.
  - What happens with descriptive attributes?
  - In this case, we have to create an**intersection entity** to possess the attributes.
- Numbers instead of arrowheads indicating cardinality.
  - Symbols, 1, n and m used.
  - E.g. 1 to 1, 1 to n, n to m.
  - Easier to understand than arrowheads.

- A range of numbers indicating **optionality** of relationship. (See Elmasri & Navathe, p 58.)
  - E.g (0,1) indicates minimum zero (optional), maximum 1.
  - Can also use (0,n), (1,1) or (1,n).
  - Typically used on near end of link - confusing at first, but gives more information.
  - E.g. entity 1 (0,1) -- (1,n) entity 2 indicates that entity 1 is related to between 0 and 1 occurrences of entity 2 (optional).
  - Entity 2 is related to at least 1 and possibly many occurrences of entity 1 (mandatory).
- **Multivalued** attributes may be indicated in some manner.
  - Means attribute can have more than one value.
  - E.g. hobbies.
  - Has to be normalized later on.
- **Extended E-R diagrams** allowing more details/constraints in the real world to be recorded. (See Elmasri & Navathe, chapter 21.)
  - Composite attributes.
  - Derived attributes.
  - Subclasses and superclasses.

- Generalization and specialization.

Roles in E-R Diagrams

The function that an entity plays in a relationship is called its **role**. Roles are normally explicit and not specified.

They are useful when the meaning of a relationship set needs clarification.

For example, the entity sets of a relationship may not be distinct. The relationship *works-for* might be ordered pairs of *employees* (first is manager, second is worker).

In the E-R diagram, this can be shown by labelling the lines connecting entities (rectangles) to relationships (diamonds). (See figure 2.11).

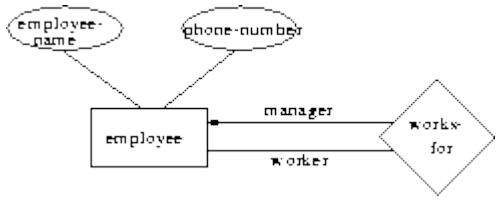


Figure 2.11: E-R diagram with role indicators

Weak Entity Sets in E-R Diagrams

A weak entity set is indicated by a doubly-outlined box. For example, the previously-mentioned weak entity set *transaction* is dependent on the strong entity set *account* via the relationship set *log*.

Figure 2.12) shows this example.

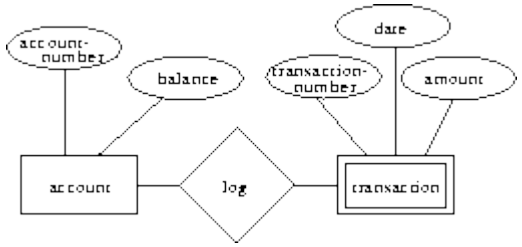


Figure 2.12: E-R diagram with a weak entity set

Nonbinary Relationships

Non-binary relationships can easily be represented. Figure 2.13) shows an example.

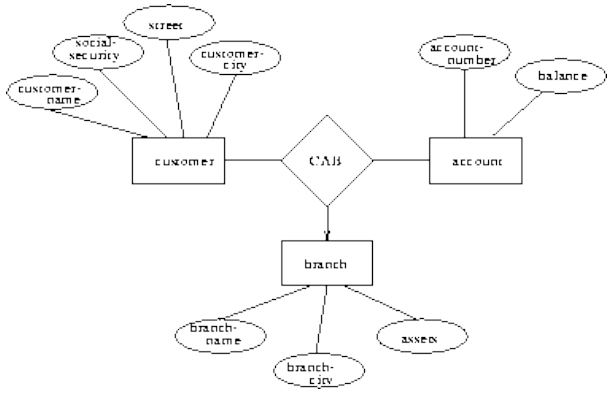


Figure 2.13: E-R diagram with a ternary relationship

This E-R diagram says that a customer may have several accounts, each located in a specific bank branch, and that an account may belong to several different customers.

Reducing E-R Diagrams to Tables:

A database conforming to an E-R diagram can be represented by a collection of tables. We'll use the E-R diagram of Figure2.14) as our example.

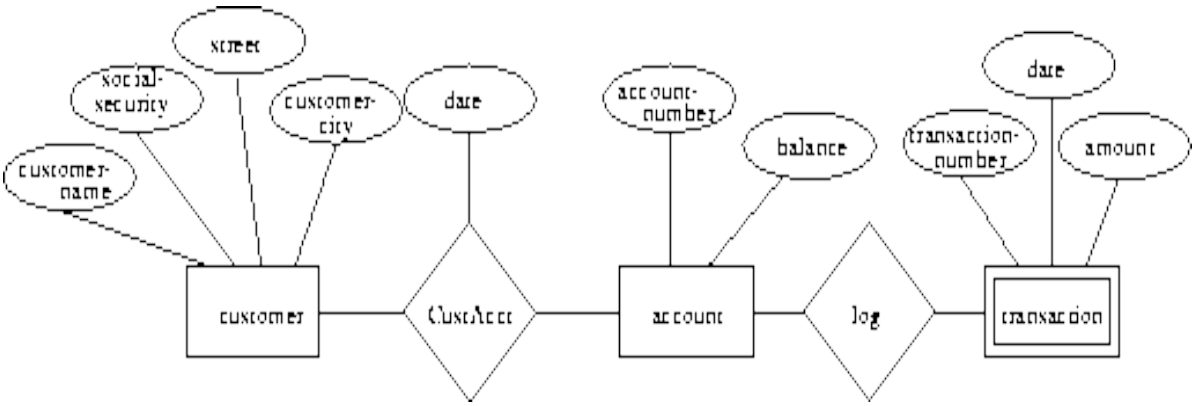


Figure 2.14: E-R diagram with strong and weak entity sets

For each entity set and relationship set, there is a **unique** table which is assigned the name of the corresponding set. Each table has a number of columns with unique names. (E.g. Figs. 2.14 - 2.18 in the text).

Representation of Strong Entity sets:

We use a table with one column for each attribute of the set. Each row in the table corresponds to one entity of the entity set. For the entity set *account*, see the table of figure 2.14.

We can add, delete and modify rows (to reflect changes in the real world).

A row of a table will consist of an n-tuple where n is the number of attributes.

Actually, the table contains a subset of the set of all possible rows. We refer to the set of all possible rows as the **cartesian product** of the sets of all attribute values.

We may denote this as

$$D_1 \times D_2 \text{ or } \times_{i=1}^{2} D_i$$

for the account table, where D_1 and D_2 denote the set of all account numbers and

all account balances, respectively.

In general, for a table of n columns, we may denote the cartesian product of

$$D_1, D_2, \ldots, D_n$$ by

$$\times_{i=1}^{n} D_i$$

Database Management System Sets:

For a weak entity set, we add columns to the table corresponding to the primary key of the strong entity set on which the weak set is dependent.

For example, the weak entity set *transaction* has three attributes: *transaction-number*, *date* and *amount*. The primary key of *account* (on which *transaction*depends) is *account-number*. This gives us the table of figure 2.16.

Representation Relationship Sets:

Let R be a relationship set involving entity sets $E_1, E_2, \ldots, E_m$.

The table corresponding to the relationship set R has the following attributes:

$$\bigcup_{i=1}^{m} \text{primary-key}(E_i)$$

If the relationship has k descriptive attributes, we add them too:

$$\bigcup_{i=1}^{m} \text{primary-key}(E_i) \bigcup \{a_1, a_2, \ldots, a_k\}$$

An example:

- The relationship set *CustAcct* involves the entity sets*customer* and *account*.
- Their respective primary keys are *S.I.N.* and *account-number*.
- *CustAcct* also has a descriptive attribute, *date*.
- This gives us the table of figure 2.17.

Non-binary Relationship Sets

The ternary relationship of Figure 2.13 gives us the table of figure 2.18. As required, we take the primary keys of each entity set. There are no descriptive attributes in this example.

Linking a Weak to a Strong Entity

These relationship sets are many-to-one, and have no descriptive attributes. The primary key of the weak entity set is the primary key of the strong entity set it is existence-dependent on, plus its discriminator.

The table for the relationship set would have the same attributes, and is thus redundant.

Concept of Enhanced E-R Diagram:

We have seen weak entity sets, generalization and aggregation. Designers must decide when these features are appropriate.

- Strong entity sets and their dependent weak entity sets may be regarded as a single ``object'' in the database, as weak entities are existence-dependent on a strong entity.
- It is possible to treat an aggregated entity set as a single unit without concern for its inner structure details.

- Generalization contributes to modularity by allowing common attributes of similar entity sets to be represented in one place in an E-R diagram.

Excessive use of the features can introduce unnecessary complexity into the design.

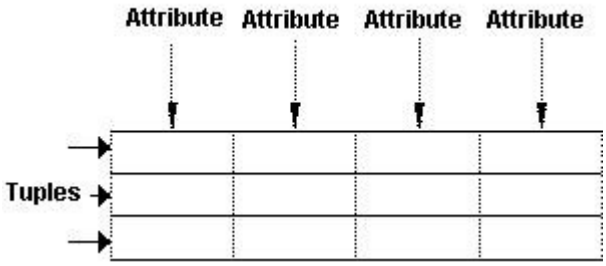
The Relational Data Model:

The Relational Data Model has the relation at its heart, but then a whole series of rules governing keys, relationships, joins, functional dependencies, transitive dependencies, multi-valued dependencies, and modification anomalies.

The Relation

The *Relation* is the basic element in a relational data model.

Figure 3 - Relations in the Relational Data Model



A relation is subject to the following rules:

1. Relation (file, table) is a two-dimensional table.
2. Attribute (i.e. field or data item) is a **column** in the table.
3. Each column in the table has a unique name within that table.

4. Each column is homogeneous. Thus the entries in any column are all of the same type (e.g. age, name, employee-number, etc).
5. Each column has a **domain**, the set of possible values that can appear in that column.
6. A Tuple (i.e. record) is a row in the table.
7. The order of the rows and columns is not important.
8. Values of a row all relate to some thing or portion of a thing.
9. Repeating groups (collections of logically related attributes that occur multiple times within one record occurrence) are not allowed.
10. Duplicate rows are not allowed (candidate keys are designed to prevent this).
11. Cells must be single-valued (but can be variable length). Single valued means the following:
    - Cannot contain multiple values such as 'A1,B2,C3'.
    - Cannot contain combined values such as 'ABC-XYZ' where 'ABC' means one thing and 'XYZ' another.

A relation may be expressed using the notation`R(A,B,C, ...)` where:

- R = the name of the relation.
- (A,B,C, ...) = the attributes within the relation.
- A = the attribute(s) which form the primary key.

Keys

1. A **simple** key contains a single attribute.
2. A **composite key** is a key that contains more than one attribute.
3. A **candidate key** is an attribute (or set of attributes) that uniquely identifies a row. A candidate key must possess the following properties:
    - Unique identification - For every row the value of the key must uniquely identify that row.
    - Non redundancy - No attribute in the key can be discarded without destroying the property of unique identification.
4. A **primary key** is the candidate key which is selected as the principal unique identifier. Every relation must contain a primary key. The primary key is usually the key selected to identify a row when the database is physically implemented. For example, a part number is selected instead of a part description.
5. A **superkey** is any set of attributes that uniquely identifies a row. A superkey differs from a candidate key in that it does not require the non redundancy property.
6. A **foreign key** is an attribute (or set of attributes) that appears (usually) as a non key attribute in one relation and as a primary key attribute in another relation. I say *usually* because it is possible for a foreign key to also be the whole or part of a primary key:
    - A many-to-many relationship can only be implemented by introducing an intersection or link table which then becomes the child in two one-to-many relationships. The intersection table therefore has a foreign key for each of its parents, and its primary key is a composite of both foreign keys.
    - A one-to-one relationship requires that the child table has no more than one occurrence for each parent, which can only be enforced by letting the foreign key also serve as the primary key.
7. A **semantic** or **natural** key is a key for which the possible values have an obvious meaning to the user or the data. For example, a semantic primary key for a COUNTRY entity might contain the

value 'USA' for the occurrence describing the United States of America. The value 'USA' has meaning to the user.

8. A **technical** or **surrogate** or **artificial** key is a key for which the possible values have no obvious meaning to the user or the data. These are used instead of semantic keys for any of the following reasons:
    - When the value in a semantic key is likely to be changed by the user, or can have duplicates. For example, on a PERSON table it is unwise to use PERSON_NAME as the key as it is possible to have more than one person with the same name, or the name may change such as through marriage.
    - When none of the existing attributes can be used to guarantee uniqueness. In this case adding an attribute whose value is generated by the system, e.g from a sequence of numbers, is the only way to provide a unique value. Typical examples would be ORDER_ID and INVOICE_ID. The value '12345' has no meaning to the user as it conveys nothing about the entity to which it relates.
9. A key functionally determines the other attributes in the row, thus it is always a determinant.
10. Note that the term 'key' in most DBMS engines is implemented as an index which does not allow duplicate entries.

Relationships:

One table (relation) may be linked with another in what is known as a **relationship**. Relationships may be built into the database structure to facilitate the operation of relational joins at runtime.

1. A relationship is between two tables in what is known as a **one-to-many** or **parent-child** or**master-detail** relationship where an occurrence on the 'one' or 'parent' or 'master' table may have any number of associated occurrences on the 'many' or 'child' or 'detail' table. To achieve this the **child** table must contain fields which link back the **primary key**on the **parent** table. These fields on the **child**table are known as a **foreign key**, and the**parent** table is referred to as the **foreign**table (from the viewpoint of the child).
2. It is possible for a record on the **parent** table to exist without corresponding records on the**child** table, but it should not be possible for an entry on the **child** table to exist without a corresponding entry on the **parent** table.
3. A **child** record without a corresponding **parent**record is known as an **orphan**.
    - It is possible for a table to be related to itself. For this to be possible it needs a**foreign key** which points back to the**primary key**. Note that these two keys cannot be comprised of exactly the same fields otherwise the record could only ever point to itself.
    - A **table** may be the subject of any number of relationships, and it may be the **parent** in some and the **child** in others.
    - Some database engines allow a **parent**table to be linked via a **candidate key**, but if this were changed it could result in the link to the **child** table being broken.
    - Some database engines allow relationships to be managed by rules known as **referential integrity** or**foreign key restraints**. These will prevent entries on **child** tables from being created if the **foreign key** does not exist on the **parent** table, or will deal with entries on **child** tables when the entry on the **parent** table is updated or deleted.

Relational Joins

The join operator is used to combine data from two or more relations (tables) in order to satisfy a particular query. Two relations may be joined when they share at least one common attribute. The join is implemented by considering each row in an instance of each relation. A row in relation R1 is joined to a row in relation R2 when the value of the common attribute(s) is equal in the two relations. The join of two relations is often called a **binary join.**

The join of two relations creates a new relation. The notation 'R1 x R2' indicates the join of relations R1 and R2. For example, consider the following:

Relation R1		
A	**B**	**C**
1	5	3
2	4	5
8	3	5
9	3	3
1	6	5
5	4	3
2	7	5

Relation R2		
B	**D**	**E**
4	7	4
6	2	3
5	7	8
7	2	3
3	2	2

Note that the instances of relation R1 and R2 contain the same data values for attribute B. Data normalisation is concerned with decomposing a relation (e.g. R(A,B,C,D,E) into smaller relations (e.g. R1 and R2). The data values for attribute B in this context will be identical in R1 and R2. The instances of R1 and R2 are projections of the instances of R(A,B,C,D,E) onto the attributes (A,B,C) and (B,D,E) respectively. A projection will not eliminate data values - duplicate rows are removed, but this will not remove a data value from any attribute.

The join of relations R1 and R2 is possible because B is a common attribute. The result of the join is:

Relation R1 x R2				
A	**B**	**C**	**D**	**E**

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

The row (2 4 5 7 4) was formed by joining the row (2 4 5) from relation R1 to the row (4 7 4) from relation R2. The two rows were joined since each contained the same value for the common attribute B. The row (2 4 5) was not joined to the row (6 2 3) since the values of the common attribute (4 and 6) are not the same.

Lossless Joins:

A set of relations satisfies the lossless join property if the instances can be joined without creating invalid data (i.e. new rows). The term lossless join may be somewhat confusing. A join that is not lossless will contain extra, invalid rows. A join that is lossless will not contain extra, invalid rows. Thus the term **gainless join** might be more appropriate.

To give an example of incorrect information created by an invalid join let us take the following data structure:

`R(student, course, instructor, hour, room, grade)`

Assuming that only one section of a class is offered during a semester we can define the following functional dependencies:

1. (HOUR, ROOM) $\rightarrow$ COURSE
2. (COURSE, STUDENT) $\rightarrow$ GRADE
3. (INSTRUCTOR, HOUR) $\rightarrow$ ROOM
4. (COURSE) $\rightarrow$ INSTRUCTOR
5. (HOUR, STUDENT) $\rightarrow$ ROOM

Take the following sample data:

STUDENT	COURSE	INSTRUCTOR	HOUR	ROOM	GRADE
Smith	Math 1	Jenkins	8:00	100	A
Jones	English	Goldman	8:00	200	B
Brown	English	Goldman	8:00	200	C

Green	Algebra	Jenkins	9:00	400	A

The following four relations, each in 4th normal form, can be generated from the given and implied dependencies:

- R1(STUDENT, HOUR, COURSE)
- R2(STUDENT, COURSE, GRADE)
- R3(COURSE, INSTRUCTOR)
- R4(INSTRUCTOR, HOUR, ROOM)

Note that the dependencies (HOUR, ROOM) $\longrightarrow$ COURSE and (HOUR, STUDENT) $\longrightarrow$ ROOM are not

explicitly represented in the preceding decomposition. The goal is to develop relations in 4th normal form that can be joined to answer any ad hoc inquiries correctly. This goal can be achieved without representing every functional dependency as a relation. Furthermore, several sets of relations may satisfy the goal.

Determinant and Dependant:

The terms determinant and dependent can be described as follows:

1. The expression X $\longrightarrow$ Y means 'if I know the value of X, then I can obtain the value of Y' (in a table

   or somewhere).
2. In the expression X $\longrightarrow$ Y, X is the**determinant** and Y is the **dependent**attribute.

3. The value X **determines** the value of Y.
4. The value Y **depends on** the value of X.

Functional Dependencies (FD)
A functional dependency can be described as follows:

1. An attribute is functionally dependent if its value is determined by another attribute.
2. That is, if we know the value of one (or several) data items, then we can find the value of another (or several).
3. Functional dependencies are expressed as X $\longrightarrow$ Y, where X is the determinant and Y is the

   functionally dependent attribute.
4. If A $\longrightarrow$ (B,C) then A $\longrightarrow$ B and A $\longrightarrow$ C.

5. If (A,B) $\longrightarrow$ C, then it is not necessarily true that A $\longrightarrow$ C and B $\longrightarrow$ C.

6. If A $\longrightarrow$ B and B $\longrightarrow$ A, then A and B are in a 1-1 relationship.

7. If A $\longrightarrow$ B then for A there can only ever be one value for B.

Transitive Dependencies (TD)
A transitive dependency can be described as follows:

1. An attribute is transitively dependent if its value is determined by another attribute *which is not a key*.
2. If X $\rightarrow$ Y and X is not a key then this is a transitive dependency.
3. A transitive dependency exists when A $\rightarrow$ B $\rightarrow$ C but NOT A $\rightarrow$ C.

Multi-Valued Dependencies (MVD)
A multi-valued dependency can be described as follows:

1. A table involves a multi-valued dependency if it may contain multiple values for an entity.
2. A multi-valued dependency may arise as a result of enforcing 1st normal form.
3. X $\twoheadrightarrow$ Y, ie X multi-determines Y, when for each value of X we can have more than one value of Y.
4. If A $\twoheadrightarrow$ B and A $\twoheadrightarrow$ C then we have a single attribute A which multi-determines two other independent attributes, B and C.
5. If A $\twoheadrightarrow$ (B,C) then we have an attribute A which multi-determines a set of associated attributes, B and C.

Join Dependencies (JD)
A join dependency can be described as follows:

1. If a table can be decomposed into three or more smaller tables, it must be capable of being joined again on common keys to form the original table.

Modification Anomalies
A major objective of data normalisation is to avoid modification anomalies. These come in two flavours:

1. An **insertion anomaly** is a failure to place information about a new database entry into all the places in the database where information about that new entry needs to be stored. In a properly normalized database, information about a new entry needs to be inserted into only one place in the database. In an inadequately normalized database, information about a new entry may need to be inserted into more than one place, and, human fallibility being what it is, some of the needed additional insertions may be missed.
2. A **deletion anomaly** is a failure to remove information about an existing database entry when it is time to remove that entry. In a properly normalized database, information about an old, to-be-gotten-rid-of entry needs to be deleted from only one place in the database. In an inadequately normalized database, information about that old entry may need to be deleted from more than one place, and, human fallibility being what it is, some of the needed additional deletions may be missed.

An update of a database involves modifications that may be additions, deletions, or both. Thus 'update anomalies' can be either of the kinds of anomalies discussed above.

All three kinds of anomalies are highly undesirable, since their occurrence constitutes corruption of the database. Properly normalised databases are much less susceptible to corruption than are unnormalised databases.

Types of Relational Join:

A JOIN is a method of creating a result set that combines rows from two or more tables (relations). When comparing the contents of two tables the following conditions may occur:

- Every row in one relation has a match in the other relation.
- Relation R1 contains rows that have no match in relation R2.
- Relation R2 contains rows that have no match in relation R1.

INNER joins contain only matches. OUTER joins may contain mismatches as well.

Inner Join
This is sometimes known as a **simple** join. It returns all rows from both tables where there is a match. If there are rows in R1 which do not have matches in R2, those rows will **not** be listed. There are two possible ways of specifying this type of join:

SELECT * FROM R1, R2 WHERE R1.r1_field = R2.r2_field; SELECT * FROM R1 INNER JOIN R2 ON R1.field = R2.r2_field

If the fields to be matched have the same names in both tables then the `ON` condition, as in:

ON R1.fieldname = R2.fieldname ON (R1.field1 = R2.field1 AND R1.field2 = R2.field2)

can be replaced by the shorter `USING` condition, as in:

USING fieldname USING (field1, field2)

Natural Join
A natural join is based on all columns in the two tables that have the same name. It is semantically equivalent to an INNER JOIN or a LEFT JOIN with a `USING` clause that names all columns that exist in both tables.

SELECT * FROM R1 NATURAL JOIN R2

The alternative is a **keyed** join which includes an `ON` or `USING`condition.

Left [Outer] Join
Returns all the rows from R1 even if there are no matches in R2. If there are no matches in R2 then the R2 values will be shown as null.

SELECT * FROM R1 LEFT [OUTER] JOIN R2 ON R1.field = R2.field

Right [Outer] Join
Returns all the rows from R2 even if there are no matches in R1. If there are no matches in R1 then the R1 values will be shown as null.

SELECT * FROM R1 RIGHT [OUTER] JOIN R2 ON R1.field = R2.field

Full [Outer] Join
Returns all the rows from both tables even if there are no matches in one of the tables. If there are no matches in one of the tables then its values will be shown as null.

SELECT * FROM R1 FULL [OUTER] JOIN R2 ON R1.field = R2.field

Self Join
This joins a table to itself. This table appears twice in the FROM clause and is followed by table aliases that qualify column names in the join condition.

SELECT a.field1, b.field2 FROM R1 a, R1 b WHERE a.field = b.field

Cross Join
This type of join is rarely used as it does not have a join condition, so every row of R1 is joined to every row of R2. For example, if both tables contain 100 rows the result will be 10,000 rows. This is sometimes known as a **cartesian product**and can be specified in either one of the following ways:

SELECT * FROM R1 CROSS JOIN R2 SELECT * FROM R1, R2

The Relational Algebra:

The **relational algebra** is a procedural query language.

1. Six fundamental operations:
   - select (unary)
   - project (unary)
   - rename (unary)
   - cartesian product (binary)
   - union (binary)
   - set-difference (binary)
2. Several other operations, defined in terms of the fundamental operations:
   - set-intersection
   - natural join
   - division
   - assignment
3. Operations produce a new relation as a result.

Fundamental Operation:

- **The Select Operation**

  Select selects tuples that satisfy a given predicate. Select is denoted by a lowercase Greek sigma (σ), with the predicate appearing as a subscript. The argument relation is given in parentheses

  following the σ.

  For example, to select tuples (rows) of the borrow relation where the branch is "SFU", we would write

  $$\sigma_{bname = "SFU"}(borrow)$$

  Let Figure 3.3 be the borrow and branch relations in the banking example.

The new relation created as the result of this operation consists of one tuple: $(SFU, 15, Hayes, 1500)$.

We allow comparisons using =, $\neq$, <, $\leq$, > and $\geq$ in the selection predicate.

We also allow the logical connectives $\vee$ (or) and $\wedge$ (and). For example:

$$\sigma_{bname = \text{"Downtown"} \wedge amount > 1200}(borrow)$$

cname	banker
Hayes	Jones
Johnson	Johnson

Figure 3.4: The client relation.

Suppose there is one more relation, client, shown in Figure 3.4, with the scheme

$$Client\_scheme = (cname, banker)$$

we might write

$$\sigma_{cname = banker}(client)$$

to find clients who have the same name as their banker.

☐ **The Project Operation**

Project copies its argument relation for the specified attributes only. Since a relation is a **set**, duplicate rows are eliminated.

Projection is denoted by the Greek capital letter pi (Π). The attributes to be copied appear as subscripts.

For example, to obtain a relation showing customers and branches, but ignoring amount and loan#, we write

$$\Pi_{bname, cname}(borrow)$$

We can perform these operations on the relations resulting from other operations.

To get the names of customers having the same name as their bankers,

$$\Pi_{cname}(\sigma_{cname = banker}(client))$$

Think of **select** as taking rows of a relation, and **project** as taking columns of a relation.

☐ **The Cartesian Product Operation**

The **cartesian product** of two relations is denoted by a cross ($\times$), written

$$r_1 \times r_2 \text{ for relations } r_1 \text{ and } r_2$$

The result of $r_1 \times r_2$ is a new relation with a tuple for each possible**pairing** of tuples from r_1 and r_2.

In order to avoid ambiguity, the attribute names have attached to them the name of the relation from which they came. If no ambiguity will result, we drop the relation name.

The result $client \times customer$ is a very large relation. If r_1 has n_1 tuples, and r_2 has n_2 tuples,

then $r = r_1 \times r_2$ will have $n_1 n_2$ tuples.

The resulting scheme is the concatenation of the schemes of r_1 and r_2, with relation names added as

mentioned.

To find the clients of banker Johnson and the city in which they live, we need information in both client and customer relations. We can get this by writing

$$\sigma_{banker="Johnson"}(client \times customer)$$

However, the stomer.cname column contains customers of bankers other than Johnson. (Why?)

We want rows whereclient.cname = customer.cname. So we can write

$$\sigma_{client.cname=customer.cname}(\sigma_{banker="Johnson"}(client \times customer))$$

to get just these tuples.

Finally, to get just the customer's name and city, we need a projection:

$$\Pi_{client.cname,ccity}\ (\sigma_{client.cname=customer.cname}\ (\sigma_{banker="Johnson"}(client \times customer)))$$

- **The Select Operation**

  Select selects tuples that satisfy a given predicate. Select is denoted by a lowercase Greek sigma (σ), with the predicate appearing as a subscript. The argument relation is given in parentheses

  following the σ.

  For example, to select tuples (rows) of the *borrow* relation where the branch is ``SFU'', we would write

  $$\sigma_{bname="SFU"}(borrow)$$

  Let Figure 3.3 be the *borrow* and *branch* relations in the banking example.

bname	loan#	cname	amount
Downtown	17	Jones	1000
Lougheed_Mall	23	Smith	2000
SFU	15	Hayes	1500

bname	assets	heity
Downtown	9,000,000	Vancouver
Lougheed_Mall	21,000,000	Burnaby
SFU	17,000,000	Burnaby

  Figure 3.3: The *borrow* and *branch* relations.

  The new relation created as the result of this operation consists of one tuple:

  $$(SFU, 15, Hayes, 1500)$$

  We allow comparisons using =, $\neq$, <, $\leq$, > and $\geq$ in the selection predicate.

  We also allow the logical connectives $\vee$ (or) and $\wedge$ (and). For example:

  $$\sigma_{bname="Downtown" \wedge amount > 1200}(borrow)$$

cname	banker
Hayes	Jones
Johnson	Johnson

Figure 3.4: The *client* relation.

Suppose there is one more relation, *client*, shown in Figure 3.4, with the scheme

$$Client\_scheme = (cname, banker)$$

we might write

$$\sigma_{cname=banker}(client)$$

to find clients who have the same name as their banker.

☐ **The Project Operation**

Project copies its argument relation for the specified attributes only. Since a relation is a **set**, duplicate rows are eliminated.

Projection is denoted by the Greek capital letter pi (Π). The attributes to be copied appear as subscripts.

For example, to obtain a relation showing customers and branches, but ignoring amount and loan#, we write

$$\Pi_{bname,cname}(borrow)$$

We can perform these operations on the relations resulting from other operations.

To get the names of customers having the same name as their bankers,

$$\Pi_{cname}(\sigma_{cname=banker}(client))$$

Think of **select** as taking rows of a relation, and **project** as taking columns of a relation.

☐ **The Cartesian Product Operation**

The **cartesian product** of two relations is denoted by a cross ($\times$), written

$$r_1 \times r_2 \text{ for relations } r_1 \text{ and } r_2$$

The result of $r_1 \times r_2$ is a new relation with a tuple for each possible **pairing** of tuples from r_1 and r_2.

In order to avoid ambiguity, the attribute names have attached to them the name of the relation from which they came. If no ambiguity will result, we drop the relation name.

The result $client \times customer$ is a very large relation. If r_1 has n_1 tuples, and r_2 has n_2 tuples,

then $r = r_1 \times r_2$ will have $n_1 n_2$ tuples.

The resulting scheme is the concatenation of the schemes of r_1 and r_2, with relation names added as

mentioned.

To find the clients of banker Johnson and the city in which they live, we need information in both *client* and *customer* relations. We can get this by writing

$$\sigma_{banker="Johnson"}(client \times customer)$$

However, the *customer.cname* column contains customers of bankers other than Johnson. (Why?)
We want rows where *client.cname* = *customer.cname*. So we can write

$$\sigma_{client.cname=customer.cname}(\sigma_{banker="Johnson"}(client \times customer))$$

to get just these tuples.
Finally, to get just the customer's name and city, we need a projection:

$$\Pi_{client.cname,city}\ (\sigma_{client.cname=customer.cname}\ (\sigma_{banker="Johnson"}(client \times customer)))$$

Formal Definition of Relational Algebra:

- A basic expression consists of either
  - A relation in the database.
  - A constant relation.
- General expressions are formed out of smaller subexpressions using
  - $\sigma_p(E_1)$ select (p a predicate)
  - $\Pi_s(E_1)$ project (s a list of attributes)
  - $\rho_x(E_1)$ rename (x a relation name)
  - $E_1 \cup E_2$ union
  - $E_1 - E_2$ set difference
  - $E_1 \times E_2$ cartesian product

The Tuple Relational Calculas:

- The tuple relational calculus is a nonprocedural language. (The relational algebra was procedural.) We must provide a formal description of the information desired.
- A query in the tuple relational calculus is expressed as

$$\{t \mid P(t)\}$$

i.e. the set of tuples t for which predicate P is true.

☐ We also use the notation

- $t[a]$ to indicate the value of tuple t on attribute a.
- $t \in r$ to show that tuple t is in relation r.

Relational Database Design:

One of the best ways to understand database design is to start with an all-in-one, flat-file table design and then toss in some sample data to see what happens. By analysing the sample data, you'll be able to identify problems caused by the initial design. You can then modify the design to eliminate the problems, test some more sample data, check for problems, and re-modify, continuing this process until you have a consistent and problem-free design.

Once you grow accustomed to the types of problems poor table design can create, hopefully you'll be able to skip the interim steps and jump immediately to the final table design.

A Simple Design Process:

Let's step through a sample database design process.

We'll design a database to keep track of students' sports activities. We'll track each activity a student takes and the fee per semester to do that activity.

Step 1: Create an Activities table containing all the fields: student's name, activity and cost. Because some students take more than one activity, we'll make allowances for that and include a second activity and cost field. So our structure will be: Student, Activity 1, Cost 1, Activity 2, Cost 2

Step 2: Test the table with some sample data. When you create sample data, you should see what your table *lets you get away with.* For instance, nothing prevents us from entering the same name for different students, or different fees for the same activity, so do so. You should also imagine trying to ask questions about your data and getting answers back (essentially querying the data and producing reports). For example, how do I find all the students taking tennis?

Activities Table

Student	Activity1	Cost1	Activity2	Cost2
John Smith	Tennis	$36	Swimming	$17
Jane Bloggs	Squash	$40	Swimming	$17
John Smith	Tennis	$36		
Mark Antony	Swimming	$15	Golf	$47

Step 3: Analyse the data. In this case, we can see a glaring problem in the first field. We have two John Smiths, and there's no way to tell them apart. We need to find a way to identify each student uniquely.

Uniquely identify records

Let's fix the glaring problem first, then examine the new results.

Step 4: Modify the design. We can identify each student uniquely by giving each one a unique ID, a new field that we add, called ID. We scrap the Student field and substitute an ID field. Note the asterisk (*) beside this field in the table below: it signals that the ID field is a *key field*, containing a unique value in each record. We can use that field to retrieve any specific record. When you create such a key field in a database program, the program will then prevent you from entering duplicate values in this field, safeguarding the uniqueness of each entry.

Our table structure is now: ID, Activity 1, Cost 1, Activity 2, Cost 2

While it's easy for the computer to keep track of ID codes, it's not so useful for humans. So we're going to introduce a second table that lists each ID and the student it belongs to. Using a database program, we can create both table structures and then link them by the common field, ID. We've now turned our initial *flat-file* design into a *relational database*: a database containing multiple tables linked together by key fields. If you were using a database program that can't handle relational databases, you'd basically be stuck with our first design and all its attendant problems. With a relational database program, you can create as many tables as your data structure requires.

The Students table would normally contain each student's first name, last name, address, age and other details, as well as the assigned ID. To keep things simple, we'll restrict it to name and ID, and focus on the Activities table structure.

Step 5: Test the table with sample data.

Students Table

Student	ID*
John Smith	084
Jane Bloggs	100
John Smith	182
Mark Antony	219

Activities Table

ID*	Activity1	Cost1	Activity2	Cost2
084	Tennis	$36	Swimming	$17
100	Squash	$40	Swimming	$17
182	Tennis	$36		
219	Swimming	$15	Golf	$47

Step 6: Analyse the data. There's still a lot wrong with the Activities table:

1. Wasted space. Some students don't take a second activity, and so we're wasting space when we store the data. It doesn't seem much of a bother in this sample, but what if we're dealing with thousands of records?
2. Addition anomalies. What if #219 (we can look him up and find it's Mark Antony) wants to do a third activity? School rules allow it, but there's no space in this structure for another activity. We can't add another record for Mark, as that would violate the unique key field ID, and it would also make it difficult to see all his information at once.

3. Redundant data entry. If the tennis fees go up to $39, we have to go through *every* record containing tennis and modify the cost.
4. Querying difficulties. It's difficult to find all people doing swimming: we have to search through Activity 1*and* Activity 2 to make sure we catch them all.
5. Redundant information. If 50 students take swimming, we have to type in both the activity and its cost each time.
6. Inconsistent data. Notice that there are conflicting prices for swimming? Should it be $15 or $17? This happens when one record is updated and another isn't.

Eliminate recurring fields

The Students table is fine, so we'll keep it. But there's so much wrong with the Activities table let's try to fix it in stages.

Step 7: Modify the design. We can fix the first four problems by creating a separate record for each activity a student takes, instead of one record for all the activities a student takes.

First we eliminate the Activity 2 and Cost 2 fields. Then we need to adjust the table structure so we can enter multiple records for each student. To do that, we redefine the key so that it consists of two fields, ID and Activity. As each student can only take an activity once, this combination gives us a unique key for each record.

Our Activities table has now been simplified to: ID, Activity, Cost. Note how the new structure lets students take any number of activities – they're no longer limited to two.

Step 8: Test sample data.

Students Table

Student	ID*
John Smith	084
Jane Bloggs	100
John Smith	182
Mark Antony	219

Activities Table

ID*	Activity*	Cost
084	Swimming	$17
084	Tennis	$36
100	Squash	$40
100	Swimming	$17
182	Tennis	$36
219	Golf	$47
219	Swimming	$15
219	Squash	$40

Step 9: Analyse the data. We know we still have the problems with redundant data (activity fees repeated) and inconsistent data (what's the correct fee for swimming?). We need to fix these things, which are both problems with editing or modifying records.

Introduction to SQL:

SQL is a standard computer language for accessing and manipulating databases.

☐ SQL stands for **S**tructured **Q**uery **L**anguage

☐ SQL allows you to access a database

☐ SQL is an ANSI standard computer language

☐ SQL can execute queries against a database

☐ SQL can retrieve data from a database

☐ SQL can insert new records in a database

☐ SQL can delete records from a database

☐ SQL can update records in a database

SQL is a Standard - BUT....

SQL is an ANSI (American National Standards Institute) standard computer language for accessing and manipulating database systems. SQL statements are used to retrieve and update data in a database. SQL works with database programs like MS Access, DB2, Informix, MS SQL Server, Oracle, Sybase, etc.

Unfortunately, there are many different versions of the SQL language, but to be in compliance with the ANSI standard, they must support the same major keywords in a similar manner (such as SELECT, UPDATE, DELETE, INSERT, WHERE, and others).

Note: Most of the SQL database programs also have their own proprietary extensions in addition to the SQL standard!

Basics of The Select Statement:

In a relational database, data is stored in tables. An example table would relate Social Security Number, Name, and Address:

Employee Address Table					
SSN	**First Name**	**Last Name**	**Address**	**City**	**State**
512687458	Joe	Smith	83 First Street	Howard	Ohio
758420012	Mary	Scott	842 Vine Ave.	Losanti ville	Ohio
102254896	Sam	Jones	33 Elm St.	Paris	New York
876512563	Sarah	Ackerman	440 U.S. 110	Upton	Michigan

Now, let's say you want to see the address of each employee. Use the SELECT statement, like so:

```
SELECT FirstName, LastName, Address, City, State
FROM EmployeeAddressTable;
```

The following is the results of your *query* of the database:

First Name	**Last Name**	**Address**	**City**	**State**
Joe	Smith	83 First Street	Howard	Ohio
Mary	Scott	842 Vine Ave.	Losanti ville	Ohio
Sam	Jones	33 Elm St.	Paris	New York

Sarah	Ackerman	440 U.S. 110	Upton	Michigan

To explain what you just did, you asked for the all of data in the EmployeeAddressTable, and specifically, you asked for the*columns* called FirstName, LastName, Address, City, and State. Note that column names and table names do not have spaces...they must be typed as one word; and that the statement ends with a semicolon (;). The general form for a SELECT statement, retrieving all of the *rows* in the table is:

```
SELECT ColumnName, ColumnName, ...
                                FROM TableName;
```

To get all columns of a table without typing all column names, use:

```
SELECT *
                                FROM TableName;
```

Each database management system (DBMS) and database software has different methods for logging in to the database and entering SQL commands; see the local computer "guru" to help you get onto the system, so that you can use SQL.

Conditional Selection

To further discuss the SELECT statement, let's look at a new example table (for hypothetical purposes only):

EmployeeStatisticsTable			
Employee IDNo	**Salary**	**Benefits**	**Position**
010	75000	15000	Manager
105	65000	15000	Manager
152	60000	15000	Manager
215	60000	12500	Manager
244	50000	12000	Staff
300	45000	10000	Staff
335	40000	10000	Staff
400	32000	7500	Entry- Level
441	28000	7500	Entry- Level

Logical Operators

There are six logical operators in SQL, and after introducing them, we'll see how they're used:

=	Equal

<> or != (see manual)	Not Equal
<	Less Than
>	Greater Than
<=	Less Than or Equal To
>=	Greater Than or Equal To

The *WHERE* clause is used to specify that only certain rows of the table are displayed, based on the criteria described in that*WHERE clause*. It is most easily understood by looking at a couple of examples.

If you wanted to see the EMPLOYEEIDNO's of those making at or over $50,000, use the following:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE SALARY >= 50000;
```

Notice that the >= (greater than or equal to) sign is used, as we wanted to see those who made greater than $50,000, or equal to $50,000, listed together. This displays:

```
EMPLOYEEIDNO
------------
010
105
152
215
244
```

The WHERE> description, SALARY >= 50000, is known as a condition The same can be done for text columns:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE POSITION = 'Manager'>
```

This displays the ID Numbers of all Managers. Generally, with text columns, stick to equal to or not equal to conditions, and make sure that any text that appears in the statement is surrounded by single quotes (').

More Complex Conditions: Compound Conditions

The AND operator joins two or more conditions, and displays a row only if that row's data satisfies **ALL**conditions listed (i.e. all conditions hold true). For example, to display all staff making over $40,000, use:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE SALARY > 40000 AND POSITION = 'Staff';
```

The OR operator joins two or more conditions, but returns a row if **ANY** of the conditions listed hold true. To see all those who make less than $40,000 or have less than $10,000 in benefits, listed together, use the following query:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE SALARY < 40000 OR BENEFITS < 10000;
```

AND & OR can be combined, for example:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE POSITION = 'Manager' AND SALARY > 60000 OR BENEFITS > 12000;
```

First, SQL finds the rows where the salary is greater than $60,000 or the benefits is greater than $12,000, then taking this new list of rows, SQL then sees if any of these rows satisfies the condition that the Position column if equal to 'Manager'. Subsequently, SQL only displays this second new list of rows, as the AND operator forces SQL to only display such rows satisfying the Position column condition. Also note that the OR operation is done first.

To generalize this process, SQL performs the OR operation(s) to determine the rows where the OR operation(s) hold true (remember: any one of the conditions is true), then these results are used to compare with the AND conditions, and only display those remaining rows where the conditions joined by the AND operator hold true.

To perform AND's before OR's, like if you wanted to see a list of managers or anyone making a large salary (>$50,000) and a large benefit package (>$10,000), whether he or she is or is not a manager, use parentheses:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE POSITION = 'Manager' OR (SALARY > 50000 AND BENEFIT > 10000);
```

IN & BETWEEN

An easier method of using compound conditions uses IN or BETWEEN. For example, if you wanted to list all managers and staff:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE POSITION IN ('Manager', 'Staff');
```

or to list those making greater than or equal to $30,000, but less than or equal to $50,000, use:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE SALARY BETWEEN 30000 AND 50000;
```

To list everyone not in this range, try:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE SALARY NOT BETWEEN 30000 AND 50000;
```

Similarly, NOT IN lists all rows excluded from the IN list.

Using LIKE

Look at the EmployeeStatisticsTable, and say you wanted to see all people whose last names started with "L"; try:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEEADDRESSTABLE
WHERE LASTNAME LIKE 'L%';
```

The percent sign (%) is used to represent any possible character (number, letter, or punctuation) or set of characters that might appear after the "L". To find those people with LastName's ending in "L", use '%L', or if you wanted the "L" in the middle of the word, try '%L%'. The '%' can be used for any characters, in that relative position to the given characters. NOT LIKE displays rows not fitting the given description. Other possiblities of using LIKE, or any of these discussed conditionals, are available, though it depends on what DBMS you are using; as usual, consult a manual or your system manager or administrator for the available features on your system, or just to make sure that what you are trying to do is available and allowed. This

disclaimer holds for the features of SQL that will be discussed below. This section is just to give you an idea of the possibilities of queries that can be written in SQL.

Joins:

In this section, we will only discuss *inner* joins, and*equijoins*, as in general, they are the most useful. For more information, try the SQL links at the bottom of the page.

Good database design suggests that each table lists data only about a single *entity*, and detailed information can be obtained in a relational database, by using additional tables, and by using a *join*.

First, take a look at these example tables:

AntiqueOwners

OwnerID	OwnerLastName	OwnerFirstName
01	Jones	Bill
02	Smith	Bob
15	Lawson	Patricia
21	Akins	Jane
50	Fowler	Sam

Orders

OwnerID	ItemDesired
02	Table
02	Desk
21	Chair
15	Mirror

Antiques

SellerID	BuyerID	Item
01	50	Bed
02	15	Table
15	02	Chair
21	50	Mirror
50	01	Desk
01	21	Cabinet
02	21	Coffee Table
15	50	Chair
01	15	Jewelry Box
02	21	Pottery
21	02	Bookcase
50	01	Plant Stand

Keys:

First, let's discuss the concept of keys>. A primary key is a column or set of columns that uniquely idenifies the rest of the data in any given row. For example, in the AntiqueOwners table, the OwnerID column uniquely identifies that row. This means two things: no two rows can have the same OwnerID, and, even if two owners have the same first and last names, the OwnerID column ensures that the two owners will not be confused with each other, because the unique OwnerID column will be used throughout the database to track the owners, rather than the names.

A foreign key is a column in a table where that column is a primary key of another table, which means that any data in a foreign key column must have corresponding data in the other table where that column is the primary key. In DBMS-speak, this correspondence is known as referential integrity. For example, in the Antiques table, both the BuyerID and SellerID are foreign keys to the primary key of the AntiqueOwners table (OwnerID; for purposes of argument, one has to be an Antique Owner before one can buy or sell any items), as, in both tables, the ID rows are used to identify the owners or buyers and sellers, and that the OwnerID is the primary key of the AntiqueOwners table. In other words, all of this "ID" data is used to refer to the owners, buyers, or sellers of antiques, themselves, without having to use the actual names.

Performing a Join

The purpose of these keys is so that data can be related across tables, without having to repeat data in every table--this is the power of relational databases. For example, you can find the names of those who bought a chair without having to list the full name of the buyer in the Antiques table...you can get the name by relating those who bought a chair with the names in the AntiqueOwners table through the use of the OwnerID, which relates the data in the two tables. To find the names of those who bought a chair, use the following query:

```
SELECT OWNERLASTNAME, OWNERFIRSTNAME
FROM ANTIQUEOWNERS, ANTIQUES
WHERE BUYERID = OWNERID AND ITEM = 'Chair';
```

Note the following about this query...notice that both tables involved in the relation are listed in the FROM clause of the statement. In the WHERE clause, first notice that the ITEM = 'Chair' part restricts the listing to those who have bought (and in this example, thereby owns) a chair. Secondly, notice how the ID columns are related from one table to the next by use of the BUYERID = OWNERID clause. Only where ID's match across tables and the item purchased is a chair (because of the AND), will the names from the AntiqueOwners table be listed. Because the joining condition used an equal sign, this join is called an equi join. The result of this query is two names: Smith, Bob & Fowler, Sam.

Dot notation refers to prefixing the table names to column names, to avoid ambiguity, as such:

```
SELECT ANTIQUEOWNERS.OWNERLASTNAME, ANTIQUEOWNERS.OWNERFIRSTNAME
FROM ANTIQUEOWNERS, ANTIQUES
WHERE ANTIQUES.BUYERID = ANTIQUEOWNERS.OWNERID AND ANTIQUES.ITEM = 'Chair';
```

As the column names are different in each table, however, this wasn't necessary.

DISTINCT and Eliminating Duplicates

Let's say that you want to list the ID and names of**only** those people who have sold an antique. Obviously, you want a list where each seller is only listed once--you don't want to know how many antiques a person sold, just the fact that this person sold one (for counts, see the Aggregate Function section below). This

means that you will need to tell SQL to eliminate duplicate sales rows, and just list each person only once. To do this, use the DISTINCT keyword.

First, we will need an equijoin to the AntiqueOwners table to get the detail data of the person's LastName and FirstName. However, keep in mind that since the SellerID column in the Antiques table is a foreign key to the AntiqueOwners table, a seller will only be listed if there is a row in the AntiqueOwners table listing the ID and names. We also want to eliminate multiple occurences of the SellerID in our listing, so we use DISTINCT **on the column where the repeats may occur.**

To throw in one more twist, we will also want the list alphabetized by LastName, then by FirstName (on a LastName tie), then by OwnerID (on a LastName and FirstName tie). Thus, we will use the ORDER BY clause:

```
>SELECT DISTINCT SELLERID, OWNERLASTNAME, OWNERFIRSTNAME
FROM ANTIQUES, ANTIQUEOWNERS
WHERE SELLERID = OWNERID
ORDER BY OWNERLASTNAME, OWNERFIRSTNAME, OWNERID
```

In this example, since everyone has sold an item, we will get a listing of all of the owners, in alphabetical order by last name. For future reference (and in case anyone asks), this type of join is considered to be in the category of inner joins.

Aliases in Sub Queries:

In this section, we will talk about *Aliases*, *In* and the use of subqueries, and how these can be used in a 3-table example. First, look at this query which prints the last name of those owners who have placed an order and what the order is, only listing those orders which can be filled (that is, there is a buyer who owns that ordered item):

```
SELECT OWN.OWNERLASTNAME Last Name, ORD.ITEMDESIRED Item Ordered
FROM ORDERS ORD, ANTIQUEOWNERS OWN
WHERE ORD.OWNERID = OWN.OWNERID
AND ORD.ITEMDESIRED IN

      (SELECT ITEM
      FROM ANTIQUES);
```

This gives:

```
Last Name Item Ordered
--------- ------------
Smith Table
Smith Desk
Akins Chair
Lawson Mirror
```

There are several things to note about this query:

1. First, the "Last Name" and "Item Ordered" in the Select lines gives the headers on the report.
2. The OWN & ORD are aliases; these are new names for the two tables listed in the FROM clause that are used as prefixes for all dot notations of column names in the query (see above). This eliminates ambiguity, especially in the equijoin WHERE clause where both tables have the column named OwnerID, and the dot notation tells SQL that we are talking about two different OwnerID's from the two different tables.
3. Note that the Orders table is listed first in the FROM clause; this makes sure listing is done off of that table, and the AntiqueOwners table is only used for the detail information (Last Name).
4. Most importantly, the AND in the WHERE clause forces the In Subquery to be invoked ("= ANY" or "= SOME" are two equivalent uses of IN). What this does is, the subquery is performed, returning all of the Items owned from the Antiques table, as there is no WHERE clause. Then, for a row from the Orders table to be listed, the ItemDesired must be in that returned list of Items owned from the Antiques table, thus listing an item only if the order can be filled from another owner. You can think of it this way: the subquery returns a set of Items from which each ItemDesired in the Orders table

is compared; the In condition is true only if the ItemDesired is in that returned set from the Antiques table.

Whew! That's enough on the topic of complex SELECT queries for now. Now on to other SQL statements.

Normalization in Database:

Normalization is the process of efficiently organizing data in a database.

There are two goals of the normalization process: eliminating redundant data (for example, storing the same data in more than one table) and ensuring data dependencies make sense (only storing related data in a table). Both of these are worthy goals as they reduce the amount of space a database consumes and ensure that data is logically stored.

The Normal Forms

The database community has developed a series of guidelines for ensuring that databases are normalized. These are referred to as normal forms and are numbered from one (the lowest form of normalization, referred to as first normal form or 1NF) through five (fifth normal form or 5NF). In practical applications, you'll often see 1NF, 2NF, and 3NF along with the occasional 4NF. Fifth normal form is very rarely seen and won't be discussed in this article.

Before we begin our discussion of the normal forms, it's important to point out that they are guidelines and guidelines only. Occasionally, it becomes necessary to stray from them to meet practical business requirements. However, when variations take place, it's extremely important to evaluate any possible ramifications they could have on your system and account for possible inconsistencies. That said, let's explore the normal forms.

First Normal Form (1NF)

First normal form (1NF) sets the very basic rules for an organized database:

- Eliminate duplicative columns from the same table.
- Create separate tables for each group of related data and identify each row with a unique column or set of columns (the primary key).

Second Normal Form (2NF)

Second normal form (2NF) further addresses the concept of removing duplicative data:

- Meet all the requirements of the first normal form.
- Remove subsets of data that apply to multiple rows of a table and place them in separate tables.
- Create relationships between these new tables and their predecessors through the use of foreign keys.

Third Normal Form (3NF)

Third normal form (3NF) goes one large step further:

- Meet all the requirements of the second normal form.
- Remove columns that are not dependent upon the primary key.

Fourth Normal Form (4NF)

Finally, fourth normal form (4NF) has one additional requirement:

- Meet all the requirements of the third normal form.

- A relation is in 4NF if it has no multi-valued dependencies.

Remember, these normalization guidelines are cumulative. For a database to be in 2NF, it must first fulfill all the criteria of a 1NF database.

When designing a relational database, it is normally a good thing to "normalize" the database. There are different degrees of normalization, but in general, relational databases should be normalized to the "third normal form". Simply put, this means that the attributes in each table should "depend on the key, the whole key and nothing but the key".

An example of a de-normalized database table is provided below. The database designer has assumed that there will never be a need to have more than two order items in any one order:

By moving repeating groups of attributes to a separate database table, the database design becomes more flexible. A single order can now support any number of order items; not just just two. The primary key (PK) of the Order Item table is the "Order Nbr" (represented by the relationship) plus the "Order Item Nbr":

The "Order Item Description" field is dependent on the "Order Item Code"; not the unique identifier of the Order Item Table (i.e. "Order Nbr" + "Order Item Nbr"). By creating a classification table, the database become even more flexible. New codes can easily be added. The "Order Item Description" for a given code can easily be altered should the need ever arise (e.g. "blue widget" => "light blue widget"):

A RDBMS alone will not solve all data management issues. A good data analyst and/or database analyst is needed to design a flexible and efficient relational database.

There are many different vendors that currently produce relational database management systems (RDBMS). Relational databases vary significantly in their capabilities and in costs. Some products are proprietary while others are open source. The leading vendors of RDBMS are listed below:

RDBMS Vendors	RDBMS
Computer Associates	INGRES
IBM	DB2
INFORMIX Software	INFORMIX
Oracle Corporation	Oracle
Microsoft Corporation	MS Access
Microsoft Corporation	SQL Server
MySQL AB	MySQL
NCR Teradata	
PostgreSQL Dvlp Grp	PostgreSQL
Sybase	Sybase 11

Although most businesses manage their corporate data in relational database management systems (RDBMS), many businesses still operate application systems that use flat files for data storage. Many of these systems are legacy "batch" systems that can't support online data transactions. A flat file can be stored on computer tape or on a hard drive of some sort.

Network databases such as IDMS became popular in the 1980s, when computers were much less powerful than the ones that exist today. Although network databases supported online transactions, the databases were relatively inflexible. Once a database was designed, it was often costly to implement changes.

Hierarchical databases were also popular in the 1970s and 1980s.

Eliminate Repeating Groups(1NF):

In the original member list, each member name is followed by any databases that the member has experience with. Some might know many, and others might not know any. To answer the question, "Who knows DB2?" we need to perform an awkward scan of the list looking for references to DB2. This is inefficient and an extremely untidy way to store information.

Moving the known databases into a seperate table helps a lot. Separating the repeating groups of databases from the member information results in **first normal form**. The MemberID in the database table matches the primary key in the member table, providing a foreign key for relating the two tables with a join operation. Now we can answer the question by looking in the database table for "DB2" and getting the list of members.

Eliminate Redundant Data(2NF):

In the Database Table, the primary key is made up of the MemberID and the DatabaseID. This makes sense for other attributes like "Where Learned" and "Skill Level" attributes, since they will be different for every member/database combination. But the database name depends only on the DatabaseID. The same database name will appear redundantly every time its associated ID appears in the Database Table.

Suppose you want to reclassify a database - give it a different DatabaseID. The change has to be made for every member that lists that database! If you miss some, you'll have several members with the same database under different IDs. This is an update anomaly.

Or suppose the last member listing a particular database leaves the group. His records will be removed from the system, and the database will not be stored anywhere! This is a delete anomaly. To avoid these problems, we need**second normal form**.

To achieve this, separate the attributes depending on both parts of the key from those depending only on the DatabaseID. This results in two tables: "Database" which gives the name for each DatabaseID, and "MemberDatabase" which lists the databases for each member.

Now we can reclassify a database in a single operation: look up the DatabaseID in the "Database" table and change its name. The result will instantly be available throughout the application.

Eliminate Columns Not Dependent on Key(3NF):

Eliminate Columns Not Dependent On Key(3NF)

The Member table satisfies first normal form - it contains no repeating groups. It satisfies second normal form - since it doesn't have a multivalued key. But the key is MemberID, and the company name and location describe only a company, not a member. To achieve third normal form, they must be moved into a separate table. Since they describe a company, CompanyCode becomes the key of the new "Company" table.

The motivation for this is the same for second normal form: we want to avoid update and delete anomalies. For example, suppose no members from the IBM were currently stored in the database. With the previous design, there would be no record of its existence, even though 20 past members were from IBM!

Member Table

MID	Name	Company	CompLoc
1	John Smith	ABC	Alabama
2	Dave Jones	MCI	Florida
3	Mike Beach	IBM	Delaware
4	Jerry Miller	MCI	Florida
5	Ben Stuart	AIC	Nebraska
6	Fred Flint	ABC	Alabama
7	Joe Blow	RU Nuts	Iowa
8	Greg Brown	XYZ	New York
9	Doug Hope	IBM	Delaware

Member Table

MID	Name	CID
1	John Smith	1
2	Dave Jones	2
3	Mike Beach	3
4	Jerry Miller	2
5	Ben Stuart	4
6	Fred Flint	1
7	Joe Blow	5
8	Greg Brown	6
9	Doug Hope	3

Company Table

CID	Name	Location
1	ABC	Alabama
2	MCI	Florida
3	IBM	Delaware
4	AIC	Nebraska
5	RU Nuts	Iowa
6	XYZ	New York

BCNF(Boyce-Codd Normal Form):

BCNF. Boyce-Codd Normal Form

Boyce-Codd Normal Form states mathematically that:
A relation R is said to be in BCNF if whenever X -> A holds in R, and A is not in X, then X is a candidate key for R.
BCNF covers very specific situations where 3NF misses inter-dependencies between non-key (but candidate key) attributes. Typically, any relation that is in 3NF is also in BCNF. However, a 3NF relation won't be in BCNF if (a) there are multiple candidate keys, (b) the keys are composed of multiple attributes, and (c) there are common attributes between the keys.

Basically, a humorous way to remember BCNF is that all functional dependencies are:
"The key, the whole key, and nothing but the key, so help me Codd."

Isolate Independent Multiple Relationships(4NF):

This applies primarily to key-only associative tables, and appears as a ternary relationship, but has incorrectly merged 2 distinct, independent relationships.

The way this situation starts is by a business request list the one shown below. This could be any 2 M:M relationships from a single entity. For instance, a member could know many software tools, and a software tool may be used by many members. Also, a member could have recommended many books, and a book could be recommended by many members.

Initial business request

So, to resolve the two M:M relationships, we know that we should resolve them separately, and that would give us 4th normal form. But, if we were to combine them into a single table, it might look right (it is in 3rd normal form) at first. This is shown below, and violates 4th normal form.

Incorrect solution

To get a picture of what is wrong, look at some sample data, shown below. The first few records look right, where Bill knows ERWin and recommends the ERWin Bible for everyone to read. But something is wrong with Mary and Steve. Mary didn't recommend a book, and Steve Doesn't know any software tools. Our solution has forced us to do strange things like create dummy records in both Book and Software to allow the record in the association, since it is key only table.

Sample data from incorrect solution

The correct solution, to cause the model to be in 4th normal form, is to ensure that all M:M relationships are resolved independently if they are indeed independent, as shown below.

Correct 4th normal form

NOTE! This is not to say that ALL ternary associations are invalid. The above situation made it obvious that Books and Software were independently linked to Members. If, however, there were distinct links between all three, such that we would be stating that "Bill recommends the ERWin Bible as a reference for ERWin", then separating the relationship into two separate associations would be incorrect. In that case, we would lose the distinct information about the 3-way relationship.

Isolate Semantically Related Multiple Relationship:

OK, now lets modify the original business diagram and add a link between the books and the software tools, indicating which books deal with which software tools, as shown below.

Initial business request

This makes sense after the discussion on Rule 4, and again we may be tempted to resolve the multiple M:M relationships into a single association, which would now violate 5th normal form. The ternary association looks identical to the one shown in the 4th normal form example, and is also going to have trouble displaying the information correctly. This time we would have even more trouble because we can't show the relationships between books and software unless we have a member to link to, or we have to add our favorite dummy member record to allow the record in the association table.

Incorrect solution

The solution, as before, is to ensure that all M:M relationships that are independent are resolved independently, resulting in the model shown below. Now information about members and books, members and software, and books and software are all stored independently, even though they are all very much semantically related. It is very tempting in many situations to combine the multiple M:M relationships because they are so similar. Within complex business discussions, the lines can become blurred and the correct solution not so obvious.

Correct 5th normal form