UNIT - I

CHAPTER - 1

BASIC CONCEPTS

ALGORITHM SPECIFICATION

Introduction

The concept of an algorithm is fundamental to computer science.

Definition:

An algorithm is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria:

1. Input. There are zero or more quantities that are externally supplied.
2. Output. At least one quantity is produced.
3. Definiteness. Each instruction is clear and unambiguous
4. Finiteness. If we trace out the instructions of an algorithm then for all cases, the algorithm terminates after a finite number of steps.
5. Effectiveness. Every instruction mu7st be basic enough to be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in (3); it also must be feasible.

We can use a natural language like English, also.

Example [Binary search]: Assume that we have $n \geq 1$ distinct integers that are already sorted and stored in the array list. That is, list [0]≤ list [1]≤ …≤ list [n-1]. We must figure out if an integer seachnum is in this list. If it is we should return an index, i, such that list[i] = searchnum. If searchnum, isnot present, we should return- 1. Since the list is sorted we may use the following method to search for the value.

Let left and right, respectively, denore the left and right ends of the list to be searched. Initially. left = 0 and right= n-1. Let middle = (left+right)/2 be the middle position in the list. If we compare list with seachnum,we obtain oe of three reults:

1. searchnum < list [midddle]. In this case, if searchnum is present, it must be in the positions between 0 and middle – 1. Therefore,we set return middle.

2. searchnum = list [middle]. In this case, we return middle.

3. searchnum > list [middle].In this case, if searchnum is present, it must be in the positions between middle + 1 and n-1. So, we set left to middle + 1.

```
# include < stdio.h>

# include <math.h>

#define MAX_SIZE 101

#define SWAP(x,y,t) = ((t) = (x), (x)= (y), (y) = (t))

void sort ( int [], int); / *selection sort*/

void main (void)

{

int i, n;

int list[MAZ _ SIZE];

printf (“ Enter the number of numbers to generate: “);

scanf(“%d “, &n)‘‘

if ( n < 1 ll  n>  MAX_size)    {

fprintf(stderr, “ Improper value of n\n”);

exit (EXIT_ FALLURE);

}

for (i= 0; i<n; i++) {/* randomly generate numbers*/

list [i] = rand ( ) % 1000;

printf( “% d “, list[i];

}

sort (list,n);
printf(“ \n Sorted array:\n”);

for (i= 0; i< n; i++)/* print out sorted numbers*/

Printf(“% d “, list[i]);

printf(“\n”);

}

void sort (int list[ ], int n)

{

int I, j, min, temp;

for (i=0; i< n-1; i++) {

min = I;

for (j=i+1; j< n; j++)

if (list[j] < list[min])

min= j;

SWAP (list[i], list [min], temp);

            }

}
```

Basic concepts

If searchnum has not been found and there are still integers to check, we recalculate middle and continue the search. Program 1.5 implements this searching strategy. The algorithm contains two subtasks: (1) determining if there are any integers left to check, and (2) comparing seachnum to list [middle].

```
While (there are more integers to check){

Middle = (left +right)/2;

If (searchnum < list [ middle])
```

Return middle;

Else left= middle+1;

Program : Searching a sorted list

We can handle the comparisons through either a function or a macro. In either case, we must specify values to signify less than, equal, or greater than. We will use the strategy followed in C"s library function:

- We return a negative number (-1) if the first number is less than the second.
- We return a 0 if the two numbers are equal.
- We return a positive number (1) if the first number is greater than the second.

Although we present both a function (program 1.6) and a macro, we will use the macro throughout the text since it works with any data type.

```
Int compare ( int x, int y)

{/* compare x and y, return -1 for less than, 0 for equal,

1 for greater*/

If (x< y) return -1;

Else if ( x = = y ) return 0;

Else return 1;

Int binsearch ( int list [ ], int searchnum, int left,


int right)

{/* search list [0] < = list [1] <=… <= list [n-1] for

searchnum. Returnits position if found. Otherwise

return – 1* /

int middle;
while (left< = right ) {

middle = (left + right)/2;

switch (COMPARE (list [middle], seachnum)){

case -1 : left = middle + 1;

break;

case 0 : return middle;

case1 : right = middle – 1;

}

}

return -1;

}
```

The search strategy just outlined is called binary search. The previous examples have shown that algoriths are implemented as functions in C. Indeed functions are the primary vehicle used to divide a large program into managea ble pieces.

They make the program easier to read, and, because the functions can be tasted separately, increase the probability that it will run correctly.

Recursive Algorithms

Typically, beginning programmers view a function as something that is invoked (called) by another function. Functions can called themselves (direct recursion) or they may call other functions that invoke the calling function again(indirect recursion). These recursive mechanisms are not only extremely powerful, but they also frequently alllow us to express an otherwise complex process in very clear terms.

```
Int binsearch ( int list [ ], int searchnum, int left,

int right)

{/* search list [0] < = list [1] <=… <= list [n-1] for
searchnum. Returnits position if found. Otherwise

return – 1* /

int middle;

if (left< = right ) {

middle = (left + right)/2;

switch (COMPARE (list [middle], seachnum)){

case -1 : return

    binsearch (list, searchnum, middle + 1, right);

case 0 : return middle;

case 1: return

    binsearch ( list, searchnum, left, middle -1);

            }

    }

return -1;

}
```

The factorial function n ! has value 1 when $n \leq 1$ and value n* (n-1)! When n> 1. Write both a recursive and an iterative C function to compute n!.

The Fibonacci numbers are defined as : $f_0 = 0$, $f_1 = 1$, and $f_i = f_{i-1} + f_{i-2}$ for $i > 1$.

Write both a recursive and n iterative C function to compute $f_{i.}$

DATA ABSTRACTION

All programming languages provide at least a minimal set of predefined data types, plus the ability to construct new, or user – defined types. It is appropriate to ask the question, “ what is a data type?”

Definition: A data type is a collection of objects and a set of operations that act on those object.

Whether your program is dealing with predefined data types or user- defined data types, these two aspects must be considered: objects and operations.

It has been observed by many software designers that hiding the representation of objects of a data type from its users is a good design strategy. In that case, the user is constrained to manipulate the objects solely through the functions that are provided.

Definition: An abstract data type (ADT) is a data type that is organized in such a way that the specification of the objects and the specification of the operations on the objects is separated from the representation of the objects and the implementation of the operations.

Some programming languages provide explicit mechanisms to support the distinction between specification and implementation. For example, Ada has a concept called a package, and C++ has a concept called a class. Both of these assist the programmer in implementing abstract data types.

How does the specification of the operations of an ADT differ from the implementation of the operations? The specification consists of the names of every function, the type of its arguments, and the type of its result. There should also be a description of what the function does, but without appealing to internal representation or implementation details. This requirement is quite important, and it implies that and abstract data type is implementation – independent.

ADT Natural Number is

Objects: an ordered subrange of the integers starting at zero and ending at the maximum integer (INT _ MAX) on the computer

Functios:

For all x,y € Natural Number, TRUE, FALSE € Boolean

And where +, -, <, and == are the usual integer operations

Natural Number Zero(_ : := 0

Boolean IsZero( )   ::= if (x) return False else return True

```
Booleaqn Equal (x,y) ::= if (x==y) return truee else return false

Natural Number Successor(x) ::= if (x == INT_ MAX) return x

Else return x+1

Natural Number Add (x,y) ::= if (( x+1) < = INT _ MAX) return x + y

Else return  int_max

Natural Number Subtract (x,y) ::= if (x < y) return 0

Else return x – y

End Natural Number
```

EXERCISE

Add the following operations to the Natural Number ADT: Predecessor, IsGreater, Multiply, Divide.

Create an ADT, Boolean. The minimal operations are And, Or, Not,Xor (Exclusiveor), and Implies.

PERFORMANCE ANALYSIS

There are many criteria, upon which we can judge a program, including:

1. Does the program meet the original specifications of the task?
2. Does it work correctly?
3. Does the program contain documentation that shows how to use it and how it works?
4. Does the program effectively use functions to create logical units?
5. Is the program‟s code readable?

Although the above criteria are vitally important particularly in the development of large systems , it is difficult to explain how to achieve them. The criteria are associated with the development of a good programming style and this takes experience and practice. We also can judge a program on more concrete criteria, and so we add two more criteria.

These criteria focus on performance evaluation, which we can loosely devide into two distinct fields. The first fields focuses on obtaining estimates of timer and space that are machine independent. We call this field performance analysis, but is subject matter is the heart of an important branch of computer science known as complexity theory. The second field, which we call performance measurement, obtains machine-dependent running times.

Definition: The space complexity of a program is the amount of memory that it needs to run to completion. The time complexity of a program is the amount of computer time that it needs to run to completion.

Space Complexity

The space needed by a program is the sum of the following components:

1. Fixed space requirements: This component refers to space requirements that do not

Depend on the number and size of the program‟s inputs and outputs. The fixed requirements include the instruction space (space needed to store the code), space for simple variable‟s fixed-size structured variables(such as structs) and constants.

This component consists of the space needed by structured variables whose size depends on the particular instance, I, of the problem being solved. It also includes the additional space required when a function uses recursion. The variable space requirement of a program P working on an instance I is denoted S_p (I).

We can express the total space requirement S(p) of any program as:

$S(P)= c + S_p(I)$

Where c is a constant representing the fixed space requirements.

Time Complexity

The time, T(P), taken by a program, P, is the sum of its compile time and its run (or execution) time. The compile time is similar to the fixed space component since it does not depend on the instance characteristics, we are really concerned only with the program‟s execution time, T_p.

Determining T_p is not an easy task because it requires a detailed knowledge of the compiler‟s attributes. That is, we must know how the compiler translate our source program into object code.

That are performed when the program is run with instance characteristic n.

Asymptotic Notation (OΩθ)

Our motivation to determine step counts is to be able to compare the time complexities of two programs that compute the same function and also to predict the growth in run time as the instance characteristics change.

Determining the exact step count(either worst case or average) of a program can prove to be an exceedingly difficult task. Expending immense effort to determine the step count exactly isn‟t very worthwhile endeavor as the notion of a step is itself inexact. (Both the instructions x = y and x=y+z+(x/y) + (x*y*z*-x/z) count as one step.) Because of the inexactness of what a step stands for, the exact step count isn‟t very

```
Void prod (int a [ ] [MAX SIZE], int b[ ] [ MAX SIZE],

Int c[ ] [ MAX SIZE], int rowsa, int colsb int colsa)

{

        Int I, j, k;

For (i = 0; I < rowsa; i ++)

For (j =0; j< closb; j++)

{

C[i] [j] = 0;

For (k = 0; k< colsa; k++)

C[i] [j]+ = a [i] [k] * b [k] [j];

        }

}
```

Program 1.21: Matrix product function

```
Void transpose ( int a[ ] [MAX _ SIZE])

{
```

```
Int i, j, temp;

For (I = 0; i< MAX SIZE - 1; i++)

For (j = i+1; j< MAX SIZE; j++)

SWAP (a [i] [j], a[j] [i], temp;
```

Definition: [Big „ oh‟] f (n) = O (g(n)) (read as „ „ f of n is big oh of g of n‟) iff (if and only if) there exist positive constants c and n_0 such that $f(n) \leq cg(n)$ for all $n, n \geq n_0$.

As illustrated by the previous example, the statement f(n) = O(g(n)) only states that g(n) is an upper bound on the value of f(n) for all $n, n \geq n_0$.

Definition: [Omega] $f(n) = \Omega(g(n))$ (readas „ f of n is omega of g of n‟) iff there exist positive constants c and n_0 such that $f(n) \geq cg(n)$ for all $n, n \geq n_0$.

The theta notation is more precise than both the „ big oh‟ and omega notations.

$f(n) = \Theta(g(n))$ iff g (n) is both an upper and lower bound on f(n).

ARRAYS AND STRUCTURES

ARRAYS

We begin our discussion by considering an array as an ADT. An array is a set of pairs, < index, value>, such that each index that is defined has a value associated with it. In mathematical terms, we call this a correspondence or a mapping.

ADT Array is

Objects: A set of pairs< index, value > where for each value of index. There is a value fro, the set item. Index is a finite ordered set of one or more dimensions, for example, { 0,…, n-1 } for oe dimension, {(0,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2) for twp dimensions, etc.

Functions:

For all A€ Array, i € index, x € item, j, size € integer

Array Create (j, list) ::= return an array of j dimensions where list is a j- tuple whose ith element is the size of the ith dimension. Items are undefined.

Item Retrieve (A,i) ::= if (I in index) return the items associated with index value I in array A else return error

Array Store (A,i,x) ::= if (i in index)

Return an array that is identical to array A except the new pair <I,x> has been inserted else return error.

ADT: Abstract Data Type Array

Returns the value associated with the index if the index is valid, or an error if the index is invalid. Store accepts an array, an index, and an item, and returns the original array augmented with the new < index, value> pair. The advantage of this ADT definition is that it clearly points out the fact that the array is a more general structure than “a consecutive set of memory locations.”

POLYNOMIALS

The Abstract Data Type

Array are not only data structures is their own right, we can also use them to implement other abstract data types. For instance, let us consider one of the simplest and most commonly found data structures: the order or linear list.

- Day of the week: (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday)
- Values in a deck of cards: (Ace, 2,3,4,5,6,7,8,9,10,Jack,Queen,King)

We can perform many operations on lists, including:

- Finding the length, n, of a list.
- Regarding the items in a list from left to right (or right to left).
- Retrieving the ith item from a list, $0\leq$ in $<$ n.
- Replacing the item in the ith position of a list, $0 \leq$ in $<$ n.
- Inserting a new item in the ith position of a list, $0 \leq$ in $<$ n. The items previously numbered I, i+1,…, n-1 become items numbered I, i+1,.., n-2.

The most common implementation is to represent an ordered list as an array where we associate the list element, item $_i$, with the array idex i. WE call this a sequential mapping because, assuming the standard implementation of an array we are storing item I, item i+1 into consecutive slots i and i+1 of the array. Sequential mapping works well for most of the operations listed above. Thus, we can retrieve an item, replace an item, or find the length of a list, in constant time. We also can read the items in the list, from either direction, by simply changing subscripts in a controlled way. Only insertion and deletion pose problems since the sequential allocation forces us to move items so that the sequential mapping preserved.

Viewed from a mathematical perspective, a polynomial is a sum of terms, where each term has a form ax^e, where x is the variable, a is the coefficient and e is the exponent. Two example polynominals are:

$A(x) = 3x^{20}+2x^5+4$ and $B(x) = x^4 + 10x^3+ 3x^2+1$

The largest (or leading) exponent of a polynomial is called its degree. Coefficients that are zero are not displayed. The term with exponent equal to zero does not show the variable since x raised to a power of zero is 1.

Polynominal Representation

We are now ready to make some representation decisions. A very reasonable first decision requires unique exponents arranged in decreasing order. This requirement considerably simplifies many of the operations.

This algorithm works by comparing terms from the two polynominals until one or both of the polynomials becomes empty. The switch statement performs the comparisons and adds the proper term to the new polynomial, d. if one of the polynominals becomes empty, e copy the remaining terms from the nonempty polynomial into d. With these insights, we now considered the representation question more carefully.

ADT Polynomial is

Objects: $p(x) = a^{1} x^{e1} + \ldots + a^{n}x^{en}$; a set of ordered pairs of < e_i, a_i> where a I in Coefficients and e_i in Exponents, e_i are integers >=0

Functions:

For all poly, poly 1, poly 2€ Polynomial, coef € Coffucients, expon € Exponents

Polynomial Zero ( )	::=	return the polynomial, p (x) =0
Boolean Is Zero (poly)	::=	if (poly) return FALSE else return TRUE
Coefficient Coef (poly, expon)	::=	if (expon € poly) retrun is cofficient else return zero
Exponent Lead Exp (poly)	::=	return the largest exponent in poly
Polynomial Attach (poly, coef, expon)	::=	if(expon € poly) return error else return the polynomical poly with the term<coef, expon> inserted

Polynomial Remove (poly,expon)	::=	if (expon € poly) return the polynomical poly with the term whose exponent is expon deleted else return error
Polynomial Single Mult(poly, coef, expon)	::=	return the polynomial poly. coef. X ^{expon}
Polynomial Add (poly 1, poly 2)	::=	return the polynomial poly 1+ poly 2
polynomial Mult (plly 1, poly 2)	:=	return the polynomial poly 1 poly 2

Polynomical Addition

We would now like to write a C function that adds two polynomials, A and B, represented as above to obtain D = A+B. To produce D(x), padd adds A (x) and B (x) term by term.

Function to add two polynomials

Analysis of Padd:

Since the number of nonzero terms in A and B are the most important factors in the time complexity, we will carry out the analysis using them. Therefore, let m ad n be the number of nonzeroterms in A and B, respectively. If m > 0 and n> 0, the while loop is entered. Each interation of the loop requires o (1) time. At each iteration, we increment the value of start A or start B or both. Since the iteration terminates

```
Void attach (float coefficient, int exponent)

{ /* add a new term to the polynomial*/

if( avail  > = MAX _TERMS)          {

        fprintf (staderr,” Too many terms in the polynomial \n”);

        exit (EXIT _ FALLURE);

        }

terms [avail]. Coefficient;

terms [avail ++]. expon = exponenet;
```

Function to add a new term

When either start A or start B exceeds finish A or finish B, respectively, the number of iterations is bounded by m+n -1. This worst case occurs when:

A(x) = $\sum_{i=0}^{n} x2i$ and B(x) = $\sum_{i=0}^{n} x2i + 1$

The time for the remaining two loops is bounded by O(n + m) because we cannot iterate the first loop more than m times and the second more than n times. So, the asymptotic computing time of this algorithm is O(n+m).

EXERCISE

- Write a function, pmult, that multiplies two polynomials, Figure out the computing time of your function.
- Let A(x) = $x^{2n} + x^{2n-2} + .. + x^2 + x^0$ and B(x) = $x^{2n+1}\ x^{2n} + \ldots + x^3 + x$. For these polynomials, determine the exact number of times each statement of padd is executed.

SPARSE MATRICES:

A general matrix consists of m rows and n columns of numbers. Such a matrix has sm n elements. Matrices has many zero entries. Such a matrix is called sparse.

Ordinary Matrix

$$\begin{pmatrix} 4\ 6\ 8 \\ 5\ 0\ 2 \end{pmatrix}$$

2X3

Sparse Matrix

$$\begin{pmatrix} 0\ 0\ 1 \\ 0\ 0\ 0 \end{pmatrix}$$

2X3

REFRESENTATION OF ARRAYS:

If an array is declared A(I_1:u_1,I_2:u_2 …., I_n:u_n), then it is easy to see that the number of elements is

$$\sum_{1=1}^{n} (u1 - Ii + 1)$$

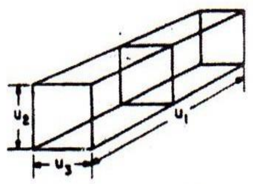
One of the common ways to represent an array is in row major order. If we have the declaration.

A (4:5, 2:4, 1:2, 3:4)

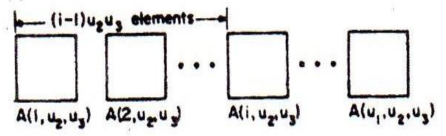
The two dimensional array $(A_1{:}u_1)$, $1u_2)$ may be interpreted as u_1 rows : row_1. row_2,…, each row consisting of u_2 elements.

If α is the address of A(1,1), then the address of (A I, 1) is α + (I -1) u_2, as there are i - 1 rows each of size u_2 preceding the first element in the i-th row. Knowing the address of A(I -1), we can say that the address of A(I, I) is then simply α + (I, 1), we can say that the address of A(I, 1) is then simply α + (i-1) u_2 + (i - 1).

"Repeating in this way the address for A(i_1, i_2 ….Im…..) is



(a) 3-dimensional array $A(u_1,u_2,u_3)$ regarded as u_1 2-dimensional arrays.



(b) Sequential row major representation of a 3-dimensional array. Each 2-dimensional array is represented as in Figure

Sequential representation of A(u_1,u_2,u_3)

α + (i1 - 1)u2 u3….un

+ (i2 - 1) u3 u4…un

+ (i3 - 1) u4 u5 …un

:

:

+ (in-1 - 1)un

+ (in - 1)

$$\alpha + \sum_{j-1}^{n}(i_j - 1)a_i \text{ with } \begin{cases} a_i = \prod_{k-j+1}^{n} u_k\ 1 \leqq j < n \\ a_n = 1 \end{cases}$$

Stack

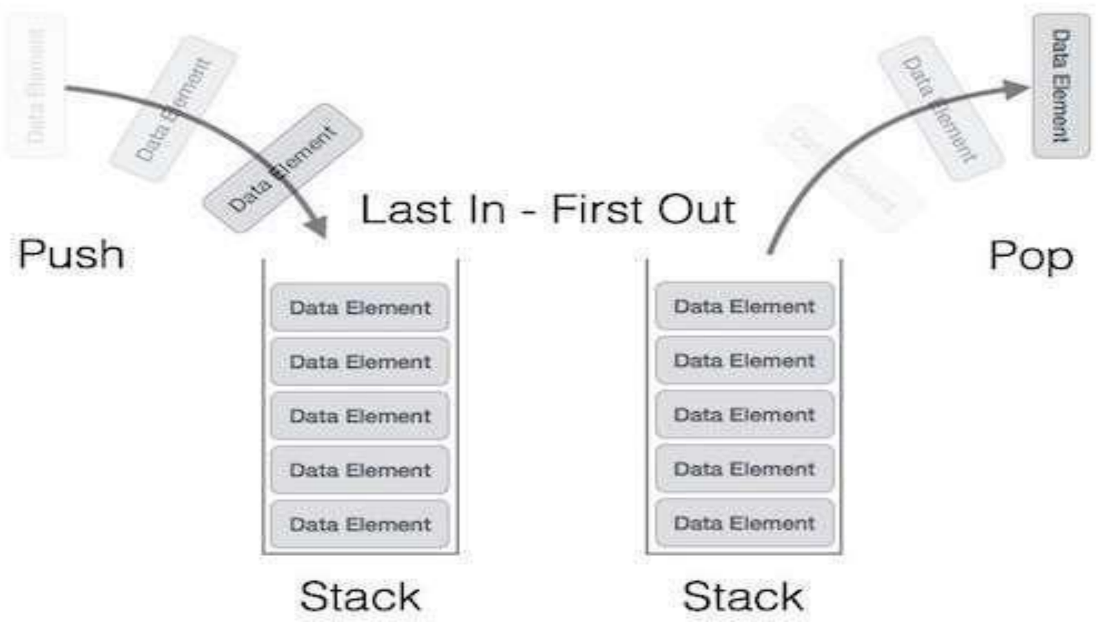
A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

Stack Representation

The following diagram depicts a stack and its operations −



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

- **push()** − Pushing (storing) an element on the stack.
- **pop()** − Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

- **peek()** − get the top data element of the stack, without removing it.
- **isFull()** − check if stack is full.
- **isEmpty()** − check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

First we should learn about procedures to support stack functions −

peek()

Algorithm of peek() function −

```
begin procedure peek

   return stack[top]

end procedure
```

Implementation of peek() function in C programming language −

```
int peek() {
   return stack[top];
}
```

isfull()

Algorithm of isfull() function −

```
begin procedure isfull


   if top equals to MAXSIZE
      return true
   else
      return
   false endif


end procedure
```

Implementation of isfull() function in C programming language −

```
bool isfull() {
   if(top == MAXSIZE)
      return true;
   else
      return false;
}
```

isempty()

Algorithm of isempty() function −

```
begin procedure isempty


   if top less than 1
      return true
   else
      return
   false endif


end procedure
```

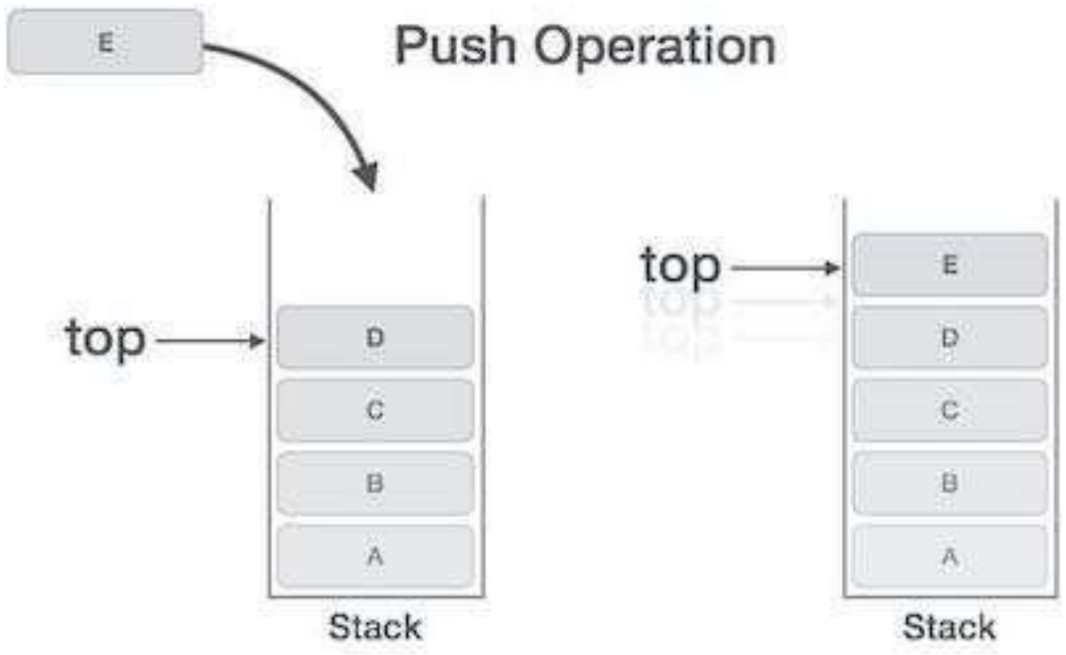
Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1 to determine if the stack is empty. Here's the code −

```
bool isempty() {
   if(top == -1)
      return true;
   else
      return false;
}
```

Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps −

- **Step 1** − Checks if the stack is full.
- **Step 2** − If the stack is full, produces an error and exit.
- **Step 3** − If the stack is not full, increments **top** to point next empty space.
- **Step 4** − Adds data element to the stack location, where top is pointing.
- **Step 5** − Returns success.



If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

Algorithm for PUSH Operation

A simple algorithm for Push operation can be derived as follows −

```
begin procedure push: stack, data


   if stack is full
      return null
   endif


   top ← top + 1


   stack[top] ← data


end procedure
```

Implementation of this algorithm in C, is very easy. See the following code −

```
void push(int data) {
   if(!isFull()) {
      top = top + 1;
      stack[top] = data;
   }else {
      printf("Could not insert data, Stack is full.\n");
   }
}
```

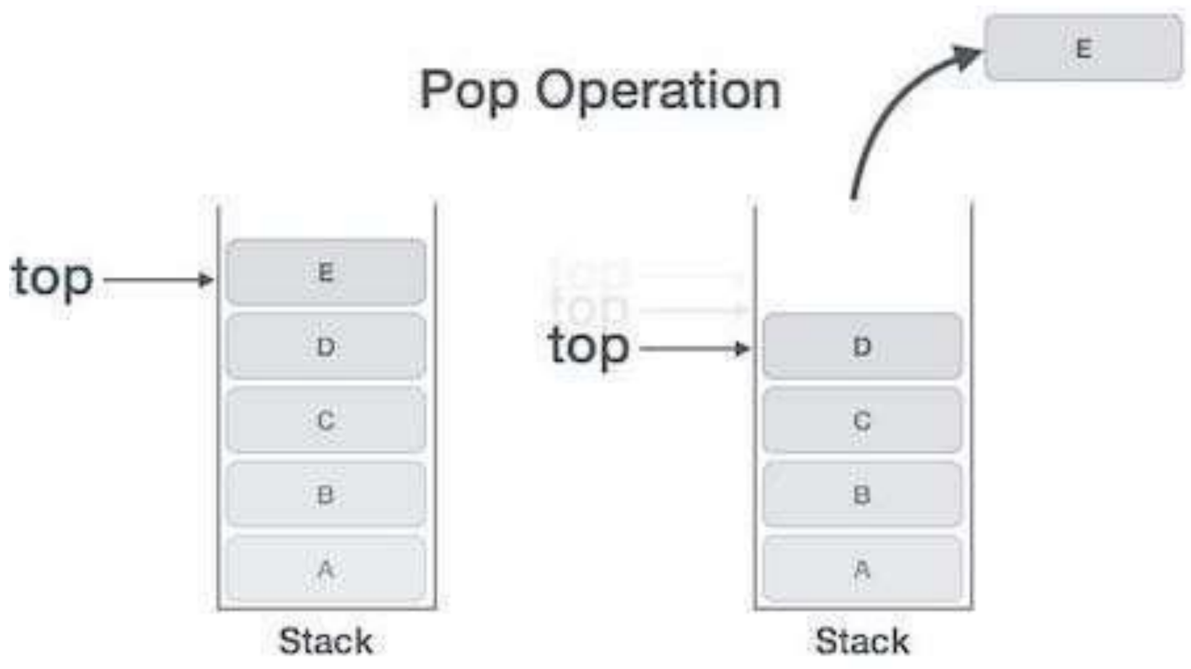
Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps −

- **Step 1** − Checks if the stack is empty.
- **Step 2** − If the stack is empty, produces an error and exit.

- **Step 3** – If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** – Decreases the value of top by 1.
- **Step 5** – Returns success.



Algorithm for Pop Operation

A simple algorithm for Pop operation can be derived as follows –

```
begin procedure pop: stack

   if stack is empty
      return null
   endif

   data ← stack[top]

   top ← top - 1

   return data

end procedure
```

Implementation of this algorithm in C, is as follows −

```
int pop(int data) {

   if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   }else {
      printf("Could not retrieve data, Stack is empty.\n");
   }
}
```

For a complete stack program in C programming language, please click here.

Stack Program in C

We shall see the stack implementation in C programming language here. You can try the program by clicking on the Try-it button. To learn the theory aspect of stacks, click on visit previous page.

Implementation in C

```
#include <stdio.h>

int MAXSIZE = 8;
int stack[8];
int top = -1;

int isempty() {

   if(top == -1)
      return 1;
   else
      return 0;
}
```

```
int isfull() {

   if(top ==
      MAXSIZE)
      return 1;
   else
      return 0;
}

int peek() {
   return stack[top];
}



int pop() {
   int data;

   if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   }else {
      printf("Could not retrieve data, Stack is empty.\n");
   }
}

int push(int data) {

   if(!isfull()) {
      top = top + 1;
      stack[top]
   = data;
   }else {
      printf("Could not insert data, Stack is full.\n");
   }
}
```

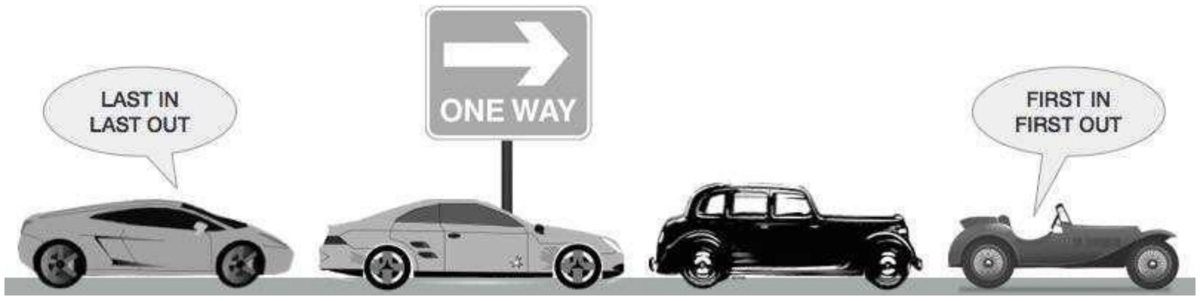
```
int main() {
    // push items on to the
    stack  push(3);
    push(5);
    push(9);
    push(1);
    push(12);
    push(15);

    printf("Element at top of the stack: %d\n" ,peek());
    printf("Elements: \n");

     // print stack data
     while(!isempty()) {
       int data = pop();
       printf("%d\n",data);
    }

    printf("Stack full: %s\n" , isfull()?"true":"false");
    printf("Stack empty: %s\n" , isempty()?"true":"false");

    return 0;
 }
```

If we compile and run the above program, it will produce the following result −

```
 Element at top of the stack: 15
 Elements:
 15
 12
 1
 9
 5
 3
 Stack full: false

Stack empty: true
```

Queue

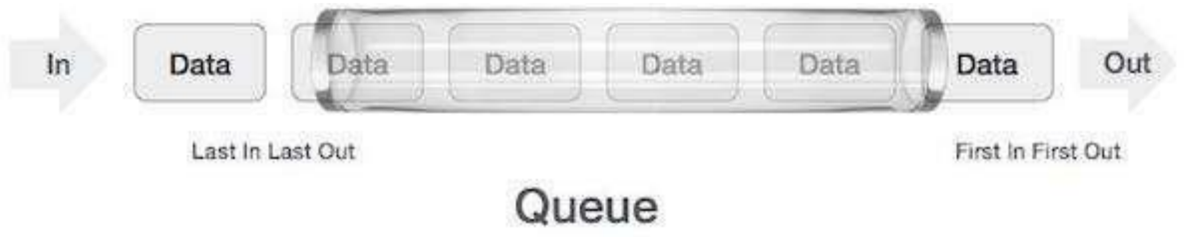
Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.



A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure −



As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues −

- **enqueue()** − add (store) an item to the queue.

- **dequeue()** − remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are −

- **peek()** − Gets the element at the front of the queue without removing it.
- **isfull()** − Checks if the queue is full.
- **isempty()** − Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueing (or storing) data in the queue we take help of **rear** pointer.

Let's first learn about supportive functions of a queue −

peek()

This function helps to see the data at the **front** of the queue. The algorithm of peek() function is as follows −

```
begin procedure peek

   return queue[front]

end procedure
```

Implementation of peek() function in C programming language −

```
int peek() {
   return queue[front];
}
```

isfull()

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function −

```
begin procedure isfull

   if rear equals to MAXSIZE
      return true
   else
```

```
return false
    endif


end procedure
```

Implementation of isfull() function in C programming language −

```
bool isfull() {
   if(rear == MAXSIZE - 1)
      return true;
   else
      return false;
}
```

isempty()

Algorithm of isempty() function −

```
begin procedure isempty


   if front is less than MIN OR front is greater than rear
      return true
   else
      return
   false endif


end procedure
```

If the value of **front** is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

Here's the C programming code −

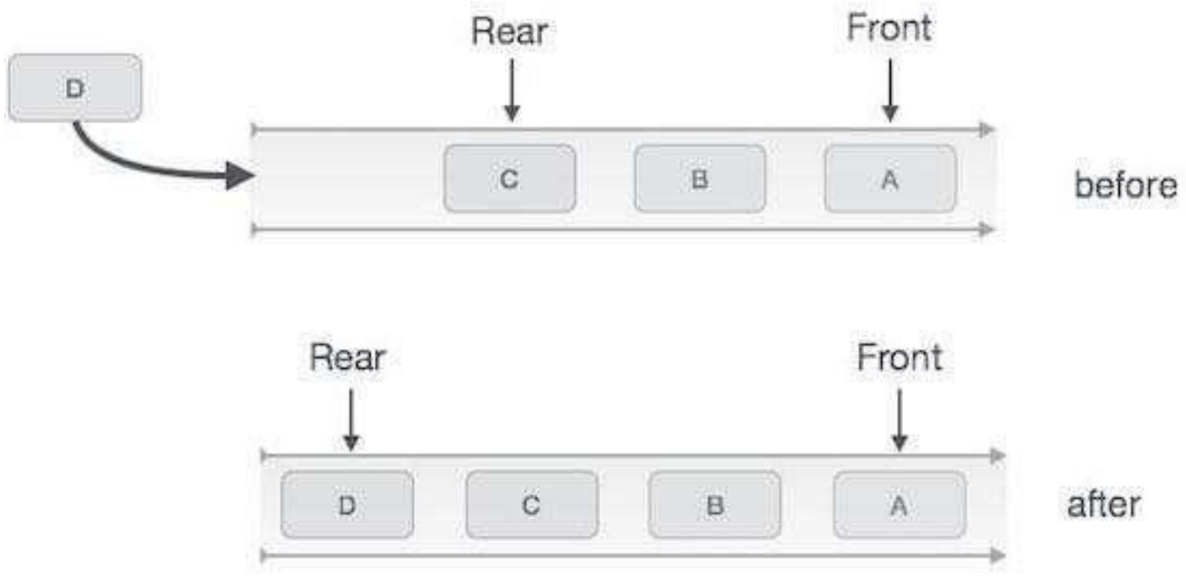
```
bool isempty() {
   if(front < 0 || front > rear)
      return true;
   else
      return false;
}
```

Enqueue Operation

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue −

- **Step 1** − Check if the queue is full.
- **Step 2** − If the queue is full, produce overflow error and exit.
- **Step 3** − If the queue is not full, increment **rear** pointer to point the next empty space.
- **Step 4** − Add data element to the queue location, where the rear is pointing.
- **Step 5** − Return success.



Queue Enqueue

Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

Algorithm for enqueue Operation

```
procedure enqueue(data)
   if queue is full

      return

   overflow endif


   rear ← rear + 1
   queue[rear] ← data
   return true


end procedure
```

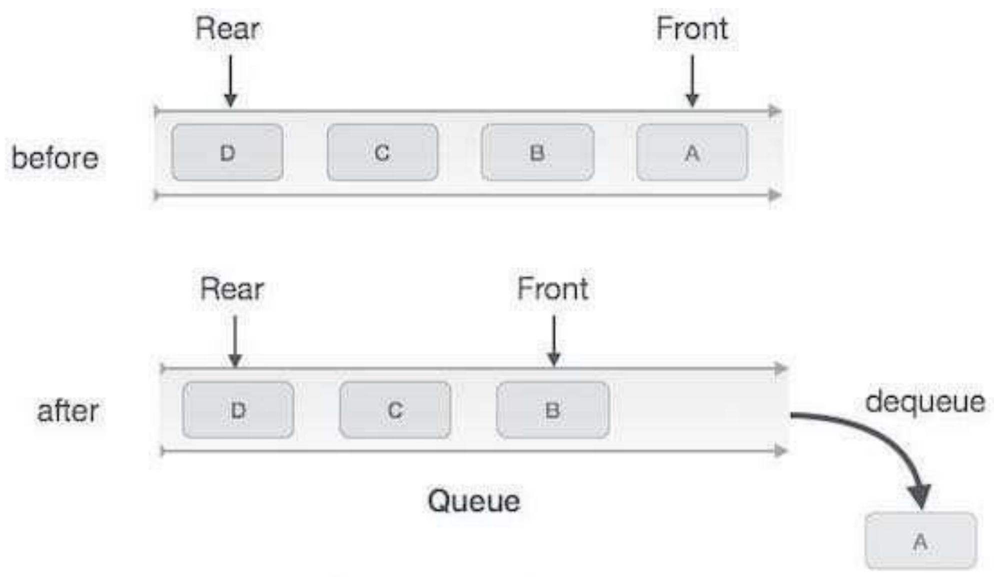
Implementation of enqueue() in C programming language –

```
int enqueue(int data)
   if(isfull())
      return 0;


   rear = rear + 1;
   queue[rear] = data;


   return 1;
end procedure
```

Dequeue Operation

Accessing data from the queue is a process of two tasks – access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation –

- **Step 1** – Check if the queue is empty.
- **Step 2** – If the queue is empty, produce underflow error and exit.
- **Step 3** – If the queue is not empty, access the data where **front** is pointing.
- **Step 4** – Increment **front** pointer to point to the next available data element.
- **Step 5** – Return success.



Queue Dequeue

Algorithm for dequeue Operation

```
procedure dequeue if
   queue is empty
      return
   underflow end if


   data = queue[front]
   front ← front + 1


   return true
end procedure
```

Implementation of dequeue() in C programming language –

```
int dequeue() {
   if(isempty())
      return 0;
   int data = queue[front];
   front = front + 1;


   return data;
}
```

Queue Program in C

We shall see the stack implementation in C programming language here. You can try the program by clicking on the Try-it button. To learn the theory aspect of stacks, click on visit previous page.

Implementation in C

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>

#define MAX 6

int intArray[MAX];
int front = 0;
int rear = -1;
int itemCount = 0;

int peek(){
   return intArray[front];
}

bool isEmpty(){
   return itemCount == 0;
}
bool isFull(){
   return itemCount == MAX;
}

int size(){
   return itemCount;
}

void insert(int data){

   if(!isFull()){
      if(rear ==
        MAX- 1){
        rear = -1;
      }

      intArray[++rear] =
      data; itemCount++;
   }
}

int removeData(){
   int data = intArray[front++];

   if(front ==
      MAX){ front
      = 0;
   }
   itemCount--;
   return data;
}

int main() {
   /* insert 5
   items */
   insert(3);
   insert(5);
   insert(9);
   insert(1);
   insert(12);

   // front : 0
   // rear : 4
   // ------------------
   // index : 0 1 2 3 4
   // ------------------
   // queue  :  3  5  9  1
   12 insert(15);
```

```
// front : 0
// rear : 5
// ---------------------
// index : 0 1 2 3 4   5
// ---------------------
// queue : 3 5 9 1 12 15

if(isFull()){
   printf("Queue is full!\n");
}

// remove one item
int num = removeData();

printf("Element removed: %d\n",num);
// front : 1
// rear : 5
// -------------------
// index : 1 2 3 4     5
// -------------------
// queue : 5 9 1 12 15

// insert
                mor
e items
insert(16);

// front : 1
// rear : -1
// ----------------------
// index : 0 1 2 3 4 5 //
---------------------
// queue : 16 5 9 1 12 15

// As queue is  full, elements will  not
       be inserted. insert(17);
insert(18);

// ----------------------
// index : 0 1 2 3 4 5 //
```

```
   // queue : 16 5 9 1 12 15 printf("Element
   at front: %d\n",peek());


   printf("----------------------
   \n"); printf("index : 5 4 3 2 1
   0\n"); printf("-------------------
   ---\n"); printf("Queue: ");


   while(!isEmpty()){
      int n = removeData();
      printf("%d ",n);
   }
}
```

```
Queue is full!
Element removed: 3
Element at front: 5
----------------------
index : 5 4 3 2 1 0
---------------------
Queue: 5 9 1 12 15 16
```

If we compile and run the above program, it will produce the following result

EVALUATION OF EXPRESSIONS:

An expression is made up of operands, and delimiters. For instance if A=4, B= C = 2, D = E = 3, we might want x to be assigned the value

a. A/ (B**C) + (E*D) – (A*C)

4/(2**3) + (*3(-(4*2)

(4/4)+ 9-8

Result -2

b. ((A/B) ** (C +D)* (E-A) * C

(4/2) **(2+3) + (3-4) * 2

(4/2)** 5 * -1 *2

(2**)* -2

32* -2

Result = -64

If e is an expression with operators, he conventional way of writing e is called infix, because the operators come in- between the operands, (Unary operators precede their operand). The postfix from of an expression call for each operators to appear after its operands. For example.

Ifix : A* B/C has postfix: AB * C/.

For example A* (B + C) * D has the postfix form ABC + * D*, and so the algorithm

Infix to Postfix:

Next Token	Stack	Output
None	Empty	None
A	Empty	A
*	*	A
(	*	A
B	*	AB
+	+	AB
C	+	ABC
)	*	ABC+
*	*	ABC+*
D	*	ABC+* D
Done	Empty	ABC+* D*

The rule will be that operators are taken out of the stack as long as their in- stack priority is p, is greater than equal to the incoming priority, icp of the new operator.

```
Procedure POSTFIX(E)

// Convent the infix expression E to postfix. Assume the last character of E is a „4‟ which will also be the last character of the postfix. Procedure NEXT0TOKEN returns either the next the operator, operand or delimiter- whichever comes next.

Stack ( 1:n ) is the used as a stack and the character „4‟ with

ISP( /‟4‟)= - 1 is used at the bottom of the stack. ISP and ICP are functios.*

STACK ( 1)← („ – „4‟) ; top ←1* initialize stack*

X ←NEWZT – TOKEN (E)

Case

: x = „4‟ while top > 1 do * empty the stack*

Print(STACK)(top)); ←top – 1

End

Print („4‟)

Return

: x is an operand : print (x)

: x = „)‟ while STCACK (top)f‟(„ do *unstuck until‟(„*

Print (STACK (top)); top ← top-1

end

top ←top – 1 * delete‟(„*

:else while ISP ( TACK(top)); ICP (x) do

Print (STACK(top)); top ← top – 1

end

call ADDS( x, STACK , n,top)* insert x in STACK*

end

forever

end POSTFIX
```

EXERCISES

Implement the *Stack Empty* function.

1. Implement the queue Full and queue Empty functions for the noncircular queue.
2. Implement the queue Full and queue Empty functions for the circular queue.
3. Using the noncircular queue implementation, produce a series of adds deletes that requires O (MAZ_QUEUE_SIZE) for each add.

4. Write the postfix form of the following expressions:

a) a*b*c

b) –a + b –c +d

c) a* -b + c

d) (a + b) * d+ e / (f + a* d) + c

5. Write the prefix form of the expressions in Exercise 1.

6. Write a C function that transforms a prefix expression into a postfix one. Carefully state any assumptions you make regarding the input. How much time and space does your function take?

7. Write a C function that transforms a postfix expression into a prefix one. How much time and space does your function take?

8. We must represent two stacks in an array, memory. [MEMORY_ SIZE]. Write C functions that add and delete an item from stack i, 0< I < n. Your functions should be able to add elements to the stacks as long as the total number of elements in both stacks is less than MEMORY_ SIZE-1.

9. Write a C function that implements the stack Full strategy discussed in the text.

10. Using the add and delete functions discussed in the text and stack Full form Exercise 3, Produce a sequence of additions / deletions that requires O (MEMORY _ SIZE) time for each add. Assume that you have two stacks and that your are starting from a configuration representing a full utilization of memory (MEMORY _ SIZE).

LINKED LISTS

SINGLY LINKED LISTS AND CHAINS

We studied the representation of simple data structures using an array and a sequential mapping. These representations had the property that successive nodes of the data object were stored a fixed distance apart.

When a sequential mapping is used for ordered lists, operations such as insertion and deletion of arbitrary elements become expensive. For example, considered the following list of three letter English words ending in AT:

(BAT, CAT, EAT, FAT, HAT, JAT, LAT, MAT, OAT, PAT, RAT, SAT,VAT,WAT)

To make this list complete we naturally ant to add the word GAT, which means gun or revolver. If we are using an array and a sequential mapping to keep this list, then the insertion of GAT will require us to move elements already in the list either one location higher or lower. We mush move either HAT, JAT, LAT,…, WAT or BAT, CAT, EAT and FAT.

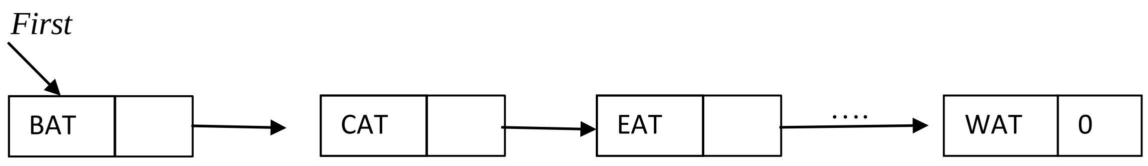
Suppose we decide to remove the word LAT, which refers to the Latvian monetary unit. Then again, we have to move many elements so as to maintain the sequential representation of the list.

An elegant solution to this problem of data movement in *sequential* representations is achieved by using *linked* representations. To access list elements in the correct order, with each element we store the address or location of the next element in that list. Thus, associated with each data item in a linked representation is pointer or like to the next item. In general, a linked list is comprised of nodes; each node has zero or more data fields and one or more link or pointer fields.

It is customary to draw linked lists as an ordered sequence of nodes with links being represented by arrows, as in Figure 4.2. Notice that we do not explicitly put in the values of the pointers but simply draw arrows to indicate they are there. The linked structures of Figures 4.1 and 4.2 are called singly linked lists or chains. In a *singly linked list*, each node has exactly one pointer field. A chain is a singly linked list that is comprised of zero or more node. When the number of nodes is zero, the chain is empty. The nodes of a non- empty chain are ordered so that the first node links to the second node; the second to the third; and so on. The last node of a chain has a 0 link.

	data		*link*
1	HAT		15
2			
3	CAT		4
4	EAT		9
5			
6			
7	WAT		0
8	BAT		3
9	FAT		1
10			
11	VAT		7
	. . .		. . .

Non sequential list- representation



Usual way to draw a linked list

It is easier to make insertions and deletions at arbitrary positions using a linked list rather than a sequential list. To insert the data item Gat between FAT and HAT, the following steps are adequate.

1. Get a node a that is currently unused.
2. Set the data field of a to GAT
3. Set the *link* field of a to point to the node after FAT, which contains HAT.
4. Set the *link* field of the node containing FAT to a.

	data		***link***
1	HAT		15
2			
3	CAT		4
4	EAT		9
5			1
6			
7	WAT		0
8	BAT		3
9	FAT		5
10			
11	VAT		7

LINKED STACK AND QUEUES:

Bothe the stack & queue are implemented as linked list

One can easily add a node at the top or delete one form the top. In figure 4.5 (b), one can easily add a node at the rear and both addition and deletions can performed at the front. Though for a queue we normally would not wish to add nodes at the front.

T(i) = Top of the stack

F(i)= Front of the queue

R(i)=Rear of the queue

Initial conditions:

T (i)=0

F(i) =0

Boundary conditions:

T(i)=0 iff stack i empty

F(i)=0 iff queue I empty

```
procedure DADDS(i,Y)
        *add element Y onto stack i*
call GETNODE(X)
DATA(X) ←Y *store data value Y into new node*
LINK (X) ←T (i) * attach new node to top of i-th stack*
T(i) ←X * reset stack pointer*
end ADDS
procedure DELETE(i, Y)
add Y to the ith queue*
call GETNODE(X)
DATA (X) ← Y LINK (X) ←0
if F(i) = 0 then [F(i) ← R(i) ←(X) * the queue was empty*
        else [ LINK (R(i)) ← X]* the queue was not empty*
end ADDQ]
Procedure DELETE (i,Y)
*delete the first node in the ith queue, set & to its DATA field*
if F(i) = 0 then an QUEUE _ EMPTY
else (X← F(i): F(i) ← LINK(X)* set X to front node*
```

Y ← DATA(X) : CALL RET (X) *remove data and return node*

end DELETEQ

EXERCISES

1. Rewrite delete(Program 4.3) so that it uses only two pointers, *first and trail*.
2. Assume that we have a list of integers as in Example 4.2 Create a function that searches for an integer, num. If num is in the list, the function should return a pointer to the node that contains num. Otherwise it should return NULL.

ADDITIONAL LIST OPERATIONS

Operations For Circularly Linked Lists

By keeping a pointer last to last node in the list rather than to the firs, we are able to inst an element at both the front and end with ease.

Program. Inserting at the front of a list

```
Int length ( list pointer last)
{ / * find the length of the circular list last */
        List pointer temp;
        Int count = 0;
        If ( last) {
                Temp = last;
                Do      {
                        Count + + ;
                        Temp = temp→link ;
                } While (temp ! = last) ;
        }
}
```

```
        return count;
}
```

DOUBLY LUNKED LISTS AND STORAGE MANAGEMENT:

Double linked list is a two- way list, which can be traversed in two directions, in the usual forward direction from the beginning of the list to the end or in the backward direction from the end of the list to the beginning.

A node in a doubly linked list has at least 3 fields, say DATA, LLINK (left link) and RLINK (right link). A double linked list may or may not be circular. A sample doubly linked circular list with 3 nodes is given in figure.

Algorithm DDLETE deletes node X from list L.

```
Procedure DDLETE(X,L)
if X= L then call No_ MORE_NODES
* L is a list with at one node*
RLINK(LLINK(X)) ←RLINK(X)
LLINK (LLINK(X)) ← LLINK(X)
call RET(X)
end DDLETE
procedure DINSERT(P,X)
*insert node P to the right of node X*
LLINK (P) ← X *set LLINK and RLINK fields of node P*
RLINK(P) RLINK(X)
RLINK(X) P
end DINSERT
```

UNIT – III

TREES

INTRODUCTION

A tree is a finite set of one or more nodes such that

i. There is a specially designated node called the root

ii. The remaining nodes are portioned into n≥ 0 disjoint tress T_1, T_2,… T_n where each of these sets is a tree. T_1,… T_n, are also called subtrees of the root.

In the above definition a tree is defined in terms of trees. So, obviously it is a recursive definition.

There are many terms which are often used when referring to trees. They are many terms which are often used when referring to trees. They are defined one by one in the following paragraphs.

A node stands for the item of information plus the branches to other items. The tree in the following figure has 13 nodes, each item of data being a single letter for convenience. The root is „A‟ and trees are normally drawn with the root at the top. The number of subtrees of a node is called its degree. So, in the tree below, the degree of „A‟ is 3 of c is 1 and of F is Zero. Nodes that have degree zero are called leaf or terminals nodes. Therefore,
{K,L,F,G,M,I,J} is the set of leaf nodes. The nodes with non- zero degrees are called non terminals. The roots of the subtrees of a node, say X, are the children, of X. X is the parent of its children. Thus the children of A are B, C and D. Also H,I and J are iblings. This terminology care be extended so that, we can ask for the grandparent of a node which is the parent of the parents of the node. For example grant of M is D.

The degree of a maximum degree of the nodes in the tree. For example the degree of the above tree is 3. The ancestors of a node are all the nodes along the path from the root to that node. The ancestors of M are A, D and H. the level of a node is defined by in initially letting the root be at level 1. If a node is at level „n‟, then its children are at level „n+1‟. The weight or depth of a tree is defined to be the maximum level of any node in the trees.

A forest is a set of n ≥ 0 disjoint trees. The nation of a forest is closely related to that of a tree, because, if we remove the root of a tree we get a forest.

BINARY TREE:

A binary tree is a finite set of nodes which is either empty or consists of a root and two disjoint trees called the left subtree and the right sub tree.

Example:

The maximum number of nodes on level „I" of a binary tree is 2^{i-1}, i> =1.

The maximum number of nodes in a Binary Tree of depth K is 2^k-1, K> = 1.

$$\sum_{i=1}^{k} 2i - 1 = 2k - 1$$

BINARY TREE TRAVERSAL:

When traversing a binary tree we want to treat each node and its subtrees in the same fashion. If we let L, D, R, stand for moving left, printing the data, and moving right when at a node then there are six possible combinations of traversal: LDR, LRD, DLR, DR, RDL, and RLD. IF we adopt the convention that we traverse left before right then only three traversals remain: LDR, LRD and DLR. To these we assign the names inorder, postorder and preorder and preorder because there is a natural correspondence between these travels and producing the infix, postfix and prefix forms of an expression.

```
Procedure INORDER(T)

*T is a binary tree where each node has three fields L- CHILD. DATA, RCHILD* if T f 0
then [call INORDER (LCHILD(T))

print (DATA(T))

 call ( INORDER (RCHILD(T))]

end INORDER
```

Output : A/B * *8 C* D + E

procedure PREORDER(T)

T is a binary tree where each node has three fields L-CHILD. DATA, RCHILD if T f0 then [print (DATA(t))

call PREORDER(LCHILD(T))

call PREORDER (RCHILD(T))]

end PREODER

Output : + */A * * BCDE

procedure POST ORDER(T)

T is a binary tree where each node has three fields L-CHILD.DATA,RCHILD

if I f0 then [call POASTORDER (LCHILD(T))

call POSTORDER (RCHILD(T))

print (DATA(T))]

end POSTORDER

Output : A B C * * / D * E +

Iterative Inorder Treaversal

We can develop equivalent iterative functions for teunsive traversal. Let us take inorder traversal as an example. To simulate the recursion, we must create our own stack.

A node that has no action indicates that the node is added to the stack, while a node that has a *Printf* action indicates that the node is removed from the stock. Notice that the left nodes are stacked until a null node is reached, the node is then removed from the stack, and the node‟s right child is stacked. The traversal then continues with the left child. The traversal is complete when the stack is empty.

Analysis of iterInorder: Let n be the number of nodes in the tree. If we consider the action of iterInorder, we note that every node of the tree is placed on and removed from the stack exactly once. So, if the number if nodes in the tree is n, the time complexity is O (n). The space requirement is equal to the depth of the tree which is O(n).

```
Void iterInorder (treepointer node)

{

        int top = -1; /* initialize stack */
trepointer stack [MAZ_STACK_SIZE];
for (; ;) {
for (; node ; node = node→leftChild)
        push (node); / * add to stack * /
node = pop( ) ; / * delete from stack * /
if ( ! node) break; / * empty stack * /
printf ( " % d", node →data);
node = node→right Child;
        }
}
```

THREADED BINARY TREES:

The idea is to replace the null links by pointers, called threads, to other nodes in the tree If the RCHILD(P) is normally equal to zero, we will replace it by a pointer to the node which would be printed after P when traversing the tree in inorder. A null LCHILD link at node P is replaced by a pointer to the node which immediately precedes node P in inorder.

The tree T has 9 nodes and 10 null links which have been replaced by threads. If we traverse T in inorder the nodes will be visited in the order HDIBEAFCG.

For example node E has a predecessor thread which points to B and a successor thread which points to A.

In the memory representation we must be able to distinguish between threads and normal pointers. The is done by adding two extra one bit fields LBIT and RBIT.

LBIT(P) =1 if LCHILD (P) is a normal pointer

LBIT (P) = 0 if LCHID(P) is a thread

RBIT(P) = 1 if RCHILD(P) is a normal pointer

RBIT (P)= 0 if RCHILD (P) is a thread

Figure 5-1) : Threadec

```
procedure INSUC(X)

*find the inorder st

S← RCHIULD (X)

if RBIT(X) = I then

S← LCHILD(S) * uni

end ]

return(S)

end INSUC

procedure TINORDER (T)

*travers the threaded binary tree, T, in order*

HEAD ← T

loop

if T = HEAD then return

print (DATA(T))

forever

end TINORDER

procedure INSERT_RIGHT (S,)

*insert node T as the right child of S in a threaded binary tree*

RCHILD (T) ← RCHILD(S) : RBIT(T) ← RBIT(S)

LCHILD (T) ← S: LBIT (T) ← 0 * LCHILD (T) is a thread*

RCHILD (S) ← T: RBIT (S) ← 1 * attach node T to S*

if RBIT(T) ← I [W INSUCT(T) * S had a right child *

LCHILD(W) ← T]
```

end INSERT_ RIGHT

1. Write out the inorder, preorder, postorder, and level-order treversals for the binary trees of Figure 5.10.
2. Do Exercise 1for the binary tree of Figure 5.11.
3. Do Exercise 1for the binary tree of Figure 5.15.

EXERCISES

1. Draw the binary tree of Figure 5.15, showing its threaded representation.
2. Write a function, insertLeft, that inserts a new node, child, as the left child of node parent in a threaded binary tree. The left child pointer of parent becomes the left child pointer of child.
3. Write a function that traverses a threaded binary tree in postorder. What are the time and space requirements of your method?

1. Suppose that each node in a binary search tree also has the field leftsize as described in the text. Write a function to insert a pair into such a binary search tree. The complexity of your function should be O(h), where *h* is the height of the search tree. Show that is the case.
2. Any algorithm that merges together two sorted lists of size n and m, respectively, must make at least n + m – 1 comparisons in the worst case. What implications does this result have on the time complexity of any comparison- based algorithm that combines two binary search trees that have n and m pairs, respectively?

BINARY SEARCH TREE ALGORITHM:

Definition:

A binary search tree T is a binary tree; either it is empty or each node in the tree contains an identifier. All identifiers in the left subtree of T are less (Numerically or Alphabetically) than the identifier in the rot node T. The left and Right subtree are also binary search tree.

Function Binary search(keys, X Low, High Result) Given a vector keys (an input parameter) whose elements are in ascending order, this procedure searches the vector for a given element whose value is given by the input parameter X and returns in the position in the vector to the calling program in the parameter result. Low & High are input parameter defining the current search interval. Initially low denotes the first subscript of vector keys, so that keys (Low) is the smallest value and High denotes the last subscript of the vector keys, so that keys (High) is the largest value middle denotes the midpoint of the interval.

```
1. Is desired element absent If low High and keys (Low) f X then wirte (search is un successful)
Result ←0

Return

 end if

2. Obtain position of midpoint of interval.

MIDDLE = (Low + High) div2

3. Compare

if keys (middle) = x

        then write ( „search is successful)

result ← middle

else if keys ( middle) < X

then call Bin – search recursive

else( Key, X, middle + 1 Low Result)

else ( Key, X, middle – 1 High Result)

end if

 end if
```

BINARY TREE SEARCH:

Function BTS (ROOT, item)

given the input parameters root and item, as described previously, this function performs a search operation on the tree structure. The node structure (NODE) contains a left pointer (LPTR) an item description (INFO) and a right pointer (RPTR). The is initially invoked with the head node of a tree and recursively searches the tree for a node whose info field matches item

Then return (Root) end if

```
*node found*
if item = INFO(ROOT)
if ITEM < INFOR (ROOT)
then return(root)         * node not in tree *
else return * BTS ( LPTR,(ROOT), ITEM))*
end if * search left sub tree*
end if
*if search Right subtree*
if ITEM > INFOT(ROOT)
then if RPTR (ROOT) – NULL
then return (ROOT)
else Return (BTS (RPTR, (ROOT), ITE)
end if
end if
```

FORESTS

A *forests* is a set of n ≥0 disjoint trees.

A Forest is obtained, when the root of a tree is removed

Transforming a Forest into a Binary Tree

To transform a forest into a single binary tree, we first obtain the binary tree representation of each of the trees in the forest and then link these binary trees through the right child field of the root nodes. Using this transformation, the forest or Figure 5.35 becomes the binary tree of Figure

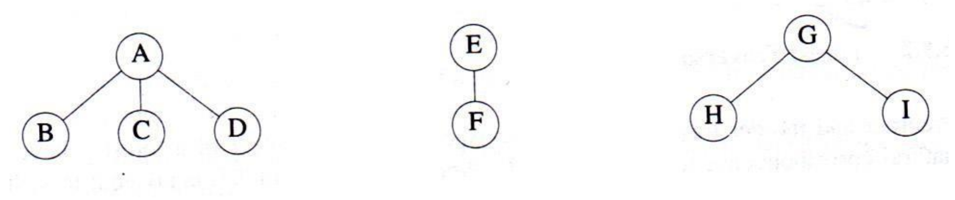


Fig : A – three tree forest

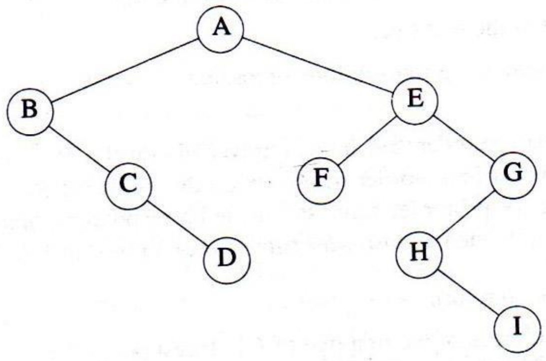


Fig : B – Binary tree representation of forest of fig. A

Definition: If T_1…, T_n is a forest of trees, then the binary tree corresponding to this forest, denoted by B (T_1,..,T_n),

1) Is empty if n= 0
2) Has root equal to root (T_1); has left subtree equal to B(T_{11}, T_{12},…,T_{1m}), whereT_{11},…, are the subtrees of root (T_1); and has right subtree B (T_2,…, T_n).

UNIT - IV

GRAPHS

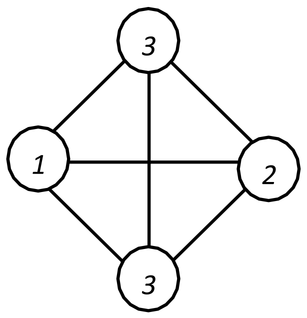
The Graph Abstract Data Type

Introduction

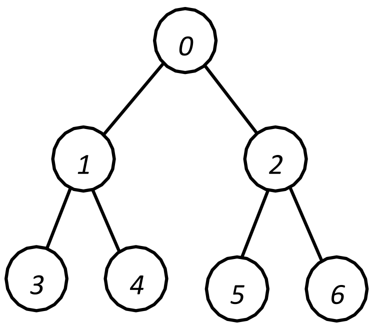
Graphs have been used in a wide variety of applications. Some of these applications are: analysis of electrical circuits, finding shortest routes, project planning, identification of chemical compounds, statistical mechanics, genetics, cybernetics, linguistics, social sciences, and so on. Indeed, it might well be said that of all mathematical structures, graphs are the most widely used.

Definitions

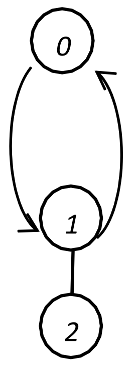
A graph, G, consists of two sets, V and E. V is a finite, nonempty set of vertices. E is a set of pairs of vertices; these pairs are called edges. V(G) and E(G) will represent the sets of vertices and edges, respectively, of graph G. We will also write G = (V, E) to represent a graph. In an undirected graph the pair of vertices representing any edge is unordered. Thus, the pairs (u,v) and (v,u) represent the same edge. In a directed graph each edge is represented by a directed pair <u, v>; u is the tail and v the head of the edge$^+$. Therefore, <v,u> and <u,v> represent two different edges. Figure 6.2 shows three graphs: G_1, G_2 and G_3. The graphs G_1 and G_2 undirected. G_3 is a directed graph.



G 1



G2



G3

The set representation of each of these graph is

$V(G_1) = \{0,1,2,3\}$; $E(g1) = \{(0,1), (0,2), (0,3), (1,2), (1,3), (2,3)\}$

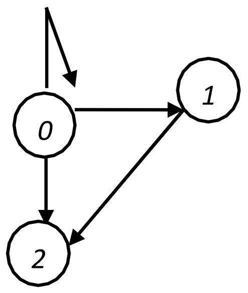
$V(G_2) = \{0,1,2,3,4,5,6\}$; $E(G2) = \{(0,1), (0,2), (1,3), (1,4), (2,5), (2,6)\}$

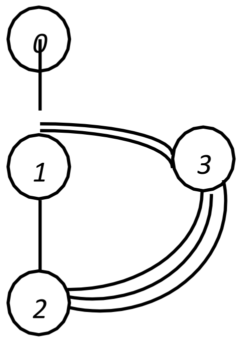
$V(G_3) = \{0,1,2\}$; $E(G3) = \{<0,1>, <1,0>, <1,2>\}$.

Notice that the edges of a directed graph are drawn with an arrow from the tail to the head. The graph G_2 is a tree; the graphs G_1 and G_3 are not.

Since we define the edges and vertices of a graph assets, we impose the following restrictions on graphs:

1) A graph may not have an edge from a vertex, v, back to itself. That is, edges of the form (v,v) and <v, v> are not legal. Such edges are known as self edges or self loops. If we permit self edges, we obtain a data object referred to as a graph with self edges. An example is shown in Figure 6.3(a).





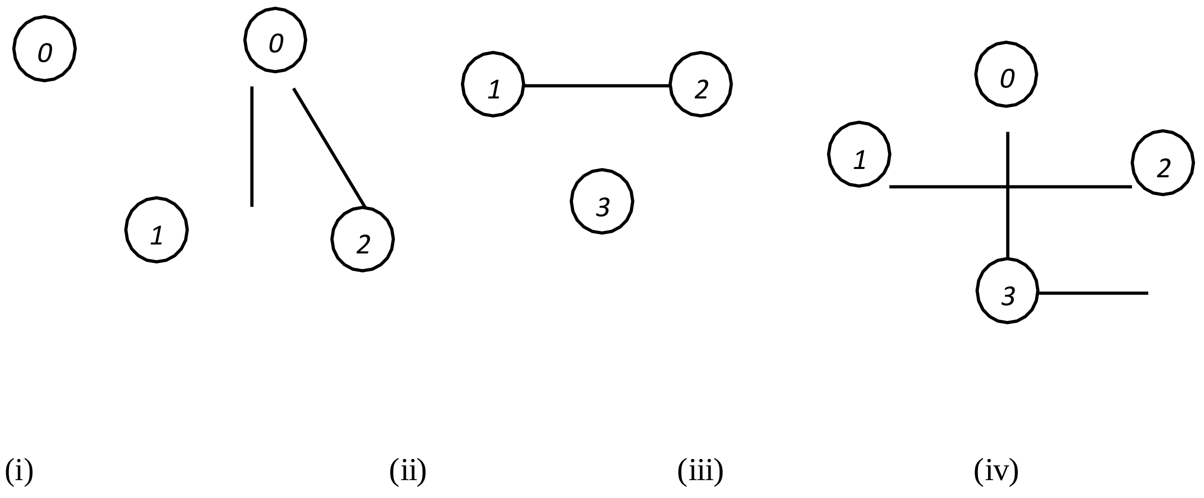
(a) Graph with a self edge b) Multigraph

2) A graph may not have multiple occurrences of the same edge. If we remove this restriction, we obtain a data object referred to as a multigraph (see Figure 6.3 (b)).

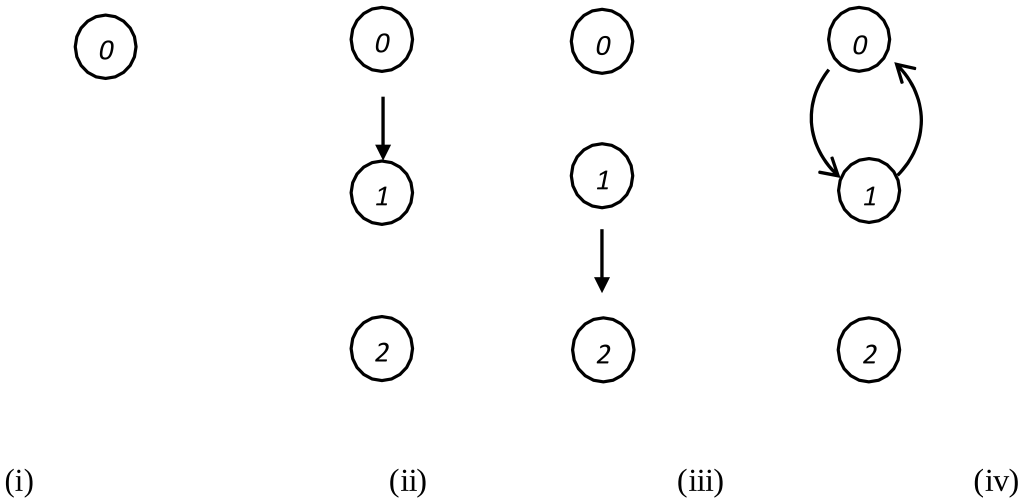
The number of district unordered pairs (u,v) with $u \neq v$ in a graph with n vertices is n(n-1)/2. This is the maximum number of edges in any n-vertex, undirected graph. An n- vertex, undirected graph with exactly n(n-1)/2 edges is said to be complete. The grapn G_1 of Figure 6.2(a) is the complete graph on four vertices, whereas G_2 and G_3 are not compete graphs. In the case of a directed graph on n vertices, the maximum number of edges in n(n-1)

If (u,v) is an edge in E(g), then we shall say the vertices u and v are adjacent and that the edge (u,v) is incident on vertices u and v. The vertices adjacent to vertex 1 in G_2 are 3, 4 and 0. The edges incident on vertex 2 in G2 are (0,2), (2,5), and (2,6). If <u,v> is a directed edge, then vertex u is adjacent to v, and v is adjacent from u. The edge <u,v> is incident to u and v. In G3, the edges incident to vertex 1 are <0,1>, <1,0>, and <1, 2>.

A subgraph of G is a graph G such that V(G) ⊆ V(G) and E(G) ⊆ E(G). Figure 6.4 shows some of the subgraphs of G_1 and G_3.



(a) Some of the subgraphs of G_1

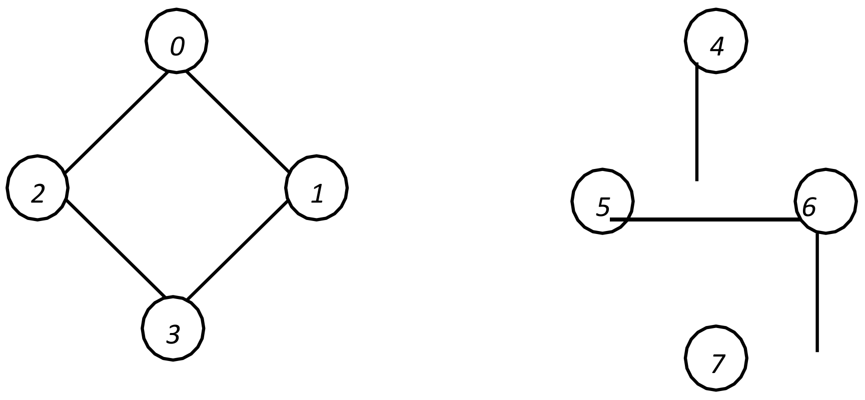


(a) Some of the subgraphs of G_1

A path from vertex u to vertex v in graph G is a sequence of vertices u, i1,i2, …., ik, v such that (u,i1), (i1, i2), …., (I,, v) are edges in E(G). The length of a path is the number of edges on it. A Simple path is a path in which all vertices except possibly the first and last are district. A path such as (0,1) (1,3) (3,2) is also written as 0,1,3,2. Paths 0,1,3,2 and 0,1,3,1 of G1 are both of length 3. The first is a simple path; the second is not. 0,1,2 is a simple directed path in G3. 0,1,2,1 is not a path in G3, as the edge <2, 1> is not in E(G3).

A cycle is a simple path in which the first and last vertices are the same. 0,1,2,0 is a cycle in G1. 0,1,0 is a cycle in G3. For the case of directed graphs we normally add the prefix “directed” to the terms cycle and path.

In an undirected graph. G, two vertices u and v are said to be connected iff there is a path from v to u). An undirected graph is said to be connected iff for every pair of district vertices u an v in V(G) there is a path from u to v in G. Graphs G_1 and G_2 are connected, whereas G4 of Figure 6.5 is not. A connected component (or simply a component), H of an undirected graph is a maximal connected subgraph. By maximal, we mean that G contains no other subgraph that is both connected and properly contains H. G_4 has two components, H_1 and H_2 (see Figure 6.5)



G_4

A tree is a connected a cyclie (ie. has no cycles) graph.

A directed graph G is said to be strongly connected iff for every pair of district vertices u and v in V(G), there is a directed path from u to v and also from v to u. The graph G_3 is not strongly connected as there is no path from vertex 2 to 1. A strongly connected component is a maximal subgraph that is strongly connected. G_3 has two strongly connected components (see Figure 6.6).

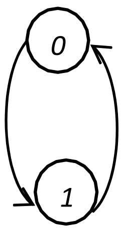


Figure : Strongly connected components of G3

The degree of a vertex is the number of edges incident to that vertex. The degree of vertex 0 in G1 is 3. If G is a directed graph, we define the in-degree of a vertex v to be the number of edges for which v is the head. The out-degree is defined to be the number of edges for which v is the tail. Vertex 1 of G3 has in-degree 1, out-degree 2, and degree 3. If d, is the degree of vertex I in a graph G with n vertices and e edges, then the number of edges is

$$e = \left(\sum_{i=0}^{n-1} d_i\right) / 2$$

We shall refer to a directed graph as a digraph.

ADT Graph is

Objects: a nonempty set of vertices and a set of undirected edges, where each edge is a pair of vertices.

Functions:

for all graph ϵ Graph, v, v1, and v2 ϵ Vertices

Graph Create ( )	::=	return an empty graph.
Graph Insert Vertex (graph, v)	::=	return a graph with v inserted. v has no incident edeges.
Graph Insert Edge (graph, v1, v2)	::=	return a graph with a new edge between V1 and V2
Graph Delete Vertex (graph, v)	::=	return a graph in which v and all edges incident to it are removed.
Graph Delete Edge (graph, v1, v2)	::=	return a graph in which the edge (v1, v2) is removed. Leave the incident nodes in the graph.
Boolean Is Empty (graph)	::=	if (graph== empty graph) return TRUE else return FALSE.
List Adjacent (graph, v)	::=	return a list of all vertices that are adjacent to v.

ADT Abstract data type Graph

The operations in ADT. 6.1 are a basic set in that allow us to create any arbitrary graph and do some elementary tests. In the later sections of this chapter we shall see functions that traverse a graph (depth first or breadth first search) and that determine if a graph has special properties (connected, biconnected, planar).

The three most commonly used: adjacency matrices, adjacency lists, and adjacency multilists.

Adjacency Matrix

Let G = (V, E) be a graph with n vertices, n≥ 1. The adjacency matrix of G is a two. dimensional n x n array, say a, with the property that a [i] [j] = 1 iff the edge (i, j) (<I, j> for a directed graph) is in E(G). a[i] [j] = 0 if there is no such edge in G. The adjacency matrices for the graphs G1, G3 and G4 are shown in Figure 6.7 The adjacency matrix for an undirected graph is symmetric, as the edge (i, j) is in E(G) iff the edge (j,i) is also in E(G). The adjacency matrix for a directed graph may not be symmetric (as in the case for G3). The space needed to represent a graph using its adjacency matrix is n2 bits. About half this space can be saved in the case of undirected graphs by storing only the upper or lower triangle of the matrix.

$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 2 & 1 & 1 & 0 & 1 \\ 3 & 1 & 1 & 1 & 0 \end{array}
\qquad
\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 2 & 0 & 0 & 0 \end{array}
\qquad
\begin{array}{c|cccccccc} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 3 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 7 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{array}$$

From the adjacency matrix, one may readily determine if there is an edge connecting any two vertices i and j. For an undirected graph the degree of nay vertex i is its row sum:

$$\sum_{j=0}^{n-1} a[i][j]$$

For a directed graph the row sum is the out-degree, and the column sum is the in-degree.

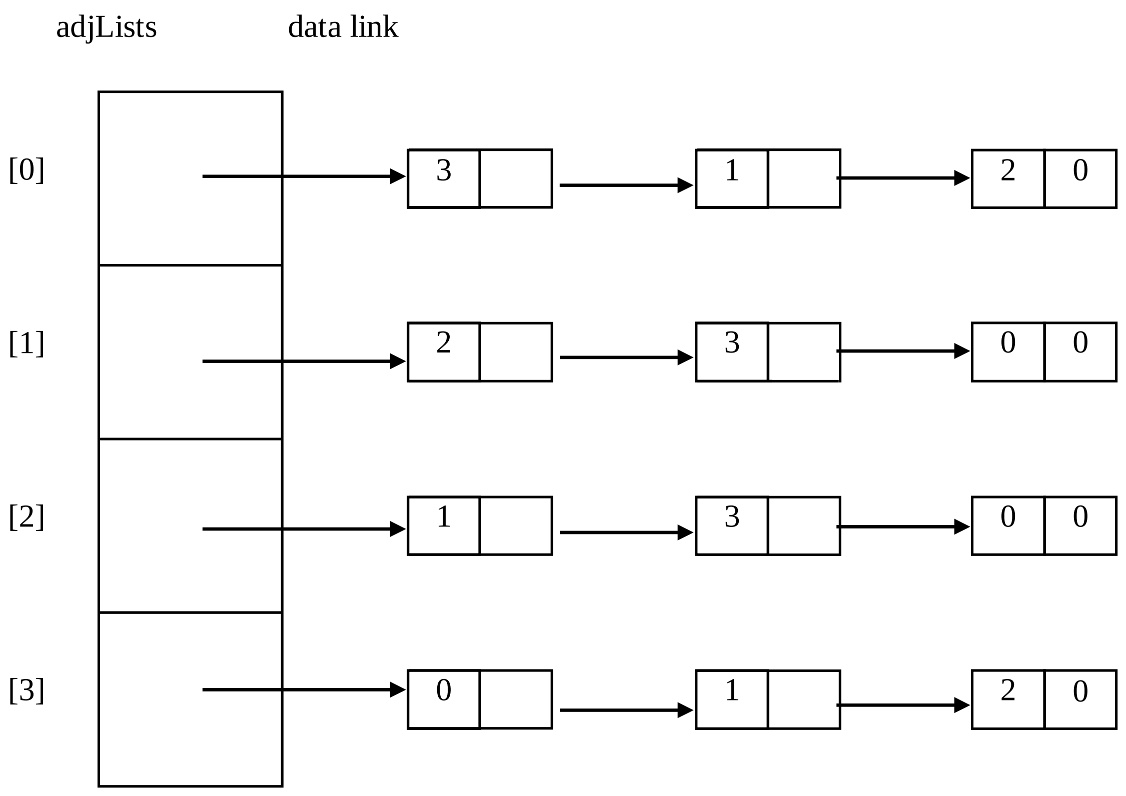
Adjacency Lists

In this representation of graphs, the n rows of the adjacency matrix are represented as n chains (though sequential lists could be used just as well). There is on chain for each vertex in G. The nodes in chain I represent the vertices that are adjacent from vertex i. The data field of a chain node stores the index of an adjacent vertex. The adjacency lists for G1, G3 and G4 are shown in Figure 6.8. Notice that the vertices in each chain are not required to be ordered. An array adjlists is used so that we can access the adjacency list for any vertex in O(1) time. adjLists [i] is a pointer to the first node in the adjacency list for vertex i.

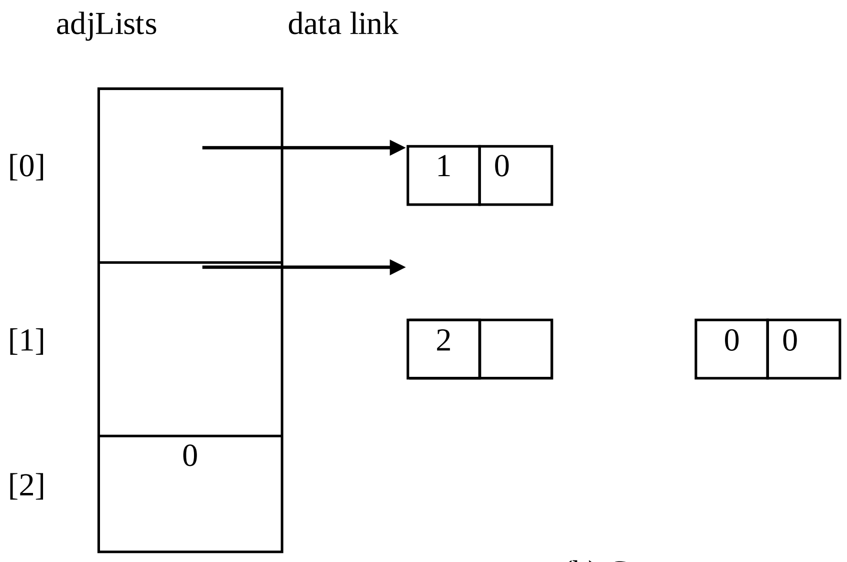
For an undirected graph with n vertices and e edges, the linked adjacency lists representation requires an array of size n and 2e chain nodes. Each chain node has two fields. In terms of the number of bits of storage needed, the node count should be multiplied by logn n for the array positions and log n + log e for the chain nodes, as it takes O(logm) bits to represent a number of value m. If instead of chains, we use sequential lists, the adjacency lists may be packed into an integer array node [n+2e +1]. In one possible sequential mapping, node [i] gives the starting point of the list for vertex I, 0 ≤ < n, and node [n] is set to n + 2e +
1. The vertices adjacent from vertex I are stored in node [i], …, node [i+1] -1, 0 ≤ I < n. Figure 6.9 shows the representation for the graph G4 of Figure 6.5

The degree of any vertex in an undirected graph may be determined by just counting the number of nodes in its adjacency list.

For a digraph, the number of list nodes is only e. The out-degree of any vertex may be determined by counting the number of nodes on its adjacency list. Determining the in-degree of a vertex is a little more complex. If there is a need to access repeatedly all vertices adjacent to another vertex, then it may be worth the effort to keep another set

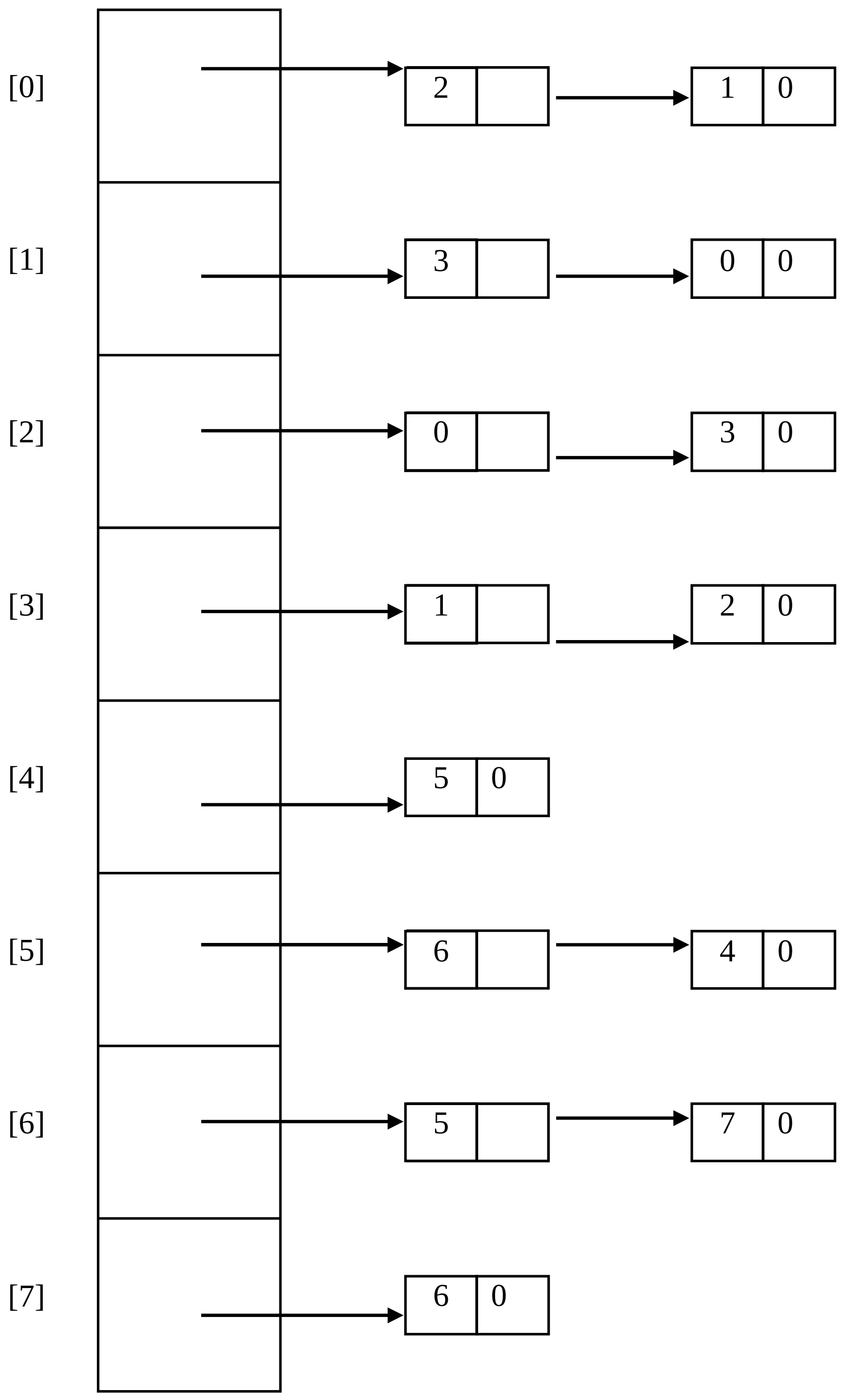


(a) G1



(b) G_3

adjLists



G 4

int nodes [n+2*e+1];

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
9	11	13	15	17	18	20	22	23	2	1	3	0	0	3	1	2	5	6	4	5	7	6

Figure : Sequential representation of graph G4

of lists in addition to the adjacency lists. This set of list, called inverse adjacency lists, will contain one list for each vertex. Each list will contain a node for each vertex adjacent to the vertex it represents (see Figure 6.10)

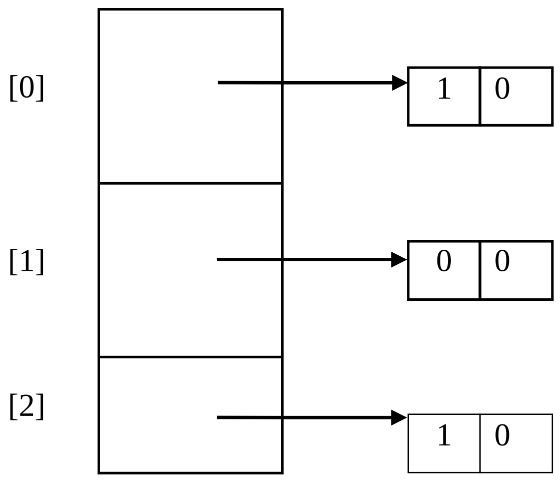


Figure : Inverse adjacency lists for G3 (Figure (c)

Figure 6.11 shows the resulting structure for the graph G3 of Figure 6.2©. The header nodes are stored sequentially. The first two fields in each node give the head and tail of the edge represented by the node, the remaining two fields are links for row and column chains.

Adjacency Multilists

In the adjacency-list representation of an undirected graph, each edge (u,v) is represented by two entries, one on the list for u and the other on the list for v. As we shall see, in some situations it is necessary to be able to determine the second entry for a particular edge and mark that edge as having been examined. This can be accomplished easily if the adjacency lists are actually maintained as multilists (i.e. lists in which nodes

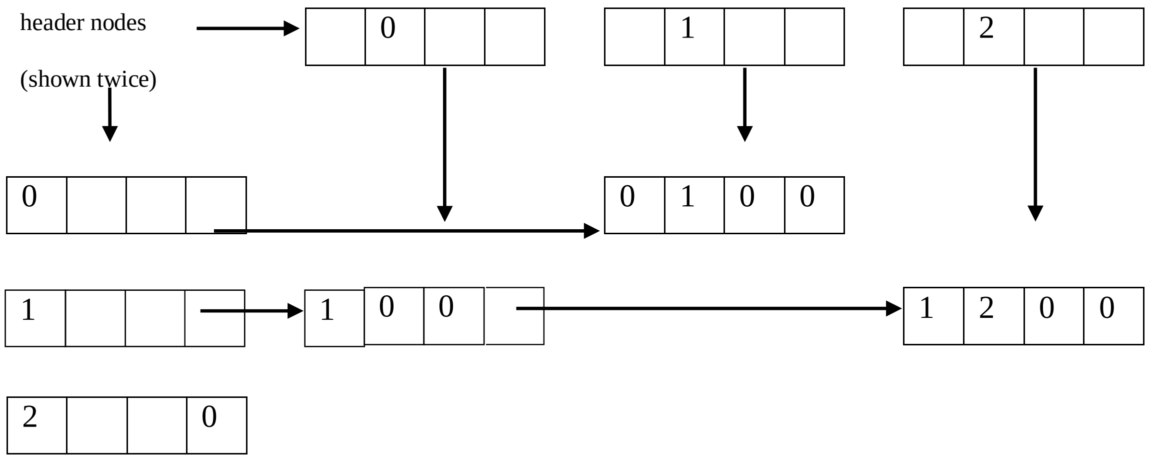


Figure: Orthogonal list representation for G3 of Figure 6.2(c)

may be shared among several lists). For each edge there will be exactly one node, but this node will be in two lists (i.e. the adjacency lists for each of the two nodes to which it is incident). The new node structure is.

m	vertex1	vertex2	link1	link2

where m is a Boolean mark field that may be used to indicate whether or not the edge has been examined. The storage requirements are the same as for normal adjacency lists, except for the addition of the mark bit m. Figure 6.12 shows the adjacency multilists for G_1 of Figure 6.2(a).

Weighted Edges

In many applications, the edges of a graph have weights assigned to them. These weights may represent the distance from one vertex to another or the cost of going from one vertex to an adjacent vertex.

Figure: A diagraph

3. Draw the complete undirected graphs on one, two, three, four, and five vertices. Prove that the number of edges in an n-vertex complete graph is n(n-1)/2.

4. Is the directed graph of Figure 6.15 strongly connected? List all the simple paths.

6. Show that the sum of the degrees of the vertices of an undirected graph is twice the number of edges.

7. a) Let G be a connected, undirected graph on n vertices. Show that G

   must have at least n-1 edges and that all connected, undirected graphs with n-1 edges are trees.

   b) What is the minimum number of edges in a strongly connected digraph on n vertices? What form do such digraphs have?

ELEMENTRY GRAPH OPERATIONS

Given the root node of a binary tree, one of the most common things one wishes to do is to traverse the tree and visit the nodes, in some order. Given an undirected graph G+ (V,E) and a vertex v in V (G) we are two way Depth First Search and Breadth First Search.

Depth First Search:

Depth First Search of an undirected graph proceeds as follows. The start vertex v is visited. Next an unvisited vertex w adjacent to v is selected and a depth first search from w initiated. When a vertex u is reached such that all its adjacent vertices have been visited, we back up to the last vertex visited which has an unvisited vertex w adjacent to it and initiate a depth first search from w. The search terminates when no unvisited vertex can be reached from any of the visited ones. This procedure is best described recursively as in

```
Procedure DFS(v)

*Given an undirected graph G=( V,E) with n vertices and an arrayVISITED (n) initially set to zero, this algorithm visits all vertices reachable from v, G and VISITED are global. *

VISITED(V) ← 1

for each vertex w adjacent to v do

if VISITED (w) = 0 then call DFS(w)
```

```
end

end DFS
```

Breath First Search:

Starting at vertex v and marking it as visited, breadth first search differs from depth first search in that all unvisited vertices adjacent to v are visited next. Then unvisited vertices adjacent to these vertices are visited and so on. A breadth first search beginning at vertex v1. Next vertices v4, v5,v6, and v7 will be visited and finally v8. Algorithm BFS given the details.

```
Procedure BFS(v)

*A breadth first search of G is carried out beginning at vertex y. All vertices visited are marked as VISITED (i) = 1. The graph G and array VISITED are global and VISITED is initialize Q to be empty *Q is a queue*

loop
        for all vertices w adjacent to v do
         then [call ADD1(w,Q); VISITED (w) ← 1]          *mark w as VISITED*\
end

if Q is empty then return
call DELETEQ(v,Q)
forever
end BFS
```

Example: DFS: $v_1,v_2, v_4,v_8,v_5,v_6, v_3, v_7$

BFS: $v_1,v_2,v_3,v_4,v_5,v_6,v_7,v_8$

MINIMUM COST SPANNING TREES

The cost of a spanning tree of a weighted undirected graph is the sum of the costs (weights) of the edges in the spanning tree. A minimum cost spanning tree is a spanning tree of least cost. Three different algorithms can be used to obtain a minimum cost spanning tree of a connected undirected graph. All three use an algorithm design strategy called the greedy method. We shall refer to the three algorithms as Kruskal"s, Prim"s, and Sollin"s algorithms, respectively.

For spanning trees, we use a least cost criterion. Our solution must satisfy the following constraints :

1) we must use only edges within the graph
2) we must use exactly n-1 edges
3) we may not use edges that would produce a cycle.

Shortest – Path Algorithm:

Let G be a directed graph with m nodes v1, v2 … vm suppose G is weighted v, suppose each edge e in G is assigned a non negative number W(e) called weight or length of the edge. e Then G may be maintained in memory by it weight matrix W = W ij defined as follows

W_{ij} = W(e) if there is an edge e from vi, to vj

O if there is no edge from vi vj.

The path matrix P tells us whether or not there are path bit the nodes. Now we want to find a matrix Q which will tell us the length of the shortest path bit the nodes or more exactly , a matrix Q = Wij where Wij = length of a shortest path from vi to vj.

Here we define a sequence of matrices Qo Qi… Qm (analog to the above matrix P1, P2, .. Pm) whose entries are defined as follow.

Qk(I,j) = MIN (Qk-1(i,j),Qk-1(k,j)

The initial matrix Q 0 is the same as the weight matrix w except that 0 win W is replace by D. The final matrix Qm will be desired matrix Q.

Consider the weighted graph G in Figure. Then the weight W of G is as follows we obtain the following matrics Q0, Q2 Q3 and Q 4 = Q

We indicate how the circled entries and obtained.

Q1(4,2) = MIN (Q0(4,2), + q0 (1,2))

Q2(1,3) = a

Q3 (4,2) =4

Q4(3,1) =9

Algorithm:

A weighted graph „G" with M node is are maintenance in memory by its weight matrix A. This algorithm finds a matrix Q such that q (I,j) is the length of a shortest path from node vi to node vj. INFINITY is a very large number and MIN is the minimum value function.

```
1. Repeat for I,J = 1,2 …m * initialize Q*
W (I,I) = o then set
Q( I,J) = INFINITY
else set Q (I,J) = W(I,J) = w( I,J)
end of loop
2. Repeat steps 3 and 4 for K = 1,2….m *updates Q*
3. Repeat step 4 for
I = 1,2 …. m;
1. Repeat for j= 1,2…. m
set Q (i,j) = MIN (Q(i, j), Q(i,k)+ Q(k,j)]
end
end of step 3
end of step 2
exit.
```

Kruskal's Algorithm

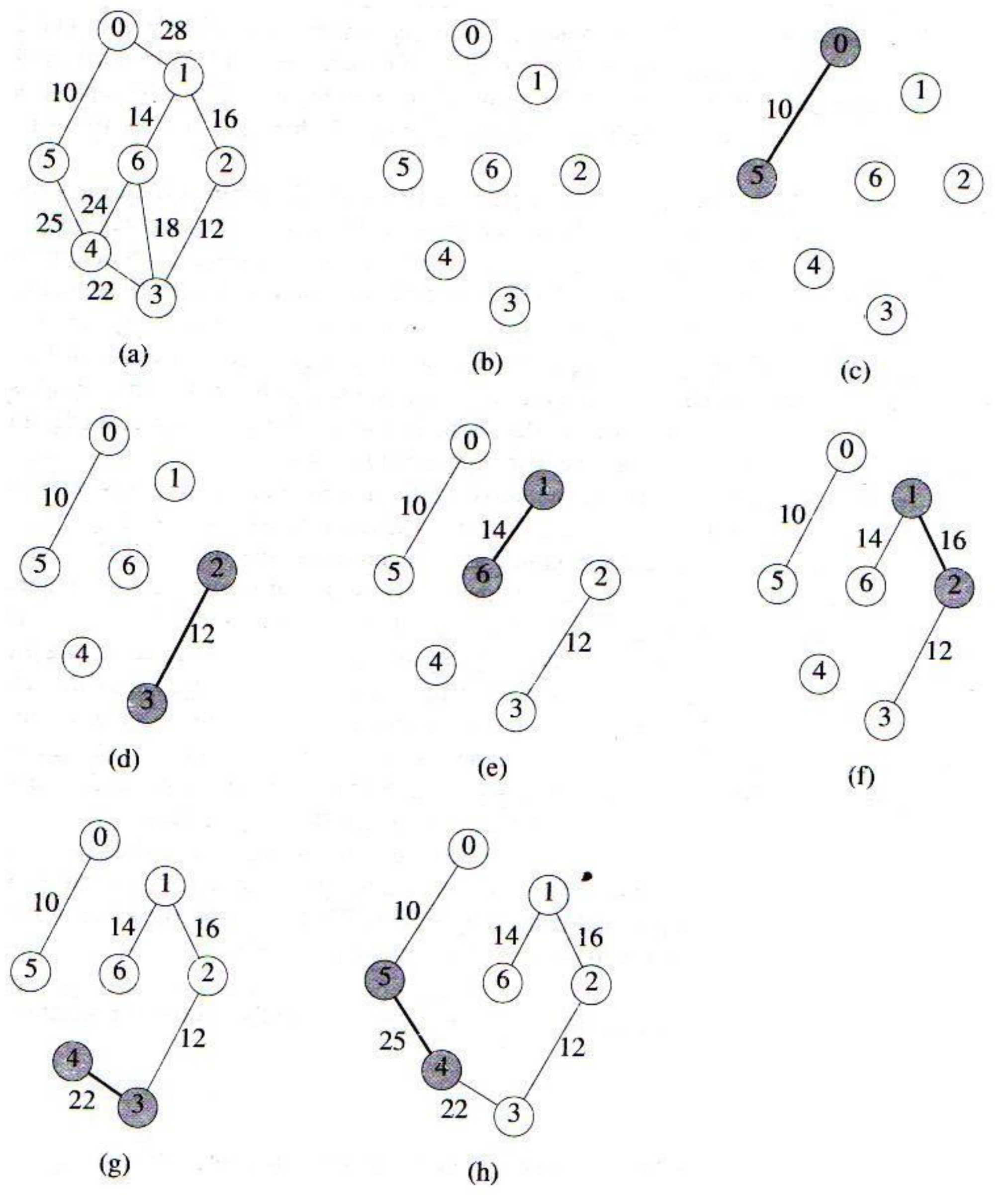
Kruskal"s algorithm builds a minimum cost spanning tree T by adding edges to T one at a time. The algorithm selects the edges for inclusion in T in nondecreasing order of their cost. An edge is added to T if it does not from a cycle with the edges that are already in T. Since G is connected and has n>0 vertices, exactly n-1 edges will be selected for inclusion in T.

We assume that initially E is the set of all edges in G. To implement Kruskal‟s algorithm, we must be able to determine an edge with minimum cost and delete that edge. We can handle both of these operations efficiently if we maintain the edges in E as a sorted sequential list.

We can sort the edges in E in O(e log e) time. Actually, it is not necessary to sort the edges in E as long as we are able to find the next least cost edge quickly. Obviously a min heap is ideally suited for this task since we can determine and delete the next least cost edge in O(log e) time. Construction of the heap itself requires O(e) time.

To check that the new edge, (v, w), does not form a cycle in T and to add such an edge to T. we may use the union-find operations discussed in Section 5.9. This means that we view each connected component in T as a set containing the vertices in that component. Initially, T is empty and each vertex of G is in a different set (see Figure 6.22(b)). Before we add an edge, (v,w) we use the find operation to determine if v and w are in the same set. If they are, the two vertices are already connected and adding the edge (v,w) would cause a cycle. For example, when we consider the edge (3,2) the sets would be {0}, {1, 2, 3}, {5},
{6}. Since vertices 3 and 2 are already in the same set, the edge (3,2) is rejected. The next edge examined is (1,5) Since vertices 1 and 5 are in different sets, the edge is accepted. This edge connects the two components {1,2,3} and {5}. Therefore, we perform a union on these sets to obtain the set {1, 2, 3, 5}.

Since the union-find operations require less time than choosing and deleting an edge (lines 3 and 4), the latter operations determine the total computing time of Kruskal‟s algorithm. Thus, the total computing time is O(e log e).



Edge	Weight	Result	Figure
--	--	initial	Figure 6.22 (b)
(0,5)	10	added to tree	Figure 6.22 (c)
(2,3)	12	added	Figure 6.22 (d)
(1,6)	14	added	Figure 6.22 (e)
(1,2)	16	added	Figure 6.22 (f)
(3,6)	18	discarded	
(3,4)	22	added	Figure 6.22 (g)
(4,6)	24	discarded	

(4,5) (0,1)	25 28	added not considered	Figure 6.22 (h)

Figure : summary of Kruskal"s algorithm applied to Figure (a)

```
T = { } ;
while (T contains less than n-1 edges & & E is not empty) {
    choose a least cost edge (v,w) from E;
    delete (v,w) from E;
    if ((v,w) does not create a cycle in T)
        add (v,w) to T;
    else
        discard (v,w) ;
    }
    if (T contains fewer than n-1 edges)
        print f("No spanning tree/n") ;
```

Prim"s Algorithm

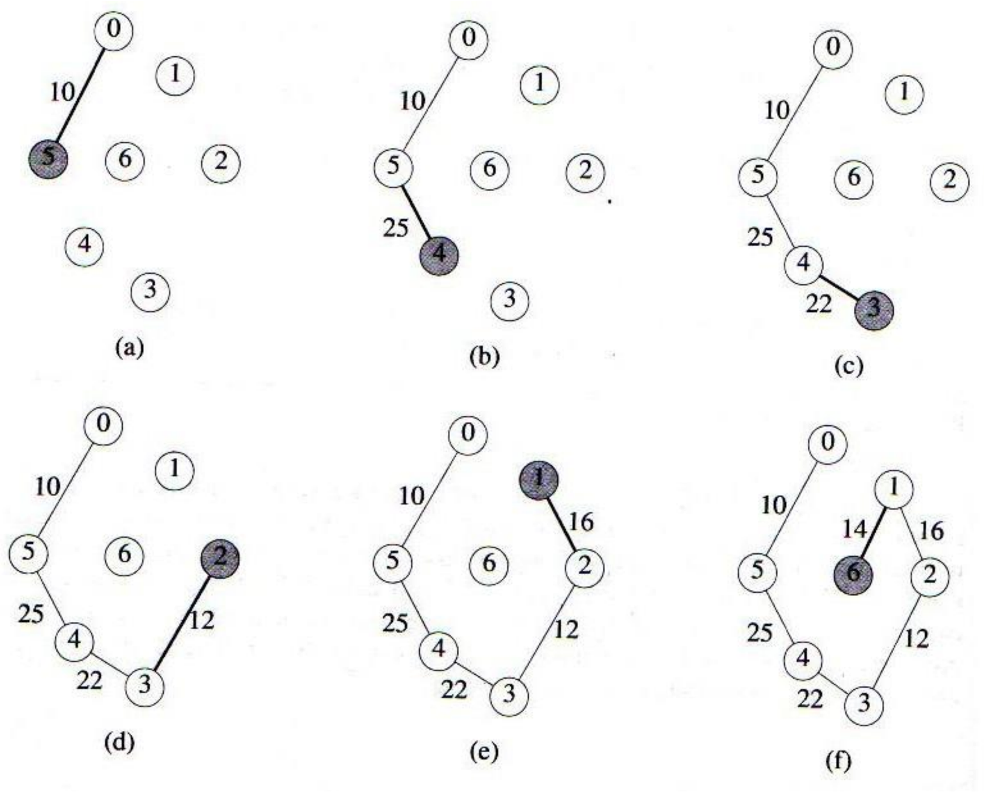
Prim"s algorithm, like Kruskl"s, constructs the minimum cost spanning tree one edge at a time. However, at each stage of the algorithm, the set of selected edges forms a tree. By contrast, the set of selected edges in Kruskal"s algorithm forms a forest at each stage. Prim"s algorithm begins with a tree, T, that contains a single vertex. This may be any of the vertices in the original graph. Next, we add a least cost edge (u,v) to T such that T $\cup$ { (u, v)} is also a tree. We repeat this edge addition step until T contain n-1 edges. To make sure that the added edge does not form a cycle, at each step we choose the edge (u,v) such that exactly one of u or v is in T. Program 6.8 contains a formal description of Prim"s algorithm. T is the set of tree edges, and TV is the set of tree vertices, that is, vertices that are currently in the tree. Figure 6.24 shows the progress of Prim"s algorithm on the graph of Figure 6.22(a).

T = { } ;

TV = { 0 } ; / * start with vertex 0 and no edges * /

while (T contains fewer than n-1 edges) {

let (u, v) be a least cost edge such that u ∈ TV and cost (near(v), v) is minimum over all such choices for near(v). (We assume that cost (v, w) = ∞ if (v, w) ∉ E). At each stage we select v so that cost (near (v), v) is minimum and v∉ TV. Using this strategy we can implement Prim‟s algorithm in O(n^2), where n is the number of vertices in G. Asymptotically faster implementations are also possible.

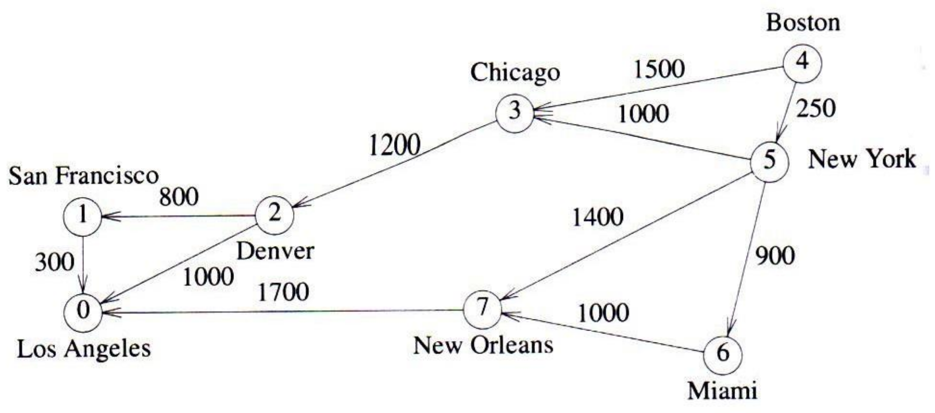


We not consider the general case when some or all of the edges of the directed graph G may have negative length. To see that function shortespath (Program 6.9) does not necessarily give the correct results on such graphs, consider the graph of Figure 6.29. Let V = 0 be the source vertex. Since n=3, the loop of lines 7 to 14 is iterated just once; u=2 in line 8, and no changes are made to dist. The function terminates with dist [1] = 7 and dist [2] = 5. The shortest path from 0 to 2 is 0, 1, 2. This path has length 2, which is less than the computed value of dist [2].

When negative edge lengths are permitted, we require that the graph have no cycles of negative length. This is necessary so as to ensure that shortest paths consist of a finite number of edges. For example, consider the graph of Figure 6.30. The length of the shortest path from vertex 0 to vertex 2 is - ∞ as the length of the path.

0, 1, 0, 1, 0, 1, . . ., 0, 1, 2

can be made arbitrarily small. This is so because of the presence of the cycle 0, 1, 0 which has a length of -1.



(a) Digraph

$$\begin{array}{c|cccccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
0 & 0 & & & & & & & \\
1 & 300 & 0 & & & & & & \\
2 & 1000 & 800 & 0 & & & & & \\
3 & & & 1200 & 0 & & & & \\
4 & & & & 1500 & 0 & 250 & & \\
5 & & & & 1000 & & 0 & 900 & 1400 \\
6 & & & & & & & 0 & 1000 \\
7 & 1700 & & & & & & & 0
\end{array}$$

(b) Length – adjacency matrix

Figure : Digraph for Example

When there are no cycles of negative length, there is a shortest path between any two vertices of an n-vertex graph that has at most n-1 edges on it. To see this, observe that a path that has more than n-1 edges must repeat at least one vertex and hence must contain a cycle. Elimination of the cycles from the path results in another path with the same source and destination. This path is cycle-free and has a length that is no more than that of the original path, as the length of the eliminated cycles was at least zero. We can use this observation on the maximum number of edges on a cycle-free shortest path to obtain an algorithm to determine a shortest path from a source vertex to all remaining vertices in the graph.

As in the case of function shortest Path (Program 6.9) We shall compute only the length, dist [u], of the shortest path from the source vertex v to u. An exercise examines the extension needed to construct the shortest paths.

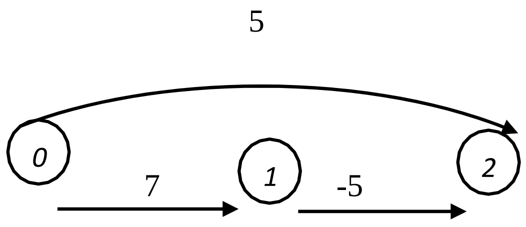


Figure : Directed graph with a negative – length edge.

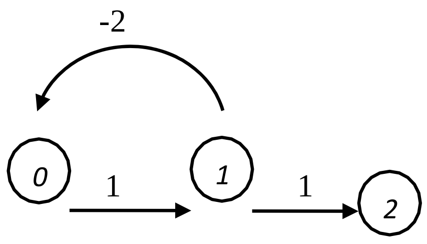


Figure: Directed graph with a cycle of negative length

Let dist1 [u] be the length of a shortest path from the source vertex v to vertex u under the constraint that the shortest path contains at most l edges. Then, $dist^1$ [u] = length [v] [u], $0 \leq u < n$. As noted earlier, when there are no cycles of negative length, we can limit our search for shortest paths to paths with at most n-1 edges. Hence, distn-1 [u] is the length of an unrestricted shortest path from v to u.

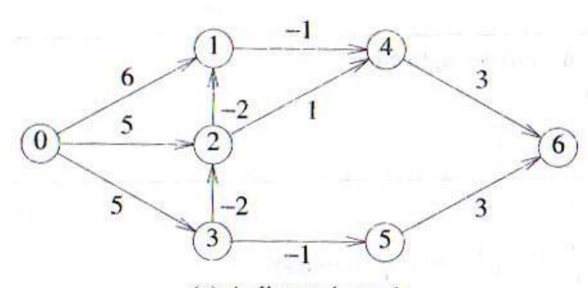
Our goal then is to compute $dist^{n-1}$ [u] for all u. This can be done using the dynamic programming methodology. First, we make the following observations:

1) If the shortest path from v to u with at most k, k > 1, edges has no more than $^{k-1}$ edges, then $dist^{k}$[u] = distk-1[u].
2) If the shortest path from v to u with at most k, k>1, edges has exactly k edges, then it is comprised of a shortest path from v to some vertex j followed by the edge <j, u>. The path from v to j has k-1 edges, and its length is $dist^{k-1}$ [j]. All vertices I such that the edge <I, u> is in the graph are candidates for j. Since we are interested in a shortest path, the i that minimizes dist k-1 [i] + length [i] [u] is the correct value for j.

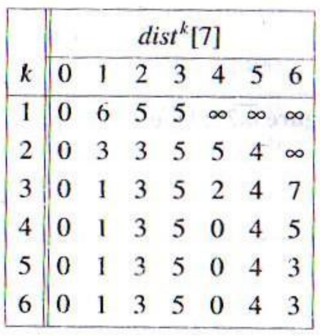
These observations result in the following recurrence for dist:

$dist^{k}[u] = \min \{dist^{k-1}[u], \min \{dist^{k-1}[i] + length\ [i]\ [u]\}\}$

This recurrence may be used to compute $dist^{k}$ from $dist^{k-1}$ for k=2,3, …..n-1.



(a) A directed graph



	$dist^k[7]$						
k	0	1	2	3	4	5	6
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	∞
3	0	1	3	5	2	4	7
4	0	1	3	5	0	4	5
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3

(b) $dist^k$

All Pairs shortest Paths

In the all-pairs-shortest-path problem we must find the shortest paths between all pairs of vertices, v1, vj, i≠ j. We could solve this problem using shortestpath with each of the vertices in V(G) as the source. Since G has n vertices and shortestpath has a time complexity of $O(n^2)$, the total time required would be $O(n^3)$. However, we can obtain a conceptually simpler algorithm that works correctly even if some edges in G have negative weights. (We do require that G has no cycles with a negative length). Although this algorithm still has a computing time of $O(n^3)$, it has a smaller constant factor. This new algorithm used the dynamic programming method.

We represent the graph G by its cost adjacency matrix with cost [i] [j] = 0, i =j. If the edge <I, j>, i ≠ j is not in G, we set cost [i] [j] to some sufficiently large number using the same restrictions discussed in the single source problem. Let A^k [i] [j] be the cost of the shortest path form i to j, using only those intermediate vertices with an index ≤ k. The cost of the shortest path from i to j is A^{n-1} [i] [j] as no vertex in G has an index greater than n-1. Further A^{-1} [i] [j] = cost [i] [j] since the only i to j paths allowed have no intermediate vertices on them.

The basic idea in the all pairs algorithm is to begin with the matrix A^{-1} and successively generate the matrices A^0, A^1, A^2, …., A^{n-1}. If we have already generated A^{k-1}, then we may generate A^k by realizing that for any pair of vertices i, j one of the two rules below applies.

1) The shortest path from i to j going through no vertex with index greater than k does not go through the vertex with index k and so its cost is A^{k-1} [i] [j]

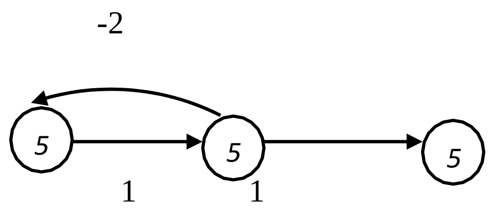
2) The shortest such path does go through vertex k. Such a path consists of a path from i to k followed by one from k to j. Neither of these goes through a vertex with index greate than k-1. Hence, their costs are Ak-1 [i] [j] and A^{k-1} [k] [j].

These rules yield the following formulas for Ak [i] [j]:

Ak [i] [j] = min {A^{k-1}[i] [j], A^{k-1} [i] [k] + Ak-1 [k] [j]}, k≥0

and

A-1 [i] [j] = cost [i] [j]



(a) Directed graph

$$\begin{pmatrix} 0 & 1 & \infty \\ -2 & 0 & 1 \\ \infty & \infty & 0 \end{pmatrix}$$

(b) A^{-1}

```
void allcosts (int cost [ ] [MAX – VERTICES],

                    int distance [ ] [MAX-VERTICES], int n)

     { / * compute the shortest distance from each vertex

     to every other, cost is the adjacency matrix,

     distance is the matrix of computed distances */

int I, j, k;

for (I = 0 ; i < n ; I + +)

     for (j = 0; j < n; j + +)

          distance [i] [j] = cost [i] [j] ;

     for (k = 0; k < n; k ++)

          for (i = o; j < n ; i ++)

              for (j = 0 ; j < n ; j + +)

                     if (distance [i] [k] + distance [k] [j] <

                                                  distance [i] [j])

                          distance [i] [j] =

                          distance [i] [k] + distance [k] [j] ;
```

Program 6.12 : All pairs, shortest paths function

Analysis of all costs: This algorithm is especially easy to analyze because the looping is independent of the data in the distance matrix. The total time for all costs is O(n^3)

EXERCISES

1. Let T be a tree with root v. The edges of T are undirected, Each edge in T has a nonnegative length. Write a C function to determine the length of the shortest paths from v to the remaining vertices of T. Your function should have complexity O(n), where n is the number of vertices in T. Show that this is the case.

2. Let G be a directed, acyclic graph with n vertices. Assume that the vertices are numbered 0 through n-1 and that all edges are of the form <i, j>, where i < j. Assume that the graph is available as a set of adjacency lists and that each edge has a length (which may be negative) associated with it. Write a C++ function to determine the length of the shortest paths form vertex 0 to the remaining vertices. The complexity of your algorithm should be O(n+e), where e is the number of edges in the graph, Show that this is the case.

9. Using the directed graph of Figure 6.36, explain why shortestpath will not work properly. What is the shortest path between vertices 0 and 6?

UNIT – V

Chapter 7

Sorting

Motivation

We use the term list to man a collection of records, each record having one or more fields. The fields used to distinguish among the records are known as keys.

One way to search for a record with the specified key is to examine the lit of records in left-to-right or right-to-left order. Such a search is known as a sequential search.

The data type of each record is element and each record is assumed to have an integer field key. Program 7.1 gives a sequential search function that examines the records in the list a [l :n] in left-to-right order.

```
int seqsearch element a [ ], int k, int n )

{ / * search all : n]; return the least i such that

    a [i]. key = k ; return 0, if k is not in the array * /

    int i ;

    for (I = 1 ; i < = n & & a [i], key ! = k ; i + +)

        ;

    if (i > n) return 0 ;

    return i ;

}
```

If a [l : n] does not contain a record with key k, the search is unsuccessful.

When all keys are district and a [i] is being searched for, i key comparisons are made. So, the average number of comparisons for a successful search is

$$\left(\sum_{1 \le i \le n} i\right) /n=(n+1)/2.$$

It is possible to do much better than this when looking up phone numbers. The fact that the entries in the list (i.e., the telephone directory) are in lexicographic order (on the name key) enables one to look up a number while examining only a very few entries in the list. Binary search (see Chapter 1) is one of the better-known methods for searching an ordered, sequential list. A binary search takes only O(logn) time to search a list with n records. This is considerably better than the O(n) time required by a sequential search.

Two important uses of sorting: (1) as an aid in searching and (2) as a means for matching entries in lists. Sorting also finds application in the solution of many other more complex problems from areas such as optimization, graph theory and job scheduling. Consequently, the problem of sorting has great relevance in the study of computing. Unfortunately, no one sorting method is the best for all applications. We shall therefore study several methods, indicating when one is superior to the others.

We characterize sorting methods into two broad categories: (1) internal methods (i.e., methods to be used when the list to be sorted is small enough so that the entire sort can be carried out in main memory) and (2) external methods (i.e. methods to be used on larger lists). The following internal sorting methods will be developed: insertion sort, quick sort, merge sort, heap sort, and radix sort.

In the recursive formation we divide the list to be sorted into two roughly equal parts called the left and the right sublists. These sublists are sorted recursively, and the sorted sublists are merged.

INSERTION (A,N)

(This algorithm sorts the array A with N Elements)

```
        set A (0)= - D ( Initialize sentient element)
        Repeat steps 3 to 5 for K = 2,3,… N
        Set TEMP = A(k) and
PTR = K -1
        Repeat while TEMP < A (PTR)
        Set A(PTR + 1) = A(PTR)
        Moves element forward
        Set PTR = PTR – 1
        end of loop)
        Set A ( PTR+1) = TEMP
```

Insert the elements in proper place

End of step 2 loop

Return

Quick sort:

Let A be list of n data items. Sorting A refers to the operation of rearranging the elements of A so that they are in some logical order. This is an algorithm of the divide and conquer types.

44, 33, 11, 55, 77, 90, 4o, 60, 99, 22, 88, 66

22, 33, 11, 55, 77, 90, 40, 60, 99, 44, 88, 66

22, 33, 11, 44, 77, 90, 40, 60, 99, 55, 88, 66

22, 33, 11,40, 77, 90, 44, 60, 99, 55, 88, 60

22, 33, 11, 40, 44, 90,77, 60, 99,55,88,66

Etc., In Q.S. the division in to two sub tiles is made such that the sorted sub files do not need to be later merged.

Procedure QSORT

\// sort record Rm,…. Rn into no decreasing order on key k, key Km is arbitrarily chose as the control key pointers I & j are used to partition the sub tile so that any time k, # K, & c1 and k 1 ≥ K,P < J and Ki ≥ K, P > j it is assured that

```
K m# Ln+1

If M< n n

Then [ I ← j : m← n+1 : K ← Km

Loop

Repeat j ← i+ 1 until Kj ≥ K

Repeat j← I -1 until Kj # K

If I < j

Then call INTERCHANGE (R(j), (R(j))

Else exit

For ever

CALL INTERCHANGE (R(m), R(j))

CALL Q SORT (M, j-q)
```

CALL Q SORT(J+1, n)

End QSORT

TWO MERGE SORT:

Before looking at the merge sort algorithm to sorts records let us see how one may merge two fiels (X1,…. Xm) and (Xm+1 …. Xn) that are already sorted to get a third file (Zi,…., Zn) that is also sorted. Since this merging scheme is very simple, we directly present the algorithm.

Procedure MERGE(X,1,m,n,Z)

// (X1,….. Xm) and (Xm+1… .... n) are two sorted files with keys X1 # … # Xm +1 #.......# Xn. They are merged to obtain the sorted file (Zi……….Zn) such such that Z1#. ..... # Zn//

i← K← 1; j←m+1 // j,j, and k are position in the thee files//

while I # m and j# n do

if Xi # Xj then [Zk← Xi;i← i + 1]

else [Zk ←Xj ; j← j+1]

k←k+1

end

if i> m then (Zk,…. Zn) ← (X i… ...., Xn)

else (Zk,…..Zn) ← (Xi,...... Xm)

Example -1

Set -1:

6,8,10

Set -2:

3,7, 20

Merge Sort:

3, 6,7,8,10,20

Example – 2

26,5, 77, 1

5,26,1,77

1,5,26,77

Example : The input list (26, 5, 77, 1, 61, 11, 59, 15, 49, 19) is to be sorted using the recursive formulation of merge sort. If the sublist from left ot right is currently to be sorted, then ists two sblists are indexed form left to [(left + right)/2] and from [(left + right)/2] + 1 to rights. The sublist partitioning that takes place is described by the binary tree of Figure 7.5. Note that the sublists begin merged are different from those being merged in mergeSort.

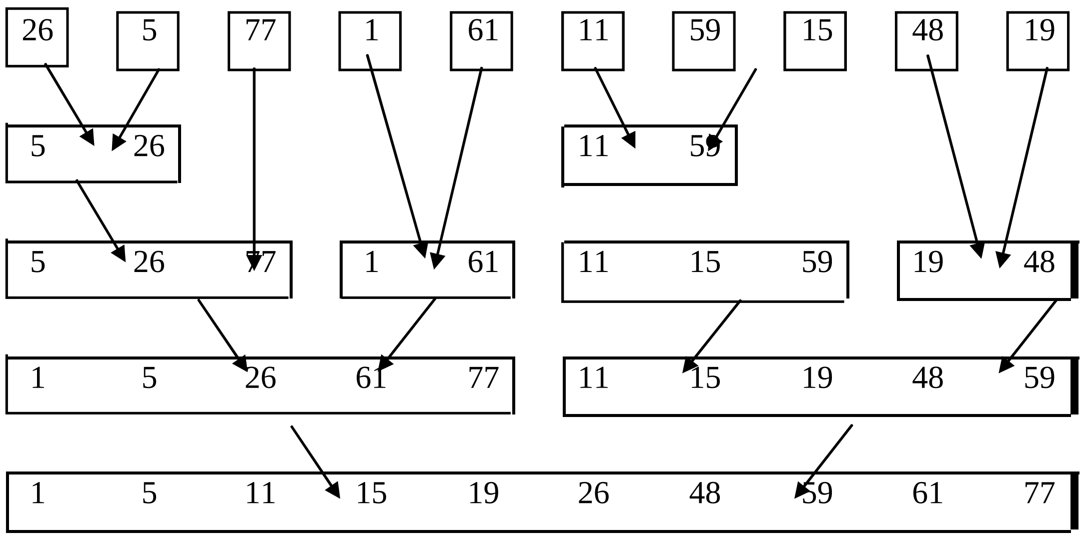


Figure : Sublist partitioning for recursive merge sort

To eliminate the record copying that takes place when merge (Program 7.7) is used to merge sorted sublists we associate an integer pointer with each record. For this purpose, we employ an integer array link [l:n] such that link [i] gives the record that follows record i in the sorted sublist. In cast link [i] = 0, there is no next record. With the addition of this array of links, record copying is replaced by link changes and the runtime of our sort function becomes independent of the size s of a record. Also the additional space required is O(n). By comparison, the iterative merge sort described earlier takes O(snlogn) time and O(sn) additional space. On the down side, the use of an array of links yields a sorted chain of records and we must have a follow up process to physically rearrange the records into the sorted order dictated by the final chain. We describe the algorithm for this physical rearrangement in section.

We assume that initially link [i] = 0, 1 ≤ i ≤ n. Thus, each record is initially in a chain containing only itself. Let start 1 and start 2 be pointers to two chains of records. The records on each chain are in nondecreasing order. Let list Merge (a, link, start1, start2) be a function that merges two chain that is linked in nondecreasing order of key values. The recursive version of merge sort is given by function rmerrmerge sort (Program 7.10) (a, link, 1, n). The start of the chain ordered as described earlier is returned. Function list merge is given in Program.

```
int rmergesort (element a [], in link [], int left, in rights)

{/* a [left : right] is to be sorted, link [i] is initially 0

    for all I, returns the index of the first element in the

    sorted chain */

    if (left > = right) return left;

    int mid = (left + right) / 2;

    return listMerge (a, link,

                rmegeSort (a, link, left, mid),

                        /* sort left half */

                rmergeSort (a, link, mid + 1, right)) ;

                        /* sort right half */
```

Analysis of rmergeSort : It is easy to se that recursive merge sort is stable, and its computing time is O(n log n).

```
int listMerge (element a [ ], int link [ ], int start1, int start2)

(/* sorted chains beginning at start1 and start2,
respectively, are merged; link [0] is used as a

temporary header; returns start of merged chain */

int last1, last2, lastResult = 0;

for (last1 = start1, last2 = start2; last1 & & last1;)

    if (a [last1] < = a [last2)] {

            link [lastResult] = last1 ;

            lastResult = least1; last1 = link [last1];

    }

    else {

            link [lastResult] = last2;

            lastResult = last2; last2 = link [last2];

    }

    /* attach remaining records to result chain */

    if (last1 = = 0) link [lastResult] = last2;

    else link [lastResult] = lst1;

    return link [0] ;

}
```

EXERCISES

1. Write the status of the list (12, 2, 16, 30, 8, 28, 4, 10, 20, 6, 18) at the end of each phase of merge Sort (Program 7.9).
2. Write an iterative natural merge sort function using arrays as in function mergeSort. How much time does this function take on an initially sorted list? Note that mergeSort takes O(n log n) on such an input list. What is the worst-case computing time of the new function? How much additional space is needed?

7.6 HEAPSORT

Merge sort has a computing time of O(n log n), both in the worst case and as average behavior, it requires additional storage proportional to the number of records to be sorted. heap sort, requires only a fixed amount of additional storage and at the same time has as its worst-case and average computing time O(n log n). However, heap sort is slightly slower than merge sort.

The deletion and insertion functions associated with max heaps directly yield an O(n log n) sorting method. The n records are first inserted into an initially empty max heap. Next, the records are extracted from the max heap one at a time. It is possible to create the max heap of n records faster than by inserting the records one by one into an initially empty heap. Fro this, we use the function adjust (Program 7.12) which starts with a binary tree whose left and right subtrees are max heaps and rearranges records so that the entire birnary tree is a max heap. The binary tree is embedded within an array using the standard mapping. If the depth of the tree is d, then the for loop is executed at most d times. Hence the computing time of adjust is O(d).

To sort the lsit, first we create a max heap by using adjust repeatedly, as in the first for loop of function heapsort (Program 7.13) Next, we swap the first and last records in the heap. Since the first record has the maximum key, the swap moves the record with maximum key into its correct position in the sorted array. We then decrement the heap size and readjust the heap. This swap, decrement heap size, readjust heap process is called a pass. For example, on the first pass, we place the record with the highest key in the nth position; on the second pass, we place the record with the second highest key in position n-1; and on the ith pass, we palace the record with the second highest key in position n-1; and on the ith pass, we place the record with the ith highest key in position n-i+1.

Example 7.7 : The input list is (26, 5, 77, 1, 61, 11, 59, 15, 48, 19) If we interpret this list as a binary tree, we get the tree of Figure 7.7(a) Figure 7.7(b) depicts the max heap

```
void adjust (element a [ ], int root, int n)

{/* adjust the binary tree to establish the heap */

    int child, rootkey;
    element temp;

    temp = a [root]. key;

    child = 2 * root ;                        /*left child */

    while (child < = n ) {

        if ((child < n) & &

        (a [child]. key < a [child] . key) /* compare root and

                                          max. child */

                break;

        else {

                a [child /2] = a [child] ; /* move to parent */

                child * =2;

        }

    }

    a [child/2] = temp;

}
```

program 7.12 : Adjusting a max heap

after the first for loop of heapsort. Figure 7.8 shows the array of records following each of the first seven iterations of the second for loop. The portion of the array that still represents a max heap is shown as a binary treet; the sorted part of the array is shown as an array.

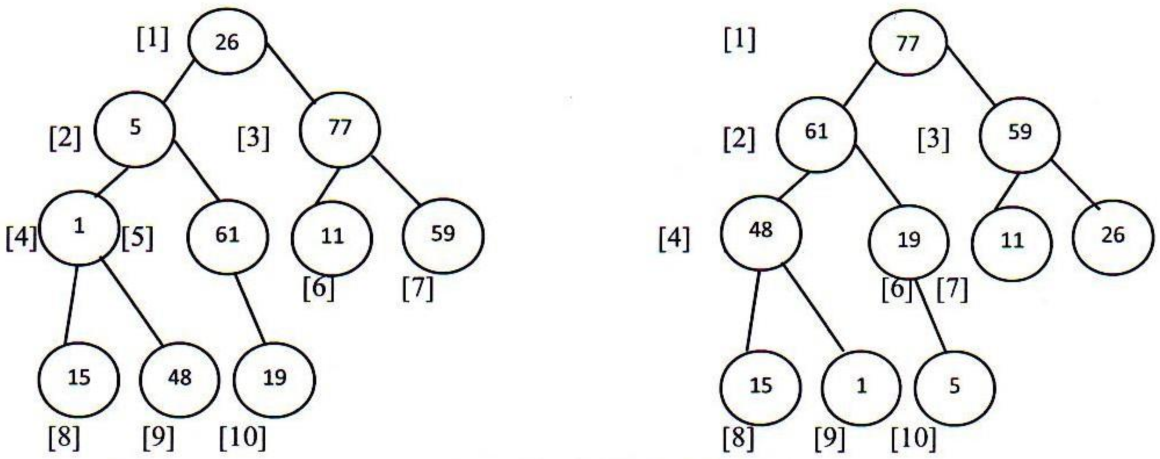
Analysis of heapSort : Suppose 2k-1 ≤ n < 2k, so the tree has k levels and the number of nodes on level I is ≤ 2i-1. In the first for loop, adjust (Program 7.12) is called once for each node that has a child. Hence, the time required for this loop is the sum, over each level, of the number of nodes on a level multiplied by the maximum distance the node can move. This is no more than

$$\sum_{1 \le i \le n} 2^{i-1(k-i)} = \sum_{1 \le i \le k-1} 2^{k-i-1 i \le n} \quad \sum_{1 \le i \le k-1} i / 2^{i} < 2n = O(n)$$

In the next for loop, n-1 applications of adjust are made with maximum tree-depth k = [$\log_2$ (n+1)] and SWAP is invoked n-1 times. Hence, the computing time for this loop is O(n logn). Consequently, the total computing time is O(n logn).

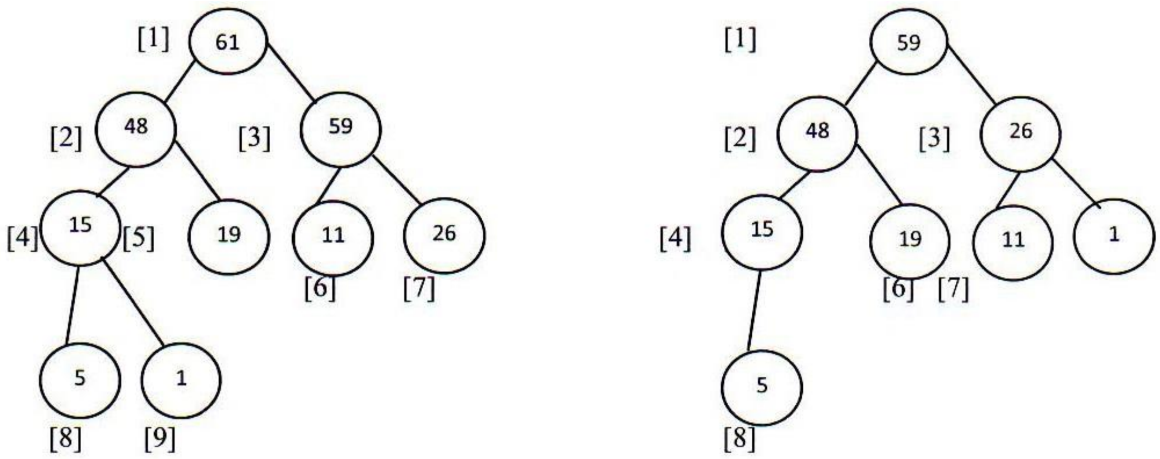
```
void heapsort (element a [ ], int n)

{/* perform a heap sort on a [1 : n] * /

    int I, j ;

    element temp ;

    for (i = n/2; i > 0 ; i - -)

        adjust (a, i, n) ;

    for ( i = n-1 ; i > 0 ; i - - ) {

        SWAP (a [1], a [i +1], temp) ;

        adjust (a, 1, i) ;

    }

}
```

Program: Heap sort



(a) Input array (b) Initial heap

Figure: Array interpreted as a binary tree



(a) Heap size = 9 (b) Heap size = 8

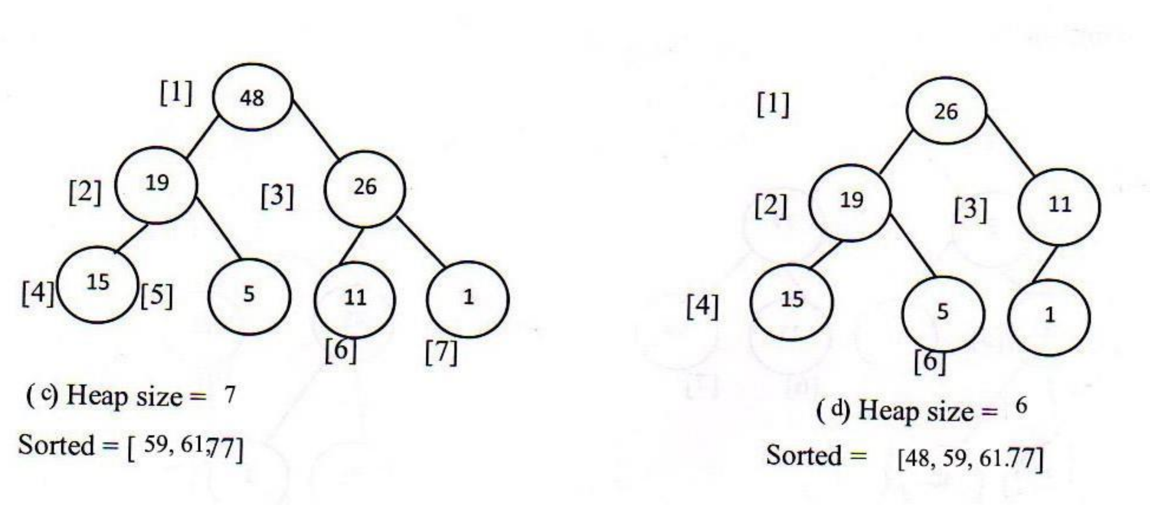
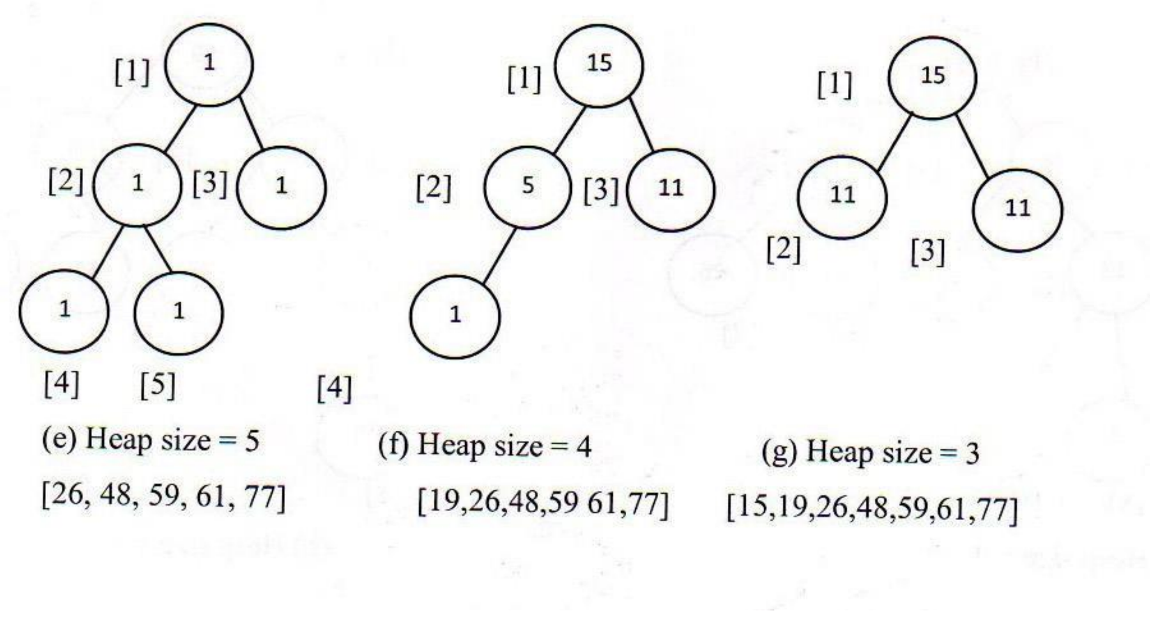


Figure : Heap sort example



EXERCISES

1. Write the status of the list (12, 2, 16, 30, 8, 28, 4, 10, 20, 6, 18) at the end of the first for loop as well as at the end of each iteration of the second for loop of heap sort (Program 7.13).
2. Heap sort is unstable. Give an example of an input list in which the order of records with equal keys is not preserved.

HASHING

HASH TABLES:

In the tables, the search for an identifier key is carried out via a sequence of comparisons. Hashing differ from this in that the address or location of an identifier, X, is obtained by computing some arithmetic function f, of X. f(X) gives the address of X in the table. This address will be referred to as the hash or home address of X. The memory available to maintain the symbol table is assumed to be sequential. This memory is referred to as the hash table, HT. The hash table is partitioned into b buckets, HT(0),…, Ht (b-1). Each bucket is capable of holding s records. Thus, a bucket is said to consist of s slots each slot being large enough to hold 1 record. Usually s = 1 and each bucket can hold exactly 1 record. A hashing function, f(X), is used to perform an identifier transformation on X. f(X) maps the set of possible identifiers onto the integers 0 through b -1. If the identifiers were drawn from the set of all legal Fortran variable names them there would be T = ∑ 0# i #5 26 x 36i >1.6 x 10^9 distinct possible values for X. any reasonable program, however, would use far less than all of these identifiers. The ration / T is the identifier density, while α –n /(sb) is the loading density or loading factor. Since the number of identifiers, n, in use is usually identifiers, T, the number of buckets b, in the hash table is also much less then T. Therefore, the has the has function f must map several different identifiers into the same bucket. Two identifiers I1 I 2 are said to be synonyms with respect to f if $f(I_1)$ – $f(I_2)$. Distinct synonyms are entered into the same buckets b, in the hash table is also much less then T. Therefore, the has the has function f must map several different identifiers into the same bucket. Two identifiers I 1, I 2 are said to be synonyms with respect to f if f(I1) – f(I2). Distinct synonyms are entered into the same bucket so long as all the s slots in that bucket has not been used. An overflow is said to occur when a new identifier I is mapped or hashed by f into a full bucket. A collision occurs when two non identical identifiers are hashed into the same bucket. When the bucket size s is 1, collisions and overflows occur simultaneously.

As an example, let us consider the hash table HT with b= 26 buckets, each bucket having exactly two slots, i.e., s=2. Assume that there are n= 10 distinct identifiers in the program and that each identifier begins with a letter. The loading factor, α, for this table is 10 /52 – 0.19. The hash function f must map each of the possible identifiers into one of the numbers 1-26.If the internal binary representation for the letters A- Z corresponds to the numbers 1-26 respectively, then the function f defined by : f(X) – the first character of X;

will has all identifiers x into the hash table. The identifiers GA,D,A,G,L,A2,A1,A4,A4 and E will be hashed into buckets 7,4,1,7,12,12,1,1,1,1 and 5 respectively by this function. The identifiers A,A1,A2,A3 and A4 are synonyms.

	SLOT 1	SLOT 2
1	A	A2
2	0	0
3	0	0
4	D	0
5	0	0
6 7 :	0 GA :	0 G :
26	0	0

Zeros indicate empty slots So also are G and G a. Figure shows the identifiers GA,D,A,G and A2 entered into the hash table. Note the GA and G are in the same bucket and each bucket has two slots. Similarly, the synonyms A and A2 are in the same bucket. The next identifier, a1, hashes into the bucket HT (1). This bucket is full and a search of the bucket indicates that A1 is not in the bucket. And overflow has now occurred.

Has function is obtained by using the models operator

Fd(x)= x mod M