MODULE-1

Syllabus

INTRODUCTION TO DATA STRUCTURES: Data Structures, Classifications (Primitive & Non-Primitive), Data structure Operations
Review of pointers and dynamic Memory Allocation,
ARRAYS and STRUCTURES: Arrays, Dynamic Allocated Arrays, Structures and Unions, Polynomials, Sparse Matrices, representation of Multidimensional Arrays, Strings
STACKS: Stacks, Stacks Using Dynamic Arrays, Evaluation and conversion of Expressions

Introduction

- **Data** is a value or a set of values.
  Example 90, Bob
- A data item refers to a single unit of values. Data items that are divided into sub items are called **group items.**
  Example : Name of an employee can be divided to three subitems- first name, middle name and last name
- Data items that are not divided into sub items are called **elementary items.**

Data Structures: A **data structure** is a particular method of storing and organizing data in acomputer so that it can be used efficiently. The data Structure is classified into

a. **Primitive data structure**: These can be manipulated directly by the machine instructions. Example integer character, float etc
b. **Non primitive data structures**: They cannot be manipulated directly by the machine instructions. The non primitive data structures are further classified into linear and non linear data structures.
   - **Linear data structures**: show the relationship of adjacency between the elements of the data structures. Example are arrays, stacks, queues , list etc.
   - **Non linear data structure**: They do not show the relationship of adjacency between the elements. Example are Trees and graphs

Operations on data structures

1. **Create:** Creating a new data structure
2. **Insert:** Adding a new record to the structure.
3. **Delete:** Removing the record from the structure.
4. **Search:** Finding the location of a particular record with a given key value, or finding the location of all records which satisfy one or more conditions.
5. **Sort:** Managing the data or record in some logical order (Ascending or descending order).
6. **Merge:** Combining the record in two different sorted files into a single sorted file.
7. **Traversal**: Accessing each record exactly once so that certain items in the record may be processed.

Review of arrays

- Array is a collection of elements of the same data type
- An array is declared by appending brackets to the name of a variable.

For example

```
int list[5];                    // declares an array that can store 5 integers
```

In C all array index start at 0 and so list[0],list[1],list[2],list[3],list[4] are the names of the five array elements each of which contains an integer value.

Structures : Structure is basically a user-defined data type that can store related information thatmay be of same or different data types together.

The major difference between a structure and an array is that an array can store only information of same data type. A structure is therefore a collection of variables under a single name. The variables within a structure are of different data types and each has a name that is used to select it from the structure.

For example,

```
Struct student {
        char sname[10];
        int age;
        float average_marks;
} st;
```

It creates a variable whose name is ***st*** and that has three fields:

- a name that is a character array
- an integer value representing the age of the student
- a **float** value representing the average marks of the individual student.

To assign values to these fields dot operator **(.) is used** as the structure member operator. We use this operator to select a particular member of the structure.

```
strcpy(st.sname,"james");
st.age = 10;
st.average_marks = 35;
```

We can create our own structure data types by using the **typedef** statement. Consider an example that creates a structure for the employee details.

```
typedef struct Employee {
        char name[10];
        int age;
        float salary;
};
          or
typedef struct {
        char name[10];
        int age;
        float salary;
 } Employee;
```

Comparing structures: Return TRUE if employee 1 and employee 2 are the same otherwisereturn FALSE

```
int EmployeeEqual(employee p1, Employee p2)
{
        if (!strcmp(p1.name, p2.name))
                return FALSE;
        if (p1.age != p2.age)
                return FALSE;
        if (p1.salary != p2.salary)
                return FALSE;
                return TRUE;
}
```

A typical function call might be:

```
if (EmployeeEqual(p1,p2))
                printf("The two employee are the same\n");
        else
                printf("The two Employee are not the same\n");
```

Nested Structure: A structure can be embedded within another structure. That is a structure canhave another structure as its member such a structure is called a nested structure.

For example, associated with our *employee* structure we may wish to include the date of Birth of an employee by using nested stucture

```
typedef struct {
        int month;
        int  day;
        int year;
} date;
typedef struct {
        char name[10];
        int age;
        float salary;
        date dob;
}employee;
```

If we have to store the information of a person we declare a variable as
Employee p1;
A person born on September 10, 1974, would have the values for the *date* **struct** set as:

```
p1.dob.month = 9;
p1.dob.day = 10;
p1.dob.year = 1974;
```

Array of Structures: In the case of a student or the employee we may not store the details ofonly 1 student or 1 employee. When we have to store the details of a group of students we can declare an array of structures.

Example: struct student s[10];

Self-Referential Structures: A *self-referential structure* is one in which one or more of its data member is a pointer to itself. They require dynamic memory allocation (*malloc* and *free*) to explicitlyobtain and release memory.

Example:

```
typedef struct list {
        int data;
        list *link ;
} ;
```

- Each instance of the structure *list* will have two components, *data* and *link. data* is a single character, while *link* is a pointer to a *list* structure.
- The value of *link* is either the address in memory of an instance of *list* or the null pointer.

Consider these statements, which create three structures and assign values to their respective fields:

```
list item1, item2, item3;
        item1.data = 5
        item2.data =  10
        item3.data = 15
        item1.link = item2.link = item3.link = NULL;
```

We can attach these structures together by replacing the null *link* field in *item* 2 with one that points to *item* 3 and by replacing the null *link* field in *item* 1 with one that points to *item* 2.

```
item1.link = &item2; item2.1ink = &item3;
```

Unions: A **union is** a user-defined data type that can store related information that may be of different data types or same data type, but the fields of a **union** must share their memory space. This means that only one field of the **union** is "active" at any given time.

Example1: Suppose a program uses either a number that is int or float we can define a union as

```
Union num
{
        int a;
        float b;
};
Union num n1;
```

Now we can store values as n1.a=5 or n2.b= 3.14 only one member is active at a point of time.

Pointer variables, Declaration and Definition

- **Pointer variable** is a variable that stores the address of another variable.
- **Declaring Pointer variables**

  A variable can be declared as a pointer variable by using an indirection operator(*)

  Syntax: type * identifier; // type signifies the type of the pointer variable

  For example

```
char *p;
int *m;
```

//The variable p is declared as a pointer variable of type character. The variable m is declared as a pointer variable of type integer.

Initialization of pointer variables: Uninitialized variables have unknown garbage values stored in them, similarly uninitialized pointer variables will have uninitialized memory address stored inside them which may be interpreted as a memory location, and may lead to runtime error.

These errors are difficult to debug and correct, therefore a pointer should always be initialized with a valid memory address.

A pointer can be initialized as follows

```
int a;
int *p;
```

//Here the variable a and the pointer variable p are of the same data type. To make p to point at a we have to write a statement

```
p=&a;    // now the address of a is stored in the pointer variable p and now p is said to be
         pointing at a.
```

If we do not want the pointer variable to point at anything we can initialize it to point at NULL

For example: int *p =NULL;
When we dereference a null pointer, we are using address zero, which is a valid address in the computer.

NOTE: A pointer variable can only point at a variable of the same type.
We can have more than one pointer variable pointing at the same variable. For example

```
int a;
int *p,*q;
p=&a;
q=&a;
```

now both the pointer variable p and q are pointing at the same variable a. There is no limit to the number of pointer variable that can point to a variable.

Accessing variables through pointers

- A Variable can be accessed through a pointer variable by using an indirection operator.
- Indirection operator(*): An indirection operator is a unary operator whose operand must be a pointer value.
- For example to access a variable a through a pointer variable p we have to code it as follows

```
Void main()
{
        int a=5;
        int *p
        p=&a;// p is now pointing at a
        *p=*p+1
        printf(" %d   %d  %p",  a, *p,p);
}
```

Output: 6 6 XXXXX(address of variable a)
Now the value of a is modified through the pointer variable p

Note:

- we need parenthesis for expressions like (*p) ++ as the precedence of postfix increment is more than precedence of the indirection operator (*). If the parenthesis is not used the address will be incremented.
- The indirection and the address operators are the inverse of each other when combined in an expression such as *&a they cancel each other

Write a program to add two numbers using pointers.

```
#include <stdio.h>
int main()
{
  int num1, num2, *p, *q, sum;
  printf("Enter two integers to add\n");
  scanf("%d%d", &num1, &num2);
        p = &num1;    q = &num2;
        sum = *p + *q;
  printf("Sum of the numbers = %d\n", sum)
 return 0;
}
```

Write a program to swap two numbers.

```
#include <stdio.h>
int main()
{
  int num1, num2, *p, *q, sum;
  printf("Enter two integers to swap\n");
  scanf("%d%d", &num1, &num2);
        p = &num1;    q = &num2;
        temp = *p;
        *p = *q;
        *q = temp;
  printf("After Swapping p = %d, q = %d\n", p,q);
  return 0;
}
```

Pointers and Functions

When we call a function by passing the address of a variable we call it as pass by reference. By passing the address of variables defined in main we can directly store the data in the calling function rather returning the value. Pointers can also be used when we have to return more than one value from a function

Program to swap two characters using Functions.

```
void main()
{
        char a ,b;
        printf("\nenter two characters\n");
        scanf("%c %c", &a,&b);
        printf("the value before swap: %c %c" a,b);
        swap(&a,&b);
        printf("the value after swap: %c %c" a,b);
}
void swap(char *p1,char *p2)
{
        char temp;
        temp=*p1;
        *p1=*p2;
        *p2=temp;
}
```

Memory allocation functions: In high level languages the data structures are fully defined at compile time. Modern languages like C can allocate memory at execution this feature is known as dynamic memory allocation.

There are two ways in which we can reserve memory locations for a variable

- **Static memory allocation**: the declaration and definition of memory should be specified in the source program. The number of bytes reserved cannot be changed during runtime
- **Dynamic memory allocation** : Data definition can be done at runtime .It uses predefined functions to allocate and release memory for data while the program is running. To use dynamic memory allocation the programmer must use either standard data types or must declare derived data types

Memory usage: Four memory management functions are used with dynamic memory. malloc, calloc and realloc are used for memory allocation. The function free is used to return memory when it is not used.

Heap: It is the unused memory allocated to the program When requests are made by memory allocating functions, memory is allocated from the heap at run time.

Memory Allocation (malloc)

- When a malloc function is invoked requesting for memory, it allocates a block of memory that contains the number of bytes specified in its parameter and returns a pointer to the start of the allocated memory.
- When the requested memory is not available the pointer NULL is returned.

syntax: void *malloc (size_t size);

Example: void *malloc(sizeof(int));

- The pointer returned by the malloc function can be type cast to the pointer of the required type by making use of type cast expressions

Example: To allocate an integer in the heap we code

```
int *pint
pint=(int*)malloc(sizeof(int))
```

Releasing memory (free): When memory locations allocated are no longer needed, they should be freed by using the predefined function free.

Syntax: free(void*);

Example:

```
int *p,a;
    p=&a;
    free(p);
```

Program showing the allocation of memory using malloc

```
int i,*pi;
float f,*pf;
    Pi= (int*) malloc (sizeof((int));
    Pf= (float *) malloc(sizeof(float));
    *pi= 1344;
    *pf= 3.14
Printf("integer value= %d  float value= %f",*pi, *pf);
Free(pi);
Free(pf);
```

Contiguous memory allocation (calloc)

- This function is used to allocate contiguous block of memory. It is primarily used to allocate memory for arrays.
- The function calloc() allocates a user specified amount of memory and initializes the allocated memory to 0.
- A pointer to the start of the allocated memory is returned.
- In case there is insufficient memory it returns NULL

syntax : void * calloc (size_t count , size_t size);

Example: To allocate a one dimensional array of integers whose capacity is n the following code can be written.

```
int *ptr
ptr=(int*)calloc(n,sizeof(int))
```

Reallocation of memory(realloc): The function realloc resizes the memory previously allocated by either malloc or calloc.

syntax: Void * realloc (void * ptr , size_t new_size);

Example

```
int *p;
p=(int*)calloc(n,sizeof(int))
p=realloc(p,s)                    /*where s is the new size*/
```

The statement realloc(p,s) -- Changes the size of the memory pointed by p to s. The existing contents of the block remain unchanged.

- When s> oldsize(Block size increases) the additional (s – oldsize)have unspecified value
- When s<oddsize (Block size reduces) the rightmost (oldsize-s) bytes of the old block are freed.
- When realloc is able to do the resizing it returns a pointer to the start of the new block
- When is not able to do the resizing the old block is unchanged and the function returns the value NULL.

Dangling Reference: Once a pointer is freed using the free function then there is no way to retrieve this storage and any reference to this location is called dangling reference.

Example2:

```
int i,*p,*f;
i=2;
p=&i;
f=p;
free(p);
*f=*f+2 /* Invalid dangling reference*/
```

The location that holds the value 2 is freed but still there exist a reference to this location through f and pointer f will try to access a location that is freed so the pointer f is a dangling reference

Pointers can be dangerous: When pointers are used the following points needs to be taken care

1. When a pointer is not pointing at any object it is a good practise to set it to NULL so that there is no attempt made to access a memory location that is out of range of our program or that does not contain a pointer reference to the legitimate object.

2. Use explicit type casts when converting between pointer types.

```
int *pi;
float *pf;
Pi=  (int*) malloc (sizeof((int));
Pf= (float *)pi;
```

3. Define explicit return types for functions. If the return type is omitted it defaults to integer which has the same size as a pointer and can be later interpreted as a pointer

Dynamically allocated Arrays

One dimensional array: When we cannot determine the exact size of the array the space of the array can be allocated at runtime.

For example consider the code given below

```
int i,n,*list;
printf("enter the size of the array");
scanf("%d",&n);
if (n<1)
{
fprintf(stderr,"Improper values of n \n");
exit();
}
list=(int*) malloc (n*sizeof(n))/* or list=(int*)calloc(n,sizeof(int))
```

Two dimensional Arrays

- Example for representing a 2 dimensional array int x[3][5];
- Here a one dimensional array is created whose length is 3 where each element of x is a one dimensional array of length 5

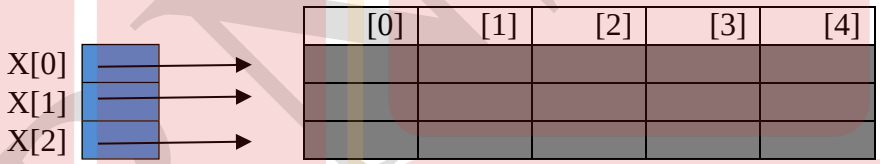


Figure 1.2 Memory Representation of two dimensional array

In C we find the element x[i][j] by first accessing the pointer in x[i]. This pointer gives the address of the zeroth element of row i of the array. Then by adding j*sizeof(int) to this pointer, the address of the jth element of the ith row is determined

Example to find x[1][3] we first access the pointer in x[1] this pointer gives the address of x[1][0] now by adding 3*sizeof (int) the address of the element x[1][3] is determined.

Program to create a 2 dimensional array at run time.

```
int ** maketwodarray(int rows,int cols)
{
    int **x, i;
    x=(int*)malloc( rows  * sizeof(*x));
    for (i=0;i< rows; i++)
    x[i]= malloc(cols*sizeof(int));
    return x;
}
```

The function can be invoked as follows to allocate memory

```
int ** twodarray
twodarray= maketwodarray(5,10);
```

Arrays

Linear Arrays: A Linear Array is a list of finite number (n) of homogenous data elements.

a. The elements of the array are referenced by an index set consisting of n consecutive numbers(0. ..(n-1)).
b. The elements of the array are stored in successive memory locations
c. The number n of elements is called the length or size of the array. **Length of the array** can be obtained from the index set using the formula
   Length = Upper bound – Lowe bound +1
d. The elements of an array may be denoted by a[0],a[2].....a[n-1]. The number k in a[k] is called a subscript or index and a[k] is called the subscripted value.
e. An array is usually implemented as a consecutive set of memory locations

Declaration: Linear arrays are declared by adding a bracket to the name of a variable. The size of the array is mentioned within the brackets.

Eg :- int list[5]; // Declares an array containing five integer elements.

In C all arrays start at index 0. Therefore, list[0], list[1], list[2], list[3], and list[4] are the names of the five array elements ,each of which contains an integer value.

Representation of Linear Arrays in memory

- When the compiler encounters an array declaration such as **int list[5],** to create list, it allocates five consecutive memory locations. Each memory location is large enough to hold a single integer.
- The address of the first element list[0], is called the base address
  base address=address(list[0])
- Using the base address the address of any element of list can be calculated using the formula
  address(list[k])= base addres + w.k
  // where w is the size of each element in the array list

Example: **int list[5]**

- The elements of the array is list[0] list[4]
- If the size of an integer on the machine is denoted by sizeof(int), then the memory
  Address(list [k])= base address+ sizeof (int).k.

Variable Memory Address canlculatio

Let α the base address , the address of list[0]
Let w=sizeof(int)

address(list[1])= α + w*1
address(list[2]) = α + w*2
address(list[3]) = α + w*3
address(list[4]) = α + w*4

Multi dimensional arrays

Two dimensional arrays: C uses the array of array representation to represent a multidimensional array. In this representation a 2 dimensional array is represented as a one dimensional array in which each element is itself a one dimensional array.

- A two dimensional m X n array A is a collection of m* n data elements such that each element is specified by a pair of integers called subscripts.
- An element with subscript i and j will be represented as A[i][j]
- **Declaration**: int A[3][5];
  // It declares an array A that contains three elements where each element is a one dimensional array. Each one dimensional array has 5 integer elements.

Example : A 2 dimensional array A[3][4] is represented as

		0	1	2	3
	0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
Rows	1	A[1][0]	A[1][1]	A[1][2]	A[1][3]
	2	A[2][0]	A[2][1]	A[2][2]	A[2][3]

Representation of two dimensional arrays in memory: A two dimensional m X n array A is stored in the memory as a collection of m*n sequential memory locations.If the array is stored column by column it is called column major order and if the array is stored row by row it is called row major order.

Example: Representation of the two dimensional array A[3][4] in row major order and column major order

A	Subscript	
	A[0][0]	**Row1**
	A[0][1]	
	A[0][2]	
	A[0][3]	
	A[1][0]	**Row2**
	A[1][1]	
	A[1][2]	
	A[1][3]	
	A[2][0]	**Row3**
	A[2][1]	
	A[2][2]	
	A[2][3]	

Row Major Order

A	Subscript	
	A[0][0]	**Column1**
	A[1][0]	
	A[2][0]	
	A[0][1]	**Column2**
	A[1][1]	
	A[2][1]	
	A[0][2]	**Column3**
	A[1][2]	
	A[2][2]	
	A[0][3]	**Column4**
	A[1][3]	
	A[2][3]	

Column major order

Figure 1.1 Representation of Two Dimensional array

- Using the base address , the address of any element in an array A of size row X col can be calculated using the formula.
- **Row Major order**
  Address (A[i][j]) = Base address + w[i*col+ j] considering the array indexing starts at 0
- **Column Major order**
  Address (A[i][j]) = Base address + w[i+row.j] considering the array indexing starts at 0

Example : When the compiler encounters an array declaration such as **int A[3][4] it** creates an array A and allocates 20 consecutive memory locations. Each memory location is large enough to hold a single integer.

Let α be the address of the first element A[0][0], is called the base address

Considering Row major order: Using the bases address we can calculate the addresses of other element

Address of A[0][1] = 100 +2[0*4+ 1]= 100 +2=102
Address of A[0][2] = 100 +2[0*4+ 2]= 100 +4=104
Address of A[1][0] = 100 +2[1*4+ 0]= 100 +8=108
Addres of A[2][3]= 100 +2[2*4+3]= 100+22= 122

Representation of Multidimensional Arrays: In C, multidimensional arrays are represented using the array-of-arrays representation The linear list is then stored in consecutive memory just as we store a one-dimensional array.

If an n-dimensional array a is declared as a[upper0][upper1] ⋯ [uppern-1]; then the number of elements in the array is: $upper_0 * upper_1 * \ldots\ldots upper_{n-1}$ also represented as

$$\prod_{i=0}^{n-1} upper_i$$

where Π is the product of the upperi's. For instance, if we declare a as a[10][10][10], then we require 10·10·10 = 1000 memory cell to hold the array. There are two common ways to represent multidimensional arrays: row major order and column major order. We consider only row major order here. As its name implies, row major order stores multidimensional arrays by rows.

Two dimensional arrays

- For instance, we interpret the two dimensional Array A[upper0][upper1] as upper0 rows, , each row containing upper1 elements.
- If we assume that α the base address is the address of A[0][0]
- Then the address of an arbitrary element,
  Address(a[i][j])= α + i·upper1 + j ----------(1)
- Here the size is not considered. Considering the size the formula can be written as
  Address (a[i][j]) = α + w(i·upper1 + j) where w is the size of each unit of memory location.

Representation of a three-dimensional array

- A[upper0][upper1][upper2], we interpret the array as upper0 two dimensional arrays of dimension upper1 × upper2.

address of A[i][j][k]=□α + i·upper1·upper2 + j·upper2 + k --------- (2)

Representation of a fourth-dimensional array

- A[upper0][upper1][upper2][upper3]
- We interpret the array as upper0 three dimensional arrays of dimension upper1 x upper2 x upper3 .
  address of A [i][j][k][l]=□α + i·upper1·upper2.upper3 + j·upper2..upper3 + k.upper3 + l

Representation of an n-dimensional array

- Generalizing on the preceding discussion, we can obtain the addressing formula for any element A[i0][i1] … [in-1] in an n dimensional array declared as:
  A[upper0][upper1] … [uppern-1]

- If α is the address for A[0][0] … [0] the base address Then
  Address of a[i0][0][0][0] … [0] = □α + i0 upper1 upper2 … upper n-1

// address of a[i0][i1][0] … [0] = □α+i0 upper1 upper2 … uppern-1 + i1 upper2 upper3 … uppern-1

- Repeating in this way the address for A[i0][i1] … [in-1] is=

$$\begin{aligned}&\alpha + i_0 upper_1 upper_2 \ldots upper_{n-1} \\ &\quad + i_1 upper_2 upper_3 \ldots upper_{n-1} \\ &\quad + i_2 upper_3 upper_4 \ldots upper_{n-1} \\ &\quad . \\ &\quad . \\ &\quad . \\ &\quad + i_{n-2} upper_{n-1} \\ &\quad + i_{n-1}\end{aligned}$$

$$= \alpha + \sum_{j=0}^{n-1} i_j a_j \text{ where: } \begin{cases} a_j = \prod_{k=j+1}^{n-1} upper_k & 0 \le j < n-1 \\ a_{n-1} = 1 \end{cases}$$

Array Operations: Operations that can be performed on any linear structure whether it is anarray or a linked list include the following

a. **Traversal**- processing each element in the list
b. **Search**- Finding the location of the element with a given key.
c. **Insertion**- Adding a new element to the list
d. **Deletion**- Removing an element from the list.
e. **Sorting**- Arranging the elements in some type of order.
f. **Merging**- combining two list into a single list.

Traversing Linear Arrays: Traversing an array is accessing and processing each element exactly once. Considering the processing applied during traversal as display of elements the array can be traversed as follows

```
void displayarray(int a[])
{
   int i;
   printf("The Array Elements are:\n");
   for(i=0;i<n;i++)
   printf("%d\t",a[i]);
}
```

Insertion

- Inserting an element at the end of the array can be done provided the memory space allocated for the array is large enough to accommodate the additional element.
- If an element needs to be inserted in the middle then all the elements form the specified position to the end of the array should be moved down wards to accommodate the new element and to keep the order of the other element.
- The following function inserts an element at the specified position

```
void insertelement(int item, int pos ,int *n, int a[],)
{
   int i;
   if (pos<0 || pos>n)
      printf("Invalid Position\n");
```

```
    else
    {
        for(i=n-1;i>=pos;i--)
        a[i+1]=a[i];        //Make space for the new element in the given position

        a[pos]=element;
        *n++;
    }
}
```

Deletion

- If an element needs to be deleted in the middle then all the elements form the specified position to the end of the array should be moved upwards to fill up the array.
- The following function deletes an element at the specified position

```
void deleteelement(int a[],int pos,int* n)
{
   int i;
   if (pos<0 || pos>n-1)
      printf("Invalid Position\n");
   else
   {
   printf("The Deleted Element is %d\n",a[pos]);
   for(i=pos;i<n;i++)
      a[i]=a[i+1];    //Delete by pushing up other elements

   *n--;
   }
}
```

Sorting: Sorting refers to the operation of rearranging the elements of an array in increasing or decreasing order.

Example: Write a program to sort the elements of the array in ascending order using bubble sort.

```
#include<stdio.h>
void main()
{
int a[10],i,j,temp,n;
printf("enter the size of the array : ");
scanf("%d",&n);
printf("enter the elements of the array\n");
for(i=0;i<n;i++)
scanf("%d",&a[i]);

for(i=1;i<=n-1;i++)
 for(j=0;j<n-i ;j++)
   if (a[j] >a[j+1])
    {
      temp=a[j];
     a[j]=a[j+1];
```

```
        a[j+1]= temp;
          }
            printf("the sorted array is \n");
            for(i=0;i<n;i++)
            printf("%d \t",a[i]);
            return(0);
}
```

Complexity of the bubble sort algorithm is $O(n^2)$

- The time for sorting is measured in terms of the number of comparisons. In the bubble sort there are n-1 comparison in the first pass which places the largest element in the last position.
- There are n-2 comparison in the second pass which places the second largest element in the next to last position and so on, therefore

  $C(n)= (n-1) +(n-2) + \quad 2+1= n(n-1)/2= n^2/2 = O(n^2)$

Searching:

- Let DATA be a collection of data elements in memory and suppose a specific ITEM of information is given.
- Searching refers to the operation of finding the Location LOC of the ITEM in DATA or printing a message that the item does not appear here.
- The search is successful if the ITEM appear in DATA and unsuccessful otherwise.

The algorithm chosen for searching depends on the way the data is organised. The two algorithm considered here is linear search and binary search.

LINEAR SEARCH: This program traverses the array sequentially to locate key

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
        int a[10],i,key,pos,n,flag=0;
        printf("enter the size of the array : ");
        scanf("%d",&n);
        printf("enter the elements of the array\n");
        for(i=0;i<n;i++)
        scanf("%d",&a[i]);
        printf("enter the key \n");
        scanf("%d",&key);
        for(i=0;i<=n-1;i++)
          if (a[i]== key)
          {
            printf("key %d found at %d",key,pos+1);
            exit();
          }
        printf("key not found");
}
```

Complexity of Linear search: The complexity is based on the number of comparison C(n) required to find the key in the array element.

- **The best case** occurs when the key is found at first position. $C(n) \in O(1)$

- **Worst case** occurs when key element is not found in the array or when the element is in the last position. Thus in worst case the running time is proportional **to n C(n) ∈O(n)**
- The running time of the average case uses the probabilistic notation of expectation. Number of comparison can be any number from 1 to n and each occurs with probability p= 1/n then
  c(n) = 1.1/n +2.1/n+.....n.1/n
  = (1+2+3… .....+n).1/n
  =n(n+1)/2.1/n=n+1/2

BINARY SEARCH: This algorithm is useful when the array is sorted.
For example when searching for a name in a telephone directory this algorithm is more efficient than linear search as the number of element to search is reduced by half in each iteration.

```
#include<stdio.h>
#include<stdlib.h>
int main()
        {
        int a[10],i,key,mid,low,high,n;
        printf("enter the size of the array : ");
        scanf("%d",&n);
        printf("enter the elements of the array in ascending order\n");
        for(i=0;i<n;i++)
        scanf("%d",&a[i]);
        printf("enter the key \n");
        scanf("%d",&key);

        low=0;
        high=n-1;
        while(low<=high)
        {
           mid=(low+high)/2;
           if (key==a[mid])
           {
              printf("element %d found at %d",key,mid+1);
              exit(0);
           }
           else
           {
              if (key<a[mid])
                high = mid-1;
                else
                  low=mid+1;
           }
}
printf("key not found");
return(0);
}
```

Complexity of binary search algorithm :

- The complexity is based on the number of comparison C(n) required to find the key in the array element.
- Each comparison reduces the sample size in half so C(n) is of the order $\log_2 n + 1$

Limitation of binary search :

1. The list must be sorted
2. One must have a direct access to the middle element in any subset.

Polynomials: 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.

Example for polynomials :

$A(x) = 3x^{20} + 2x^5 + 4$

$B(x) = x4 + 10x3 + 3x2 + 1$

The largest (or leading) exponent of a polynomial is called its *degree.* Coefficients that are zero are not displayed.

- Standard mathematical definitions for the sum and product of polynomials are:
- Assume that we have two polynomials

$$A(x) = \sum a_i x^i \qquad B(x) = \sum b_i x^i$$

then

$$A(x) + B(x) = \sum (a_i + b_i) x^i$$

$$A(x) \cdot B(x) = \sum (a_i x^i \cdot \sum (b_j x^j))$$

ADT Polynomial is **objects:** a set of ordered pairs of <ei, ai> where ai is Coefficients and ei is Exponents, ei are integers >= 0

Functions:

for all poly,poly1,poly2 ∈ Polynomial,coef ∈ Coefficients, expon ∈ Exponents

Polynomial Zero()	::=	**return** the polynomial, p(x) = 0
Boolean IsZero(poly)	::=	**if** (poly) **return** FALSE **else return** TRUE
Coefficient Coef(poly,expon)	::=	**if** (expon ∈ poly) **return** its coefficient **else return** zero
Exponent LeadExp(poly)	::=	**return** the largest exponent in poly
Polynomial Attach(poly,coef,expon)	::=	**if** (expon ∈ poly) **return** error **else return** the polynomial poly with the term <coef, expon> inserted
Polynomial Remove(poly,expon)	::=	**if** (expon ∈ poly) **return** the polynomial poly with the term whose exponent is expon deleted **else return** error
Polynomial SingleMult(poly,coef,expon)	::=	**return** the polynomial poly · coef · x^{expon}
Polynomial Add(poly1,poly2)	::=	**return** the polynomial poly1 + poly2
Polynomial Mult(poly1,poly2)	::=	**return** the polynomial poly1 · poly2
end Polynomial	::=	

Polynomial Representation:

- A polynomial can be represented as an array of structures as follows.
- Only one global array, *terms,* is used to store all the polynomials.
- The C declarations needed are:

```
# define MAX_TERMS 100 /*size of terms array*/
typedef struct {
float coef;
int expon;
} polynomial;
polynomial terms[MAX_TERMS];
int avail = 0;
```

Consider the two polynomials

$A(x) = 2x^{1000} + 1$ and

$B(x) = x^4 + 10x^3 + 3x^2 + 1$.

Figure below shows how these polynomials are stored in the array terms. The index of the first term of A and B is given by startA and startB, respectively, finishA and finishB give the index of the last term of A and B respectively. The index of the next free location in the array is given by avail.

For our example, startA = 0, finishA = 1, startB = 2, finishB = 5, and avail =6.

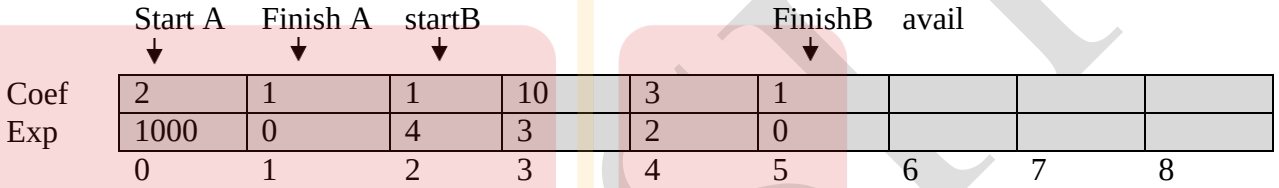


Figure 1.3 Representation of polynomial in array

- There is no limit on the number of polynomials that we can place in terms.
- The total number of nonzero terms must not be greater than MAX_TERMS.

since *A* (*x*) = 2*x* 1000 + 1 uses only six units of storage: one for *startA*, one for *finishA*, two for the coefficients, and two for the exponents. However, when all the terms are nonzero, the current representation requires about twice as much space as the first one. This representation is useful only when the number of non zero terms are more.

Polynomial addition

- C function that adds two polynomials, A and B to obtain the resultant polynomial D = A + B. The polynomial is added term by term.
- The attach function places the terms of D into the array, terms starting at position avail,.
- If there is not enough space in terms to accommodate D, an error message is printed to the standard error device and we exit the program with an error condition.

```
void padd(int startA,int finishA,int startB, int finishB, int *startD,int *finishD)
{
        / * add A(x) and B(x) to obtain D(x) */
        float coefficient;
        *startD = avail;
        while (startA <= finishA && startB <= finishB)
                {
                        switch(COMPARE(terms[startA].expon, terms[startB].expon))
                        {
                                case -1: attach(terms[startB].coef,terms[startB].expon);
```

```
                                        startB++;
                                        break;
                                case 0: coefficient = terms[startA].coef + terms[startB].coef;
                                        if (coefficient)
                                        attach(coefficient,terms[startA].expon);
                                        startA++;
                                        startB++;
                                        break;
                                case 1: attach(terms[startA].coef,terms[startA].expon);
                                        startA++;
                        }
                }
                while(startA <= finishA)
                {
                attach(terms[startA].coef,terms[startA].expon);  /* add in remaining terms of A(x) */
                startA++;
                }

                While(startB <= finishB)
                {
                attach(terms[startB].coef, terms[startB].expon);  /* add in remaining terms of B(x) */
                startB++;
                }

                *finishD = avail-1;
}

/* add a new term to the polynomial */

void attach(float coefficient, int exponent)
{
                if (avail >= MAX_TERMS)
                 {
                        fprintf(stderr,"Too many terms in the polynomial\n") ;
                        exit(EXIT_FAILURE);
                }
                terms[avail].coef = coefficient;
                terms[avail].expon = exponent;
                avail++;
}
```

Time complexity analysis

- The number of non zero terms in *A* and in *B* are the most important factors in the time complexity.
- The first loop can iterate *m* times and the second can iterate *n* times. So, the asymptotic computing time of this algorithm is O(*n* +*m*).

Sparse Matrices

- If a matrix contains m rows and n columns the total number of elements in such a matrix is m*n. If m equals n the matrix is a square matrix.

- When a matrix is represented as a two dimensional array defined as a[max_rows][max_cols], we can locate each element quickly by writing a[i][j] where i is the row index and j is the column index.
- Consider the matrix given below. It contains many zero entries, such a matrix is called a sparse matrix

	Col0	Col1	Col2	Col3	Col4	Col5
Row0	15	0	0	22	0	-15
Row1	0	11	3	0	0	0
Row2	0	0	0	-6	0	0
Row3	0	0	0	0	0	0
Row4	91	0	0	0	0	0
Row5	0	0	28	0	0	0

When a sparse matrix is represented as a two dimensional array space is wasted for example if we have 1000x 1000 matrix with only 2000 non zero element, the corresponding 2 dimensional array requires space for 1,000,000 elements

ADT Sparse Matrix objects: a set of triples, <row, column, value>, where row and column are integers and form a unique combination, and value comes from the set item.
Functions:
for all a, b∈SparseMatrix, x∈item, i, j, maxCol, maxRow∈index

Sparse MatrixCreate(maxRow, maxCol) ::=	**return** a SparseMatrix that can hold up to maxItems = maxRow × maxCol and whose maximum row size is maxRow and whose maximum column size is maxCol.
Sparse MatrixTranspose(a) : =	**return** the matrix produced by interchanging the row and column value of every triple.
Sparse MatrixAdd(a, b) :=	**if** the dimensions of a and b are the same **return** the matrix produced by adding corresponding items, namely those with identical row and column values **else return** error
Sparse MatrixMultiply(a, b) :=	**if** number of columns in a equals number of rows in b **return** the matrix d produced by multiplying a by b according to the formula: d[i][j] = Σ (a[i][k]·b[k][j]) **else return** error.

Sparse Matrix Representation

- A Sparse matrix can be represented by using an array of triple <*row*, *col*, *value* >.
- In addition to ensure the operations terminate , it is necessary to know the number of rows and columns, and the number of nonzero elements in the matrix. Putting all this information together a sparse matrix can be created as follows

SparseMatrix Create(*maxRow*, *maxCol*) ::=

```
#define MAX_TERMS 101 /* maximum number of terms +1*/
typedef struct {
            int col;
            int row;
            int value;
    } term;
term a[MAX_TERMS] ;
```

Example

	Col0	Col1	Col2	Col3	Col4	Col5
Row0	15	0	0	22	0	-15
Row1	0	11	3	0	0	0
Row2	0	0	0	-6	0	0
Row3	0	0	0	0	0	0
Row4	91	0	0	0	0	0
Row5	0	0	28	0	0	0

a) Two dimensional array

	Row	Col	Value
a[0]	6	6	8
a[1]	0	0	15
a[2]	0	3	22
a[3]	0	5	-15
a[4]	1	1	11
a[5]	1	2	3
a[6]	2	3	-6
a[7]	4	0	91
a[8]	5	2	28

b) Sparse matrix stored as triples

Figure 1.4 two dimensional array and its sparse matrix stored as triples

Write a program to store a sparse matrix in triplet form and search an element specified by the user

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
struct sparse
{
        int  r;
        int  c;
        int v;
};
struct poly s[100];
int ele,i,j,k,n,m,key;
printf("enter the size of the array ; ");
scanf("%d %d",&m,&n);
k=1;

s[0].r=m;
s[0].c=n;
printf("\n enter the elements of the array\n");
for(i=0;i<m;i++)
for(j=0;j<n;j++)
{
   scanf("%d",&ele);

if(ele !=0)
{
        s[k].r=i;
        s[k].c=j;
        s[k].v= ele;
        k++;
```

```
}
s[0].v=k-1;
}
for(i=0;i<=s[0].v;i++)
printf(" %d\t %d \t %d \n ",s[i].r, s[i].c, s[i].v);
printf(" enter the key to be searched");
scanf("%d",&key);
for(i=0;i<=s[0].v;i++)
   if (key== s[i].v)
   {
            printf("element found at %d row and %d column",s[i].r,s[i].c);
            exit(0);
   }
printf("element not found ");
return(0);
}
```

Transposing a Matrix: To transpose a matrix we must interchange the rows and columns. This means that each element a[i][j] in the original matrix becomes element b[j][i] in the transposematrix.

The algorithm finds all the elements in column 0 and store them in row 0 of the transpose matrix, find all the elements in column 1 and store them in row 1, etc." Since the original matrix was ordered by rows and the columns were ordered within each row. The transpose matrix will also be arranged in ascending order. The variable, *currentb,* holds the position in *b* that will contain the next transposed term. The terms in b is generated by rows by collecting the nonzero terms from column i of a

The transpose b of the sparse matrix a of figure 1.4b is shown in figure 1.5

	Row	Col	value
b[0]	6	6	8
b[1]	0	0	15
b[2]	0	4	91
b[3]	1	1	11
b[4]	2	1	3
b[5]	2	5	28
b[6]	3	0	22
b[7]	3	2	-6
b[8]	5	0	-15

Figure 1.5 Transpose of the matrix

Function to find the transpose of a sparse matrix

```
void transpose(term a[], term b[])        /* b is set to the transpose of a */
{
int n,i,j, currentb;
n = a[0].value;                /* total number of elements */
b[0].row = a[0].col;           /* rows in b = columns in a */
b[0] .col = a[0] .row; /* columns in b = rows in a */
b[0].value = n;
```

```
if (n > 0 )        /* non zero matrix */
{
currentb = 1;
for (i = 0; i < a[0].col; i++)      /* transpose by the columns in a */
for (j = 1; j <= n; j++)
        if (a[j].col == i)     /* find elements from the current column */
        {
                b[currentb].row = a[j].col;    /* element is in current column, add it to b */
                b[currentb].col = a[j].row;
                b[currentb].value = a[j].value;
                currentb++;
        }
}
}
```

Analysis of *transpose:* Hence, the asymptotic time complexity of the transpose algorithm is O(*columns·elements*).

Algorithm to Transpose of a two dimensional arrya of size *rows* × *columns*
Input: two dimensional array A of size rows*columns
Output: two dimensional array B of size columns*rows that stores the transpose of A

```
for (i = 0; i < rows; i++)
for (j = 0; j < columns; j++)
b[j][i] = a[i][j];
```

time required is O(*columns·rows*)

Fast Transpose : We can transpose a matrix represented as a sequence of triples inO(*columns* + *elements*) time. This algorithm, *fastTranspose* is listed below .

It first determines the number of elements in each column of the original matrix. This gives us the number of elements in each row of the transpose matrix. From this information, we can determine the starting position of each row in the transpose matrix. We now can move the elements in the original matrix one by one into their correct position in the transpose matrix. We assume that the number of columns in the original matrix never exceeds *MAX_COL*.

Program Fast Transpose

```
void fastTranspose(term a[], term b[])                    /* the transpose of a is placed in b */
{
int rowTerms[MAX_COL], startingPos[MAX_COL];
int i,j, numCols = a[0].col, numTerms = a[0].value;
b[0].row = numCols;
b[0].col = a[0].row;
b[0].value = numTerms;
if (numTerms > 0) { /* nonzero matrix */
        for (i = 0; i < numCols; i++)
                rowTerms[i] = 0;
        for (i = 1; i <= numTerms; i++)
                rowTerms[a[i].col]++;
        startingPos[0] = 1;
```

```
        for (i = 1; i < numCols; i++)
                startingPos[i] = startingPos[i-1] + rowTerms[i-1];

        for (i = 1; i <= numTerms; i++)
         {
                j = startingPos[a[i].col]++;
                b[j].row = a[i].col;
                b[j].col = a[i].row;
                b[j].value = a[i].value;
        }
}
}
```

Analysis of Fast Transpose

- The first two **for** loops compute the values for *rowTerms*, the third **for** loop carries out the computation of *startingPos*, and the last **for** loop places the triples into the transpose matrix. These four loops determine the computing time of *fastTranspose.*
- The bodies of the loops are executed *numCols*, *numTerms*, *numCols* - 1, and *numTerms* times, respectively. The computing time for the algorithm is O(*columns* + *elements*).
- However, *transpose* requires less space than *fastTranspose* since the latter function must allocate space for the *rowTerms* and *startingPos* arrays.

Strings: A string is an array of characters that is delimited by the null character (\0).

Example1: Char s[100] = {“class”} ;
The string is internally represented as follows

C	**L**	**A**	**S**	**S**	**\0**
S[0]	**S[1]**	**S[2]**	**S[3]**	**S[4]**	**S[5]**

The same can also be declared as char s[]={“class”} ;

Using this declaration the compiler would have reserved just enough space to hold each character word including the null character. In such cases we cannot store a string of length more than 5 in s

ADT string is **Objects** : a finite set of zero or more characters

Functions : for all s,t ∈ string,i,j,m ∈ non negative integers

String Null(m) ::=	Return a string whose length is m characters long, but is initially set to NULL. We write NULL as “”
Integer compare(s, t)::=	If s equals t return 0 Else if s precedes t return -1 Else return +1
Boolean ISNull(s) ::=	If (compare(s, NULL)) return FALSE Else return TRUE
Integer Length(s) ::=	If(compare(s, NULL)) Returns the number of characters in s else returns 0
String concat(s,t) ::=	If(compare(t, NULL)) Return a string s whose elements are those of s followed by those of t

String substr(s, i, j) ::=	If((j>0) && (i+j-1)<length(s)) Return the string containing the characters of s at position i to i+j-1 Else return NULL

C provides several string functions which we access by including the header file string.h
Given below is a set of C string functions

char *strcat(char *dest, const char *src)	Appends the string pointed to, by *src* to the end of the string pointed to by *dest*.
char *strncat(char *dest, const char *src, size_t n)	Appends the string pointed to, by *src* to the end of the string pointed to, by *dest* up to n characters long.
int strcmp(const char *str1, const char *str2)	Compares the string pointed to, by *str1* to the string pointed to by*str2*.
int strncmp(const char *str1, const char *str2, size_t n)	Compares first n characters Returns<0 if str1<str2 0 is str1=str2 >0 if str1>str2
char *strcpy(char *dest, const char *src)	Copies the string pointed to, by *src* to *dest and return dest*
char *strncpy(char *dest, const char *src, size_t n)	Copies n characters from the string pointed to, by *src* to *dest* and returns dest
size_t strlen(const char *str)	Returns the length of the string str . But not including the terminating null character.
char *strchr(const char *str, int c)	Returns pointer to the first occurrence of c in str . Returns NULL if not present
char *strrchr(const char *str, int c)	Returns pointer to the last occurrence of c in str . Returns NULL if not present
char *strtok(char *str, const char *delim)	Returns a token from string *str* . Tokens are separated by *delim*.
char *strstr(char *str, const char *pat)	Returns pointer to start of pat in str
size_t strspn(const char *str, const char *spanset)	Scan str for characters in spanset, returns the length of the span
size_t strcspn(const char *str, const char *spanset)	Scans str for character not in spanset, returns the length of the span
char *strpbrk(const char *str, const char *spanset)	Scan str for characters in spanset, returns pointer to first occurrence of a character from spanset

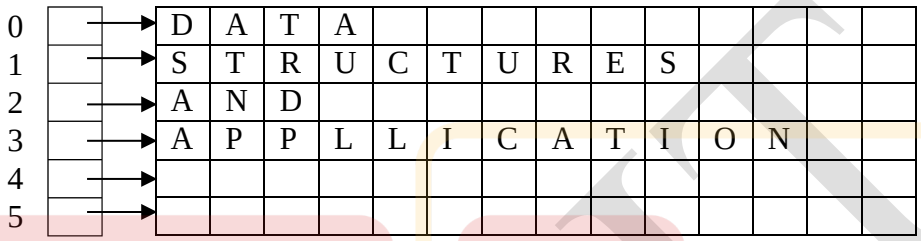
Storing Strings

Strings are stored in three types of structures

1. Fixed Length structure
2. Variable Length structure with fixed maximums
3. Linked structures

1. **Fixed length Storage, record oriented:** In this structure each line of text is viewed as a record where all records have the same length or have the same number of characters

Example: Assuming our record has a maximum of 12 characters per record the strings are stored as follows



Advantages:

- Ease of accessing data from any given record
- Ease of updating data in any given record(provided the length of the new data does not exceed the record length

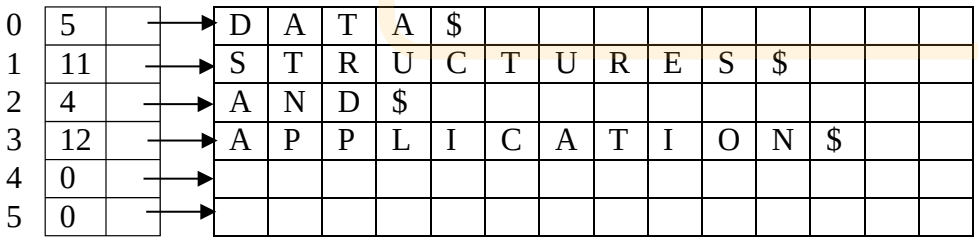
Disadvantages

- Time is wasted reading an entire record if most of the storage consist of in essential blank spaces
- Certain records may require more space or data than available
- When the correction consist of more or fewer characters than original text, updation requires the entire record to be changed(the disadvantage can be resolved by using array of pointers)

2. **Variable Length storage with fixed maximum:** The storage of variable length strings in memory cells wth fixed lengths can be done in two ways

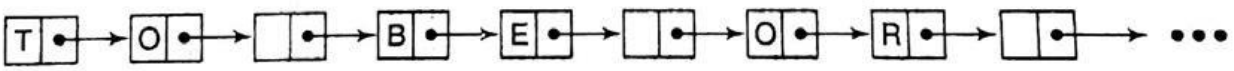
- Use a marker such as ($) to mark the end of the string
- List the length of the string as an additional field in the pointer array

Example :

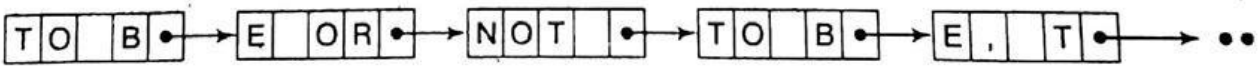


3. **Linked storage:** Linked list is an ordered sequence of memory cells called nodes, where each node stores two information the data and also stores the address of the next node in the list. Strings may be stored in linked list as each node storing one character or a fixed number of characters and a link containing the address of the node containing the next group of characters.

Example:



(a) One character per node.



(b) Four characters per node.

String insertion function

Example: Insert string t in string s at position 1

S =

A	m	o	b	i	l	e	'\0\

t=

U	t	o	'\0\

Initially
Temp=

'\0\

Strncpy(temp,s,i)

a	'\0\

Strcat(temp,t)

a	U	t	o	'\0\

Strcat(temp,(s+i))

a	u	T	o	m	o	b	i	L	e	'\0\

Consider two string str1 and str2 . insert srting str2 into str1 at position i.

```
# include<string.h>
# define max_size 100
Char str1[max_size];
Char str2 [max_size];

Void strins(char *s, char *t, int i)
{
 Char str[max_size], *temp= string;

If (i<0 && i>strlen(s)
{
        Printf(" position is out of bound");
        Exit(0);
}
else
if (strlen(t))
{
```

```
        strncpy(temp,s,i);
        strcat(temp,t);
        strcat(temp,(s+i));
        strcpy(s,temp);
} }
```

Pattern matching : Consider two strings *str* and *pat* where *pat* is a pattern to be searched for in *stri*. The easiest way to find if the pat is in str is by using the built in function strstr.

Example: If we have a declaration as follows

```
Char pat[max_size], str[max_size], *t;
```

The pattern matching can be carried out as follows

```
if((t= strstr(str,pat))
Printf("The pattern found in the string is %s",t);
Else
Printf(" The pattern was not found");
```

Since there are different methods to find pattern matching discussed below are two functions that finds pattern matching in a more efficient way.

The easiest and the least efficient method in pattern matching is sequential search. The computing time is of O(n.m).

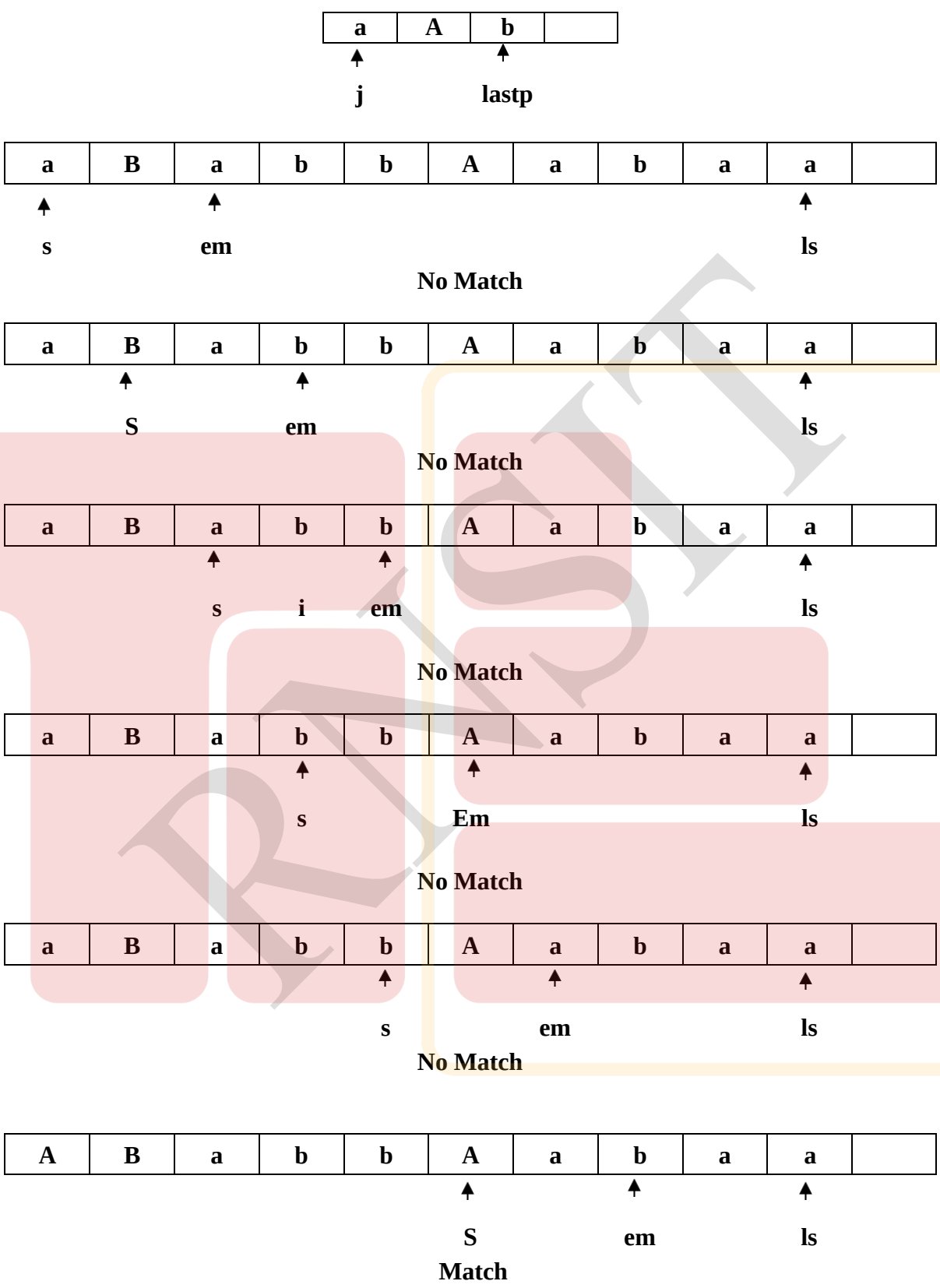
Exhaustive patter matching is improved in nfind function by

- Quitting when strlen(pat) is greater than the number of remaining characters
- Compare the first and last character if pat and string before we check the remaining characters

```
int nfind(char *string,char *pat)
{
Int i,j,start=0;
int lasts=strlen(string)-1;
int lastp=strlen(pat)-1;
int endmatch= lastp

for(i=0;endmatch<=lasts;endmatch++,strt++)
{
        If(string[endmatch] == pat[lastp])
       {
            j=0;i= start;
          while(j<lastp && string[i]== pat[j])
                {
                        i++;
                        j++);
                }
       }
  if(j==lastp)
  return start;
}
return -1
```

}

Simulation of nfind



Analysis of nfind algorithm: The speed of the program is linear O(m) the length of the string in the best and average case but the worst case computing is still O(n.m)

Knuth, Morris, pratt pattern matching algorithm

- Ideally we would like an algorithm that works in O(strlen(str)+ strlen(pat)) time. This is optimal for this problem as in the worst cast it is necessary to look at all characters in the pattern and string at least once.
- We want to search the string for the pattern without moving backwards in the string. That is ifa mismatch occurs we want to use the knowledge of the characters in the pattern and the position in the pattern where the mismatch occurred to determine where the search should continue. Knuth, Morris, and pratt have developed an algorithm that works in this way and has linear complexity.

The following declarations are
assumed.#define
max_string_size 100
#define max_pat_size 100

```
int pmatch(char *string,char *pat)
{
        int i=0,j=0;
        int lens=
        strlen(string); int
        lenp=
        strlen(spat);
        while (i<lens
        && j<lenp)
        {
        if (string[i] == pat[j])
        {
        i++; j++; }
        else    if (j==0)  i++;
        else      j= failure[j-1] +1;
        }
        return ((j=lenp) ? (i-lenp) : -1);
}
```

Example: For the pattern pat=abcabcacab we have the failure values calculated as below

j	1	2	3	4	5	6	7	8	9	10
pat	a	b	c	a	b	c	a	c	a	b
failure	0	0	0	1	2	3	1	3	1	2

Analysis of pmatch: The time complexity of function pmatch is O (m) = O(strlen(string))

Analysis of fail: The computing time of fails is O(n)=O(strlem(pa)).

Therefore when the failure function is not known in advance the total
computing time isO(strlen(string)) + O(strlem(pa))

Stack

Stack Definition and Examples

- Stack is an ordered list in which insertions (also called push) and deletions (also called pops) are made at one end called the top.
- Given a stack S = (a0, ..., an-1), we say that a0 is the bottom element, an-1 is the top element,and ai is on top of element ai-1, 0 < i < n.
- Since the last element inserted into a stack is the first element removed, a stack is also known as a Last-In-First-Out (LIFO) list.

Illustration of stack operations

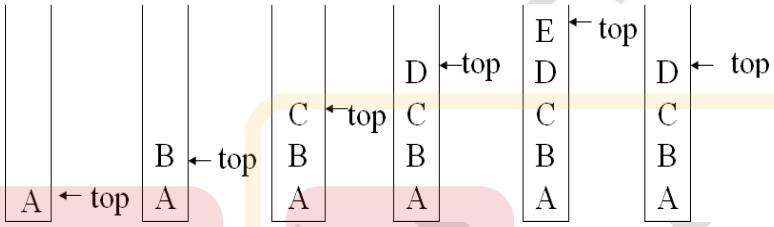


Fig: Illustration of the push and pop operations

Implementation of stack

Stack is implemented by using a one-dimensional array, stack [MAX_STACK_SIZE], where MAX is the maximum number of entries.

- The first, or bottom, element of the stack is stored in stack [0], the second in stack [1], and the ith in stack [i-1].
- Variable top points to the top element in the stack.
- Top= -1 to denote an empty stack.

ADT*Stack* is

objects: a finite ordered list with zero or more elements.

functions: for all *stack*∈*Stack*, *item*∈*element*, *maxStackSize*∈ positive integer

Stack CreateS(maxStackSize) ::=	create an empty stack whose maximum size is maxStackSize
BooleanIsFull(stack, maxStackSize) ::=	**if** (number of elements in stack == maxStackSize) **return**TRUE **else return**FALSE
StackPush(stack, item) ::=	**if** (IsFull(stack)) stackFull **else** insert item into top of stack and **return** BooleanIsEmpty(stack) ::= **if** (stack == CreateS(maxStackSize)) **return**TRUE **else return**FALSE
ElementPop(stack) ::=	**if** (IsEmpty(stack)) **return** **else** remove and return the element at the top of the stack.

Array Implementation of a stack of integers

```
}#include<stdio.h>
#define MAX 10
int top= -1,stack[MAX];

 void push(int item)
{
        if (top==MAX-1)
        printf("Stack Overflow\n");
        else
        stack[++top]=item;
}

 int  pop()
{
         int itemdel;
        if (top==-1)
        return 0;
        else
        {
                itemdel=stack[top--];
                return itemdel;
        }
}

 void display()
{
        int i;
        if(top==-1)
        printf("Stack Empty\n");
        else
        {
                printf("Elements Are:\n");
                for(i=top;i>=0;i--)
```

```
                printf("%d\n",stack[i]);
        }
}

 void main()
{
        int ch,item,num,itemdel;
        while(1)
        {
                printf("\nEnter the Choice\n1.Push\n2.Pop\n3.Display\n4.Exit\n");
                scanf("%d",&ch);
                switch(ch)
               {
                        case 1: printf("Enter item to be inserted\n");
                                scanf("%d",&item);
                                push(item);
                                break;

                        case 2: itemdel=pop();
                                if(itemdel)
                                printf("\n Deleted Item is:%d\n",itemdel);
                                else
                                printf("Stack Underflow\n");
                                break;
                        case 3: display();
                                break;
                        case 4: exit(0);
                }
 }
```

2.1.3 Stacks Using Dynamic Arrays

If we do not know the maximum size of the stack at compile time, space can be allocated for the elements dynamically at run time and the size of the array can be increases as needed.

Creation of stack: Here the capacity of the stack is taken as 1. The value of the capacity can be altered specific to the application

StackCreateS() ::=

```
int *stack
Stack=(int*)malloc(stack, sizeof(int));
int capacity = 1;
int top = -1;
```

BooleanIsEmpty(Stack) ::= top < 0;

BooleanIsFull(Stack) ::= top >= capacity-1;

The function push remains the same except that MAX_STACK_SIZE is replaced with capacity

```
void push(element item)
{
        if (top >=capacity-1)
        stackFull();
        stack[++top] = item;
}
```

The code for the pop function remains unchanged element

```
pop()
{/* delete and return the top element from the stack */
        if (top == -1)
        return stackEmpty(); /* returns an error key */
        return stack[top--];
}
```

Stackfull with Array doubling: The code for stackFull is changed. The new code for stackFull attempts to increase the capacity of the array stack so that we can add an additional element to the stack. In array doubling, the capacity of the array is doubled whenever it becomes necessary to increase the capacity of an array.

```
void stackFull()
{
        stack=(int*)realloc(stack, 2 * capacity * sizeof(int))
        capacity =capacity * 2;
}
```

Analysis

- In the worst case, the realloc function needs to allocate 2*capacity *sizeof (*stack) bytes of memory and copy capacity*sizeof (*stack)) bytes of memory from the old array into the new one.
- Under the assumptions that memory may be allocated in O(1) time and that a stack element can be copied in O(1) time, the time required by array doubling is O(capacity). The total time spent in array doubling is of O(n) where n is the total number of push operations.
- Hence even with the time spent on array doubling in the total time of push over all n pushes in O(n). This conclusion is valid even the stack array is resized by a factor c>1.

Application of stack

- Conversion of Expression
- Evaluation of expression
- Recursion

Infix, postfix(Suffix) and Prefix(polish)

- Expression is a collection of operands and operators
- An expression can be represented in three different ways.
  - Infix expression: operators are in between the two operands . Example a+b
  - Prefix expression: operators are present before the operands. Example +ab
  - Postfix expression: operators are present after the operands. Example ab+
- The prefixes “pre”, “post”, and “in” refer to the relative position of the operator with respect to the two operands.

To convert an expression from infix to prefix or postfix we follow the rules of precedence.

- Precedence : The order in which different operators are evaluated in an expression is called precendence
- Associativity : The order in which operators of same precedence are evaluated in an expression is called Associativity.

The operators are listed in the order of higher precedence down to lower precedence

Operator	Associativity
--,++	left-to-right
Unary operators ~,!,-,+, &, *,sizeof	Right to left
*,/,%	left-to-right
+,-	left-to-right

Converting an expression from infix to postfix

The operands in the infix and the postfix expression are in the same order. With respect to operators , precedence of operators plays an important role in converting an infix expression to postfix expression. We make use of the stack to insert the operators according to their precedence.

The following operations are performed to convert an infix expression to postfix.

- Scan the symbol character by character
  - If the symbol is an operand place it in the postfix string
  - If the symbol is an opening parenthesis push it on to the stack
  - If the symbol is a closing parenthesis pop the contents of the stack until we see an opening parenthesis and place it in the postfix string. The opening parenthesis and the closing parenthesis is not placed in the postfix string.
  - If the symbol is an operator and if the precedence of the input symbol is more than the precedence of the symbol on top of the stack, then the operator is pushed on to the stack. If the precedence of the input symbol is lesser than the symbol on top of the stack we pop each such operators and place it in the postfix string

Algorithm Polish(Q,P)

Suppose Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent Postfix expression P.

1. Push '(' on to STACK to identify the end of the stack
2. Scan Q from Left to right and repeat steps 3 to 6 for each character of Q until the end of the string
   3. If an operand is encountered add it to P
   4. If a left parenthesis is encountered ,push it to onto STACK.
   5. If an operator ⊗ is encountered then
      a. Repeatedly pop each operator that has equal or higher precedence and add to P.
      b. Add ⊗ to Stack
6. If a right Parenthesis is encountered, then
   a. Repeatedly pop from stack and add to P each operator on top of stack until until a left parenthesis is encountered.
   b. Remove the left Paranthesis.[do not add left parenthesis to stack]
   [End of If Structure.]

[End of Step 2 loop]

7. Repeatedly pop stack until it is empty and add to P
8. Exit

Function to convert an infix expression to postfix

Assume the stack and the top is declared globally

```
char s[25];
int top=-1;

int precd(char op)
{
        int r;
        switch(op)
        {
                Case '^':
                Case '$' r=3; break;
                case '*':
                case '/': r=2;break;
                case '+':
                case '-': r=1;break;
                case '(': r=0;break;
                case '#': r=-1;break;
        }
        return(r);
}

void infix_postfix(char infix[],char postfix[])
{
          int i,p=0;
```

```
        char symbol,item;
        push('#');
        for (i=0;infix[i]!='\0';i++)
        {
                symbol=infix[i];
                switch(symbol)
                {
                        case '(': push(symbol);
                        break;

                        case ')':item=pop();
                        while(item!='(')
                        {
                                postfix[p++]=item;
                                item=pop();
                        }
                        break;
                        case '+':
                        case '-':
                        case '*':
                        case '/':
                        case '%':while(precd(s[top])>=precd(symbol))
                        {
                                item=pop();
                                postfix[p++]=item;
                        }
                        push(symbol);
                        break;

                        case '^': while(precd(s[top])>precd(symbol))
                        {
                                item=pop();
                                postfix[p++]=item;
                        }
                        push(symbol);
                        break;
                        default: postfix[p++]=symbol;
                        break;
                }
        }
        while(top>0)
        {
                item=pop();
                postfix[p++]=item;
        }
        postfix[p]='\0';
}
```

Example: Translation of the infix string a*(b+c)*d to postfix

Infix Token	Stack				Postfix
	[0]	[1]	[2]	[3]	
a	#				a
*	#	*			a
(	#	*	(		a
b	#	*	(		ab
+	#	*	(	+	ab
c	#	*	(	+	abc
)	#	*			abc+
*	#	*			abc+*
d	#	*			abc+*d
Eos(\0)	#				abc+*d*

Analysis: Let n be length of the infix string. Θ(n) time is spent extracting tokens . There are two while loop where the total time spent is Θ(n) since the number of tokens that get stacked and unstacked is linear in n . So the complexity of the function is Θ(n)

Evaluating a postfix expression

Each operator in a postfix string refers to the previous two operands in the string. If we are parsing a string, each time we read operands we push it to the stack and when we read a operator, its operands will be the two topmost elements in the stack. We can then pop these two operands and perform the indicated operation and push the result back to the stack so that it can be used by the next operator.

Example: Evaluation of postfix string 6 2 3 + - 3 8 2 / + * 2 $ 3 +

Token	Stack content				Operand1	Operand2	Result
	[0]	[1]	[2]	[3]			
6	6						
2	6	2					
3	6	2	3				
+	6	5			2	3	5
-	1				6	5	1
3	1	3					
8	1	3	8				
2	1	3	8	2			
/	1	3	4		8	2	4
+	1	7			3	4	7
*	7				1	7	7
2	7	2					
$	49				7	2	49
3	49	3					
+	52				49	3	52

The following function evaluates a postfix expression using a stack and a stack of float elements is declared globally

```
float s[25];
int top;

 float Evaluate(char *postfix)
{
        int i=0;
        float res=0,op1,op2;
        char symbol;
        while(postfix[i]!='\0')
        {
                symbol=postfix[i];
                if isdigit(symbol)
                {
                        push(symbol-'0');
                }
                else
                {
                        op2=pop();
                        op1=pop();
                        res=operation(symbol,op1,op2);
                        push(res);
                }
        }
        res=pop();
        return(res);
}

 float operation(char op, float op1,float op2)
{
        float res;
        switch(op)
        {
                case '+': res= op1+op2;
                case '-': res= op1-op2;
                case '*': res= op1*op2;
                case '/': res= op1/op2;
                case '^': res= pow(op1,op2);
        }
         return (res);
}
```

Limitations of the program

- It does not check if the postfix expression is valid or not. If we input erroneous expression it returns wrong result
- We cannot enter negative numbers, as the symbol to indicate negation will be misinterpreted as subtraction operation

Analysis: Let n be length of the postfix string then the complexity of the function is Θ(n)

Algorithm PostfixEval(P)

This algorithm finds the VALUE of an arithmetic expression P written in postfix notation

1. Scan P from Left to righ
2. Repeats steps 3 and 4 until we reach the end of P
3. If an operand is encountered put it on stack
4. If an operator ⊗ is encountered then:
    a) remove two top elements of STACK, where A is the top element and B is the next top- element
    b) Evaluate B ⊗ A
    c) Place the result of (4) in STACK.

   [End of If structure]

   [End of step 2 Loop]
5. Set VALUE equal to the top element of STACK.
6. EXit

Recursion: Recursion is the process of defining an object in terms of a simpler case of itself.

Suppose p is a function containing either a call statement to itself (direct recursion) or a call statement to a second function that may eventually result in a call statement back to the original function P(indirect recursion). Then the function P is called a recursive function.

A recursive function must have the following two properties.

- There must be a certain argument called the base value for which the function will not call itself.
- Each time the function refers to itself the argument of the function must be closer to the base value.

Factorial function: The factorial of a number n is got by finding the product of all the numberform 1 to n. ie 1*2*3…*n.. It is represented as n!

Example 4!=4*3*2*1=24

5!=5*4*3*2*1=120

0!=1

From a close observation it can be observed that 5!= 5*4! . Therefore n!=n*(n-1)!

Accordingly the factorial function can be defined as

Factorial function definition

a) If n=0, then n!=1
b) If n>0, then n!=n*(n-1)!

- The definition is recursive since the function refers to itself for all value of n>0.
- The value of n! is explicitly given as 1 when the value of n=0 which can be taken as the base value.
- This can be implemented by the code

```
factorial(int n)
{
        f=1;
        for(i=1;i<=n;i++)
        f=f*i;
        return(f)
}
```

This is the iterative implementation of the factorial function

The factorial function can be implemented as a recursive function.

For example

```
factorial (5)=5*factorial(4)
factorial (4)=4*factorial(3)
factorial (3)=3*factorial(2)
factorial(2)=2*factorial(1)
```

```
factorial(int n)
{
        if (n==0 )
        return(1);
        else
        return(n*factorial(n-1))
}
```

Fibonacci numbers in C

- Fibonacci sequence is a sequence of integers. 0 1 1 2 3 5 8 ….. fib(0) =0 and fib(1)=1 then each element in this sequence is the sum of the previous two numbers.
- fib(n)=n if n=0 or if n=1
- fib(n)=fib(n-2)+ fib(n-1) if n>=2

Recursive function to generate the nth Fibonacci number

```
int fib(int n)
{
        if  (n<=1)
        return(n);
        else
        return(fibo(n-1)+fibo(n-2));
}
```

Example:

```
Fibo(4)= fibo(3)+fibo(2)
        =fibo(2)+fibo(1)+fibo(2)
        =fibo(1)+fibo(0)+fibo(1)+ fibo(2)
        = 1+ fibo(0)+fibo(1)+ fibo(2)
        =1+0 + fibo(1)+ fibo(2)
```

=1+0+1+fibo(2)
=2+ fibo(1)+ fibo(0)
=2+1+fibo(0)
=2+1+0= 3

GCD of two numbers: The function accepts two numbers as input and returns the gcd of the two numbers

```
gcd(int m, int n)
{
        if (n==0)
        return m;
        retrun(gcd(n,m%n))
}
```

Example:

gcd(2,0)=2
gcd(0,2)= gcd(2,0)=2
gcd(4,2)= gcd(2,0)=2
gcd(7,3)= gcd(3,1)= gcd(1,0)=1

Binary search in C

```
int binsearch(int *a , int key, int low, int high)
{
        If (low>high)
        return(-1);
        mid=(low+high)/2;
        if (key==a[mid])
        return(mid)
        else
        if (key>a[mid])
        return(binsearch(a,key,mid+1,high));
        else
        return(binsearch(a,key,low,mid-1));
}
```

The Tower of Hanoi problem :

- We have to move n disks from peg A to peg C using peg B as auxiliary disk.
- The disks are placed in such a way that the larger disks are always below a smaller disk.
- Only the top disk on any peg can be moved to any other peg.
- Larger disk should never rest on a smaller disk.

A recursive solution to the tower of Hanoi problem is given as follows

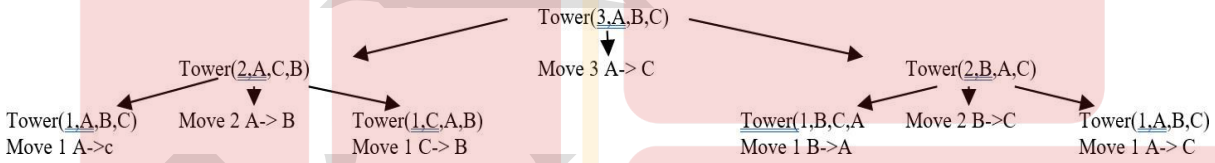
- To move n disks from A to c using B as auxiliary
  - If n==1 move the single disk from A to C and stop
  - Move the top n-1 disks form A to B using C as auxiliary
  - Move the n^{th} disk from A to C
  - Move n-1 disk from B to C using A as auxiliary

Write a program to solve the Tower of Hanoi problem using a recursive function

```
void tower(int n,char source,char temp,char dest)
{
        if(n==1)
         {
                printf("Move disc 1 from %c to %c\n",source,dest);
                count++;
                return;
        }
        tower(n-1,source,dest,temp);
        printf("Move disc %d from %c to %c\n",n,source,dest);
        count++;
        tower(n-1,temp,source,dest);
}

Void main()
{
          int n,count;
           printf("Enter the number of discs\n");
           scanf("%d",&n);
           tower(n,'A','B','C');
           printf("The number of moves=%d\n",count);
}
```

Example: n=3 disks

Tracing of function with n=3 disks

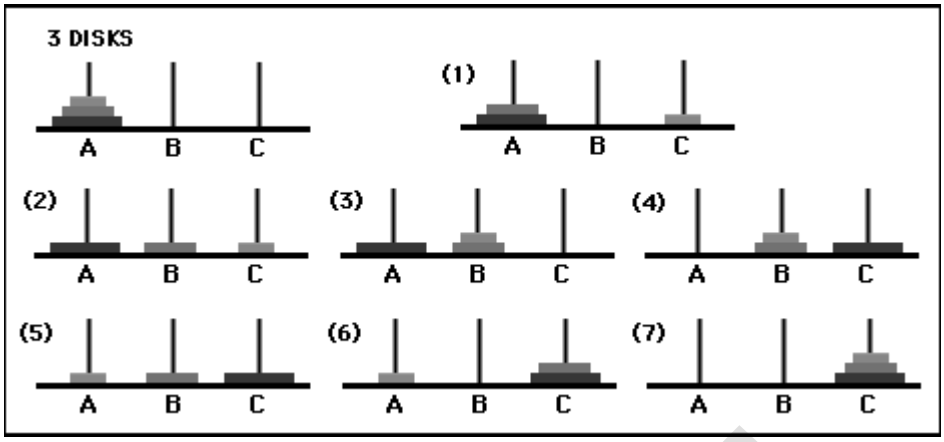


The moves for n=3 disk is summarized below.

Move 1 from A to C
Move 2 from A to B
Move 1 from C to B
Move 3 from A to C
Move 1 from B to A
Move 2 from B to C
Move 1 from A to C

Note: Ideal number of moves to solve Tower of Hanoi is given as 2^n -1 where n is the total number of disks

Problem solved with n=3 disks



Ackermann Function

The Ackermann function is a function with two arguments each of which can be assigned any non negative integer 0,1,2…..... This function finds its application in mathematical logic.
This function is defined as follows

a) If m = 0 then A(m,n) = n + 1
b) If m != 0 but n = 0 then A(m,n) = A(m - 1,1)
c) If m != 0 and n != 0 then A(m,n) = A(m - 1, A(m,n - 1))

The function is given as follows

```
int A(int m,int n)
{
        if (m == 0)
        return(n+1);
        if ((m !=0) && (n== 0 ))
        return(A(m-1,1));
        if ((m!= 0) && (n != 0))
        return(A(m-1, A(m,n-1)));
}
```

Example1:

A(1,2)	=A(0,A(1,1))
	=A(0,A(0,A(1,0)))
	= A(0,A(0,A(0,1))
	=A(0,A(0,2)
	=A(0,3)
	4

Example 2:

A(1,3)	= A(0,A(1,2))
	=A(0,A(0,A(1,1)))
	=A(0,A(0,A(0,A(1,0))))
	=A(0,A(0,A(0,A(0,1))))
	=A(0,A(0,A(0,2)
	=A(0,A(0,3)
	=A(0,4)
	5

Disadvantages of using Recursion

- Non recursive version of the program is more efficient when compared to recursive version interms of time and space
- Non recursive versions do not have the overhead of entering and exiting from the block.
- In recursive program the local variables have to be maintained using a stack,which can beavoided in an iterative(non recursive) version

Advantage of using Recursion

- Recursive program is the most natural and logical way of solving some of the problems
- If a stack can be eliminated in a recursive program without having any local variables thenthey can be as fast its non recursive version,

Difference between iterative and recursive functions

Iterative	**Recursive**
Implemented using looping statements	Implemented using recursive calls to functions
Executes faster	Takes more time to execute
Memory utilization is Less	Memory utilization is more
Lines of code are more	Lines of code are lesser
Does not require stack	Implementation requires stack

MODULE-2

QUEUES: Queues, Circular Queues, Using Dynamic Arrays, Multiple Stacks and queues.

LINKED LISTS : Singly Linked, Lists and Chains, Representing Chains in C, Linked Stacks and Queues, Polynomials

Queues:

A queue is an ordered list in which insertions and deletions take place at different ends. The end at which new elements are added is called the rear, and that from which old elements are deleted is called the front. Queues are also known as First-In-First-Out (FIFO) lists.

The ADT specification of the queue

objects: a finite ordered list with zero or more elements.

functions: for all queue∈Queue, item∈element, maxQueueSize∈ positive integer

Queue CreateQ(maxQueueSize) ::=	create an empty queue whose maximum size is maxQueueSize
BooleanIsFullQ(queue, maxQueueSize) ::=	**if** (number of elements in queue == maxQueueSize) **return**TRUE **else return**FALSE
QueueAddQ(queue, item) ::=	**if** (IsFullQ(queue)) queueFull **else** insert item at rear of queue and return queue
BooleanIsEmptyQ(queue) ::=	**if** (queue == CreateQ(maxQueueSize)) **return**TRUE **else return** FALSE
Element DeleteQ(queue) ::=	**if** (IsEmptyQ(queue)) **return** **else** remove and return the item at front of queue.

Example
Initially f =-1 r=-1 queue empty

Element						
index	[0]	[1]	[2]	[3]	[4]	[5]

f=-1
Insert 3

Element	3					
index	[0]	[1]	[2]	[3]	[4]	[5]
	,r					

f=-1
Insert 5

Element	3	5				
index	[0]	[1]	[2]	[3]	[4]	[5]
		r				

f=-1
Insert 7

Element	3	5	7			
Index	[0]	[1]	[2]	[3]	[4]	[5]
			r			

delete

Element		5	7			
Index	[0]	[1]	[2]	[3]	[4]	[5]
	F		r			

Deleted item =3

C implementation of queues for an integer array: A queue can be represented by using an array to hold the elements of the array and to use two variables to hold the position of the first and lastelement of the queue.

```
#define size 10
int q[size];
int front=-1 ,rear=-1;
```

- The condition where the queue is empty is called **queue underflow.**
- When we define the size of the array, we cannot insert elements more than the size of the array, this condition where the queue is full is called as **queue overflow.**

Insert operation

The insert operation first checks for queue overflow. If the queue is not full it inserts one element into the queue at the rear.

```
Void insert(int item)
{
        If rear==size-1)
        Printf("queue overflow");
else
{
        rear++;
        q[rear]=item;
}
}
```

Delete operation: Delete operation checks for queue underflow condition and if the queue is not empty it will remove the element at the front.

```
int delete()
{
        int itemdel;
        if (front ==rear)
        {
                Printf("queue underflow");
                return(0);
        }
        else
        {
                front++
                itemdel=q[front];
                return(itemdel);
        }
}
```

Display operation: The display operation will display the elements of the queue if the queue is not empty.

```
void display(s)
{
        if (front==rear)
        printf("queue empty");
        else
                {
                for(i=front+1;i<=rear;i++)
                printf("%d",q[i]);
                }
}
```

Disadvantage of linear queue: The following example illustrates the disadvantage of linear queue

f	r	Q[0]	Q[1]	Q[2]	Q[3]	Q[4]	Comment
-	-1						Queue is empty
-1	0	2					Insert 2
-1	1	2	3				Insert 3
-1	2	2	3	4			Insert 4
-1	3	2	3	4	5		Insert 5
-1	4	2	3	4	5	6	Insert 6
0	4		3	4	5	6	Delete- item deleted=2
1	4			4	5	6	Delete- item deleted=3
							Insert 3(disadvantage) Queue full. Element cannot be inserted since rear= size-1

Even if the queue is empty since the value of rear= size-1 elements cannot be inserted into the queue. This is the disadvantage of linear queue.

Circular Queue: In a circular queue the queue is arranged in a circular fashion. It is a moreefficient representation of queue.

Example:

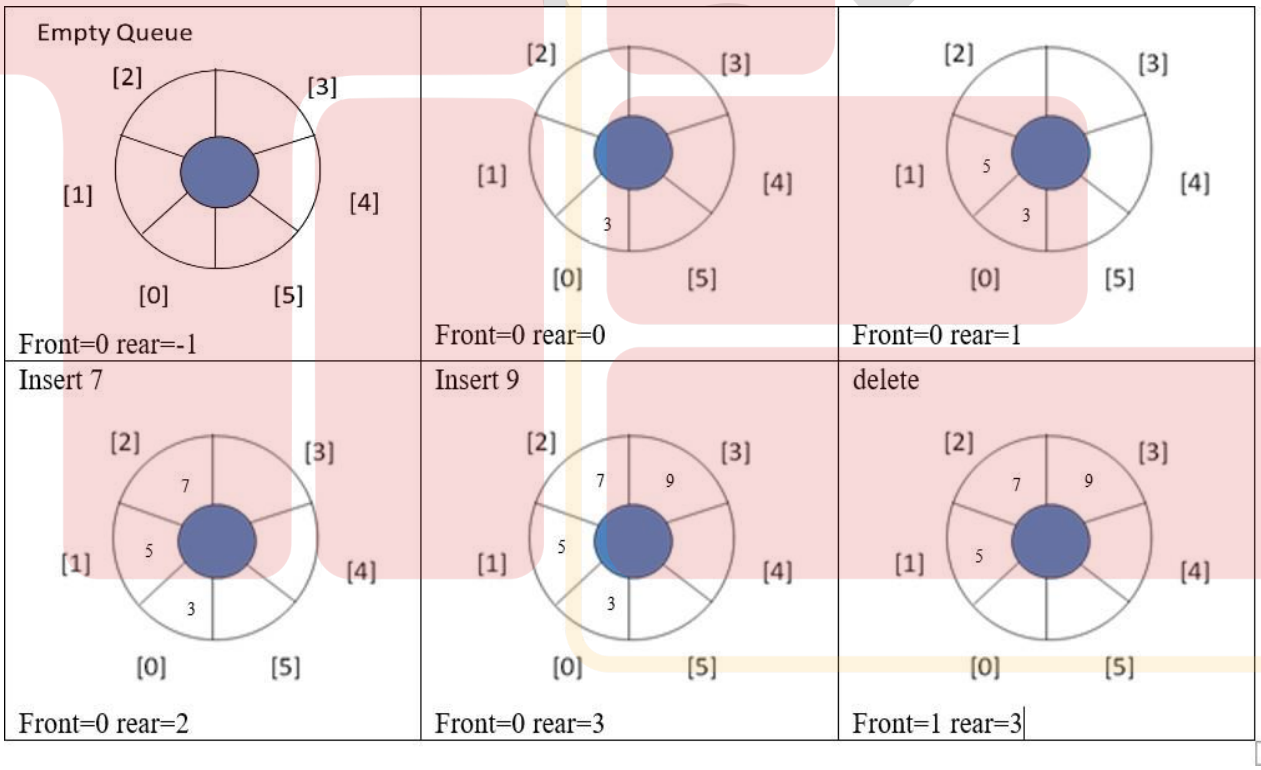


Fig: Implementation of circular queue for an array of integers

Modification in code to implement circular queue

- Initial value of front and rear=0
- To advance the pointer from size-1 to 0 the code rear++ is modified to rear=(rear +1)%size and the code front++ is modified to front=(front+1)%size

- Initially when front=0 rear=0 i.e when front == rear the queue is empty now after 6 insertions are made again fron=0 and rear= 0 that is the queue is full. So, we cannot distinguish between an empty and a full queue.
- To avoid the resulting confusion, the value of the rear is incremented before we check for the condition front == rear for queue overflow.

```
#define MAX_QUEUE_SIZE 6
int q[size];
int front=0 ,rear=0;
```

Insert operation: The insert operation first checks for queue overflow. If the queue is not full it inserts one element into the queue at the rear.

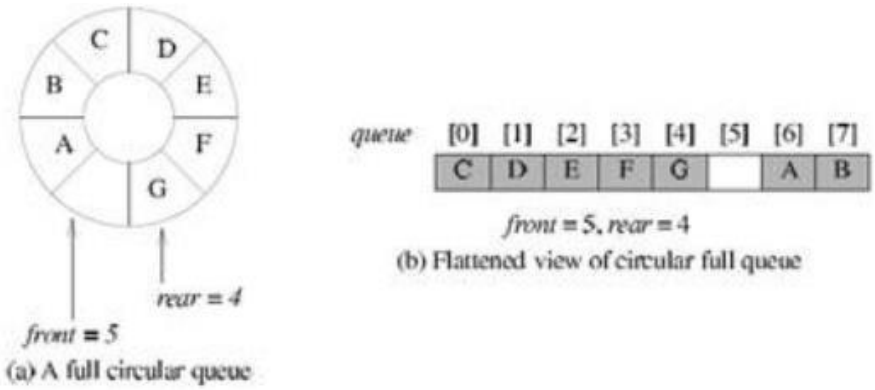
```
void addq(element item)
{
        if (front == (rear+1) % MAX_QUEUE_SIZE )
        printf("Queue full");
        else
        {
                rear= (rear+1) % MAX_QUEUE_SIZE
                queue[rear] = item;
        }
}
```

Delete operation: Delete operation checks for queue underflow condition and if the queue is not empty it will remove the element at the front.

```
element deleteq()
{
        element item;
        if (front == rear)
        return queueEmpty();
        front = (front+1) % MAX_QUEUE_SIZE;
        return queue[front];
}
```

Circular Queues Using Dynamically Allocated Arrays

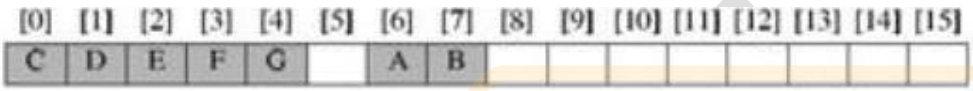
- To add an element to a full queue, we must first increase the size of this array using a function such as realloc.
- As with dynamically allocated stacks, we use array doubling. However, it isn't sufficient to simply double array size using realloc.
- Consider the full queue . This figure shows a queue with seven elements in an array whose capacity is 8. To visualize array doubling when a circular queue is used, the array is flattened out as shown in the array of Figure (b).



(a) A full circular queue

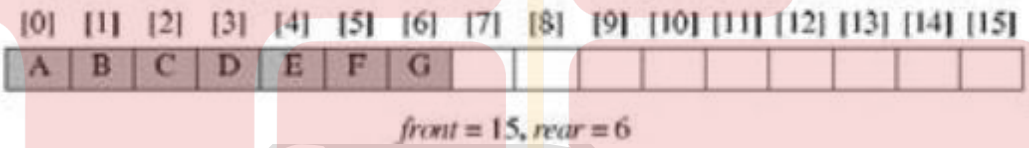
(b) Flattened view of circular full queue

Figure (c) shows the array after array doubling by realloc.



(c) After array doubling

To get a proper circular queue configuration, The number of elements copied can be limited to capacity - 1 by customizing the array doubling code so as to obtain the configuration as shown below.



This configuration may be obtained as follows:

1. Create a new array newQueue of twice the capacity.
2. Copy the second segment (i.e., the elements queue [front + 1] through queue [capacity - 1]) to positions in newQueuebeginning at 0.
3. Copy the first segment (i.e., the elements queue [0] through queue [rear]) to positions in newQueue beginning at capacity-front-1.

Function to add to a circular queue has no change

```
void addq(int item)
{
        rear = (rear+1) % capacity;
        if (front == (rear+1) % capacity)
        queueFull(); /* double capacity */
        else
        {
                rear = (rear+1) % capacity;
                queue[rear] = item;
        }
}
```

Function to double queue capacity

```
void queueFull()
{
        int* newQueue;
        newQueue=(int*)malloc( 2 * capacity * sizeof(int));
                                                        /* copy from queue to newQueue */
        int start = (front+1) % capacity;
        if (start < 2)
        copy(queue+start, queue+start+capacity-1, newQueue);
        else
        {                                                       /* queue wraps around */
        copy(queue+start, queue+capacity, newQueue);
        copy(queue, queue+rear+1, newQueue+capacity-start);
        }

        /* switch to newQueue */
        front= 2 * capacity - 1;
        rear = capacity - 2;
        capacity *= 2;
        free(queue);
        queue = newQueue;
}
```

The function copy(a,b,c) copies elements from locations a through b-1 to locations beginning at c

Deques: A deque (pronounced either as deck or dequeue) is a linear list in which elements can be added or removed at either end but not in the middle. The term deque is a contraction of the namedouble ended queue.

Representation: It is represented as a circular array deque with pointers left and right, which point to the two ends of the queue. It is assumed that the elements extend from the left end to the right end in the array. The term circular comes from the fact that DEQUE[0] comes after DEQUE[n-1] in the array.

Example1:

Left=2 Right=4			A	B	C		
	[0]	[1]	[2]	[3]	[4]	[5]	[6]

Example2:

Left=5 Right=1	A	B				D	E
	[0]	[1]	[2]	[3]	[4]	[5]	[6]

There are two types of deque:

- **Input restricted**:- insertion is allowed only at one end, deletion is allowed at both ends.
- **Output restricted**: deletion is allowed only at one end, insertion is allowed at both ends

Priority queue: A priority queue is a collection of elements such that each element has been assigned a priority such that the order in which the elements are deleted and processed comes from the following rules.

1. An element of higher priority is processed before any element of lower priority.
2. Two elements with the same priority are processed according to the order in which they were added to the queue.

Example: Time sharing system: programs of higher priority are processed first and programs with the same priority form a standard queue.

Representation using multiple queue: Use a separate queue for each level of priority. Each queue will appear in its own circular array and must have its own pair of pointers, front and rear. If each queue is allocated the same amount of space, a two dimensional array can be used instead of the linear arrays.

Example : Consider the queue given below with the jobs and its priorities and its representation. A job with priority 1 is considered to have the highest priority

J1	1
J2	1
J3	2
J4	4
J5	4
J6	6

The queue is represented as a two dimensional array as shown below

	Front	rear
1	1	2
2	2	3
3		
4	4	5
5		
6	6	6

	1	2	3	4	5	6
1	J1	J2				
2			J3			
3						
4				J4	J5	
5						
6						J6

Delete operation

Algorithm:

1. Find the smallest k such that front[k]!=rear[k] ie Find the first non empty queue
2. Delete the process at the front of the queue
3. Exit

Insert operation

Algorithm: this algorithm adds an ITEM with priority number P to a priority queue maintained by a two dimensional array

1. Inset ITEM as the rear element in row P-1 of queue
2. exit

Multiple Stacks and Queues

- If there is a single stack, the starting point is top=-1 and maximum size is SIZE-1
- If there are two stacks to be represented in a single array then we use stack [0] for the bottom element of the first stack, and stack[MEMORY_SIZE - 1] for the bottom element of the second stack. The first stack grows toward stack[MEMORY_SIZE - 1] and the second grows toward stack[0]. With this representation, we can efficiently use all the available space.
- Representing more than two stacks within the same array poses problems since we no longer have an obvious point for the bottom element of each stack. Assuming that we have n stacks, we can divide the available memory into n segments. This initial division may be done in proportion to the expected sizes of the various stacks, if this is known. Otherwise, we may divide the memory into equal segments.
- Assume that i refers to the stack number of one of the n stacks.To establish this stack, we must create indices for both the bottom and top positions of this stack.The convention we use is that
    - bottom [i], $0 \le i <$ MAX_STACKS, points to the position immediately to the left of the bottom element of stack i.
    - top[i], $0 \le i <$ MAX_STACKS points to the top element.
    - Stack i is empty if bottom[i] = top[i].

The relevant declarations are:

```
#define MEMORY_SIZE 100          /* size of memory */
#define MAX_STACKS 10            /* max number of stacks */
/* global memory declaration */
element stack[MEMORY_SIZE];
int top[MAX_STACKS];
int bottom[MAX_STACKS];
int n;                           /* number of stacks entered by the user */
```

To divide the array into roughly equal segments we use the following code:

```
top[0] = bottom[0] = -1;
for (j = 1; j < n; j++)
top[j] = bottom[j] = (MEMORY_SIZE/n)*j-1;
bottom[n] = MEMORY_SIZE-1;
```

Stack i can grow from bottom[i] + 1 to bottom [i + 1] before it is full. Boundary for the last stack, boundary [n] is set to MEMORY_SIZE- 1

Initial configuration of the stack is shown below m is the size of the memory

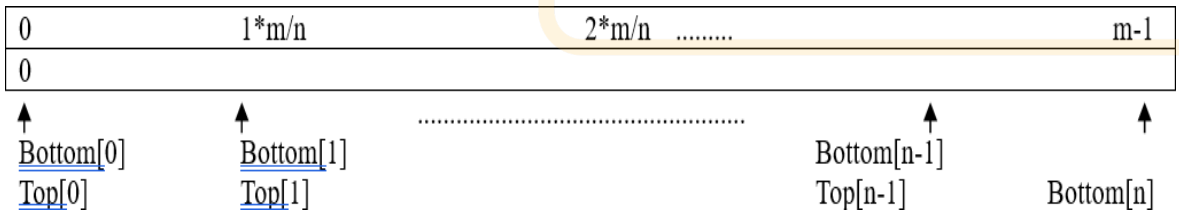


Fig: All stacks are empty and roughly divided equally

Function to add an item to the i^{th} stack

```
void push(int i, element item)
{
        if (top[i] == bottom[i+1])
        stackFull(i);
        stack[++top[i]] = item;
}
```

Function to delete an item from the ith stack

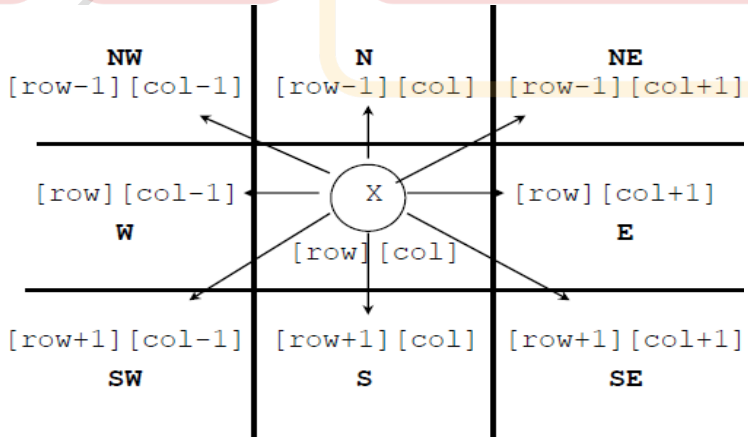
```
element pop(int i)
{
        if (top[i] == bottom[i])
        return stackEmpty(i);
        return stack[top[i]--];
}
```

Mazing Problem

Representation: Maze is represented as a two dimensional array in which zeros represent the open paths and ones the barriers. The location in the maize can be determined by the row number and column number Figure below shows a simple maze.

0	1	0	0	0	1	1	0	0	0	1	1	1	1	1
1	0	0	0	1	1	0	1	1	1	0	0	1	1	1
0	1	1	0	0	0	0	1	1	1	1	0	0	1	1
1	1	0	1	1	1	1	0	1	1	0	1	1	0	0
1	1	0	1	0	0	1	0	1	1	1	1	1	1	1
0	0	1	1	0	1	1	1	0	1	0	0	1	0	1
0	1	1	1	1	0	0	1	1	1	1	1	1	1	1
0	0	1	1	0	1	1	0	1	1	1	1	1	0	1
1	1	0	0	0	1	1	0	1	1	0	0	0	0	0
0	0	1	1	1	1	1	0	0	0	1	1	1	1	0
0	1	0	0	1	1	1	1	1	0	1	1	1	1	0

From each location there are eight directions of movement N, NE,E,SE,S,SW,W,NW.



If the position is on the border then there are less than eight directions. To avoid checking for border conditions the maze can be surrounded by a border of ones. Thus an m*p maize will require an (m+2)*(p+2) array the entrance is at position [1][1] and exit is at [m][p]. The possible direction to move can be predefined in an array move as shown below where the eight possible directions are numbered from 0 to 7. For each direction we indicate the vertical and horizontal offset.

```
typedef struct offset
{
        Int vert;
        Int horiz;
};

Offset move[8];
```

Table of moves: The array moves is initialized according to the table given below.

Name	Dir	Move[dir].vert	Move[dir].horiz
N	0	-1	0
NE	1	-1	1
E	2	0	1
SE	3	1	1
S	4	1	0
SW	5	1	-1
W	6	0	-1
NW	7	-1	-1

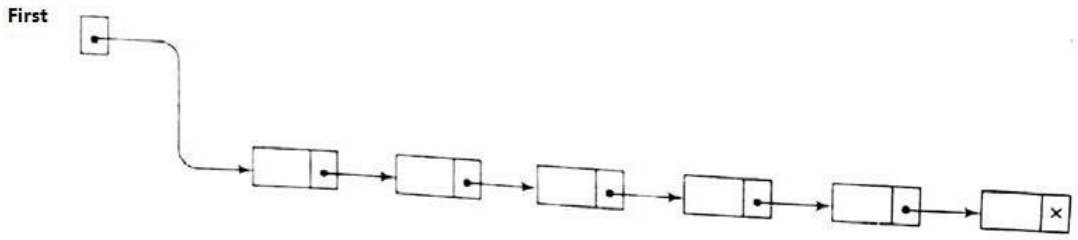
As we move through the maze, we may have the choice of several directions of movement. Since we do not know the best choice we save our current position and arbitrarily pick a possible move. By saving the current position we can return to it and try another path. A second two dimensional array can be maintained to keep track of the visited locations. The maze can be implemented by making use of a stack where the element is defined as follows.

```
typedef struct element
{
        int row;
        int col;
        int dir;
};
```

Linked List

Introduction

Linked list is a collection of zero or more nodes ,where each node has some information. Given the address of the first node, any node in the list can be obtained. Every node consis of two parts one is the information part and the other is the address of the next node. The pointer of the last node contains a special value called NULL.



Schematic representation of Linked List

Advantages of array representation(static allocation)

1. Data accessing is faster: Using static allocation, we can access any data element in the array efficiently by specifying the array index for the required element. The accessing time for a[0] is equal to a[1000].
2. Array's are simple: Arrays are very simple to understand and it is simple to use arrays in programming.

Disadvantages of Arrays (Static allocation)

1. Array size is fixed: In static allocation method, it is required to declare in advance the amount of memory to be utilized.
2. The array elements are stored continuously so the size of the array cannot be increased.
3. Insertion and deletion of elements in an array is expensive as more time is spent shifting the data.

Advantages of linked list

1. Size is not fixed
2. Data can be stored in non contiguous blocks
3. Insertion and deletion is efficient

Disadvantage of Linked List

1. More memory is required as each node stores the address of the next node
2. Difficult to access arbitrary element

The different types of linear linked lists are :

1. Singly linked lists
2. Circular singly linked lists
3. Doubly linked lists
4. Circular doubly linked lists.

Representation of linked list: Each item in the list is called a node and contains two fields

- Information field - The information field holds the actual elements in the list
- Link field- The Link field contains the address of the next node in the list

To create a linked list of integers the node can be defined as follows using a self referential structure.

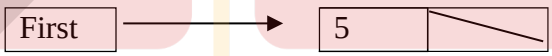
```
Sruct Node
{
int info;
struct Node * link;
};
Typedef struct Node NODE;
```

After the node is created we have to **create a new empty list as follows**

node * first=NULL;

- **The pointer first stores the address of the first node in the list.** With this information we will be able to access the location of all the other nodes in the list.
- To obtain a node we use the statement
- First=(node*) malloc(sizeof(node)
- To place the information 5 ,we can use the statement Firs->info= 5;
- As there are no other nodes in the list the link part can be made NULL as follow
- First->link=NULL

The node created is shown is represented as follows



Memory allocation and garbage collection

- The maintenance of linked lists in memory assumes the possibility of inserting new nodes into the lists and hence requires some mechanism which provides unused memory space for the new nodes. Similarly a mechanism is required which makes the deleted node available for future use.
- Together with the linked list in memory, a special list is maintained which consist of unused memory cells.
- This list which has its own pointer is called the list of available space or the free storage list or the free pool. Such a list is also called AVAIL

Instead of using the malloc function the following getnode() function can be used to get a new node

```
NODE * getNode(void)
{
    /* provide a node for use */NODE
* new;
if (avail)
            {
                new = avail;
                avail = avail→link;
                return new;
            }
        else
{
                new=( NODE *)malloc(sizeof(NODE));
                return new;
}
}
```

Instead of the free function the following retnode function can be used

```
void retNode(NODE *temp)
{/* return a node to the available list */
    temp→link = avail;
    avail = avail;
 }
```

Garbage collection

- Suppose some memory space becomes reusable because a node is deleted from a list or an entire list is deleted from a program, we can make this space to be available for future use. One way is to immediately reinsert the space into the free storage list.
- This is done when a list is implemented by linear arrays. But this method may be too time consuming for the operating system of the computer. So an alternate method is devised.
- The operating system of a computer may periodically collect all the deleted space on to the free storage list this technique si called **garbage collection**

Garbage collection takes place into two steps

- Runs through all the list an marks those cells which are currently in use
- Then the computer runs through the memory collecting all unmarked space on to the free storage list

Garbage collection takes place when

- The space is minimal or
- No space is left or
- when the CPU is idle and has time to do collection.

Garbage collection is invisible to the user.

- **Overflow:** When new data has to be inserted in to a data structure and if there is no available space in the free storage list , then we call this condition overflow
- **Underflow:** Refers to a condition when one tries to delete data from a data structure that is empty

Operations on Linked List

Insertion into a linked list

It is assumed that the following declaration is made globally

```
Sruct Node
{
int info;
struct Node * link;
};
typedef struct Node  NODE;
```

Function to insert a node in the front of a linked list

```
Void insert_front( int item)
{
NODE * temp;
temp=( NODE *)malloc(sizeof(NODE);
  if(temp==NULL)
  {
    printf("No space available\n");
exit(0);
     }
else
{
      temp->info=item;
     temp->link=first;
     first=temp;
}
}
```

Function to insert a node in the end of a linked list

```
void  insert_rear(int item)
{
  NODE *temp, *cur;
 temp=(NODE *)malloc(sizeof(NODE));
  if(temp==NULL)
  {
    printf("No space available\n");
exit(0);
   }
  temp->info=item;
```

```
        temp->link=NULL;
        if (first==NULL)
            first=temp;
       else
{
cur=first
while(cur->link!=NULL)cur=cur-
>link;
cur->link=temp
}
       }
```

Inserting an element after a node with location loc

```
       Void insert(int item, NODE* loc)
       {
       NODE * temp,*next;
       temp=( NODE *)malloc(sizeof(NODE);
         if(temp==NULL)
         {
           printf("No space available\n");
             }
       else
{
 temp->info=item;
next=loc->link; loc-
>link=temp; Temp-
>link=next;
       }
```

Deletion from a linked list

Function to delete a NODE in the front of a linked list

```
 void  delete-front()
 {
 NODE * cur;
 If (first==NULL)
 Printf("List empty");
 else
{
Cur=first first=first-
>linkfree(cur);

}
  }
```

Function to delete a NODE in the end of a linked list

```
Void delete-end()
{
NODE * cur,*prev;
If (first==NULL)
Printf("List empty");
else
        if (first->link==NULL)
         {
        Cur=first
        First=NULL
        Free(cur);
        }
      else
        {
         Prev=NULL;
        Cur=first;
        While(cur->link!=NULL)
                {
                Prev=cur;
                Cur=cur-.link;
                }
                Prev->link=NULL
                Free(cur);
}
 }
```

Delete the nodes from a linked list pointed by first whose information part is specified is item

```
NODE * delete_item(NODE *first, int item)
{
   NODE *prev, *cur;
 If (first==NULL)
{
printf("list is empty");
return(first);
}
if(first->info==item)
{
cur=first
first=first->link
free(cur);
return(first);
}
prev=NULL;
```

```
 cur=first;
 while (cur!=NULL)
 {
 If  (cur->info==item)
{
prev->link=cur->link;
free(cur); return(first);
}
 else
          {
          prev=cur;
          cur=cur->link;
          }
 }
 Printf("node with item not found")
 return(first);
  }
```

Delete the NODE present at location loc, the NODE that precedes is present at location locp. If there is only one NODE then locp=NULL

```
 Void delete(NODE *loc, NODE*locp)
 { NODE * next;
 If(locp==NULL)
{
            Free(Loc)
          }
 Else
          {
Next=loc->link; Locp-
>link=next;Free(Loc)
 }
 }
```

3.4.3 Traversal of Linked List

Function to display the contents of the list

```
 Void  display(Node * first)
 {
   NODE *cur;

   if (first==NULL)
printf("list is empty);
 else
```

```
{
cur=first
while(cur!=NULL)
{
 Printf("%d \t", cur->info);
cur=cur->link;
}
 }
```

Function to find the length of the the list

```
 int Length(Node * first)
 {
   NODE *cur;
 int count=0

   if (first==NULL)
{
printf("list is empty);
return(0)
}

 cur=first
 while(cur!=NULL)
{
 count++ cur=cur-
>link;
}
 return(count)
 }
```

SEARCHING A LINKED LIST

Search an item from an unsorted list

```
Void search(int item,struct NODE* first)
{
Struct NODE *cur;If
(first==NULL)
                printf("list empty");
else
                {
                cur=first;
                while(cur!=NULL)
                {
                        If(item==cur->info)
                          {
                            printf("search successful")
```

```
                    retrun;
                    }
                cur =cur->link;
            }
            }
            Printf("search unsuccessfull");
}
```

Search an item from a sorted list

```
Void search(int item,struct NODE* first)
{
Struct NODE *cur;If
(first==NULL)
            printf("list empty");
        else
            {
            cur=first;
            while(cur!=NULL&& item>=cur->nfo)
            {
                If(item==cur->info)
                  {
                   printf("search successful")
                   retrun;
                   }
                cur =cur->link;
            }
            }
            Printf("search unsuccessfull");
}
```

Additional Operations on a linked list

Function to concatenate two linked list pointed by list1 and list2

```
NODE* concat(Struct NODE * list1,stuct NODE *list2)
{
struct NODE* temp;
if(list1==NULL)
        return(list2)
if(list2==NULL)
        return(list1);
temp=list1
         while(temp->link!=NULL)
            temp=temp->link;
temp->link=list2
return(list1)
}
```

Function to reverse a linked list

```
NODE* reverse(struct NODE *first)
{
NODE *prev,*cur,*next;
cur=first;
prev=NULL;
while(cur!=NULL)
{
next= cur->linkcur-
>link=prev;prev= cur;
cur=next;
}
return(prev);
}
```

Implementation of stack using linked list

In a stack elements are inserted and deleted at only one end. The order of insertion and deletion follows LIFO order. To implement stack the push operation can be implemented by using insertion in front. Pop operation can be implemented by deletion from front so that insertion and deletion is happening at the same end.

Implementation of queue using linked list

To implement a queue using a linked list we use two pointers the front and the rear. The front pointer will have the address of the first NODE and the rear pointer will have the address of the last NODE. A new NODE is always attached to the rear of the list and the NODE at the front will be deleted first.

```
struct node
{
        int info;
        struct node *link;
};
      typedef struct node NODE;
        NODE * front,*rear;
        front=NULL;
        rear=NULL;
```

Boundary condition when front=NULL queue empty

The function inserts an element at the rear of the queue

```
void  insert_rear(int item,NODE * rear)
{
         NODE temp;
        temp=(NODE*)malloc(sizeof(NODE));
        if(temp==NULL)
         {
```

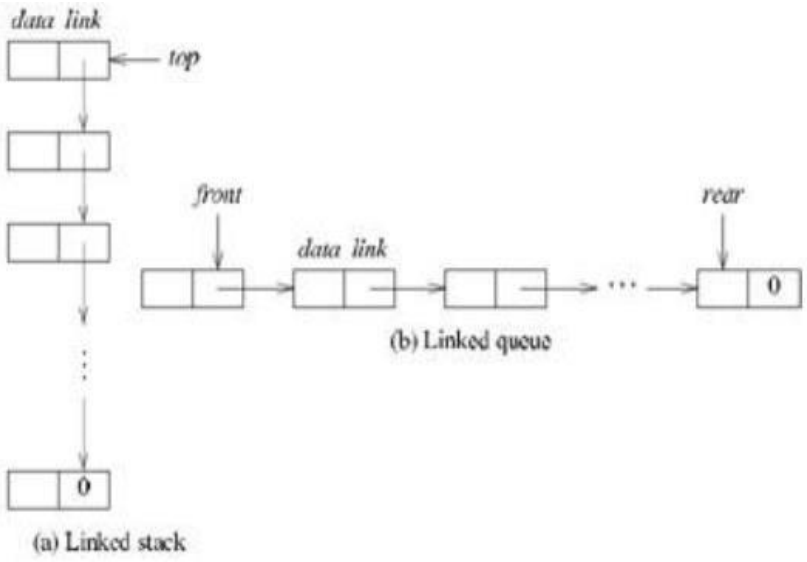
```
            printf("queue overflow\n");
            return(first)
}

            temp->info=item;
            temp->link=NULL;
            if (front==NULL)
{
            rear=temp;
front=temp;
}
            else
            {
            rear->link=temp;
            rear=temp;
            }
}
```

Function deletes the NODE in the front and returns the item

```
int  del_front(NODE * front)
{
  NODE cur;int
itemdel;
            if(front==NULL)
            {
              printf("Queue underflow\n");
              return front;
            }
               cur=front;
              itemdel=cur->info;
              front=front->link;
              free(cur);
            return(itemdel);
}
```

Linked stacks and queues

- We represented stacks and queues sequentially. However, when several stacks and queues coexisted, there was no efficient way to represent them sequentially. Fig below shows linked stack and a linked queue.
- Notice that the direction of links for both the stack and the queue facilitate easy insertion and deletion of NODEs. We can easily add or delete a NODE from the top of the stack. we can easily add a NODE to the rear of the queue and delete a NODE at the front.



If we wish to represent n ≤ MAX_STACKS stacks simultaneously, we begin with the declarations:

```
#define MAX_STACKS 10 /* maximum number of stacks */struct
Stack
{
int data;
struct Stack * link;
};
 typedef struct Stack stack stack *

top[MAX_STACKS]
```

We assume that the initial condition for the stacks is: top[i] = NULL, 0≤i < MAX_STACKS and the boundary condition is: top[i] = NULL if the ith stack is empty

Function push creates a new NODE, temp, and inserts the NODE in front of the ith stack.

```
void push(int i, int item)
{/* add item to the ith stack */ stack *
temp; temp=(stack*)malloc(sizeof(stack))
temp→data = item;
temp→link = top[i];top[i] =
temp;
}
```

Function pop returns the top element and changes top to point to the address contained in its link field.

```
int pop(int i)
{/* remove top element from the ith stack */int
itemdel;
Stack * temp;
if (top[i]==NULL) return
stackEmpty();

temp  =  top[i]; itemdel =
temp→data;top[i] =
top[i]→link; free(temp);
return item;
}
```

To represent m ≤ MAX_QUEUES queues simultaneously, we begin with the declarations:

```
#define MAX_QUEUES 10 /* maximum number of queues */
 struct queue {int data;
struct queue * link;
};
typedef struct queue Queue

Queue *front[MAX_QUEUES],* rear[MAX_QUEUES];
```

We assume that the initial condition for the queues is: front[i] = NULL, ,rear[i]=NULL0 ≤ i < MAX_QUEUES
and the boundary condition is: front[i] = NULL iff the ith queue is empty

Function addq adds the item to the ith queue

```
void addq(i, item)
{/* add item to the rear of queue i */Queue *
temp;
temp= (Queue*)malloc(sizeof(Queue));
if(temp==NULL)
{
printf("Queue Overflow");return;
}
temp→data = item;
temp→link = NULL;if
(front[i]==NULL)
```

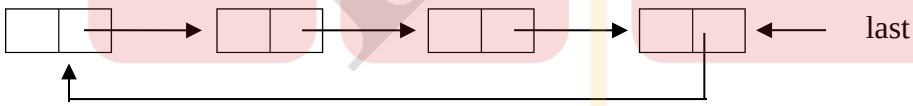
```
{
                front[i] = temp;
                rear[i] = temp;
}
else
{
                rear[i]→link = temp;
                rear[i]=temp;

}
}
```

Function deleteq deletes the item in the front of the ith queue

```
int  deleteq(int i)
{/* delete an element from queue i */Queue *
temp;
int itemdel
if (front[i]==NULL) return
queueEmpty();

temp =  front[i]; itemdel =
temp→data; front[i]=
front[i]→link;free(temp);
return itemdel;
}
```

Circular Linked List

A singly linked list in which the last NODE has a null link is called a chain. If the link field of the last NODE points to the first NODE in the list, then such a linked list is called a circular list.



By keeping a pointer at the last instead of the front we can now insert easily at the front and end of the list

Function inserts an element to the front of the list

```
void insertFront(NODE *last, NODE * New)
{/* insert an item at the front of the circular list whose last NODE is last */
```

```
if (last==NULL)
            {
/* list is empty, change last to point to new entry */last =
new;
last→link = last;
}
else
{
/* list is not empty, add new entry at front */
new→link = last→link;
last→link = new;
}
}
```

Function to insert an element in the end

```
void insertend(NODE *last, NODE * New)
{/* insert an item at the front of the circular list whose last NODE is last */

if (last==NULL)
            {
/* list is empty, change last to point to new entry */last =
new;
last→link = last;
}
else
{
/* list is not empty, add new entry at front */
New→link = last→link;
last→link = New;
last=New;
}
}
```

Function to find the length of a circular linked list

```
int length(NODE* last)
{/* find the length of the circular list last */NODE*
temp;
int count = 0;
If(last==NULL)
return(0);
temp = last->link /* temp is now pointing at the first NODE*/count++;
while(temp!=last)
{
```

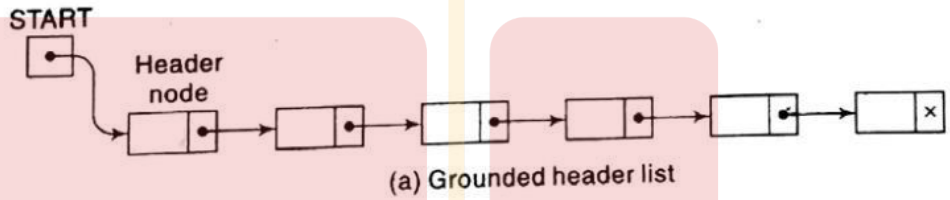
```
count++;
temp = temp→link;
}

return count;
}
```

Header Linked List

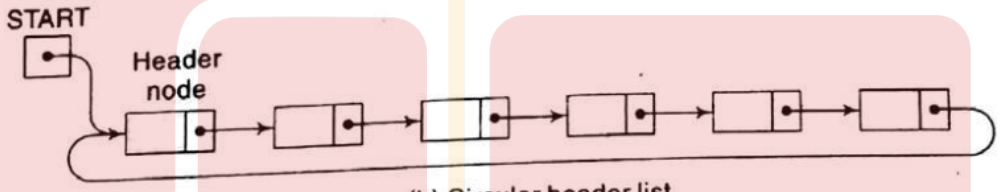
A header linked list is a linked list which always contains a special NODE, called the header NODE, at the beginning of the list. There are two types of header list.

1. **A grounded header list:** Last NODE contains the null pointer.
2. **Circular header list:** Last NODE points back to the header NODE.

Note: Unless stated it is assumed that the linked list is circular header list.



(a) Grounded header list



(b) Circular header list

Polynomials Polynomial

Representation

We should be able to represent any number of different polynomials as long as memory is available. In general, A polynomial is represented as :

$$A(x)= a_{m-1}x^{m-1} + ... a_0x^0$$

where the a_i are nonzero coefficients and the e^i are nonnegative integer exponents such that e m-1 > em-2 >.....> e1 > e0 ≥ 0.

We represent each term as a NODE containing coefficient and exponent fields, as well as a pointer to the next term. Assuming that the coefficients are integers, the type declarations are:

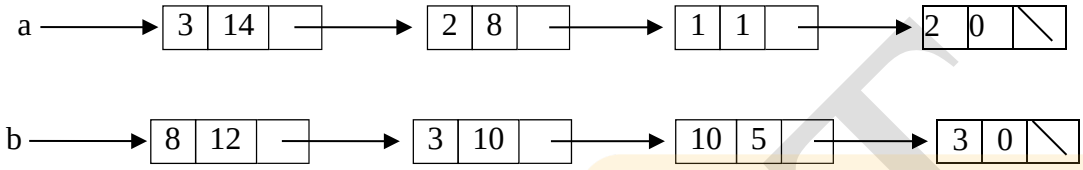
```
struct polyNode {int coef;
```

```
int expon;
struct polyNode * link;
};
        Typedef struct polyNode POLY;
```

A polyNODE can be drawn as

coef	expon	link

Consider the polynomials a = $3x^{14} + 2x^8 + 1x+2$ and b = $8x^{12}- 3x^{10} + 10x^5 +3$ It can be represented as follows



Adding Polynomials

To add two polynomials, we examine their terms starting at the NODEs pointed to by a and b.

1. If a→expon = b→expon, we add the two coefficients a→coef + b→coef and create a new term for the result c. a = a→link; b = b→link;
2. If a→expon < b→expon, then we create a duplicate term of b, attach this term to the result,called c, and b = b→link;
3. If a→expon > b→expon, then we create a duplicate term of a, attach this term to the result, called c, and a = a→link;

Adding two polynomials represented as singly Linked List

```
POLY *Pointer padd(POLY * a, POLY * b) /* return a polynomial which is the sum of a andb */
{
POLY  * c,*tempa, *tempb,*lastc;int sum;
c= (POLY*)malloc(sizeof(POLY))c-
>link=NULL
tempa=a
tempb=blastc=c;
while (tempa!=NULL && tempb!=NULL)
{
                switch (COMPARE(tempa→expon, tempb→expon))
                {
                case -1: lastc=attach(tempb→coef, tempb→expon,lastc);
                        tempb = tempb→link;
                        break;
```

```
                case 0: sum =tempa→coef + tempb→coef;
                        if (sum)
                          {
                           lastc=attach(sum, tempa→expon, lastc);
                           tempa = tempa→link; tempb = tempb→link;
                           }
                         break;
                case 1: lastc=attach(tempa→coef,tempa→expon,lastc);
                        tema = tempa→link;
                }
}

/* copy rest of list a and then list b */
While(tempa!=NULL)
{
                lastc=attach (tempa→coef,tempa→expon,lastc);
                tempa=tempa->link;
}
While(tempb!=NULL)
{
                lastc=attach (tempb→coef,tempb→expon,lastc);
                tempb=tempb->link;
}

return(c);
}
```

Attach a NODE to the end of a list

```
POLY*  attach(float c, int e, POL * rear)
{
                temp = (POLY*)malloc(sizeof(POLY));
                temp→coef = c;
                temp→expon = e;
                rear->link=temp;
                rear=temp;
                return(rear);
}
```

Adding two polynomials represented as circular lists with header NODEs

```
POLY *Pointer padd(POLY * a, POLY * b) /* return a polynomial which is the sum of a andb */
{
POLY  * c,*tempa, *tempb,*lastc;int sum;
c=      (POLY*)malloc(sizeof(POLY))      c-
>link=c
```

```
tempa=a->link;
tempb=b->link;lastc=c;
while (tempa!=a && tempb!=b)
{
                switch (COMPARE(tempa→expon, tempb→expon))
                {
                case -1: lastc=attach(tempb→coef, tempb→expon,lastc);
                        tempb = tempb→link;
                        break;

                case 0: sum =tempa→coef + tempb→coef;
                      if (sum)
                          {
                            lastc=attach(sum, tempa→expon, lastc);
                            tempa = tempa→link; tempb = tempb→link;
                            }
                         break;
                case 1:  lastc=attach(a→coef,a→expon,lastc);
                        tempa = tempa→link;
                }
}

/* copy rest of list a and then list b */
While(tempa!=a)
{
                lastc=attach (a→coef,a→expon,lastc);
                tempa=tempa->link;
}
While(tempb!=b)
{
                lastc=attach (b→coef,b→expon,lastc);
                tempb=tempb->link;
}

lastc->link=c
return(c);
}
```

Attach a NODE to the end of a list

```
POLY*  attach(float c, int e, POL * rear)
{
                temp = (POLY*)malloc(sizeof(POLY));
```

```
        temp→coef = c;
        temp→expon = e;
        temp->link =rear->link;
        rear->link=temp;
        rear=temp;
        return(rear);
}
```

Analysis of Polynomial function

The number of non zero terms in A and in B are the most important factors in the time complexity

Therefore, let m and n be the number of nonzero terms in A and B, respectively.

A(x)= am-1xm-1 +..... a0x0

B(x)= bm-1xn-1 +..... b0x0

If m > 0 and n > 0, the while loop is entered. Each iteration of the loop requires O(1) time.At each iteration, either a or b moves to the next term or both move to the next term. Since the iteration terminates when either a or b reaches the end of the list, therefore, the number of iterations is bounded by m + n - 1.

The time for the remaining two loops is bounded by O(n + m). The first loop can iterate m times and the second can iterate n times. So, the asymptotic computing time of this algorithm is **O(n +m).**

Erasing a circular list

If we wish to compute more polynomials, it would be useful to reclaim the NODEs that are being used. This function erases a circular list pointed by ptr and adds the NODE to the availability list.

```
void cerase(POLY *ptr)
{
POLY* temp;
if (ptr )
{
temp = ptr>link ptr->link=avail avail = temp;
*ptr = NULL;
}
   }
```

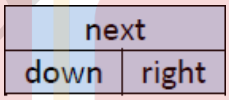
MODULE – 3

LINKED LISTS : Sparse Matrices, Doubly Linked List.

TREES: Introduction, Binary Trees, Binary Tree Traversals, Threaded Binary Trees.

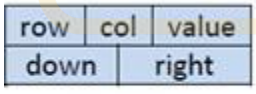
Sparse matrix representation

- We can save space and computing time by retaining only the nonzero terms of sparse matrices.
- In the sequential scheme each nonzero term was represented by a NODE with three fields: *row*, *column*, and *value*. These NODEs were organized sequentially. However, we found that when we performed matrix operations such as addition, subtraction, or multiplication, the number of nonzero terms varied.
- In this section, we study a linked list representation for sparse matrices. Usually, linked lists allow us to efficiently represent structures that vary in size, a benefit that also applies to sparse matrices.
- In our data representation, we represent each column of a sparse matrix as a circularly linked list with a header NODE. We use a similar representation for each row of a sparse matrix.
- Each NODE has a tag field, which we use to distinguish between header NODEs and entry NODEs.
- Each header NODE has three additional fields: *down*, *right*, and *next*.
  - *down* field links into a column list and the
  - *right* field links into a row list.
  - The *next* field links the header NODEs together.
  - The header NODE for row *i* is also the header NODE for column *i*, and the total number of header NODEs is max {number of rows, number of columns}.



Header Node

- Each entry NODE consist of the following structure



Entry Node

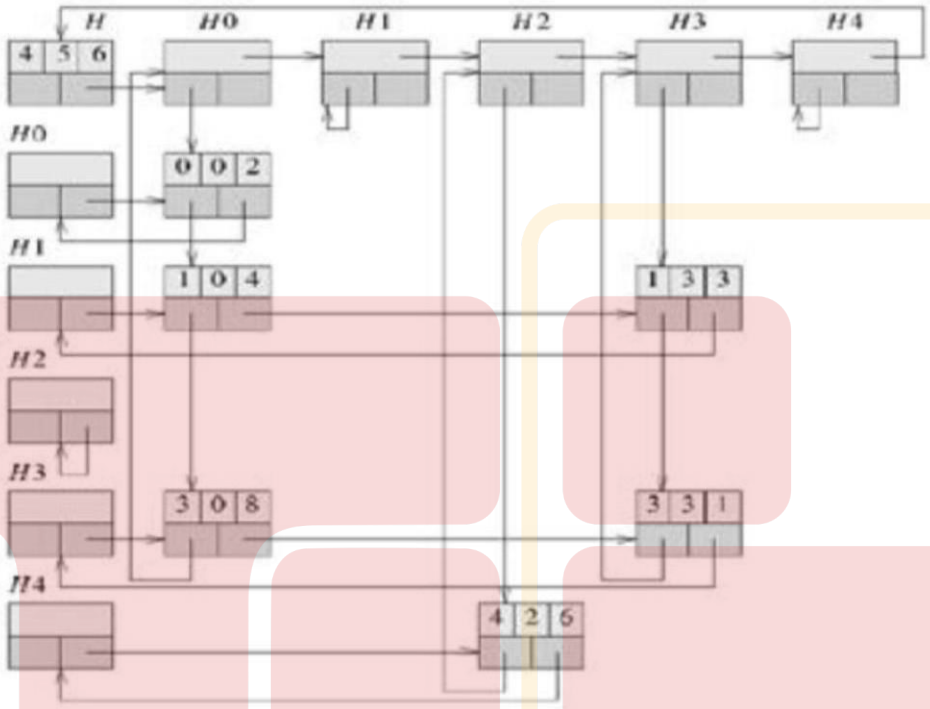
- Each header NODE is in three lists: a list of rows, a list of columns, and a list of header NODEs. The list of header NODEs also has a header NODE that has the same structure as an entry NODE.
- The row and col value of the header NODE consist of the dimension of the matrix

Example:

Consider the sparse matrix shown below.

$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 4 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 1 \\ 0 & 0 & 6 & 0 \end{bmatrix}$$

The Linked The Linked representation is shown below



Since there are two different types of NODEs a union is used to create the appropriate data structure. The necessary C declarations are as follows:

```
#define MAX_SIZE 50 /*size of largest matrix*/

typedef enum {head, entry} tagfield;

typedef struct entryNODE {
int row;
int col;
int value;
};

typedef struct matrixNODE {
matrixPointer *down;
matrixPointer *right;
tagfield tag;
union {
```

```
matrixPointer *next;
entryNODE entry;
} u;
};

typedef struct matrixNODE matrixPointer;
matrixPointer *hdNODE[MAX_SIZE];
```

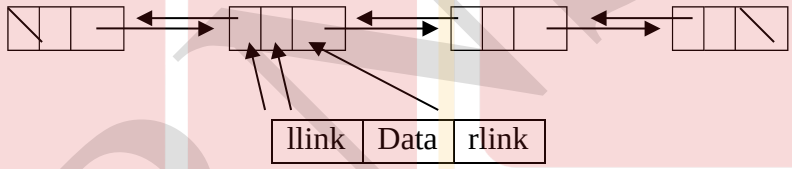
Doubley Linked List

Disadvantage of singly linked list

- If we are pointing to a specific NODE, say *p*, then we can move only in the direction of the links. The only way to find the NODE that precedes *p* is to start at the beginning of the list.
- If we wish to delete an arbitrary NODE from a singly linked list. Easy deletion of an arbitrary NODE requires knowing the preceding NODE.
- Can traverse only in one direction.
- Difficult to delete arbitrary NODEs.

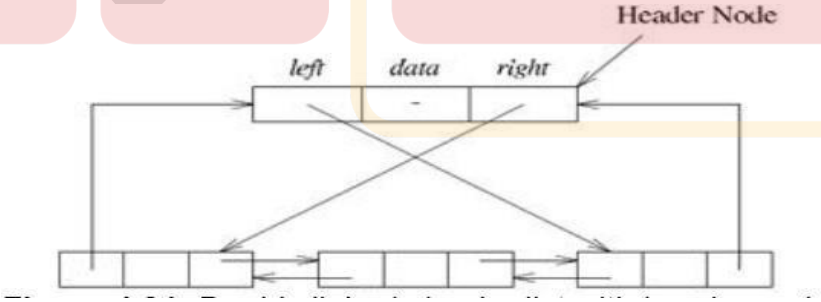
Doubly Linked List representation

- Each NODE now has two link fields, one linking in the forward direction and the other linking in the backward direction.
- A NODE in a doubly linked list has at least three fields, a left link field (*llink*), a data field (*data*), and a right link field (*rlink*). It can be represented as follows



A Doubly Linked List without Header NODE

A doubly linked list may or may not be circular. The data field of the header NODE usually contains no information.



A Circular Doubly Linked List with a Header NODE

If head is a pointer to the header NODE then head->llink= head-rlink= head

An empty doubly circular list with a header NODE

The necessary declarations are:

```
 struct NODE
{
strunct NODE * llink;
int data;
struct NODE * rlink;
};

typedef struct NODE DNODE;
```

The function dinsert() inserts a newNODE into a doubly linked list after a NODE pointed by ptr

```
dinsert(DNODE *ptr, DNODE *newNODE)
{
   DNODE * next
next=ptr->rlink;
ptr->rlink=newNODE;
newNODE->llink=ptr;
newNODE->rlink=next;
next->llink=newNODE;
}
```

The function **ddelete()** deletes a NODE from a doubly linked list pointed by head

```
Void ddelete(DNODE *NODE,DNODE *head)
{
if (NODE==head)
printf("deletion of header NODE not permitted");
else
{
        prev=NODE->llink;
        next=NODE->rlink;
        prev->rlink=next;
        next->llink=prev
        free(cur)
```

```
}
}
```

Advantages of doubly linked list

- Can delete arbitrary NODE
- Can traverse in both directions

Disadvantage of doubly linked list

Space Efficiency: We have the overhead of storing two pointers for each element.

Polynomials Polynomial

Representation

We should be able to represent any number of different polynomials as long as memory is available. In general, A polynomial is represented as :

$$A(x)= a_{m-1}x^{m-1} + ... a_0x^0$$

where the a_i are nonzero coefficients and the e^i are nonnegative integer exponents such that
e m-1 > em-2 >.....> e1 > e0 ≥ 0.

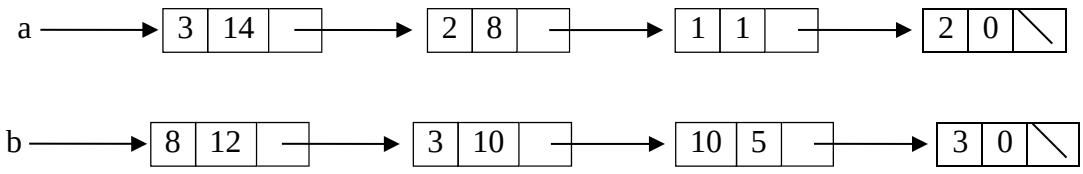
We represent each term as a NODE containing coefficient and exponent fields, as well as a pointer to the next term. Assuming that the coefficients are integers, the type declarations are:

```
struct polyNode {
int coef;
int expon;
struct polyNode * link;
};
  Typedef struct polyNode POLY;
```

A polyNODE can be drawn as

coef	expon	link

Consider the polynomials a = $3x^{14} + 2x^8 + 1x+2$ and b = $8x^{12}- 3x^{10} + 10x^5 +3$ It can be represented as follows



Adding Polynomials

To add two polynomials, we examine their terms starting at the NODEs pointed to by a and b.

1. If a→expon = b→expon, we add the two coefficients a→coef + b→coef and create a new term for the result c. a = a→link; b = b→link;
2. If a→expon < b→expon, then we create a duplicate term of b, attach this term to the result,called c, and b = b→link;
3. If a→expon > b→expon, then we create a duplicate term of a, attach this term to the result, called c, and a = a→link;

Adding two polynomials represented as singly Linked List

```
POLY *Pointer padd(POLY * a, POLY * b) /* return a polynomial which is the sum of a and
b */
{
POLY  * c,*tempa, *tempb,*lastc;
int sum;
c= (POLY*)malloc(sizeof(POLY))
c->link=NULL
tempa=a
tempb=b
lastc=c;
while (tempa!=NULL && tempb!=NULL)
{
        switch (COMPARE(tempa→expon, tempb→expon))
        {
        case -1: lastc=attach(tempb→coef, tempb→expon,lastc);
                 tempb = tempb→link;
                 break;
        case 0: sum =tempa→coef + tempb→coef;
                 if (sum)
                   {
                    lastc=attach(sum, tempa→expon, lastc);
                    tempa = tempa→link; tempb = tempb→link;
                    }
                  break;
        case 1: lastc=attach(tempa→coef,tempa→expon,lastc);
                 tema = tempa→link;
        }
}

/* copy rest of list a and then list b */
While(tempa!=NULL)
{
        lastc=attach (tempa→coef,tempa→expon,lastc);
        tempa=tempa->link;
}
```

```
While(tempb!=NULL)
{
        lastc=attach (tempb→coef,tempb→expon,lastc);
        tempb=tempb->link;
}

return(c);
}
```

Attach a NODE to the end of a list

```
POLY*  attach(float c, int e, POL * rear)
{
        temp = (POLY*)malloc(sizeof(POLY));
        temp→coef = c;
        temp→expon = e;
        rear->link=temp;
        rear=temp;
        return(rear);
}
```

Adding two polynomials represented as circular lists with header NODEs

```
POLY *Pointer padd(POLY * a, POLY * b) /* return a polynomial which is the sum of a and
b */
{
POLY  * c,*tempa, *tempb,*lastc;
int sum;
c= (POLY*)malloc(sizeof(POLY))
c->link=c
tempa=a->link;
tempb=b->link;
lastc=c;
while (tempa!=a && tempb!=b)
{
        switch (COMPARE(tempa→expon, tempb→expon))
        {
        case -1: lastc=attach(tempb→coef, tempb→expon,lastc);
                 tempb = tempb→link;
                 break;

        case 0: sum =tempa→coef + tempb→coef;
              if (sum)
                  {
                   lastc=attach(sum, tempa→expon, lastc);
                   tempa = tempa→link; tempb = tempb→link;
                   }
```

```
                    break;
            case 1:  lastc=attach(a→coef,a→expon,lastc);
                    tempa = tempa→link;
            }
    }

    /* copy rest of list a and then list b */
    While(tempa!=a)
    {
            lastc=attach (a→coef,a→expon,lastc);
            tempa=tempa->link;
    }
    While(tempb!=b)
    {
            lastc=attach (b→coef,b→expon,lastc);
            tempb=tempb->link;
    }

    lastc->link=c
    return(c);
    }
```

Attach a NODE to the end of a list

```
POLY*  attach(float c, int e, POL * rear)
    {
            temp = (POLY*)malloc(sizeof(POLY));
            temp→coef = c;
            temp→expon = e;
            temp->link =rear->link;
            rear->link=temp;
            rear=temp;
            return(rear);
    }
```

Analysis of Polynomial function

The number of non zero terms in A and in B are the most important factors in the time complexity

Therefore, let m and n be the number of nonzero terms in A and B, respectively.

A(x)= am-1xm-1 +..... a0x0

B(x)= bm-1xn-1 +..... b0x0

If m > 0 and n > 0, the while loop is entered. Each iteration of the loop requires O(1) time.At each iteration, either a or b moves to the next term or both move to the next term. Since the iteration terminates when either a or b reaches the end of the list, therefore, the number of iterations is bounded by m + n - 1.

The time for the remaining two loops is bounded by O(n + m). The first loop can iterate m times and the second can iterate n times. So, the asymptotic computing time of this algorithm is **O(n +m).**

Erasing a circular list

If we wish to compute more polynomials, it would be useful to reclaim the NODEs that are being used. This function erases a circular list pointed by ptr and adds the NODE to the availability list.

```
void cerase(POLY *ptr)
{
POLY* temp;
if (ptr )
        {
        temp = ptr>link
        ptr->link=avail
        avail = temp;
        *ptr = NULL;
        }
  }
```

TREES

Definition of Tree:
A **tree** is a finite set of one or more nodes such that

- There is a special node called the root.
- The remaining nodes are partitioned into $n \geq 0$ disjoint sets T1, ⋯, Tn, where each of these sets are called the subtrees of the root.

Example: Consider the tree given below

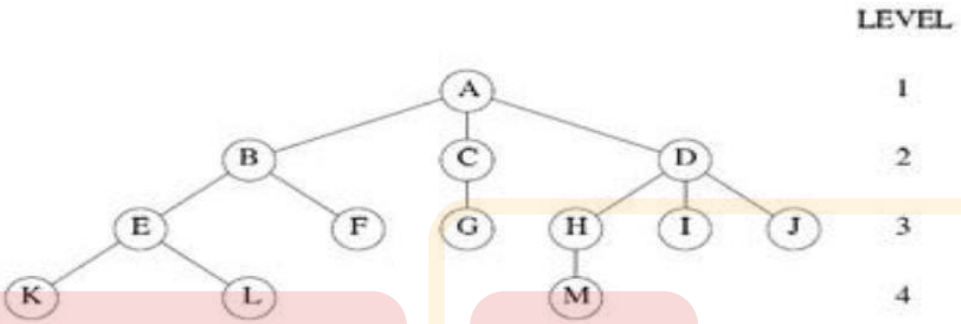


Figure 4.1 A sample Tree

The tree has 13 nodes and has one character as its information. A tree is always drawn with its root at the top. Here the node A is the root.

- **Degree of a node**: The number of subtrees of a node is called its degree.
  - Example : The degree of A = 3, C = 1, and of F = 0.
- **Degree** of **a Tree** :The degree of a tree is the maximum degree of the nodes in the tree. The tree shown in the example has degree 3.
- **Leaf or terminal nodes or external nodes:** Nodes that have degree zero is called the leaf nodes
  - Example : {K,L,F,G,M,I,J} is the set of leaf nodes
- **Non-terminal nodes/internal nodes :** Nodes that have at least a degree one or two. (nodes other than the leaf nodes)
  - Example : {B,C,D,E,F,H,A} is the set of Non-terminal nodes
- **Siblings :** Children of the same parent are said to be siblings
  - Example : {H,I,J} are siblings.
- **The ancestors:** all the nodes along the path from the root to that node.
  - Example : The ancestors of M are A, D, and H.
- **The level of a node is** .The root is considered be at level one[1]. If a node is at level l, then its children are at level l + 1.
- **The height or depth** is maximum level of any node in the tree. Thus, the depth of the tree in the example is 4.

Binary Trees
The Abstract Data type

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

Objects: a finite set of nodes either empty or consisting of a root node, left Binary_Tree, and right Binary_Tree.

Functions: for all bt,bt1,bt2 ∈ BinTree, item ∈ element

BinTree Create() ::=	creates an empty binary tree
Boolean IsEmpty(bt) ::=	if (bt == empty binary tree) return TRUE else return FALSE
BinTree MakeBT(bt1, item,bt2) ::=	return a binary tree whose left subtree is bt1, whose right subtree is bt2, and whose root node contains the data item.
BinTree Lchild(bt) ::=	if (IsEmpty(bt)) return error else return the left subtree of bt.
Element Data(bt) ::=	if (IsEmpty(bt)) return error else return the data in the root node ofbt.
BinTree Rchild(bt) ::=	if (IsEmpty(bt)) return error else return the right subtree of bt

Difference between binary tree and tree.

Binary tree	Tree
Empty tree exist	No tree exist with zero nodes
Order of the child is considered	Order of the child is not considered
Each node can be partitioned to only two disjoint subtrees	Each node can be partitioned to T1,T2…..Tn disjoint subtrees

Properties of Binary Tree

Lemma 1: [Maximum number of nodes]

.

1. The maximum number of nodes on level i of a binary tree is 2^{i-1}, $i \geq 1$

   Proof:
   The proof is by induction on i.

   Induction Base: The root is the only node on level i = 1. Hence, the maximum number of nodes
   on level i = 1 is $2^{i-1} = 2^0 = 1$.

   Induction Hypothesis: Let i be an arbitrary positive integer greater than 1. Assume that the maximum number of nodes on level i - 1 is 2^{i-2}.

   Induction Step:

The maximum number of nodes on level i - 1 is 2^{i-2} by the induction hypothesis

Since each node in a binary tree has a maximum degree of 2, the maximum number of nodes on level i is two times the maximum number of nodes on level i-1,
i.e $2^* 2^{i-2} = 2^{i-1}$
Hence Prooved

2. The maximum number of nodes in a binary tree of depth k is $2^k - 1$, $k \geq 1$.
The maximum number of nodes in a binary tree of depth k is

$$\sum_{i=1}^{k} (\text{maximum number of nodes on level } i) = \sum_{i=1}^{k} 2^{i-1} = 2^k - 1$$

Lemma 2: [Relation between number of leaf nodes and degree-2 nodes]: For any non empty binary tree, T, if n0 is the number of leaf nodes and n2 the number of nodes of degree 2, then n0 = n2 + 1.

Proof:
Let n1 be the number of nodes of degree one and n the total number of nodes. Since all nodes in T are at most of degree two, we have
n=n0+n1+n2--------- (1)

If we count the number of branches in a binary tree, we see that every node except the root has a branch leading into it.

If B is the number of branches, then n = B + 1--------- (2)

All branches stem out from a node of degree one or two.
Thus, B = n1 + 2n2. Hence, we obtain -- (-3)
Sum of the branches that stem out of a node(outdegree) is always equal to the sum of the branches that stem into a node(indegree). Therefore Substitutuing eq(3) in eq(2)
n=B+1
n= n1 + 2n2 +1.....(4)
Subtracting Eq. (4) from Eq. (1) and rearranging terms, we get
n0 = n2 + 1

Definition

Full Binary Tree: A full binary tree of depth k is a binary tree of depth k having $2^k - 1$ nodes, $k \geq 0$.

Example

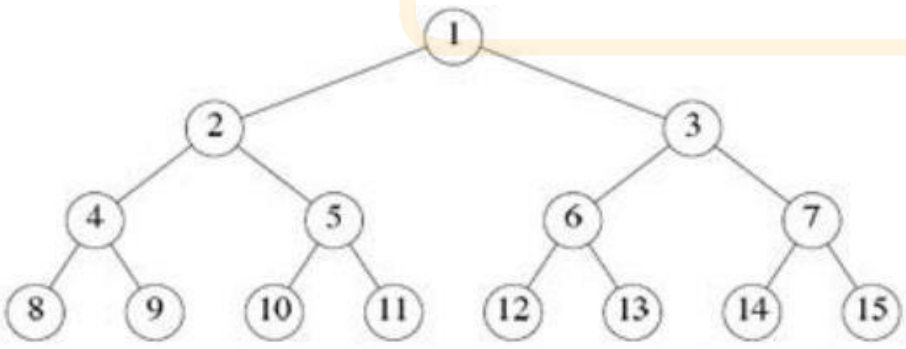
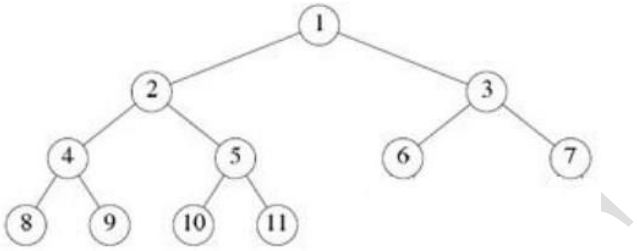


Figure 4.6 Full Binary tree of depth 4

The nodes are numbered in a full binary tree starting with the root on level 1, continuing with the nodes on level 2, and so on. Nodes on any level are numbered from left to right.

Complete binary tree : A binary tree is complete if the number of nodes in each level i except possibly the last level is 2^{i-1}. The number of nodes in the last level appears as left as possible.

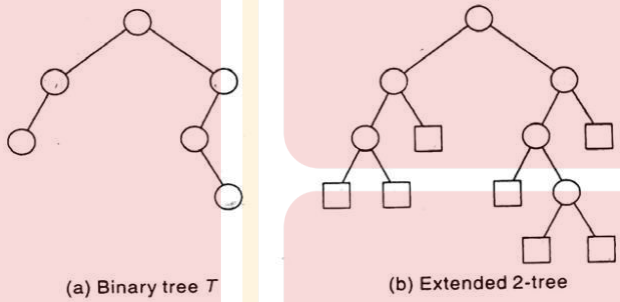
Example: A complete tree T11 with 11 nodes is shown below. This is not a full binary tree.



Complete Binary Tree

Extended Binary tree

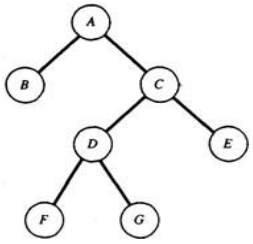
- A binary tree T is said to be a 2-tree or an extended binary tree if each node N has either 0 or 2 children.
- The nodes with 2 children are called internal nodes and the nodes with 0 children are called external nodes. Sometimes the nodes are distinguished in diagrams by using circles for internal nodes and squares for external nodes
- The term extended binary tree comes from the operation where tree T is converted into a 2-tree by replacing each empty subtree by new node and the new tree is a 2-tree . The nodes in the original tree T are internal nodes in the extended tree and the new nodes are the external nodes in the extended tree.



(a) Binary tree *T* (b) Extended 2-tree

Strictly Binary Tree is a tree where every non leaf node in a binary tree has non empty left and right subtrees.A strictly binary tree with n leaves always contain 2n-1 nodes

Example:

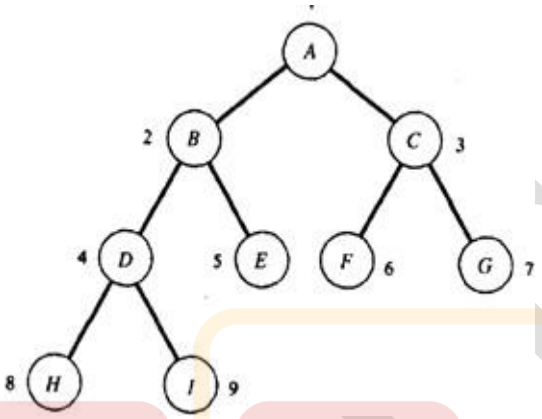


Example For a Strictly Binary Tree

Almost Complete Binary Tree: A binary tree of depth d is an almost complete binary tree if

a. A node n at level less than d-1 has two sons
b. For any node n in the tree with a right descendant at level d, n must have a left son and every left descendant of n is either a leaf at level d or has two sons

Example



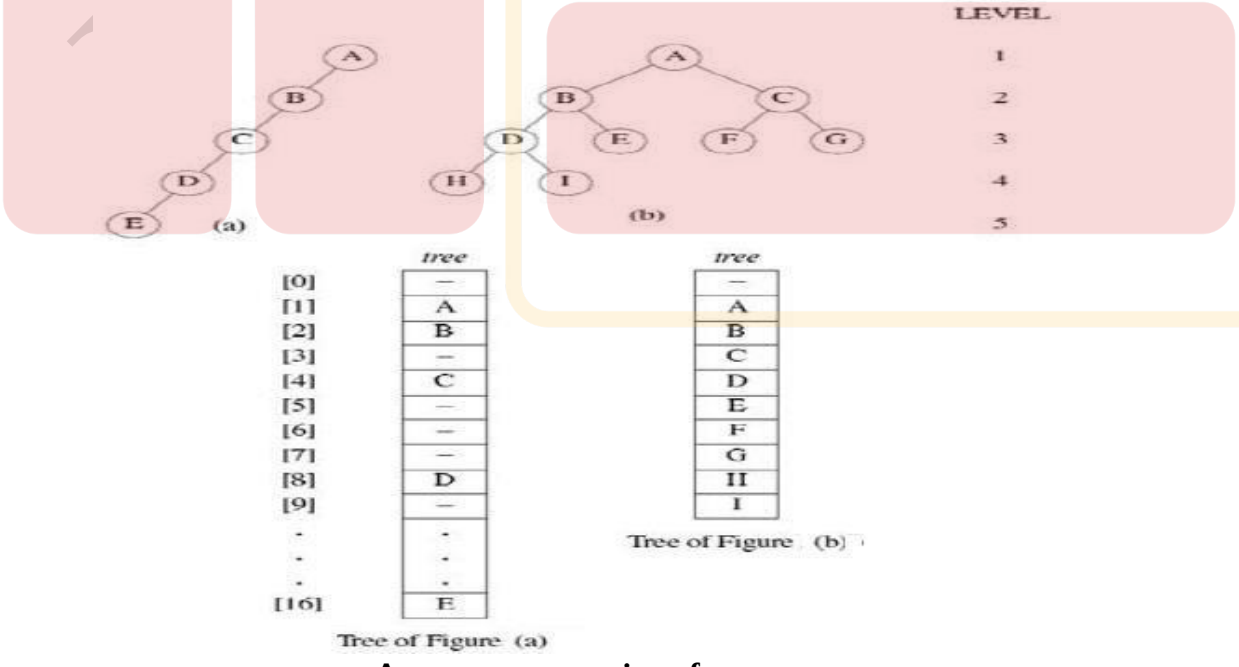
Almost Complete Binary Tree.

Binary Tree representations

Array Representation: If a complete binary tree with n nodes is represented sequentially, then for any node with index i, $1 \leq i \leq n$, we have

- parent (i) is at $[i / 2]$ if $i \neq 1$. If i = 1, i is at the root and has no parent.
- leftChild (i) is at 2i if $2i \leq n$. If $2i > n$, then i has no left child.
- rightChild (i) is at $2i + 1$ if $2i + 1 \leq n$. If $2i + 1 > n$, then i has no right child.

Example



Array representation of tree

This representation can be used for any binary tree. In most cases there will be a lot of unutilized spaces. For complete binary tree such as

Linked Representation

Disadvantage of array representation

- The array representation is good for complete binary trees but, it wastes a lot of space for many other binary trees.
- Insertion and deletion of nodes from the middle of a tree require the movement of potentially many nodes to reflect the change in level number of these nodes.
- These problems can be overcome easily through the use of a linked representation.

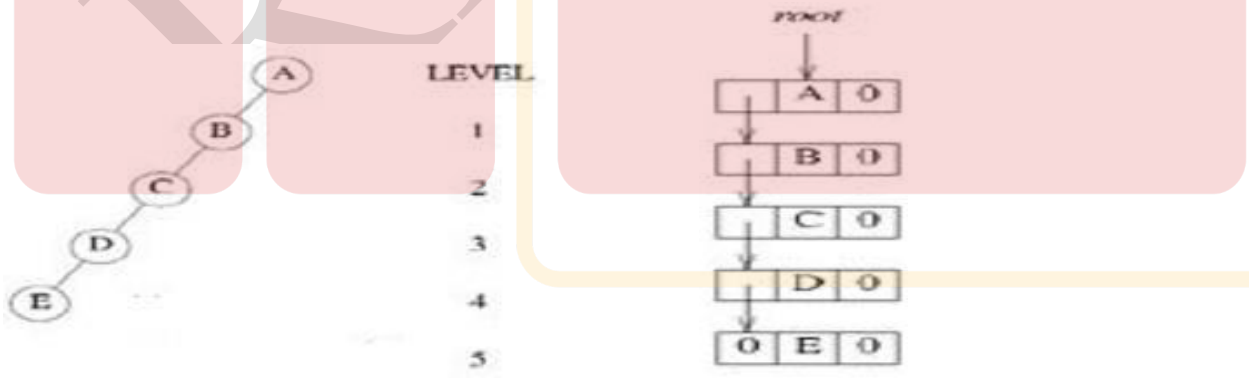
Each node has three fields, leftChild, data, and rightChild.
A node can be defined as:

```
struct node
{
        int data;
        struct node * leftChild;
        struct node * rightChild;
};

Typedef struct node TreeNode;
```

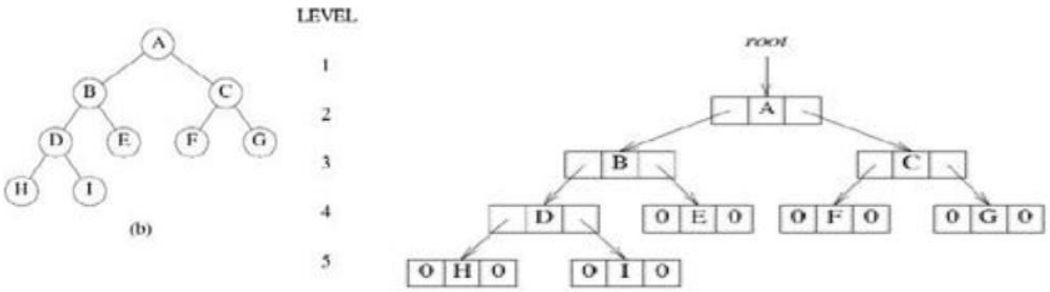
A node in a tree can be represented as follows

Node Representation

With this node structure it is difficult to determine the parent of a node, If it is necessary to be able to determine the parent of random nodes, then a fourth field, parent, may be included in the class TreeNode



Linked Representation of Skewed Tree



Linked Representation of Complete Tree

Binary Tree Traversals

- Traversing a tree is visiting each node in the tree exactly once.
- When a node is visited, some operation (such as outputting its *data* field) is performed on it.
- A full traversal produces a linear order for the nodes in a tree.
- When traversing a tree each node and its subtrees must be treated in the same fashion.
- Based on this there are three types of traversals inorder, preorder and postorder traversals.

Inorder Traversal

- Inorder traversal move down the tree toward the left until we can go no farther.
- Then "visit" the node,
- move one node to the right and continue.
- If we cannot move to the right, go back one more node and continue
- A precise way of describing this traversal is by using recursion as follows

```
void inorder(TreeNode * ptr)
{/* inorder tree traversal */
if (ptr)
      {
      inorder(ptr→leftChild);
      printf("%d",ptr→data);
      inorder(ptr→rightChild);
      }
}
```

Preorder Traversal

- visit a node
- traverse left, and continue.
- When you cannot continue, move right and begin again or move back until you can move right and resume."

```
void preorder(TreeNode *ptr)
{/* preorder tree traversal */
if (ptr)
       {
      printf("%d",     ptr→data);
      preorder(ptr→leftChild);
      preorder(ptr→rightChild);
```

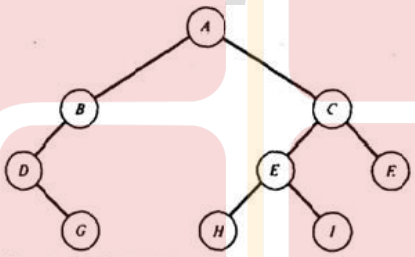
```
    }
}
```

Postorder Traversal

- traverse left, and continue.
- When you cannot continue, move right and traverse right as far as possible
- Visit the node

```
void postorder(TreeNode  *ptr)
{/* postorder tree traversal */
if (ptr)
    {
    postorder(ptr→leftChild);
    postorder(ptr→rightChild);
    printf("%d",ptr→data);
    }
}
```

Example:



Inorder Traversal: DGBAHEICF

Preorder Traversal: ABDGCEHIF

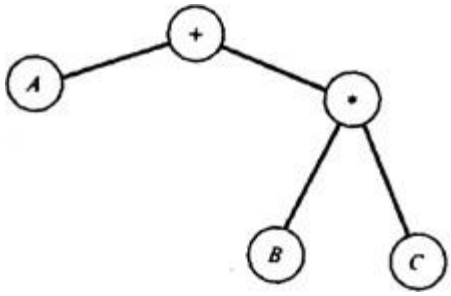
Postorder Traversal: GDBHIEFCA

Expression Tree: An expression containing operands and binary operators can be representedby a binary tree.

Representation and traversal of expression tree

A node representing an operator is a non leaf. A node representing an operand is a leaf. The root of the tree contains the operator that has to be applied to the results of evaluating the left subtree and the right subtree.

Example1: A+ B * C can be represented as follows



When the binary expression trees are traversed preorder we get the preorder expression. When we traverse the tree postorder we get the postorder expression . When we traverse it inorder we get the inorder expression.

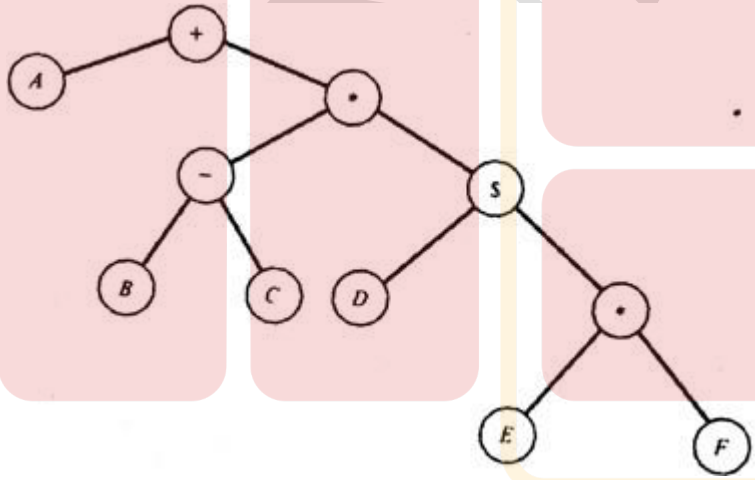
For Example : consider the traversals for the tree given above

Inorder traversal giving rise to infix expression A+B*C

Preorder traversal giving rise to prefix expression +A*BC

Postorder traversal giving rise to postfix expression ABC*+

Example2: A+(B-C)*D$(E*F) is represented as follows



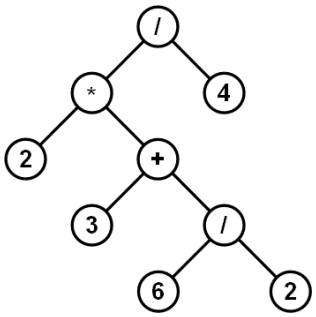
Inorder traversal giving rise to infix expression A+(B-C)*D$(E*F)

Preorder traversal giving rise to prefix expression : +A*-BC$D*EF

Postorder traversal giving rise to postfix expressionABC-DEF*$*+

Evaluation of Expression Tree

Example:



Result = 3

Additional Binary Tree operations

4.4.1. Copying Binary Trees

This function returns a pointer to an exact copy of the original tree.

```
TreeNode* Copy(TreeNode *original)
{
TreeNode * temp;
if (original!=NULL)
{
Temp=(TreeNode*)malloc(sizeof(TreeNode));
Temp->leftchild= copy(original->leftchild);
Temp->rightchild= copy(original->rightchild);
Temp->data=original ->data;
retrun temp;
}
return Null;
}
```

Testing Equality

- Equivalent Binary trees have the same structure and the same information in the corresponding nodes.
- Same structure means every branch in one tree corresponds to a branch in the second tree that is the branching of the trees is identical.
- This function returns true if the two trees are equivalent and false otherwise.

```
int equal(TreeNode *first,TreeNode *second)
{
If (first==NULL && second==NULL)
return true;

if(first !=NULL && second!=NULL && first->data==second->data &&
```

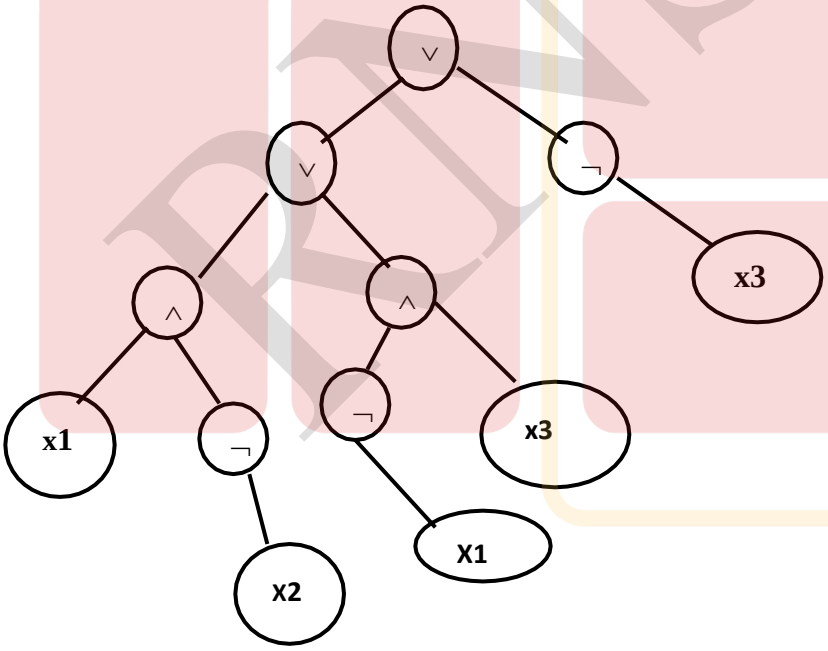
```
 equal(first-> leftchild ,second->leftchild) &&
equal(first->rightchild,second->rightchild))
return true;

return false;
}
```

The Satisfiability problem

- Conisder the set of formulas that we can construct by taking the variables x1,x2,……xn and operators ∧(AND) V ∨(OR) and ¬ (NOT)
- The variables can hold any one of the two possible values true or false.
- The set of expressions that can be formed using these variables and operators is defined by the following rules.
    1. A variable is an expression
    2. If x and y are expressions then ¬x, x∧y, x∨y are expressions
    3. Order of evaluation first ¬ then ∧ then ∨.
    4. Parenthesis can be used to alter the normal order of evaluations.

Defintion: The satisfiability problem for formulas of the propositional calculus asks if there is an assignment of values to the variables that causes the value of the expression to be true.

Example: Representation of the expression (x1 ∧ ¬x2) ∨ (¬x1 ∧ x3) ∨ ¬x3 as a binary tree



Propositional formula in a binary Tree

Inorder Traversal of the tree is x1 ∧ ¬ x2 ∨ ¬x1 ∧ x3 ∨ ¬x3 this is the infix form of the expression.

Note: The node containing ¬ has only a right branch since ¬ is a unary operator.

To determine satisfiability (x1,x2,x3) must take all possible combinations of true or false values and check the formula for each combination. For n variables ther are 2^n possible combinations of true=t and false=f.

Example: For n=3 the eight combinations are (t,t,t),(t,t,f),(t,f,t),(t,f,f),(f,t,t), (f,t,f), (f,f,t),(f,f,f).

The node structure for the satisfaibility problem is given as follows

leftchild	Data	value	rightchild

The node structure can be defined as follows

```
typedef enum {not,and, or, true, false } logical;
struct node
{
Struct node *leftchild;
Logical data;
Short int value;
Structnode * rightchild;
};
Typedef struct node  TNODE;
```

Algorithm to determine satisfiability

```
For ( all 2^n possible combinations)
{
        Generate the next combination;
        Replace the variables by their values;
        Evaluate the root by traversing it in postorder;
        If (root->value)
        {
        Printf(<combination>);
        Return;
         }
}
Printf(" No satisfiable combination");
```

Analysis: This algorithm will take $O(g.2^n)$ or exponential time, where g is the time to substitute values for x1,x2,..... xn and evaluate the expression.

Postorder Evaluation function: To evaluate an expression, the tree is traversed in postorder. When a node is visited the value of the expression represented by the left and right sub trees of a node are computed first. So the recursive postorder traversal algorithm is modified to obtain the function that evaluates the tree.

```
Void postorderEval(TNODE *node)
{
If (node)
{
```

```
        postorderEval(node->leftchild);
        postorderEval(node->rightchild);
        switch(node->data)
        {
                Case not: node->value= ! node->rightchild->value;
                        Break;
                Case and: node->value= node->rightchild->value && node>lefchild->value;
                        Break;
                Case or: node->value= node->rightchild->value || node>lefchild->value;
                        Break;
                 Case true: node->value= true; break;
                 Case false: node->value= false; break;
        }
}
```

Threaded Binary Tree

Threads

A binary tree has more NULL links than pointers. These null links can be replaced by special pointers, called threads, to other nodes in the tree.

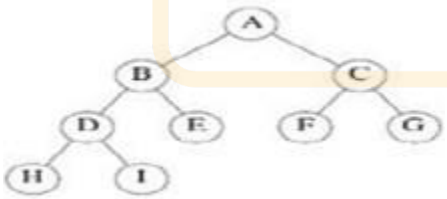
Two way threading
To construct the threads we use the following rules (assume that ptr represents a node):

- If ptr → leftChild is null, replace ptr → leftChild with a pointer to the inorder predecessor of ptr.
- . If ptr → rightChild is null, replace ptr → rightChild with a pointer to the inorder successor of ptr.

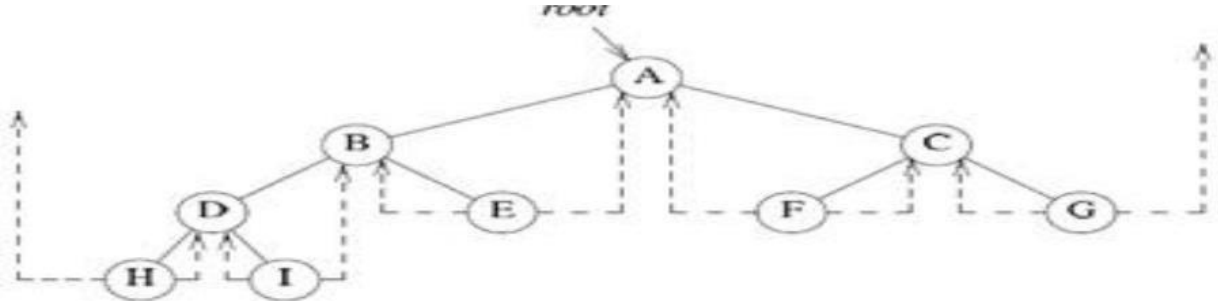
One way threading: If ptr → rightChild is null, replace ptr → rightChild with a pointer to the inorder successor of ptr. Ptr->left child remains unchanged.

Note : unless specified we consider threading corresponds to the inorder traversal. To distinguish the threads from ordinary pointers, threads are always drawn with broken links

Example: consider the binary tree and the corresponding threaded tree given below



Binary tree



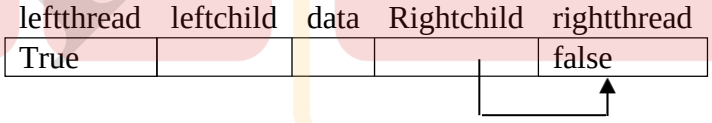
Threaded Binary Tree

- When we represent the tree in memory, we must be able to distinguish between threads and normal pointers.
- This is done by adding two additional fields to the node structure, *leftThread* and *rightThread*.
- Assume that *ptr* is an arbitrary node in a threaded tree.
  - If *ptr → leftThread = TRUE*, then *ptr→ leftChild* contains a thread; otherwise it contains a pointer to the left child.
  - Similarly, if *ptr → rightThread = TRUE*, then *ptr → rightChild* contains a thread; otherwise it contains a pointer to the right child.

This node structure is defined as follows

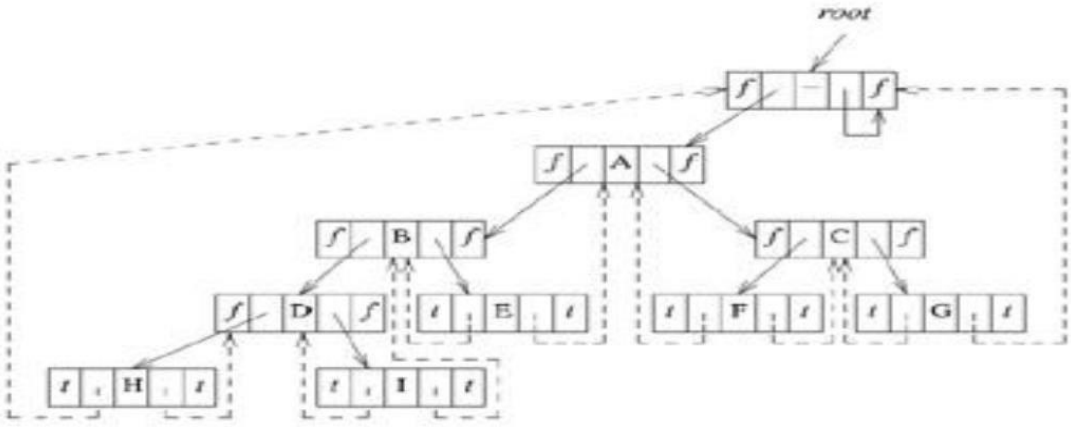
```
struct node{
short int leftThread;
struct node * leftChild;
char data;
thread node *rightChild;
short int rightThread;
};
typedef struct node ThreadNode;
```

- In Figure two threads have been left dangling: one in the left child of H, the other in the right child of G.
- To avoid loose threads, a header node is assumed for all threaded binary trees.
- The original tree is the left subtree of the header node.
- An empty binary tree is represented by its header node as in figure below



Empty threaded Binary tree

The complete memory representation for the tree is as below



Memory representation of threaded Tree

The variable root points to the header node of the tree, while root → leftChild points to the start of the first node of the actual tree.

4.6.1 Inorder Traversal of a Threaded Binary Tree

By using the threads, we can perform an inorder traversal without making use of a stack.

- For any node, *ptr*, in a threaded binary tree, if *ptr → rightThread = TRUE*, the inorder successor of *ptr* is *ptr → rightChild* by definition of the threads.
- Otherwise we obtain the inorder successor of *ptr* by following a path of left-child links from the right-child of *ptr* until we reach a node with *leftThread = TRUE*.

Finding the inorder successor of a node: The function *insucc* finds the inorder successor of any node in a threaded tree without using a stack.

```
ThreadNode  * insucc(ThreadNode  *tree)
{
ThreadNode * temp;
temp = tree→rightChild;
if (tree→rightThread==’f’)
while (temp→leftThread==’f’)
        temp = temp→leftChild;
return temp;
}
```

Inorder traversal of a threaded binary tree

- To perform an inorder traversal we make repeated calls to insucc
- This function assumes that the tree is pointed to by the header node's left child and that the header node's right thread is FALSE.

```
void Tinorder(ThreadNode * tree)
{


for (;;)
{

ThreadNode * temp = tree;


temp = insucc(temp);
if (temp == tree) break;
printf("%c", temp→data);
}
}
```

Analysis: The computing time for tinorder is still O(n)

Binary Search Trees

ADT Dictionary
objects: a collection of n > 0 pairs, each pair has a key and an associated item
functions: for all d ∈ Dictionary, item ∈ Item, k ∈ Key, n ∈ integer

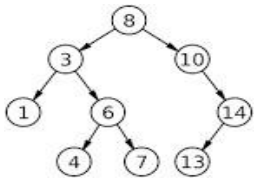
DictionaryCreate(max_size) ::=	create an empty Dictionary
Boolean IsEmpty(d, n) ::=	**if** (n > 0) **return** TRUE **else return** FALSE
Element Search(d, k) ::=	**return** item with key k, **return** NULL if no such element.
Element Delete(d, k) ::=	delete and return item (if any) with key k;
void Insert(d,item,k) ::=	insert item with key k into d.

Definition Binary search tree

A binary search tree is a binary tree. It may be empty. If it is not empty then it satisfies the following properties:

1) Each node has exactly one key and the keys in the tree are distinct.
2) The keys (if any) in the left subtree are smaller than the key in the root.
3) The keys (if any) in the right subtree are larger than the key in the root.
4) The left and right subtrees are also binary search trees.
5) The root has a key.

Example:



Binary search Tree

Searching a Binary Search Tree

To search for a node whose key is *k*. We begin at the root of the binary search tree.

- If the root is *NULL*, the search tree contains no nodes and the search is unsuccessful.
- we compare *k* with the key in root. If *k* equals the root's key, then the search terminates successfully.
- If *k* is less than root's key, then, we search the left subtree of the root.
- If *k* is larger than root's key value, we search the right subtree of the root.

Structure of the node can be defined as follows

```
struct node
{
Struct node *lchild;
struct
{
int  item;                /* Itype represents the data type of the element*/
int key;
}data;

Struct node *rchild;
};

Typedef struct node TreeNode;
```

Recursive search of a binary search tree: Return a pointer to the element whose key is k, if there is no such element, return NULL. We assume that the data field of the element is of type elemenet and it has two components key and item.

```
Treenode * search(TreeNode * tree, int k)
{
        if (tree==NULL) return NULL;
        if (k == tree→data.key)
        return (tree);
        if (k < tree→data.key)
            return search(tree→leftChild, k);
        return search(tree→rightChild, k);
}
```

Iterative search of a Binary Search tree

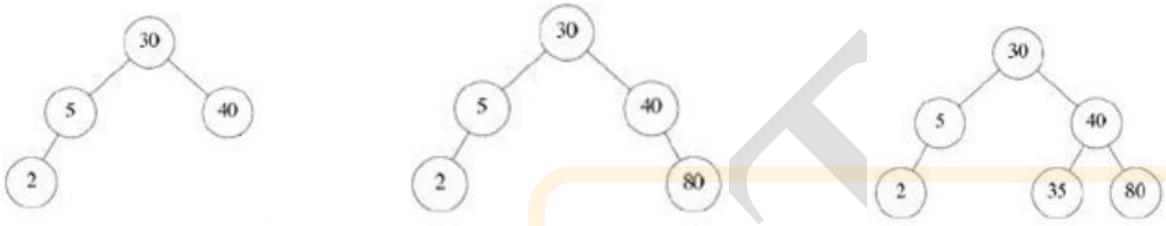
```
Treenode* iterSearch(TreeNode * tree, int k)

{
while (tree!=null)
{
        if (k == tree→data.key)
         return (tree);
        if (k < tree→data.key)
        tree = tree→leftChild;
        else
        tree = tree→rightChild;
}
return NULL;
```

```
}
```

Analysis of Search(Both iterative and recursive): If h is the height of the binary search tree, then we can perform the search using either search in O(h). However, recursive search has an additional stack space requirement which is O(h).

Inserting in to a Binary Search Tree: Binary search tree has distinct values, first we searchthe tree for the key and if the search is unsuccessful the key is inserted at the point the search terminated

Example : consider the tree given below



BST Tree **Insert 80** **Insert 35**

Inserting a dictionary pair into a binary search tree

If k is in the tree pointed at by node do nothing. Otherwise add a new node with data = (k, item)

```
TreeNode*  insert(TreeNode *root, int k, int Item)
{
TreeNode * ptr, *lastnode;

lastnode=Modifiedsearch(root,k);
Ptr=(TreeNode*)malloc(sizeof(TreeNode));
ptr→data.key = k;
ptr→data.item = Item;
ptr→leftChild = ptr→rightChild = NULL;


if (root==NULL)
{
 root=ptr;
return(root);
}

if(lastnode!=NULL)
{
        if (k < lastnode→data.key)
        lastnode→leftChild = ptr;
        else
        lastnode→rightChild = ptr;
        return (root);
}
```

```
}
```

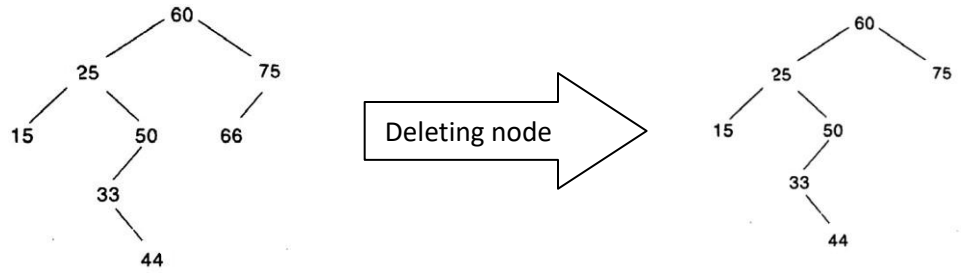
If the element is present or if the tree is empty the function Modifiedsearch returns NULL. If the element is not present it retrun a pointer to the last node searched.

```
Modifiedsearch(Treenode *root,int k)

{

TreeNode *temp,*prev;
temp==node;
prev=NULL;
If(temp==NULL)
return(NULL);
while(temp!=NULL)
{
        if(temp->data.key==k)
        {
                printf("element already found");
                return(NULL);
        }
        if(key<temp->data.key)
        {
        Prev=temp;
        temp=temp->rcchild;
        }
        else
        {
        Prev=temp;
        Temp=temp->rchild;
        }
}
retrun(prev);

}
```

Deletion from a binary search tree: Suppose T is a a binary search tree. The function to delete an item from tree T first search the tree to find the location of the node with the item and the location of the parent of N and the deletion of the node N depends on three cases:

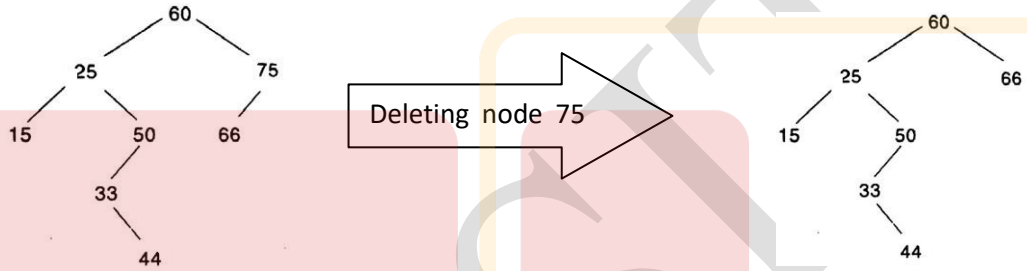
Case 1: N has no children. Then N is deleted from T by replacing the location of the node N in the parent(N) by the NULL pointer

Example: Deleting Node 66 with NO children



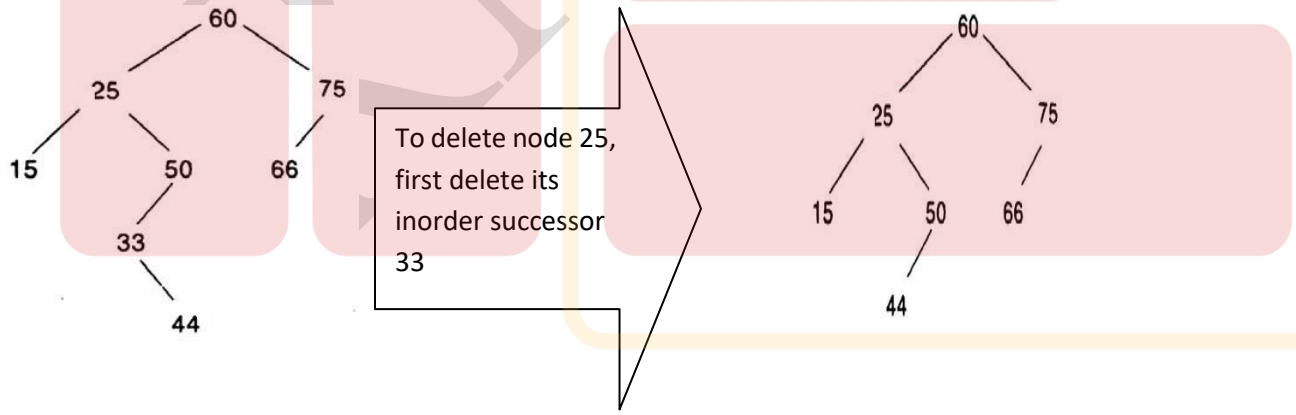
Case 2: If N has exactly one child. Then N is deleted from T by replacing the location of N in Parent (N) by the location of the only child of N.

Example: Deleting Node 75 with exactly one children

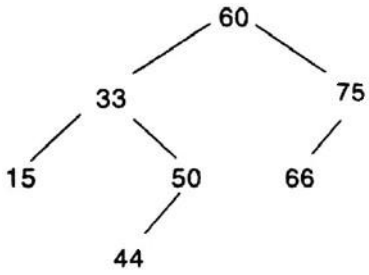


Case 3: N has Two children. Let S(N) denote the inorder successor of N(S(N) does not have a left child).Then N is deleted from T by first deleting S(N) from T (by using case1 or cae 2) and then replacing node N in T by the node S(N).

Example: Deleting Node 25 with two children



Now replace the node 25 with its inorder successor 33



Recursive function to delete a node in a BST

```
TreeNode *delete_element(TreeNode *node, int key)
{
TreeNode * temp;

   if (node == NULL)
      return node;


   if (key < node->data.key)
      node->lchild = delete_element(node->lchild, key);
   else if (key > node->data.key)
     node->rchild = delete_element(node->rchild, key);
else
{
  // node with only one child
  if (node->lchild == NULL)
  {
     temp = node->rchild;
     free(node);
     return temp;
  }
  else if (node->rchild == NULL)
  {
     temp = node->lchild;
     free(node);
     return temp;
  }
  // node with two children

  else
  {
     temp = node->rchild;
     while(temp->lchild!=NULL)      //Get the inorder successor
              temp=temp->lchild;
      node->data.item = temp->data.item;
     node->data.key=temp->data.key;
      node->rlink = delete_element(node->rchild, temp->data.key);
      return node;

  }
}}
```

MODULE - 4

TREES(Cont..): Binary Search trees, Selection Trees, Forests, Representation of Disjoint sets, Counting Binary Trees,
GRAPHS: The Graph Abstract Data Types, Elementary Graph Operations

Binary Search Trees

ADT Dictionary
objects: a collection of n > 0 pairs, each pair has a key and an associated item
functions: for all d ∈ Dictionary, item ∈ Item, k ∈ Key, n ∈ integer

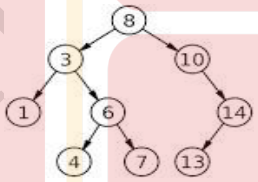
DictionaryCreate(max_size) ::=	create an empty Dictionary
Boolean IsEmpty(d, n) ::=	**if** (n > 0) **return** TRUE **else return** FALSE
Element Search(d, k) ::=	**return** item with key k, **return** NULL if no such element.
Element Delete(d, k) ::=	delete and return item (if any) with key k;
void Insert(d,item,k) ::=	insert item with key k into d.

Definition Binary search tree

A binary search tree is a binary tree. It may be empty. If it is not empty then it satisfies the following properties:

1) Each node has exactly one key and the keys in the tree are distinct.
2) The keys (if any) in the left subtree are smaller than the key in the root.
3) The keys (if any) in the right subtree are larger than the key in the root.
4) The left and right subtrees are also binary search trees.
5) The root has a key.

Example:



Binary search Tree

Searching a Binary Search Tree

To search for a node whose key is *k*. We begin at the root of the binary search tree.

- If the root is *NULL*, the search tree contains no nodes and the search is unsuccessful.
- we compare *k* with the key in root. If *k* equals the root's key, then the search terminates successfully.
- If *k* is less than root's key, then, we search the left subtree of the root.
- If *k* is larger than root's key value, we search the right subtree of the root.

Structure of the node can be defined as follows

```
struct node
{
Struct node *lchild;
struct
{
int  item;                /* Itype represents the data type of the element*/
int key;
}data;

Struct node *rchild;
};

Typedef struct node TreeNode;
```

Recursive search of a binary search tree: Return a pointer to the element whose key is k, if there is no such element, return NULL. We assume that the data field of the element is of type elemenet and it has two components key and item.

```
Treenode * search(TreeNode * tree, int k)
{
        if (tree==NULL) return NULL;
        if (k == tree→data.key)
        return (tree);
        if (k < tree→data.key)
            return search(tree→leftChild, k);
        return search(tree→rightChild, k);
}
```

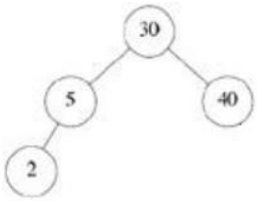
Iterative search of a Binary Search tree

```
Treenode* iterSearch(TreeNode * tree, int k)

{
while (tree!=null)
{
        if (k == tree→data.key)
         return (tree);
        if (k < tree→data.key)
        tree = tree→leftChild;
        else
        tree = tree→rightChild;
}
return NULL;
}
```

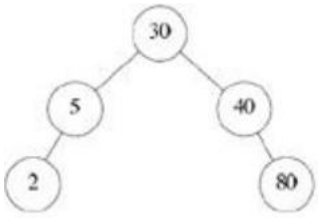
Analysis of Search(Both iterative and recursive): If h is the height of the binary search tree, then we can perform the search using either search in O(h). However, recursive search has an additional stack space requirement which is O(h).

Inserting in to a Binary Search Tree: Binary search tree has distinct values, first we searchthe tree for the key and if the search is unsuccessful the key is inserted at the point the search terminated

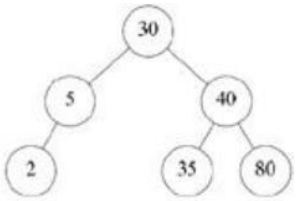
Example : consider the tree given below



BST Tree



Insert 80



Insert 35

Inserting a dictionary pair into a binary search tree

If k is in the tree pointed at by node do nothing. Otherwise add a new node with data = (k, item)

```
TreeNode*  insert(TreeNode *root, int k, int Item)
{
TreeNode * ptr, *lastnode;

lastnode=Modifiedsearch(root,k);
Ptr=(TreeNode*)malloc(sizeof(TreeNode));
ptr→data.key = k;
ptr→data.item = Item;
ptr→leftChild = ptr→rightChild = NULL;


if (root==NULL)
{
 root=ptr;
return(root);
}

if(lastnode!=NULL)
{
        if (k < lastnode→data.key)
        lastnode→leftChild = ptr;
        else
        lastnode→rightChild = ptr;
        return (root);
}

}
```

If the element is present or if the tree is empty the function Modifiedsearch returns NULL. If the element is not present it retrun a pointer to the last node searched.

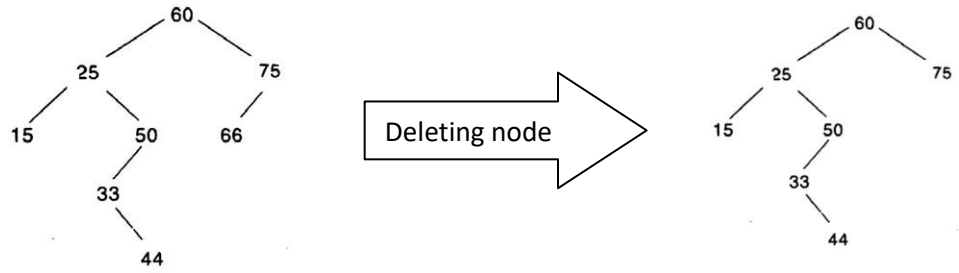
```
Modifiedsearch(Treenode *root,int k)
```

```
{

TreeNode *temp,*prev;
temp==node;
prev=NULL;
If(temp==NULL)
return(NULL);
while(temp!=NULL)
{
        if(temp->data.key==k)
        {
                printf("element already found");
                return(NULL);
        }
        if(key<temp->data.key)
        {
        Prev=temp;
        temp=temp->rcchild;
        }
        else
        {
        Prev=temp;
        Temp=temp->rchild;
        }
}
retrun(prev);

}
```

Deletion from a binary search tree: Suppose T is a a binary search tree. The function to delete an item from tree T first search the tree to find the location of the node with the item and the location of the parent of N and the deletion of the node N depends on three cases:

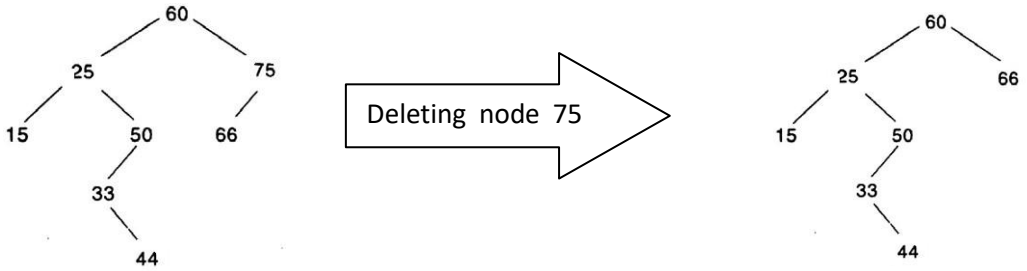
Case 1: N has no children. Then N is deleted from T by replacing the location of the node N in the parent(N) by the NULL pointer

Example: Deleting Node 66 with NO children



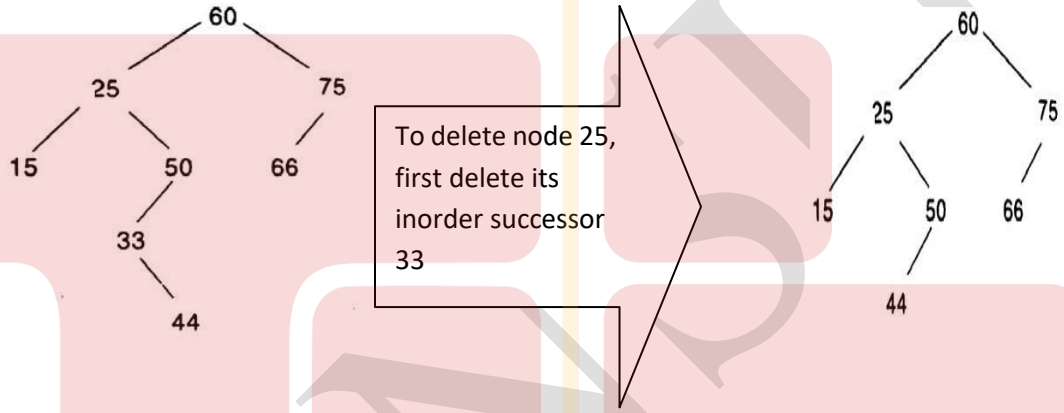
Case 2: If N has exactly one child. Then N is deleted from T by replacing the location of N in Parent (N) by the location of the only child of N.

Example: Deleting Node 75 with exactly one children

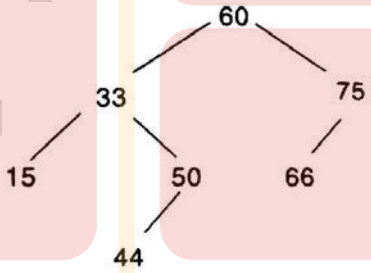


Case 3: N has Two children. Let S(N) denote the inorder successor of N(S(N) does not have a left child).Then N is deleted from T by first deleting S(N) from T (by using case1 or cae 2) and then replacing node N in T by the node S(N).

Example: Deleting Node 25 with two children



Now replace the node 25 with its inorder successor 33



Recursive function to delete a node in a BST

```
TreeNode *delete_element(TreeNode *node, int key)
{
TreeNode * temp;

   if (node == NULL)
      return node;


   if (key < node->data.key)
      node->lchild = delete_element(node->lchild, key);
   else if (key > node->data.key)
     node->rchild = delete_element(node->rchild, key);
```

```
else
{
   // node with only one child
   if (node->lchild == NULL)
   {
      temp = node->rchild;
      free(node);
      return temp;
   }
   else if (node->rchild == NULL)
   {
      temp = node->lchild;
      free(node);
      return temp;
   }
   // node with two children

   else
   {
      temp = node->rchild;
      while(temp->lchild!=NULL)      //Get the inorder successor
               temp=temp->lchild;
       node->data.item = temp->data.item;
      node->data.key=temp->data.key;
       node->rlink = delete_element(node->rchild, temp->data.key);
       return node;

   }
}
}
```

GRAPHS

Introduction

The first recorded evidence of the use of graph dates back to 1736. When Leonhard Euler used them to solve the classical Konigsberg bridge problem.

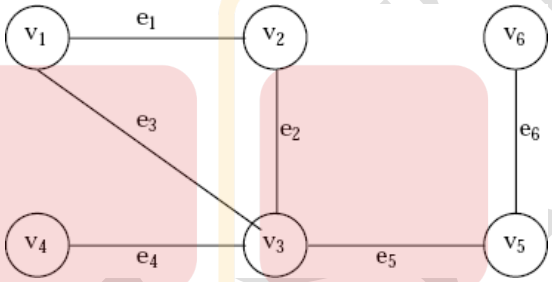
Definitions

Graph: A graph G consist of two sets V and E

1. V is a finite nonempty set of vetices and
2. E is a set of pairs of vertices these pairs are called edges

A graph can be represents as G = (V, E). V(G) will represent the set of vertices and E(G) will represent the set of edges of the graph G

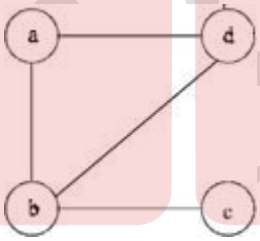
Example:



V(G)= {v1, v2, v3, v4, v5, v6}
E(G) = {e1, e2, e3, e4, e5, e6} E(G) = {(v1, v2) (v2, v3) (v1, v3) (v3, v4), (v3, v5) (v5, v6)}.
There are six edges and six vertex in the graph

Undirected Graph: In a undirected graph the pair of vertices representing an edge is unordered. thus the pairs (u,v) and (v,u) represent the same edge.

Example:

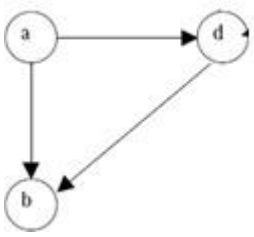


V(G)={a,b,c,d}

E(G)={(a,b),(a,d),(b,d),(b,c)

Directed Graph (digraph): In a directed graph each edge is represented by a directed pair (u,v), v is the head and u is the tail of the edge. Therefore (v,u) and(<u,v) represent two different edges

Example:

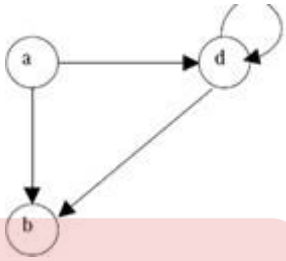


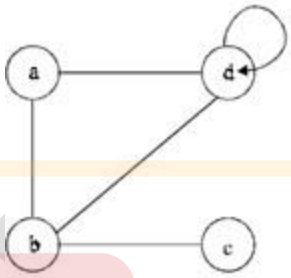
V(G)={a,b,d}

E(G)={(a,d), (a,b), (d,b)}

Self Edges/Self Loops: Edges of the form(v,v) are called self edges or self loops . It is an edge which starts and ends at the same vertex.

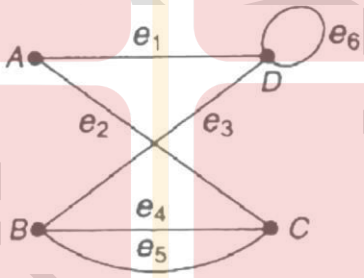
Example:





Mutigraph: A graph with multiple occurrences of the same edge is called a multigraph

Example:



Complete Graph: An undirected graph with n vertices and exactly n(n-1)/2 edges is said to be a complete graph. In a graph all pairs of vertices are connected by an edge.

Example : A complete graph with n=3 vertices

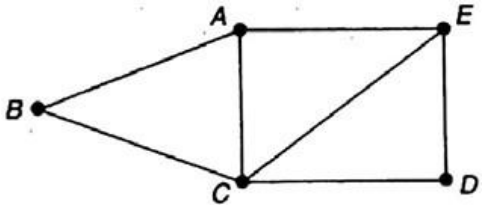
Adjacent Vertex

If (u,v) is an edge in E(G), then we say that the vertices u and v are adjacent and the edge(u,v) is incident on vertices u and v.

Path: A path from vertex u to v in graph g is a sequence of vertices u,i1,i2,…….ik,v such that (u,i1),(i1,i2)……(ik,v) are edges in E(G). if G' is directed then the path consists of <u,i1>,<i1,i2>……<ik,v> edges in E(G').

The length of the path is the number of edges in it.

Example:



(B,C),(C,D) is a path from B to D the length of the path is 2

A simple path is a path in which all the vertices are distinct.

Cycle: A cycle is a simple path in which all the vertices except the first and last vertices are distinct. The first and the last vertices are same.

Example :

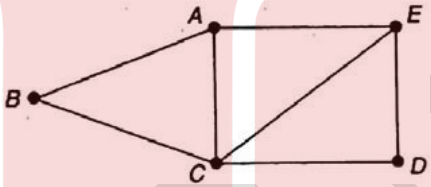
(B,C),(C,D)(D,E)(E,A)(A,B) is a cycle

Degree of a vertex : In **a undirected graph** degree of a vertex is the number of edges incident on a vertex.

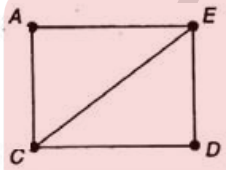
In a **directed graph** the **in-degree** if a vertex v is the number of edges for which v is the head i.e. the number of edges that are coming into a vertex. The **out degree** is defined as the number of edges for which v is the tail i.e. the number of edges that are going out of a vertex

Subgraph: A subgraph of G is a graph G' such that V(G') $\subseteq$ V(G) and E(G') $\subseteq$ E(G)

Example :



Graph(G)



Subgraph(G')

Connected Graph: An undirected graph G is said to be connected if for every pair of distinct vertices u and v in V(G) there is a path from u to v in G.

Connected Component is a maximal connected subgraph

Strongly connected graph : A directed graph G is said to be strongly connected if for every pair of distinct vertices u an v in V(G), there is a directed path from u to v and from v to u.

Tree: A tree is a connected acyclic connected graph.

ADT Graph

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

Functions: for all graph$\in$ Graph, v,v1,v2 $\in$ vertices

Example: Create():=	**return** an empty graph
Graph InsertVertex(graph,v):=	**return** a graph with v inserted. v has no incident edges
Graph InsertEdge(graph,v1,v2) :=	**retrun** a graph with a new edge between v1 and v2
Graph DeleteVertex(graph,v) :=	**return** a graph in which v and all edges incident to it is removed
Graph DeleteEdge(graph,v1,v2):=	**retrun** a graph in which the edge (v1,v2) is removed, leave the incident nodes in the graph
Boolean IsEmpty:=	If (graph == empty graph) retrun TRUE else Retrun FALSE
List Adjacent(graph,v) :=	**retrun** a list of all vertices that are adjacent to v

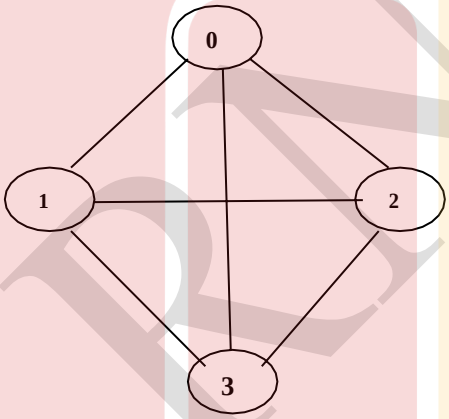
Graph Representation

The three most commonly used representations are

- Adjacency Matrix
- Adjacency List
- Adjacency Multilist

Adjacency Matrix: Let G=(V,E) be a graph with n vertices, n>=1. The adjacency matrix of G is a two dimensional n*n array for example a, with the property that a[i][j]=1 if there exist ane edge (i,j) (for a directed graph edge <i,j> is in E(G).a[i][j]=0 if no such edge in G.

Example:



Adjacency Matrix

	0	1	2	3
0	0	1	1	1
1	1	0	1	1
2	1	1	0	1
3	1	1	1	0

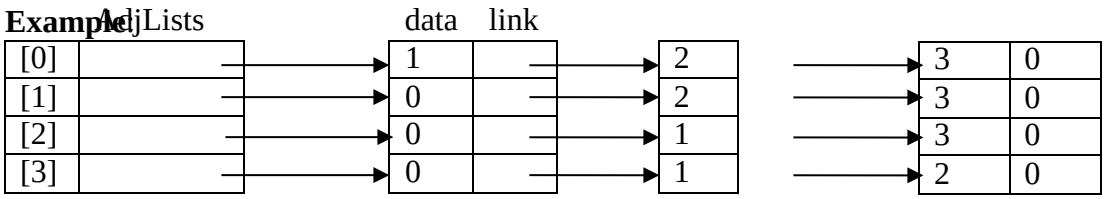
Figure 5.1 Graph G1

- The space requirement to store an adjacency matrix is n^2 bits.
- The adjacency matrix for a undirected graph is symmetric .About half the space can be saved in an undirected graph by storing only the upper or lower triangle of the matrix.
- For an undirected graph the degree of any vertex i is its row sum. For a directed graph the row sum is the out-degree and the column sum is the in-degree.

Adjacency list: In adjacency matrix the n rows of the adjacency matrix are represented as n chains. There is one 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.

Example: the adjacency list of graph G1 in figure 5.1 is shown below

Example AdjLists



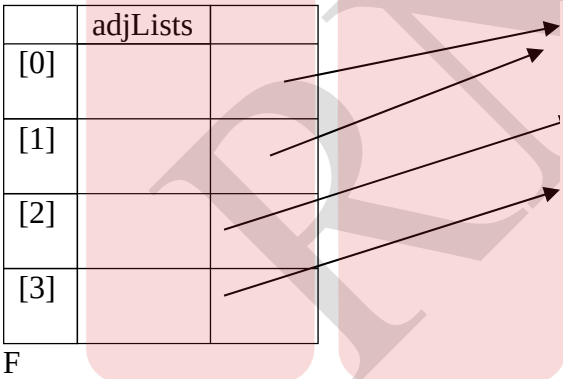
- For an undirected graph with n vertices and e edges. The linked adjacency lists representation requires an array of size n and 2e chain nodes.
- The degree of any vertex in an undirected graph may be determined by counting the number of nodes in the adjacency list.
- For a digraph the number of list nodes is only e.

Adjacency Multi lists: For each edge there will be exactly one node, but this node will be in two list(i.e., the adjacency list for each of the two nodes to which it is incident). A new field is necessary to determine if the edge is determined and mark it as examined.

The new node structure is

m	Vertex1	Vertex2	Link1	Link2

Example: The adjacency multilist for graph G1 is shown below



F

N0 | | 0 | 1 | N1 | N3 | edge(0,1)

N1 | | 0 | 2 | N2 | N3 | edge(0,2)

N2 | | 0 | 3 | 0 | N4 | edge(0,3)

N3 | | 1 | 2 | N4 | N5 | edge(1,2)

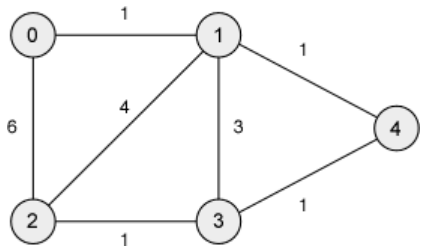
N4 | | 1 | 3 | 0 | N5 | edge(1,3)

N5 | | 2 | 3 | 0 | 0 | edge(2,3)

The Lists are Vertex 0: N0->N1->N2
Vertex 1: N0->N3->N4
Vertex 2: N1->N3->N5
Vertex 3: N2->N4->N5

Weighted Edges: In many applications the edges of a graph have weight assigned to them. These weights may represent the distance from one vertex t o another or the cost for going from one vertex to an adjacent vertex. The adjacency matrix and list maintains the weight information also. A graph with weighted edges are also called network.

Example:



Elementary Graph Operations

Given an undirected graph G=(V,E) and a vertex v in V(G) ,there are two ways to find all the vertices that are reachable from v or are connected to v .

- Depth First Search and
- Breadth First Search

Depth First Search

1. Visit the starting vertex v. (visiting consist of printing node's vertex)
2. Select an unvisited vertex w from v's adjacency and carry a depth first search on w.
3. A stack is maintained to preserve the current position in v's adjacency list.
4. When we reach a vertex u that has no unvisited vertices on adjacency list, remove a vertex from the stack and continue processing its adjacency list. Previously visited vertices are discarded and unvisited vertices are placed on stack
5. The search terminates when the stack is empty.

A recursive implementation of depth first search is shown below.

A global array visited is maintained , it is initialized to false, when we visit a vertex i we change the visited[i] to true.

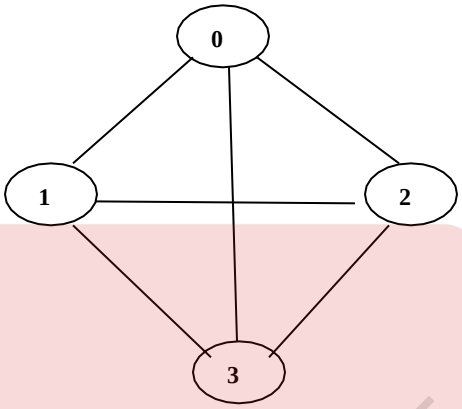
Global Declaraions

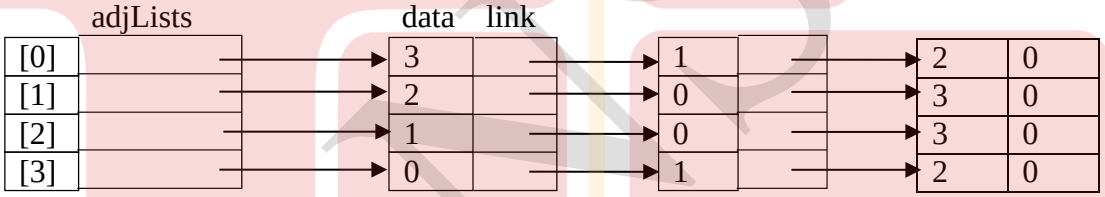
```
# define FALSE 0
# define true 1
Short int visited[max_vertices];

void dfs(int v)
{
        visited[v]=TRUE;
        printf("%d",v);
        w=graph[v]
        while(w!=NULL)
        {
                If(visited[w->vertex]==FALSE)
                        dfs(w->vertex);
                w=w->link;

        }}
```

Analysis:

- If we represent G by its adjacency list then we can determine the vertices adjacent to v by following a chain of links. Since dfs examines each node in the adjacency list at most once then the time to complete the search is O(e).
- If we represent G by its adjacency matrix then determining all vertices adjacent o v requires O(n) time. Since we visit at most n vertices the total time is $O(n^2)$.

Example: For the graph given below if the search is initiated from vertex 0 then the vertices are visited in the order vertex 3, 1, 2





Breadth first Search

1. Search starts at vertex v marks it as visited.
2. It then visits each of the vertices on v's adjacency list.
3. As we visit each vertex it is placed on a queue.
4. When all the vertices in the adjacency list is visited we remove a vertex from the queue and proceed by examining each of the vertices in its adjacency list.
5. Visited vertices are ignored and unvisited vertices are placed on the queue
6. The search terminates when the queue is empty.

The queue definition and the function prototypes

```
 struct node
{
        int vertex;
        struct node * link;
};
typedef struct node queue;
```

```
queue * front,*rear;
int visied[max_vertics];

void addq(int);
int delete();

void bfs(int v)
{
front=rear=NULL;
printf("%d",v);
visisted[v]= TRUE;
addq(v);
while(front)
{
v=deleteq();
while(w!=NULL)
        {
                if(visited[w->vertex]==FALSE)
                {
                        printf("%d",w->vertex);
                        addq(w->vertex);
                        visited[w->vertex]=TRUE;
                }
                        w=w->link;
        }
}
}
```

Analysis of BFS:

- For each vertex is placed on the queue exactly once, the while loop is iterated at most n times.
- For the adjacency list representation the loop has a total cost of O(e). For the adjacency matrix representation the loop takes O(n) times
- Therefore the total time is O (n^2).

MODULE - 5

HASHING: Introduction, Static Hashing, Dynamic Hashing
PRIORITY QUEUES: Single and double ended Priority Queues, Leftist Trees
INTRODUCTION TO EFFICIENT BINARY SEARCH TREES: Optimal Binary Search Trees

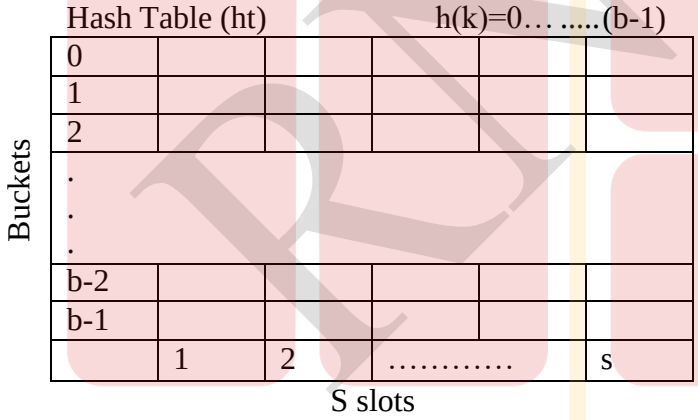
Hashing

Hashing enables us to perform dictionary operations like search insert and delete in O(1) time. There are two types of hashing

- Static and
- Dynamic

Static Hashing

- In static Hashing the dictionary pairs are stored in a table, ht called the **hash table**.
- The hash table is partitioned into **b buckets**, ht[0],…….ht[b-1]
- Each bucket is capable of holding s dictionary pairs.
- Thus a bucket is said to consist **of s slots**. usually s=1
- The address or location of a pair whose key is k is determined by **hash function h which maps keys into buckets.**
- Thus for any key k, h(k) is an integer in range 0 through b-1



The key density of a hash table is the ratio n/T

- n is the number of pairs in the table
- T is possible keys

The loading density or loading factor of a hash table is a = n/(sb)

- s is the number of slots
- b is the number of buckets

b=26, s=2
n=10 distinct identifiers- each representing a C library function
Loading factor a = n/(sb) = 10/52=0.19
f(x)= first character of x
x: acos, define, float, exp, char, atan, ceil, floor, clock, ctime
f(x) : 0, 3, 5, 4, 2, 0, 2, 5, 2, 2

	Slot0	Slot1
0	**Acos**	**atan**
1		
2	**Char**	**ceil**
3	**Define**	
4	**exp**	
5	**foat**	**floor**
. .		
24		
25		

Hash Functions: A hash function maps a key into a bucket in the hash table. A function H from the set K of keys into the set L of memory addresses is called the hash function

H:K->L

Desired Properties are
- Easy computation
- Minimal number of collisions
- Uniformly distribute the hash addresses throughout the set L

Division : Chose a number m larger than the number n of keys in K. The number m is chosen to be a prime number or a number without small divisors to reduce collisions. The function is defined as

h(k)=k%m or h(K)= k mod m

Bucket addresses range from 0 to m-1 and the hash table must have m buckets

Example:if m=10 then h(25)=5, h(32)=2

Mid Square: In this method the square of the key is found and appropriate number of bits are used from the middle of the square to obtain the bucket address

- $F(K)=middle(K^2)$
- The number of bits used to obtain bucket address depends on table size.
- If r bits are used the range of values is 0 through 2^r-1

Example: K=3205 K^2 = 10272025 H(K)= 72

Folding: Partition the keys k into several parts

- All parts except for the last one have the same length
- The parts are added together to obtain the hash address
- Two possibilities

Example k= 12320324111220

x1=123, x2=203, x3=241, x4=112, x5=20, address= 123+203+241+112+20= 699

Digit Analysis

- Useful in the case of a static file where all the keys in the table are known in advance
- Each key is interpreted using some radix r.
- The same radix is used for all the keys in the table
- Digits are examined with this radix
- Digits having the most skewed distributions are deleted.
- Enough digits are deleted so that the remaining digits are small enough to give and address in the range of hash table

Converting keys to integers

Two methods used for converting keys to integer are

- Converting each character to a unique integer and summing these unique integers.
- Shifting the integer corresponding to every other character by 8 bits and then summing it up

Over Flow Handling

Synonyms: Hash function h maps several different keys into the same bucket
Two keys, k1 and k2 are synonyms with respect to h
if h(k1) = h(k2)

An **overflow** occurs when home bucket for a new dictionary pair is full when we wish to insert this pair

A **collision** occurs when the home bucket for the new pair is not empty at the time of insertion.

Two popular ways To handle overflows

- Open Addressing/ Linear Probing
- Chaining

Open Addressing/Linear Probing

- When inserting a new pair whose key is k we search the hash table in the order
  ht[h(k) +i]%b, 0<=i <=b-1, where
  - h is the hash function and
  - b is the number of buckets
- The search terminates when we reach the first unfilled bucket and the new pair is inserted into this bucket.
- Incase no such bucket is found, the table is full and the size of the hash table needs to be

increased.

- For good performance the table size is increased when loading density exceeds a prescribed threshold such as 0.75 rather when the table is full.
- When the hash table is resized
  - Hash function changes
  - Home bucket of each key may change

Example:

Suppose the table T has 11 memory locations T[1]……T[11] and suppose the file f contains 8 records with the following hash addresses

Records	A	B	C	D	E	X	Y	Z
H(K)	4	8	2	11	4	11	5	1

Suppose these 8 records are entered into the hash table in the above order the hash table will look as shown below.

Table T	X	C	Z	A	E	Y	-	B	-	-	D
Address	1	2	3	4	5	6	7	8	9	10	11

The average number S of probes for a successful search is S= (1+1+1+1+2+2+2+3)/8=13/8=1.6

The average number U of probes for a unsuccessful search is

U= (7+6+5+4+3+2+1+2+1+1+8)/11=40/11=3.6

Example-2

Assume a 13 bucket table with 1 slot per bucket

Identifier	Additive Transform	x	Hash
for	102+111+114	327	2
do	100+111	211	3
while	119+104+105+108+101	537	4
if	105+102	207	12
else	101+108+115+101	425	9
function	102+117+110+99+116+105+111+110	870	12

0	1	2	3	4	5	6	7	8	9	10	11	12
function		for	do	while					else			if

Searching a key using linear Probing

- Compute h(k)
- Examine the hash table in the order ht[h(k) +i]%b, 0<=i <=b-1, untill one of the follwing happens
  - The bucket ht[h(k) +i]%b contains the key k and the desired pair is found
  - ht[h(k) +i]%b is empty; k is not in the table.
  - Return to ht[h(k)], the table is full and k is not in the table

Drawbacks of Linear Probing

- Identifiers tend to cluster together
- Adjacent cluster tend to coalesce
- Increase the search time

Example showing the drawback

Insert acos, atoi,char,define,exp,ceil,cos, float, atol, floor , ctime into a 26 bucket hash table

bucket	x	Bucket searched
0	acos	1
1	atoi	2
2	char	1
3	define	1
4	exp	1
5	ceil	4
6	cos	5
7	float	3
8	atol	9
9	floor	5
10	ctime	9
…..		
25		

We see the number of searches increasing and the keys clustering together

Quadratic Probing

- Quadratic probing uses a quadratic function of *i* as the increment
- Suppose a record R with key k has the hash addres H(k)=h then instead of searching the locations with h,h+1, h+2,……. we linearly search locations with h,h+1,h+4,h+9, ... $h+i^2$
- If the number m of locations in the table T is a prime number, then the above sequence will access half of the locations T

Double hashing

Here a second hash function H' is used for resolving a collision, as follows.

Suppose a record R with key k has the hash address H(k)=h and h'(k)=h'≠ m then we linearly search locations with addresses h, h+h',h=2h'.h+3h',……

If m is a prime number then the above sequence will access all the locations in the table T.

Note: One **major disadvantage in any type of open addressing procedure** is in the implementation of deletion.

Suppose a record r is deleted from location T(r) , suppose we reach this location during a search, it does not mean the search is unsuccesssfull..

Thus when deleting a record the location should be labeled to indicate that previously it did contain a record

Chaining

- Maintain one list per bucket

- Each list containing the synonyms for that bucket.
- Search involves
  - Computing the hash address h(k), and
  - Examining the keys in the list of h(k)

Example: Insert acos, atoi,char,define,exp,ceil,cos, float, atol, floor , ctime into a 26 bucket hash table maintained as hash chain

[0]	→	acos-> atoi-> atol
[1]	→	NULL
[2]	→	char -. Ceil-> cos -> ctime
[3]	→	define
[4]	→	exp
[5]	→	float-.floor
[6]	→	NULL
. . .		
[25]	→	NULL

5.7.4 Rehashing: When the hash table becomes nearly full, the number of collisions increases, thereby degrading the performance of insertion and search operations. In such cases, a better option is to create a new hash table with size double of the original hash table.

All the entries in the original hash table will then have to be moved to the new hash table. This is done by taking each entry, computing its new hash value, and then inserting it in the new hash table. Though rehashing seems to be a simple process, it is quite expensive and must therefore not be done frequently.

Example:
Consider the hash table of size 5 given below. The hash function used is h(x)= x % 5.

0	1	2	3	4
	26	31	43	17

Rehash the entries into to a new hash table using hash function—h(x)= x % 10.

0	1	2	3	4	5	6	7	8	9
	31		43			26	17		

Dynamic hashing

Limitation of static hashing: when the table tends to be full, overflow increases and reduces performance.

To ensure good performance, it is necessary to increase the size of a hash table whenever the loading density exceeds a prescribed threshold.

When the loading density increases array doubling is used to increase the size of the array to 2b+1.Change in divisor causes us to rebuild the hash table by reinserting the key in the smaller table. Dynamic hashing or extendible hashing reduces the rebuild time.

There are two forms of dynamic hashing

- Dynamic hashing using directories
- Directory less dynamic hashing

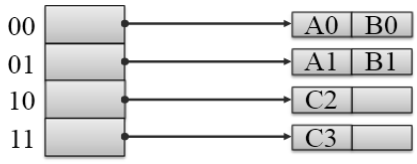
Example: Hash function that transforms keys into 6 bit non negative integers. H(k,t) denote the integers formed by the '**t**' least significant bits of h(k).
The example taken is a two letter key. H transforms Letter A,B,C into bit sequnce 100,101 and 110 respectively Digits 0 through 7 are transformed into their 3 bit representation

k	h(k)
A0	100 000
A1	100 001
B0	101 000
B1	101 001
C1	110 001
C2	110 010
C3	110 011
C5	110 101

Dynamic Hashing using directories

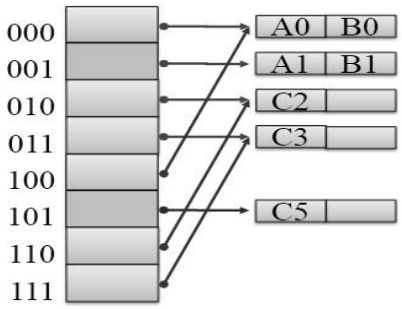
- A directory of d of pointers to buckets are used
- The number of bits of h(k) used to index the directory is called the directory depth.
- Size of directory depends on the number of bits of h(k) used to index into the directory.
- Directory size d= 2^t where t is the number of bits used to identify all h(k).
- Initially t=2 bits then d= 2^2=4
- H(k,t) denote the integers formed by the t least significant bits of h(k).

Example: Figure below shows a dynamic hash table that contain the keys A0, B0,A1,B1,C2 and C3. Here the directory depth is 2 and uses buckets that have 2 slots. For each key k,



we examine the bucket pointed to by d[h(k,t)] where t is the directory depth. Suppose we insert C5 into the hash table since h(c5,2)=01 we follow the pointer d[01] and this bucket is full. To resolve the overflow, we determine the least u such that h(k,u) is not the same for all keys. Incase u is greater than the directory depth we increase the directory depth to this least value u. Figure below

shows the table after inserting C5



Advantages

- Only the directory doubles hash table remains the same
- Only the entries that overflows needs to be rehashed

Directory Less Dynamic Hashing

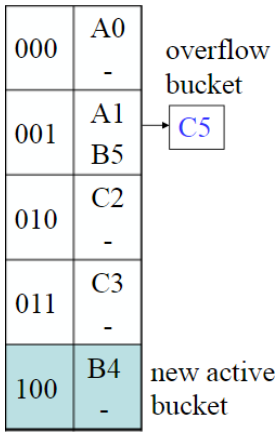
- Also known as liner dynamic hashing
- Directory is not used instead an **array ht of buckets** is used.
- We assume that this array is as large as possible so there is no possibility of increasing the size dynamically
- To avoid initializing such a large array, two variables are used **r and q,** $0<=q<=2^r$. It keeps track of the active buckets.
- At any time only the buckets 0 to $2^r + q-1$ are active
- Each active bucket is the start of a chain of buckets.
- The remaining buckets in the chain are called overflow buckets.
- Each dictionary pair is either in a active or an overflow bucket.

Figure Below shows a directory less hash table ht with r=2 and q=0. The number of active bucket is 4. The index of the active bucket identifies its chain.. Each active bucket has 2 slots.

00	B4 A0
01	A1 B5
10	C2 -
11	C3 -

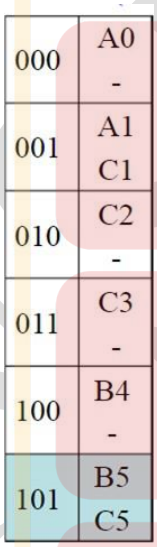
r=2, q=0

When we insert C5 into the table, chain 01 is examined and we verify that C5 is not present. Since the active bucket for the searched chain is full we get an overflow. An overflow is handled by activating bucker 2^r+q, reallocating the entries in the chain q then the value of q is incremented by 1.incase q becomes 2^r. We increment r by 1 and reset q to 0. The reallocation is done usingh(k,r+1). Finally the new pair is inserted into the chain.



r=2,q=1

Insert C1 will again result in an overflow at 001 so the bucket 5=100 is activated . Rehashing is done and the table is as shown below.



r=2,q=2