

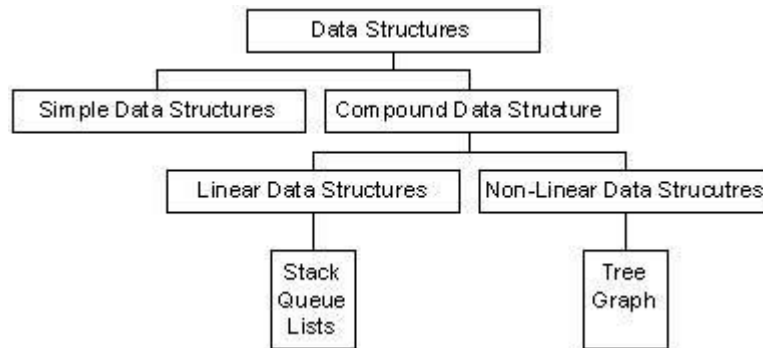
**Module -1****Introduction****Teaching Hours: 10**

	<b>CONTENTS</b>	<b>Pg no</b>
<b>1.</b>	<b><i>Introduction to Data Structures</i></b>	<b><i>2</i></b>
	1.1. Classification of Data Structures	<b><i>2</i></b>
	1.2. Data structure Operations	<b><i>3</i></b>
	1.3. Review of Structures Unions and Pointers	<b><i>3</i></b>
	1.4. Self Referential Structures	<b><i>10</i></b>
<b>2.</b>	<b><i>Arrays</i></b>	<b><i>11</i></b>
	2.1. Operations	<b><i>14</i></b>
	2.2. Multidimensional Arrays	<b><i>20</i></b>
	2.3. Applications of Arrays	<b><i>22</i></b>
<b>3.</b>	<b><i>Strings</i></b>	<b><i>23</i></b>
	3.1. String manipulation Applications	<b><i>26</i></b>
<b>4.</b>	<b><i>Dynamic Memory Management Functions</i></b>	<b><i>26</i></b>

## 1. Introduction to Data Structures

Data Structure is a way of collecting and organising data in such a way that we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage. For example, we have data player's name "Virat" and age 26. Here "Virat" is of **String** data type and 26 is of integer data type

### 1.1. Classification of Data Structures



**Data structures can be classified as**

- Simple data structure
- Compound data structure
- Linear data structure
- Non linear data structure

#### **Simple Data Structure:**

Simple data structure can be constructed with the help of primitive data structure. A primitive data structure used to represent the standard data types of any one of the computer languages. Variables, arrays, pointers, structures, unions, etc. are examples of primitive data structures.

#### **Compound Data structure:**

Compound data structure can be constructed with the help of any one of the primitive data structure and it is having a specific functionality. It can be designed by user. It can be classified as

- 1) Linear data structure
- 2) Non-linear data structure

Linear data structure :

Collection of nodes which are logically adjacent in which logical adjacency is maintained by pointers

## 1.2. Data structure Operations

The following list of operations applied on linear data structures

1. Add an element
2. Delete an element
3. Traverse
4. Sort the list of elements
5. Search for a data element

## 1.3. Review of Structures Unions and Pointers

### STURCTURES

Arrays allow to define type of variables that can hold several data items of the same kind. Similarly structure is another user defined data type available in C that allows to combine data items of different kinds. Structures are used to represent a record. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book –

- Title
- Author
- Subject
- Book ID

### Defining a Structure

To define a structure, you must use the **struct** statement. The struct statement defines a new data type, with more than one member. The format of the struct statement is as follows –

```
• struct [structure tag] {  
•  
•     member definition;  
•     member definition;  
•     ...
```

- member definition;
- } [one or more structure variables];

### Accessing Structure Members

To access any member of a structure, we use the **member access operator (.)**. The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use the keyword **struct** to define variables of structure type. The following example shows how to use a structure in a program –

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

struct Books {
    char title[50];
    char author[50];
    char subject[100];
    int book_id;
};

int main( ) {

    struct Books Book1;    /* Declare Book1 of type Book */
    struct Books Book2;    /* Declare Book2 of type Book */

    /* book 1 specification */
    strcpy( Book1.title, "C Programming");
    strcpy( Book1.author, "Nuha Ali");
    strcpy( Book1.subject, "C Programming Tutorial");
    Book1.book_id = 6495407;

    /* book 2 specification */
    strcpy( Book2.title, "Telecom Billing");
    strcpy( Book2.author, "Zara Ali");
```

```
strcpy( Book2.subject, "Telecom Billing Tutorial");
Book2.book_id = 6495700;

/* print Book1 info */
printf( "Book 1 title : %s\n", Book1.title);
printf( "Book 1 author : %s\n", Book1.author);
printf( "Book 1 subject : %s\n", Book1.subject);
printf( "Book 1 book_id : %d\n", Book1.book_id);

/* print Book2 info */
printf( "Book 2 title : %s\n", Book2.title);
printf( "Book 2 author : %s\n", Book2.author);
printf( "Book 2 subject : %s\n", Book2.subject);
printf( "Book 2 book_id : %d\n", Book2.book_id);

return 0;
}
```

## UNIONS

A **union** is a special data type available in C that allows to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple-purpose.

### Defining a Union

To define a union, you must use the **union** statement in the same way as you did while defining a structure. The union statement defines a new data type with more than one member for your program. The format of the union statement is as follows –

```
union [union tag] {
    member definition;
    member definition;
    ...
    member definition;
```

```
} [one or more union variables];
```

The union tag is optional and each member definition is a normal variable definition, such as `int i;` or `float f;` or any other valid variable definition. At the end of the union's definition, before the final semicolon, you can specify one or more union variables but it is optional. Here is the way you would define a union type named `Data` having three members `i`, `f`, and `str` –

```
union Data {  
    int i;  
    float f;  
    char str[20];  
} data;
```

Now, a variable of `Data` type can store an integer, a floating-point number, or a string of characters. It means a single variable, i.e., same memory location, can be used to store multiple types of data. You can use any built-in or user defined data types inside a union based on your requirement.

The memory occupied by a union will be large enough to hold the largest member of the union. For example, in the above example, `Data` type will occupy 20 bytes of memory space because this is the maximum space which can be occupied by a character string. The following example displays the total memory size occupied by the above union –

```
#include <stdio.h>  
#include <string.h>  
  
union Data {  
    int i;  
    float f;  
    char str[20];  
};  
  
int main( ) {  
  
    union Data data;
```

```
printf( "Memory size occupied by data : %d\n", sizeof(data));

return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
Memory size occupied by data : 20
```

### Accessing Union Members

To access any member of a union, we use the member access operator (.). The member access operator is coded as a period between the union variable name and the union member that we wish to access. You would use the keyword union to define variables of union type. The following example shows how to use unions in a program –

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

union Data {
    int i;
    float f;
    char str[20];
};

int main( ) {

    union Data data;

    data.i = 10;
    data.f = 220.5;
    strcpy( data.str, "C Programming");
```

```
printf( "data.i : %d\n", data.i);  
printf( "data.f : %f\n", data.f);  
printf( "data.str : %s\n", data.str);  
  
return 0;
```

## POINTERS

Pointers in C are easy and fun to learn. Some C programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers. So it becomes necessary to learn pointers to become a perfect C programmer. Let's start learning them in simple and easy steps.

As you know, every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory. Consider the following example, which prints the address of the variables defined –

```
#include <stdio.h>  
  
int main () {  
  
    int var1;  
    char var2[10];  
  
    printf("Address of var1 variable: %x\n", &var1 );  
    printf("Address of var2 variable: %x\n", &var2 );  
  
    return 0;  
}
```

When the above code is compiled and executed, it produces the following result –

```
Address of var1 variable: bff5a400  
Address of var2 variable: bff5a3f6
```

### What are Pointers?



A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is –

```
type *var-name;
```

Here, type is the pointer's base type; it must be a valid C data type and var-name is the name of the pointer variable. The asterisk \* used to declare a pointer is the same asterisk used for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Take a look at some of the valid pointer declarations –

```
int *ip; /* pointer to an integer */
double *dp; /* pointer to a double */
float *fp; /* pointer to a float */
char *ch /* pointer to a character */
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

### How to Use Pointers?

There are a few important operations, which we will do with the help of pointers very frequently. (a) We define a pointer variable, (b) assign the address of a variable to a pointer and (c) finally access the value at the address available in the pointer variable. This is done by using unary operator \* that returns the value of the variable located at the address specified by its operand. The following example makes use of these operations

```
#include <stdio.h>

int main () {

    int var = 20; /* actual variable declaration */
    int *ip;      /* pointer variable declaration */

    ip = &var; /* store address of var in pointer variable*/
```

```
printf("Address of var variable: %x\n", &var );

/* address stored in pointer variable */
printf("Address stored in ip variable: %x\n", ip );

/* access the value using the pointer */
printf("Value of *ip variable: %d\n", *ip );

return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

#### 1.4. Self Referential Structures

A self referential structure is used to create data structures like linked lists, stacks, etc. Following is an example of this kind of structure:

```
struct struct_name
{
    datatype datatype_name;
    struct_name * pointer_name;
};
```

A self-referential structure is one of the data structures which refer to the pointer to (points) to another structure of the same type. For example, a linked list is supposed to be a self-referential data structure. The next node of a node is being pointed, which is of the same struct type. For example,

```
typedef struct listnode {
    void *data;
    struct listnode *next;
} linked_list;
```

In the above example, the listnode is a self-referential structure – because the \*next is of the type struct listnode.

## 2. Arrays

In C programming, one of the frequently arising problem is to handle similar types of data. For example: If the user want to store marks of 100 students. This can be done by creating 100 variable individually but, this process is rather tedious and impracticable. These type of problem can be handled in C programming using arrays. An array is a sequence of data item of homogeneous value(same type).

### Arrays are of two types:

One-dimensional arrays

Multidimensional arrays( will be discussed in next chapter )

Declaration of one-dimensional array

```
data_type array_name[array_size];
```

For example:

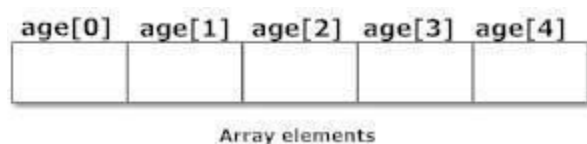
```
int age[5];
```

Here, the name of array is age. The size of array is 5,i.e., there are 5 items(elements) of array age. All element in an array are of the same type (int, in this case).

### Array elements

Size of array defines the number of elements in an array. Each element of array can be accessed and used by user according to the need of program. For example:

```
int age[5];
```



Note that, the first element is numbered 0 and so on.

Here, the size of array age is 5 times the size of int because there are 5 elements.

Suppose, the starting address of age[0] is 2120d and the size of int be 4 bytes. Then, the next address (address of a[1]) will be 2124d, address of a[2] will be 2128d and so on.

### Initialization of one-dimensional array:

Arrays can be initialized at declaration time in this source code as:

```
int age[5]={2,4,34,3,4};
```

It is not necessary to define the size of arrays during initialization.

```
int age[]={2,4,34,3,4};
```

In this case, the compiler determines the size of array by calculating the number of elements of an array.

age[0]	age[1]	age[2]	age[3]	age[4]
2	4	34	3	4

Initialization of one-dimensional array

### Accessing array elements

In C programming, arrays can be accessed and treated like variables in C.

For example:

```
scanf("%d",&age[2]);
```

```
/* statement to insert value in the third element of array age[].
```

```
*/ scanf("%d",&age[i]);
```

```
/* Statement to insert value in (i+1)th element of array age[]. */
```

```
/* Because, the first element of array is age[0], second is age[1], ith is age[i-1] and (i+1)th is age[i].
```

```
*/ printf("%d",age[0]);
```

```
/* statement to print first element of an array. */
```

```
printf("%d",age[i]);
```

```
/* statement to print (i+1)th element of an array. */
```

Example of array in C programming

```
/* C program to find the sum marks of n students using arrays */
```

```
#include <stdio.h>
```

```
int main(){
```

```
    int marks[10],i,n,sum=0;
```

```
    printf("Enter number of students: ");
```

```
scanf("%d",&n);

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

    printf("Enter marks of student%d: ",i+1);

    scanf("%d",&marks[i]);

    sum+=marks[i];

}

printf("Sum= %d",sum);

return 0;

}
```

### Output

Enter number of students: 3

Enter marks of student1: 12

Enter marks of student2: 31

Enter marks of student3: 2

sum=45

## 2.1 Operations

### Insertion Operation

Insert operation is to insert one or more data elements into an array. Based on the requirement, new element can be added at the beginning, end or any given index of array.

Here, we see a practical implementation of insertion operation, where we add data at the end of the array –

### Algorithm

Let **Array** is a linear unordered array of **MAX** elements.

### Example

### Result

Let LA is a Linear Array (unordered) with N elements and K is a positive integer such that  $K \leq N$ . Below is the algorithm where ITEM is inserted into the  $K^{\text{th}}$  position of LA –

1. Start
2. Set  $J=N$
3. Set  $N = N+1$
4. Repeat steps 5 and 6 while  $J \geq K$
5. Set  $LA[J+1] = LA[J]$
6. Set  $J = J-1$
7. Set  $LA[K] = \text{ITEM}$
8. Stop

#### Example

Below is the implementation of the above algorithm –

```
#include <stdio.h>

main() {
    int LA[] = {1,3,5,7,8};
    int item = 10, k = 3, n = 5;
    int i = 0, j = n;

    printf("The original array elements are :\n");

    for(i = 0; i<n; i++) {
        printf("LA[%d] = %d \n", i, LA[i]);
    }

    n = n + 1;

    while( j >= k){
        LA[j+1] = LA[j];
        j = j - 1;
    }

    LA[k] = item;
```

```
printf("The array elements after insertion :\n");

for(i = 0; i<n; i++) {
    printf("LA[%d] = %d \n", i, LA[i]);
}
}
```

When compile and execute, above program produces the following result –

The original array elements are :

LA[0]=1  
LA[1]=3  
LA[2]=5  
LA[3]=7  
LA[4]=8

The array elements after insertion :

LA[0]=1  
LA[1]=3  
LA[2]=5  
LA[3]=10  
LA[4]=7  
LA[5]=8

For other variations of array insertion operation click [here](#)

### Deletion Operation

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

#### Algorithm

Consider LA is a linear array with N elements and K is a positive integer such that  $K \leq N$ . Below is the algorithm to delete an element available at the  $K^{\text{th}}$  position of LA.

1. Start
2. Set  $J=K$
3. Repeat steps 4 and 5 while  $J < N$
4. Set  $LA[J-1] = LA[J]$
5. Set  $J = J+1$
6. Set  $N = N-1$

## 7. Stop

### Example

Below is the implementation of the above algorithm –

```
#include <stdio.h>

main() {
    int LA[] = {1,3,5,7,8};
    int k = 3, n = 5;
    int i, j;

    printf("The original array elements are :\n");

    for(i = 0; i<n; i++) {
        printf("LA[%d] = %d \n", i, LA[i]);
    }

    j = k;

    while(j < n){
        LA[j-1] = LA[j];
        j = j + 1;
    }

    n = n -1;

    printf("The array elements after deletion :\n");

    for(i = 0; i<n; i++) {
        printf("LA[%d] = %d \n", i, LA[i]);
    }
}
```



When compile and execute, above program produces the following result –

The original array elements are :

LA[0]=1

LA[1]=3

LA[2]=5

LA[3]=7

LA[4]=8

The array elements after deletion :

LA[0]=1

LA[1]=3

LA[2]=7

LA[3]=8

### Search Operation

You can perform a search for array element based on its value or its index.

#### Algorithm

Consider LA is a linear array with N elements and K is a positive integer such that  $K \leq N$ . Below is the algorithm to find an element with a value of ITEM using sequential search.

1. Start
2. Set J=0
3. Repeat steps 4 and 5 while  $J < N$
4. IF LA[J] is equal ITEM THEN GOTO STEP 6
5. Set J = J +1
6. PRINT J, ITEM
7. Stop

#### Example

Below is the implementation of the above algorithm –

```
#include <stdio.h>

main() {
    int LA[] = {1,3,5,7,8};
    int item = 5, n = 5;
    int i = 0, j = 0;
```

```
printf("The original array elements are :\n");

for(i = 0; i<n; i++) {
    printf("LA[%d] = %d \n", i, LA[i]);
}

while( j < n){

    if( LA[j] == item ){
        break;
    }

    j = j + 1;
}

printf("Found element %d at position %d\n", item, j+1);
}
```

When compile and execute, above program produces the following result –

```
The original array elements are :
LA[0]=1
LA[1]=3
LA[2]=5
LA[3]=7
LA[4]=8
Found element 5 at position 3
```

### Update Operation

Update operation refers to updating an existing element from the array at a given index.

#### Algorithm

Consider LA is a linear array with N elements and K is a positive integer such that  $K \leq N$ . Below is the algorithm to update an element available at the  $K^{\text{th}}$  position of LA.

1. Start
2. Set  $LA[K-1] = \text{ITEM}$
3. Stop

Example

Below is the implementation of the above algorithm –

```
#include <stdio.h>

main() {
    int LA[] = {1,3,5,7,8};
    int k = 3, n = 5, item = 10;
    int i, j;

    printf("The original array elements are :\n");

    for(i = 0; i<n; i++) {
        printf("LA[%d] = %d \n", i, LA[i]);
    }

    LA[k-1] = item;

    printf("The array elements after updation :\n");

    for(i = 0; i<n; i++) {
        printf("LA[%d] = %d \n", i, LA[i]);
    }
}
```

When compile and execute, above program produces the following result –

The original array elements are :  
LA[0]=1  
LA[1]=3  
LA[2]=5

```
LA[3]=7
LA[4]=8
The array elements after updation :
LA[0]=1
LA[1]=3
LA[2]=10
LA[3]=7
LA[4]=8
```

## 2.2 Multidimensional Arrays

C programming language allows multidimensional arrays. Here is the general form of a multidimensional array declaration –

```
type name[size1][size2]...[sizeN];
```

For example, the following declaration creates a three dimensional integer array –

```
int threedim[5][10][4];
```

### Two-dimensional Arrays

The simplest form of multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size  $[x][y]$ , you would write something as follows –

```
type arrayName [ x ][ y ];
```

Where type can be any valid C data type and arrayName will be a valid C identifier. A two-dimensional array can be considered as a table which will have x number of rows and y number of columns. A two-dimensional array a, which contains three rows and four columns can be shown as follows –

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in the array a is identified by an element name of the form  $a[i][j]$ , where 'a' is the name of the array, and 'i' and 'j' are the subscripts that uniquely identify each element in 'a'.

### Initializing Two-Dimensional Arrays

Multidimensional arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row has 4 columns.

```
int a[3][4] = {  
    {0, 1, 2, 3} ,    /*initializers for row indexed by 0 */  
    {4, 5, 6, 7} ,    /*initializers for row indexed by 1 */  
    {8, 9, 10, 11} /* initializers for row indexed by 2 */  
};
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to the previous example –

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

### Accessing Two-Dimensional Array Elements

An element in a two-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example –

```
int val = a[2][3];
```

The above statement will take the 4th element from the 3rd row of the array. You can verify it in the above figure. Let us check the following program where we have used a nested loop to handle a two-dimensional array –

```
#include <stdio.h>  
  
int main () {  
  
    /* an array with 5 rows and 2 columns*/  
    int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};  
    int i, j;  
  
    /* output each array element's value */  
    for ( i = 0; i < 5; i++ ) {
```

```
for ( j = 0; j < 2; j++ ) {  
    printf("a[%d][%d] = %d\n", i,j, a[i][j] );  
}  
}  
  
return 0;  
}
```

When the above code is compiled and executed, it produces the following result –

```
a[0][0]: 0  
a[0][1]: 0  
a[1][0]: 1  
a[1][1]: 2  
a[2][0]: 2  
a[2][1]: 4  
a[3][0]: 3  
a[3][1]: 6  
a[4][0]: 4  
a[4][1]: 8
```

## 2.3 Applications of Arrays

Arrays are used to implement mathematical vectors and matrices, as well as other kinds of rectangular tables. Many databases, small and large, consist of (or include) one-dimensional arrays whose elements are records.

Arrays are used to implement other data structures, such as heaps, hash tables, deques, queues, stacks, strings, and VLists.

One or more large arrays are sometimes used to emulate in-program dynamic memory allocation, particularly memory pool allocation. Historically, this has sometimes been the only way to allocate "dynamic memory" portably.

Arrays can be used to determine partial or complete control flow in programs, as a compact alternative to (otherwise repetitive) multiple IF statements. They are known in this context as control tables and are used in conjunction with a purpose built interpreter whose control flow is altered according to values contained in the

array. The array may contain subroutine pointers (or relative subroutine numbers that can be acted upon by SWITCH statements) that direct the path of the execution.

### 3. Strings

Strings are actually one-dimensional array of characters terminated by a nullcharacter '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a null.

The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

If you follow the rule of array initialization then you can write the above statement as follows –

```
char greeting[] = "Hello";
```

Following is the memory presentation of the above defined string in C/C++ –

Index	0	1	2	3	4	5
Variable	H	e	l	l	o	\0
Address	0x23451	0x23452	0x23453	0x23454	0x23455	0x23456

Actually, you do not place the *null* character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print the above mentioned string –

```
#include <stdio.h>
```

```
int main () {
```

```
    char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

```
printf("Greeting message: %s\n", greeting );  
return 0;  
}
```

When the above code is compiled and executed, it produces the following result –

Greeting message: Hello

C supports a wide range of functions that manipulate null-terminated strings –

S.N.	Function & Purpose
1	<b>strcpy(s1, s2);</b>  Copies string s2 into string s1.
2	<b>strcat(s1, s2);</b>  Concatenates string s2 onto the end of string s1.
3	<b>strlen(s1);</b>  Returns the length of string s1.
4	<b>strcmp(s1, s2);</b>  Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	<b>strchr(s1, ch);</b>  Returns a pointer to the first occurrence of character ch in string s1.
6	<b>strstr(s1, s2);</b>  Returns a pointer to the first occurrence of string s2 in string s1.

The following example uses some of the above-mentioned functions –



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

int main () {

    char str1[12] = "Hello";
    char str2[12] = "World";
    char str3[12];
    int len ;

    /* copy str1 into str3 */
    strcpy(str3, str1);
    printf("strcpy( str3, str1) : %s\n", str3 );

    /* concatenates str1 and str2 */
    strcat( str1, str2);
    printf("strcat( str1, str2):  %s\n", str1 );

    /* total length of str1 after concatenation */
    len = strlen(str1);
    printf("strlen(str1) : %d\n", len );

    return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
strcpy( str3, str1) : Hello
strcat( str1, str2): HelloWorld
strlen(str1) : 10
```

### 3.1. String manipulation Applications

Strings are often needed to be manipulated by programmer according to the need of a problem. All string manipulation can be done manually by the programmer but, this makes programming complex and large. To solve this, the C supports a large number of string handling functions.

There are numerous functions defined in "string.h" header file. Few commonly used string handling functions are discussed below:

Function	Work of Function
strlen()	Calculates the length of string
strcpy()	Copies a string to another string
strcat()	Concatenates(joins) two strings
strcmp()	Compares two string
strlwr()	Converts string to lowercase
strupr()	Converts string to uppercase

Strings handling functions are defined under "string.h" header file, i.e, you have to include the code below to run string handling functions.

### 4. Dynamic Memory Management Functions

The exact size of array is unknown until the compile time, i.e., time when a compiler compiles code written in a programming language into an executable form. The size of array you have declared initially can be sometimes insufficient and sometimes more than required. Dynamic memory allocation allows a program to obtain more memory space, while running or to release space when no space is required.

Although, C language inherently does not have any technique to allocate memory dynamically, there are 4 library functions under "stdlib.h" for dynamic memory allocation.

Function	Use of Function
----------	-----------------

Function	Use of Function
malloc()	Allocates requested size of bytes and returns a pointer first byte of allocated space
calloc()	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
free()	deallocate the previously allocated space
realloc()	Change the size of previously allocated space

**malloc()**

The name malloc stands for "memory allocation". The function malloc() reserves a block of memory of specified size and return a pointer of type void which can be casted into pointer of any form.

**Syntax of malloc()**

```
ptr=(cast-type*)malloc(byte-size)
```

Here, ptr is pointer of cast-type. The malloc() function returns a pointer to an area of memory with size of byte size. If the space is insufficient, allocation fails and returns NULL pointer.

```
ptr=(int*)malloc(100*sizeof(int));
```

This statement will allocate either 200 or 400 according to size of int 2 or 4 bytes respectively and the pointer points to the address of first byte of memory.

**calloc()**

The name calloc stands for "contiguous allocation". The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

**Syntax of calloc()**

```
ptr=(cast-type*)calloc(n,element-size);
```

This statement will allocate contiguous space in memory for an array of n elements. For example:

```
ptr=(float*)calloc(25,sizeof(float));
```

This statement allocates contiguous space in memory for an array of 25 elements each of size of float, i.e, 4 bytes.

### **free()**

Dynamically allocated memory with either calloc() or malloc() does not get return on its own. The programmer must use free() explicitly to release space.

syntax of free()

```
free(ptr);
```

This statement cause the space in memory pointer by ptr to be deallocated.

**Module -2****Linear Data Structures and their Sequential Storage Representation****Teaching Hours: 10**

<b>CONTENTS</b>	<b>Pg no</b>
<b>1. Stack</b>	<b>30</b>
<b>1.1 Operations and Applications</b>	<b>30</b>
<b>1.2 Polish and reverse polish expressions</b>	<b>32</b>
<b>1.3 Infix to postfix conversion</b>	<b>32</b>
<b>1.4 Evaluation of postfix expression</b>	<b>36</b>
<b>1.5 Infix to prefix</b>	<b>39</b>
<b>1.6 Postfix to infix conversion</b>	<b>39</b>
<b>2. Recursion</b>	<b>39</b>
<b>2.1 Factorial</b>	<b>39</b>
<b>2.2 GCD</b>	<b>41</b>
<b>2.3 Fibonacci Sequence</b>	<b>42</b>
<b>2.4 Tower of Hanoi</b>	<b>42</b>
<b>2.5 Binomial Co-efficient(<math>nCr</math>)</b>	<b>43</b>
<b>2.6 Ackerman's Recursive function</b>	<b>44</b>
<b>3. Queue</b>	<b>45</b>
<b>3.1 Operations</b>	<b>45</b>
<b>3.2 Queue Variants</b>	<b>45</b>
<b>3.3 Applications of Queues</b>	<b>49</b>

### 1. *Stack*

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



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

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

#### 1.1 Operations and Applications

Basic Operations Performed on Stack :

1. Create
2. Push
3. Pop
4. Empty

A. Creating Stack :

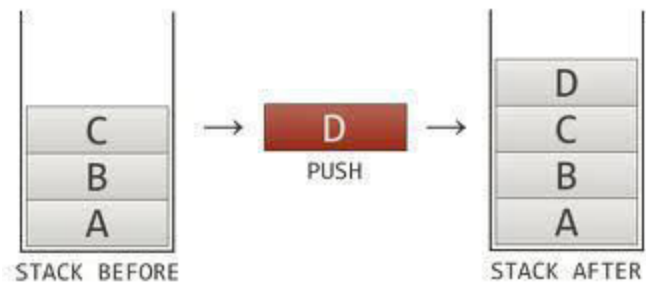
1. Stack can be created by declaring the structure with two members.
2. One Member can store the actual data in the form of array.
3. Another Member can store the position of the topmost element.

```
typedef struct stack {  
    int data[MAX];  
    int top;
```

```
}stack;
```

### B. Push Operation on Stack :

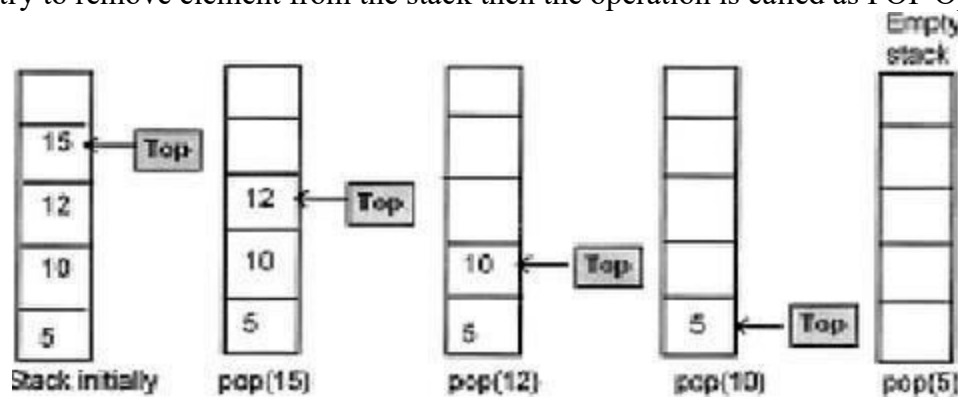
We have declared data array in the above declaration. Whenever we add any element in the `_data` array then it



will be called as “**Pushing Data on the Stack**”. Suppose `—top` is a pointer to the top element in a stack. After every push operation, the value of `—top` is incremented by one.

### C. Pop Operation on Stack :

Whenever we try to remove element from the stack then the operation is called as POP Operation on Stack.



Some basic Terms :

Concept	Definition
Stack Push	The procedure of inserting a new element to the top of the stack is known as <b>Push Operation</b>
Stack Overflow	Any attempt to insert a new element in already full stack is results into Stack Overflow.
Stack Pop	The procedure of removing element from the top of the stack is

Concept	Definition
	called <b>Pop Operation</b> .
Stack Underflow	Any attempt to delete an element from already empty stack results into Stack Underflow.

## Application

1. Expression Evolution
2. Expression conversion
  1. Infix to Postfix
  2. Infix to Prefix
  3. Postfix to Infix
  4. Prefix to Infix
3. Parsing
4. Simulation of recursion
5. Function call

### 1.2 Polish and reverse polish expressions

Reverse Polish notation (RPN) is a mathematical notation in which every operator follows all of its operands, in contrast to Polish notation(PN), which puts the operator before its operands. It is also known as postfix notation and does not need any parentheses as long as each operator has a fixed number of operands.

### 1.3 Infix to postfix conversion

infix to postfix conversion algorithm

There is an algorithm to convert an infix expression into a postfix expression. It uses a stack; but in this case, the stack is used to hold operators rather than numbers. The purpose of the stack is to reverse the order of the operators in the expression. It also serves as a storage structure, since no operator can be printed until both of its operands have appeared.

In this algorithm, all operands are printed (or sent to output) when they are read. There are more complicated rules to handle operators and parentheses.

Example:



1.  $A * B + C$  becomes  $A B * C +$

The order in which the operators appear is not reversed. When the '+' is read, it has lower precedence than the '\*', so the '\*' must be printed first.

We will show this in a table with three columns. The first will show the symbol currently being read. The second will show what is on the stack and the third will show the current contents of the postfix string. The stack will be written from left to right with the 'bottom' of the stack to the left.

	current symbol	operator stack	postfix string
1	A		A
2	*	*	A
3	B	*	A B
4	+	+	A B * {pop and print the '*' before pushing the '+'}
5	C	+	A B * C
6			A B * C +

The rule used in lines 1, 3 and 5 is to print an operand when it is read. The rule for line 2 is to push an operator onto the stack if it is empty. The rule for line 4 is if the operator on the top of the stack has higher precedence than the one being read, pop and print the one on top and then push the new operator on. The rule for line 6 is that when the end of the expression has been reached, pop the operators on the stack one at a time and print them.

2.  $A + B * C$  becomes  $A B C * +$

Here the order of the operators must be reversed. The stack is suitable for this, since operators will be popped off in the reverse order from that in which they were pushed.

	current symbol	operator stack	postfix string
1	A		A
2	+	+	A

3	B	+	A B
4	*	+ *	A B
5	C	+ *	A B C
6			A B C * +

In line 4, the '\*' sign is pushed onto the stack because it has higher precedence than the '+' sign which is already there. Then when the are both popped off in lines 6 and 7, their order will be reversed.

3.  $A * (B + C)$  becomes A B C + \*

A subexpression in parentheses must be done before the rest of the expression.

	current symbol	operator stack	postfix string
1	A		A
2	*	*	A
3	(	*(	A B
4	B	*(	A B
5	+	*( +	A B
6	C	*( +	A B C
7	)	*	A B C +
8			A B C + *

Since expressions in parentheses must be done first, everything on the stack is saved and the left parenthesis is pushed to provide a marker. When the next operator is read, the stack is treated as though it were empty and the new operator (here the '+' sign) is pushed on. Then when the right parenthesis is read, the stack is popped until the corresponding left parenthesis is found. Since postfix expressions have no parentheses, the parentheses are not printed.

4.  $A - B + C$  becomes A B - C +

When operators have the same precedence, we must consider association. Left to right association means that the operator on the stack must be done first, while right to left association means the reverse.

	current symbol	operator stack	postfix string
1	A		A
2	-	-	A
3	B	-	A B
4	+	+	A B -
5	C	+	A B - C
6			A B - C +

In line 4, the '-' will be popped and printed before the '+' is pushed onto the stack. Both operators have the same precedence level, so left to right association tells us to do the first one found before the second.

5.  $A * B ^ C + D$  becomes  $A B C ^ * D +$

Here both the exponentiation and the multiplication must be done before the addition.

	current symbol	operator stack	postfix string
1	A		A
2	*	*	A
3	B	*	A B
4	^	* ^	A B
5	C	* ^	A B C
6	+	+	A B C ^ *
7	D	+	A B C ^ * D
8			A B C ^ * D +

When the '+' is encountered in line 6, it is first compared to the '^' on top of the stack. Since it has lower precedence, the '^' is popped and printed. But instead of pushing the '+' sign onto the stack now, we must compare it with the new top of the stack, the '\*'. Since the operator also has higher precedence than the '+', it also must be popped and printed. Now the stack is empty, so the '+' can be pushed onto the stack.

6.  $A * (B + C * D) + E$  becomes  $A B C D * + * E +$

	current symbol	operator stack	postfix string
1	A		A
2	*	*	A
3	(	* (	A
4	B	* (	A B
5	+	* ( +	A B
6	C	* ( +	A B C
7	*	* ( + *	A B C
8	D	* ( + *	A B C D
9	)	*	A B C D * +
10	+	+	A B C D * + *
11	E	+	A B C D * + * E
12			A B C D * + * E +

### 1.4 Evaluation of postfix expression

In normal algebra we use the infix notation like  $a+b*c$ . The corresponding postfix notation is  $abc*+$ . The algorithm for the conversion is as follows :

- Scan the Postfix string from left to right.
- Initialise an empty stack.
- If the scanned character is an operand, add it to the stack. If the scanned character is an operator, there will be atleast two operands in the stack.
  - If the scanned character is an Operator, then we store the top most element of the stack(topStack) in a variable temp. Pop the stack. Now evaluate  $\text{topStack}(\text{Operator})\text{temp}$ . Let the result of this operation be retVal. Pop the stack and Push retVal into the stack.

Repeat this step till all the characters are scanned.

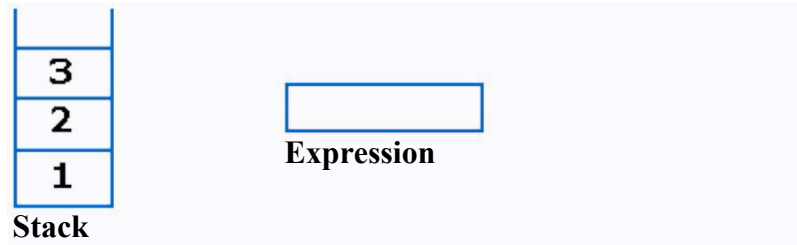
- After all characters are scanned, we will have only one element in the stack. Return topStack.

#### Example :

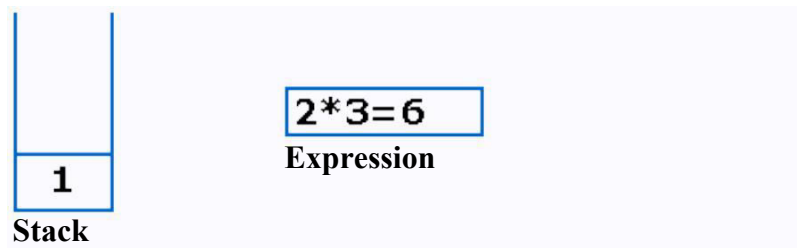
Let us see how the above algorithm will be implemented using an example.

Postfix String : 123\*+4-

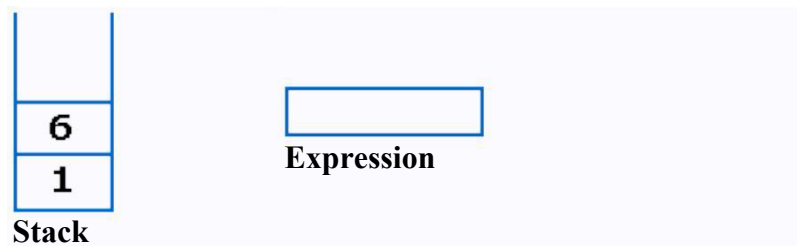
Initially the Stack is empty. Now, the first three characters scanned are 1,2 and 3, which are operands. Thus they will be pushed into the stack in that order.



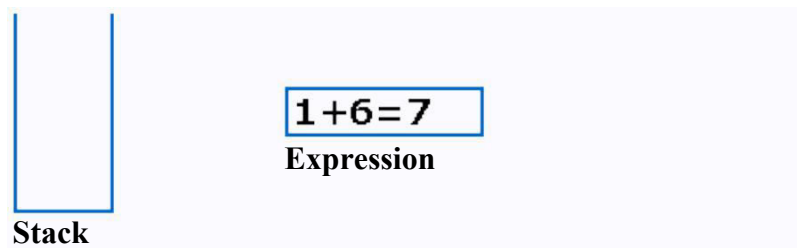
Next character scanned is "\*", which is an operator. Thus, we pop the top two elements from the stack and perform the "\*" operation with the two operands. The second operand will be the first element that is popped.



The value of the expression(2\*3) that has been evaluated(6) is pushed into the stack.



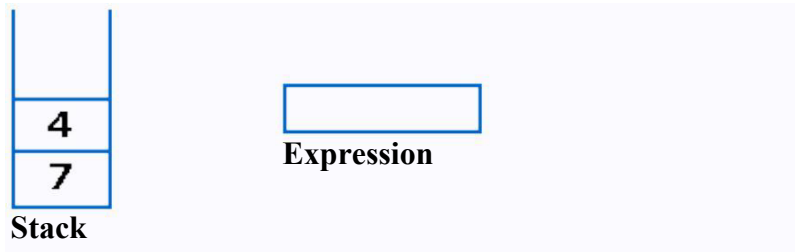
Next character scanned is "+", which is an operator. Thus, we pop the top two elements from the stack and perform the "+" operation with the two operands. The second operand will be the first element that is popped.



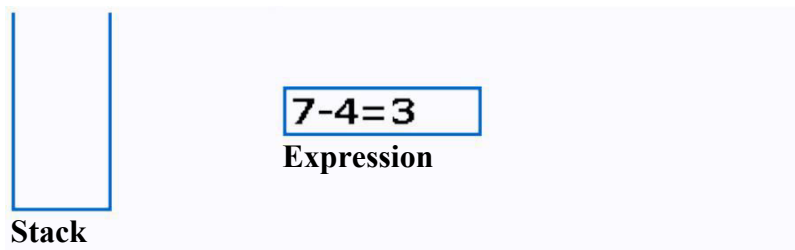
The value of the expression(1+6) that has been evaluated(7) is pushed into the stack.



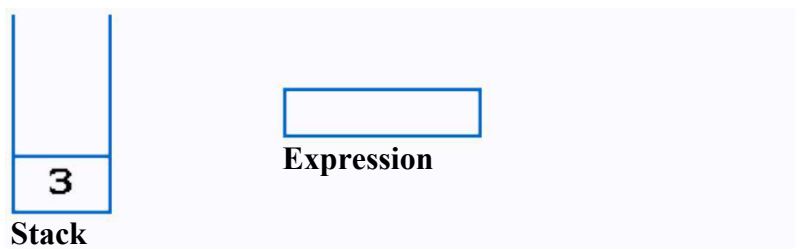
Next character scanned is "4", which is added to the stack.



Next character scanned is "-", which is an operator. Thus, we pop the top two elements from the stack and perform the "-" operation with the two operands. The second operand will be the first element that is popped.



The value of the expression(7-4) that has been evaluated(3) is pushed into the stack.



Now, since all the characters are scanned, the remaining element in the stack (there will be only one element in the stack) will be returned.

End result :

- Postfix String : 123\*+4-
- Result : 3

### 1.5 Infix to prefix

1. Step 1. Push —) onto STACK, and add —( to end of the A
2. Step 2. Scan A from right to left and repeat step 3 to 6 for each element of A until the STACK is empty
3. Step 3. If an operand is encountered add it to B
4. Step 4. If a right parenthesis is encountered push it onto STACK
5. Step 5. If an operator is encountered then:
  6. a. Repeatedly pop from STACK and add to B each operator (on the top of STACK) which has same
  7. or higher precedence than the operator.
  8. b. Add operator to STACK
9. Step 6. If left parenthesis is encountered then
  10. a. Repeatedly pop from the STACK and add to B (each operator on top of stack until a left parenthesis is encountered)
  11. b. Remove the left parenthesis
12. Step 7. Exit

### 1.6 Postfix to infix conversion

1. While there are input symbol left
2. Read the next symbol from input.
3. If the symbol is an operand
4. Otherwise,
  - the symbol is an operator.
5. If there are fewer than 2 values on the stack
  - Show Error /\* input not sufficient values in the expression \*/
6. Else
  - Pop the top 2 values from the stack.
  - Put the operator, with the values as arguments and form a string.
  - Encapsulate the resulted string with parenthesis.
  - Push the resulted string back to stack.
7. If there is only one value in the stack
8. If there are more values in the stack
  - Show Error /\* The user input has too many values \*/

## 2. Recursion

Recursion in computer science is a method where the solution to a problem depends on solutions to smaller instances of the same problem (as opposed to iteration). The approach can be applied to many types of problems, and recursion is one of the central ideas of computer science.

### 2.1 Factorial

The factorial of a positive number  $n$  is given by:

$$\text{factorial of } n (n!) = 1*2*3*4*...n$$

The factorial of a negative number doesn't exist. And, the factorial of 0 is 1.

You will learn to find the factorial of a number using recursion in this example. Visit this page to learn, how you can find the factorial of a number using loop .

Example: Factorial of a Number Using Recursion

```
#include <stdio.h>

long int multiplyNumbers(int n);

int main()
{
    int n;
    printf("Enter a positive integer: ");
    scanf("%d", &n);
    printf("Factorial of %d = %ld", n, multiplyNumbers(n)); return
    0;
}

long int multiplyNumbers(int n)
{
    if (n >= 1)
        return n*multiplyNumbers(n-1);
    else
        return 1;
}
```



## 2.2 GCD

```
#include <stdio.h>

int hcf(int n1, int n2);

int main()

{

    int n1, n2;

    printf("Enter two positive integers: ");

    scanf("%d %d", &n1, &n2);

    printf("G.C.D of %d and %d is %d.", n1, n2, hcf(n1,n2)); return

    0;

}

int hcf(int n1, int n2)

{

    if (n2!=0)

        return hcf(n2, n1%n2);

    else

        return n1;

}
```

### 2.3 Fibonacci Sequence

```
#include<stdio.h>

void printFibonacci(int);

int main(){

    int k,n;
    long int i=0,j=1,f;

    printf("Enter the range of the Fibonacci series: ");
    scanf("%d",&n);

    printf("Fibonacci Series: ");
    printf("%d %d ",0,1);
    printFibonacci(n);

    return 0;
}

void printFibonacci(int n){

    static long int first=0,second=1,sum;

    if(n>0){
        sum = first + second;
        first = second;
        second = sum;
        printf("%ld ",sum);
        printFibonacci(n-1);
    }

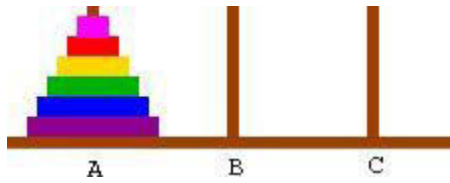
}
```

### 2.4 Tower of Hanoi

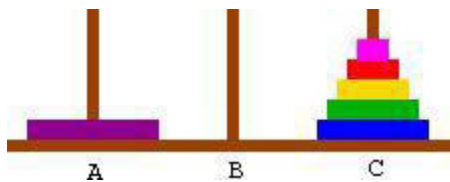
Using recursion often involves a key insight that makes everything simpler. Often the insight is determining what data exactly we are recursing on - we ask, what is the essential feature of the problem that should change as we call ourselves? In the case of isAJew, the feature is the person in question: At the top level, we are asking about a person; a level deeper, we ask about the person's mother; in the next level, the grandmother; and so on.

In our Towers of Hanoi solution, we recurse on the largest disk to be moved. That is, we will write a recursive function that takes as a parameter the disk that is the largest disk in the tower we want to move. Our function will also take three parameters indicating from which peg the tower should be moved (*source*), to which peg it should go (*dest*), and the other peg, which we can use temporarily to make this happen (*spare*).

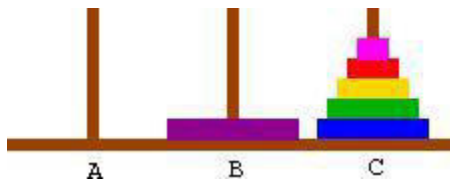
At the top level, we will want to move the entire tower, so we want to move disks 5 and smaller from peg A to peg B. We can break this into three basic steps.



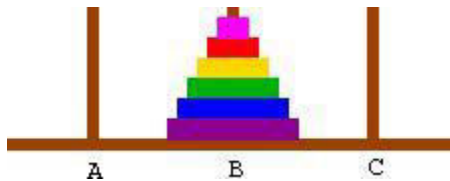
1. Move disks 4 and smaller from peg A (*source*) to peg C (*spare*), using peg B (*dest*) as a spare. How do we do this? By recursively using the same procedure. After finishing this, we'll have all the disks smaller than disk 4 on peg C. (Bear with me if this doesn't make sense for the moment - we'll do an example soon.)



2. Now, with all the smaller disks on the spare peg, we can move disk 5 from peg A (*source*) to peg B (*dest*).



3. Finally, we want disks 4 and smaller moved from peg C (*spare*) to peg B (*dest*). We do this recursively using the same procedure again. After we finish, we'll have disks 5 and smaller all on *dest*.



## 2.5 Binomial Co-efficient( $nCr$ )

The binomial coefficient  $\binom{n}{k}$  is the number of ways of picking  $k$  *unordered* outcomes from  $n$  possibilities, also known as a combination or combinatorial number. The symbols  ${}_nC_k$  and  $\binom{n}{k}$  are used to denote a binomial coefficient, and are sometimes read as " $n$  choose  $k$ ."

$\binom{n}{k}$  therefore gives the number of  $k$ -subsets possible out of a set of  $n$  distinct items. For example, The 2-subsets of  $\{1, 2, 3, 4\}$  are the six pairs  $\{1, 2\}$ ,  $\{1, 3\}$ ,  $\{1, 4\}$ ,  $\{2, 3\}$ ,  $\{2, 4\}$ , and  $\{3, 4\}$ , so  $\binom{4}{2} = 6$ . The number of lattice paths from the origin  $(0, 0)$  to a point  $(a, b)$  is the binomial coefficient  $\binom{a+b}{a}$  (Hilton and Pedersen 1991).

The value of the binomial coefficient for nonnegative  $n$  and  $k$  is given explicitly by

$${}_n C_k \equiv \binom{n}{k} \equiv \frac{n!}{(n-k)! k!},$$

## 2.6 Ackerman's Recursive function

The Ackermann function is the simplest example of a well-defined total function which is computable but not primitive recursive, providing a counterexample to the belief in the early 1900s that every computable function was also primitive recursive (Dötzel 1991). It grows faster than an exponential function, or even a multiple exponential function.

The Ackermann function  $A(x, y)$  is defined for integer  $x$  and  $y$  by

$$A(x, y) \equiv \begin{cases} y + 1 & \text{if } x = 0 \\ A(x - 1, 1) & \text{if } y = 0 \\ A(x - 1, A(x, y - 1)) & \text{otherwise.} \end{cases} \quad (1)$$

Special values for integer  $x$  include

$$A(0, y) = y + 1 \quad (2)$$

$$A(1, y) = y + 2 \quad (3)$$

$$A(2, y) = 2y + 3 \quad (4)$$

$$A(3, y) = 2^{y+3} - 3 \quad (5)$$

$$A(4, y) = \frac{2^{2^{\cdot^{\cdot^2}}}}{y+3} - 3. \quad (6)$$

Expressions of the latter form are sometimes called power towers.  $A(0, y)$  follows trivially from the definition.  $A(1, y)$  can be derived as follows:

$$A(1, y) = A(0, A(1, y - 1)) \quad (7)$$

$$= A(1, y - 1) + 1 \quad (8)$$

$$= A(0, A(1, y - 2)) + 1 \quad (9)$$

$$= A(1, y - 2) + 2 \quad (10)$$

$$= \dots \quad (11)$$

$$= A(1, 0) + y \quad (12)$$

$$= A(0, 1) + y = y + 2.$$

### 3. Queue

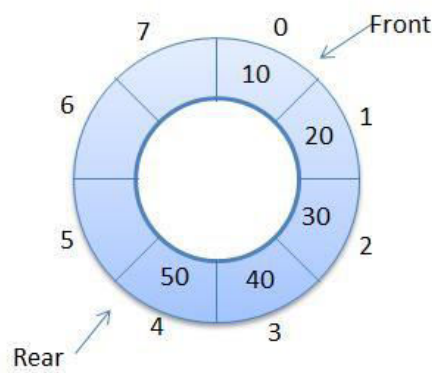
#### 3.1 Operations

a particular kind of abstract data type or collection in which the entities in the collection are kept in order and the principal (or only) operations on the collection are the addition of entities to the rear terminal position, known as *enqueue*, and removal of entities from the front terminal position, known as *dequeue*. This makes the queue a First-In-First-Out (FIFO) data structure. In a FIFO data structure, the first element added to the queue will be the first one to be removed. This is equivalent to the requirement that once a new element is added, all elements that were added before have to be removed before the new element can be removed. Often a *peek* or *front* operation is also entered, returning the value of the front element without dequeuing it. A queue is an example of a linear data structure, or more abstractly a sequential collection.

#### 3.2 Queue Variants

##### Circular Queue

A circular queue is an abstract data type that contains a collection of data which allows addition of data at the end of the queue and removal of data at the beginning of the queue. Circular queues have a fixed size. Circular queue follows FIFO principle. Queue items are added at the rear end and the items are deleted at front

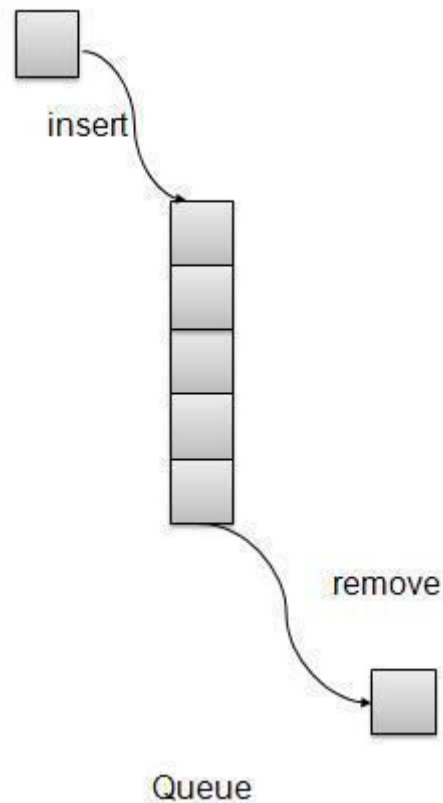


end of the circular queue.

Algorithm for Insertion in a circular queue

```
1.
2. Insert CircularQueue ( )
3.
4. 1. If (FRONT == 1 and REAR == N) or (FRONT == REAR + 1) Then
5.
6.
7.
8. 3. Else
9.
10. 4.If (REAR == 0) Then [Check if QUEUE is empty]
11.
12.         (a) Set FRONT = 1
13.
14.         (b) Set REAR = 1
15.
16. 5. Else If (REAR == N) Then [If REAR reaches end of QUEUE]
17.
18. 6.         Set REAR = 1
19.
20. 7. Else
21.
22. 8.         Set REAR = REAR + 1 [Increment REAR by 1]
23.
24. [End of Step 4 If]
25.
26. 9. Set QUEUE[REAR] = ITEM
27.
28. 10. Print: ITEM inserted
29.
30. [End of Step 1 If]
31.
32. 11. Exit
```

## Priority Queue

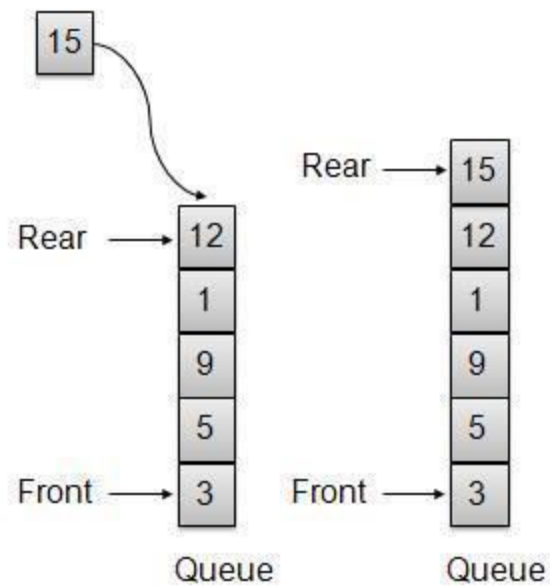


We're going to implement Queue using array in this article. There are few more operations supported by queue which are following.

- **Peek** – get the element at front of the queue.
- **isFull** – check if queue is full.
- **isEmpty** – check if queue is empty.

### Insert / Enqueue Operation

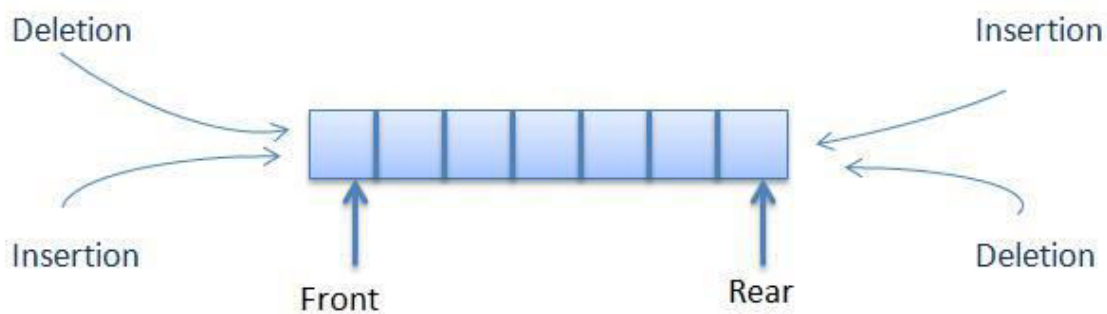
Whenever an element is inserted into queue, priority queue inserts the item according to its order. Here we're assuming that data with high value has low priority.



One item inserted at rear end

### Double Ended Queue

A double-ended queue is an abstract data type similar to an simple queue, it allows you to insert and delete from both sides means items can be added or deleted from the front or rear end.



Algorithm for Insertion at rear end

- 1.
2. Step -1: [Check for overflow]



```
3.
4.     if(rear==MAX)
5.
6.         Print("Queue is Overflow");
7.
8.     return;
9.
10. Step-2: [Insert element]
11.
12.     else
13.
14.         rear=rear+1;
15.
16.         q[rear]=no;
17.
18.         [Set rear and front pointer]
19.
20.         if rear=0
21.
22.             rear=1;
23.
24.         if front=0
25.
26.             front=1;
27.
28. Step-3: return
29.
```

### 3.3 Applications of Queues

Queue, as the name suggests is used whenever we need to have any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios :

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
2. In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.

**Module -3****Linear Data Structures and their Linked Storage Representation****Teaching Hours: 10**

<b>CONTENTS</b>	<b>Pg no</b>
<b><i>1. Linked List</i></b>	<b><i>51</i></b>
1.1. Definition	<b><i>51</i></b>
1.2. Representation	<b><i>51</i></b>
1.3. Operations	<b><i>51</i></b>
<b><i>2. Types:</i></b>	<b><i>52</i></b>
2.1. Singly Linked List	<b><i>52</i></b>
2.2. Doubly Linked list	<b><i>54</i></b>
2.3. Circular linked list	<b><i>54</i></b>
<b><i>3. Linked implementation</i></b>	<b><i>56</i></b>
3.1. Stack	<b><i>56</i></b>
3.2. Queue and its variants	<b><i>59</i></b>
<b><i>4. Applications of Linked lists</i></b>	<b><i>62</i></b>
4.1. Polynomial Manipulation,	<b><i>62</i></b>
4.2. Multiprecision arithmetic,	<b><i>71</i></b>
4.3. Symbol table organizations,	<b><i>75</i></b>
4.4. Sparse matrix representation with multilinked data structure.	<b><i>76</i></b>
<b><i>5. Programming Examples</i></b>	<b><i>76</i></b>
5.1. length of a list	<b><i>76</i></b>
5.2. Merging two lists	<b><i>76</i></b>
5.3. Removing duplicates	<b><i>76</i></b>
5.4. Reversing a list	<b><i>79</i></b>
5.5. Union and intersection of two lists	<b><i>80</i></b>

## 1. *Linked List*

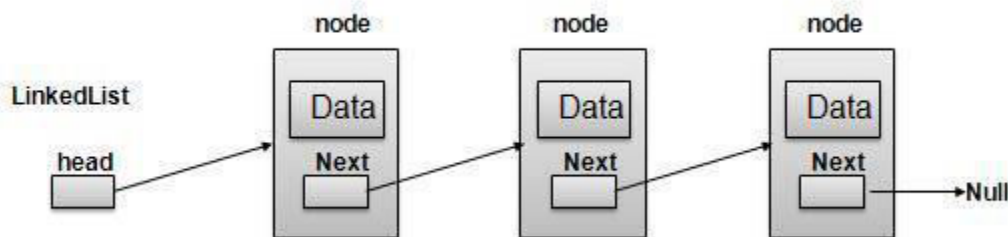
### 1.1. Definition

A linked-list is a sequence of data structures which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list the second most used data structure after array. Following are important terms to understand the concepts of Linked List.

- **Link** – Each Link of a linked list can store a data called an element.
- **Next** – Each Link of a linked list contain a link to next link called Next.
- **LinkedList** – A LinkedList contains the connection link to the first Link called First.

### 1.2. Representation



As per above shown illustration, following are the important points to be considered.

- LinkedList contains an link element called first.
- Each Link carries a data field(s) and a Link Field called next.
- Each Link is linked with its next link using its next link.
- Last Link carries a Link as null to mark the end of the list.

### 1.3. Operations

Following are the basic operations supported by a list.

- **Insertion** – add an element at the beginning of the list.
- **Deletion** – delete an element at the beginning of the list.
- **Display** – displaying complete list.
- **Search** – search an element using given key.

- **Delete** – delete an element using given key.

## 2. Types:

### 2.1. Singly Linked List

Linked list is one of the fundamental data structures, and can be used to implement other data structures. In a linked list there are different numbers of nodes. Each node consists of two fields. The first field holds the value or data and the second field holds the reference to the next node or null if the linked list is empty.

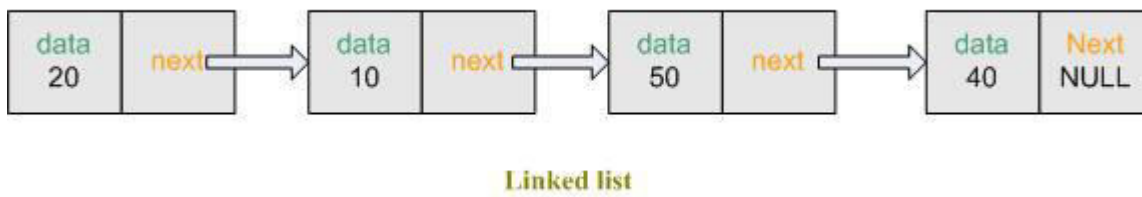


Figure: Linked list

Pseudocode:

```
LinkedList Node {  
    data // The value or data stored in the node  
    next // A reference to the next node, null for last node  
}
```

The singly-linked list is the easiest of the linked list, which has one link per node.

#### Pointer

To create linked list in C/C++ we must have a clear understanding about pointer. Now I will explain in brief what is pointer and how it works.

A pointer is a variable that contains the address of a variable. The question is why we need pointer? Or why it is so powerful? The answer is they have been part of the C/C++ language and so we have to use it. Using pointer we can pass argument to the functions. Generally we pass them by value as a copy. So we cannot change them. But if we pass argument using pointer, we can modify them. To understand about pointers, we must know how computer store variable and its value. Now, I will show it here in a very simple way.

Let us imagine that a computer memory is a long array and every array location has a distinct memory location.

```
int a = 50 // initialize variable a
```

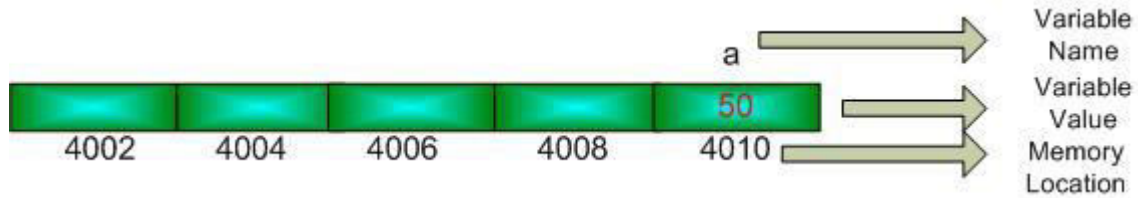


Figure: Variable value store inside an array

It is like a house which has an address and this house has only one room. So the full address is-

Name of the house: a

Name of the person/value who live here is: 50

House Number: 4010

If we want to change the person/value of this house, the conventional way is, type this code line

```
a = 100 // new initialization
```

But using pointer we can directly go to the memory location of 'a' and change the person/value of this house without disturbing `_a`. This is the main point about pointer.

Now the question is how we can use pointer. Type this code line:

```
int *b; // declare pointer b
```

We transfer the memory location of a to b .

```
b = &a; // the unary operator & gives the address of an object
```

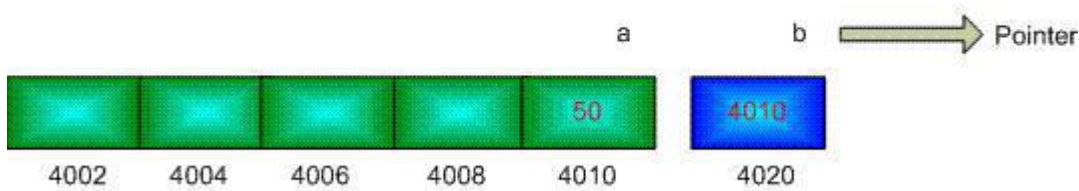


Figure: Integer pointer b store the address of the integer variable a

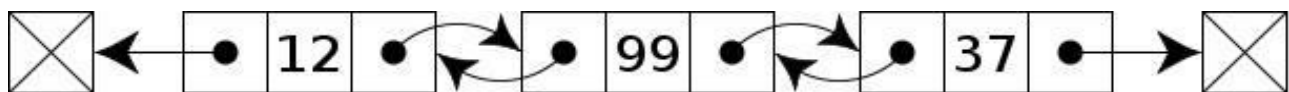
## 2.2. Doubly Linked list

A doubly linked list is a sequential data structure that consists of set of linked records called nodes.

In a doubly linked list, each node consists of:-

1. Two address fields, one pointing to next node, and other pointing to the previous pointer.
2. and one data part that is used to store the data of the node.

The following diagram shows the representation of doubly link-list:-

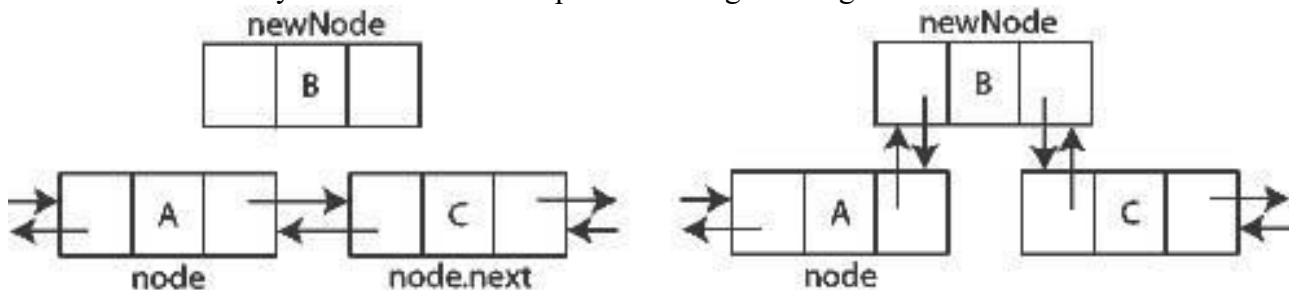


Here I have a complete C program for the implementation of doubly linked list.

I have performed the following operations on the doubly linked list:-

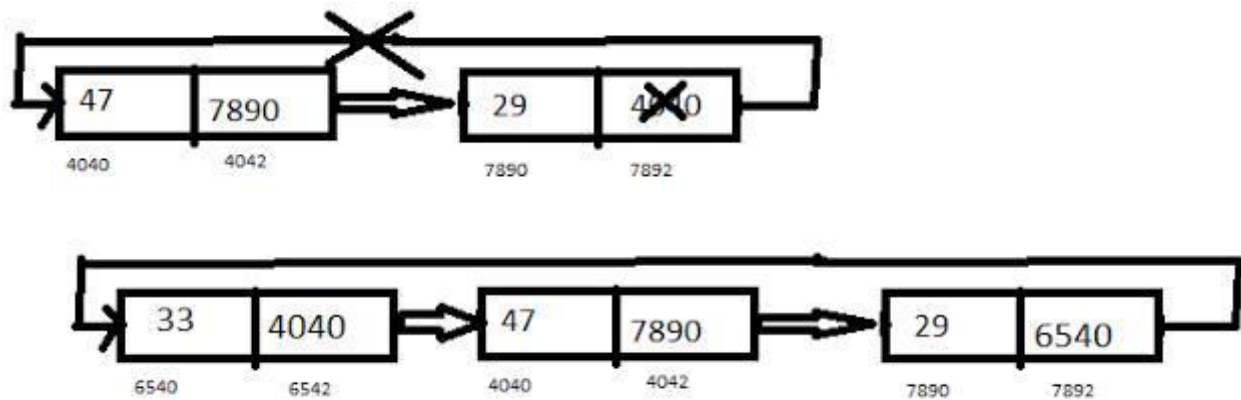
1. Insertion of node after a specific node
2. Insertion of node before a specific node
3. Forward printing
4. Reverse printing
5. Deletion of a specific node

Simple insertion in a doubly linked list can be explained through this figure:-



## 2.3. Circular linked list

let us consider the code in insertbeg().we check that is linked list empty or not by checking value of Start!=NULL. (Note: Click on image for better view)



if start=null then the new created node is assign to Start else consider the code

```
temp=start;
```

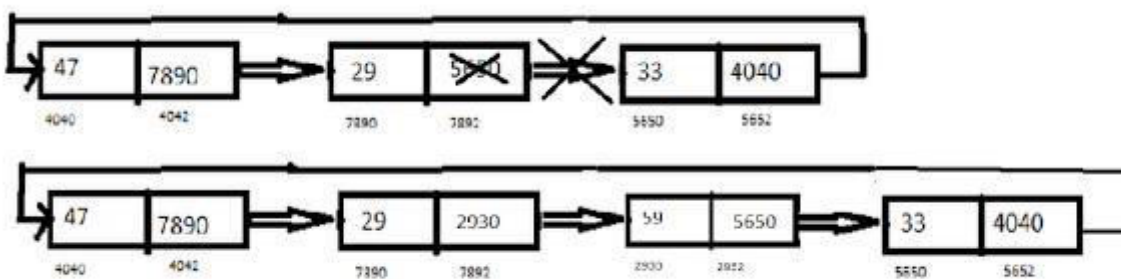
```
while(temp->next!=start)
```

```
    temp=temp->next;
```

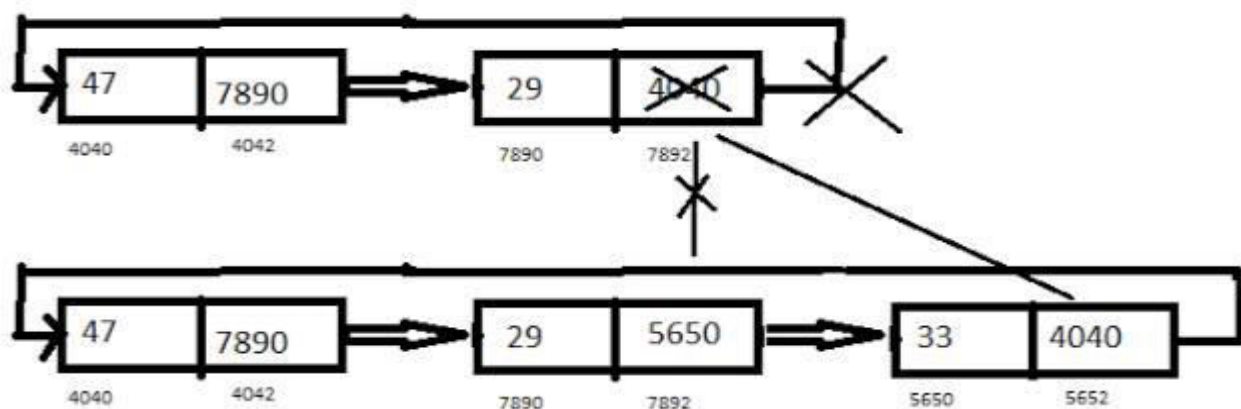
```
temp->next=nn;
```

```
nn->next=start; start=nn;
```

assume the above image, we want to add 33 at the begg. so the temp pointer is traversed to the end of list, inserting the address of new node in temp->next and inserting address pointed by start in the new nodes next, make the start pointer to point new node.

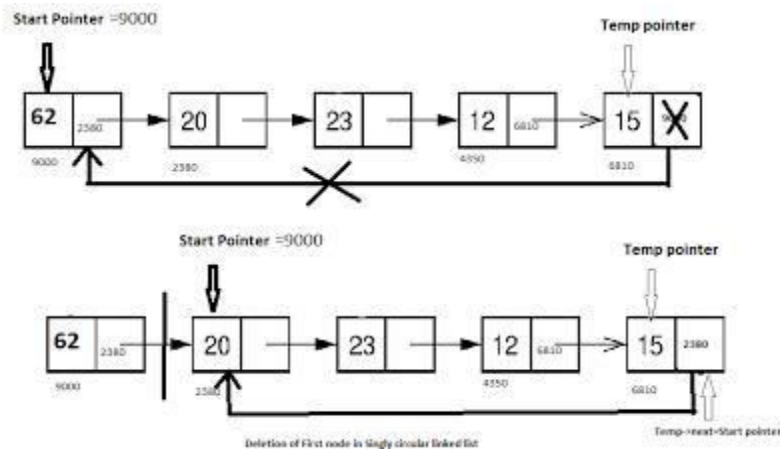


No Explanation is needed for inserting element at the mid, it remains the same as that of singly linked list

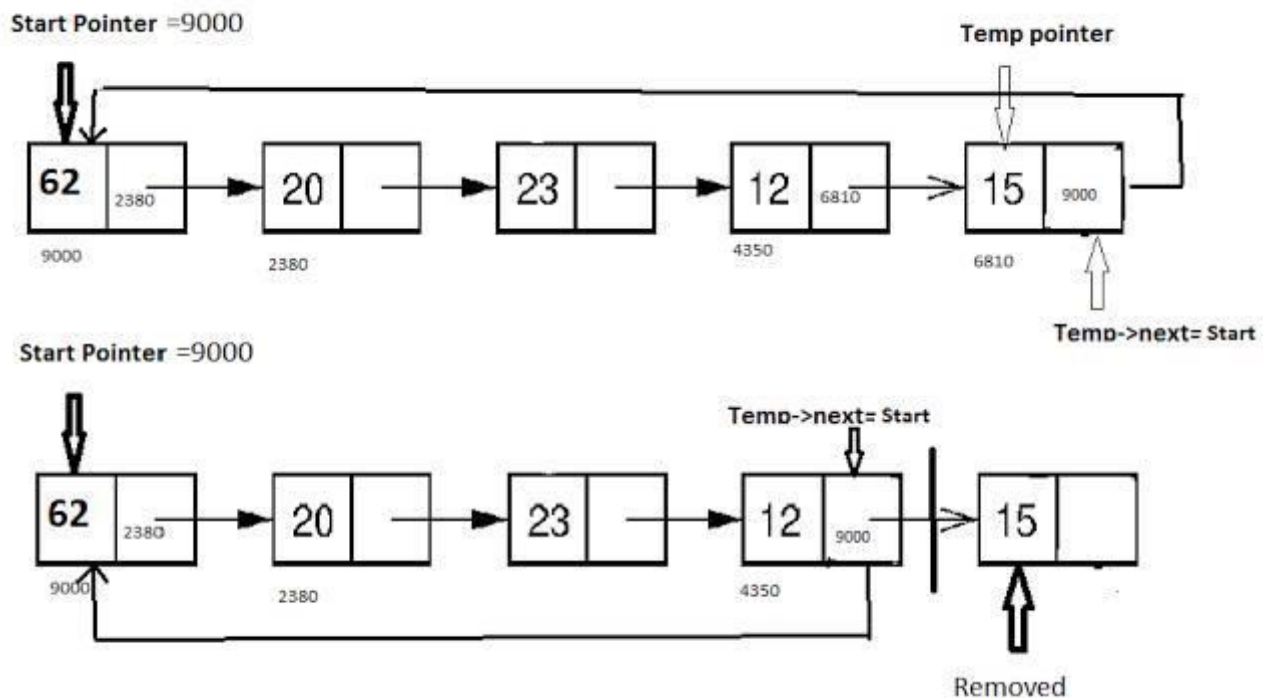


Inserting 33 at the end of list

Consider the case of **inserting element at the end of singly circular linked list**. Assume we wanted to add 33 at the end of list, we will traverse the temporary pointers till the temp's next address is not start. So at this point we have last node pointed by temp. we will insert temp->next address in new nodes next and new nodes address in temp->next.



**Deletion of First node.** For deletion of first node again we have to traverse the temp pointer till the end of list so that address of start can be removed from temp->next and address of start's next is inserted in temp->next. Start pointer is made to point the address contained by start->next. Deletion of middle node is same as that of singly linked list



### 3. Linked implementation

#### 3.1. Stack

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



```
struct Node
{
    int Data;
    struct Node *next;
}*top;

void popStack()
{
    struct Node *temp, *var=top;
    if(var==top)
    {
        top = top->next;
        free(var);
    }
    else
        printf("\nStack Empty");
}

void push(int value)
{
    struct Node *temp;
    temp=(struct Node *)malloc(sizeof(struct Node));
    temp->Data=value;
    if (top == NULL)
    {
        top=temp;
        top->next=NULL;
    }
    else
    {
        temp->next=top;
        top=temp;
    }
}

void display()
{
    struct Node *var=top;
    if(var!=NULL)
    {
        printf("\nElements are as:\n");
        while(var!=NULL)
        {
            printf("\t%d\n",var->Data);
            var=var->next;
        }
    }
}
```

```
    printf("\n");
}
else
    printf("\nStack is Empty");
}

int main(int argc, char *argv[])
{
    int i=0;
    top=NULL;
    printf(" \n1. Push to stack");
    printf(" \n2. Pop from Stack");
    printf(" \n3. Display data of Stack");
    printf(" \n4. Exit\n");
    while(1)
    {
        printf(" \nChoose Option: ");
        scanf("%d",&i);
        switch(i)
        {
            case 1:
            {
                int value;
                printf("\nEnter a valueber to push into Stack: ");
                scanf("%d",&value);
                push(value);
                display();
                break;
            }
            case 2:
            {
                popStack();
                display();
                break;
            }
            case 3:
            {
                display();
                break;
            }
            case 4:
            {
                struct Node *temp;
                while(top!=NULL)
                {
                    temp = top->next;
                    free(top);
                }
            }
        }
    }
}
```

```
        top=temp;
    }
    exit(0);
}
default:
{
    printf("\nwrong choice for operation");
}
}
}
```

### 3.2. Queues

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

```
struct node
{
    int info;
    struct node *ptr;
}*front,*rear,*temp,*front1;
```

```
int fruntelement();
void enq(int data);
void deq();
void empty();
void display();
void create();
void queuesize();
```

```
int count = 0;
```

```
void main()
{
    int no, ch, e;

    printf("\n 1 - Enque");
    printf("\n 2 - Deque");
    printf("\n 3 - Front element");
    printf("\n 4 - Empty");
    printf("\n 5 - Exit");
    printf("\n 6 - Display");
    printf("\n 7 - Queue size");
    create();
    while (1)
    {
        printf("\n Enter choice : ");
        scanf("%d", &ch);
```

```
switch (ch)
{
case 1:
    printf("Enter data : ");
    scanf("%d", &no);
    enq(no);
    break;
case 2:
    deq();
    break;
case 3:
    e = frontelement();
    if (e != 0)
        printf("Front element : %d", e);
    else
        printf("\n No front element in Queue as queue is
empty"); break;
case 4:
    empty();
    break;
case 5:
    exit(0);
case 6:
    display();
    break;
case 7:
    queuesize();
    break;
default:
    printf("Wrong choice, Please enter correct choice ");
    break;
}
}
}

/* Create an empty queue */
void create()
{
    front = rear = NULL;
}

/* Returns queue size */
void queuesize()
{
    printf("\n Queue size : %d", count);
}
```

```
/* Enqueing the queue */
void enq(int data)
{
    if (rear == NULL)
    {
        rear = (struct node *)malloc(1*sizeof(struct node));
        rear->ptr = NULL;
        rear->info = data;
        front = rear;
    }
    else
    {
        temp=(struct node *)malloc(1*sizeof(struct node));
        rear->ptr = temp;
        temp->info = data;
        temp->ptr = NULL;

        rear = temp;
    }
    count++;
}
```

```
/* Displaying the queue elements */
void display()
{
    front1 = front;

    if ((front1 == NULL) && (rear == NULL))
    {
        printf("Queue is empty");
        return;
    }
    while (front1 != rear)
    {
        printf("%d ", front1->info);
        front1 = front1->ptr;
    }
    if (front1 == rear)
        printf("%d", front1->info);
}
```

```
/* Dequeing the queue */
void deq()
{
    front1 = front;

    if (front1 == NULL)
```

```

{
    printf("\n Error: Trying to display elements from empty
    queue"); return;
}
else
    if (front1->ptr != NULL)
    {
        front1 = front1->ptr;
        printf("\n Dequed value : %d", front->info);
        free(front);
        front = front1;
    }
    else
    {
        printf("\n Dequed value : %d", front->info);
        free(front);
        front = NULL;
        rear = NULL;
    }
    count--;
}

```

/\* Returns the front element of queue \*/

```

int frontelement()
{
    if ((front != NULL) && (rear != NULL))
        return(front->info);
    else
        return 0;
}

```

/\* Display if queue is empty or not \*/

```

void empty()
{
    if ((front == NULL) && (rear == NULL))
        printf("\n Queue empty");
    else
        printf("Queue not empty");
}

```

#### 4. *Applications of Linked lists*

##### 4.1. **Polynomial Manipulation**

A polynomial is an expression that contains more than two terms. A term is made up of coefficient and exponent. An example of polynomial is

$$P(x) = 4x^3 + 6x^2 + 7x + 9$$

A polynomial thus may be represented using arrays or linked lists. Array representation assumes that the exponents of the given expression are arranged from 0 to the highest value (degree), which is represented by the subscript of the array beginning with 0. The coefficients of the respective exponent are placed at an appropriate index in the array. The array representation for the above polynomial expression is given below:

arr	9	7	6	4	(coefficients)
	0	1	2	3	(exponents)

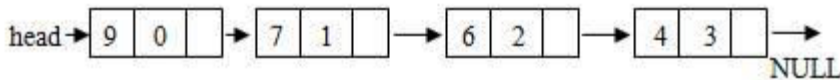
A polynomial may also be represented using a linked list. A structure may be defined such that it contains two parts- one is the coefficient and second is the corresponding exponent. The structure definition may be given as shown below:

```

struct polynomial
{
    int coefficient;
    int exponent;
    struct polynomial *next;
};

```

Thus the above polynomial may be represented using linked list as shown below:



### Addition of two Polynomials:

For adding two polynomials using arrays is straightforward method, since both the arrays may be added up element wise beginning from 0 to n-1, resulting in addition of two polynomials. Addition of two polynomials using linked list requires comparing the exponents, and wherever the exponents are found to be same, the coefficients are added up. For terms with different exponents, the complete term is simply added to the result thereby making it a part of addition result. The complete program to add two polynomials is given in subsequent section.

### Multiplication of two Polynomials:

Multiplication of two polynomials however requires manipulation of each node such that the exponents are added up and the coefficients are multiplied. After each term of first polynomial is operated upon with each term of the second polynomial, then the result has to be added up by comparing the exponents and adding the coefficients for similar exponents and including terms as such with dissimilar exponents in the result. The \_C\_ program for polynomial manipulation is given below:

### Program for Polynomial representation, addition and multiplication

```
/*Polynomial- Representation, Addition, Multiplication*/
```

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

struct poly
{
int coeff;
int exp;
struct poly *next;
}*head1=NULL,*head2=NULL,*head3=NULL,*head4=NULL,*temp,*ptr;

void create();
void makenode(int,int);
struct poly *insertend(struct poly *);
void display(struct poly *);
struct poly *addtwopoly(struct poly *,struct poly *,struct poly *);
struct poly *subtwopoly(struct poly *,struct poly *,struct poly *);
struct poly *multwopoly(struct poly *,struct poly *,struct poly
*); struct poly *dispose(struct poly *); int search(struct poly
*,int);

void main()
{
int ch,coefficient,exponent;
int listno;
while(1)
{
clrscr();
printf("\nMenu\n");
printf("\n1. Create First Polynomial.\n");
printf("\n2. Display First Polynomial.\n");
printf("\n3. Create Second Polynomial.\n");
printf("\n4. Display Second Polynomial.\n");
printf("\n5. Add Two Polynomials.\n");
printf("\n6. Display Result of Addition.\n");
printf("\n7. Subtract Two Polynomials.\n");
printf("\n8. Display Result of Subtraction.\n");
printf("\n9. Multiply Two Polynomials.\n");
printf("\n10. Display Result of Product.\n");
printf("\n11. Dispose List.\n");
printf("\n12. Exit\n");
printf("\nEnter your choice?\n");
scanf("%d",&ch);
switch(ch)
{
case 1:
printf("\nGenerating first polynomial:\n");
printf("\nEnter coefficient?\n");
```



```
scanf("%d",&coefficient);
printf("\nEnter exponent?\n");
scanf("%d",&exponent);
makenode(coefficient,exponent);
head1 = insertend(head1);
break;
case 2:
display(head1);
break;
case 3:
printf("\nGenerating second polynomial:\n");
printf("\nEnter coefficient?\n");
scanf("%d",&coefficient);
printf("\nEnter exponent?\n");
scanf("%d",&exponent);
makenode(coefficient,exponent);
head2 = insertend(head2);
break;
case 4:
display(head2);
break;
case 5:
printf("\nDisposing result list.\n");
head3=dispose(head3);
head3=addtwopoly(head1,head2,head3);
printf("\nAddition successfully done!\n");
break;
case 6:
display(head3);
break;
case 7:
head3=dispose(head3);
head3=subtwopoly(head1,head2,head3);
printf("\nSubtraction successfully done!\n");
getch();
break;
case 8:
display(head3);
break;
case 9:
head3=dispose(head3);
head4=dispose(head4);
head4=multwopoly(head1,head2,head3);
break;
case 10:
display(head4);
break;
```

```
case 11:
printf(—Enter list number to dispose(1 to 4)?\n);
scanf(—%d\n,&listno);
if(listno==1)
head1=dispose(head1);
else if(listno==2)
head2=dispose(head2);
else if(listno==3)
head3=dispose(head3);
else if(listno==4)
head4=dispose(head4);
else
printf(—Invalid number specified.\n);
break;
case 12:
exit(0);
default:
printf(—Invalid Choice!\n);
break;
}
}
}
```

```
void create()
{
ptr=(struct poly *)malloc(sizeof(struct poly));
if(ptr==NULL)
{
printf(—Memory Allocation Error!\n);
exit(1);
}
}
```

```
void makenode(int c,int e)
{
create();
ptr->coeff = c;
ptr->exp = e;
ptr->next = NULL;
}
```

```
struct poly *insertend(struct poly *head)
{
if(head==NULL)
head = ptr;
else
{
temp=head;
```

```
while(temp->next != NULL)
temp = temp->next;
temp->next = ptr;
}
return head;
}
```

```
void display(struct poly *head)
{
if(head==NULL)
printf("List is empty!\n");
else
{
temp=head;
while(temp!=NULL)
{
printf("(%d,%d)->|",temp->coeff,temp->exp);
temp=temp->next;
}
printf("bb ");
}
getch();
}
```

```
struct poly *addtwopoly(struct poly *h1,struct poly *h2,struct poly *h3)
{
/*
(5,3)->(6,1) + (7,3)->(9,2) = (12,3)->(6,1)->(9,2)
*/
struct poly *temp1,*temp2,*temp3;
temp1=h1;
temp2=h2;
while(temp1!=NULL || temp2!=NULL)
{
if(temp1->exp==temp2->exp)
{
makenode(temp1->coeff+temp2->coeff,temp1->exp);
h3=insertend(h3);
}
else
{
makenode(temp1->coeff,temp1->exp);
h3=insertend(h3);
makenode(temp2->coeff,temp2->exp);
h3=insertend(h3);
}
temp1=temp1->next;
```

```

temp2=temp2->next;
}
if(temp1==NULL && temp2!=NULL)
{
while(temp2!=NULL)
{
makenode(temp2->coeff,temp2->exp);
h3=insertend(h3);
temp2=temp2->next;
}
}
if(temp2==NULL && temp1!=NULL)
{
while(temp1!=NULL)
{
makenode(temp2->coeff,temp2->exp);
h3=insertend(h3);
temp1=temp1->next;
}
}
return h3;
}

struct poly *subtwopoly(struct poly *h1,struct poly *h2,struct poly *h3)
{
/*
(5,3)->(6,1) - (7,3)->(9,2) = (-2,3)+(6,1)-(9,2)
*/
struct poly *temp1,*temp2,*temp3;
temp1=h1;
temp2=h2;
while(temp1!=NULL || temp2!=NULL)
{
if(temp1->exp==temp2->exp)
{
makenode(temp1->coeff-temp2->coeff,temp1->exp);
h3=insertend(h3);
}
else
{
makenode(temp1->coeff,temp1->exp);
h3=insertend(h3);
makenode(-temp2->coeff,temp2->exp);
h3=insertend(h3);
}
temp1=temp1->next;
temp2=temp2->next;
}

```

```

}
if(temp1==NULL && temp2!=NULL)
{
while(temp2!=NULL)
{
makenode(temp2->coeff,temp2->exp);
h3=insertend(h3);
temp2=temp2->next;
}
}
if(temp2==NULL && temp1!=NULL)
{
while(temp1!=NULL)
{
makenode(-temp2->coeff,temp2->exp);
h3=insertend(h3);
temp1=temp1->next;
}
}
return h3;
}

struct poly *multwopoly(struct poly *h1,struct poly *h2,struct poly *h3)
{
/*
h1=(5,3)->(6,1) * h2=(7,3)->(9,2)
(5,3)->(7,3),(9,2) = (35,6),(45,5)
(6,1)->(7,3),(9,2) = (42,4),(54,3)
h3->(35,6)->(45,5)->(42,4)->(54,3)
(35,6)+(45,5)+(42,4)+(54,3)=Result
*/
int res=0;
struct poly *temp1,*temp2,*temp3;
printf("\nDisplaying First Polynomial:nttll);
display(h1);
printf("\nDisplaying Second Polynomial:nttll);
display(h2);

temp1=h1;
while(temp1!=NULL)
{
temp2=h2;
while(temp2!=NULL)
{
makenode(temp1->coeff*temp2->coeff,temp1->exp+temp2->exp);
h3=insertend(h3);
temp2=temp2->next;

```

```
}
temp1=temp1->next;
}

printf("\nDisplaying Initial Result of Product:\n");
display(h3);
getch();

temp1=h3;
while(temp1!=NULL)
{temp2=temp1->next;
res=0;
while(temp2!=NULL)
{
if(temp1->exp==temp2->exp)
res += temp2->coeff;
temp2=temp2->next;
}
if(search(head4,temp1->exp)==1)
{
makenode(res+temp1->coeff,temp1->exp);
head4=insertend(head4);
}
temp1=temp1->next;
}
return head4;
}

int search(struct poly *h,int val)
{
struct poly *tmp;
tmp=h;
while(tmp!=NULL)
{if(tmp->exp==val)
return 0;
tmp=tmp->next;
}
return 1;
}

struct poly *dispose(struct poly *list)
{
if(list==NULL)
{
printf("\nList is already empty.\n");
return list;
}
else
```

```
{
temp=list;
while(list!=NULL)
{
free(temp);
list=list->next;
temp=list;
}
return list;
}
}
```

#### 4.2. Symbol table organizations

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

```
struct hash *hashTable = NULL;
int eleCount = 0;
```

```
struct node {
    int key, age;
    char name[100];
    struct node *next;
};
```

```
struct hash {
    struct node *head;
    int count;
};
```

```
struct node * createNode(int key, char *name, int age) {
    struct node *newnode;
    newnode = (struct node *) malloc(sizeof(struct node));
    newnode->key = key;
    newnode->age = age;
    strcpy(newnode->name, name);
    newnode->next = NULL;
    return newnode;
}
```

```
void insertToHash(int key, char *name, int age) {
    int hashIndex = key % eleCount;
    struct node *newnode = createNode(key, name, age);
    /* head of list for the bucket with index "hashIndex" */
    if (!hashTable[hashIndex].head) {
```

```
    hashTable[hashIndex].head = newnode;
    hashTable[hashIndex].count = 1;
    return;
}
/* adding new node to the list */
newnode->next = (hashTable[hashIndex].head);
/*
 * update the head of the list and no of
 * nodes in the current bucket
 */
hashTable[hashIndex].head = newnode;
hashTable[hashIndex].count++;
return;
}

void deleteFromHash(int key) {
    /* find the bucket using hash index */
    int hashIndex = key % eleCount, flag = 0;
    struct node *temp, *myNode;
    /* get the list head from current bucket */
    myNode = hashTable[hashIndex].head;
    if (!myNode) {
        printf("Given data is not present in hash Table!!\n");
        return;
    }
    temp = myNode;
    while (myNode != NULL) {
        /* delete the node with given key */
        if (myNode->key == key) {
            flag = 1;
            if (myNode == hashTable[hashIndex].head)
                hashTable[hashIndex].head = myNode->next;
            else
                temp->next = myNode->next;

            hashTable[hashIndex].count--;
            free(myNode);
            break;
        }
        temp = myNode;
        myNode = myNode->next;
    }
    if (flag)
        printf("Data deleted successfully from Hash
Table\n"); else
        printf("Given data is not present in hash
Table!!!!\n"); return;
}
```



```
}

void searchInHash(int key) {
    int hashIndex = key % eleCount, flag = 0;
    struct node *myNode;
    myNode = hashTable[hashIndex].head;
    if (!myNode) {
        printf("Search element unavailable in hash table\n");
        return;
    }
    while (myNode != NULL) {
        if (myNode->key == key) {
            printf("VoterID : %d\n", myNode->key);
            printf("Name : %s\n", myNode->name);
            printf("Age : %d\n", myNode->age);
            flag = 1;
            break;
        }
        myNode = myNode->next;
    }
    if (!flag)
        printf("Search element unavailable in hash table\n");
    return;
}

void display() {
    struct node *myNode;
    int i;
    for (i = 0; i < eleCount; i++) {
        if (hashTable[i].count == 0)
            continue;
        myNode = hashTable[i].head;
        if (!myNode)
            continue;
        printf("\nData at index %d in Hash Table:\n", i);
        printf("VoterID   Name       Age  \n");
        printf("-----\n");
        while (myNode != NULL) {
            printf("%-12d", myNode->key);
            printf("%-15s", myNode->name);
            printf("%d\n", myNode->age);
            myNode = myNode->next;
        }
    }
    return;
}
```

```
int main() {
    int n, ch, key, age;
    char name[100];
    printf("Enter the number of elements:");
    scanf("%d", &n);
    eleCount = n;
    /* create hash table with "n" no of buckets */
    hashTable = (struct hash *) calloc(n, sizeof(struct
    hash)); while (1) {
        printf("\n1. Insertion\t2. Deletion\n");
        printf("3. Searching\t4. Display\n5. Exit\n");
        printf("Enter your choice:");
        scanf("%d", &ch);
        switch (ch) {
            case 1:
                printf("Enter the key value:");
                scanf("%d", &key);
                getchar();
                printf("Name:");
                fgets(name, 100, stdin);
                name[strlen(name) - 1] = '\0';
                printf("Age:");
                scanf("%d", &age);
                /*inserting new node to hash table */
                insertToHash(key, name, age);
                break;

            case 2:
                printf("Enter the key to perform deletion:");
                scanf("%d", &key);
                /* delete node with "key" from hash table */
                deleteFromHash(key);
                break;

            case 3:
                printf("Enter the key to search:");
                scanf("%d", &key);
                searchInHash(key);
                break;
            case 4:
                display();
                break;
            case 5:
                exit(0);
            default:
                printf("U have entered wrong option!!\n");
                break;
        }
    }
}
```

```

    }
}
return 0;
}

```

### 4.3. Sparse matrix representation with multilinked data structure

#### Linked List Representation Of Sparse Matrix

If most of the elements in a matrix have the value 0, then the matrix is called spare matrix.

#### Example For 3 X 3 Sparse Matrix:

```

| 1 0 0 |
| 0 0 0 |
| 0 4 0 |

```

#### 3-Tuple Representation Of Sparse Matrix Using Arrays:

```

| 3 3 2 |
| 0 0 1 |
| 2 1 4 |

```

Elements in the first row represents the number of rows, columns and non-zero values in sparse matrix.

First Row - | 3 3 2 |

3 - rows

3 - columns

2 - non- zero values

Elements in the other rows gives information about the location and value of non-zero elements.

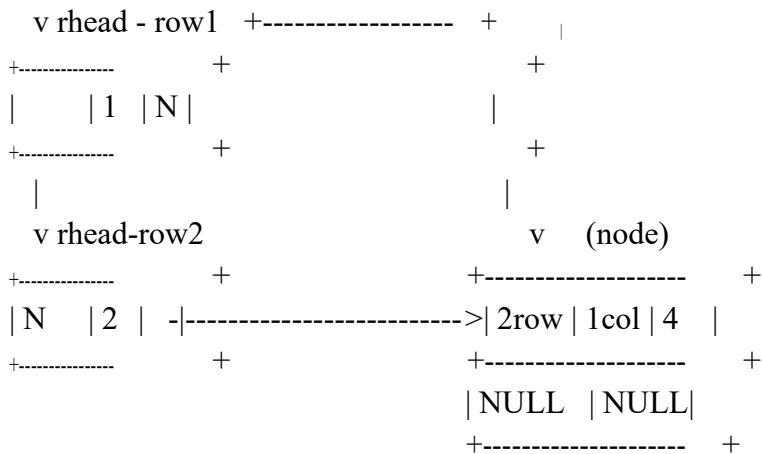
| 0 0 1 | ( Second Row) - represents value 1 at 0th Row, 0th column

| 2 1 4 | (Third Row) - represents value 4 at 2nd Row, 1st column

#### 3-Tuple Representation Of Sparse Matrix Using Linked List:

AH - Additional Header (sparseHead)

AH	chead-col 0	chead - col 1	chead - col 2
3   3   - >	0   - >	1   - >	2   N
v rhead -row 0	v (node)	+	
0   - >   0row   0col   1		+	
	NULL   NULL	+	



## 5. Programming Examples

### 5.1. length of a list

```
int getCount(head)
1) If head is NULL, return 0.
2) Else return 1 + getCount(head->next)
```

### 5.2. Merging two lists

```
struct node* SortedMerge(struct node* a, struct node* b)
```

```
{
    struct node* result = NULL;

    /* Base cases */
    if (a == NULL)
        return(b);
    else if (b==NULL)
        return(a);

    /* Pick either a or b, and recur */
    if (a->data <= b->data)
    {
        result = a;
        result->next = SortedMerge(a->next, b);
    }
    else
    {
        result = b;
        result->next = SortedMerge(a, b->next);
    }
    return(result);
}
```

### 5.3. Removing duplicates

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

```
/* A linked list node */
struct node
{
    int data;
    struct node *next;
};

/* Function to remove duplicates from a unsorted linked list */
void removeDuplicates(struct node *start) {

    struct node *ptr1, *ptr2, *dup;
    ptr1 = start;

    /* Pick elements one by one */
    while(ptr1 != NULL && ptr1->next != NULL)
    {
        ptr2 = ptr1;

        /* Compare the picked element with rest of the elements */
        while(ptr2->next != NULL)
        {
            /* If duplicate then delete it */
            if(ptr1->data == ptr2->next->data)
            {
                /* sequence of steps is important here */
                dup = ptr2->next;
                ptr2->next = ptr2->next->next;
                free(dup);
            }
            else /* This is tricky */
            {
                ptr2 = ptr2->next;
            }
        }
        ptr1 = ptr1->next;
    }
}

/* UTILITY FUNCTIONS */
/* Function to push a node */
void push(struct node** head_ref, int new_data);

/* Function to print nodes in a given linked list */
void printList(struct node *node);

/* Driver program to test above function */
int main()
```

```
{
    struct node *start = NULL;

    /* The constructed linked list is:
    10->12->11->11->12->11->10*/
    push(&start, 10);
    push(&start, 11);
    push(&start, 12);
    push(&start, 11);
    push(&start, 11);
    push(&start, 12);
    push(&start, 10);

    printf("\n Linked list before removing duplicates ");
    printList(start);

    removeDuplicates(start);

    printf("\n Linked list after removing duplicates ");
    printList(start);

    getchar();
}

/* Function to push a node */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
    while(node != NULL)
    {
        printf("%d ", node->data);
    }
}
```

```
    node = node->next;
}
}
```

#### 5.4. Reversing a list

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

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Function to reverse the linked list */
static void reverse(struct node** head_ref)
{
    struct node* prev = NULL;
    struct node* current = *head_ref;
    struct node* next;
    while (current != NULL)
    {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *head_ref = prev;
}

/* Function to push a node */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}
```

```
/* Function to print linked list */
void printList(struct node *head)
{
    struct node *temp = head;
    while(temp != NULL)
    {
        printf("%d ", temp->data);
        temp = temp->next;
    }
}

/* Driver program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

    push(&head, 20);
    push(&head, 4);
    push(&head, 15);
    push(&head, 85);

    printList(head);
    reverse(&head);
    printf("\n Reversed Linked list \n");
    printList(head);
    getchar();
}
```

### 5.5. Union and intersection of two lists

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

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* A utility function to insert a node at the beginning of
a linked list*/
void push(struct node** head_ref, int new_data);

/* A utility function to check if given data is present in a list
*/ bool isPresent(struct node *head, int data);
```



```
/* Function to get union of two linked lists head1 and head2 */
struct node *getUnion(struct node *head1, struct node *head2)
```

```
{
    struct node *result = NULL;
    struct node *t1 = head1, *t2 = head2;

    // Insert all elements of list1 to the result list
    while (t1 != NULL)
    {
        push(&result, t1-
            >data); t1 = t1->next;
    }

    // Insert those elements of list2 which are not
    // present in result list
    while (t2 != NULL)
    {
        if (!isPresent(result, t2->data))
            push(&result, t2->data);
        t2 = t2->next;
    }

    return result;
}
```

```
/* Function to get intersection of two linked lists
   head1 and head2 */
```

```
struct node *getIntersection(struct node *head1,
                             struct node *head2)
{
    struct node *result = NULL;
    struct node *t1 = head1;

    // Traverse list1 and search each element of it in
    // list2. If the element is present in list 2, then
    // insert the element to result
    while (t1 != NULL)
    {
        if (isPresent(head2, t1->data))
            push (&result, t1->data);
        t1 = t1->next;
    }

    return result;
}
```

```
/* A utility function to insert a node at the beginning of a linked list*/
```

```
void push (struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}
```

```
/* A utility function to print a linked list*/
void printList (struct node *node)
{
    while (node != NULL)
    {
        printf ("%d ", node->data);
        node = node->next;
    }
}
```

```
/* A utility function that returns true if data is
present in linked list else return false */
bool isPresent (struct node *head, int data)
{
    struct node *t = head;
    while (t != NULL)
    {
        if (t->data == data)
            return 1;
        t = t->next;
    }
    return 0;
}
```

```
/* Driver program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head1 = NULL;
    struct node* head2 = NULL;
    struct node* intersecn = NULL;
```

```
struct node* unin = NULL;

/*create a linked lits 10->15->5->20 */
push (&head1, 20);
push (&head1, 4);
push (&head1, 15);
push (&head1, 10);

/*create a linked lits 8->4->2->10 */
push (&head2, 10);
push (&head2, 2);
push (&head2, 4);
push (&head2, 8);

intersecn = getIntersection (head1, head2);
unin = getUnion (head1, head2);

printf ("\n First list is \n");
printList (head1);

printf ("\n Second list is \n");
printList (head2);

printf ("\n Intersection list is \n");
printList (intersecn);

printf ("\n Union list is \n");
printList (unin);

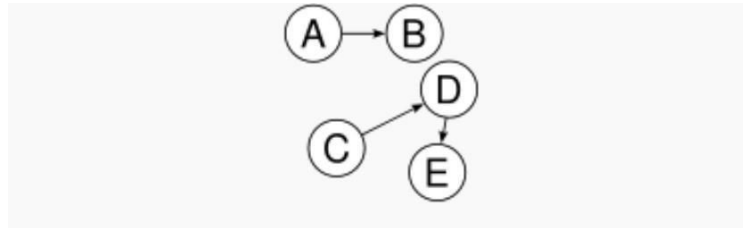
return 0;
}
```

**Module-4****Nonlinear Data Structures****Teaching Hours: 10**

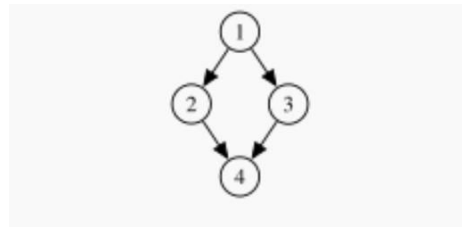
<b>CONTENTS</b>	<b>Pg no</b>
1. <i>Trees</i>	<b>85</b>
2. <i>Types</i>	<b>86</b>
3. <i>Traversal methods</i>	<b>87</b>
4. <i>Binary Search Trees</i>	<b>90</b>
5. <i>Expression tree</i>	<b>93</b>
6. <i>Threaded binary tree</i>	<b>94</b>
7. <i>Conversion of General Trees to Binary Trees</i>	<b>98</b>
8. <i>Constructing BST from traversal orders</i>	<b>98</b>
9. <i>Applications of Trees</i>	<b>103</b>
10. <i>Evaluation of Expression</i>	<b>104</b>
11. <i>Tree based Sorting</i>	<b>106</b>

## Trees

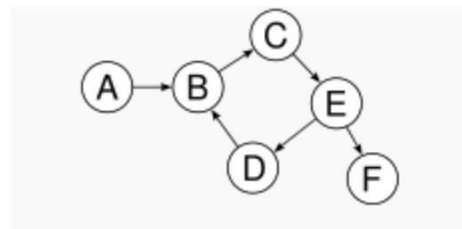
A tree is a (possibly non-linear) data structure made up of nodes or vertices and edges without having any cycle. The tree with no nodes is called the **null** or **empty** tree. A tree that is not empty consists of a root node and potentially many levels of additional nodes that form a hierarchy



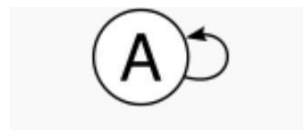
Not a tree: two non-connected parts,  $A \rightarrow B$  and  $C \rightarrow D \rightarrow E$



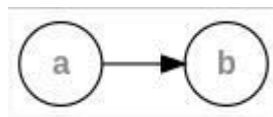
Not a tree: undirected cycle 1-2-4-3



Not a tree: cycle  $B \rightarrow C \rightarrow E \rightarrow D \rightarrow B$



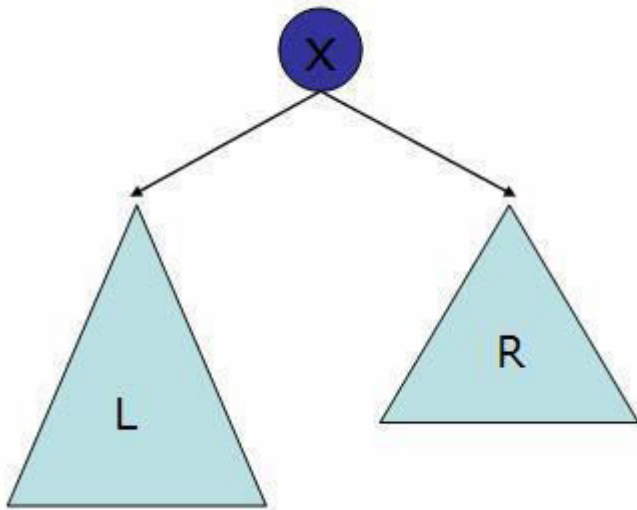
Not a tree: cycle  $A \rightarrow A$



## 1. Types

### Binary Trees

We will see that dealing with **binary** trees, a tree where each node can have no more than two children is a good way to understand trees.

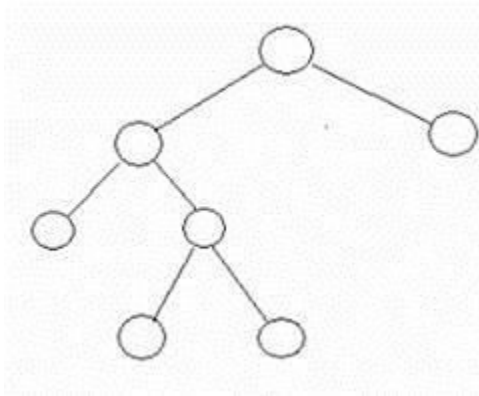
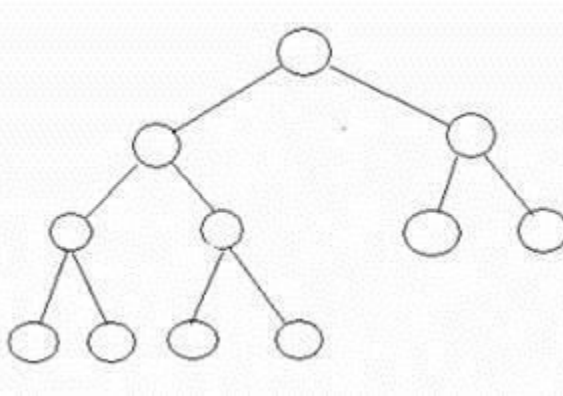


Here is a Java prototype for a tree node:

```
public class BNode
{
    private Object data;
    private BNode left, right;
    public BNode()
    {
        data=left=right=null;
    }
    public BNode(Object data)
    {
        this.data=data;
        left=right=null;
    }
}
```

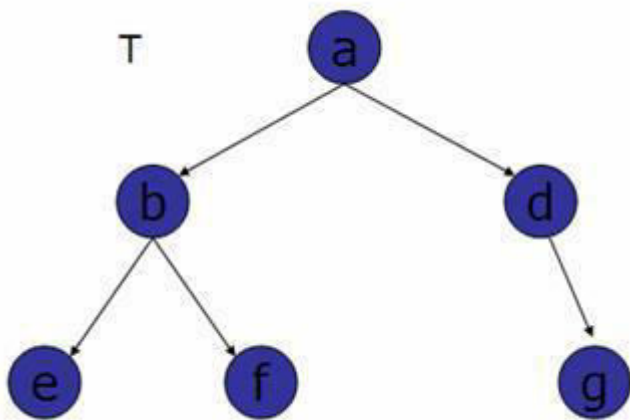
A binary tree in which each node has exactly zero or two children is called a **full binary tree**. In a full tree, there are no nodes with exactly one child.

A **complete binary tree** is a tree, which is completely filled, with the possible exception of the bottom level, which is filled from left to right. A complete binary tree of the height  $h$  has between  $2^h$  and  $2^{(h+1)}-1$  nodes. Here are some examples:

**full tree****complete tree**

### Binary Search Trees

Given a binary tree, suppose we visit each node (recursively) as follows. We visit left child, then root and then the right child. For example, visiting the following tree



In the order defined above will produce the sequence  $\{e, b, f, a, d, g\}$  which we call  $\text{flat}(T)$ . A binary search tree (BST) is a tree, where  $\text{flat}(T)$  is an ordered sequence. In other words, a binary search tree can be —searched|| efficiently using this ordering property. A —balanced|| binary search tree can be searched in  $O(\log n)$  time, where  $n$  is the number of nodes in the tree.

## 2. Traversal methods

Trees can be traversed in *pre-order*, *in-order*, or *post-order*.<sup>[1]</sup> These searches are referred to as *depth-first search* (DFS), as the search tree is deepened as much as possible on each child before going to the next sibling.

For a binary tree, they are defined as display operations recursively at each node, starting with the root, whose algorithm is as follows:

The general recursive pattern for traversing a (non-empty) binary tree is this: At node N you must do these three things:

(L) recursively traverse its left subtree. When this step is finished you are back at N again.

(R) recursively traverse its right subtree. When this step is finished you are back at N again.

(N) Actually process N itself.

We may do these things *in any order* and still have a legitimate traversal. If we do (L) before (R), we call it left-to-right traversal, otherwise we call it right-to-left traversal.

### ***Pre-order***

1. Display the data part of the root (or current node).
2. Traverse the left subtree by recursively calling the pre-order function.
3. Traverse the right subtree by recursively calling the pre-order function.

### ***In-order***

1. Traverse the left subtree by recursively calling the in-order function.
2. Display the data part of the root (or current node).
3. Traverse the right subtree by recursively calling the in-order function.

In a [search tree](#), in-order traversal retrieves data in sorted order.

### ***Post-order***

1. Traverse the left subtree by recursively calling the post-order function.
2. Traverse the right subtree by recursively calling the post-order function.
3. Display the data part of the root (or current node).

The trace of a traversal is called a sequentialisation of the tree. The traversal trace is a list of each visited root. No one sequentialisation according to pre-, in- or post-order describes the underlying tree uniquely. Given a tree with distinct elements, either pre-order or post-order paired with in-order is sufficient to describe the tree uniquely. However, pre-order with post-order leaves some ambiguity in the tree structure.

### ***Generic tree***

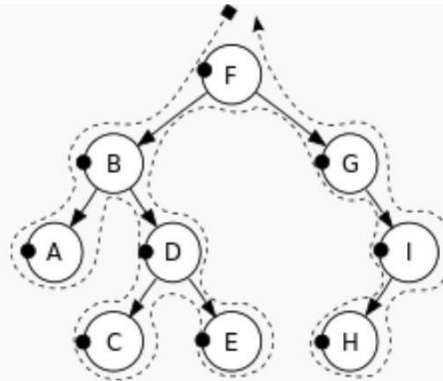
To traverse any tree with depth-first search, perform the following operations recursively at each node:

1. Perform pre-order operation.
2. For each  $i$  from 1 to the number of children do:
  1. Visit  $i$ -th, if present.

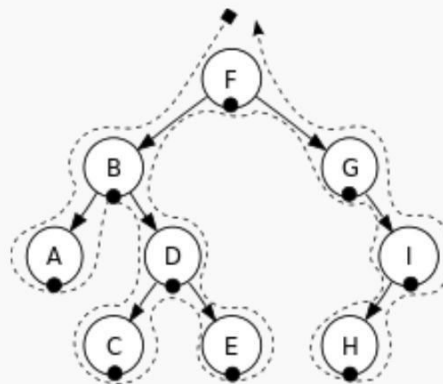


2. Perform in-order operation.
3. Perform post-order operation.

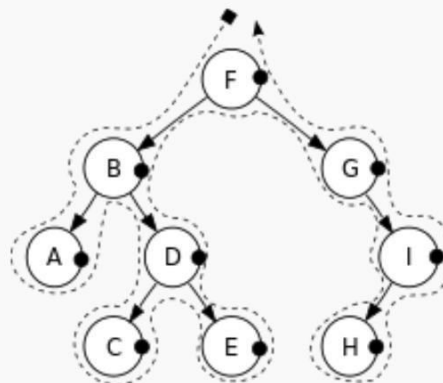
Depending on the problem at hand, the pre-order, in-order or post-order operations may be void, or you may only want to visit a specific child, so these operations are optional. Also, in practice more than one of pre-order, in-order and post-order operations may be required. For example, when inserting into a ternary tree, a pre-order operation is performed by comparing items. A post-order operation may be needed afterwards to re-balance the tree.



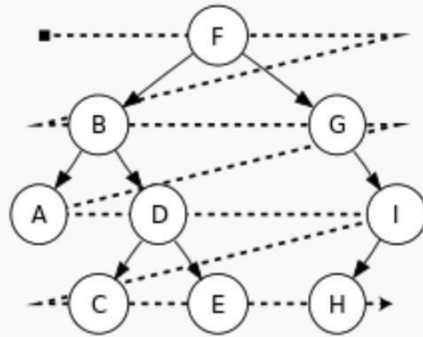
Pre-order: F, B, A, D, C, E, G, I, H.



In-order: A, B, C, D, E, F, G, H, I.



Post-order: A, C, E, D, B, H, I, G, F.



Level-order: F, B, G, A, D, I, C, E, H

### 3. Binary Search Trees

**Binary search trees (BST)**, sometimes called **ordered** or **sorted binary trees**, are a particular type of containers: data structures that store "items" (such as numbers, names etc.) in memory. They allow fast lookup, addition and removal of items, and can be used to implement either dynamic sets of items, or lookup tables that allow finding an item by its *key* (e.g., finding the phone number of a person by name).

Binary search trees keep their keys in sorted order, so that lookup and other operations can use the principle of binary search: when looking for a key in a tree (or a place to insert a new key), they traverse the tree from root to leaf, making comparisons to keys stored in the nodes of the tree and deciding, based on the comparison, to continue searching in the left or right subtrees. On average, this means that each comparison allows the operations to skip about half of the tree, so that each lookup, insertion or deletion takes time proportional to the logarithm of the number of items stored in the tree. This is much better than the linear time required to find items by key in an (unsorted) array, but slower than the corresponding operations on hash tables.

Several variants of the binary search tree have been studied in computer science; this article deals primarily with the basic type, making references to more advanced types when appropriate.

#### Operations

Binary search trees support three main operations: insertion of elements, deletion of elements, and lookup (checking whether a key is present).

#### Searching

Searching a binary search tree for a specific key can be programmed recursively or iteratively.

We begin by examining the root node. If the tree is *null*, the key we are searching for does not exist in the tree. Otherwise, if the key equals that of the root, the search is successful and we return the node. If the key is less than that of the root, we search the left subtree. Similarly, if the key is greater than that of the root, we search the right subtree. This process is repeated until the key is found or the remaining subtree is *null*. If the searched key is not found before a *null* subtree is reached, then the key is not present in the tree. This is easily expressed as a recursive algorithm:

```
1 def search_recursively(key, node):
2     if node is None or node.key == key:
3         return node
4     elif key < node.key:
5         return search_recursively(key, node.left)
6     else: # key > node.key
7         return search_recursively(key, node.right)
```

The same algorithm can be implemented iteratively:

```
1 def search_iteratively(key, node):
2     current_node = node
3     while current_node is not None:
4         if key == current_node.key:
5             return current_node
6         elif key < current_node.key:
7             current_node = current_node.left
8         else: # key > current_node.key:
9             current_node =
current_node.right 10 return None
```

These two examples rely on the order relation being a total order.

If the order relation is only a total preorder a reasonable extension of the functionality is the following: also in case of equality search down to the leaves in a direction specifiable by the user. A binary tree sort equipped with such a comparison function becomes stable.

Because in the worst case this algorithm must search from the root of the tree to the leaf farthest from the root, the search operation takes time proportional to the tree's *height* (see tree terminology). On average, binary search trees with  $n$  nodes have  $O(\log n)$  height.<sup>[a]</sup> However, in the worst case, binary search trees can have  $O(n)$  height, when the unbalanced tree resembles a linked list (degenerate tree).

## Insertion

Insertion begins as a search would begin; if the key is not equal to that of the root, we search the left or right subtrees as before. Eventually, we will reach an external node and add the new key-value pair (here encoded as a record 'newNode') as its right or left child, depending on the node's key. In other words, we examine the root and recursively insert the new node to the left subtree if its key is less than that of the root, or the right subtree if its key is greater than or equal to the root.

Here's how a typical binary search tree insertion might be performed in a binary tree in C++:

```
void insert(Node*& root, int key, int value) {  
    if (!root)  
        root = new Node(key, value);  
    else if (key < root->key)  
        insert(root->left, key, value);  
    else // key >= root->key  
        insert(root->right, key, value);  
}
```

The above *destructive* procedural variant modifies the tree in place. It uses only constant heap space (and the iterative version uses constant stack space as well), but the prior version of the tree is lost. Alternatively, as in the following Python example, we can reconstruct all ancestors of the inserted node; any reference to the original tree root remains valid, making the tree a persistent data structure:

```
def binary_tree_insert(node, key, value):  
    if node is None:  
        return NodeTree(None, key, value, None)  
    if key == node.key:  
        return NodeTree(node.left, key, value, node.right)  
    if key < node.key:  
        return NodeTree(binary_tree_insert(node.left, key, value), node.key, node.value, node.right)  
    else:  
        return NodeTree(node.left, node.key, node.value, binary_tree_insert(node.right, key, value))
```

The part that is rebuilt uses  $O(\log n)$  space in the average case and  $O(n)$  in the worst case.

In either version, this operation requires time proportional to the height of the tree in the worst case, which is  $O(\log n)$  time in the average case over all trees, but  $O(n)$  time in the worst case.

Another way to explain insertion is that in order to insert a new node in the tree, its key is first compared with that of the root. If its key is less than the root's, it is then compared with the key of the root's left child. If its key is greater, it is compared with the root's right child. This process continues, until the new node is compared with

a leaf node, and then it is added as this node's right or left child, depending on its key: if the key is less than the leaf's key, then it is inserted as the leaf's left child, otherwise as the leaf's right child.

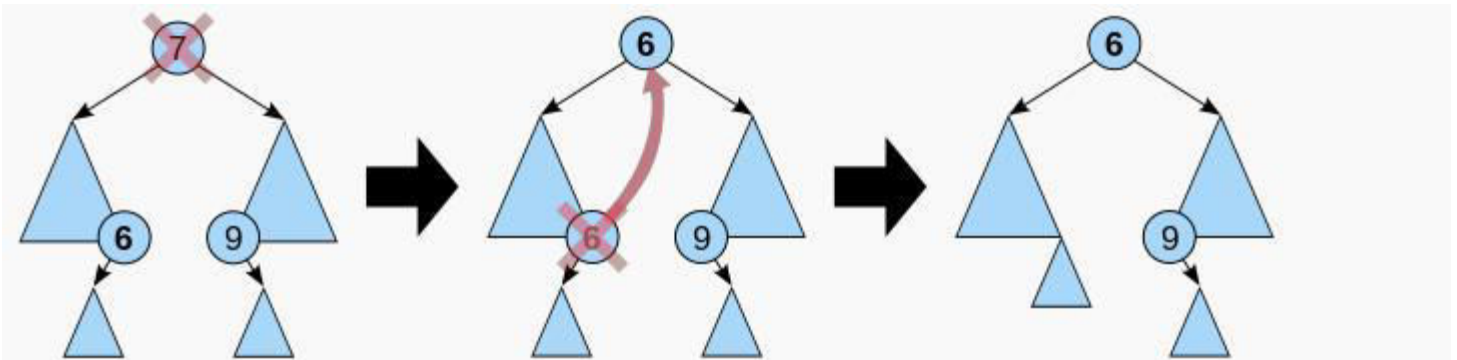
There are other ways of inserting nodes into a binary tree, but this is the only way of inserting nodes at the leaves and at the same time preserving the BST structure.

### Deletion

There are three possible cases to consider:

- Deleting a node with no children: simply remove the node from the tree.
- Deleting a node with one child: remove the node and replace it with its child.
- Deleting a node with two children: call the node to be deleted  $N$ . Do not delete  $N$ . Instead, choose either its in-order successor node or its in-order predecessor node,  $R$ . Copy the value of  $R$  to  $N$ , then recursively call delete on  $R$  until reaching one of the first two cases. If you choose in-order successor of a node, as right sub tree is not NIL (Our present case is node has 2 children), then its in-order successor is node with least value in its right sub tree, which will have at a maximum of 1 sub tree, so deleting it would fall in one of the first 2 cases.

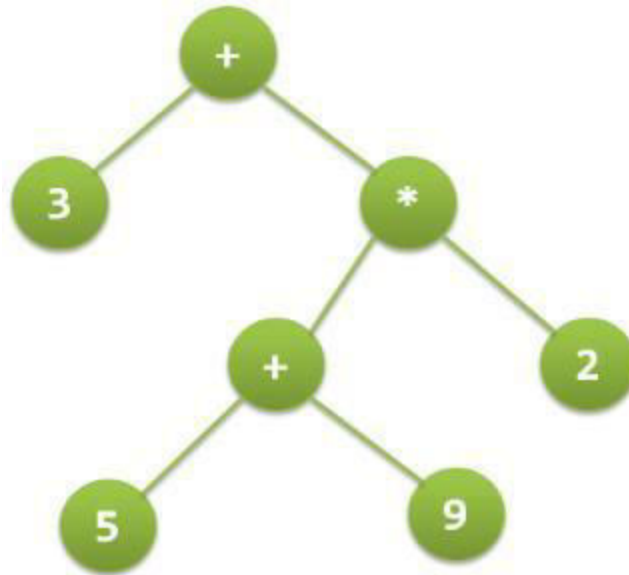
Broadly speaking, nodes with children are harder to delete. As with all binary trees, a node's in-order successor is its right subtree's left-most child, and a node's in-order predecessor is the left subtree's right-most child. In either case, this node will have zero or one children. Delete it according to one of the two simpler cases above.



Deleting a node with two children from a binary search tree. First the rightmost node in the left subtree, the inorder predecessor 6, is identified. Its value is copied into the node being deleted. The inorder predecessor can then be easily deleted because it has at most one child. The same method works symmetrically using the inorder successor labelled 9

#### 4. Expression tree

Expression tree is a binary tree in which each internal node corresponds to operator and each leaf node corresponds to operand so for example expression tree for  $3 + ((5+9)*2)$  would be:



Inorder traversal of expression tree produces infix version of given postfix expression (same with preorder traversal it gives prefix expression)

#### Evaluating the expression represented by expression tree:

Let t be the expression tree

If t is not null then

    If t.value is operand then

        Return t.value

    A = solve(t.left)

    B = solve(t.right)

    // calculate applies operator 't.value'

    // on A and B, and returns value

    Return calculate(A, B, t.value)

#### Construction of Expression Tree:

Now For constructing expression tree we use a stack. We loop through input expression and do following for every character.

- 1) If character is operand push that into stack
- 2) If character is operator pop two values from stack make them its child and push current node again. At the end only element of stack will be root of expression tree.

#### 5. Threaded binary tree

Inorder traversal of a Binary tree is either be done using recursion or with the use of a auxiliary stack. The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion. A

binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node (if it exists).

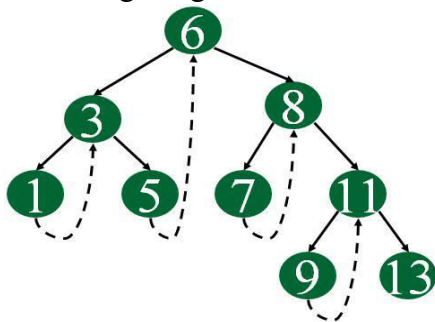
There are two types of threaded binary trees.

**Single Threaded:** Where a NULL right pointers is made to point to the inorder successor (if successor exists)

**Double Threaded:** Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.

The threads are also useful for fast accessing ancestors of a node.

Following diagram shows an example Single Threaded Binary Tree. The dotted lines represent threads.



### C representation of a Threaded Node

Following is C representation of a single threaded node.

```
struct Node
{
    int data;

    Node *left, *right;

    bool rightThread;
}
```

Run on IDE

Since right pointer is used for two purposes, the boolean variable rightThread is used to indicate whether right pointer points to right child or inorder successor. Similarly, we can add leftThread for a double threaded binary tree.

### Inorder Taversal using Threads

Following is C code for inorder traversal in a threaded binary tree.

// Utility function to find leftmost node in atree rooted with n

```
struct Node* leftMost(struct Node *n)
{
    if (n == NULL)
```

```
    return NULL;

while (n->left != NULL)

    n = n->left;

return n;
}

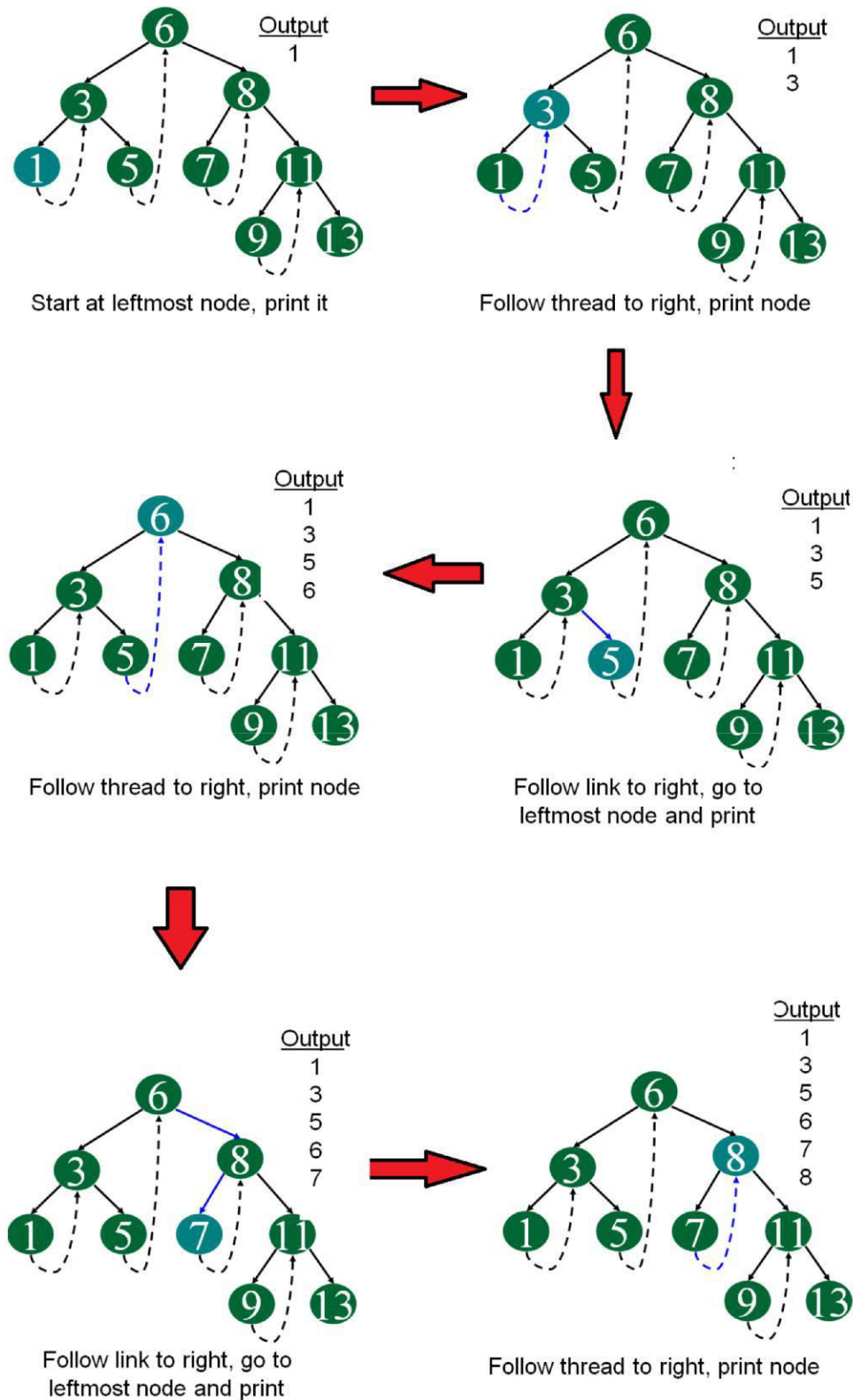
// C code to do inorder traversal in a threaded binary
tree void inOrder(struct Node *root)
{
    struct Node *cur = leftmost(root);
    while (cur != NULL)
    {
        printf("%d ", cur->data);

        // If this node is a thread node, then go to
        // inorder successor
        if (cur->rightThread)
            cur = cur->right;
        else // Else go to the leftmost child in right subtree
            cur = leftmost(cur->right);
    }
}
```

Run on IDE



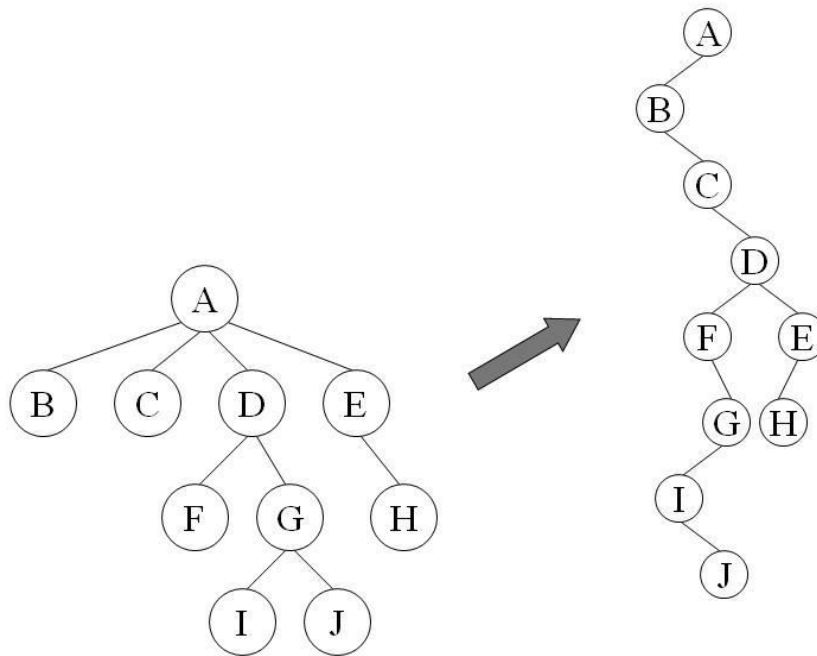
Following diagram demonstrates inorder order traversal using threads.



**continue same way for remaining node.....**

### 6. Conversion of General Trees to Binary Trees

- *Binary tree left child = leftmost child*
- *Binary tree right child = right sibling*



### 7. Constructing BST from traversal orders

Given two arrays that represent preorder and postorder traversals of a full binary tree, construct the binary tree. A **Full Binary Tree** is a binary tree where every node has either 0 or 2 children. Following are examples of Full Trees.

```

  1
 / \
2   3
/ \ / \
4 5 6 7

```

```

  1

```

```

  / \
2   3
  / \
4   5
  / \
6   7

```

```

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

```

It is not possible to construct a general Binary Tree from preorder and postorder traversals (See this). But if we know that the Binary Tree is Full, we can construct the tree without ambiguity. Let us understand this with the help of following example.

Let us consider the two given arrays as  $pre[] = \{1, 2, 4, 8, 9, 5, 3, 6, 7\}$  and  $post[] = \{8, 9, 4, 5, 2, 6, 7, 3, 1\}$ ; In  $pre[]$ , the leftmost element is root of tree. Since the tree is full and array size is more than 1. The value next to 1 in  $pre[]$ , must be left child of root. So we know 1 is root and 2 is left child. How to find the all nodes in left subtree? We know 2 is root of all nodes in left subtree. All nodes before 2 in  $post[]$  must be in left subtree. Now we know 1 is root, elements  $\{8, 9, 4, 5, 2\}$  are in left subtree, and the elements  $\{6, 7, 3\}$  are in right subtree.

```

    1
   / \
  /   \
 /     \
{8, 9, 4, 5, 2} {6, 7, 3}

```

We recursively follow the above approach and get the following tree.

```

1

```

```
    /  \
   2    3
  / \  / \
 4  5 6  7

 /\
8 9
```

```
/* program for construction of full binary tree */
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
/* A binary tree node has data, pointer to left child
```

```
and a pointer to right child */
```

```
struct node
```

```
{
```

```
    int data;
```

```
    struct node *left;
```

```
    struct node *right;
```

```
};
```

```
// A utility function to create a node
```

```
struct node* newNode (int data)
```

```
{
```

```
    struct node* temp = (struct node *) malloc( sizeof(struct node) );
```

```
temp->data = data;

temp->left = temp->right = NULL;


return temp;
}


// A recursive function to construct Full from pre[] and post[].
// preIndex is used to keep track of index in pre[].
// l is low index and h is high index for the current subarray in post[]
struct node* constructTreeUtil (int pre[], int post[], int* preIndex,
                                int l, int h, int size)
{
    // Base case
    if (*preIndex >= size || l > h)
        return NULL;

    // The first node in preorder traversal is root. So take the node at
    // preIndex from preorder and make it root, and increment
    preIndex struct node* root = newNode ( pre[*preIndex] );
    ++*preIndex;

    // If the current subarray has only one element, no need to recur
    if (l == h)
        return root;
```

```
// Search the next element of pre[] in
post[] int i;
for (i = l; i <= h; ++i)
    if (pre[*preIndex] ==
        post[i]) break;

// Use the index of element found in postorder to divide postorder array in
// two parts. Left subtree and right subtree
if (i <= h)
{
    root->left = constructTreeUtil (pre, post, preIndex, l, i, size); root-
    >right = constructTreeUtil (pre, post, preIndex, i + 1, h, size);
}

return root;
}

// The main function to construct Full Binary Tree from given preorder and
// postorder traversals. This function mainly uses constructTreeUtil()
struct node *constructTree (int pre[], int post[], int size)
{
    int preIndex = 0;
    return constructTreeUtil (pre, post, &preIndex, 0, size - 1, size);
}
```

// A utility function to print inorder traversal of a Binary Tree

```
void printInorder (struct node* node)
```

```
{  
    if (node ==  
        NULL) return;  
    printInorder(node->left);  
    printf("%d ", node->data);  
    printInorder(node->right);  
}
```

// Driver program to test above functions

```
int main ()
```

```
{  
    int pre[] = {1, 2, 4, 8, 9, 5, 3, 6, 7};  
    int post[] = {8, 9, 4, 5, 2, 6, 7, 3, 1};  
    int size = sizeof( pre ) / sizeof( pre[0] );
```

```
    struct node *root = constructTree(pre, post, size);
```

```
    printf("Inorder traversal of the constructed tree: \n");
```

```
    printInorder(root);
```

```
    return 0;
```

```
}
```

## ***8. Applications of Trees***

### Why Tree?

Unlike Array and Linked List, which are linear data structures, tree is hierarchical (or non-linear) data structure.

- 1) One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer:

file system

```

      / <-- root
     /  \
    ...  home
        /  \
       ugrad course
      /  /  |  \
     ... cs101 cs112 cs113

```

- 2) If we organize keys in form of a tree (with some ordering e.g., BST), we can search for a given key in moderate time (quicker than Linked List and slower than arrays). Self-balancing search trees like AVL and Red-Black trees guarantee an upper bound of  $O(\text{Log}n)$  for search.
- 3) We can insert/delete keys in moderate time (quicker than Arrays and slower than Unordered Linked Lists). Self-balancing search trees like AVL and Red-Black trees guarantee an upper bound of  $O(\text{Log}n)$  for insertion/deletion.
- 4) Like Linked Lists and unlike Arrays, Pointer implementation of trees don't have an upper limit on number of nodes as nodes are linked using pointers.

As per Wikipedia, following are the common uses of tree.

1. Manipulate hierarchical data.
2. Make information easy to search (see tree traversal).
3. Manipulate sorted lists of data.
4. As a workflow for compositing digital images for visual effects.
5. Router algorithms

## 9. Evaluation of Expression

Consider arithmetic expressions like  $(35 - 3*(3+2)) / 4$ :

For our discussion we'll consider the four arithmetic operators: +, -, \*, /.

We'll study *integer arithmetic*

Here, the /-operator is integer-division (e.g.,  $2/4 = 0$ )



Each operator is a *binary* operator

Takes two numbers and returns the result (a number)

There is a natural precedence among operators:  $\backslash$ ,  $*$ ,  $+$ ,  $-$ .

We'll use parentheses to force a different order, if needed:

Thus  $35 - 3*3+2/4 = 26$  (using integer arithmetic)

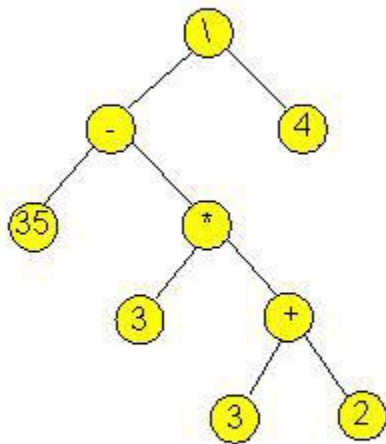
Whereas  $(35 - 3*(3+2)) / 4 = 5$

Now let's examine a *computational procedure* for expressions:

At each step, we apply one of the operators.

The rules of precedence and parentheses tell us the order.

The computational procedure can be written as an expression tree:



In an expression tree:

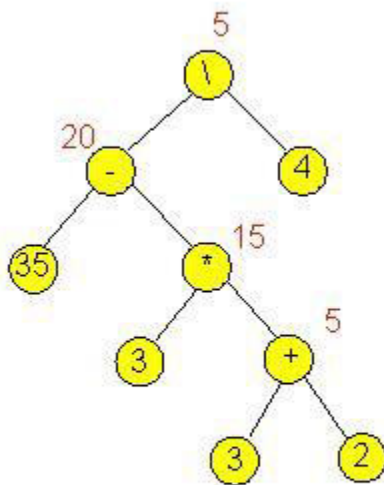
The leaves are numbers (the operands).

The *value* of a leaf is the number.

The non-leaf nodes are operators.

The value of a non-leaf node is the result of the operator applied to the values of the left and right child.

The root node corresponds to the final value of the expression.



### 10. Tree based Sorting

A binary search tree can be used to implement a simple sorting algorithm. Similar to heapsort, we insert all the values we wish to sort into a new ordered data structure—in this case a binary search tree—and then traverse it in order.

The worst-case time of `build_binary_tree` is  $O(n^2)$ —if you feed it a sorted list of values, it chains them into a linked list with no left subtrees. For example, `build_binary_tree([1, 2, 3, 4, 5])` yields the tree `(1 (2 (3 (4 (5))))))`.

There are several schemes for overcoming this flaw with simple binary trees; the most common is the self-balancing binary search tree. If this same procedure is done using such a tree, the overall worst-case time is  $O(n \log n)$ , which is asymptotically optimal for a comparison sort. In practice, the added overhead in time and space for a tree-based sort (particularly for node allocation) make it inferior to other asymptotically optimal

sorts such as heapsort for static list sorting. On the other hand, it is one of the most efficient methods of *incremental sorting*, adding items to a list over time while keeping the list sorted at all times.

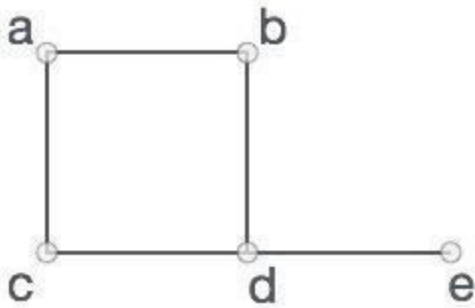
**Module-5****Graph****Teaching Hours: 10**

<b>CONTENTS</b>	<b>Pg no</b>
<b><i>1. Graphs Introduction</i></b>	<b><i>109</i></b>
<b><i>2. Traversal methods</i></b>	<b><i>110</i></b>
2.1. Breadth First Search	<b><i>110</i></b>
2.2. Depth First Search	<b><i>113</i></b>
<b><i>3. Sorting and Searching</i></b>	<b><i>116</i></b>
3.1. Insertion Sort	<b><i>116</i></b>
3.2. Radix sort	<b><i>118</i></b>
3.3. Address Calculation Sort	<b><i>120</i></b>
<b><i>4. Hashing</i></b>	<b><i>121</i></b>
4.1. The Hash Table organizations	<b><i>121</i></b>
4.2. Hashing Functions	<b><i>122</i></b>
4.3. Static and Dynamic Hashing	<b><i>123</i></b>
4.4. Collision-Resolution Techniques	<b><i>125</i></b>
<b><i>5. File Structures</i></b>	<b><i>131</i></b>

### 1. Graphs Introduction

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**.

Formally, a graph is a pair of sets  $(V, E)$ , where  $V$  is the set of vertices and  $E$  is the set of edges, connecting the pairs of vertices. Take a look at the following graph –



In the above graph,

$$V = \{a, b, c, d, e\}$$

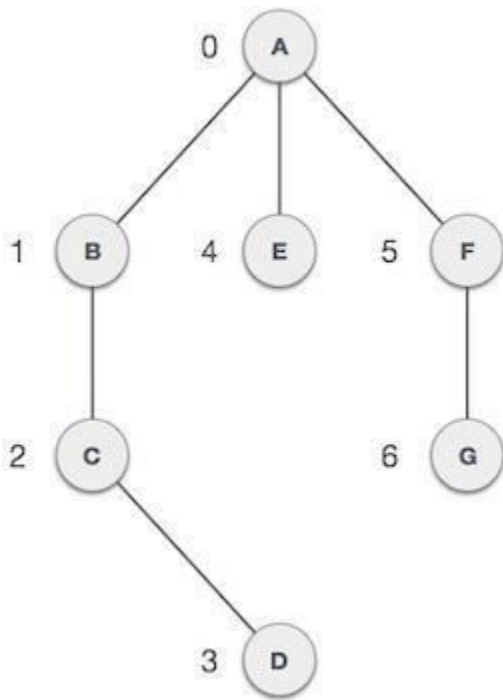
$$E = \{ab, ac, bd, cd, de\}$$

#### Graph Data Structure

Mathematical graphs can be represented in data-structure. We can represent a graph using an array of vertices and a two dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms –

- **Vertex** – Each node of the graph is represented as a vertex. In example given below, labeled circle represents vertices. So A to G are vertices. We can represent them using an array as shown in image below. Here A can be identified by index 0. B can be identified using index 1 and so on.
- **Edge** – Edge represents a path between two vertices or a line between two vertices. In example given below, lines from A to B, B to C and so on represents edges. We can use a two dimensional array to represent array as shown in image below. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.
- **Adjacency** – Two node or vertices are adjacent if they are connected to each other through an edge. In example given below, B is adjacent to A, C is adjacent to B and so on.

- **Path** – Path represents a sequence of edges between two vertices. In example given below, ABCD represents a path from A to D.



### Basic Operations

Following are basic primary operations of a Graph which are following.

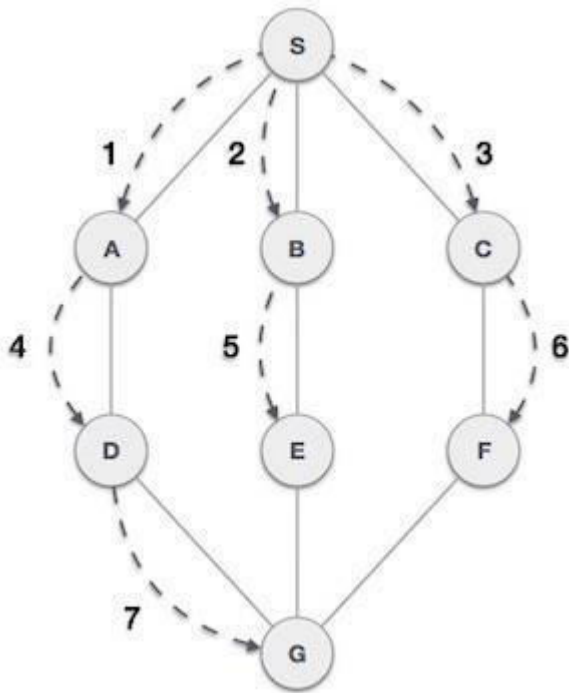
- **Add Vertex** – add a vertex to a graph.
- **Add Edge** – add an edge between two vertices of a graph.
- **Display Vertex** – display a vertex of a graph.

To know more about Graph, please read Graph Theory Tutorial. We shall learn traversing a graph in coming chapters.

## 2. Traversal methods

### 2.1. Breadth First Search

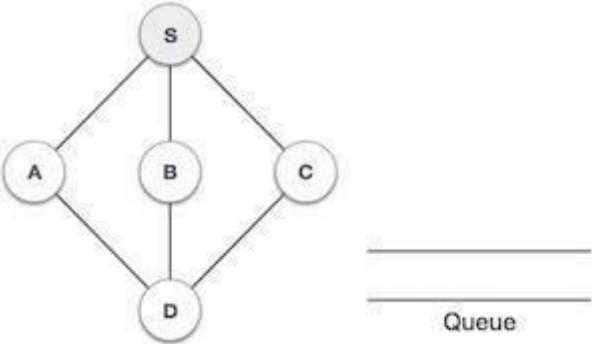
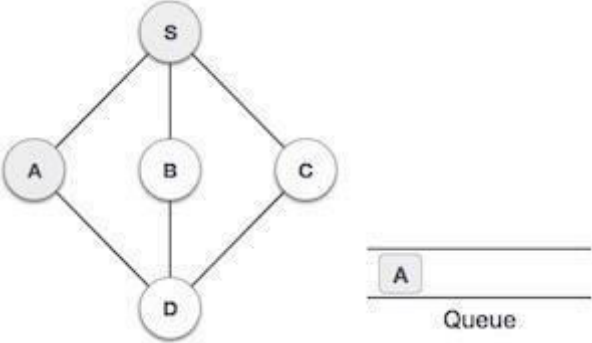
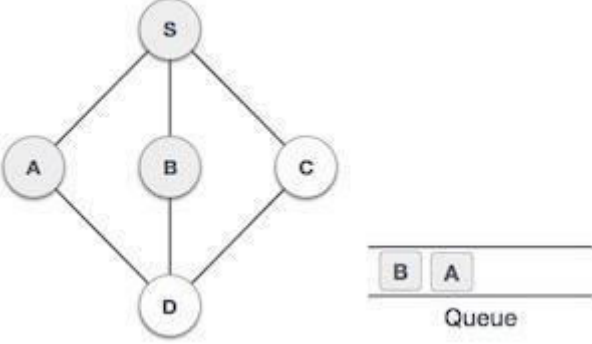
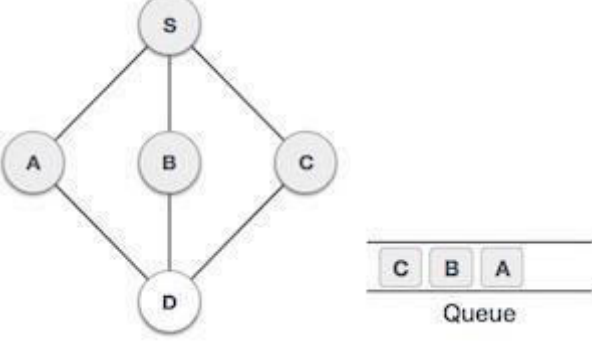
Breadth First Search algorithm(BFS) traverses a graph in a breadthwards motion and uses a queue to remember to get the next vertex to start a search when a dead end occurs in any iteration.



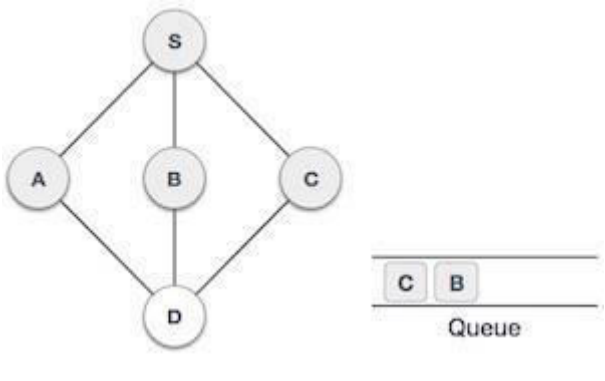
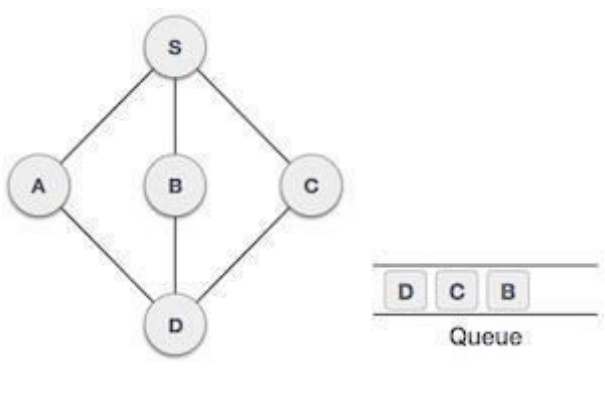
As in example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs following rules.

- **Rule 1** – Visit adjacent unvisited vertex. Mark it visited. Display it. Insert it in a queue.
- **Rule 2** – If no adjacent vertex found, remove the first vertex from queue.
- **Rule 3** – Repeat Rule 1 and Rule 2 until queue is empty.

Step	Traversal	Description
1.		Initialize the queue.

2.	 <p>_____</p> <p>_____</p> <p>Queue</p>	We start from visiting <b>S</b> (starting node), and mark it visited.
3.	 <p>A</p> <p>_____</p> <p>Queue</p>	We then see unvisited adjacent node from <b>S</b> . In this example, we have three nodes but alphabetically we choose <b>A</b> mark it visited and enqueue it.
4.	 <p>B A</p> <p>_____</p> <p>Queue</p>	Next unvisited adjacent node from <b>S</b> is <b>B</b> . We mark it visited and enqueue it.
5.	 <p>C B A</p> <p>_____</p> <p>Queue</p>	Next unvisited adjacent node from <b>S</b> is <b>C</b> . We mark it visited and enqueue it.

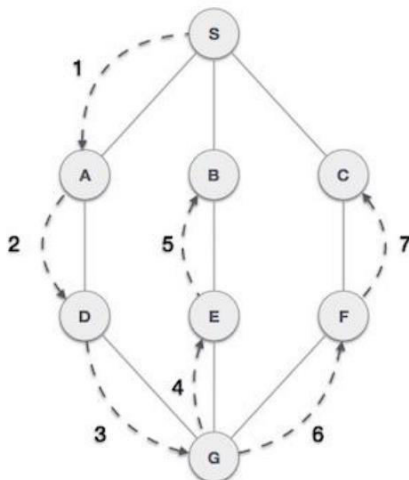


6.		<p>Now <b>S</b> is left with no unvisited adjacent nodes. So we dequeue and find <b>A</b>.</p>
7.		<p>From <b>A</b> we have <b>D</b> as unvisited adjacent node. We mark it visited and enqueue it.</p>

At this stage we are left with no unmarked (unvisited) nodes. But as per algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied the program is over.

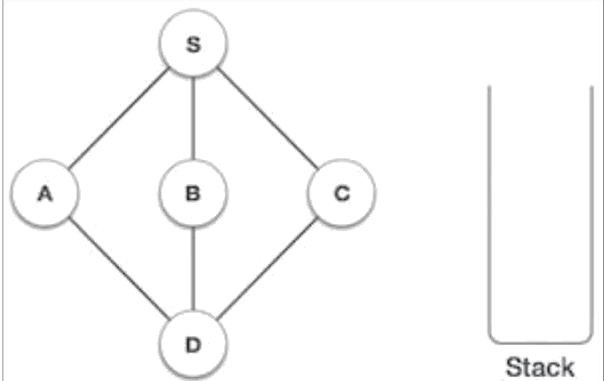
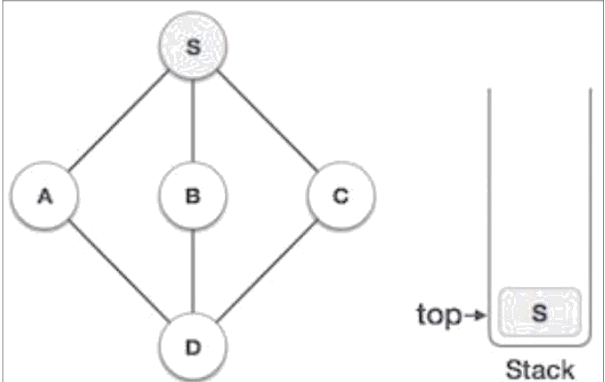
## 2.2. Depth First Search

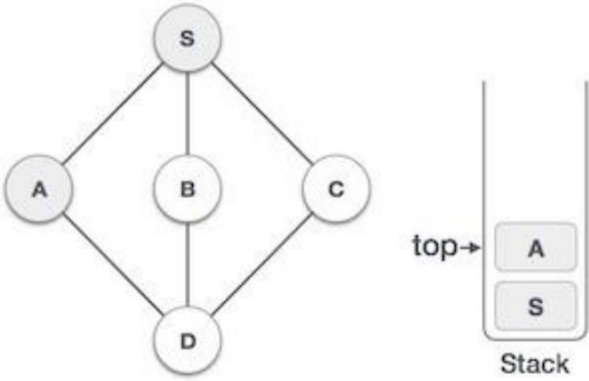
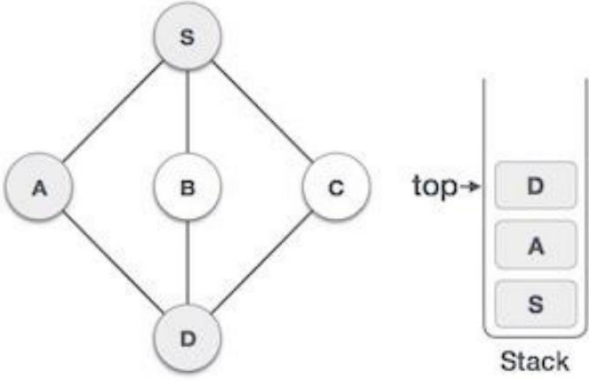
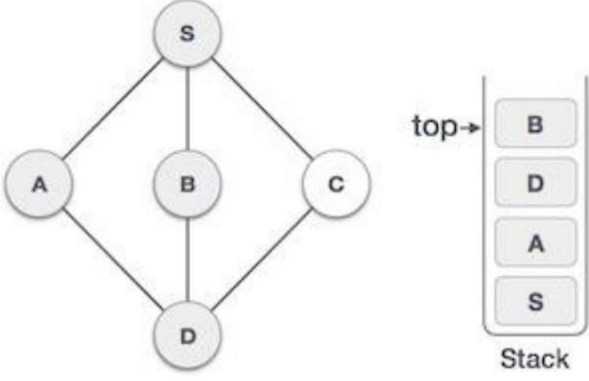
Depth First Search algorithm(DFS) traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search when a dead end occurs in any iteration.

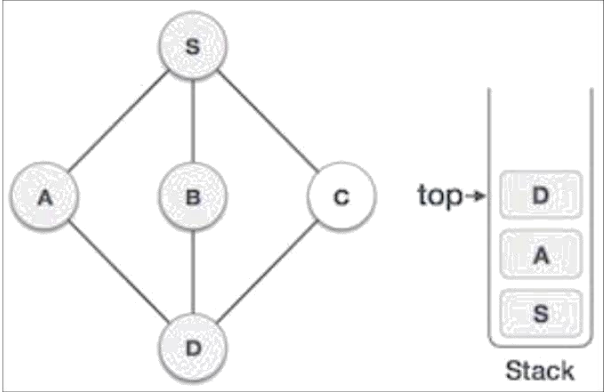
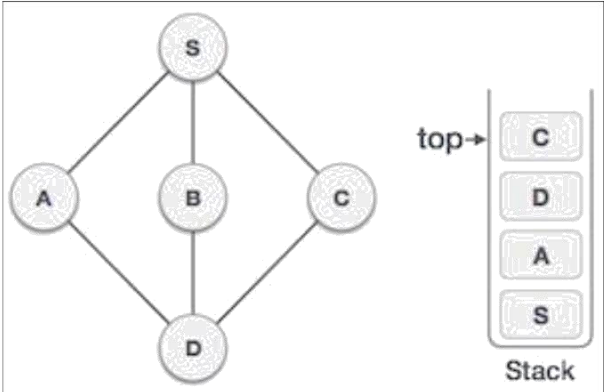


As in example given above, DFS algorithm traverses from A to B to C to D first then to E, then to F and lastly to G. It employs following rules.

- **Rule 1** – Visit adjacent unvisited vertex. Mark it visited. Display it. Push it in a stack.
- **Rule 2** – If no adjacent vertex found, pop up a vertex from stack. (It will pop up all the vertices from the stack which do not have adjacent vertices.)
- **Rule 3** – Repeat Rule 1 and Rule 2 until stack is empty.

Step	Traversal	Description
1.		Initialize the stack
2.		Mark S as visited and put it onto the stack. Explore any unvisited adjacent node from S. We have three nodes and we can pick any of them. For this example, we shall take the node in alphabetical order.

3.	 <p>The graph shows nodes S, A, B, C, and D. S is at the top, connected to A, B, and C. A, B, and C are all connected to D at the bottom. To the right, a stack is shown with 'A' at the top and 'S' below it. An arrow labeled 'top' points to the 'A' node in the stack.</p>	<p>Mark <b>A</b> as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both <b>S</b> and <b>D</b> are adjacent to <b>A</b> but we are concerned for unvisited nodes only.</p>
4.	 <p>The graph is the same as in step 3. The stack now contains 'D', 'A', and 'S' from top to bottom. An arrow labeled 'top' points to the 'D' node in the stack.</p>	<p>Visit <b>D</b> and mark it visited and put onto the stack. Here we have <b>B</b> and <b>C</b> nodes which are adjacent to <b>D</b> and both are unvisited. But we shall again choose in alphabetical order.</p>
5.	 <p>The graph is the same as in step 3. The stack now contains 'B', 'D', 'A', and 'S' from top to bottom. An arrow labeled 'top' points to the 'B' node in the stack.</p>	<p>We choose <b>B</b>, mark it visited and put onto stack. Here <b>B</b> does not have any unvisited adjacent node. So we pop <b>B</b> from the stack.</p>

6.		<p>We check stack top for return to previous node and check if it has any unvisited nodes. Here, we find <b>D</b> to be on the top of stack.</p>
7.		<p>Only unvisited adjacent node is from <b>D</b> is <b>C</b> now. So we visit <b>C</b>, mark it visited and put it onto the stack.</p>

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node which has unvisited adjacent node. In this case, there's none and we keep popping until stack is empty.

### 3. *Sorting and Searching*

#### 3.1. Insertion Sort

This is a in-place comparison based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. A element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and insert it there. Hence the name **insertion sort**.

The array is searched sequentially and unsorted items are moved and inserted into sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of  $O(n^2)$  where n are no. of items.

How insertion sort works?

We take an unsorted array for our example.



Insertion sort compares the first two elements.



It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.



Insertion sort moves ahead and compares 33 with 27.



And finds that 33 is not in correct position.



It swaps 33 with 27. Also it checks with all the elements of sorted sublist. Here we see that sorted sub-list has only one element 14 and 27 is greater than 14. Hence sorted sub-list remain sorted after swapping.



By now we have 14 and 27 in the sorted sublist. Next it compares 33 with 10,.



These values are not in sorted order.



So we swap them.



But swapping makes 27 and 10 unsorted.



So we swap them too.



Again we find 14 and 10 in unsorted order.



And we swap them. By the end of third iteration we have a sorted sublist of 4 items.



This process goes until all the unsorted values are covered in sorted sublist. And now we shall see some programming aspects of insertion sort.

### 3.2. Radix sort

Consider the following 9 numbers:

493 812 715 710 195 437 582 340 385

We should start sorting by comparing and ordering the **one's** digits:

Digit	Sublist
0	340 710

1

	2	812 582
	3	493
	4	
	5	715 195 385
6		
	7	437
8		
9		

Notice that the numbers were added onto the list in the order that they were found, which is why the numbers appear to be unsorted in each of the sublists above. Now, we gather the sublists (in order from the 0 sublist to the 9 sublist) into the main list again:

340 710 812 582 493 715 195 385 437

Note: The **order** in which we divide and reassemble the list is **extremely important**, as this is one of the foundations of this algorithm.

Now, the sublists are created again, this time based on the **ten's** digit:

Digit	Sublist
0	
1	710 812 715
2	
3	437
4	340
5	
6	
7	
8	582 385
9	493 195

Now the sublists are gathered in order from 0 to 9:

710 812 715 437 340 582 385 493 195

Finally, the sublists are created according to the **hundred's** digit:

Digit	Sublist
0	
1	195
2	
3	340 385
4	437 493
5	582
6	

7	710 715
8	812
9	

At last, the list is gathered up again:

195 340 385 437 493 582 710 715 812

And now we have a fully sorted array! Radix Sort is very simple, and a computer can do it fast. When it is programmed properly, Radix Sort is in fact **one of the fastest sorting algorithms** for numbers or strings of letters.

### 3.3. Address Calculation Sort

- In this method a function  $f$  is applied to each key.
- The result of this function determines into which of the several subfiles the record is to be placed.
- The function should have the property that: if  $x \leq y$ ,  $f(x) \leq f(y)$ , Such a function is called order preserving.
- An item is placed into a subfile in correct sequence by placing sorting method – simple insertion is often used.

#### Example:

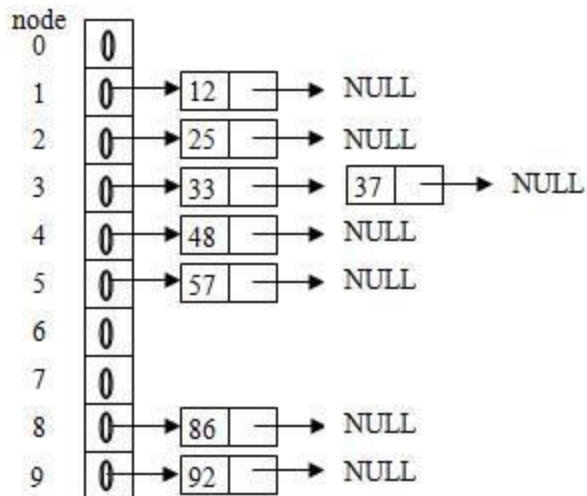
25      57      48      37      12      92      86      33

Let us create 10 subfiles. Initially each of these subfiles is empty. An array of pointer  $f(10)$  is declared, where  $f(i)$  refers to the first element in the file, whose first digit is  $i$ . The number is passed to hash function, which returns its last digit (ten's place digit), which is placed at that position only, in the array of pointers.

num=	25	—	$f(25)$	gives	2
57	—		$f(57)$	gives	5
48	—		$f(48)$	gives	4
37	—		$f(37)$	gives	3
12	—		$f(12)$	gives	1
92	—		$f(92)$	gives	9
86	—		$f(86)$	gives	8
33	—	$f(33)$ gives 3	which is repeated.		

Thus it is inserted in 3<sup>rd</sup> subfile (4<sup>th</sup>) only, but must be checked with the existing elements for its proper position in this subfile.





#### 4. Hashing

##### 4.1. The Hash Table organizations

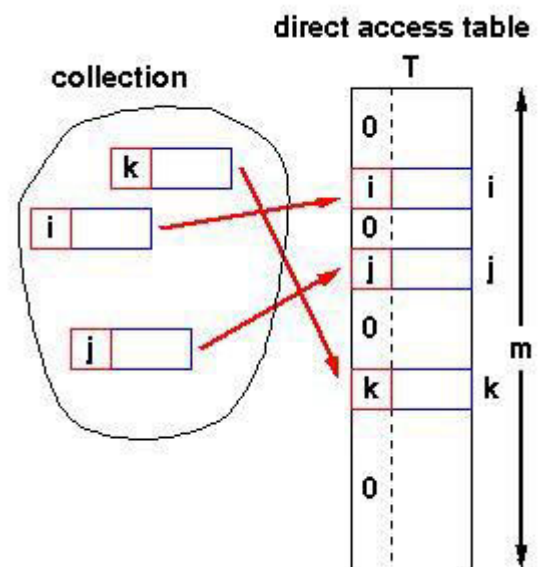
If we have a collection of  $n$  elements whose keys are unique integers in  $(1, m)$ , where  $m \geq n$ , then we can store the items in a *direct address table*,  $T[m]$ , where  $T_i$  is either empty or contains one of the elements of our collection.

Searching a direct address table is clearly an  $O(1)$  operation: for a key,  $k$ , we access  $T_k$ ,

- if it contains an element, return it,
- if it doesn't then return a NULL.

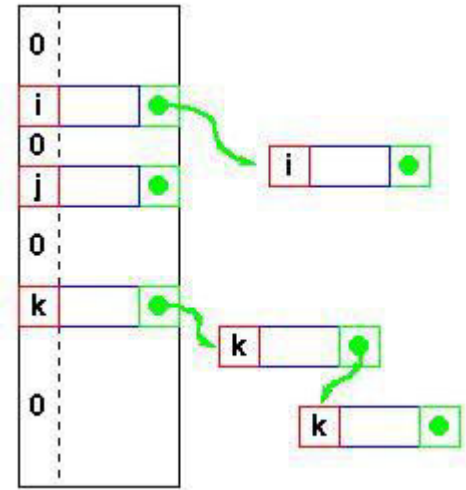
There are two constraints here:

1. the keys must be unique, and
2. the range of the key must be severely bounded.



If the keys are not unique, then we can simply construct a set of  $m$  lists and store the heads of these lists in the direct address table. The time to find an element matching an input key will still be  $O(1)$ .

However, if each element of the collection has some other distinguishing feature (other than its key), and if the maximum number of duplicates is  $\text{ndup}^{\max}$ , then searching for a specific element is  $O(\text{ndup}^{\max})$ . If duplicates are the exception rather than the rule, then  $\text{ndup}^{\max}$  is much smaller than  $n$  and a direct address table will provide good performance. But if  $\text{ndup}^{\max}$  approaches  $n$ , then the time to find a specific element is  $O(n)$  and a tree structure will be more efficient.



The range of the key determines the size of the direct address table and may be too large to be practical. For instance it's not likely that you'll be able to use a direct address table to store elements which have arbitrary 32-bit integers as their keys for a few years yet!

Direct addressing is easily generalised to the case where there is a function,

$$h(k) \Rightarrow (1, m)$$

which maps each value of the key,  $k$ , to the range  $(1, m)$ . In this case, we place the element in  $T[h(k)]$  rather than  $T[k]$  and we can search in  $O(1)$  time as before.

#### 4.2. Hashing Functions

The following functions map a single integer key ( $k$ ) to a small integer bucket value  $h(k)$ .  $m$  is the size of the hash table (number of buckets).

**Division method** (Cormen) Choose a prime that isn't close to a power of 2.  $h(k) = k \bmod m$ . Works badly for many types of patterns in the input data.

**Knuth Variant on Division**  $h(k) = k(k+3) \bmod m$ . Supposedly works much better than the raw division method.

**Multiplication Method** (Cormen). Choose  $m$  to be a power of 2. Let  $A$  be some random-looking real number. Knuth suggests  $M = 0.5 * (\text{sqrt}(5) - 1)$ . Then do the following:

$$\begin{aligned} s &= k * A \\ x &= \text{fractional part of } s \\ h(k) &= \text{floor}(m * x) \end{aligned}$$

This seems to be the method that the theoreticians like.

To do this quickly with integer arithmetic, let  $w$  be the number of bits in a word (e.g. 32) and suppose  $m$  is  $2^p$ . Then compute:

```
s = floor(A * 2^w)
x = k*s
h(k) = x >> (w-p)    // i.e. right shift x by (w-p) bits
                    // i.e. extract the p most significant
                    // bits from x
```

### 4.3. Static and Dynamic Hashing

The good functioning of a hash table depends on the fact that the table size is proportional to the number of entries. With a fixed size, and the common structures, it is similar to linear search, except with a better constant factor. In some cases, the number of entries may be definitely known in advance, for example keywords in a language. More commonly, this is not known for sure, if only due to later changes in code and data. It is one serious, although common, mistake to not provide *any* way for the table to resize. A general-purpose hash table "class" will almost always have some way to resize, and it is good practice even for simple "custom" tables. An implementation should check the load factor, and do something if it becomes too large (this needs to be done only on inserts, since that is the only thing that would increase it).

To keep the load factor under a certain limit, e.g., under  $3/4$ , many table implementations expand the table when items are inserted. For example, in Java's `HashMap` class the default load factor threshold for table expansion is  $3/4$  and in Python's `dict`, table size is resized when load factor is greater than  $2/3$ .

Since buckets are usually implemented on top of a dynamic array and any constant proportion for resizing greater than 1 will keep the load factor under the desired limit, the exact choice of the constant is determined by the same space-time tradeoff as for dynamic arrays.

Resizing is accompanied by a full or incremental table *rehash* whereby existing items are mapped to new bucket locations.

To limit the proportion of memory wasted due to empty buckets, some implementations also shrink the size of the table—followed by a rehash—when items are deleted. From the point of space-time tradeoffs, this operation is similar to the deallocation in dynamic arrays.

#### Resizing by copying all entries

A common approach is to automatically trigger a complete resizing when the load factor exceeds some threshold  $r_{\max}$ . Then a new larger table is allocated, all the entries of the old table are removed and inserted into this new table, and the old table is returned to the free storage pool. Symmetrically, when the load factor falls below a second threshold  $r_{\min}$ , all entries are moved to a new smaller table.

For hash tables that shrink and grow frequently, the resizing downward can be skipped entirely. In this case, the table size is proportional to the maximum number of entries that ever were in the hash table at one time, rather

than the current number. The disadvantage is that memory usage will be higher, and thus cache behavior may be worse. For best control, a "shrink-to-fit" operation can be provided that does this only on request.

If the table size increases or decreases by a fixed percentage at each expansion, the total cost of these resizings, amortized over all insert and delete operations, is still a constant, independent of the number of entries  $n$  and of the number  $m$  of operations performed.

For example, consider a table that was created with the minimum possible size and is doubled each time the load ratio exceeds some threshold. If  $m$  elements are inserted into that table, the total number of extra re-insertions that occur in all dynamic resizings of the table is at most  $m - 1$ . In other words, dynamic resizing roughly doubles the cost of each insert or delete operation.

### Incremental resizing

Some hash table implementations, notably in real-time systems, cannot pay the price of enlarging the hash table all at once, because it may interrupt time-critical operations. If one cannot avoid dynamic resizing, a solution is to perform the resizing gradually:

- During the resize, allocate the new hash table, but keep the old table unchanged.
- In each lookup or delete operation, check both tables.
- Perform insertion operations only in the new table.
- At each insertion also move  $r$  elements from the old table to the new table.
- When all elements are removed from the old table, deallocate it.

To ensure that the old table is completely copied over before the new table itself needs to be enlarged, it is necessary to increase the size of the table by a factor of at least  $(r + 1)/r$  during resizing.

Disk-based hash tables almost always use some scheme of incremental resizing, since the cost of rebuilding the entire table on disk would be too high.

### Monotonic keys

If it is known that key values will always increase (or decrease) monotonically, then a variation of consistent hashing can be achieved by keeping a list of the single most recent key value at each hash table resize operation. Upon lookup, keys that fall in the ranges defined by these list entries are directed to the appropriate hash function—and indeed hash table—both of which can be different for each range. Since it is common to grow the overall number of entries by doubling, there will only be  $O(\log(N))$  ranges to check, and binary search time for the redirection would be  $O(\log(\log(N)))$ . As with consistent hashing, this approach guarantees that any key's hash, once issued, will never change, even when the hash table is later grown.

### Other solutions

Linear hashing is a hash table algorithm that permits incremental hash table expansion. It is implemented using a single hash table, but with two possible lookup functions.

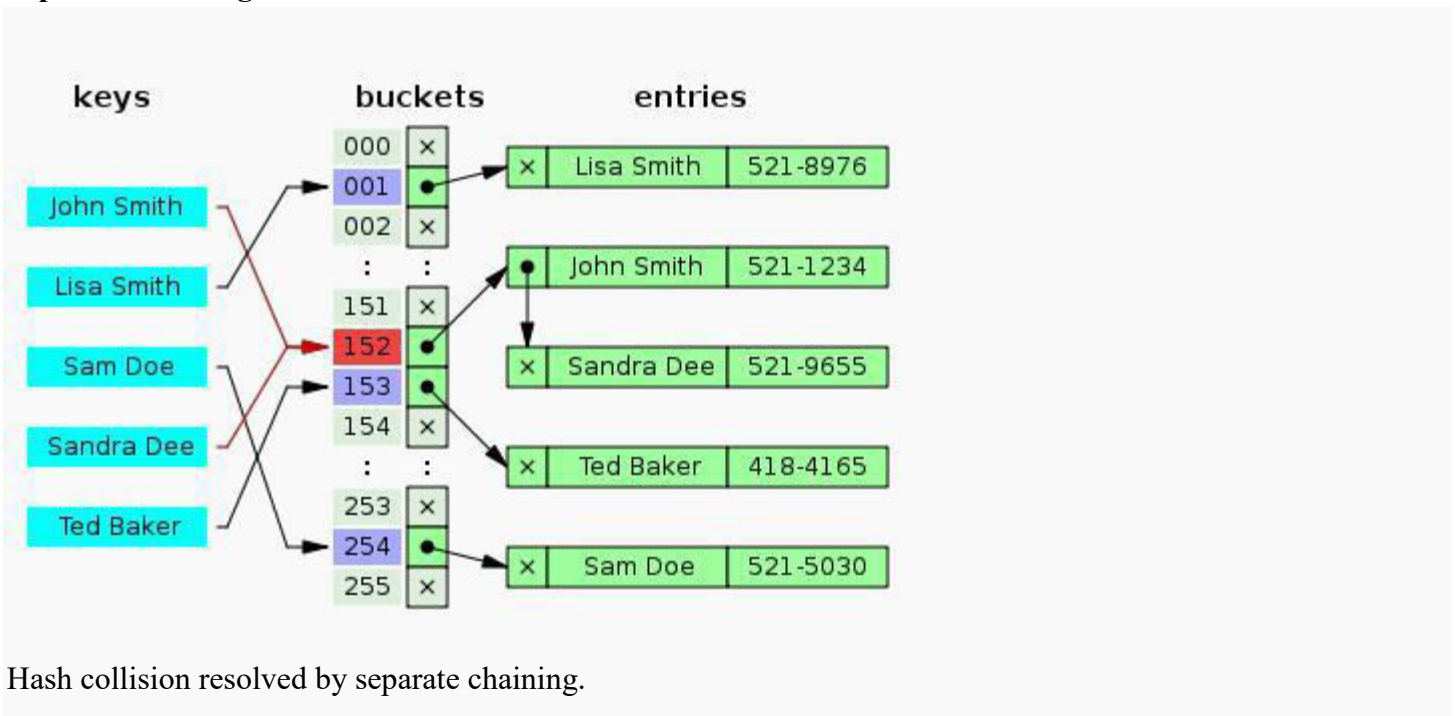
Another way to decrease the cost of table resizing is to choose a hash function in such a way that the hashes of most values do not change when the table is resized. This approach, called consistent hashing, is prevalent in disk-based and distributed hash tables, where rehashing is prohibitively costly.

#### 4.4. Collision-Resolution Techniques

Hash collisions are practically unavoidable when hashing a random subset of a large set of possible keys. For example, if 2,450 keys are hashed into a million buckets, even with a perfectly uniform random distribution, according to the birthday problem there is approximately a 95% chance of at least two of the keys being hashed to the same slot.

Therefore, almost all hash table implementations have some collision resolution strategy to handle such events. Some common strategies are described below. All these methods require that the keys (or pointers to them) be stored in the table, together with the associated values.

##### Separate chaining



Hash collision resolved by separate chaining.

In the method known as *separate chaining*, each bucket is independent, and has some sort of list of entries with the same index. The time for hash table operations is the time to find the bucket (which is constant) plus the time for the list operation.

In a good hash table, each bucket has zero or one entries, and sometimes two or three, but rarely more than that. Therefore, structures that are efficient in time and space for these cases are preferred. Structures that are efficient for a fairly large number of entries per bucket are not needed or desirable. If these cases happen often, the hashing function needs to be fixed.

### *Separate chaining with linked lists*

Chained hash tables with linked lists are popular because they require only basic data structures with simple algorithms, and can use simple hash functions that are unsuitable for other methods.

The cost of a table operation is that of scanning the entries of the selected bucket for the desired key. If the distribution of keys is sufficiently uniform, the *average* cost of a lookup depends only on the average number of keys per bucket—that is, it is roughly proportional to the load factor.

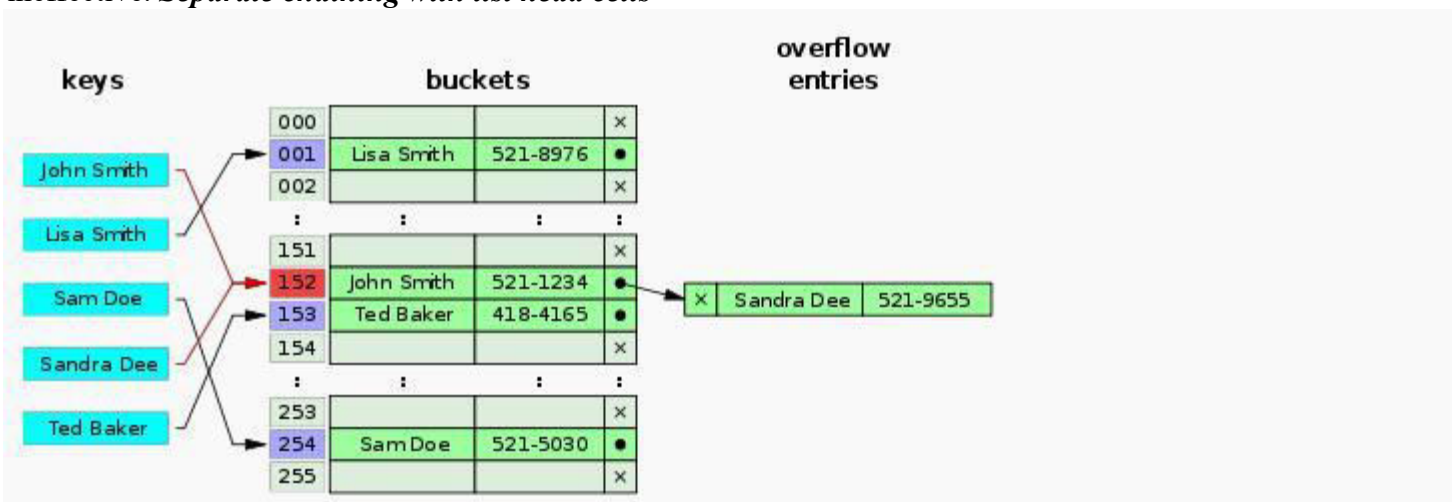
For this reason, chained hash tables remain effective even when the number of table entries  $n$  is much higher than the number of slots. For example, a chained hash table with 1000 slots and 10,000 stored keys (load factor 10) is five to ten times slower than a 10,000-slot table (load factor 1); but still 1000 times faster than a plain sequential list.

For separate-chaining, the worst-case scenario is when all entries are inserted into the same bucket, in which case the hash table is ineffective and the cost is that of searching the bucket data structure. If the latter is a linear list, the lookup procedure may have to scan all its entries, so the worst-case cost is proportional to the number  $n$  of entries in the table.

The bucket chains are often searched sequentially using the order the entries were added to the bucket. If the load factor is large and some keys are more likely to come up than others, then rearranging the chain with a move-to-front heuristic may be effective. More sophisticated data structures, such as balanced search trees, are worth considering only if the load factor is large (about 10 or more), or if the hash distribution is likely to be very non-uniform, or if one must guarantee good performance even in a worst-case scenario. However, using a larger table and/or a better hash function may be even more effective in those cases.

Chained hash tables also inherit the disadvantages of linked lists. When storing small keys and values, the space overhead of the next pointer in each entry record can be significant. An additional disadvantage is that traversing a linked list has poor cache performance, making the processor cache

ineffective. *Separate chaining with list head cells*



Hash collision by separate chaining with head records in the bucket array.

Some chaining implementations store the first record of each chain in the slot array itself. The number of pointer traversals is decreased by one for most cases. The purpose is to increase cache efficiency of hash table access.

The disadvantage is that an empty bucket takes the same space as a bucket with one entry. To save space, such hash tables often have about as many slots as stored entries, meaning that many slots have two or more entries.

### ***Separate chaining with other structures***

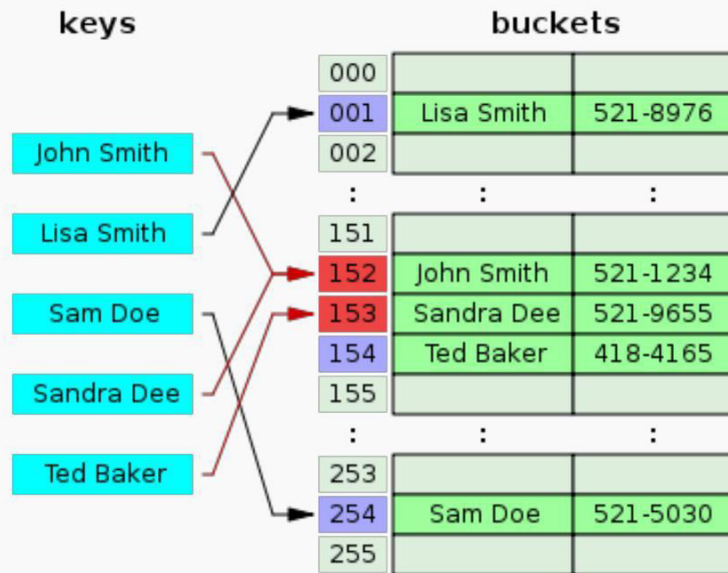
Instead of a list, one can use any other data structure that supports the required operations. For example, by using a self-balancing tree, the theoretical worst-case time of common hash table operations (insertion, deletion, lookup) can be brought down to  $O(\log n)$  rather than  $O(n)$ . However, this approach is only worth the trouble and extra memory cost if long delays must be avoided at all costs (e.g., in a real-time application), or if one must guard against many entries hashed to the same slot (e.g., if one expects extremely non-uniform distributions, or in the case of web sites or other publicly accessible services, which are vulnerable to malicious key distributions in requests).

The variant called array hash table uses a dynamic array to store all the entries that hash to the same slot. Each newly inserted entry gets appended to the end of the dynamic array that is assigned to the slot. The dynamic array is resized in an *exact-fit* manner, meaning it is grown only by as many bytes as needed. Alternative techniques such as growing the array by block sizes or *pages* were found to improve insertion performance, but at a cost in space. This variation makes more efficient use of CPU caching and the translation lookaside buffer (TLB), because slot entries are stored in sequential memory positions. It also dispenses with the next pointers that are required by linked lists, which saves space. Despite frequent array resizing, space overheads incurred by the operating system such as memory fragmentation were found to be small.

An elaboration on this approach is the so-called dynamic perfect hashing, where a bucket that contains  $k$  entries is organized as a perfect hash table with  $k^2$  slots. While it uses more memory ( $n^2$  slots for  $n$  entries, in the worst case and  $n \times k$  slots in the average case), this variant has guaranteed constant worst-case lookup time, and low amortized time for insertion. It is also possible to use a fusion tree for each bucket, achieving constant time for all operations with high probability.



## Open addressing



Hash collision resolved by open addressing with linear probing (interval=1). Note that "Ted Baker" has a unique hash, but nevertheless collided with "Sandra Dee", that had previously collided with "John Smith".

In another strategy, called open addressing, all entry records are stored in the bucket array itself. When a new entry has to be inserted, the buckets are examined, starting with the hashed-to slot and proceeding in some *probe sequence*, until an unoccupied slot is found. When searching for an entry, the buckets are scanned in the same sequence, until either the target record is found, or an unused array slot is found, which indicates that there is no such key in the table.<sup>[13]</sup> The name "open addressing" refers to the fact that the location ("address") of the item is not determined by its hash value. (This method is also called **closed hashing**; it should not be confused with "open hashing" or "closed addressing" that usually mean separate chaining.)

Well-known probe sequences include:

- ☐ Linear probing, in which the interval between probes is fixed (usually 1)
- ☐ Quadratic probing, in which the interval between probes is increased by adding the successive outputs of a quadratic polynomial to the starting value given by the original hash computation
- ☐ Double hashing, in which the interval between probes is computed by a second hash function

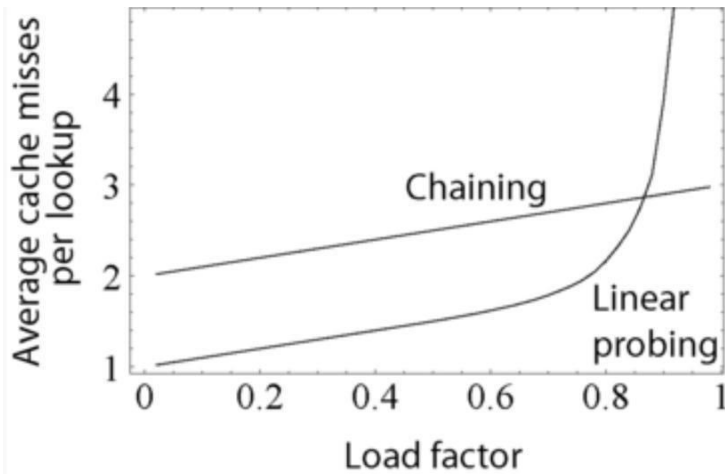
A drawback of all these open addressing schemes is that the number of stored entries cannot exceed the number of slots in the bucket array. In fact, even with good hash functions, their performance dramatically degrades when the load factor grows beyond 0.7 or so. For many applications, these restrictions mandate the use of dynamic resizing, with its attendant costs.

Open addressing schemes also put more stringent requirements on the hash function: besides distributing the keys more uniformly over the buckets, the function must also minimize the clustering of hash values that are



consecutive in the probe order. Using separate chaining, the only concern is that too many objects map to the *same* hash value; whether they are adjacent or nearby is completely irrelevant.

Open addressing only saves memory if the entries are small (less than four times the size of a pointer) and the load factor is not too small. If the load factor is close to zero (that is, there are far more buckets than stored entries), open addressing is wasteful even if each entry is just two words.



This graph compares the average number of cache misses required to look up elements in tables with chaining and linear probing. As the table passes the 80%-full mark, linear probing's performance drastically degrades.

Open addressing avoids the time overhead of allocating each new entry record, and can be implemented even in the absence of a memory allocator. It also avoids the extra indirection required to access the first entry of each bucket (that is, usually the only one). It also has better locality of reference, particularly with linear probing. With small record sizes, these factors can yield better performance than chaining, particularly for lookups. Hash tables with open addressing are also easier to serialize, because they do not use pointers.

On the other hand, normal open addressing is a poor choice for large elements, because these elements fill entire CPU cache lines (negating the cache advantage), and a large amount of space is wasted on large empty table slots. If the open addressing table only stores references to elements (external storage), it uses space comparable to chaining even for large records but loses its speed advantage.

Generally speaking, open addressing is better used for hash tables with small records that can be stored within the table (internal storage) and fit in a cache line. They are particularly suitable for elements of oneword or less. If the table is expected to have a high load factor, the records are large, or the data is variable-sized, chained hash tables often perform as well or better.

Ultimately, used sensibly, any kind of hash table algorithm is usually fast *enough*; and the percentage of a calculation spent in hash table code is low. Memory usage is rarely considered excessive. Therefore, in most cases the differences between these algorithms are marginal, and other considerations typically come into play.

***Coalesced hashing***

A hybrid of chaining and open addressing, coalesced hashing links together chains of nodes within the table itself.<sup>[13]</sup> Like open addressing, it achieves space usage and (somewhat diminished) cache advantages over chaining. Like chaining, it does not exhibit clustering effects; in fact, the table can be efficiently filled to a high density. Unlike chaining, it cannot have more elements than table slots.

***Cuckoo hashing***

Another alternative open-addressing solution is cuckoo hashing, which ensures constant lookup time in the worst case, and constant amortized time for insertions and deletions. It uses two or more hash functions, which means any key/value pair could be in two or more locations. For lookup, the first hash function is used; if the key/value is not found, then the second hash function is used, and so on. If a collision happens during insertion, then the key is re-hashed with the second hash function to map it to another bucket. If all hash functions are used and there is still a collision, then the key it collided with is removed to make space for the new key, and the old key is re-hashed with one of the other hash functions, which maps it to another bucket. If that location also results in a collision, then the process repeats until there is no collision or the process traverses all the buckets, at which point the table is resized. By combining multiple hash functions with multiple cells per bucket, very high space utilization can be achieved.

***Hopscotch hashing***

Another alternative open-addressing solution is hopscotch hashing, which combines the approaches of cuckoo hashing and linear probing, yet seems in general to avoid their limitations. In particular it works well even when the load factor grows beyond 0.9. The algorithm is well suited for implementing a resizable concurrent hash table.

The hopscotch hashing algorithm works by defining a neighborhood of buckets near the original hashed bucket, where a given entry is always found. Thus, search is limited to the number of entries in this neighborhood, which is logarithmic in the worst case, constant on average, and with proper alignment of the neighborhood typically requires one cache miss. When inserting an entry, one first attempts to add it to a bucket in the neighborhood. However, if all buckets in this neighborhood are occupied, the algorithm traverses buckets in sequence until an open slot (an unoccupied bucket) is found (as in linear probing). At that point, since the empty bucket is outside the neighborhood, items are repeatedly displaced in a sequence of hops. (This is similar to cuckoo hashing, but with the difference that in this case the empty slot is being moved into the neighborhood, instead of items being moved out with the hope of eventually finding an empty slot.) Each hop brings the open slot closer to the original neighborhood, without invalidating the neighborhood property of any of the buckets along the way. In the end, the open slot has been moved into the neighborhood, and the entry being inserted can be added to it.

***Robin Hood hashing***

One interesting variation on double-hashing collision resolution is Robin Hood hashing. The idea is that a new key may displace a key already inserted, if its probe count is larger than that of the key at the current position.

The net effect of this is that it reduces worst case search times in the table. This is similar to ordered hash tables except that the criterion for bumping a key does not depend on a direct relationship between the keys. Since both the worst case and the variation in the number of probes is reduced dramatically, an interesting variation is to probe the table starting at the expected successful probe value and then expand from that position in both directions. External Robin Hood hashing is an extension of this algorithm where the table is stored in an external file and each table position corresponds to a fixed-sized page or bucket with  $B$  records.

### **2-choice hashing**

2-choice hashing employs two different hash functions,  $h_1(x)$  and  $h_2(x)$ , for the hash table. Both hash functions are used to compute two table locations. When an object is inserted in the table, then it is placed in the table location that contains fewer objects (with the default being the  $h_1(x)$  table location if there is equality in bucket size). 2-choice hashing employs the principle of the power of two choices.

## **5. File Structures**

### **Introduction**

This chapter is mainly concerned with the way in which file structures are used in document retrieval. Most surveys of file structures address themselves to applications in data management which is reflected in the terminology used to describe the basic concepts. I shall (on the whole) follow Hsiao and Harary whose terminology is perhaps slightly non-standard but emphasises the logical nature of file structures. A further advantage is that it enables me to bridge the gap between data management and document retrieval easily. A few other good references on file structures are Roberts, Bertziss, Dodd, and Climensson.

### **Logical or physical organisation and data independence**

There is one important distinction that must be made at the outset when discussing file structures. And that is the difference between the *logical* and *physical* organisation of the data. On the whole a file structure will specify the logical structure of the data, that is the relationships that will exist between data items independently of the way in which these relationships may actually be realised within any computer. It is this logical aspect that we will concentrate on. The physical organisation is much more concerned with optimising the use of the storage medium when a particular logical structure is stored on, or in it. Typically for every unit of physical store there will be a number of units of the logical structure (probably records) to be stored in it. For example, if we were to store a tree structure on a magnetic disk, the physical organisation would be concerned with the best way of packing the nodes of the tree on the disk given the access characteristics of the disk.

The work on data bases has been very much concerned with a concept called *data independence*. The aim of this work is to enable programs to be written independently of the logical structure of the data they would interact with. The independence takes the following form, should the file structure overnight be changed from an inverted to a serial file the program should remain unaffected. This independence is achieved by interposing a *data model* between the user and the data base. The user sees the data model rather than the data base, and all his programs communicate with the model. The user therefore has no interest in the structure of the file.

There is a school of thought that says that applications in library automation and information retrieval should follow this path as well. And so it should. Unfortunately, there is still much debate about what a good data model should look like. Furthermore, operational implementations of some of the more advanced theoretical systems do not exist yet. So any suggestion that an IR system might be implemented through a data base package should still seem premature. Also, the scale of the problems in IR is such that efficient implementation of the application still demands close scrutiny of the file structure to be used.

Nevertheless, it is worth taking seriously the trend away from user knowledge of file structures, a trend that has been stimulated considerably by attempts to construct a theory of data. There are a number of proposals for dealing with data at an abstract level. The best known of these by now is the one put forward by Codd, which has become known as the relational model. In it data are described by  $n$ -tuples of attribute values. More formally if the data is described by *relations*, a relation on a set of *domains*  $D_1, \dots, D_n$  can be represented by a set of ordered  $n$ -tuples each of the form  $(d_1, \dots, d_n)$  where  $d_i$   $\llbracket$ propersubset $\rrbracket D_i$ . As it is rather difficult to cope with general relations, various levels (three in fact) of normalisation have been introduced restricting the kind of relations allowed.

A second approach is the *hierarchical* approach. It is used in many existing data base systems. This approach works as one might expect: data is represented in the form of hierarchies. Although it is more restrictive than the relational approach it often seems to be the natural way to proceed. It can be argued that in many applications a hierarchic structure is a good approximation to the natural structure in the data, and that the resulting loss in precision of representation is worth the gain in efficiency and simplicity of representation.

The third approach is the *network* approach associated with the proposals by the Data Base Task Group of CODASYL. Here data items are linked into a network in which any given link between two items exists because it satisfies some condition on the attributes of those items, for example, they share an attribute. It is more general than the hierarchic approach in the sense that a node can have any number of immediate superiors. It is also equivalent to the relational approach in descriptive power.

The whole field of data base structures is still very much in a state of flux. The advantages and disadvantages of each approach are discussed very thoroughly in Date, who also gives excellent annotated citations to the current literature. There is also a recent *Computing Survey*[11] which reviews the current state of the art. There have been some very early proponents of the relational approach in IR, as early as 1967 Maron and Levien discussed the design and implementation of an IR system via relations, be it binary ones. Also Prywes and Smith in their review chapter in the *Annual Review of Information Science and Technology* more recently recommended the DBTG proposals as ways of implementing IR systems.

Lurking in the background of any discussion of file structures nowadays is always the question whether data base technology will overtake all. Thus it may be that any application in the field of library automation and information retrieval will be implemented through the use of some appropriate data base package. This is certainly a possibility but not likely to happen in the near future. There are several reasons. One is that data base systems are *general* purpose systems whereas automated library and retrieval systems are *special* purpose. Normally one pays a price for generality and in this case it is still too great. Secondly, there now is a considerable investment in providing special purpose systems (for example, MARC) and this is not written off very easily. Nevertheless a trend towards increasing use of data-base technology exists and is well illustrated by the increased prominence given to it in the *Annual Review of Information Science and Technology*.

## A language for describing file structures

Like all subjects in computer science the terminology of file structures has evolved higgledy-piggledy without much concern for consistency, ambiguity, or whether it was possible to make the kind of distinctions that were important. It was only much later that the need for a well-defined, unambiguous language to describe file structures became apparent. In particular, there arose a need to communicate ideas about file structures without getting bogged down by hardware considerations.

This section will present a formal description of file structures. The framework described is important for the understanding of any file structure. The terminology is based on that introduced by Hsiao and Harary (but also see Hsiao and Manola and Hsiao). Their terminology has been modified and extended by Severance, a summary of this can be found in van Rijsbergen. Jonkers has formalised a different framework which provides an interesting contrast to the one described here.

### Basic terminology

Given a set of 'attributes'  $A$  and a set of 'values'  $V$ , then a *record*  $R$  is a subset of the cartesian product  $A \times V$  in which each attribute has one and only one value. Thus  $R$  is a set of ordered pairs of the form (an attribute, its value). For example, the record for a document which has been processed by an automatic content analysis algorithm would be

$$R = \{(K1, x1), (K2, x2), \dots (Km, xm)\}$$

The  $K_i$ 's are keywords functioning as attributes and the value  $x_i$  can be thought of as a numerical weight. Frequently documents are simply characterised by the absence or presence of keywords, in which case we write

$$R = \{Kt1, Kt2, \dots, Kti\}$$

where  $Kti$  is present if  $x_{ti} = 1$  and is absent otherwise.

Records are collected into logical units called files. They enable one to refer to a set of records by name, the file name. The records within a file are often organised according to relationships between the records. This logical organisation has become known as a file structure (or data structure).

It is difficult in describing file structures to keep the logical features separate from the physical ones. The latter are characteristics forced upon us by the recording media (e.g. tape, disk). Some features can be defined abstractly (with little gain) but are more easily understood when illustrated concretely. One such feature is a *field*. In any implementation of a record, the attribute values are usually positional, that is the identity of an attribute is given by the position of its attribute value within the record. Therefore the data within a record is registered sequentially and has a definite beginning and end. The record is said to be divided into *fields* and the  $n$ th field carries the  $n$ th attribute value. Pictorially we have an example of a record with associated fields in *Figure 4.1*.

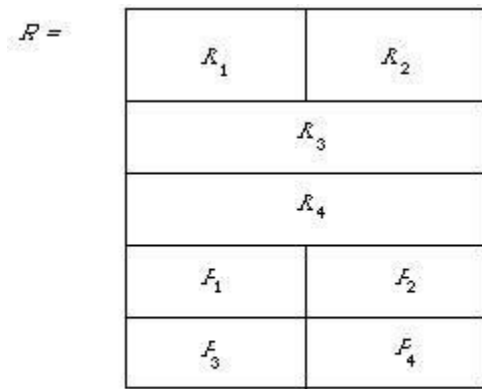


Figure 4.1. An example of a record with associated fields.

The fields are not necessarily constant in length. To find the value of the attribute  $K_4$ , we first find the address of the record  $R$  (which is actually the address of the start of the record) and read the data in the 4th field.

In the same picture I have also shown some fields labelled  $P_i$ . They are addresses of other records, and are commonly called pointers. Now we have extended the definition of a record to a set of attribute-value pairs and pointers. Each pointer is usually associated with a particular attribute-value pair. For example, (see Figure 4.2) pointers could be used to link all records for which the value  $x_1$  (of attribute  $K_1$ ) is  $a$ , similarly for  $x_2$  equal to  $b$ , etc.

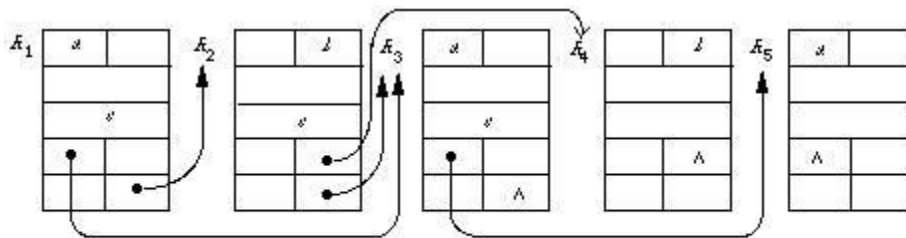


Figure 4.2. A demonstration of the use of pointers to link records.

To indicate that a record is the last record pointed to in a list of records we use the *null pointer* [[logicaland]]. The pointer associated with attribute  $K$  in record  $R$  will be called a *K-pointer*. An attribute (keyword) that is used in this way to organise a file is called a *key*.

To unify the discussion of file structures we need some further concepts. Following Hsiao and Harary again, we define a list  $L$  of records with respect to a keyword  $K$ , or more briefly a *K-list* as a set of records containing  $K$  such that:

- (1) the *K*-pointers are distinct;
- (2) each non-null *K*-pointer in  $L$  gives the address of a record within  $L$ ;
- (3) there is a unique record in  $L$  not pointed to by any record containing  $K$ ; it is called the *beginning* of the list; and

(4) there is a unique record in  $L$  containing the null  $K$ -pointer; it is the *end* of the list.

(Hsiao and Harary state condition (2) slightly differently so that no two  $K$ -lists have a record in common; this only appears to complicate things.)

From our previous example:

$K1$ -list :  $R1, R2, R5$

$K2$ -list :  $R2, R4$

$K4$ -list :  $R1, R2, R3$

Finally, we need the definition of a *directory* of a file. Let  $F$  be a file whose records contain just  $m$  different keywords  $K1, K2, \dots, Km$ . Let  $ni$  be the number of records containing the keyword  $Ki$ , and  $hi$  be the number of  $Ki$ -lists in  $F$ . Furthermore, we denote by  $aij$  the beginning address of the  $j$ th  $Ki$ -list. Then the *directory* is the set of sequences

$(Ki, ni, hi, ai1, ai2, \dots, aih_i) \ i = 1, 2, \dots, m$

We are now in a position to give a unified treatment of sequential files, inverted files, index-sequential files and multi-list files.

### ***Sequential files***

A sequential file is the most primitive of all file structures. It has no directory and no linking pointers. The records are generally organised in lexicographic order on the value of some key. In other words, a particular attribute is chosen whose value will determine the order of the records. Sometimes when the attribute value is constant for a large number of records a second key is chosen to give an order when the first key fails to discriminate.

The implementation of this file structure requires the use of a sorting routine.

Its main advantages are:

- (1) it is easy to implement;
- (2) it provides fast access to the next record using lexicographic order.

Its disadvantages:

- (1) it is difficult to update - inserting a new record may require moving a large proportion of the file;
- (2) random access is extremely slow.



Sometimes a file is considered to be sequentially organised despite the fact that it is not ordered according to any key. Perhaps the date of acquisition is considered to be the key value, the newest entries are added to the end of the file and therefore pose no difficulty to updating.

### *Inverted files*

The importance of this file structure will become more apparent when Boolean Searches are discussed in the next chapter. For the moment we limit ourselves to describing its structure.

An *inverted file* is a file structure in which every list contains only one record. Remember that a list is defined with respect to a keyword  $K$ , so every  $K$ -list contains only one record. This implies that the directory will be such that  $ni = hi$  for all  $i$ , that is, the number of records containing  $K_i$  will equal the number of  $K_i$ -lists. So the directory will have an address for each record containing  $K_i$ . For document retrieval this means that given a keyword we can immediately locate the addresses of all the documents containing that keyword. For the previous example let us assume that a non-black entry in the field corresponding to an attribute indicates the presence of a keyword and a black entry its absence. Then the directory will point to the file in the way shown in Figure 4.3. The definition of an inverted file does *not* require that the addresses in the directory are in any order. However, to facilitate operations such as conjunction ('and') and disjunction ('or') on any two inverted lists, the addresses are normally kept in record number order. This means that 'and' and 'or' operations can be performed with one pass through both lists. The penalty we pay is of course that the inverted file becomes slower to update.

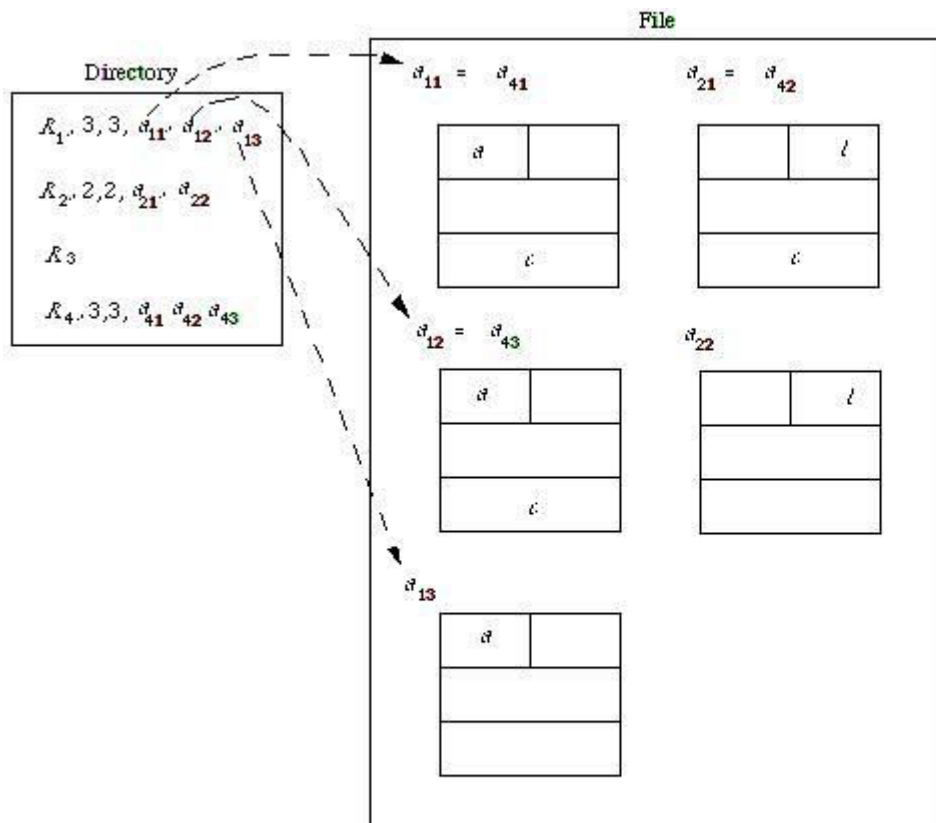


Figure 4.3. An inverted file



### *Index-sequential files*

An index-sequential file is an inverted file in which for every keyword  $K_i$ , we have  $n_i = h_i = 1$  and  $a_{11} < a_{21} \dots < a_{m1}$ . This situation can only arise if each record has just one unique keyword, or one unique attribute-value. In practice therefore, this set of records may be order sequentially by a key. Each key value appears in the directory with the associated address of its record. An obvious interpretation of a key of this kind would be the record number. In our example none of the attributes would do the job except the record number. Diagrammatically the index-sequential file would therefore appear as shown in Figure 4.4. I have deliberately written  $R_i$  instead of  $K_i$  to emphasise the nature of the key.

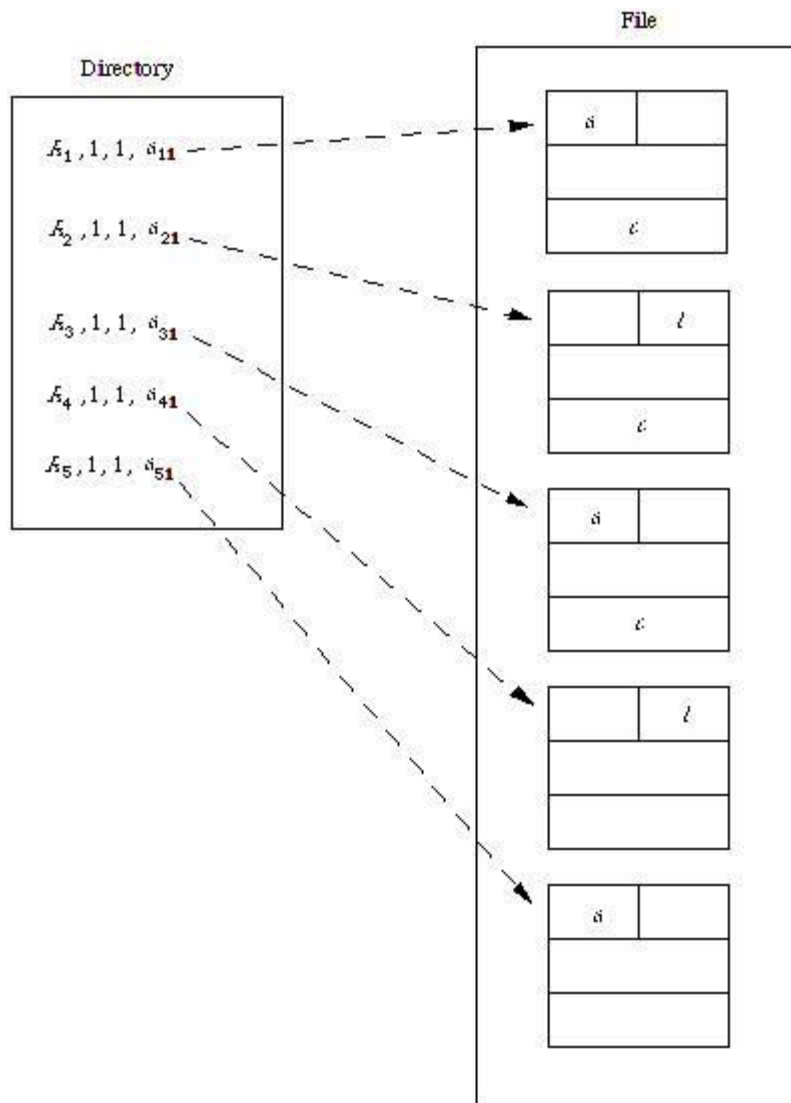


Figure 4.4. An index-sequential file

In the literature an index-sequential file is usually thought of as a sequential file with a hierarchy of indices. This does not contradict the previous definition, it merely describes the way in which the directory is implemented. It is not surprising therefore that the indexes ('index' = 'directory' here) are often oriented to the characteristics of the storage medium. For example (see Figure 4.5) there might be three levels of indexing:

track, cylinder and master. Each entry in the track index will contain enough information to locate the start of the track, and the key of the last record in the track which is also normally the highest value on that track. There is a track index for each cylinder. Each entry in the cylinder index gives the last record on each cylinder and the address of the track index for that cylinder. If the cylinder index itself is stored on tracks, then the master index will give the highest key referenced for each track of the cylinder index and the starting address of that track.

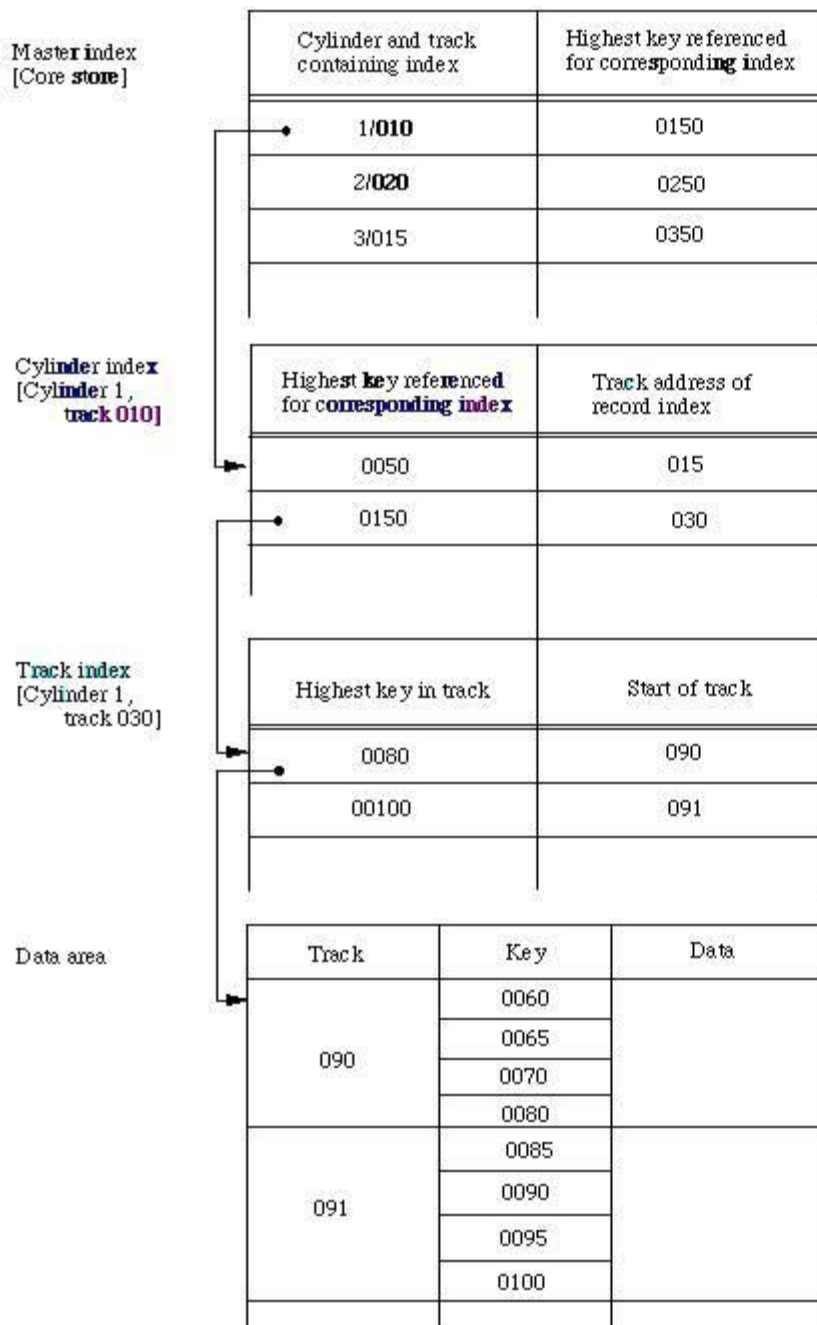


Figure 4.5: An example of an implementation of an index-sequential file (Adapted from D. R. Judd, *The Use of Files*, Macdonald and Elsevier, London and New York 1973, page 46.)

No mention has been made of the possibility of overflow during an updating process. Normally provision is made in the directory to administer an overflow area. This of course increases the number of book-keeping entries in each entry of the index.

### ***Multi-lists***

A multi-list is really only a slightly modified inverted file. There is one list per keyword, i.e.  $h_i = 1$ . The records containing a particular keyword  $K_i$  are chained together to form the  $K_i$ -list and the start of the  $K_i$ -list is given in the directory, as illustrated in *Figure 4.6*. Since there is no  $K_3$ -list, the field reserved for its pointer could well have been omitted. So could any blank pointer field, so long as no ambiguity arises as to which pointer belongs to which keyword. One way of ensuring this, particularly if the data values (attribute-values) are fixed format, is to have the pointer not pointing to the beginning of the record but pointing to the location of the next pointer in the chain.

The multi-list is designed to overcome the difficulties of updating an inverted file. The addresses in the directory of an inverted file are normally kept in record-number order. But, when the time comes to add a new record to the file, this sequence must be maintained, and inserting the new address can be expensive. No such problem arises with the multi-list, we update the appropriate  $K$ -lists by simply chaining in the new record. The penalty we pay for this is of course the increase in search time. This is in fact typical of many of the file structures. Inherent in their design is a trade-off between search time and update time.

### ***Cellular multi-lists***

A further modification of the multi-list is inspired by the fact that many storage media are divided into *pages*, which can be retrieved one at a time. A  $K$ -list may cross several page boundaries which means that several pages may have to be accessed to retrieve one record. A modified multi-list structure which avoids this is called a *cellular multi-list*. The  $K$ -lists are limited so that they will not cross the page (cell) boundaries.

At this point the full power of the notation introduced before comes into play. The directory for a cellular multi-list will be the set of sequences

$$(K_i, n_i, h_i, a_{i1}, \dots, a_{ih_i}) \quad i = 1, 2, \dots, m$$

where the  $h_i$  have been picked to ensure that a  $K_i$ -list does not cross a page boundary. In an implementation, just as in the implementation of an index-sequential file, further information will be stored with each address to enable the right page to be located for each key value.

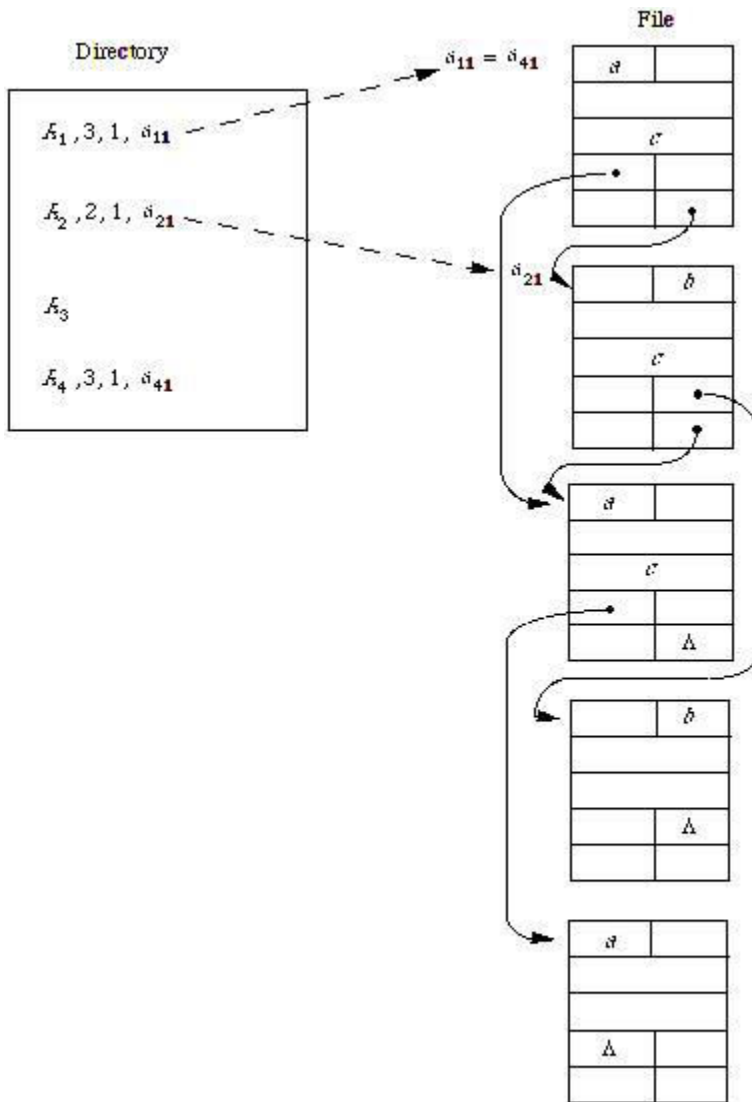


Figure 4.6. A multi-list

**Ring structures**

A *ring* is simply a linear list that closes upon itself. In terms of the definition of a *K*-list, the beginning and end of the list are the same record. This data-structure is particularly useful to show classification of data.

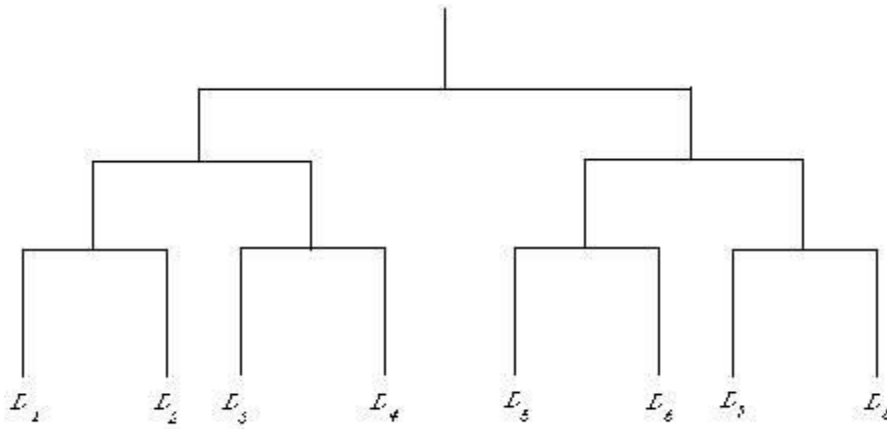


Figure 4.7. A dendrogram.

Let us suppose that a set of documents

$\{D_1, D_2, D_3, D_4, D_5, D_6, D_7, D_8\}$

has been classified into four groups, that is

$\{(D_1, D_2), (D_3, D_4), (D_5, D_6), (D_7, D_8)\}$

Furthermore these have themselves been classified into two groups,

$\{((D_1, D_2), (D_3, D_4)), ((D_5, D_6), (D_7, D_8))\}$

The dendrogram for this structure would be that given in Figure 4.7. To represent this in storage by means of ring structures is now a simple matter (see Figure 4.8).

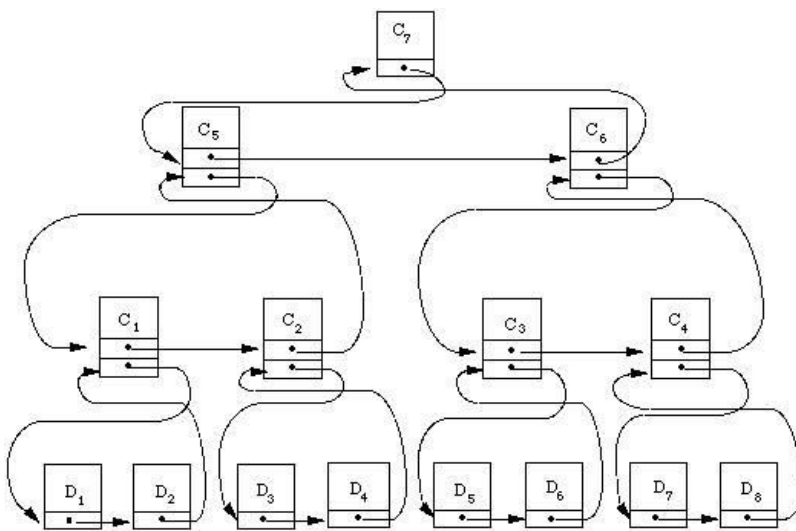


Figure 4.8. An implementation of a dendrogram via ring structure.

The  $D_i$  indicates a description (representation) of a document. Notice how the rings at a lower level are contained in those at a higher level. The field marked  $C_i$  normally contains some identifying information with respect to the ring it subsumes. For example,  $C_1$  in some way identifies the class of documents  $\{D_1, D_2\}$ .

Were we to group documents according to the keywords they shared, then for each keyword we would have a group of documents, namely, those which had that keyword in common.  $C_i$  would then be the field containing the keyword uniting that particular group. The rings would of course overlap (*Figure 4.9*), as in this example:

$$D_1 = \{K_1, K_2\}$$

$$D_2 = \{K_2, K_3\}$$

$$D_3 = \{K_1, K_4\}$$

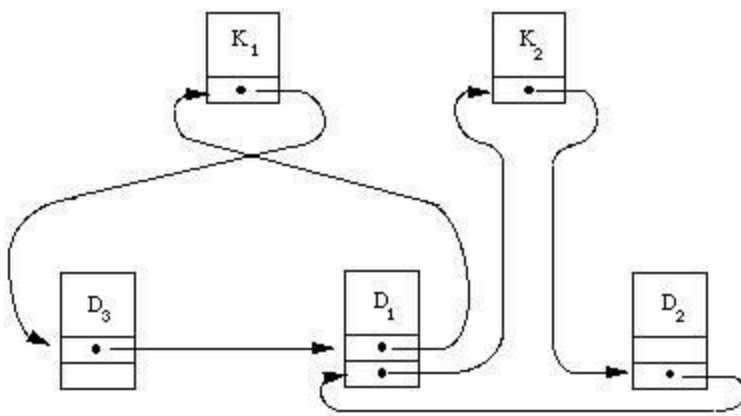


Figure 4.9. Two overlapping rings

The usefulness of this kind of structure will become more apparent when we discuss searching of classifications. If each ring has associated with it a record which contains identifying information for its members, then, a search strategy searching a structure such as this will first look at  $C_i$  (or  $K_i$  in the second example) to determine whether to proceed or abandon the search.

### Threaded lists

In this section an elementary knowledge of list processing will be assumed. Readers who are unfamiliar with this topic should consult the little book by Foster.

A simple list representation of the classification

$$((D_1, D_2), (D_3, D_4)), ((D_5, D_6), (D_7, D_8))$$

is given in *Figure 4.10*. Each sublist in this structure has associated with it a record containing *only* two pointers. (We can assume that  $D_i$  is really a pointer to document  $D_i$ .) The function of the pointers should be clear from the diagram. The main thing to note, however, is that the record associated with a list does *not* contain any identifying information.

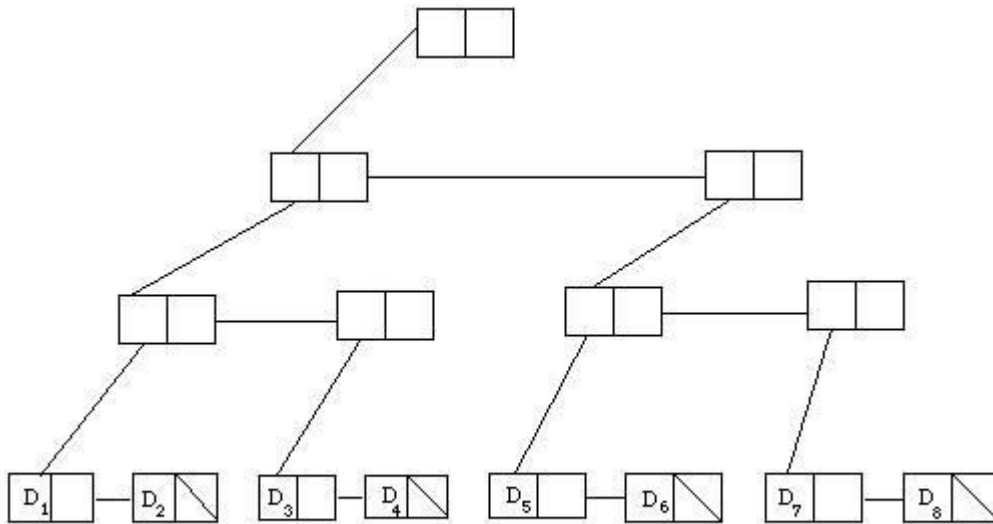


Figure 4.10. A list structure implementation of a hierarchical classification.

A modification of the implementation of a list structure like this which makes it resemble a set of ring structures is to make the right hand pointer of the *last* element of a sublist point back to the head of the sublist. Each sublist has become effectively a ring structure. We now have what is commonly called a *threaded list* (see Figure 4.11). The representation I have given is a slight oversimplification in that we need to flag which elements are data elements (giving access to the documents  $D_i$ ) and which elements are just pointer elements. The major advantage associated with a threaded list is that it can be traversed without the aid of a stack. Normally when traversing a conventional list structure the return addresses are stacked, whereas in the threaded list they have been incorporated in the data structure.

One disadvantage associated with the use of list and ring structures for representing classifications is that they can only be entered at the 'top'. An additional index giving entry to the structure at each of the data elements increases the update speed considerably.

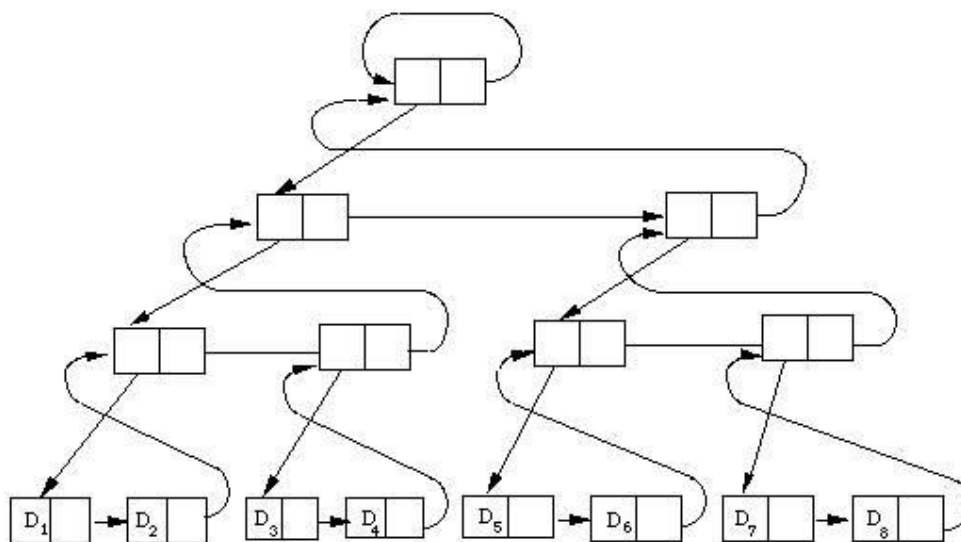
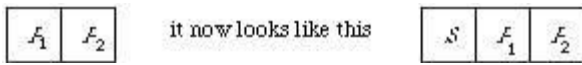


Figure 4.11. A threaded list implementation of a hierarchical classification

Another modification of the simple list representation has been studied extensively by Stanfel[21,22] and Patt[23]. The individual elements (or cells) of the list structure are modified to incorporate one extra field, so that instead of each element looking like this



where the  $P_i$ s are pointers and  $S$  is a symbol. Otherwise no essential change has been made to the simple representation. This structure has become known as the *Doubly Chained Tree*. Its properties have mainly been investigated for storing variable length keys, where each key is made up by selecting symbols from a finite (usually small) alphabet. For example, let  $\{A, B, C\}$  be the set of key symbols and let  $R_1, R_2, R_3, R_4, R_5$  be five records to be stored. Let us assign keys made of the 3 symbols, to the record as follows:

AAA R1

AB R2

AC R3

BB R4

BC R5

An example of a doubly chained tree containing the keys and giving access to the records is given in *Figure 4.12*. The topmost element contains no symbol, it merely functions as the start of the structure. Given an arbitrary key its presence or absence is detected by matching it against keys in the structure. Matching proceeds level by level, once a matching symbol has been found at one level, the  $P_1$  pointer is followed to the set of *alternative* symbols at the next level down. The matching will terminate either:

- (1) when the key is exhausted, that is, no more key symbols are left to match; or
- (2) when no matching symbol is found at the current level.

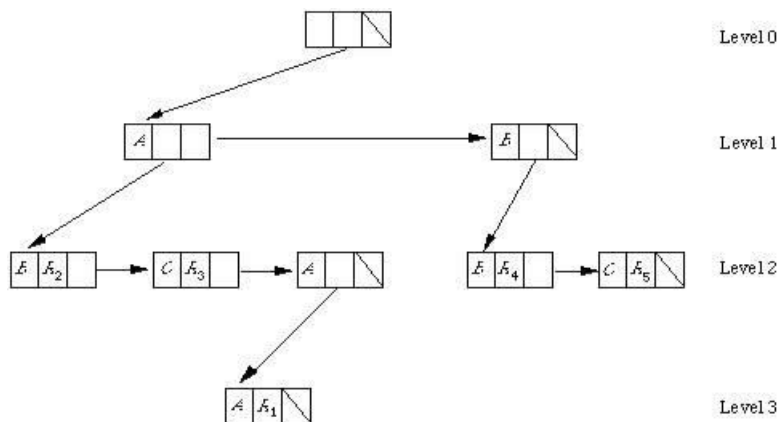


Figure 4.12. An example of a doubly chained tree



For case (1) we have:

(a) the key is present if the P1 pointer in the same cell as the last matching symbol

now points to a record;

(b) P1 points to a further symbol, that is, the key 'falls short' and is therefore not in the structure.

For case (2), we also have that the key is not in the structure, but now there is a mismatch.

Stanfel and Patt have concentrated on generating search trees with minimum expected search time, and preserving this property despite updating. For the detailed mathematics demonstrating that this is possible the reader is referred to their cited work.