Module1

1.1. Introduction to Data Structures:

Data structure is a representation of logical relationship existing between individual elements of data. In other words, a data structure defines a way of organizing all data items that considers not only the elements stored but also their relationship to each other. The term data structure is used to describe the way data is stored.

To develop a program of an algorithm we should select an appropriate data structure for that algorithm. Therefore, data structure is represented as: Algorithm + Data structure = Program

A data structure is said to be linear if its elements form a sequence or a linear list. The linear data structures like an array, stacks, queues and linked lists organize data in linear order. A data structure is said to be non linear if its elements form a hierarchical classification where, data items appear at various levels.

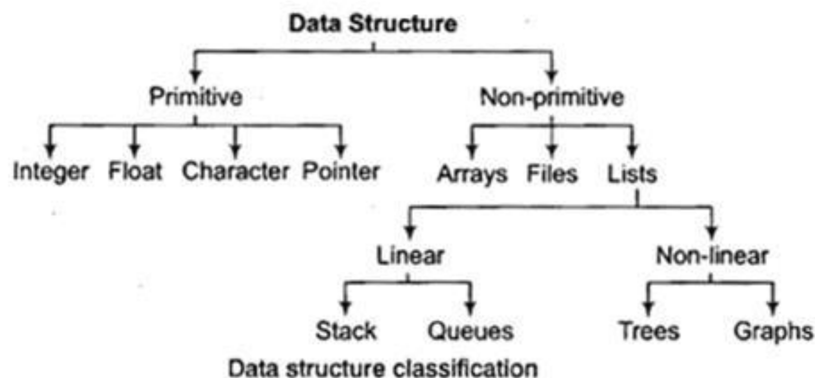
Trees and Graphs are widely used non-linear data structures. Tree and graph structures represent hierarchical relationship between individual data elements. Graphs are nothing but trees with certain restrictions removed.

Data structures are divided into two types:

- Primitive data structures.
- Non-primitive data structures.

Primitive Data Structures are the basic data structures that directly operate upon the machine instructions. They have different representations on different computers. Integers, floating point numbers, character constants, string constants and pointers come under this category.

Non-primitive data structures are more complicated data structures and are derived from primitive data structures. They emphasize on grouping same or different data items with relationship between each data item. Arrays, lists and files come under this category. Figure 1.1 shows the classification of data structures.



Data structure classification

Data Structures Operations:

1. Traversing
2. Searching
3. Inserting
4. Deleting
5. Sorting
6. Merging

1. **Traversing**- It is used to access each data item exactly once so that it can be processed.
2. **Searching**- It is used to find out the location of the data item if it exists in the given collection of data items.
3. **Inserting**- It is used to add a new data item in the given collection of data items.
4. **Deleting**- It is used to delete an existing data item from the given collection of data items.
5. **Sorting**- It is used to arrange the data items in some order i.e. in ascending or descending order in case of numerical data and in dictionary order in case of alphanumeric data.
6. **Merging**- It is used to combine the data items of two sorted files into single file in the sorted form.

Review of Arrays

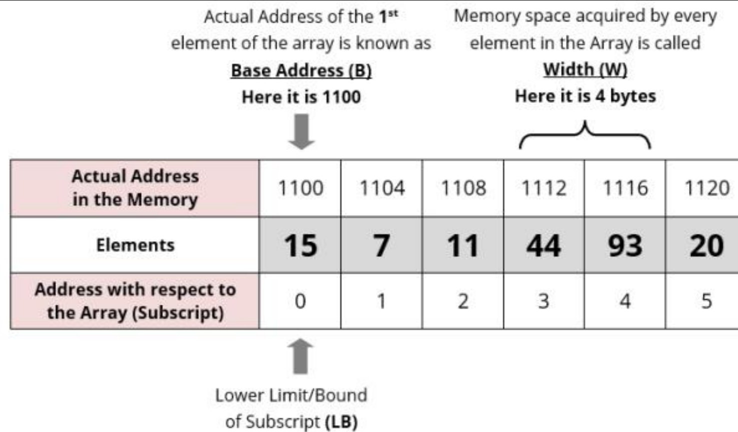
The simplest type of data structure is a linear array. This is also called one dimensional array.

Definition:

Array is a data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type. An array holds several values of the same kind. Accessing the elements is very fast. It may not be possible to add more values than defined at the start, without copying all values into a new array. An array is stored so that the position of each element can be computed from its index.

For example, an array of 10 integer variables, with indices 0 through 9, may be stored as 10 words at memory addresses 2000, 2004, 2008, 2036, so that the element with index i has the address $2000 + 4 \times i$.

Address Calculation in single (one) Dimension Array:



Array of an element of an array say "A[I]" is calculated using the following formula:

Address of A [I] = B + W * (I – LB)

Where,

B = Base address

W = Storage Size of one element stored in the array (in byte)

I = Subscript of element whose address is to be found

LB = Lower limit / Lower Bound of subscript, if not specified assume 0 (zero)

Example:

Given the base address of an array **B[1300…..1900]** as 1020 and size of each element is 2 bytes in the memory. Find the address of **B[1700]**.

Solution:

The given values are: B = 1020, LB = 1300, W = 2, I = 1700

Address of A [I] = B + W * (I – LB)

= 1020 + 2 * (1700 – 1300)

= 1020 + 2 * 400

= 1020 + 800

= 1820 **[Ans]**

CAMBRIDGE
INSTITUTE OF TECHNOLOGY

Address Calculation in Double (Two) Dimensional Array:

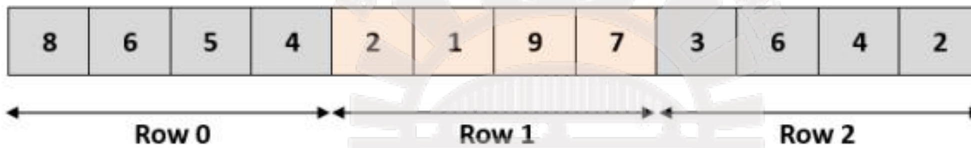
While storing the elements of a 2-D array in memory, these are allocated contiguous memory locations. Therefore, a 2-D array must be linearized so as to enable their storage. There are two alternatives to achieve linearization: Row-Major and Column-Major.

Column Index

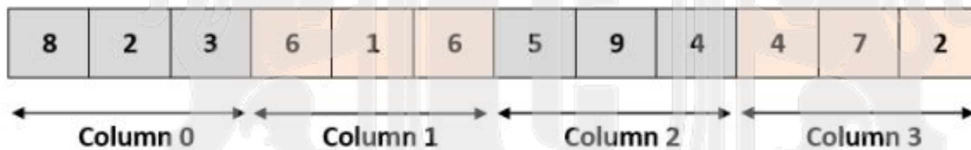
Row Index	0	1	2	3
0	**8**	**6**	**5**	**4**
1	**2**	**1**	**9**	**7**
2	**3**	**6**	**4**	**2**

Two-Dimensional Array

Row-Major (Row Wise Arrangement)



Column-Major (Column Wise Arrangement)



C allows for arrays of two or more dimensions. A two-dimensional (2D) array is an array of arrays. A three-dimensional (3D) array is an array of arrays of arrays.

In C programming an array can have two, three, or even ten or more dimensions. The maximum dimensions a C program can have depends on which compiler is being used.

More dimensions in an array means more data be held, but also means greater difficulty in managing and understanding arrays.

How to Declare a Multidimensional Array in C

A multidimensional array is declared using the following syntax:

type array_name[d1][d2][d3][d4]………[dn];

Where each **d** is a dimension, and **dn** is the size of final dimension.

Examples:

1. **int table[5][5][20];**
2. **float arr[5][6][5][6][5];**

In Example 1:

- **int** designates the array type integer.
- **table** is the name of our 3D array.
- Our array can hold 500 integer-type elements. This number is reached by multiplying the value of each dimension. In this case: **5x5x20=500**.

In Example 2:

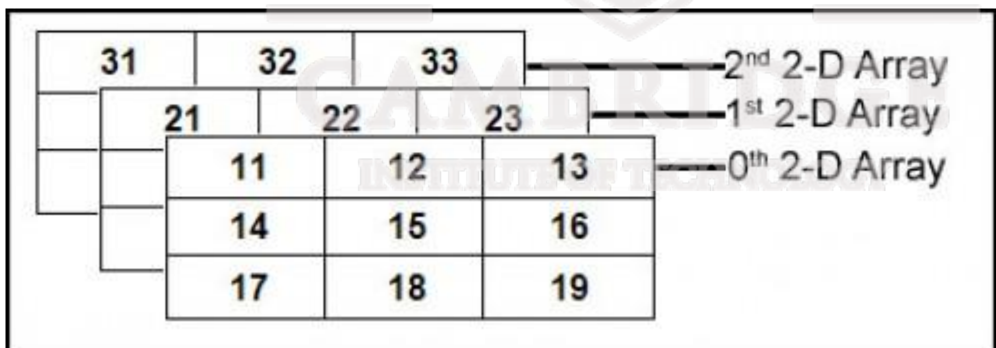
- Array **arr** is a five-dimensional array.
- It can hold 4500 floating-point elements (**5x6x5x6x5=4500**).

When it comes to holding multiple values, we would need to declare several variables. But a single array can hold thousands of values.

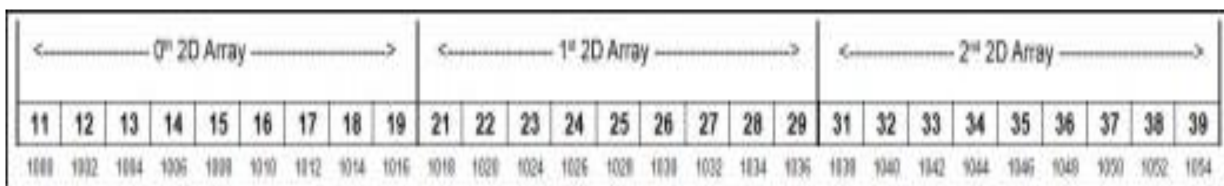
Explanation of a 3D Array

A 3D array is essentially an array of arrays of arrays: it's an array or collection of 2D arrays, and a 2D array is an array of 1D array.

The diagram below shows a 3D array representation:



3D Array Conceptual View



3D array memory map.

Example to show reading and printing a 3D array

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

void main()
{
int i, j, k;
int arr[3][3][3]=
        {
           {
           {11, 12, 13},
           {14, 15, 16},
           {17, 18, 19}
           },
           {
           {21, 22, 23},
           {24, 25, 26},
           {27, 28, 29}
           },
           {
           {31, 32, 33},
           {34, 35, 36},
           {37, 38, 39}
           },
        };
clrscr();
printf(":::3D Array Elements:::\n\n");
for(i=0;i<3;i++)
{
    for(j=0;j<3;j++)
    {
        for(k=0;k<3;k++)
        {
        printf("%d\t",arr[i][j][k]);
        }
        printf("\n");
    }
    printf("\n");
}
}
```

Note: refer notes for dynamic 1D & 2D arrays and also for pointers.

Prepared by
Prof Pankaja K. & Prof Geetha P.

Structures

Structure is a user defined data type that can hold data items of different data types.

Structure Declaration

struct tag { member 1; member 2;

----

member m; }

In this declaration, struct is a required keyword ,tag is a name that identifies structures of this type.

The individual members can be ordinary variables, pointers, arrays or other structures. The member names within a particular structure must be distinct from one another, though a member name can be same as the name of a variable defined outside of the structure and individual members cannot be initialized within a structure-type declaration. For example:

```
struct student

{ char name [80];

 int roll_no;

 float marks;

 }s1,s2;
```

we can now declare the structure variable s1 and s2 as follows: struct student s1, s2; where

s1 and s2 are structure type variables whose composition is identified by the tag student.

Nested Structure

It is possible to combine the declaration of the structure composition with that of the structure variable .It is then called a nested structure.

```
struct dob

{ int month;

int day;

int year;

 };

struct student
```

```
{ char name [80];
int roll_no;
float marks;
struct dob d1;
}st;
```

The member of the nested structure can be accessed by using the dot operator twice.(ie)st.d1.year

Array of structures

It is also possible to define an array of structures that is an array in which each element is a structure. The procedure is shown in the following example:

```
struct student{
char name [80];
int roll_no ;
float marks ;
} st [100];
```

In this declaration st is a 100- element array of structures.

It means each element of st represents an individual student record.

Accessing members of a structure using the dot operator

The members of a structure are usually processed individually, as separate entities. Therefore, we must be able to access the individual structure members. A structure member can be accessed by writing

structurevariable. member name.

This period (.) is an operator, it is a member of the highest precedence group, and its associativity is left-to-right.

e.g. if we want to print the detail of a member of a structure then we can write as

printf("%s",st.name); or printf("%d", st.roll_no) and so on. More complex expressions involving the repeated use of the period operator may also be written. For example, if a structure member is itself a structure, then a member of the embedded structure can be accessed by writing.

variable.member.submember.

Thus in the case of student and dob structure, to access the month of date of birth of a student, we would write

st.d1.month // accessing member of nested structure

The use of the period operator can be extended to arrays of structure, by writing

array [expression]. member

Structures members can be processed in the same manner as ordinary variables of the same data type. Single-valued structure members can appear in expressions. They can be passed to functions and they can be returned from functions, as though they were ordinary single-valued variables.

e.g. suppose that s1 and s2 are structure variables having the same composition as described earlier. It is possible to **copy the values of s1 to s2 simply by writing**

s2=s1; //copying one structure to another which are of the same type

USER-DEFINED DATA TYPES (typedef)

The typedef feature allows users to define new data types that are equivalent to existing data types. Once a user-defined data type has been established, then new variables, arrays, structure and so on, can be declared in terms of this new data type. In general terms, a new data type is defined as

typedef type new- type;

Where type refers to an existing data type and new-type refers to the new user-defined data type.

e.g. typedef int age;

In this declaration, age is user- defined data type equivalent to type int. Hence, the variable declaration

age male, female;

is equivalent to writing

int age, male, female;

The typedef feature is particularly convenient when defining structures, since it eliminates the need to repeatedly write struct tag whenever a structure is referenced. As a result, the structure can be referenced more concisely.

In general terms, a user-defined structure type can be written as

typedef struct { member 1; member 2: - - - - - - member m; }new-type;

The typedef feature can be used repeatedly, to define one data type in terms of other user-defined data types.

CAMBRIDGE
INSTITUTE OF TECHNOLOGY

STRUCTURES AND POINTERS

The beginning address of a structure can be accessed in the same manner as any other address, through the use of the address (&) operator.

Thus, if variable represents a structure type variable, then & variable represents the starting address of that variable. A pointer to a structure can be defined as follows:

struct student *ptr;

ptr represents the name of the pointer variable of type student. We can then assign the beginning address of a structure variable to this pointer by writing

ptr= &variable; //pointer initialisation

Let us take the following example:

```
typedef struct {

char name [ 40];

int roll_no;

float marks;

}student;

student s1,*ps;
```

In this example, s1 is a structure variable of type student, and ps is a pointer variable whose object is a structure variable of type student. Thus, the beginning address of s1 can be assigned to ps by writing.

ps = &s1;

An individual structure member can be accessed in terms of its corresponding pointer variable by using the -> operator (arrow operator)

ptr →member

Where ptr refers to a structure- type pointer variable and the operator → is comparable to the period (.) operator. The associativity of this operator is also left-to-right.

The operator → can be combined with the period operator (.) to access a submember within a structure. Hence, a submember can be accessed by writing

ptr → member.submember

PASSING STRUCTURES TO A FUNCTION

There are several different ways to pass structure–type information to or from a function. Structure member can be transferred individually , or entire structure can be transferred. The individual structures members can be passed to a function as arguments in the function call; and a single structure member can be returned via the return statement. To do so, each structure member is treated the same way as an ordinary, single- valued variable.

A complete structure can be transferred to a function by passing a structure type pointer as an argument. It should be understood that a structure passed in this manner will be passed by reference rather than by value. So, if any of the structure members are altered within the function, the alterations will be recognized outside the function. Let us consider the following example:

```
# include <stdio.h>
typedef struct{
char name[10];
int roll_no;
float marks ;
} record student={“Param”, 2,99.9};
void adj(record*ptr)
{
ptr → name=”Tanishq”;
ptr → roll_no=3;
ptr → marks=98.0;
return;
}
main ( )
{printf(“%s%d%f\n”, student.name, student.roll_no,student.marks);
adj(&student);
printf(“%s%d%f\n”, student.name, student.roll_no,student.marks);
}
```

Let us consider an example of transferring a complete structure, rather than a structure-type pointer, to the function.

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

```
typedef struct{
char name[10];
int roll_no;
float marks;
}record student={"Param," 2,99.9};
void adj(record stud) /*function definition */
{
stud.name="Tanishq";
stud.roll_no=3;
stud.marks=98.0;
return;
}
main()
{
printf("%s%d%f\n", student.name,student.roll_no,student.marks);
adj(student);
printf("%s%d%f\n", student.name,student.roll_no,student.marks);
}
```

Union is a derived datatype , like structure, i.e. collection of elements of different data types which are grouped together. Each element in a union is called member.

- Union and structure in C are same in concepts, except allocating memory for their members.
- Structure allocates storage space for all its members separately.
- Whereas, Union allocates one common storage space for all its members, or memory space is shared between its members.

- Only one member of union can be accessed at a time. All member values cannot be accessed at the same time in union. But, structure can access all member values at the same time. This is because, Union allocates one common storage space for all its members, where as Structure allocates storage space for all its members separately.
- Many union variables can be created in a program and memory will be allocated for each union variable separately.

- The table below will help you how to form a C union, declare a union, initializing and accessing the members of the union.

Type	Using normal variable	Using pointer variable

Syntax	union tag_name
	{
	data type var_name1;
	data type var_name2;
	data type var_name3;
	};
		union tag_name
		{
		data type var_name1;
		data type var_name2;
		data type var_name3;
		};

Example	union student
	{
	int mark;
	char name[10];
	float average;
	};
		union student
		{
		int mark;
		char name[10];
		float average;
		};

Declaring union variable	union student report;	union student *report, rep;
Initializing union variable	union student report = {100, “Mani”, 99.5};	union student rep = {100, “Mani”, 99.5};report = &rep;
Accessing union members	report.mark report.name report.average	report -> mark report -> name report -> average

Example program for C union:

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

union student
{
          char name[20];
          char subject[20];
          float percentage;
};

int main()
{
   union student record1;
   union student record2;

   // assigning values to record1 union variable
     strcpy(record1.name, "Raju");
     strcpy(record1.subject, "Maths");
     record1.percentage = 86.50;

     printf("Union record1 values example\n");
     printf(" Name       : %s \n", record1.name);
     printf(" Subject    : %s \n", record1.subject);
```

CAMBRIDGE
INSTITUTE OF TECHNOLOGY

```
    printf(" Percentage : %f \n\n", record1.percentage);

  // assigning values to record2 union variable
    printf("Union record2 values example\n");
    strcpy(record2.name, "Mani");
    printf(" Name       : %s \n", record2.name);

    strcpy(record2.subject, "Physics");
    printf(" Subject    : %s \n", record2.subject);

    record2.percentage = 99.50;
    printf(" Percentage : %f \n", record2.percentage);
    return 0;
}
```

Output:

```
Union record1 values example
Name :
Subject :
Percentage : 86.500000;
Union record2 values example
Name : Mani
Subject : Physics
Percentage : 99.500000
```

Explanation for above C union program:

There are 2 union variables declared in this program to understand the difference in accessing values of union members.

Record1 union variable:

- "Raju" is assigned to union member "record1.name" . The memory location name is "record1.name" and the value stored in this location is "Raju".
- Then, "Maths" is assigned to union member "record1.subject". Now, memory location name is changed to "record1.subject" with the value "Maths" (Union can hold only one member at a time).
- Then, "86.50" is assigned to union member "record1.percentage". Now, memory location name is changed to "record1.percentage" with value "86.50".
- Like this, name and value of union member is replaced every time on the common storage space.
- So, we can always access only one union member for which value is assigned at last. We can't access other member values.
- So, only "record1.percentage" value is displayed in output. "record1.name" and "record1.percentage" are empty.

Record2 union variable:

- If we want to access all member values using union, we have to access the member before assigning values to other members as shown in record2 union variable in this program.
- Each union members are accessed in record2 example immediately after assigning values to them.
- If we don't access them before assigning values to other member, member name and value will be over written by other member as all members are using same memory.
- We can't access all members in union at same time but structure can do that.

Example program – Another way of declaring C union:

In this program, union variable "record" is declared while declaring union itself as shown in the below program.

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

union student
{
        char name[20];
        char subject[20];
        float percentage;
}record;

int main()
{

        strcpy(record.name, "Raju");
        strcpy(record.subject, "Maths");
        record.percentage = 86.50;

        printf(" Name       : %s \n", record.name);
        printf(" Subject    : %s \n", record.subject);
        printf(" Percentage : %f \n", record.percentage);
        return 0;
}
```

Output:

```
Name :
Subject :
Percentage : 86.500000
```


BRIDGE TO THE FUTURE
CAMBRIDGE
INSTITUTE OF TECHNOLOGY

We can access only one member of union at a time. We can't access all member values at the same time in union. But, structure can access all member values at the same time. This is because, Union allocates one common storage space for all its members. Where as Structure allocates storage space for all its members separately.

Difference between structure and union in C:

S.no	C Structure	C Union
1	Structure allocates storage space for all its members separately.	Union allocates one common storage space for all its members. Union finds that which of its member needs high storage space over other members and allocates that much space
2	Structure occupies larger memory space.	Union occupies lower memory space over structure.
3	We can access all members of structure at a time.	We can access only one member of union at a time.

4	Structure example:
	struct student
	{
	int mark;
	double average;
	};
		Union example:
		union student
		{
		int mark;
		double average;
		};

5	For above structure, memory allocation will be like below. int mark – 2B double average – 8B Total memory allocation = 2+8 = 10 Bytes	For above union, only 8 bytes of memory will be allocated since double data type will occupy maximum space of memory over other data types. Total memory allocation = 8 Bytes

POINTERS & DYNAMIC MEMORY ALLOCATION FUNCTIONS:

When a variable is defined the compiler (linker/loader actually) allocates a real memory address for the variable.

- int x; will allocate 4 bytes in the main memory, which will be used to store an integer value.

When a value is assigned to a variable, the value is actually placed to the memory that was allocated.

- x=3; will store integer 3 in the 4 bytes of memory.

The process of allocating memory during program execution is called dynamic memory allocation.

C language offers 4 dynamic memory allocation functions. They are,

1. malloc() : malloc (number * sizeof(int));
2. calloc() : calloc (number, sizeof(int));
3. realloc() : realloc (pointer_name, number * sizeof(int));
4. free() : free (pointer_name);

1. MALLOC():

→ is used to allocate space in memory during the execution of the program.

→does not initialize the memory allocated during execution. It carries garbage value.

→returns null pointer if it couldn't able to allocate requested amount of memory.

2. CALLOC():

→ calloc () function is also like malloc () function. But calloc () initializes the allocated memory to zero. But, malloc() doesn't.

3. REALLOC():

→ Realloc () function modifies the allocated memory size by malloc () and calloc () functions to new size.

→If enough space doesn't exist in memory of current block to extend, new block is allocated for the full size of reallocation, then copies the existing data to new block and then frees the old block.

4. FREE():

→free () function frees the allocated memory by malloc (), calloc (), realloc () functions and returns the memory to the system.

DIFFERENCE BETWEEN STATIC MEMORY ALLOCATION AND DYNAMIC MEMORY ALLOCATION IN C:

Static memory allocation	**Dynamic memory allocation**
In static memory allocation, memory is allocated while writing the C program. Actually, user requested memory will be allocated at compile time.	In dynamic memory allocation, memory is allocated while executing the program. That means at run time.
Memory size can't be modified while execution. Example: array	Memory size can be modified while execution. Example: Linked list

Prepared by
Prof Pankaja K. & Prof Geetha P.

DIFFERENCE BETWEEN MALLOC() AND CALLOC() FUNCTIONS IN C:

malloc()	calloc()
It allocates only single block of requested memory	It allocates multiple blocks of requested memory
int *ptr;ptr = malloc(20 * sizeof(int)); For the above, 20*4 bytes of memory only allocated in one block. Total = 80 bytes	int *ptr;Ptr = calloc(20, 20 * sizeof(int));For the above, 20 blocks of memory will be created and each contains 20*4 bytes of memory. Total = 1600 bytes
malloc () doesn't initializes the allocated memory. It contains garbage values	calloc () initializes the allocated memory to zero
type cast must be done since this function returns void pointer int *ptr;ptr = (int*)malloc(sizeof(int)*20);	Same as malloc () function int *ptr;ptr = (int*)calloc(20, 20 * sizeof(int));

Array Operations: All operations remain same as mentioned above for data structures operations.

Note: → Refer 1st lab program for the operations.

SORTING:

Sorting takes an unordered collection and makes it an ordered one.

In bubble sort method the list is divided into two sub-lists sorted and unsorted. The smallest element is bubbled from unsorted sub-list. After moving the smallest element the imaginary wall moves one element ahead. The bubble sort was originally written to bubble up the highest element in the list. But there is no difference whether highest / lowest element is bubbled. This method is easy to understand but time consuming. In this type, two successive elements are compared and swapping is done. Thus, step-by-step entire array elements are checked. Given a list of 'n' elements the bubble sort requires up to n-1 passes to sort the data.

Algorithm for Bubble sort: Bubble_Sort(A[], N)

```
Step1  :  Repeat for p = 1 to N-1
          Begin
Step2  :       Repeat for j = 1 to N-p
               Begin
Step3  :             if (A[j] < A[j-1])
                         Swap ( A[j], A[j-1]);
               End for
          End for
Exit
```

Example:

Ex:- A list of unsorted elements are: 10 47 12 54 19 23 (Bubble up for highest value shown here)

Original List	After Pass 1	After Pass 2	After Pass 3	After Pass 4	After Pass 5
10	54	54	54	54	54
47	10	47	47	47	47
12	47	10	23	23	23
54	12	23	10	19	19
19	23	12	19	10	12
23	19	19	12	12	10

```
/* bubble sort implementation  */

#include<stdio.h>
#include<conio.h>
void main()
{
 int i,n,temp,j,arr[25];
 printf("Enter the number of elements in the Array: ");
 scanf("%d",&n);
 printf("\nEnter the elements:\n\n");
 for(i=0 ; i<n ; i++)
     scanf("%d",&arr[i]);

 for(i=0 ; i<n ; i++)
 {
   for(j=0 ; j<n-i-1 ; j++)
  {
      if(arr[j]>arr[j+1]) //Swapping Condition is Checked
      {
             temp=arr[j];
             arr[j]=arr[j+1];
            arr[j+1]=temp;
       }
    }
 }
 printf("\nThe Sorted Array is:\n\n");
```

CAMBRIDGE
INSTITUTE OF TECHNOLOGY

Prepared by
Prof Pankaja K. & Prof Geetha P.

```
for(i=0 ; i<n ; i++)
   printf(" %4d",arr[i]);
 }
```

SEARCHING TECHNIQUE

1. LINEAR SEARCH

Linear search or sequential search is a method for finding a target value within a list. It sequentially checks each element of the list for the target value until a match is found or until all the elements have been searched.

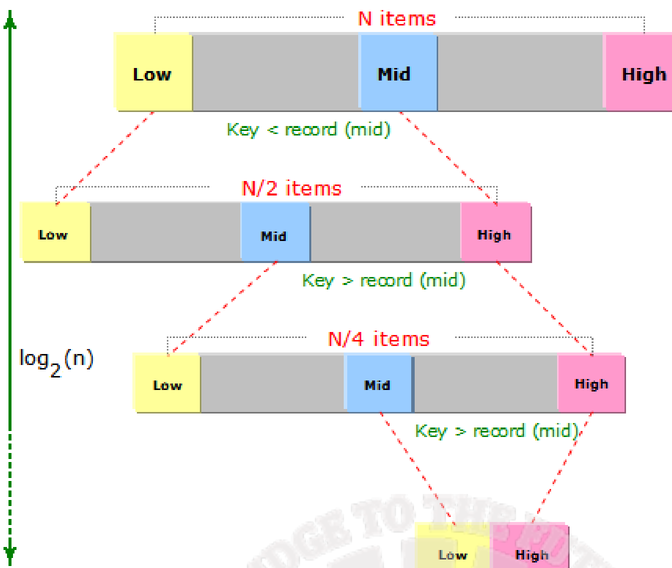
Linear search runs in at worst linear time and makes at most *n* comparisons, where *n* is the length of the list. If each element is equally likely to be searched, then linear search has an average case of n/2 comparisons, but the average case can be affected if the search probabilities for each element vary. Linear search is rarely practical because other search algorithms and schemes, such as the binary search algorithm and hash tables, allow significantly faster searching for all but short lists.

```
int linear_search(int *list, int size, int key, int* rec )
{
    // Basic Linear search
    int found = 0;
    int i;
    for ( i = 0; i < size; i++ )
         {
             if ( key == list[i] )
             found = 1;
             return found;
          }
      Return found;
 }
```

2. BINARY SEARCH

We always get a sorted list before doing the binary search. Now suppose we have an ascending order record. At the time of search it takes the middle record/element, if the searching element is greater than middle element then the element mush be located in the second part else it is in the first half. In this way this search algorithm divides the records in the two parts in each iteration and thus called binary search.

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




```
int BinarySearch(int *array, int N, int key)
{
        int low = 0, high = N-1, mid;
        while(low <= high)
        {
                mid = (low + high)/2;
                if(array[mid] < key)
                        low = mid + 1;
                else if(array[mid] == key)
                         return mid;
                else if(array[mid] > key)
                        high = mid-1;
           }
        return -1;
}
```

SPARSE MATRICES

A **sparse matrix** is a matrix in which most of the elements are zero. By contrast, if most of the elements are nonzero, then the matrix is considered **dense.** When storing and manipulating sparse matrices on a computer, it is beneficial and often necessary to use specialized algorithms and data structures that take advantage of the sparse structure of the matrix. Operations using standard dense-matrix structures and algorithms are slow and inefficient when applied to large sparse matrices as processing and memory are wasted on the zeroes. Sparse data is by nature more easily compressed and thus require significantly less storage. Some very large sparse matrices are infeasible to manipulate using standard dense-matrix algorithms.

Storing a sparse matrix

A matrix is typically stored as a two-dimensional array. Each entry in the array represents an element $a_{i,j}$ of the matrix and is accessed by the two indices i and j. Conventionally, i is the row index, numbered from top to bottom, and j is the column index, numbered from left to right. For an $m \times n$ matrix, the amount of memory required to store the matrix in this format is proportional to $m \times n$

In the case of a sparse matrix, substantial memory requirement reductions can be realized by storing only the non-zero entries.

Sparse Matrix Representations

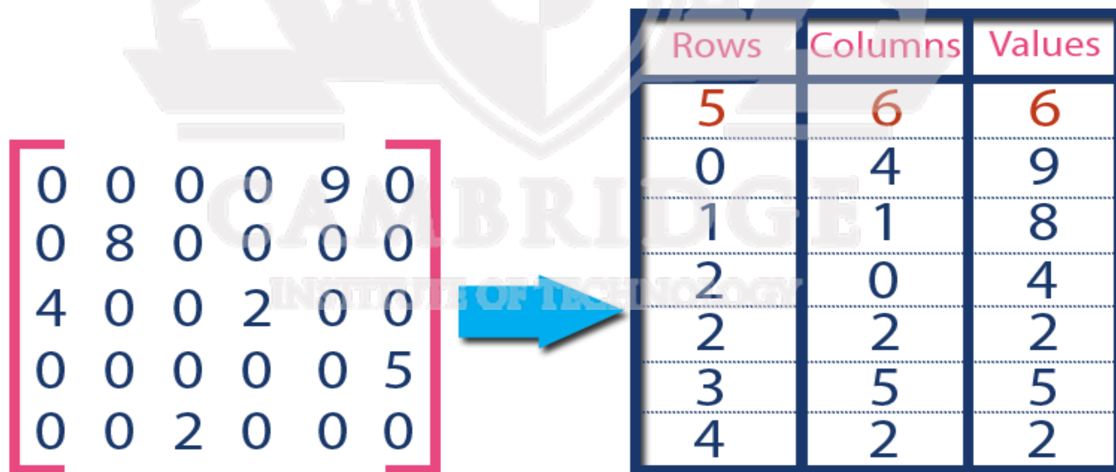
A sparse matrix can be represented by using TWO representations...

1. Triplet Representation
2. Linked Representation

1. Triplet Representation

In this representation, we consider only non-zero values along with their row and column index values. Each non zero value is a triplet of the form <R,C,Value) where R represents the row in which the value appears, C represents the column in which the value appears and Value represents the non-zero value itself. In this representation, the 0th row stores total rows, total columns and total non-zero values in the matrix.

For example, consider a matrix of size 5 X 6 containing 6 number of non-zero values. This matrix can be represented as shown in the image...



Rows	Columns	Values
5	6	6
0	4	9
1	1	8
2	0	4
2	2	2
3	5	5
4	2	2

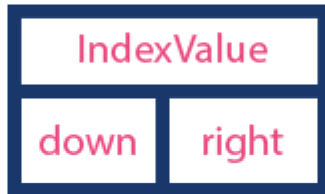
In above example matrix, there are only 6 non-zero elements (those are 9, 8, 4, 2, 5 & 2) and matrix size is 5 X 6. We represent this matrix as shown in the above image. Here the first row in the right side table is filled with values 5, 6 & 6 which indicates that it is a sparse matrix with 5 rows, 6 columns & 6 non-zero values. Second row is filled with 0, 4, & 9 which indicates the value in the matrix at 0th row, 4th column is 9. In the same way the remaining non-zero values also follows the similar pattern.

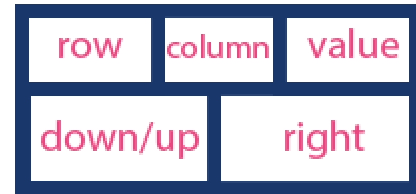
Prepared by
Prof Pankaja K. & Prof Geetha P.

BRIDGE TO THE FUTURE CAMBRIDGE INSTITUTE OF TECHNOLOGY

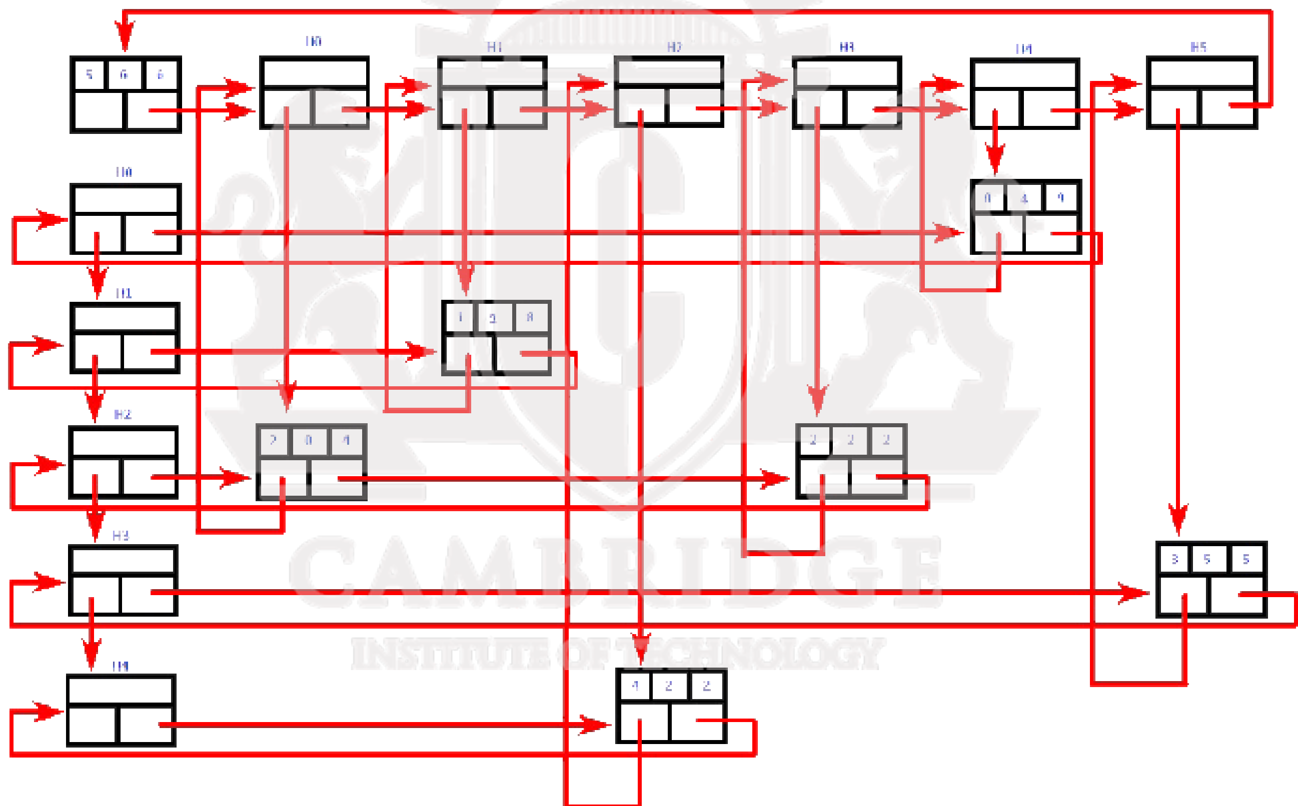
2. Linked Representation

In linked representation, we use linked list data structure to represent a sparse matrix. In this linked list, we use two different nodes namely **header node** and **element node**. Header node consists of three fields and element node consists of five fields as shown in the image...





Consider the above same sparse matrix used in the Triplet representation. This sparse matrix can be represented using linked representation as shown in the below image...



In above representation, H0, H1,...,H5 indicates the header nodes which are used to represent indexes. Remaining nodes are used to represent non-zero elements in the matrix, except the very first node which is used to represent abstract information of the sparse matrix (i.e., It is a matrix of 5 X 6 with 6 non-zero elements).

In this representation, in each row and column, the last node right field points to it's respective header node.

Prepared by
Prof Pankaja K. & Prof Geetha P.

Basic operations on Sparse Matrix

- Reading a sparse matrix
- Displaying a sparse matrix
- Searching for a non zero element in a sparse matrix

Note :Refer class notes for implementation of basic operations of sparse matrices

POLYNOMIALS

A *polynomial* object is a homogeneous ordered list of pairs < *exponent*,*coefficient*>, where each coefficient is unique.

Operations include returning the degree, extracting the coefficient for a given exponent, addition, multiplication, evaluation for a given input

Polynomial operations

- Representation
- Addition
- Multiplication

Representation of a Polynomial: 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

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

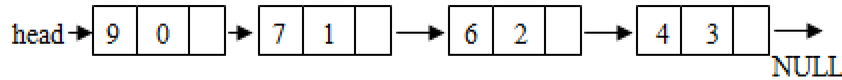
A polynomial thus may be represented using arrays or linked lists. Array representation assumes that the exponents of the given expression are arranged from 0 to the highest value (degree), which is represented by the subscript of the array beginning with 0. The coefficients of the respective exponent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

Refer notes for memory representation of polynomial

STRINGS

❖ **Strings are Character Arrays**

Strings in C are simply array of characters.

– Example: char s [10]; This is a ten (10) element array that can hold a character string consisting of ≤ 9 characters. This is because C does not know where the end of an array is at run time. By convention, C uses a NULL character '\0' to terminate all strings in its library functions

– For example: char str [10] = {'u', 'n', 'I', 'x', '\0'};

It's the string terminator (not the size of the array) that determines the length of the string.

❖ **Accessing Individual Characters**

The first element of any array in C is at index 0. The second is at index 1, and so on...

char s[10];

s[0] = 'h';

s[1] = 'i';

s[2] = '!';

s[3] = '\0';

s [0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
H	i	!	\0						

This notation can be used in all kinds of statements and expressions in C:

For example:

```
c = s[1];
if (s[0] == '-') …
switch (s[1]) ...
```

- **String Literals**

String literals are given as a string quoted by double quotes.

– printf("Long long ago.");

Initializing char array ...

– char s[10]="unix"; /* s[4] is '\0'; */

– char s[ ]="unix"; /* s has five elements */

– Printing with ptintf()

Example:

```
Char str[ ] = "A message to display";
printf ("%s\n", str);
```

printf expects to receive a string as an additional parameter when it sees %s in the format string

– Can be from a character array.

– Can be another literal string.

– Can be from a character pointer (more on this later).

– printf knows how much to print out because of the NULL character at the end of all strings.

– When it finds a \0, it knows to stop.

- **Printing with puts( )**

The puts function is a much simpler output function than printf for string printing.

– Prototype of puts is defined in stdio.h

int puts(const char * str). This is more efficient than printf because your program doesn't need to analyze the format string at run-time.

For example:

```
char sentence[] = "The quick brown fox\n";
puts(sentence);
```

Prints out: The quick brown fox

- **Inputting Strings with gets( )**

gets( ) gets a line from the standard input.

The prototype is defined in stdio.h

char *gets(char *str)

– str is a pointer to the space where gets will store the line to, or a character array.

– Returns NULL upon failure. Otherwise, it returns str.

```
char your_line[100];
printf("Enter a line:\n");
gets(your_line);
puts("Your input follows:\n");
puts(your_line);
```

You can overflow your string buffer, so be careful!

- **Inputting Strings with scanf ( )**

To read a string include:

- %s scans up to but not including the “next” white space character
- %***n***s scans the next ***n*** characters or up to the next white space character, whichever comes first

Example:

```
scanf ("%s%s%s", s1, s2, s3);
scanf ("%2s%2s%2s", s1, s2, s3);
```

- Note: No ampersand(&) when inputting strings into character arrays! (We’ll explain why later …)

Difference between gets

- gets( ) read a line
- scanf("%s",…) read up to the next space

Example:

```
#include <stdio.h>
int main () {
        char lname[81], fname[81];
        int  count, id_num;
        puts ("Enter the last name, firstname, ID number separated");
        puts ("by spaces, then press Enter \n");
        count = scanf ("%s%s%d", lname, fname,&id_num);
        printf ("%d items entered: %s %s %d\n",
                        count,fname,lname,id_num);
        return 0;
}
```

- **The C String Library**

String functions are provided in an ANSI standard string library.

- Access this through the include file:

#include <string.h>

- Includes functions such as:
  - Computing length of string
  - Copying strings
  - Concatenating strings

This library is guaranteed to be there in any ANSI standard implementation of C.

- strlen() returns the length of a NULL terminated character string:

strlen (char * str) ; Defined in string.h

A type defined in string.h that is equivalent to an unsigned int

char *str

Points to a series of characters or is a character array ending with '\0'

- strcpy() copying a string comes in the form:

  char *strcpy (char * destination, char * source);

  A copy of source is made at destination. Source should be NULL terminated and destination should have enough room (its length should be at least the size of source). The return value also points at the destination.

- strcat() comes in the form:

  char * strcat (char * str1, char * str2);

  Appends a copy of str2 to the end of str1 & a pointer equal to str1 is returned. Ensure that str1 has sufficient space for the concatenated string!

  Array index out of range will be the most popular bug in your C programming career.

Example:

```
#include <string.h>
#include <stdio.h>
int main() {
   char str1[27] = "abc";
   char str2[100];
   printf("%d\n",strlen(str1));
   strcpy(str2,str1);
   puts(str2);
   puts("\n");
   strcat(str2,str1);
        puts(str2);
}
```

- strcmp() can be compared for equality or inequality

  If they are equal - they are ASCII identical

  If they are unequal the comparison function will return an int that is interpreted as:

  < 0 : str1 is less than str2

  0 : str1 is equal to str2

  > 0 : str1 is greater than str2

Four basic comparison functions:

int strcmp (char *str1, char *str2) ;

- ❖ Does an ASCII comparison one char at a time until a difference is found between two chars
  - – Return value is as stated before

- If both strings reach a '\0' at the same time, they are considered equal.

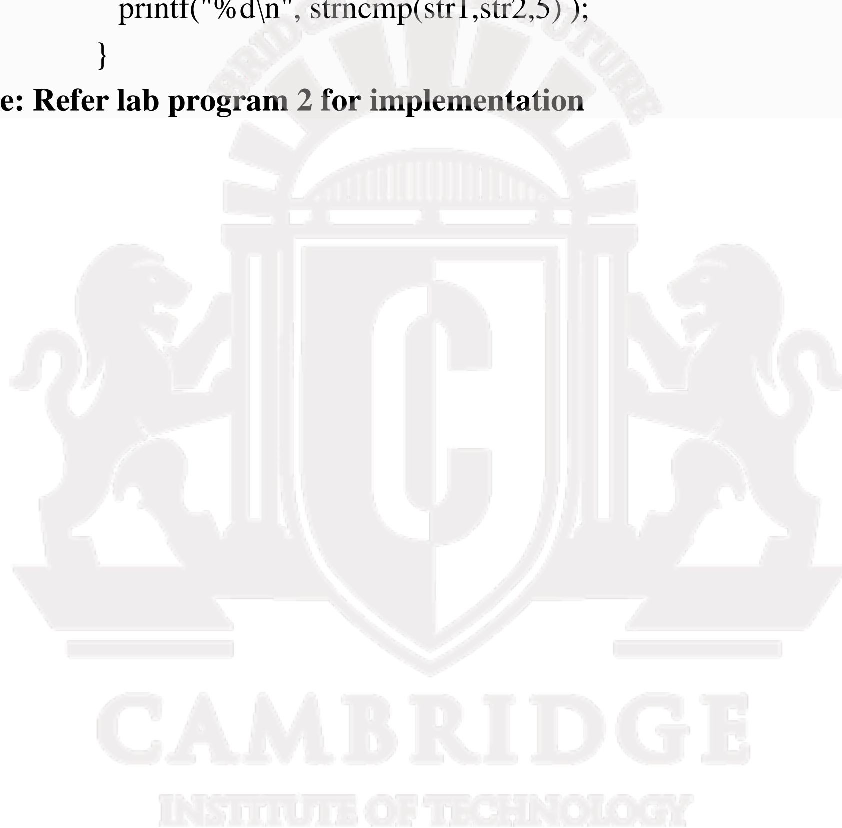
int strncmp (char *str1, char * str2, size_t n);

- Compares n chars of str1 and str2
  - Continues until n chars are compared or
  - The end of str1or str2 is encountered

Example:

```
int main() {
        char str1[] = "The first string.";
        char str2[] = "The second string.";
        size_t  n, x;
        printf("%d\n", strncmp(str1,str2,4) );
        printf("%d\n", strncmp(str1,str2,5) );
    }
```

Note: Refer lab program 2 for implementation



Prepared by
Prof Pankaja K. & Prof Geetha P.

MODULE 2
STACKS AND QUEUES

There are certain situations in computer science that one wants to restrict insertions and deletions so that they can take place only at the beginning or the end of the list, not in the middle. Two of such data structures that are useful are:

• Stack. • Queue.

Linear lists and arrays allow one to insert and delete elements at any place in the list i.e., at the beginning, at the end or in the middle.

STACK:

A stack is a list of elements in which an element may be inserted or deleted only at one end, called the top of the stack. Stacks are sometimes known as LIFO (last in, first out) lists. As the items can be added or removed only from the top i.e. the last item to be added to a stack is the first item to be removed.

The two basic operations associated with stacks are:

• **Push**: is the term used to insert an element into a stack.
• **Pop**: is the term used to delete an element from a stack.

.

All insertions and deletions take place at the same end, so the last element added to the stack will be the first element removed from the stack. When a stack is created, the stack base remains fixed while the stack top changes as elements are added and removed. The most accessible element is the top and the least accessible element is the bottom of the stack.

Representation of Stack:

Let us consider a stack with N elements capacity. This is called as the size of the stack. The number of elements to be added should not exceed the maximum size of the stack. If we attempt to add new element beyond the maximum size, we will encounter a stack overflow condition. Similarly, you cannot remove elements beyond the base of the stack. If such is the case, we will reach a stack underflow condition. (an empty stack)

When an element is added to a stack, the operation is performed by push. The removal of an element is performed by the pop operation.

Operations on Stack

	Stack's contents	TOP value	Output
1. Init_stack( )	<empty>	-1	
2. Push('a')	a	0	
3. Push('b')	a b	1	
4. Push('c')	a b c	2	
5. Pop( )	a b	1	c
6. Push('d')	a b d	2	c
7. Push('e')	a b d e	3	c
8. Pop( )	a b d	2	c e
9. Pop( )	a b	1	c e d
10. Pop( )	a	0	c e d b
11. Pop( )	<empty>	-1	c e d b a

	Push(a)	Push(b)	Push(c)	Pop( )	Push(d)	Push(e)	Pop( )	Pop( )	Pop( )	
						e				
			c		d	d	d			
		b	b	b	b	b	b	b		
	a	a	a	a	a	a	a	a	a	

Stack Using Array

A stack data structure can be implemented using one dimensional array. But stack implemented using array, can store only fixed number of data values. This implementation is very simple, just define a one dimensional array of specific size and insert or delete the values into that array by using **LIFO principle** with the help of a variable **'top'**. Initially top is set to -1. Whenever an element is to be inserted into the stack, increment the top value by one and then insert. Whenever an element is to be deleted from the stack the stack, then delete the top value and decrement the top value by one.

Stack Implementation

- Using static arrays
- Using dynamic arrays
- Using linked list

Stack Operations using Array

A stack can be implemented using array as follows...

push(value) - Inserting value into the stack

In a stack, push() is a function used to insert an element into the stack. In a stack, the new element is always inserted at **top** position. Push function takes one integer value as parameter and inserts that value into the stack. We can use the following algorithm to push an element on to the stack...

Step 1: Check whether **stack** is **FULL**. (**top == SIZE-1**)
Step 2: If it is **FULL**, then display **"Stack is FULL!!! Stack overflow!!!"** and terminate the function.
Step 3: If it is **NOT FULL**, then increment **top** value by one (**top**++) and set stack[top] to value (**stack[top] = value**).

pop() - Delete a value from the Stack

In a stack, pop() is a function used to delete an element from the stack. In a stack, the element is always deleted from **top** position. Pop function does not take any value as parameter. We can use the following algorithm to pop an element from the stack...

Step 1: Check whether **stack** is **EMPTY**. (**top == -1**)
Step 2: If it is **EMPTY**, then display **"Stack is EMPTY!!! Deletion is not possible!!!"** and terminate the function.
Step 3: If it is **NOT EMPTY**, then delete **stack[top]** and decrement **top** value by one (**top--**).

display() - Displays the elements of a Stack

We can use the following algorithm to display the elements of a stack...

Step 1: Check whether **stack** is **EMPTY**. (**top == -1**)
Step 2: If it is **EMPTY**, then display **"Stack is EMPTY!!!"** and terminate the function.
Step 3: If it is **NOT EMPTY**, then define a variable '**i**' and initialize with top. Display **stack[i]** value and decrement **i** value by one (**i--**).
Step 3: Repeat above step until **i** value becomes '0'.

Implementation of stack operations using array

```
# define N  5
int a[N],tos=-1;

void push()
{
 int key;
 if (tos==N-1)
   {
    printf("stack full\n");
    return;
   }
    printf("enter the elemnt to be inserted\n");
    scanf("%d",&key);
    a[++tos]=key;
 }

 void pop()
 {
  if (tos==-1)
   printf("underflow\n");
   else
    {
     printf("the popped element is %d",a[tos]);
     tos--;
    }
   }

   void display()
   {
    int i;
    if (tos== -1)
     {
     printf("no elements in stack\n");
     return;
     }
      for(i=tos;i>=0;i--)
       printf("%d\t",a[i]);
     }
```

Stack using dynamic arrays

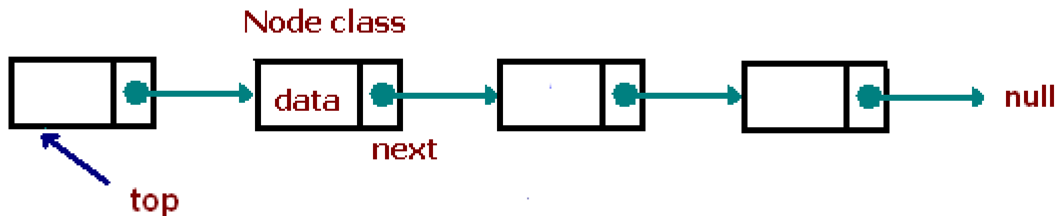
Issues with static stack

- Can run out of space when stack size is set too small
- Can waste memory if stack size set too large

Use a dynamic array implementation where memory for the stack array is set dynamically**.(Implementation –refer class notes)**

Stack using Linked list

A stack can be represented as a linked list(using a singly linked list or one way list).The data field stores the elements of the stack and the link field hold pointers to the neighbouring elements of the stack. The start pointer behaves as the top pointer and . NULL pointer in the last node indicates the bottom of the stack. In a stack push and pop operations are performed at one end called top. We can perform similar operations at one end of list using top pointer.



Push operation is done by inserting a node at the start of the list and pop is done by deleting the element pointed to by the top pointer.

Overflow/Underflow:

No limitation on the capacity of a linked stack and hence no overflow condition. Underflow or empty condition occurs when top==NULL

Linked Stack implementation

```
struct node
{
 int data;
 struct  node *next;
 };

 typedef struct node stack;
 stack  *top=NULL;

 stack *getnode()
 {
  stack *newnode;
  newnode=(stack*)malloc(sizeof(stack));
  if (newnode==NULL)
   printf("error in memory alloc\n");
   printf("enetr data\n");
   scanf("%d",&newnode->data);
   newnode->next=NULL;
   return newnode;
  }
```

// Function to traverse and display elements of stack

```
 void traverse()
 {
  stack *temp=top;
  if (temp==NULL)
  {
   printf("stack empty\n");
   return;
   }
```

CAMBRIDGE INSTITUTE OF TECHNOLOGY

```
 while(temp!=NULL)
  {
   printf("%d",temp->data);
   temp=temp->next;
  }
 }
```
//Function to implement push operation

```
 void pushlinkedstack()
 {
  stack *n1;
  n1=getnode();
  if (top==NULL)
  { top=n1;
   n1->next=NULL;
   return;
   }
   n1->next=top;
   top=n1;
  }
```
//Function to implement pop operation

```
 void poplinkedstack()
  {
   stack *temp;
   if (top==NULL)
    {
   printf("stack empty\n");
   return;
   }
    temp=top;
    top=top->next;
    printf("the element deleted is %d",temp->data);
    free(temp);
   }
```

Algebraic Expressions:

An algebraic expression is a legal combination of operators and operands. Operand is the quantity on which a mathematical operation is performed. Operand may be a variable like x, y, z or a constant like 5, 4, 6 etc. Operator is a symbol which signifies a mathematical or logical operation between the operands. Examples of familiar operators include +, -, *, /, ^ etc. An algebraic expression can be represented using three different notations. They are infix, postfix and prefix notations:

Infix: It is the form of an arithmetic expression in which we fix (place) the arithmetic operator in between the two operands.
Example: (A + B) * (C - D)
Prefix: It is the form of an arithmetic notation in which we fix (place) the arithmetic operator before (pre) its two operands. The prefix notation is called as polish notation (due to the polish mathematician Jan Lukasiewicz in the year 1920).
Example: * + A B – C D
Postfix: It is the form of an arithmetic expression in which we fix (place) the arithmetic operator after (post) its two operands. The postfix notation is called as suffix notation and is also referred to reverse polish notation.
Example: A B + C D - *

The three important features of postfix expression are:
1**.** The operands maintain the same order as in the equivalent infix expression.
2. The parentheses are not needed to designate the expression unambiguously.
3. While evaluating the postfix expression the priority of the operators is no longer relevant.
We consider five binary operations: +, -, *, / and $ or ↑ (exponentiation). For these binary operations, the following in the order of precedence (highest to lowest):

OPERATOR	**PRECEDENCE**	**VALUE**
Exponentiation ($ or ↑ or ^)	Highest	4
*, /	Next highest	3
+, -	Lowest	2
#	Lowermost (endofexpn)	1

<u>Applications of stacks</u>:
1. Stack is used by compilers to check for balancing of parentheses, brackets and braces.
2. Stack is used to evaluate a postfix expression.
3. Stack is used to convert an infix expression into postfix/prefix form.
4. In recursion, all intermediate arguments and return values are stored on the processor's stack.
5. During a function call the return address and arguments are pushed onto a stack and on return they are popped off. Conversion from infix to postfix:

<u>Conversion from infix to postfix expression</u>
Procedure to convert from infix expression to postfix expression is as follows: (algorithm)
1. Scan the infix expression from left to right.
2. a) If the scanned symbol is left parenthesis, push it onto the stack.
b) If the scanned symbol is an operand, then place directly in the postfix expression (output).
c) If the symbol scanned is a right parenthesis, then go on popping all the items from the stack and place them in the postfix expression till we get the matching left parenthesis.
d) If the scanned symbol is an operator, then go on removing all the operators from the stack and place them in the postfix expression, if and only if the precedence of the operator which is on the top of the stack is greater than (or greater than or equal) to the precedence of the scanned operator and push the scanned operator onto the stack otherwise, push the scanned operator onto the stack

Note :Refer class notes for examples and steps in conversion(detailed)

<u>Evaluation of postfix expression</u>
1. Scan the postfix expression from left to right.
2. If the scanned symbol is an operand, then push it onto the stack.
3. If the scanned symbol is an operator, pop two symbols from the stack ,assign to operand 2 and operand1 respectively. Perform operation and push onto stack
4. Repeat steps 2 and 3 till the end of the expression.
(Refer class notes for steps in conversion and examples)

Recursion

Recursion is deceptively simple in statement but exceptionally complicated in implementation. Recursive procedures work fine in many problems. Many programmers prefer recursion though simpler alternatives are available. It is because recursion is elegant to use though it is costly in terms of time and space.

Introduction to Recursion:

A function is recursive if a statement in the body of the function calls itself. Recursion is the process of defining something in terms of itself. For a computer language to be recursive, a function must be able to call itself.
For example, let us consider the function factr() shown below, which computers the factorial of an integer.

```
int factorial (int);
main() {
int num, fact;
 printf ("Enter a positive integer value: ");
  scanf ("%d", &num);
fact = factorial (num);
printf ("\n Factorial of %d =%5d\n", num, fact);
 }
int factorial (int n)
 {    int result;
  if (n == 0)   return (1);
else
result = n * factorial (n-1);
return (result);
 }
```

A non-recursive or iterative version for finding the factorial is as follows:

```
factorial (int n)
 { int i, result = 1;
if (n == 0)
return (result);
else
{
for (i=1; i<=n; i++)
result = result * i;
 return (result);
 }  }
```

The operation of the non-recursive version is clear as it uses a loop starting at 1 and ending at the target value and progressively multiplies each number by the moving product. When a function calls itself, new local variables and parameters are allocated storage on the stack and the function code is executed with these new variables from the start. A recursive call does not make a new copy of the function. Only the arguments and variables are new. As each recursive call returns, the old local variables and parameters are removed from the stack and execution resumes at the point of the function call inside the function.

When writing recursive functions, there must be an exit condition somewhere to force the function to return without the recursive call being executed. If there is no exit condition, the

recursive function will loop forever until you run out of stack space and indicate error about lack of memory, or stack overflow.

Differences between recursion and iteration:

• Both involve repetition.
• Both involve a termination test.
• Both can occur infinitely.

Iteration	Recursion
Iteration explicitly uses a repetition structure	Recursion achieves repetition through repeated function calls.
Iteration terminates when the loop ends	Recursion terminates when a base case is recognized
Iteration keeps modifying the counter until the loop continuation condition fails.	Recursion keeps producing simple versions of the original problem until the base case is reached.
Iteration normally occurs within a loop, so the extra memory usage is avoided	Recursion causes another copy of the function and hence a considerable memory space's occupied.
. It reduces the processor's operating time	. It increases the processor's operating time

Factorial of a given number:

The operation of recursive factorial function is as follows:
Start out with some natural number N (in our example, 5).
The recursive definition is:

n = 0, 0 ! = 1 Base Case
n > 0, n ! = n * (n - 1) ! Recursive Case

Recursion Factorials:
5! = 5*4! = 5*4*3! = 5*4*3*2! = 5*4*3*2*1! = 5*4*3*2*1*0! = 5*4*3*2*1*1 =120
We define 0! to equal 1, and we define factorial N (where N > 0), to be N * factorial (N-1). All recursive functions must have an exit condition that is a state when the function terminates. The exit condition in this example is when N = 0.

Tracing of the flow of the factorial () function:

When the factorial function is first called with, say, N = 5, here is what happens:
FUNCTION: Does N = 0? No Function Return Value = 5 * factorial (4)
At this time, the function factorial is called again, with N = 4.
FUNCTION: Does N = 0? No Function Return Value = 4 * factorial (3)
At this time, the function factorial is called again, with N = 3.
FUNCTION: Does N = 0? No Function Return Value = 3 * factorial (2)
At this time, the function factorial is called again, with N = 2.
FUNCTION: Does N = 0? No Function Return Value = 2 * factorial (1)
At this time, the function factorial is called again, with N = 1.
FUNCTION: Does N = 0? No Function Return Value = 1 * factorial (0)
At this time, the function factorial is called again, with N = 0.
FUNCTION: Does N = 0? Yes Function Return Value = 1

Now, trace the way back up! See, the factorial function was called six times. At any function level call, all function level calls above still exist! So, when we have N = 2, the function instances where N = 3, 4, and 5 are still waiting for their return values.

So, the function call where N = 1 gets retraced first, once the final call returns 0. So, the function call where N = 1 returns 1*1, or 1. The next higher function call, where N = 2, returns 2 * 1 (1, because that's what the function call where N = 1 returned). Just keep working up the chain till the final solution is obtained.

When N = 2, 2 * 1, or 2 was returned. When N = 3, 3 * 2, or 6 was returned. When N = 4, 4 * 6, or 24 was returned. When N = 5, 5 * 24, or 120 was returned.
And since N = 5 was the first function call (hence the last one to be recalled), the value 120 is returned.

The Towers of Hanoi:

In the game of Towers of Hanoi, there are three towers labeled 1, 2, and 3. The game starts with n disks on tower A. For simplicity, let n is 3. The disks are numbered from 1 to 3, and without loss of generality we may assume that the diameter of each disk is the same as its number. That is, disk 1 has diameter 1 (in some unit of measure), disk 2 has diameter 2, and disk 3 has diameter 3. All three disks start on tower A in the order 1, 2, 3. **The objective of the game is to move all the disks in tower 1 to entire tower 3 using tower 2. That is, at no time can a larger disk be placed on a smaller disk.**

The rules to be followed in moving the disks from tower 1 tower 3 using tower 2 are as follows:

- Only one disk can be moved at a time.
- Only the top disc on any tower can be moved to any other tower.
- A larger disk cannot be placed on a smaller disk.

The towers of Hanoi problem can be easily implemented using recursion. To move the largest disk to the bottom of tower 3, we move the remaining n – 1 disks to tower 2 and then move the largest disk to tower 3. Now we have the remaining n – 1 disks to be moved to tower 3. This can be achieved by using the remaining two towers. We can also use tower 3 to place any disk on it, since the disk placed on tower 3 is the largest disk and continue the same operation to place the entire disks in tower 3 in order.
The program that uses recursion to produce a list of moves that shows how to accomplish the task of transferring the n disks from tower 1 to tower 3 is as follows:

```
void towers_of_hanoi (int n, char *a, char *b, char *c);
int cnt=0;
int main (void)
 {
int n;
 printf("Enter number of discs: ");
scanf("%d",&n);
towers_of_hanoi (n, "Tower 1", "Tower 2", "Tower 3");
 getch();
}
void towers_of_hanoi (int n, char *a, char *b, char *c)
{
 if (n == 1)
{        ++cnt;
```

```
        printf ("\n%5d: Move disk 1 from %s to %s", cnt, a, c);
        return;
 }
 else
{
        towers_of_hanoi (n-1, a, c, b);  ++cnt;
        printf ("\n%5d: Move disk %d from %s to %s", cnt, n, a, c);
        towers_of_hanoi (n-1, b, a, c);  return;
} }
```

Output of the program:

RUN 1:

```
Enter the number of discs: 3
1: Move disk  1  from tower  1  to tower  3.
 2: Move disk  2  from tower  1  to tower  2.
 3: Move disk  1  from tower  3  to tower  2.
4: Move disk  3  from tower  1  to tower  3.
 5: Move disk  1  from tower  2  to tower  1.
6: Move disk  2  from tower  2  to tower  3.
7: Move disk  1  from tower  1  to tower  3.
```

Fibonacci Sequence Problem:

A Fibonacci sequence starts with the integers 0 and 1. Successive elements in this sequence are obtained by summing the preceding two elements in the sequence. For example, third number in the sequence is 0 + 1 = 1, fourth number is 1 + 1= 2, fifth number is 1 + 2 = 3 and so on. The sequence of Fibonacci integers is given below:

0 1 1 2 3 5 8 13 21 . . . .

A recursive definition for the Fibonacci sequence of integers may be defined as follows:

Fib (n) = n if n = 0 or n = 1

Fib (n) = fib (n-1) + fib (n-2) for n >=2

We will now use the definition to compute fib(5):

fib(5) = fib(4) + fib(3)

=fib(3) + fib(2) + fib(3)

=fib(2) + fib(1) + fib(2) + fib(3)

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

=1 + 0 + 1 + fib(1) + fib(0) + fib(3)

=1 + 0 + 1 + 1 + 0 + fib(2) + fib(1)

=1 + 0 + 1 + 1 + 0 + fib(1) + fib(0) + fib(1)

=1 + 0 + 1 + 1 + 0 + 1 + 0 + 1 = 5

fib(2) is computed 3 times, and fib(3),is computed 2 times in the above calculations. The values of fib(2) or fib(3) are saved and reused whenever needed.

A recursive function to compute the Fibonacci number in the nth position is given below:

```
main()
{
printf (“=nfib(5) is %d”, fib(5));
}

fib (int n) {
int x;
if (n==0 | | n==1) return n;
```

```
x=fib(n-1) + fib(n-2);
return (x);
}
```

Program to calculate the greatest common divisor:

```
int check_limit (int a[], int n, int prime);
int check_all (int a[], int n, int prime);
long int gcd (int a[], int n, int prime);
void main()
{
        int a[20], stat, i, n, prime;
         printf ("Enter the limit: ");
        scanf ("%d", &n);
         printf ("Enter the numbers: ");
         for (i = 0; i < n; i ++)
         scanf ("%d", &a[i]);
         printf ("The greatest common divisor is %ld", gcd (a, n, 2));
}
int check_limit (int a[], int n, int prime)
{
         int i;
        for (i = 0; i < n; i++)
         if (prime > a[i])   return 1;
        return 0;
 }
int check_all (int a[], int n, int prime)
{
        int i;
        for (i = 0; i < n; i++)
         if ((a[i] % prime) != 0)    return 0;
         for (i = 0; i < n; i++)
           a[i] = a[i] / prime;  return 1;
}

long int gcd (int a[], int n, int prime)
 {
         int i;
         if (check_limit(a, n, prime))      return 1;
         if (check_all (a, n, prime))
         return (prime * gcd (a, n, prime));
 else
        return (gcd (a, n, prime = (prime == 2) ? prime+1 : prime+2));
}
```

Output:

Enter the limit: 5
Enter the numbers: 99   55   22   77   121
The greatest common divisor is 11

Ackerman's Function

In computability theory, the Ackermann function, named after Wilhelm Ackermann, is one of the simplest andearliest-discovered examples of a total computable function that is not primitive recursive. All primitive recursive functions are total and computable, but the Ackermann function illustrates that not all total computable functions are primitive recursive. The two-argument Ackermann function, is defined as follows for nonnegative integers m and N

$$A(m,n) = \begin{cases} n+1 & \text{if } m=0 \\ A(m-1,1) & \text{if } m>0 \text{ and } n=0 \\ A(m-1,A(m,n-1)) & \text{if } m>0 \text{ and} n>0 \end{cases}$$

Its value grows rapidly, even for small inputs. For example A(4,2) is an integer of 19,729 decimal digits.

```
/* Akerman Function*/
#include<stdio.h>
#include<stdlib.h>

int ack(int,int,int);

main()
{
int m,n;
printf("Enter the value for m : ");
scanf("%d",&m);
printf("Enter the value for n : ");
scanf("%d",&n);
printf("The value is : %d\n",ack(m,n));
}

int ack(int m,int n,)
{
if(m==0)
  return(n+1);
else if(m>0 && n==0)
  return ack(m-1,1);
else
  return ack(m-1,ack(m,n-1));

}
```

Queue

A queue is a linear list in which elements can be added at one end and elements can be removed only at other end. So the information in this list is processed in same order as it was received .Hence queue is called a FIFO structure.(First In First Out).

Ex: people waiting in a line at a bus stop.

The first person in queue is the first person to take bus. Whenever new person comes he joins at end of the queue.

Type of queues
1. Linear Queue 2. Circular queue. 3. Priority queue 4.Deque

Linear Queue

It is a linear data structure.
It is considered as ordered collection of items.
It supports FIFO (First In First Out) property.
It has three components:
A **Container of items** that contains elements of queue.
A pointer **front** that points the first item of the queue.
A pointer **rear** that points the last item of the queue.
Insertion is performed from **REAR** end.
Deletion is performed from **FRONT** end.
Insertion operation is also known as **ENQUEUE** in queue.
Deletion operation is also known as DEQUEUE in queue.

Implementation of Queue

Queue can be implementing by two ways:

- Array implementation.(Static and Dynamic arrays)
- Linked List implementation.

Array Representation of Queue

In Array implementation **FRONT** pointer initialized with **0** and **REAR** initialized with **-1**.Consider the implementation: - If there are 5 items in a Queue,

REAR=-1 and FRONT=0

After 1 insertion REAR=0 and insert item 10 into queue.

REAR=0 and FRONT=0

10				

Now insert 20 into queue

REAR=1 and FRONT=0

10	20			

Suppose now we delete one item from queue, as we know that deletion can be done from FRONT end in queue.
Now, REAR=1 and FRONT=1

	20			

Now we insert 30, 40 and 50 into queue respectively.
REAR=2 and FRONT=1

	20	30		

REAR=3 and FRONT=1

	20	30	40	

REAR=4 and FRONT=1

	20	30	40	50

Note: In case of empty queue, front is one position ahead of rear : FRONT = REAR + 1;.This is the queue underflow condition.
The queue is full when REAR =N-1.This is the queue overflow condition.

The figure above ,the last case after insertion of three elements, the rear points to 4, and hence satisfies the overflow condition although the queue still has space to accommodate one more element .This problem can be overcome by making the rear pointer reset to the starting position in the queue and hence view the array as a circular representation. This is called a circular queue.

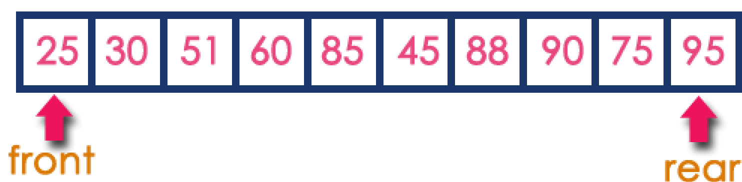
Implementation of queue using arrays

```
# include <conio.h>
 # define MAX 5
int Q[MAX];
int front=0, rear=-1;
void insertQ()  //Enqueue
{
int data;
if(rear == MAX-1)
 {   printf("\n Linear Queue is full");
  return;  }
 printf("\n Enter data: ");
scanf("%d", &data);
  Q[++rear] = data;
printf("\n Data Inserted in the Queue ");
}
void deleteQ()             // dequeue
 {  if( front>rear)       //OR  front=rear +1
{
   printf("\n\n Queue is Empty..");   return;
 }
  printf("\n Deleted element from Queue is %d", Q[front]);
  front++;
}
void displayQ()
{  int i;
 if(front >rear)
  {   printf("\n\n\t Queue is Empty");   return;  }
   printf("\n Elements in Queue are: ");
  for(i = front; i < rear; i++)
     printf("%d\t", Q[i]);
}
```

Circular Queue

In a normal Queue Data Structure, elements can be inserted until queue becomes full. But once if queue becomes full, no more elements can be inserted until all the elements are deleted from the queue. For example consider the queue below...

After inserting all the elements into the queue.



BRIDGE TO THE FUTURE
CAMBRIDGE
INSTITUTE OF TECHNOLOGY

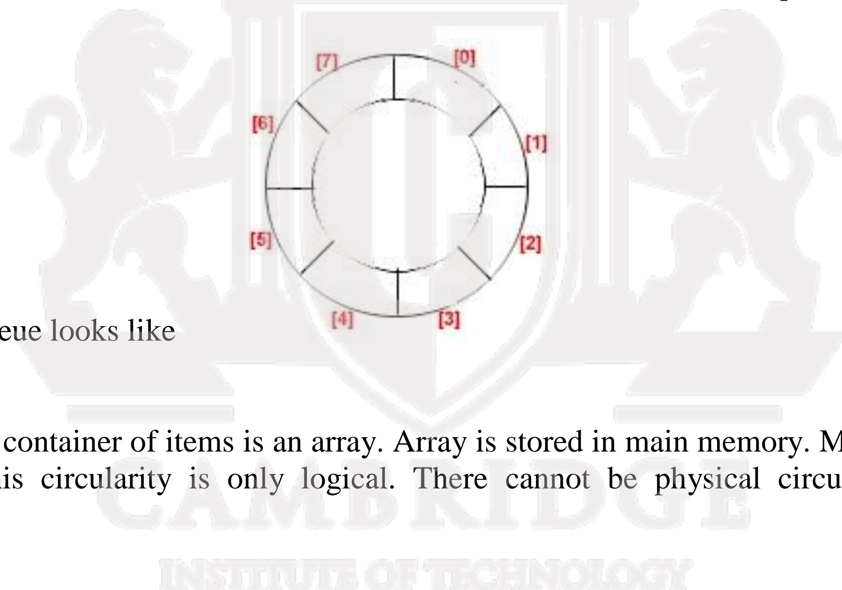
Now consider the following situation after deleting three elements from the queue...

Queue is Full (Even three elements are deleted)

This situation also says that Queue is Full and the new elementcannot be inserted because, '**rear**' is still at last position. In above situation, even though we have empty positions in the queue they cannot be used to insert new element. This is the major drawback in normal queue data structure. This is overcome in circular queue data structure.

So what's a Circular Queue?

A circular queue is linear data structure 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.



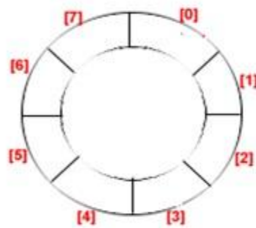
A circular queue looks like

Note:
Note that the container of items is an array. Array is stored in main memory. Main memory is linear. So this circularity is only logical. There cannot be physical circularity in main memory.

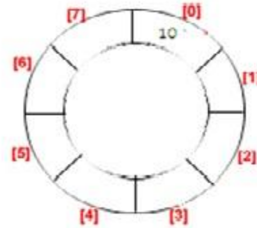
CAMBRIDGE
INSTITUTE OF TECHNOLOGY

Consider the example with Circular Queue implementation

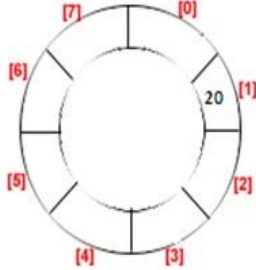
1) Initially: **Front = 0** and **rear = -1**



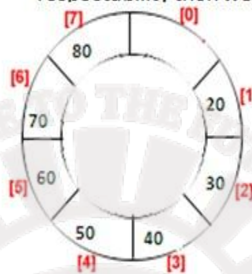
2) Add item 10 then **front = 0** and **rear =0.**



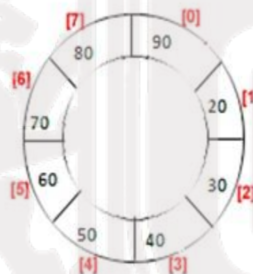
3) Now delete one item then **front = 1** and **rear =1.**



4) Like this now insert 30, 40, and 50,50,70,80 respectability then **front =1** and **rear = 7.**



5) Now in case of linear queue, we can not access 0 block for insertion but in circular queue next item will be inserted of 0 block then **front =0** and **rear = 0.**



Addition causes the increment in REAR. It means that when REAR reaches N-1 position then Increment in REAR causes REAR to reach at first position that is 0.

```
1  if( rear  == N -1 )
2     rear  = 0;
3  else
4     rear = rear + 1;
```

The short-hand equivalent representation may be

rear = (rear + 1) % N;

Deletion causes the increment in FRONT. It means that when FRONT reaches the N-1 position, then increment in FRONT, causes FRONT to reach at first position that is 0.

```
1  if( front  == N -1 )
2    front  = 0;
3  else
4    front = front + 1;
```

The short-hand equivalent representation may be

front = (front + 1) % N;

In any queue it is necessary that:

Before insertion, fullness of Queue must be checked (for overflow).

Before deletion, emptiness of Queue must be checked (for underflow).

Use count variable to hold the current position (in case of insertion or deletion).

CAMBRIDGE INSTITUTE OF TECHNOLOGY

Operation of Circular Queue using count

- Addition or Insertion operation.
- Deletion operation.
- Display queue contents

Array Implementation of Circular Queue

```
#define MAX 4
int CQ[MAX], n;
int r = -1;
int f = 0,ct=0;

void enqueue()   //function to insert an element to queue
{
   int key;

   if (ct == n )
   {
     printf("Queue Overflow\n");
     return;
   }
   printf("\nenter the element for adding in queue : ");
   r = (r+1)%n;
   scanf("%d", &key);
   CQ[r]=key;
   ct++;
}
void dequeue()          //function to remove an element from queue
{
   if (ct == 0)
   {
         printf("Queue Underflow\n");
         return ;
   }
         printf("Element deleted from queue is : %d\n", CQ[f]);
         f=(f+1)%n;
         ct--;
}
void display()
{
   int i,k=f;
   if (ct == 0)
   {
     printf("Queue is empty\n");
     return;
   }
   printf("contents of Queue are :\n");
   for (i = 0; i < ct; i++)
   {
     printf("%d\t", CQ[k]);
     k=(k+1)%n;
   }}
```

Double ended queue (deck)

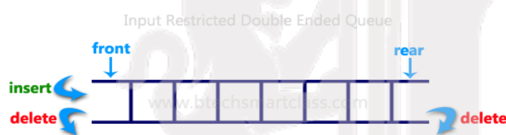
Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (**front** and **rear**). That means, we can insert at both front and rear positions and can delete from both front and rear positions.

Double Ended Queue can be represented in TWO ways, those are as follows...

Input Restricted Double Ended Queue
Output Restricted Double Ended Queue

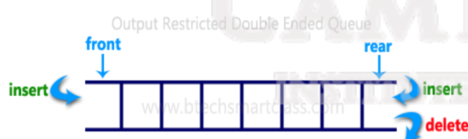
Input Restricted Double Ended Queue

In input restricted double ended queue, the insertion operation is performed at only one end and deletion operation is performed at both the ends.



Output Restricted Double Ended Queue

In output restricted double ended queue, the deletion operation is performed at only one end and insertion operation is performed at both the ends.



Deque is a variation of queue data structure, pronounced "deck", which stands for double-ended queue. In a deque values can be inserted at either the front or the back, A collection of peas in a straw is a good example..

Queues and deques are used in a number of ways in computer applications. A printer, for example, can only print one job at a time. During the time it is printing there may be many different requests for other output to be printed. To handle this printer will maintain a queue of pending print tasks. Since you want the results to be produced in the order that they are received, a queue is the appropriate data structure.

For a deque the defining property is that elements can only be added or removed from the end points. It is not possible to add or remove values from the middle of the collection.

Operations on deque

- Insertfront
- Deletefront
- INsertrear
- Deleterear

Deque Implementation

- Using arrays
- Linked list –Doubly linked list

Array Implementation of Deque

```
#include<stdio.h>
#define N 5
int dq[N],front=0,rear=-1,count=0;
```

//insert element at the rear end

```
void insertrear()
{
int key;
if (count==N)
 printf("overflow\n");
else
  {
   printf("enter the key to be inserted\n");
   scanf("%d",&key);
   rear=(rear+1)%N;
   dq[rear]=key;
   count++;
  }
}
```

//insert element at the front

```
void insertfront()
{
  int key;
  if (count==N)
    printf("overflow\n");
  else
   {
     printf("enter the key elemet\n");
     scanf("%d",&key);
     if (front==0)
       front=N-1;
     else
       front=front-1;
    dq[front]=key;
    count++;
   }}
```

//delete element from the front

```
 void deletefront()
 {
  if (count==0)
   printf("underflow\n");
 else
  {
   printf("element deleted is%d",dq[front]);
   front=(front+1)%N;
   count--;
  }
 }
```

//delete element from the rear end

```
 void deleterear()
 {
 if  (count==0)
  printf("underflow\n");
 else
  {
   printf("%d is rear value\n",rear);
   printf("element deleted is %d",dq[rear]);
   if (rear==0)
     rear=N-1;
   else
     rear=rear-1;
   count--;
   }
 }
  void display()
 {
  int i,k;
   if (count==0)
              printf("empty queue\n");
   else
   {
       k=front;
      for(i=0;i<count;i++)
      {
        printf("%d\t",dq[k]);
        k=(k+1)%N;
      }
   }
 }
```

Deque Implementation using Doubly Linked List

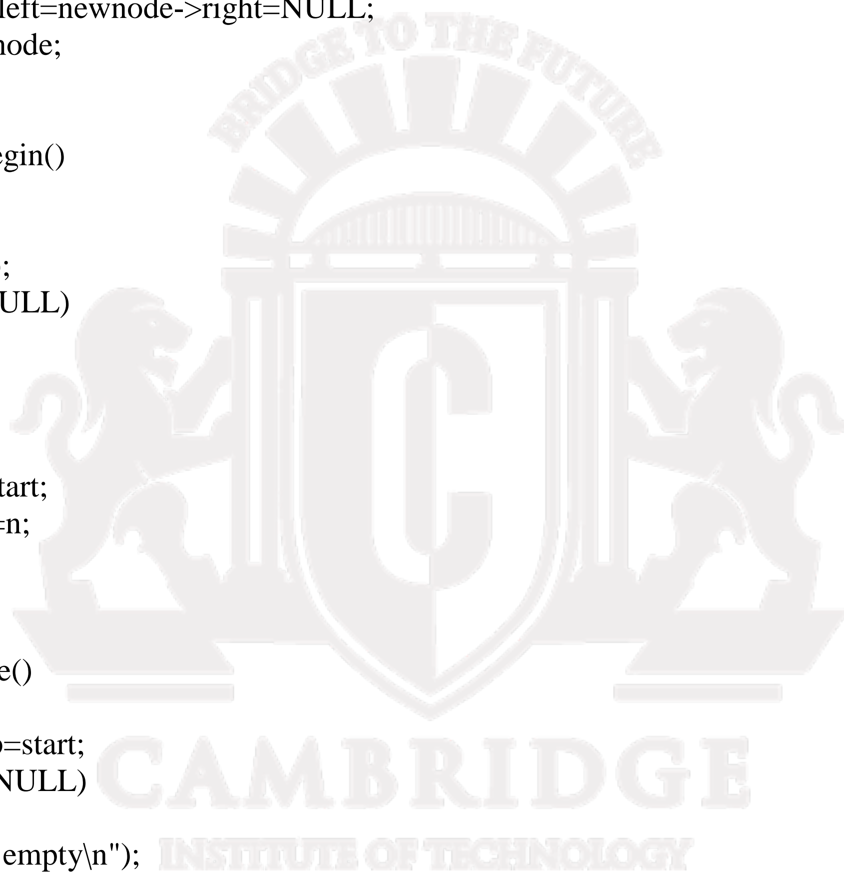
```
#include<stdio.h>
struct deque
{
 int data;
 struct node *left;
```

```
 struct node *right;
};
 typedef struct deque node;
 node *start=NULL;

node *getnode()
 {
  node *newnode;
  newnode=(node *)malloc(sizeof(node));
  if (newnode==NULL)
   printf("error in memory alloc\n");
   printf("enetr data\n");
   scanf("%d",&newnode->data);
   newnode->left=newnode->right=NULL;
   return newnode;
  }

 void insertbegin()
 {
  node *n;
  n=getnode();
  if (start==NULL)
   {
    start=n;
    return;
  }
    n->right=start;
    start->left=n;
    start=n;
  }

  void traverse()
  {
   node *temp=start;
   if (temp==NULL)
   {
    printf("list empty\n");
    return;
   }
   while(temp!=NULL)
   {
    printf("%d",temp->data);
    temp=temp->right;
   }
  }
  void insertend()
    {
     node *temp=start;
     node *n;
     n=getnode();
```



```
    if (start==NULL)
     {
      start=n;return;
     }
    while(temp->right!=NULL)
            temp=temp->right;
    n->left=temp;
    temp->right=n;
   }

  void delbegin()
  {
   node *temp;
   if (start==NULL)
   {
   printf("list  empty\n");
   return;
   }
   temp=start;
   start=temp->right;
   start->left=NULL;
   printf("the element to be deleted is %d",temp->data);
   free(temp);
   }

    void delend()
   {
    node *temp,*prev;
    if (start==NULL)
     {
      printf("list empty\n");
             return;
     }
    if (start->right==NULL)
           {
            printf("node deleted is %d",start->data);
            free(start);
            start=NULL;
            return;
           }
  temp=start;
 while(temp->right!=NULL)
{
 prev=temp;
 temp=temp->right;
}
prev->right=NULL;
printf("deleted info is %d",temp->data);
free(temp);
 }
```

Priority Queue:
A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules:
1. An element of higher priority is processed before any element of lower priority.
2. two elements with same priority are processed according to the order in which they were added to the queue.
A prototype of a priority queue is time sharing system: programs of high priority are processed first, and programs with the same priority form a standard queue.

Implementation of a Priority Queue

A priority queue can be implemented by creating a sorted or ordered list. A sorted list can be used to store the elements so that when an element is to be removed, the queue need not be searched for an element with the highest priority, since the element with the highest priority is already in the first position. Insertions are handled by inserting the elements in order.

C Program to Implement Priority Queue to Add and Delete Elements
```
#include <stdio.h>
#define MAX 5
int pri_que[MAX];
int front=0, rear=-1;

/* Function to insert value into priority queue */
void insert_by_priority(int data)
{
   if (rear== MAX - 1)
   {
      printf("\nQueue overflow no more elements can be inserted");
      return;}

   if (rear == -1))
   {
      rear++;
      pri_que[rear] = data;
   }
   else
      check(data);
      rear++;
}
/* Function to check priority and place element */
void check(int data)
{
   int i,j;
   for (i = 0; i <= rear; i++)
   {
      if (data >= pri_que[i])
      {
         for (j = rear + 1; j > i; j--)
```

```
            pri_que[j] = pri_que[j - 1];
          pri_que[i] = data;
         return;
       }
   }
   pri_que[i] = data;
}
/* Function to delete an element from queue */
void delete_by_priority(int data)
{
   int i;
   if (rear==-1)
   {
      printf("\nQueue is empty no elements to delete");
      return;
   }
  printf( "The element deleted is %d",pri_qui[front];
 front++;
 }
/* Function to display queue elements */
void display_pqueue()
{ int i;
   if (rear == -1))
   {
      printf("\nQueue is empty");
      return;}
      for (i=front; i <= rear; i++)
   {
      printf(" %d ", pri_que[i]);
   }
```

Linked Representation of a Priority Queue

A priority queue can be implemented using linked lists. When a priority queue is implemented as a linked list, then every node of the list has three fields (1)the data part (2)The priority number of the element(3)the address of the next element.

Implementation of priority queue

```
struct pqueue
{
int priority;
int info;
struct node *next;
};
typedef struct pqueue node;
node  *start=NULL;
```

```
void insert(int item,int item_priority)
{
struct node *new1,*temp;

new1=(struct node *)malloc(sizeof(struct node));
if(start==NULL)
{
printf("Memory not available\n");
return;
}
new1->info=item;
new1->priority=item_priority;
/*Queue is empty or item to be added has priority more than first element*/
if( start==NULL || item_priority < start->priority )
{
new1->next=start;
start=new1;
}
else
{
temp =start;
while( temp->next!=NULL && temp->next->priority<=item_priority )
temp=temp->next;
new1->next=temp->next;
temp->next=new1;
}
}/*End of insert()*/

void del()
{
struct node *temp;
if( start==NULL)
{
printf("Queue Underflow\n");
}
else
{
temp=start;
printf("the deleted element is %d",temp->info);
start=start->next;
free(temp);
}
}/*End of del()*/
```

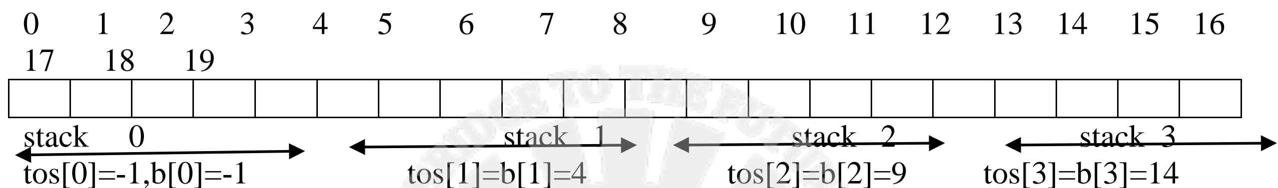
Applications of Queue:

1. It is used to schedule the jobs to be processed by the CPU.
2. When multiple users send print jobs to a printer, each printing job is kept in the printing queue. Then the printer prints those jobs according to first in first out (FIFO) basis.
3. Breadth first search uses a queue data structure to find an element from a graph.

BRIDGE TO THE FUTURE CAMBRIDGE INSTITUTE OF TECHNOLOGY

Multiple stacks

A sequential representation of a single stack using array is simple since only the top of the stack needs to be maintained and kept track. A linear structure like array can be used to represent multiple stacks. If multiple stacks are to be implemented, the array can be approximately divided into equal sized segments, each segment denoting a stack. The top and bottom of each stack are to be kept track of to manage insertions and deletions into the individual stacks.
Consider a set of N stacks to be implemented sunig an array. The array can be divided into N equal sized segments .
Say if the array can hold 20 elements, and 4 stacks are to be implemented then each individual stack hold 5 elements.



If i denotes an individual stack ,to establish multiple stacks, an array of top(tos[i])and bottom pointers (b[i])are maintained to keep track of the top and bottom of every stack.

Every stack‘s bottom and top pointer is set to B[i]=tos[i]=(size/n)*i-1 which enables dividing the stack to be divided into equal sized segments.

Overflow in any stack

tos[i]=b[i+1] // top pointer of one stack points to the bottom position of the following stack

Underflow in any stack

tos[i]=b[i] //top and bottom pointer of a stack in the same position

Implementation of multiple stacks using array

```
#define memsize 20          //size of array
#define maxstack 4          //number of stacks
int s[memsize],tos[maxstack],b[maxstack],n;
int main()
{
        int i;
        scanf("%d",&n);                 //number of stacks
        for(i=0;i<n;++)
        tos[i]=b[i]=(memsize/n)*i-1;
        b[n]=memsize-1;
// use switch case to call the different operations push, pop and display
}
void push()
{
         int ele;
        scanf("%d",&i);                 //stack number on which operation is to be done
        if (tos[i]==b[i+1])
{        printf("stack %d is full", i);return;}
        printf("enter the value to be inserted\n");
```

CAMBRIDGE INSTITUTE OF TECHNOLOGY

```
        scanf("%d",&ele);
        s[++tos[i]]=ele;
}
void pop()
{
        printf("enter the stack number\n");
        scanf("%d",&i);
        if (tos[i]==b[i])
         {
         printf("empty stack\n");return;}
        printf("deleted element is %d",s[tos[i]--];
}

void disp()
{
printf("enter the stack number\n");
scanf("%d",&i);
if (tos[i]==b[i])
 {
 printf("empty stack\n");return;}
printf("contents are \n");
for(j=b[i]+1;j<=tos[i];j++)
  printf("%d",s[j]);
}
```


Prepared by
Geetha.P,Asst Professor
Pankaja K, Asso Professor
Dept of CSE,CiTech

DO IT NOW.................SOMETIMES "LATER" BECOMES "NEVER".THERE IS NO SUBSTITUTE FOR HARD WORK!!!

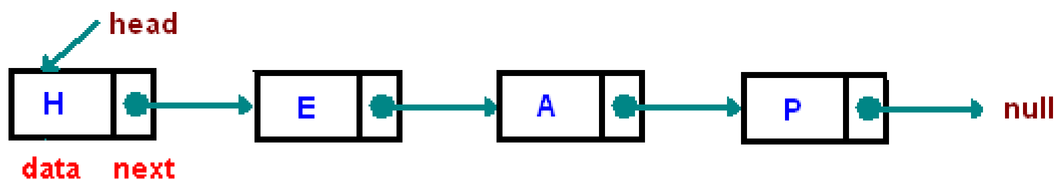
GOOD LUCK!!!!

CAMBRIDGE INSTITUTE OF TECHNOLOGY

MODULE 3

LINKED LISTS

Linked list is a linear data structure that consists of a sequence of elements where each element (usually called a **node**) comprises of two items - the data and a reference (link) to the next node. The last node has a reference to null. The entry point into a linked list is called the **head (start)** of the list. It should be noted that head is not a separate node, but the reference to the first node. If the list is empty then the start is a null reference. The list with no nodes –empty list or null list.



Note: The head is a pointer which points to the first node in the list. The implementation in the class, it has been discussed with the pointer's name as start. Head or start refers to the starting node.

A linked list is a dynamic data structure. The number of nodes in a list is not fixed and can grow and shrink on demand. Any application which has to deal with an unknown number of objects will need to use a linked list.

Arrays –drawbacks

(1)The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. (static in nature)

(2) Inserting and a new element in an array of elements is expensive, because room has to be created for the new elements and to create room existing elements have to shifted. Same holds good for deletion also.

CAMBRIDGE INSTITUTE OF TECHNOLOGY (Source : DIGINOTES)

Advantages of linked list:

Efficient memory utilization: The memory of a linked list is not pre-allocated. Memory can be allocated whenever required. And it is de-allocated when it is no longer required.

Insertion and deletion operations are easier and efficient:
Linked list provide flexibility in inserting a data item at a specified position and deletion of a data item from the given position.

Source : diginotes.in

Extensive manipulation:
We can perform any number of complex manipulations without any prior idea of the memory space available. (i.e. in stacks and queues we sometimes get overflow conditions. Here no such problem arises.)
Arbitrary memory locations
Here the memory locations need not be consecutive. They may be any arbitrary values. But even then the accessing of these items is easier as each data item contains within itself the address to the next data item. Therefore, the elements in the linked list are ordered not by their physical locations but by their logical links stored as part of the data with the node itself.

As they are dynamic data structures, they can grow and shrink during the execution of the
program

Disadvantages of linked lists

- They have a tendency to use more memory due to pointers requiring extra storage space.
- Nodes in a linked list must be read in order from the beginning as linked lists are inherently sequentially accessed.(cannot be randomly accessed)
- Nodes are stored incontiguously, greatly increasing the time required to access individual elements within the list.
- Difficulties arise in linked lists when it comes to reverse traversing. For instance, singly linked lists are cumbersome to navigate backwards[1] and while doubly linked lists are somewhat easier to read, memory is wasted in allocating space for a back pointer.

Note : No particular data structure is the best. The choice of the data structure depends on the kind of application that needs to be implemented. While for some applications linked lists may be useful, for others, arrays may be useful.

Operations on linked lists

- Creation of a list

  Creation operation is used to create a linked list. Once a linked list is created with one node, insertion operation can be used to add more elements in a node.

- Insertion of an element into a linked list

  Insertion operation is used to insert a new node at any specified location in the linked list. A new node may be inserted.

Source : diginotes.in

(a) At the beginning of the linked list
(b) At the end of the linked list
(c) At any specified position in between in a linked list

- Deletion of a node from the linked list

  Deletion operation is used to delete an item (or node) from the linked list. A nodemay be deleted from the
  (a) Beginning of a linked list
  (b) End of a linked list
  (c) Specified location of the linked list

- Traversing and displaying the elements in the list

  Traversing is the process of going through all the nodes from one end to another end of a linked list. In a singly linked list we can visit from lptr to rptr, forward traversing,nodes only. But in doubly linked list forward and backward traversing is possible

- Counting the number of elements in the list
- Searching for an element in the list
- Merging two lists(Concatenating lists)

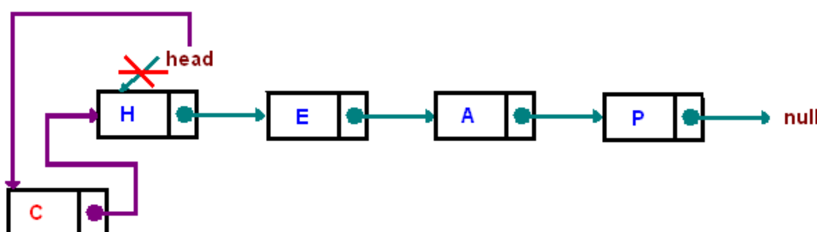
  Merging is the process of appending the second list to the end of the first list. Consider a list A having n nodes and B with m nodes. Then the operation concatenation will place the 1st node of B in the (n+1)th node in A. After concatenation A will contain (n+m) nodes

CAMBRIDGE
INSTITUTE OF TECHNOLOGY
(Source : DIGINOTES)

Linked List Basic Operations

INsertBegin

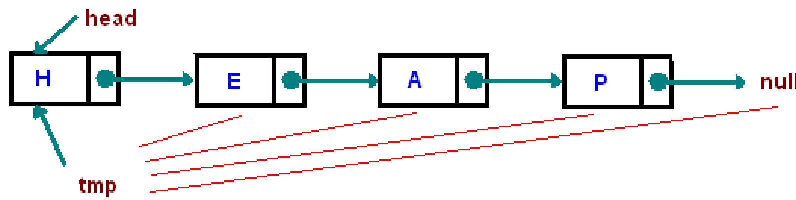
The method creates a node and inserts it at the beginning of the list.



Traversing

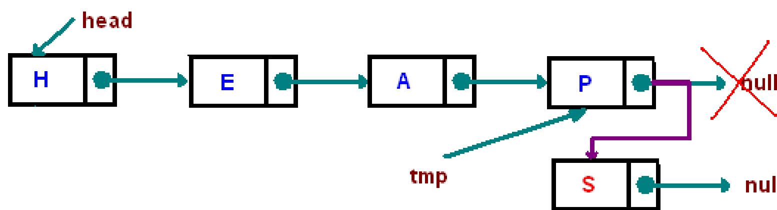
Source : diginotes.in

Start with the head and access each node until you reach null. Do not change the head reference.



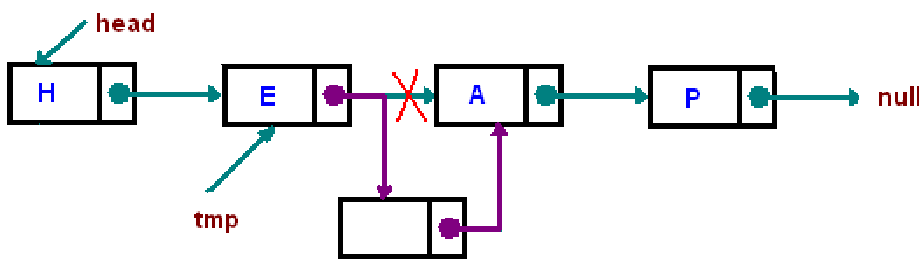
INsertend

The method appends the node to the end of the list. This requires traversing, but make sure you stop at the last node



Inserting "after"

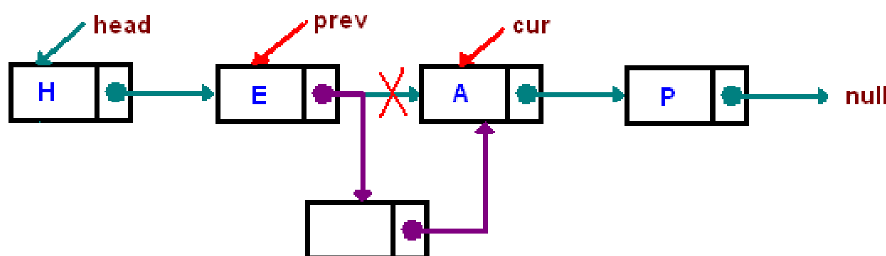
Find a node containing "key" and insert a new node after it. In the picture below, we insert a new node after "E":



(Source : DIGINOTES)

Inserting "before"

Find a node containing "key" and insert a new node before that node. In the picture below, we insert a new node before "A":

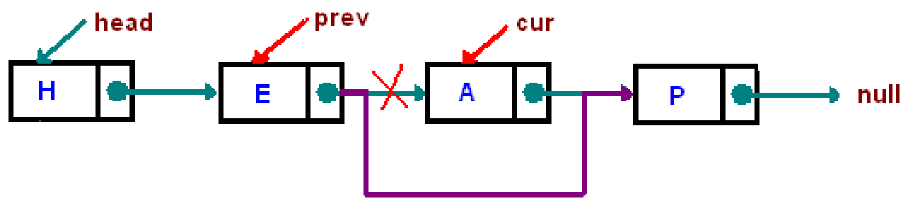


Source : diginotes.in

For the sake of convenience, we maintain two references prev and cur. When we move along the list we shift these two references, keeping prev one step before cur. We continue until cur reaches the node before which we need to make an insertion. If cur reaches null, we don't insert, otherwise we insert a new node between prev and cur.

Deletion

Find a node containing "key" and delete it. In the picture below we delete a node containing "A"



The algorithm is similar to insert "before" algorithm. It is convinient to use two references prev and cur. When we move along the list we shift these two references, keeping prev one step before cur. We continue until cur reaches the node which we need to delete. There are three exceptional cases, we need to take care of:

1. list is empty
2. delete the head node
3. node is not in the list

CAMBRIDGE
INSTITUTE OF TECHNOLOGY
(Source : DIGINOTES)

Type of linked lists

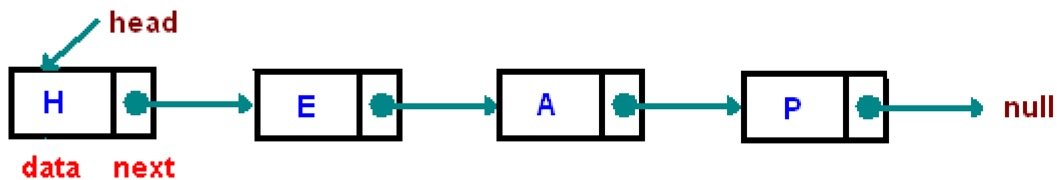
Depending on the way in which the links are used to maintain adjacency, several different types of linked lists are possible

1. Singly linked list
2. Doubly linked list
3. Circular linked list

Singly linked list

In a singly linked list, each node except the last one contains a single pointer to the next element.The last node has a null pointer to indicate the termination of the list.

Source : diginotes.in



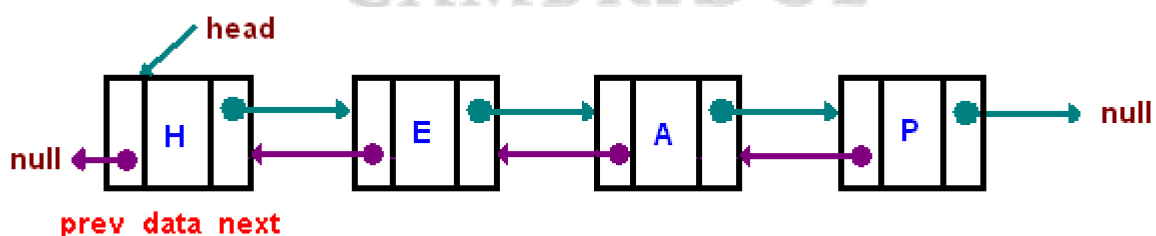
Doubly Linked List

A **doubly linked list** is a linked data structure that consists of a set of sequentially linked nodes. Each node contains two fields, called *links*, that are references to the previous and to the next node in the sequence of nodes. The beginning and ending nodes' **previous** and **next** links, respectively, point to NULL, to facilitate traversal of the list.

The two node links allow traversal of the list in either direction. While adding or removing a node in a doubly linked list requires changing more links than the same operations on a singly linked list, but the operations are simpler and more efficient **(for nodes other than first nodes)** because there is no need to keep track of the previous node during traversal or no need to traverse the list to find the previous node, so that its link can be modified.

Double-linked lists require more space per node , and their elementary operations are more expensive; but they are often easier to manipulate because they allow sequential access to the list in both directions. In particular, one can insert or delete a node in a constant number of operations given only that node's address. (Compared with singly- , which require the previous node's address in order to correctly insert or delete.) Some algorithms require access in both directions.

CAMBRIDGE



CIRCULAR LINKED LIST

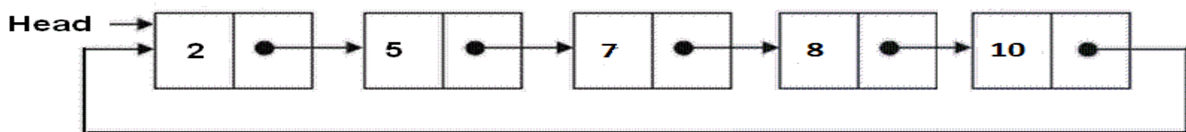
In a circularly-linked list, the first and final nodes are linked together. This can be done for both singly and doubly linked lists. To traverse a circular linked list, you begin at any node and follow the list in either direction until you return to the original node. Viewed another way, circularly-linked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

Source : diginotes.in

all other objects in the list in no particular order.Note that a circular lisr does not have a natural “first” or a “last” node. *There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.*

Convention(usual practice)

Let a **pointer(last)** point to last node of circular list and allow the following node to be the first node. If **last** is pointer to the last node of the circular list, the last node can be referenced by the last pointer and the first node by referencing last->next. If last ==NULL, it indicates an empty list



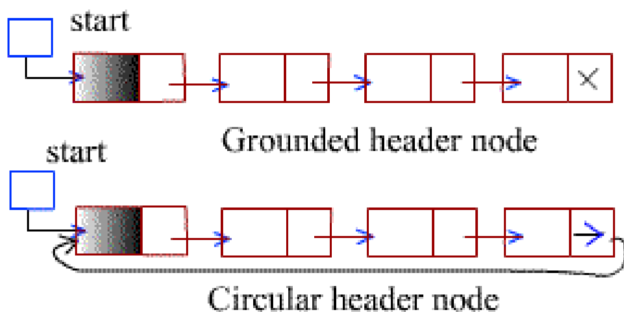
Advantages of Circular Linked Lists:

1) Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.

2) Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.

3) Multiplayer board games can be implemented using circular lists where each player waits for his turn to play in a circular fashion.

Header Linked List.



A header linked list is a linked list which always contains a special node called the

Source : diginotes.in

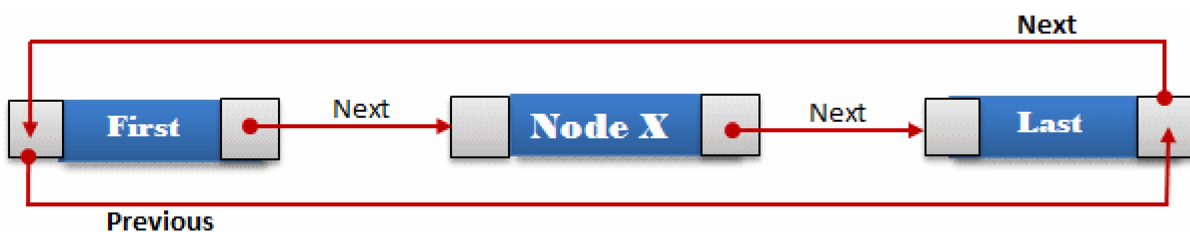
header node at the beginning of the list. It is an extra node kept at the front of a list. **Such a node does not represent an item in the list**. The information portion might be unused. There are two types of header list

1. ***Grounded header list*: is a header list where the last node contains the null pointer.**
2. ***Circular header list*: is a header list where the last node points back to the header node.**

More often , the information portion of such a node could be used to keep **global information** about the entire list such as:

- number of nodes (not including the header) in the list count in the header node must be adjusted after adding or deleting the item from the list
- pointer to the last node in the list it simplifies the representation of a queue
- pointer to the current node in the list eliminates the need of a external pointer during traversal

Circular Singly Linked Lists

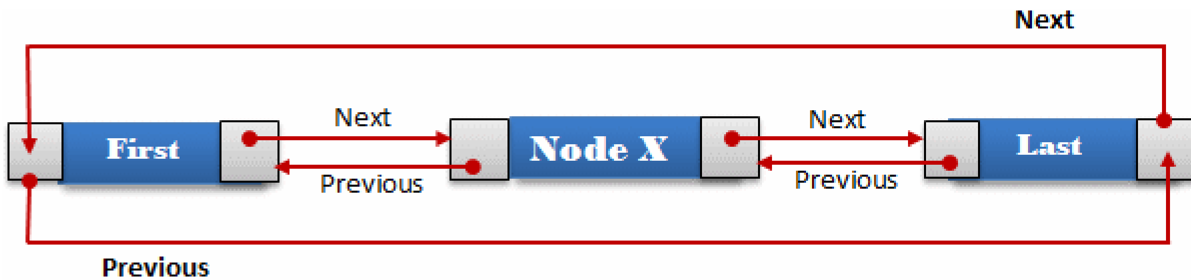


IN a singly linked circular list, the pointer field of the last node stores the address of the starting node
In the list.Hence it is easy to traverse the list given the address of any node in the list.

Circular Doubly Linked List

A *circular, doubly-linked list* is shown in Figure . The last element of the list is made the predecessor of the first node,and the first element is the successor of the last. We no longer need both a head and tail variable to keep track of the list. Even if only a single variable is used, both the first and the last list elements can be found.

Source : diginotes.in



1. Implementation of Singly linked list

```
#include<stdio.h>
#include<stdlib.h>
struct SLL
{
        char info;
        struct SLL *next;
};
typedef struct SLL node;
node *start;

node *getnodeSLL()
{
        node *newnode;
        new1=(node*)malloc(sizeof(node));
        printf("\n Enter the data");
        scanf("%d", &new1->info);
        new1->next=NULL;
        return new1;
}

void insert_front()
{
        node *n1;
        n1=getnodeSLL();
        if(start==NULL)
        {
                start=n1;
                return;
        }
                n1->next=start;
                start=n1;

}

void delete_front()
{
```


CAMBRIDGE INSTITUTE OF TECHNOLOGY (Source : DIGINOTES)

Source : diginotes.in

```
        node *temp = start;
        if(start==NULL)
        {
                printf("\n List is empty");
                return;
        }
        printf("\nt%d\t is deleted ",temp->info);
        start=temp->next;
        free(temp);
        }
}

void insert_end()
{
        node *n1,*temp = atsrt;
        n1=getnodeSLL();
        if(start==NULL)
        {
                start=n1;
                return;
        }
        while(temp-> next!=NULL)
                temp=temp->next;

        temp->next=n1;
}

void delete_end()
{
        node *temp=start, *prev;
        if(start==NULL)
        {
                printf("\n Empty list");
                return;
        }
        if(start->next==NULL)
        {
                printf("\nt%d\t is deleted",start->info);
                free(start);
                start=NULL; return;
        }
        while(temp->next != NULL)
        {
                prev = temp;
```


BRIDGE TO THE FUTURE
CAMBRIDGE
INSTITUTE OF TECHNOLOGY
(Source : DIGINOTES)

Source : diginotes.in

```
                temp = temp->next;
        }
        prev->next = NULL;
        printf("\nThe deleted node is %d\t", temp->info);
        free(temp);
}




void display()
{
        node *temp=start;
        if(start==NULL)
        {
                printf("\n Empty list");
                return;
        }
        printf("\n The details are");
        while(temp!=NULL)
        {
                printf("\n %d\t",temp->info);
                temp=temp->next;
        }

}

int main()
{
        int n,m,i;
        while(1)
        {
                printf("\n Enter 1:insert_front\n 2:insert_end\n 3:delete_front\n
                4:delete_end\n 5:display");
                scanf("%d",&m);
                switch(m)
                {
                        case1: insert_front();  break;
                        case 2: insert_end();break;
                        case 3: delete_front();break;
                        case 4 : delete_end();break;
                        case 5:display();break;
                        case 6: exit(0);
                }
```


CAMBRIDGE INSTITUTE OF TECHNOLOGY (Source : DIGINOTES)
Source : diginotes.in

```
        }
        return 0;
}
```

2. Implementation of Doubly Linked list

```
#include<stdio.h>
#include<stdlib.h>
struct DLL
{
        int info;
        struct DLL *lptr,*rptr;
};
typedef struct DLL node;
node *start=NULL;

node *getnodeDLL()
{
        node *new1;
        new1=(node*)malloc(sizeof(node));
        printf("\n Enter the data");
        scanf("%d", &new1->info);
        new1->lptr=NULL;
        new1->rptr=NULL;
        return new1;
}

void insert_front()
{
        node *n;
        n=getnodeDLL();
        if(start == NULL)
        {
                start = n;
                return;
        }
        n->rptr=start;
        start->lptr=n;
        start=n;
}

void delete_front()
```

CAMBRIDGE
INSTITUTE OF TECHNOLOGY
(Source : DIGINOTES)

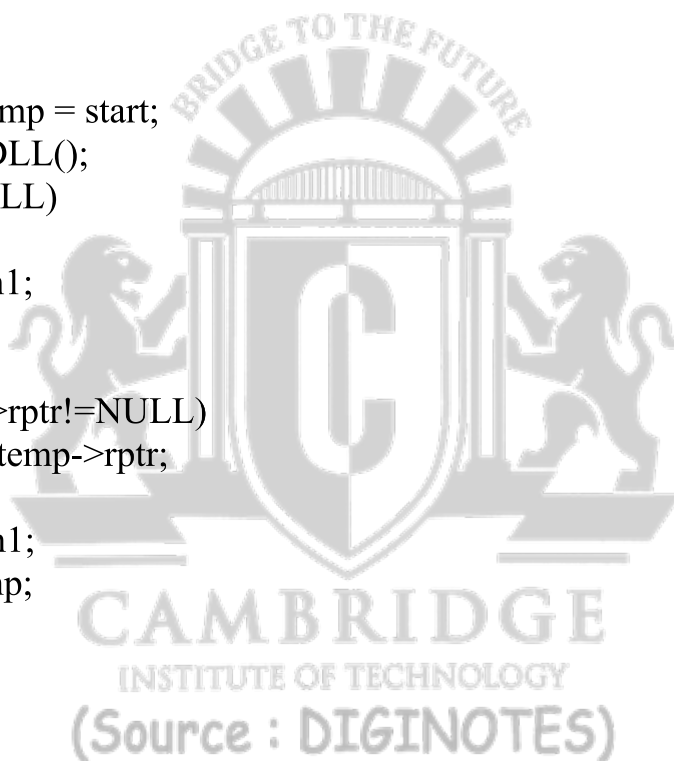
Source : diginotes.in

```
{
        node *temp = start;
        if(start==NULL)
        {
                printf("\n List is empty");
                return;
        }
        printf("\nt%d\t is deleted ",temp->info);
        start=temp->rptr;
        start->lptr=NULL;
        free(temp);
}


void insert_end()
{
        node *n1,*temp = start;
        n1=getnodeDLL();
        if(start==NULL)
        {
                start=n1;
                return;
        }
        while(temp->rptr!=NULL)
                temp=temp->rptr;

        temp->rptr=n1;
        n1->lptr=temp;
}

void delete_end()
{
        node *temp =start;
        if(start==NULL)
        {
                printf("\n Empty list");
                return;
        }
        if(start->next==NULL)
        {
                printf("\nt%d\t is deleted",start->info);
                free(start);
                start=NULL; return;
        }
```



Source : diginotes.in

```
        while(temp->rptr!=NULL)
                temp=temp->rptr;
        (temp->lptr)->rptr=NULL;
        printf("\nThe deleted node is %d\t", temp->info);
        free(temp);

}


void traverse()
{
        node *temp = start;
        if(start==NULL)
        {
                printf("\n Empty list");
                return;
        }
        printf("\n The details are");
        while(temp!=NULL)
        {
                printf("\n %d\t",temp->info);
                temp=temp->rptr;
        }

}

int main()
{
        int n,m,i;
        while(1)
        {
                printf("\n Enter 1:insert_front\n 2:insert_end\n 3:delete_front\n
                4:delete_end\n 5:display");
                scanf("%d",&m);
                switch(m)
                {
                        case1: insert_front();  break;
                        case 2: insert_end();break;
                        case 3: delete_front();break;
                        case 4 : delete_end();break;
                        case 5:traverse();break;
                        case 6: exit(0);
                }
        }
```


Source : diginotes.in

```
        return 0;
}
```

3. Implementation of Circular Singly Linked list using last pointer

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

struct CSLL
{
   int info;
   struct CSLL *next;
};
typedef struct CSLL node;
node *last = NULL;

void insert_begin()
{
  node *new1;

  new1 = getnodeSLL();
  if (last == NULL)
  {
    last =new1;
    last->next = last;
     return;
  }
  new1 -> next = last->next;
  last -> next = new1;

}

void insert_end()
{
  node *new1;
  new1 = getnodeSLL();
  if (last == NULL)
  {
     last =new1;
     last->next = last;
```


CAMBRIDGE INSTITUTE OF TECHNOLOGY (Source : DIGINOTES)
Source : diginotes.in

```
    return;
  }
  new1 -> next = last->next;
  last->next = new1;
  last = new1;
}


void del_end()
{
        node *prev,*temp=last->next;
        if (last == NULL)
        {
                printf("\n empty");
                return;
        }

         if (last->next == temp->next)
        {
                printf("deleted item is %d  ", last->info);
               free(last);
                last = NULL;
               return;
         }
        printf("deleted item is %d  ", last->info);
        while(temp->next != last)
             temp=temp->next;

        temp->next = last->next;
        free(last);
        last = temp;
}

void display()
{
        node *temp = last->next;
         if(last == NULL)
        {
                printf("list empty\n");
                return;
        }
         do
        {
                printf("%d   ",temp->info);
```


CAMBRIDGE INSTITUTE OF TECHNOLOGY (Source : DIGINOTES)
Source : diginotes.in

```
                temp=temp->next;
        }while(temp != last->next);
}

main()
{
   int choice; clrscr();
   while (1)
   {
        printf("\n1.Insert_beg\n 2.Isert_end\n 3.del end \n 4. Display\n 5.Quit\n");
        printf("Enter your choice : ");
        scanf("%d", &choice);
        switch (choice)
        {
           case 1: insert_begin();
           break;
           case 2: insert_end();
           break;
           case 3: del_end();
           break;
           case 4: display();
           break;
           case 5: exit(1);
           default:
                   printf("Wrong choice\n");
        }
    }
  getch();
}
```

CAMBRIDGE
INSTITUTE OF TECHNOLOGY
(Source : DIGINOTES)

4. Program to implement Addition of two polynomials with 3 variables in each term. Use Circular Singly Linked List with header node.

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
struct poly
{
  int coef, expo1,expo2,expo3,flag;
  struct poly *next;
};
typedef struct poly node;

void insert_end(node *h, int a, int x, int y, int z)
```


Source : diginotes.in

```
{
  node *temp = h->next, *new1;
  new1 = (node*) malloc(sizeof(node));
  new1->coef = a;
  new1->expo1 = x;
  new1->expo2 = y;
  new1->expo3 = z;
  new1->flag=0;
  while(temp->next != h)
    temp = temp -> next;

  temp -> next = new1;
  new1 -> next = h;
}

void read_poly(node *head)
{
   int a,x,y,z;
   char ch;
   do
    {
     printf("\nenter coef & expo1, expo2, expo3\n");
     scanf("%d%d%d%d",&a,&x,&y,&z);
     insert_end(head,a,x,y,z);
     printf("do u want to continue(Y/N) ?");
     ch=getche();
    }while(ch == 'Y' || ch == 'y');
}

void add_poly(node *h1,node *h2,node *h3)
{
   node *p1=h1->next, *p2;
   int x;

   while( p1 != h1)
   {
     p2 = h2 -> next;
     while(p2 != h2)
     {
        if( p1->expo1 == p2->expo1 && p1->expo2 == p2->expo2 && p1->expo3
        == p2->expo3)
         {
               x = p1->coef + p2->coef;
               insert_end(h3,x,p1->expo1,p1->expo2,p1->expo3);
```


CAMBRIDGE INSTITUTE OF TECHNOLOGY (Source : DIGINOTES)
Source : diginotes.in

```
                p1->flag=1;
                p2->flag=1;
            }
            p2 = p2->next;
        }
        p1 = p1 -> next;

    }
    p1=h1->next;
    p2=h2->next;

    while(p1 != h1)
    {
     if(p1 -> flag==0)
          insert_end(h3, p1->coef, p1->expo1, p1->expo2, p1->expo3);
     p1 = p1 -> next;
    }

    while(p2 != h2)
    {
      if(p2 -> flag == 0)
            insert_end(h3,p2->coef,p2->expo1, p2->expo2, p2->expo3);
      p2 = p2->next;
    }
}

void display(node *h)
{
  node *temp = h->next;
  if(temp == h)
  {
   printf("list empty\n");
   return;
  }
  while(temp != h)
  {
      printf(" %+ dx^%dy^%dz^%d ",temp->coef,temp->expo1,temp->expo2,temp-
>expo3);
    temp=temp->next;
  }
}

void evaluate(node *h)
{
```


CAMBRIDGE INSTITUTE OF TECHNOLOGY (Source : DIGINOTES)
Source : diginotes.in

```
	int x,y,z,sum=0;
	node *temp = h->next;
	printf("\nEvauate the resultant polynomial by giving values for X, Y and Z");
	scanf("%d%d%d",&x,&y,&z);
	while(temp != h)
	{
		sum = sum + temp->coef * pow(x,temp->expo1) * pow(y,temp->expo2) * pow(z,temp->expo3);
		temp = temp->next;
	}
	printf("\nSum = %d",sum);
}

main()
{
  node *h1,*h2,*h3;
  clrscr();
  h1 = (node*) malloc(sizeof(node));
  h1->next = h1;
  h2 = (node*) malloc(sizeof(node));
  h2->next = h2;
  h3 = (node*) malloc(sizeof(node));
  h3->next = h3;
  printf("\nenter the first poly");
  read_poly(h1);
  printf("\nenter the second poly");
  read_poly(h2);
  add_poly(h1,h2,h3);
  printf("\nTHE FIRST POLY IS\n");
  display(h1);
  printf("\nTHE SEC POLY IS\n");
  display(h2);
  printf("\nADDition of TWO poly are\n");
  display(h3);
  evaluate(h3);
  getch();
}
```

5. Write function for inserting a new node before the key element in DLL*/

```
void ins_node_bef_Key()
{
	node *temp=start,*new1;
	int key;
```


Source : diginotes.in

```
        if(start == NULL)
        {
                printf("Insertion is not possible\n");
                return;
        }
        printf("Enter the key\n");
        scanf("%d",&key);
        if(start->info==key)
        {
                insert_front();
                return;
        }
        while(temp!=NULL && temp->info!=key)
                temp=temp->rptr;

        if(temp==NULL)
        {
                printf("Key not found\n");
                return;
        }
        new1=getnode();
        (temp->lptr)->rptr=new1;
        new1->lptr=temp->lptr;
        new1->rptr=temp;
        temp->lptr=new1;
}
```

6. Write function to delete the key element in DLL*/

CAMBRIDGE
INSTITUTE OF TECHNOLOGY
(Source : DIGINOTES)

```
void del_key()
{
        node *temp=start;
        int key;
        if(start == NULL)
        {
                printf("Deletion not possible\n");
                return;
        }
        printf("Enter the key to be deleted\n");
        scanf("%d",&key);
        if(start->info == key)
        {
                printf("\n deleted element is %d ", start->info);
                start=start->rptr;
```


Source : diginotes.in

```
            free(temp);
            return;
        }

        while(temp!=NULL&&temp->info!=key)
            temp=temp->rptr;
        if(temp ==NULL)
        {
            printf("Key not found");
            return;
        }
        (temp->lptr)->rptr=temp->rptr;
        (temp->rptr)->lptr=temp->lptr;
        printf("The deleted element is %d",temp->info);
        free(temp);
        return;
}
```

7. **Write function for concatenation of two lists**

Assuming two linked lists are created with start1 pointing to 1st list and start2 pointing to 2nd list

```
node * concat(node *start1, node *start2)
{
      node  *temp = start1;
       if( start1 == NULL)
        return (start2);
      if(start2 == NULL)
        return(start1);
      if(start1 == NULL  && start2 == NULL)
      {
        Printf("list is empty");
        return;
      }
      While (temp -> next != NULL)
        temp = temp -> next;

     temp -> next = start2;
     return (start1);
}
```


CAMBRIDGE INSTITUTE OF TECHNOLOGY (Source : DIGINOTES)
Source : diginotes.in

8. Function to reverse SLL

```
void reverse()
{
        node *temp, *prev = NULL;
        while(start != NULL)
        {
                temp = start;
                start = start -> next'
                temp -> next = prev;
                prev = temp;
        }
        start = temp;
}
```

9. Function to display nodes in reverse order in DLL

```
void display()
{
        node *temp = start;
        while(temp -> rptr != NULL)
                temp = temp -> rptr;
         while(temp != NULL)
        {
                printf("%d",temp->info);
                temp = temp -> lptr;
        }
}
```

10. Write function to delete all nodes in SLL & DLL

```
void del_all()
{
        node *temp = start;
        if(start == NULL)
        {
                printf("list is empty");
```


Source : diginotes.in

```
            return;
        }
        while(temp != NULL)
        {
            start = start -> next;
            printf(" %d deleted from the list ",temp -> info);
            free(temp);
            temp = start;
        }
}
```

11. Write a program to implement queue using Singly linked list:

Method I

You should make use of Insert_front(), Delete_End() and Display() functions

Method II

Insert_End(), Delete_front and Display()

12. Write a program to implement stack using Singly linked list:

Method I

You should make use of Insert_front(), Delete_Front() and Display() functions

Method II

Insert_End(), Delete_End() and Display()

13. To find the length of the list

```
void length()
{
   node  *temp = start;
   while(temp!=NULL)
    {
     temp=temp->next;
       cnt++;
    }
    printf(" length of list are %d", cnt);
}
```

14. To display odd and even nodes in the list along with its count

```
void odd_even_inlist()
```


Source : diginotes.in

```
{
      node  *temp = start;
     printf(" even nodes are \n");
      while(temp!=NULL)
      {
        if( temp -> info % 2 == 0)
        {
             printf("   %d   ", temp -> info);
           even++;
          }
        temp = temp -> next;
       }
    temp = start;
     printf(" odd nodes are \n");
      while(temp!=NULL)
      {
        if( temp -> info % 2 != 0)
        {
             printf("   %d   ", temp -> info);
           odd++;
          }
        temp = temp -> next;
      }
      printf(" Even and odd num are  %d & %d  ", even,odd);
  }
```

CAMBRIDGE
INSTITUTE OF TECHNOLOGY
(Source : DIGINOTES)

15. To search for a given node in the SLL and display the status

```
void search_key()
{
        int key;
        node *new1,*prev,*temp = start;
        if(start == NULL)
         {
           printf("empty ");
           return;
         }
```

```
    printf("enter the key");
    scanf("%d",&key);
    while(temp != NULL && temp->info != key)
     {
       prev = temp;
       temp = temp->next;
     }

     if(temp == NULL)
     {
       printf("key not found");
       return;
     }
     printf(" key found");
}
```

16. write a function to insert a node before key element in SLL

```
void  ins_node_before_key()
{
    int key;
    node *new1,*prev,*temp = start;
    if(start == NULL)
     {
        printf("insertion not possible ");
        return;
     }

    printf("enter the key");
    scanf("%d",&key);
    while(temp != NULL && temp->info != key)
     {
       prev = temp;
       temp = temp->next;
     }

     if(temp == NULL)
```


Source : diginotes.in

```
        {
          printf("key not found");
          return;
        }

        new1 = getnode();
        if(start->info == key)
        {
          new1->next = start;
          start = new1;
          return;
        }

        new1->next = temp;
        prev->next = new1;

}
```

17. write a function to delete a key node in SLL

```
void del_key()
{
        int key;
        node *new1,*prev,*temp = start;

        if(start == NULL)
         {
           printf("list empty ");
           return;
         }
        printf("enter the key to be deleted");
        scanf("%d",&key);
        while(temp != NULL && temp->info != key)
         {
          prev = temp;
          temp = temp->next;
         }
```


CAMBRIDGE INSTITUTE OF TECHNOLOGY (Source : DIGINOTES)
Source : diginotes.in

```
        if(temp == NULL)
        {
          printf("key not found");
          return;
        }
        if(start->info == key )
        {
          start=start->next;
          free(temp);
          return;
        }

        if(temp->info == key && temp->next == NULL)
        {
          free(temp);
          prev->next=NULL;
          return;
        }

        prev->next = temp->next;
        free(temp);

}
```

18. Create SLL of integers and write C functions to perform the following

a. Create a node list with data 10,20 and 30
b. Insert a node with value 15 in between 10 and 20
c. Delete the node whose data is 20
d. Display the resulting SLL

```
struct sll
{
        int info;
        struct sll *next;
};
typedef struct sll node;
node *start = NULL;
```


Source : diginotes.in

```
void ins_front()
{
  node *new1;
  new1=getnode();
  new1->next = start;
  start=new1;
}

void ins_key_bef20()
{
        int key;
        node *new1,*prev,*temp = start;
        if(start == NULL)
         {
           printf("insertion not possible ");
           return;
         }
        while(temp != NULL && temp->info != 20)
         {
          prev = temp;
          temp = temp->next;
         }
         new1 = getnode(); /* 15 is stored in new1->info*/
         new1->next = temp;
         prev->next = new1;

}

void del_key20()
{
        int key;
        node *new1,*prev,*temp = start;

        if(start == NULL)
         {
           printf("list empty ");
           return;
```

CAMBRIDGE INSTITUTE OF TECHNOLOGY (Source : DIGINOTES)

Source : diginotes.in

```
        }
        while(temp != NULL && temp->info != 20)
        {
          prev = temp;
          temp = temp->next;
        }
        prev->next = temp->next;
        free(temp);
}

void display()
{
        node *temp=start;
        if(start == NULL)
        {
                printf("The sll is empty");
                return;
        }
        printf("The contents of sll are \n");
        while(temp!=NULL)
        {
          printf("%d  \n",temp->info);
          temp=temp->next;
        }
}

int main()
{
        int choice,n,i;
        clrscr();
          while(1)
          {
                printf("1. insert begin\t 2.insert key\t 3. Del key\t 4. Display \t 5.
                Exit\n");
                scanf("%d",&choice);
                switch(choice)
                {
                        case 1: for(i=0;i<3;i++)
```


Source : diginotes.in

```
                        ins_front(); /* this function is called 3 times with values
                                       entered first entered as 30 then  20 and
                                       last 10*/
                        break;
                case 2:ins_key_bef20();break;
                case 3:del_key20();break;
                case 4:display();break;
                case 5:printf("Exiting ... \n");exit(1);
                            break;
                default : printf("Invalid choice\n");
            }
        }
}
```

----------------------------------o----------------------------------

DATA STRUCTURES APPLICATIONS 15CS33

MODULE 4

TREES

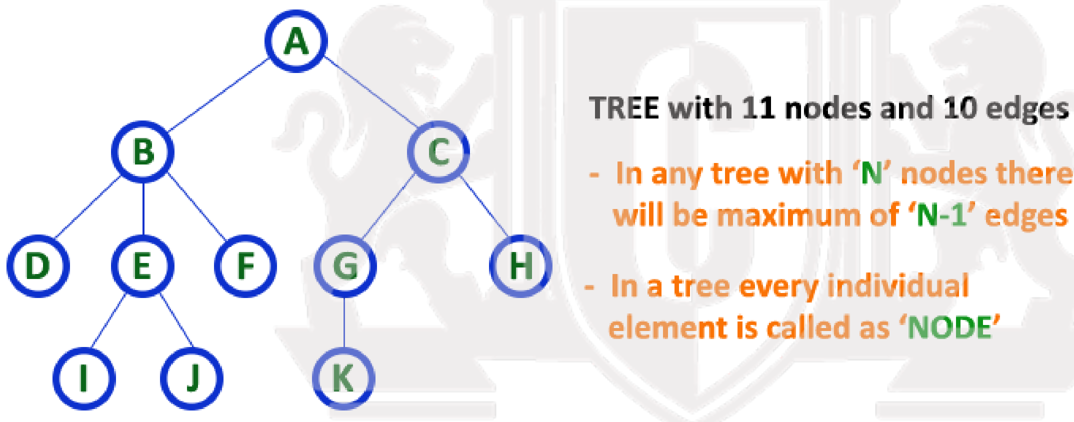
In linear data structure, data is organized in sequential order and in non-linear data structure, data is organized in random order. Tree is a very popular data structure used in wide range of applications. A tree data structure can be defined as follows...

Tree is a non-linear data structure which organizes data in hierarchical fashion and the tree structure follows a recursive pattern of organizing and storing data.

Every individual element is called as Node. Node in a tree data structure, stores the actual data of that particular element and link to next element in hierarchical structure.

if there are N number of nodes in a tree structure, then there can be a maximum of N-1 number of links.

Example



1. Root

In a tree data structure, the first node is called as Root Node. Every tree must have root node. We can say that root node is the origin of tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree. **Ex: 'A' in the above tree**

2. Edge

In a tree data structure, the connecting link between any two nodes is called as EDGE. In a tree with 'N' number of nodes there will be a maximum of 'N-1' number of edges. Ex: Line between two nodes.

3. Parent

In a tree data structure, the node which is predecessor of any node is called as PARENT NODE. In simple words, the node which has branch from it to any other node is called as parent node. Parent node can also be defined as "The node which has child / children".

Ex: A,B,C,E & G are parent nodes

4. Child

In a tree data structure, the node which is descendant of any node is called as CHILD Node. In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.Ex: B & C are children of A, G & H are children of C and K child of G

5. Siblings

In a tree data structure, nodes which belong to same Parent are called as SIBLINGS. In simple words, the nodes with same parent are called as Sibling nodes.Ex: B & C are siblings, D, E and F are siblings, G & H are siblings, I & J are siblings

6. Leaf

In a tree data structure, the node which does not have a child is called as LEAF Node. In simple words, a leaf is a node with no child. In a tree data structure, the leaf nodes are also called as External Nodes. External node is also a node with no child. In a tree, leaf node is also called as 'Terminal' node.Ex: D,I,J,F,K AND Hare leaf nodes

7. Internal Nodes

In a tree data structure, the node which has atleast one child is called as INTERNAL Node. In simple words, an internal node is a node with atleast one child.

In a tree data structure, nodes other than leaf nodes are called as Internal Nodes. The root node is also said to be Internal Node if the tree has more than one node. Internal nodes are also called as 'Non-Terminal' nodes.

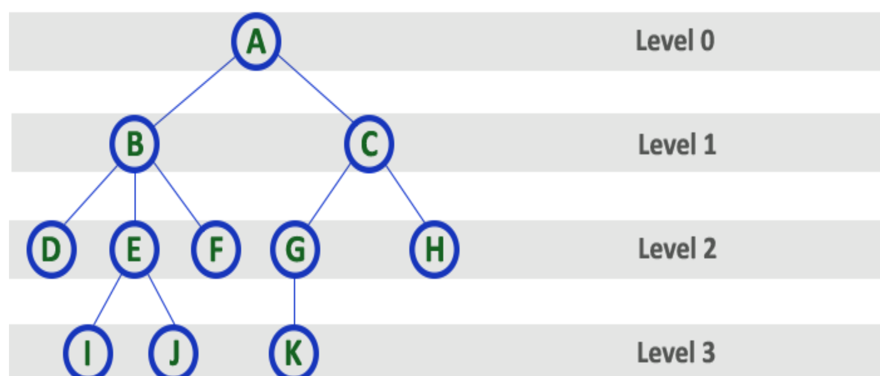
Ex: A,B,C,E & G

8. Degree of a node

In a tree data structure, the total number of children of a node is called as DEGREE of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nodes in a tree is called as **'Degree of Tree'.**Ex: Degree of B is 3, A is 2 and of F is 0

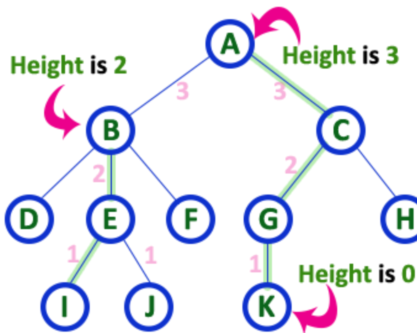
9. Level of a node

In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).



10. Height

In a tree data structure, the total number of egdes from leaf node to a particular node in the longest path is called as HEIGHT of that Node. In a tree, height of the root node is said to be height of the tree. In a tree, height of all leaf nodes is '0'.

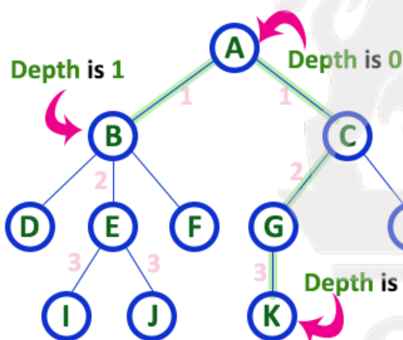


Here Height of tree is 3

- In any tree, 'Height of Node' is total number of Edges from leaf to that node in longest path.
- In any tree, 'Height of Tree' is the height of the root node.

11. Depth

In a tree data structure, the total number of egdes from root node to a particular node is called as DEPTH of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be Depth of the tree. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, depth of the root node is '0'.

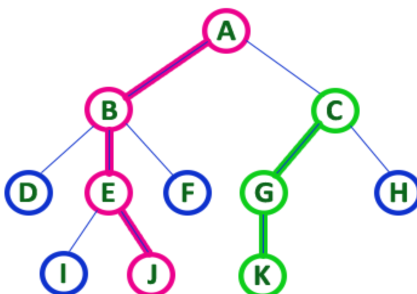


Here Depth of tree is 3

- In any tree, 'Depth of Node' is total number of Edges from root to that node.
- In any tree, 'Depth of Tree' is total number of edges from root to leaf in the longest path.

12. Path

In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as PATH between the two Nodes. Length of a Path is total number of nodes in that path. In below example the path A - B - E - J has length 4.



- In any tree, 'Path' is a sequence of nodes and edges between two nodes.

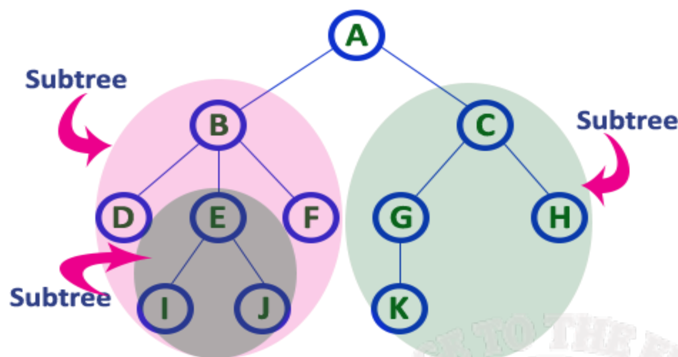
Here, 'Path' between A & J is
A - B - E - J

Here, 'Path' between C & K is
C - G - K

CAMBRIDGE INSTITUTE OF TECHNOLOGY

13. Sub Tree

In a tree data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.



The ***ancestors*** of a node are all the nodes along the path from the root to that node.

Ex: ancestor of j is B & A

A ***forest*** is a set of n 0 disjoint trees. The notion of a forest is very close to that of a tree because if we remove the root of a tree we get a forest. For example, in figure 1 if we remove A we get a forest with three trees.

General Tree Representations

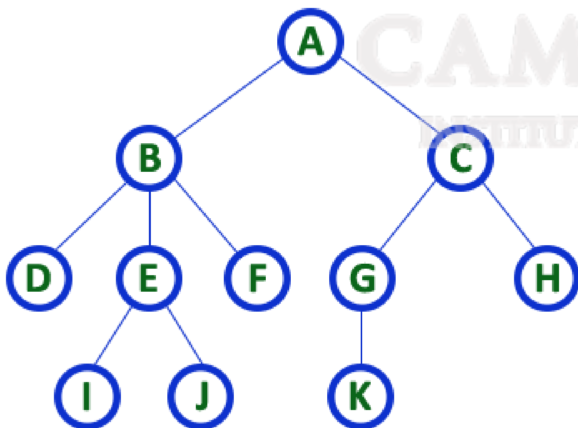
A general Tree Structure can be represented with the following three methods. Those methods are as follows...

1. List Representation

2. Left Child - Right Sibling Representation

3. Degree two Representation (Binary Tree Representation)

Consider the following tree...



TREE with 11 nodes and 10 edges

- **In any tree with 'N' nodes there will be maximum of 'N-1' edges**

- **In a tree every individual element is called as 'NODE'**

CAMBRIDGE

1. List Representation

There are several ways to draw a tree. One useful way is as a list. The tree in the above figure could be written as the list **(A(B(D,E(I,J),F),C(G(K),H)) – list representation (with rounded brackets).**The information in the root node comes first followed by a list of the subtrees of that node.Now, how do we

represent a tree in memory? If we wish to use linked lists, then a node must have a varying number of fields depending upon the number of branches.
Possible node structure for a tree of degree k called k-ary tree

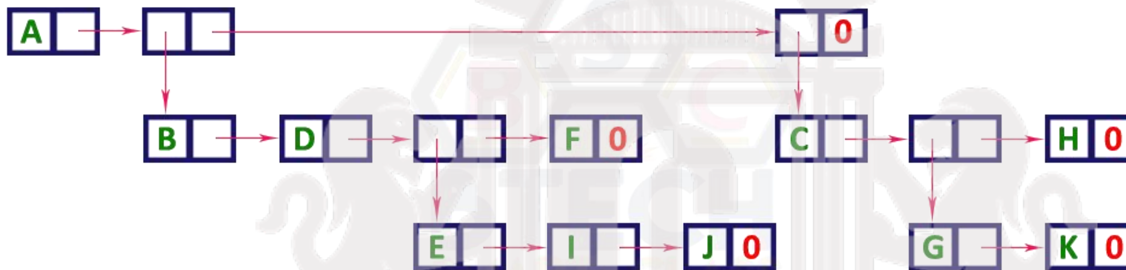
Data	link1	link2	....	link k

Each link field is used to point to a subtree. This node structure is cumbersome for the following reasons (1) Multiple node structure for different tree nodes (2) Waste of space (3) Excessive use of links.

The other alternate method is to have linked list of child nodes which allocates memory only for the nodes which have children.

In this representation, we use two types of nodes, one for representing the node with data and another for representing only references. We start with a node with data from root node in the tree. Then it is linked to an internal node through a reference node and is linked to any other node directly. This process repeats for all the nodes in the tree.

The above tree example can be represented using List representation as follows...



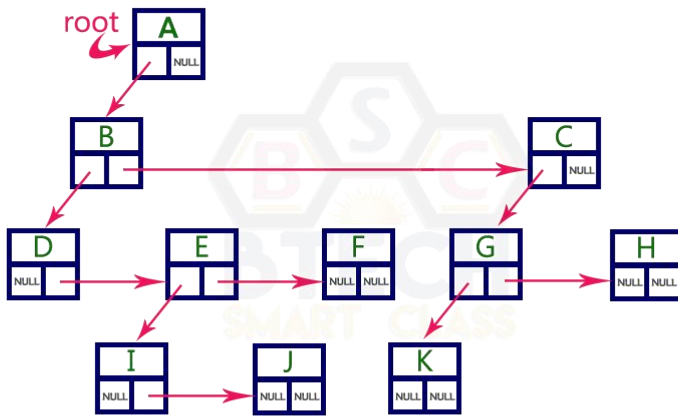
2. Left Child - Right Sibling Representation

In this representation, we use list with one type of node which consists of three fields namely Data field, Left child reference field and Right sibling reference field. Data field stores the actual value of a node, left reference field stores the address of the left child and right reference field stores the address of the right sibling node. Graphical representation of that node is as follows...

Data	
Left Child	Right Sibling

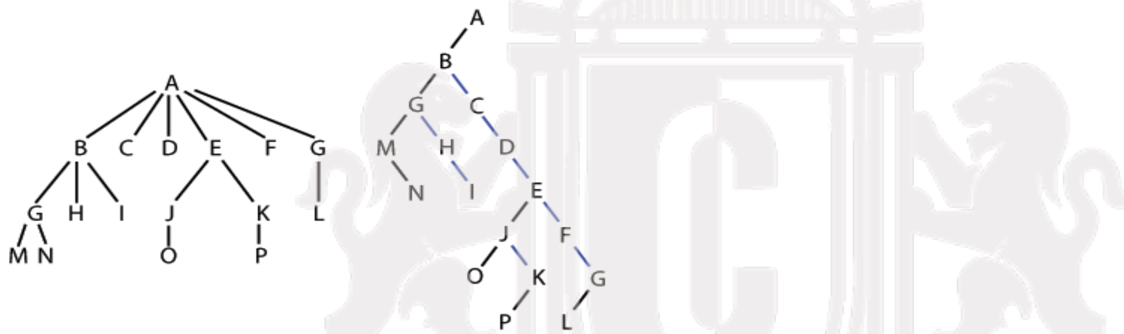
In this representation, every node's data field stores the actual value of that node. If that node has left child, then left reference field stores the address of that left child node otherwise that field stores NULL. If that node has right sibling then right reference field stores the address of right sibling node otherwise that field stores NULL. The above tree example can be represented using Left Child - Right Sibling representation as follows...

CAMBRIDGE INSTITUTE OF TECHNOLOGY



3. Degree two tree (Binary Tree)

The left child-right sibling representation can be converted to a degree two tree by simply rotating the right sibling pointers clockwise by 45 degrees.IN this representation, the two children of a node are called the left child and the right child. It is equivalent to converting a normal tree to binary tree. Degree two trees or left child-right child trees are nothing but binary trees.

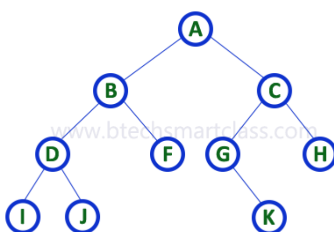


- Root of the general tree is the root of the binary tree.
- The first child node of any node in the general tree remains as the left subtree's root in the binary tree.Its right sibling in general tree becomes the right child node.

  Note: refer notes for examples.

Binary Tree

In a general tree, every node can have arbitrary number of children. Binary tree is a special type of tree data structure in which every node can have a maximum of 2 children. One is known as left child and the other is known as right child.**A tree in which every node can have a maximum of two children is called as Binary Tree.**In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children.Example

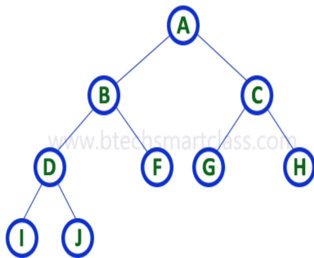


Types of Binary Trees

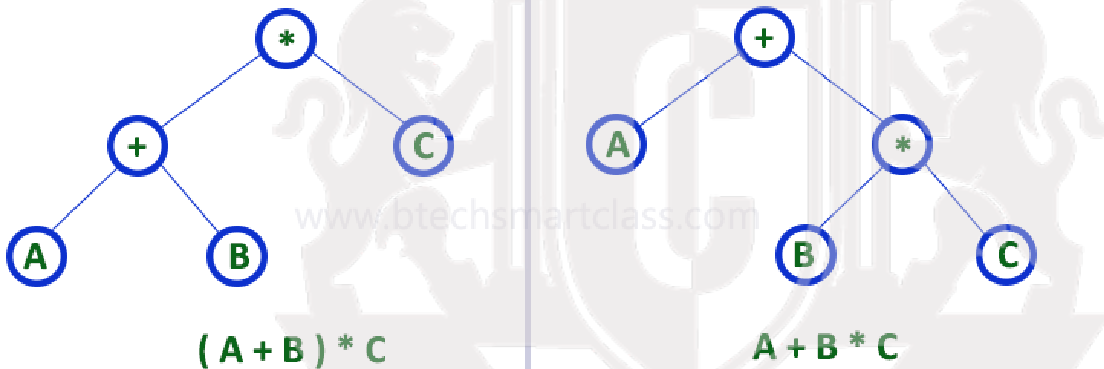
1. Strictly Binary Tree

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none. That means every internal node must have exactly two children. A strictly Binary Tree can be defined as follows...

A binary tree in which every node has either two or zero number of children is called Strictly Binary Tree. Strictly binary tree is also called as Full Binary Tree or Proper Binary Tree or 2-Tree.

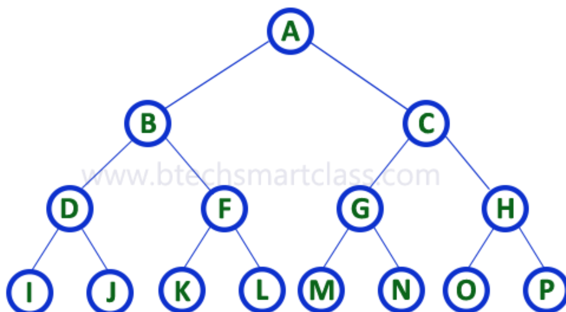


Strictly binary tree data structure is used to represent mathematical expressions.



2. Complete Binary Tree

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none and in complete binary tree all the nodes must have exactly two children and at every level of complete binary tree there must be 2level number of nodes. For example at level 2 there must be $2^2 = 4$ nodes and at level 3 there must be $2^3 = 8$ nodes.



BRIDGE TO THE FUTURE
CAMBRIDGE

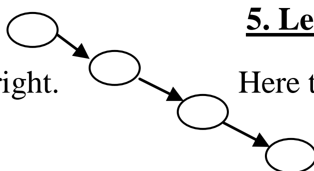
A binary tree in which every internal node has exactly two children and all leaf nodes are at same level is called Complete Binary Tree.Complete binary tree is also called as Perfect Binary Tree.

3. Almost complete Binary Tree

It is complete binary tree but completeness property is not followed in last level. In the above tree absence of leaf nodes L, M, N, O and P indicates its almost complete binary tree.

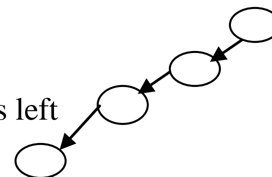
4. Right Skewed BT

Here the tree grows only towards right.



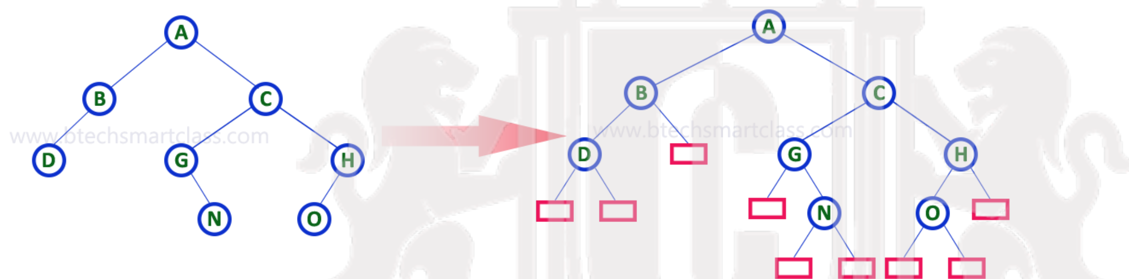
5. Left Skewed BT

Here the tree grows only towards left



6.Extended Binary Tree

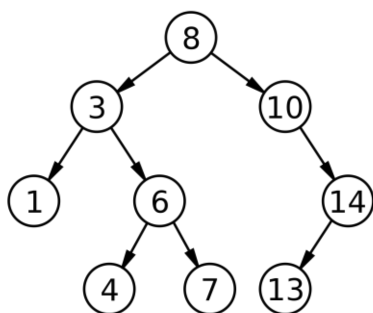
A binary tree can be converted into Full Binary tree by adding dummy nodes to existing nodes wherever required.The full binary tree obtained by adding dummy nodes to a binary tree is called as Extended Binary Tree.



In above figure, a normal binary tree is converted into full binary tree by adding dummy nodes (In pink colour).

7. Binary Search Tree(BST)

BST is a binary tree with a difference that for any node x, data of left subtree < data(x) and data of right subtree >= data(x). The above condition should be satisfied by all the nodes.



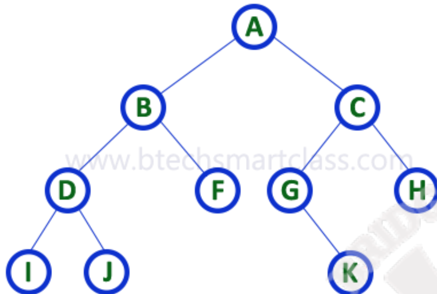
Binary Tree Representations

A binary tree data structure is represented using two methods. Those methods are as follows...

1. Array Representation

2. Linked List Representation

Consider the following binary tree...



1. Array Representation

In array representation of binary tree, we use a one dimensional array (1-D Array) to represent a binary tree. Consider the above example of binary tree and it is represented as follows...

To represent a binary tree of depth 'n' using array representation, we need one dimensional array with a maximum size of $2^{n+1} - 1$.

For any node with the position i, 2i + 1 gives the position of the left child & 2i + 2 gives the positon of the right child. For any node with a position i, its parent node position is identified by using formula (i-1) / 2.

If i is the position of the left child i+1 gives the position of right child.

Advantages of array representation

- Faster access
- Easy for implementation
- Good for complete binary trees

Disadvantages

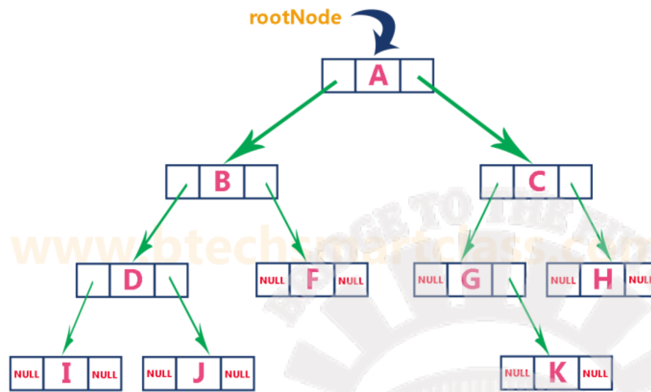
- Wastes memory for skewed trees
- Implementation of operations requires rearranging(shifting)of array elements

2. Linked List Representation

The linked notation uses a doubly linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.In this linked list representation, a node has the following structure...

Left Child Address	Data	Right Child Address

The above example of binary tree represented using Linked list representation is shown as follows...



Binary Tree Traversals

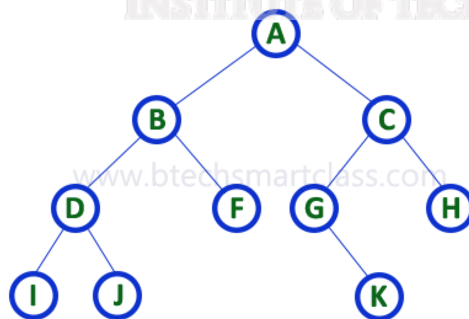
Tree traversal is a method of visiting the nodes of a tree in a particular order. The tree nodes are visited exactly once and displayed as they are visited.

Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal.

There are three types of binary tree traversals.

1. In - Order Traversal
2. Pre - Order Traversal
3. Post - Order Traversal

Consider the following binary tree...



1. In - Order Traversal (leftChild - root - rightChild)

In In-Order traversal, the root node is visited between left child and right child. In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting right child node. This in-order

CAMBRIDGE
INSTITUTE OF TECHNOLOGY

traversal is applicable for every root node of all subtrees in the tree. This is performed recursively for all nodes in the tree.

In the above example of binary tree, first we try to visit left child of root node 'A', but A's left child is a root node for left subtree. so we try to visit its (B's) left child 'D' and again D is a root for subtree with nodes D, I and J. So we try to visit its left child 'I' and it is the left most child. So first we visit 'I' then go for its root node 'D' and later we visit D's right child 'J'. With this we have completed the left part of node B. Then visit 'B' and next B's right child 'F' is visited. With this we have completed left part of node A. Then visit root node 'A'. With this we have completed left and root parts of node A. Then we go for right part of the node A. In right of A again there is a subtree with root C. So go for left child of C and again it is a subtree with root G. But G does not have left part so we visit 'G' and then visit G's right child K. With this we have completed the left part of node C. Then visit root node 'C' and next visit C's right child 'H' which is the right most child in the tree so we stop the process.

That means here we have visited in the order of I - D - J - B - F - A - G - K - C - H using In-Order Traversal.

In-Order Traversal for above example of binary tree is

I - D - J - B - F - A - G - K - C - H

2. Pre - Order Traversal (root - leftChild - rightChild)

In Pre-Order traversal, the root node is visited before left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child. This pre-order traversal is applicable for every root node of all subtrees in the tree.

In the above example of binary tree, first we visit root node 'A' then visit its left child 'B' which is a root for D and F. So we visit B's left child 'D' and again D is a root for I and J. So we visit D's left child 'I' which is the left most child. So next we go for visiting D's right child 'J'. With this we have completed root, left and right parts of node D and root, left parts of node B. Next visit B's right child 'F'. With this we have completed root and left parts of node A. So we go for A's right child 'C' which is a root node for G and H. After visiting C, we go for its left child 'G' which is a root for node K. So next we visit left of G, but it does not have left child so we go for G's right child 'K'. With this we have completed node C's root and left parts. Next visit C's right child 'H' which is the right most child in the tree. So we stop the process.

That means here we have visited in the order of A-B-D-I-J-F-C-G-K-H using Pre-Order Traversal.

Pre-Order Traversal for above example binary tree is

A - B - D - I - J - F - C - G - K - H

3. Post - Order Traversal (leftChild - rightChild - root)

In Post-Order traversal, the root node is visited after left child and right child. In this traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most node is visited.

Here we have visited in the order of I - J - D - F - B - K - G - H - C - A using Post-Order Traversal.

Post-Order Traversal for above example binary tree is

I - J - D - F - B - K - G - H - C - A

Note : refer notes for programs.

CAMBRIDGE
INSTITUTE OF TECHNOLOGY

Iterative Inorder Traversal

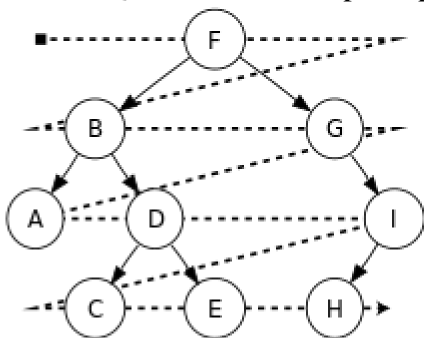
Traversal techniques using recursion consumes system stack space .The stack space used may not be acceptable for unbalanced trees of trees of larger heights.IN such cases, iterative traversal can be implemented by simulating stack space with the help of an array.Another solution would be to use threaded binary trees during traversal.

```
#define size 10
int top = -1;
struct Tree
{
    int data;
   struct Tree *lptr, *rptr;
};
typedef struct Tree node;
node *root,*stack[size];
void push(node *temp) //function to push
{
    if(top == size – 1)
    {
            Printf("stack full");
            Return;
    }
   stack[++top] = temp;
}
node *pop()  //function to pop
{
    if(top ==  – 1)
    {
            printf("stack empty");
            Return;
    }
     return(stack[top--]);
}

void iterative inorder(node *root)
{
    node *cur = root;
    while(1)
    {
         while(cur!=NULL)
         {
              push(cur);
              cur=cur->lptr;
          }
          if(top == -1)  break;
          cur = pop();
          printf("%d   ", cur->data);
          cur=cur->rptr;
     }
}
```

2. Level Order Traversal

- Level order traversal is a method of traversing the nodes of a tree level by level as in breadth first traversal.
- Level order traversal uses queue thus avoiding stack space usage.
- Here the nodes are numbered starting with the root on level zero continuing with nodes on level 1,2,3…..
- The nodes at any level is numbered from left to right
- Visiting the nodes using the ordering of levels is called level order traversal
- Queue uses FIFO principle



(ref :Wikipedia)

The level order traversal of the above tree is F B G A D I C E H

Implementation of level order traversal

```
void Level_Order(node *root)
{
   int f = 0, r = -1;  //global declaration
   node *q[size], *cur;
   q[++r] = root;

   while( r >= f)
   {
      cur = q[f++];
      printf("%d   ", cur->data);
      if(cur->lptr != NULL)
            q[++r] = cur->lptr;
      if(cur->rptr != NULL)
            q[++r] = cur->lptr;
     }
}
```

<u>Building Binary Tree from Traversal Pairs:</u>

Sometimes it is required to construct a binary tree if its traversals are known. From a single traversal it is not possible to construct unique binary tree. However any of the two traversals are given then the corresponding tree can be drawn uniquely:

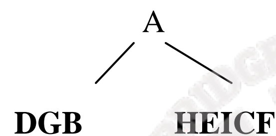
- Inorder and preorder
- Inorder and postorder
- Inorder and level order

The basic principle for formulation is as follows:

If the preorder traversal is given, then the first node is the root node. If the postorder traversal is given then the last node is the root node. Once the root node is identified, all the nodes in the left sub-trees and right sub-trees of the root node can be identified using inorder.

Same technique can be applied repeatedly to form sub-trees.
It can be noted that, for the purpose mentioned, two traversal are essential out of which one should be inorder traversal and another preorder or postorder; alternatively, given preorder and postorder traversals, binary tree cannot be obtained uniquely.

Example 1:
Construct a binary tree from a given preorder and inorder sequence:
Preorder: A B D G C E H I F Inorder: D G B A H E I C F
Solution:
From Preorder sequence A B D G C E H I F, the root is: A
From Inorder sequence D G B A H E I C F, we get the left and right sub trees:
Left sub tree is: D G B
Right sub tree is: H E I C F

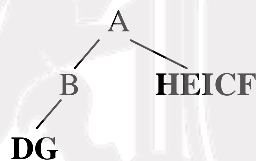


To find the root, left and right sub trees for D G B:

From the preorder sequence B D G, the root of tree is: B

From the inorder sequence D G B, we can find that D and G are to the left of B.

The Binary tree upto this point looks like:

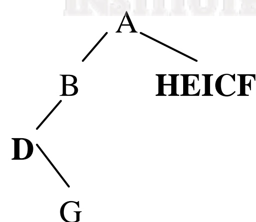


To find the root, left and right sub trees for D G:

From the preorder sequence D G, the root of the tree is: D

From the inorder sequence D G, we can find that there is no left node to D and G is at the right of D.

The Binary tree upto this point looks like

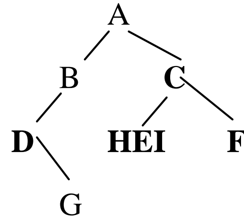


To find the root, left and right sub trees for H E I C F:

From the preorder sequence C E H I F, the root of the left sub tree is: C

From the inorder sequence H E I C F, we can find that H E I are at the left of C and F is at the right of C.

The Binary tree upto this point looks like:

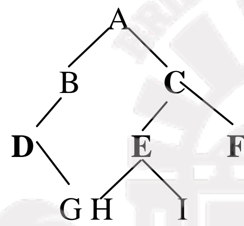


To find the root, left and right sub trees for H E I:

From the preorder sequence E H I, the root of the tree is: E

From the inorder sequence H E I, we can find that H is at the left of E and I is at the right of E.

The Binary tree upto this point looks like:



Refer class notes for other examples

Properties of Binary Tree

1. Prove that max no of nodes in a BT of depth K is $2^k - 1$

Total no of nodes = $2^0 + 2^1 + 2^2 + \ldots\ldots + 2^i$

The above sequence is in geometric progression

$= (2^{i+1} - 1) / (2 - 1)$

$= 2^{i+1} - 1$

$= 2^n - 1$ (where $i+1 = n$ = height of tree or depth of tree)

=> max no of nodes in a BT of depth $K = 2^k - 1$

2. Max no of nodes on level i of a BT is 2^{i-1}, $i>=1$ or 2^i, $i>=0$

Proof by induction on i;

Induction base: Root is the only node on level i = 1.

- ➔ Hence max no of nodes on level i =1
  - ⇨ $2^{i-1} = 2^0 = 1$

Induction Hypothesis:

- ➔ Let I be an arbitrary positive integer > 1
- ➔ Assume that max no of nodes on level i-1 is 2

Induction Step:

- ➔ We know that max no of nodes on level i-1-2 $^{i-2}$ by induction hypothesis
- ➔ We know that each node in a BT max degree is 2
- ➔ Max no of nodes on level i = twice the max no of nodes on level i-2 i.e 2 $^{i-1}$

Hence the proof

3. Prove that no of leaf nodes = no of nodes of degree-2 nodes or for any nonempty Binary Tree T, if N_0 is the no of leaf nodes and N_2 no of nodes of degree 2 then $N_0 = N_2 + 1$

Proof: Let N_1 be the no of nodes of degree 1

Let N be the total no of nodes

Since all nodes in T are atmost of degree 2 we have

$N = N_0 + N_1 + N_2$--(1)

If we count no of branches in a BT we see that every node except the root has a branch leading into it.

If B →no of branches then

N = B + 1(because all branches step from a node of degree 1 or 2)

Therefore B = N1 + 2N2

- ⇨ N = B + 1
- ⇨ N = N1 + 2N2 + 1---(2)
- ⇨ From (1) & (2) we get

$N_0 + N_1 + N_2 = N_1 + 2N_2 + 1$

$N_0 = N_2 + 1$

Hence the proof

Binary Search Tree

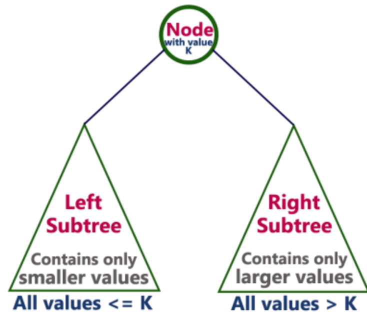
In a binary tree, every node can have maximum of two children but there is no order of nodes based on their values. In binary tree, the elements are arranged as they arrive to the tree, from top to bottom and left to right.

A binary tree has the following time complexities...

- ➔ Search Operation - O(n)
- ➔ Insertion Operation - O(1)
- ➔ Deletion Operation - O(n)

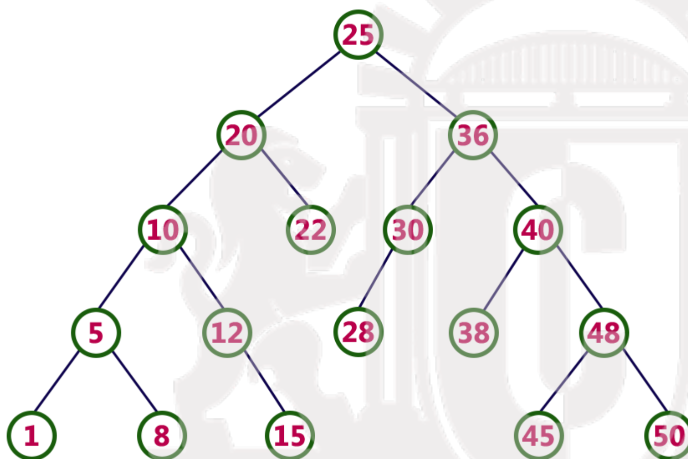
To enhance the performance of binary tree, we use special type of binary tree known as Binary Search Tree. Binary search tree mainly focus on the search operation in binary tree. Binary search tree can be defined as follows...

Binary Search Tree is a binary tree in which every node contains only smaller values in its left subtree and only larger values in its right subtree.



Example

The following tree is a Binary Search Tree. In this tree, left subtree of every node contains nodes with smaller values and right subtree of every node contains larger values.



Every Binary Search Tree is a binary tree but all the Binary Trees need not to be binary search trees.

Operations on a Binary Search Tree

The following operations are performed on a binary search tree...

- Search
- Insertion
- Deletion
- Traversal

Search Operation in BST

In a binary search tree, the search operation is performed with O(log n) time complexity. The search operation is performed as follows...

Step 1: Read the search element from the user
Step 2: Compare, the search element with the value of root node in the tree.

BRIDGE TO THE FUTURE
CAMBRIDGE
INSTITUTE OF TECHNOLOGY

Step 3: If both are matching, then display "Given node found!!!" and terminate the function
Step 4: If both are not matching, then check whether search element is smaller or larger than that node value.
Step 5: If search element is smaller, then continue the search process in left subtree.
Step 6: If search element is larger, then continue the search process in right subtree.
Step 7: Repeat the same until we find exact element or we completed with a leaf node
Step 8: If we reach the node with search value, then display "Element is found" and terminate the function.
Step 9: If we reach a leaf node and it is also not matching, then display "Element not found" and terminate the function.

Insertion Operation in BST

In a binary search tree, the insertion operation is performed with O(log n) time complexity. In binary search tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

Step 1: Create a newNode with given value and set its left and right to NULL.
Step 2: Check whether tree is Empty.
Step 3: If the tree is Empty, then set set root to newNode.
Step 4: If the tree is Not Empty, then check whether value of newNode is smaller or larger than the node (here it is root node).
Step 5: If newNode is smaller than or equal to the node, then move to its left child. If newNode is larger than the node, then move to its right child.
Step 6: Repeat the above step until we reach to a leaf node (e.i., reach to NULL).
Step 7: After reaching a leaf node, then isert the newNode as left child if newNode is smaller or equal to that leaf else insert it as right child.

Deletion Operation in BST

In a binary search tree, the deletion operation is performed with O(log n) time complexity. Deleting a node from Binary search tree has following three cases...

Case 1: Deleting a Leaf node (A node with no children)
Case 2: Deleting a node with one child
Case 3: Deleting a node with two children

Case 1: Deleting a leaf node

We use the following steps to delete a leaf node from BST...

Step 1: Find the node to be deleted using search operation

Step 2: Delete the node using free function (If it is a leaf) and terminate the function.

Case 2: Deleting a node with one child

We use the following steps to delete a node with one child from BST...

Step 1: Find the node to be deleted using search operation

Step 2: If it has only one child, then create a link between its parent and child nodes.

Step 3: Delete the node using free function and terminate the function.

Case 3: Deleting a node with two children

We use the following steps to delete a node with two children from BST...

Step 1: Find the node to be deleted using search operation

Step 2: If it has two children, then find the largest node in its left subtree (OR) the smallest node in its right subtree.

Step 3: Swap both deleting node and node which found in above step.

Step 4: Then, check whether deleting node came to case 1 or case 2 else goto steps 2

Step 5: If it comes to case 1, then delete using case 1 logic.

Step 6: If it comes to case 2, then delete using case 2 logic.

Step 7: Repeat the same process until node is deleted from the tree.

Example

Construct a Binary Search Tree by inserting the following sequence of numbers...

10,12,5,4,20,8,7,15 and 13

Refer notes for examples and implementation.

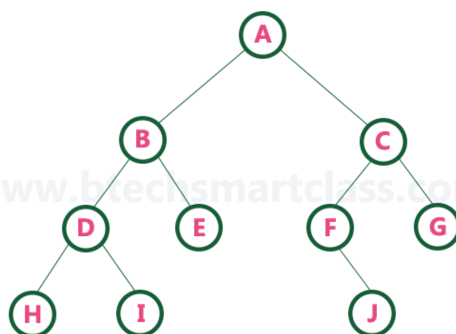
Threaded Binary Tree

A binary tree is represented using array representation or linked list representation. When a binary tree is represented using linked list representation, if any node is not having a child we use NULL pointer in that position. In any binary tree linked list representation, there are more number of NULL pointer than actual pointers. Generally, in any binary tree linked list representation, if there are 2N number of reference fields, then N+1 number of reference fields are filled with NULL (N+1 are NULL out of 2N). This NULL pointer does not play any role except indicating there is no link (no child).

A. J. Perlis and C. Thornton have proposed new binary tree called "Threaded Binary Tree", which make use of NULL pointer to improve its traversal processes. In threaded binary tree, NULL pointers are replaced by references to other nodes in the tree, called threads.

Threaded Binary Tree is also a binary tree in which all left child pointers that are NULL (in Linked list representation) points to its in-order predecessor, and all right child pointers that are NULL (in Linked list representation) points to its in-order successor. If there is no in-order predecessor or in-order successor, then it points to root node.

Consider the following binary tree...

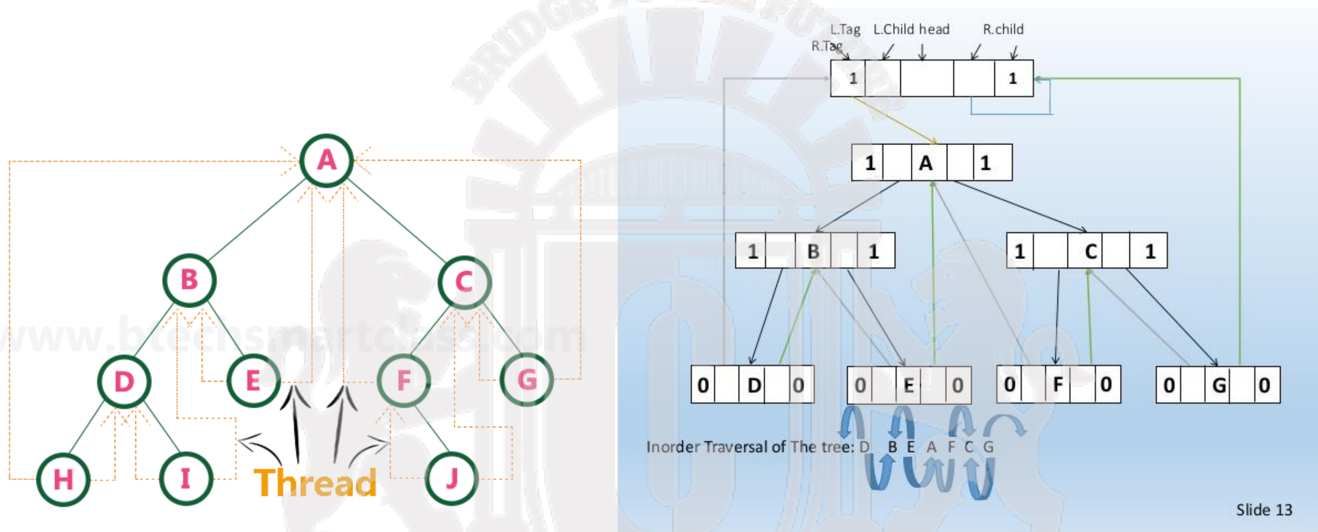


To convert above binary tree into threaded binary tree, first find the in-order traversal of that tree...

In-order traversal of above binary tree...

H - D - I - B - E - A - F - J - C - G

When we represent above binary tree using linked list representation, nodes H, I, E, F, J and G left child pointers are NULL. This NULL is replaced by address of its in-order predecessor, respectively (I to D, E to B, F to A, J to F and G to C), but here the node H does not have its in-order predecessor, so it points to the root node A. And nodes H, I, E, J and G right child pointers are NULL. This NULL pointers are replaced by address of its in-order successor, respectively (H to D, I to B, E to A, and J to C), but here the node G does not have its in-order successor, so it points to the root node A. The above example binary tree in threaded notation is as follows:



In above figure threads are indicated with dotted links.

Note :Refer notes for implementation of TBT

Expression Trees:

A Binary Expression Tree is …

A special kind of binary tree in which:

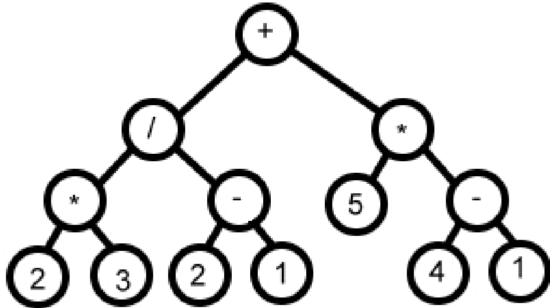
1. Each leaf node contains a single operand

2. Each nonleaf node contains a single binary operator

3. The left and right subtrees of an operator node represent subexpressions that must be evaluated before applying the operator at the root of the subtree.

CAMBRIDGE
INSTITUTE OF TECHNOLOGY

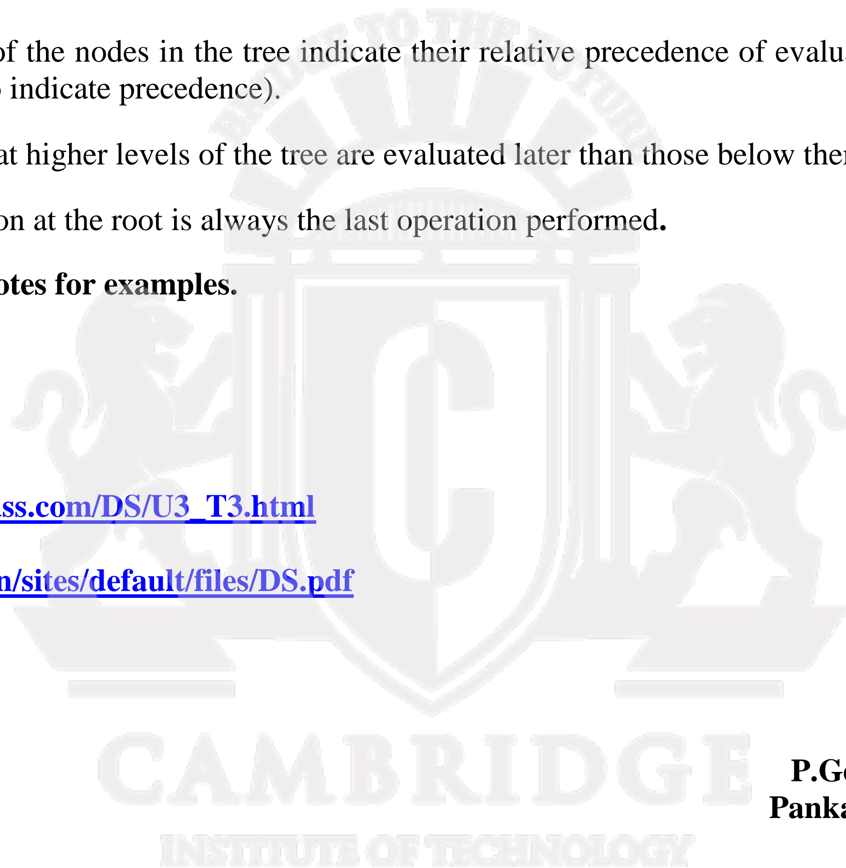
Example :



Expression tree for 2*3/(2-1)+5*(4-1)

- Levels Indicate Precedence
- The levels of the nodes in the tree indicate their relative precedence of evaluation (we do not need parentheses to indicate precedence).
- Operations at higher levels of the tree are evaluated later than those below them.
- The operation at the root is always the last operation performed**.**

Refer class notes for examples.



References:

http://btechsmartclass.com/DS/U3_T3.html

http://www.iare.ac.in/sites/default/files/DS.pdf

Prepared by

P.Geetha. Asst Professor
Pankaja K, Asst Professor

Department of CSE,
Cambridge Institute of Technology

MODULE 5

GRAPHS, HASHING, SORTING, FILES

A graph is an abstract data structure that is used to implement the mathematical concept of graphs. It is basically a collection of vertices (also called nodes) and edges that connect these vertices. A graph is often viewed as a generalization of the tree structure, where instead of having a purely parent-to-child relationship between tree nodes, any kind of complex relationship can exist.

Graphs - Terminology and Representation

Definitions: Graph, Vertices, Edges

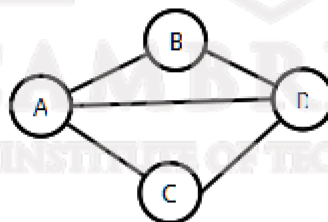
- Define a graph *G = (V, E)* by defining a pair of sets:
  1. V = a set of **vertices**
  2. E = a set of **edges**
- Edges:
  - Each edge is defined by a pair of vertices
  - An edge **connects** the vertices that define it

- Vertices:

  - Vertices also called **nodes**
  - Denote vertices with labels

  Representation:
  - Represent vertices with circles, perhaps containing a label
  - Represent edges with lines between circles

  Example:
  - V = {A,B,C,D}
  - E = {(A,B),(A,C),(A,D),(B,D),(C,D)}



  Many algorithms use a graph representation to represent data or the problem to be solved

- Examples of Graph applications:
  - Cities with distances between
  - Roads with distances between intersection points
  - Course prerequisites
  - Network and shortest routes
  - Social networks
  - Electric circuits, projects planning and many more...

Graph Classifications

- There are several common kinds of graphs
  - Weighted or unweighted
  - Directed or undirected

- Cyclic or acyclic
- Multigraphs

Kinds of Graphs: Weighted and Unweighted

- Graphs can be classified by whether or not their edges have **weights**
- **Weighted graph**: edges have a weight
    - Weight typically shows cost of traversing
    - Example: weights are distances between cities
- **Unweighted graph**: edges have no weight
    - Edges simply show connections
    - Example: course prerequisites

Kinds of Graphs: Directed and Undirected

- Graphs can be classified by whether or their edges are have direction
    - **Undirected Graphs**: each edge can be traversed in **either direction**
    - **Directed Graphs**: each edge can be traversed **only in a specified direction**

Undirected Graphs

- **Undirected Graph**: no implied direction on edge between nodes
    - The example from above is an undirected graph

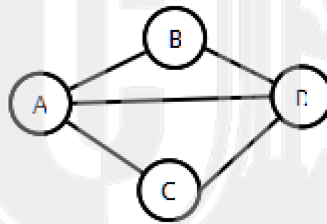


fig 1

- In diagrams, edges have no direction (ie there are no arrows)
- Can traverse edges in either directions
- In an undirected graph, an edge is an **unordered** pair
    - Actually, an edge is a set of 2 nodes, but for simplicity we write it with parenthesis
        - For example, we write (A, B) instead of {A, B}
        - Thus, (A,B) = (B,A), etc
        - If (A,B) $\in$ E then (B,A) $\in$ E

Directed Graphs

- **Digraph**: A graph whose edges are directed (ie have a direction)
    - Edge drawn as arrow
    - Edge can only be traversed in direction of arrow
    - Example: E = {(A,B), (A,C), (A,D), (B,C), (D,C)}

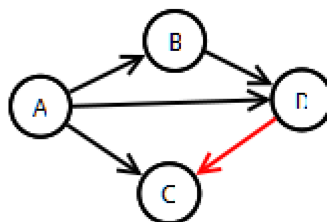


fig 2

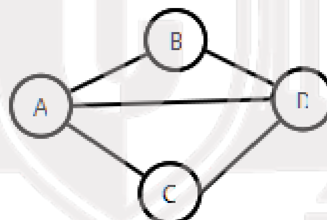
- In a digraph, an edge is an **ordered** pair
    - Thus: (u,v) and (v,u) are not the same edge
    - In the example, (D,C) ∈ E, (C,D) ∉ E

Degree of a Node

- The **degree** of a node is the number of edges incident on it.
- In the example above: (fig 1)
    - Degree 2: B and C
    - Degree 3: A and D
- A and D have **odd degree**, and B and C have **even degree**
- Can also define **in-degree** and **out-degree**
    - In-degree: Number of edges pointing **to** a node
    - Out-degree: Number of edges pointing **from** a node

Graphs: Terminology Involving Paths

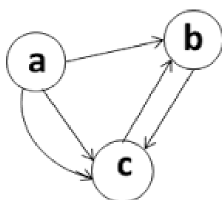
- **Path**: sequence of vertices in which each pair of successive vertices is connected by an edge
- **Cycle**: a path that starts and ends on the same vertex
- **Simple path**: a path that does not cross itself
    - That is, no vertex is repeated (except first and last)
- Simple paths cannot contain cycles
- **Length** of a path: Number of edges in the path
- Examples



Cyclic and Acyclic Graphs

- A **Cyclic** graph contains cycles
    - Example: roads (normally)
- An **acyclic** graph contains no cycles
    - Example: Course prerequisites

Multigraph: A graph with self loops and parallel edges is called a multigraph.



Connected and Unconnected Graphs and Connected Components

- An *undirected* graph is **connected** if every pair of vertices has a path between it
    - Otherwise it is unconnected

- A *directed* graph is **strongly connected** if every pair of vertices has a path between them, in **both directions**

Data Structures for Representing Graphs

- Two common data structures for representing graphs:
  - Adjacency lists
  - Adjacency matrix

Adjacency List Representation

An adjacency list is a way in which graphs can be represented in the computer's memory. This structure consists of a list of all nodes in G. Furthermore, every node is in turn linked to its own list that contains the names of all other nodes that are adjacent to it. The key advantages of using an adjacency list are:

- It is easy to follow and clearly shows the adjacent nodes of a particular node.
- It is often used for storing graphs that have a small-to-moderate number of edges. That is, an adjacency list is preferred for representing sparse graphs in the computer's memory; otherwise, an adjacency matrix is a good choice.
- Adding new nodes in G is easy and straightforward when G is represented using an adjacency list. Adding new nodes in an adjacency matrix is a difficult task, as the size of the matrix needs to be changed and existing nodes may have to be reordered. Each node has a list of adjacent nodes

Example (undirected graph): **(fig 1)**

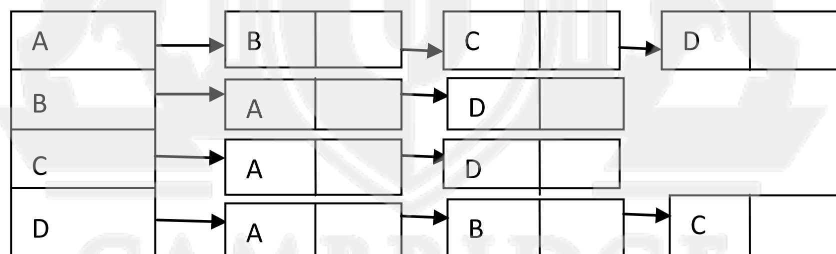


Fig (1) adjacency lsit for the graph of fig3

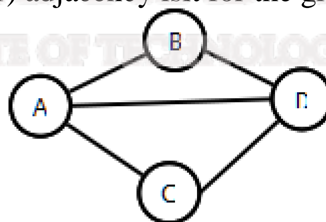


fig 3

- Example (directed graph):
- A: B, C, D
- B: D
- C: Nil
- D: C

Adjacency Matrix Representation

An adjacency matrix is used to represent which nodes are adjacent to one another. By definition, two nodes are said to be adjacent if there is an edge connecting them. In a directed graph G, if node v is adjacent to node u, then there is definitely an edge from u to v. That is, if v is adjacent to u, we can get from u to v by traversing one edge. For any graph G having n nodes, the adjacency matrix will have the dimension of n * n. In an adjacency matrix, the rows and columns are labelled by graph vertices. An entry aij in the adjacency matrix will contain 1, if vertices vi and vj are adjacent to each other. However, if the nodes are not adjacent, aij will be set to zero. It. Since an adjacency matrix contains only 0s and 1s, it is called a bit matrix or a Boolean matrix. The entries in the matrix depend on the ordering of the nodes in G. Therefore, a change in the order of nodes will result in a different adjacency matrix.

Aij = 1 if there is an edge from Vi to Vj

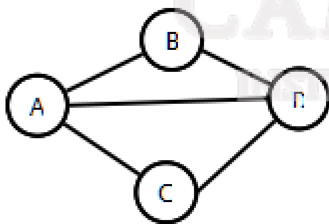
0 otherwise

Adjacency Matrix: 2D array containing weights on edges

- Row for each vertex
- Column for each vertex
- Entries contain weight of edge from row vertex to column vertex
- Entries contain ∞ if no edge from row vertex to column vertex
- Entries contain 0 on diagonal (if self edges not allowed)

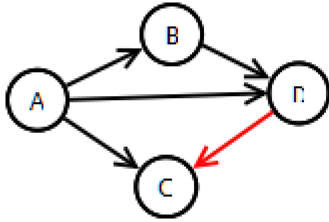
- Example undirected graph (assume self-edges not allowed):

	A	B	C	D
A	0	1	1	1
B	1	0	∞	1
C	1	∞	0	1
D	1	1	1	0



- Example directed graph (assume self-edges allowed):

	A	B	C	D
A	∞	1	1	1
B	∞	∞	∞	1
C	∞	∞	∞	∞
D	∞	∞	1	∞



<u>Disadv</u>:Adjacency matrix representation is easy to represent and feasible as long as the graph is small and connected. For a large graph ,whose matrix is sparse, adjacency matrix representation wastes a lot of memory. Hence list representation is preferred over matrix representation.

<u>Graph traversal algorithms</u>

Traversing a graph, is the method of examining the nodes and edges of the graph. There are two standard methods of graph traversal. These two methods are:

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

While breadth-first search uses a queue as an auxiliary data structure to store nodes for further processing, the depth-first search scheme uses a stack.

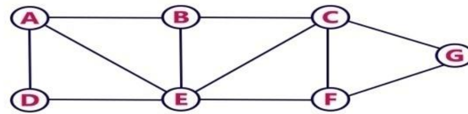
<u>Breadth-first search algorithm</u>

Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighbouring nodes. Then for each of those nearest nodes, the algorithm explores their unexplored neighbour nodes, and so on, until it finds the goal. That is, we start examining the node A and then all the neighbours of A are examined. In the next step, we examine the neighbours of neighbours of A, so on and so forth. This means that we need to track the neighbours of the node and guarantee that every node in the graph is processed and no node is processed more than once. This is accomplished by using a queue that will hold the nodes that are waiting for further processing.

<u>Algorithm for BFS traversal</u>

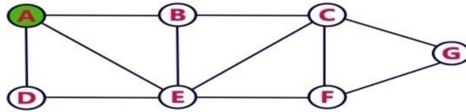
- **Step 1:** Define a Queue of size total number of vertices in the graph.
- **Step 2:** Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.
- **Step 3:** Visit all the **adjacent** vertices of the verex which is at front of the Queue which is not visited and insert them into the Queue.
- **Step 4:** When there is no new vertex to be visit from the vertex at front of the Queue then delete that vertex from the Queue.
- **Step 5:** Repeat step 3 and 4 until queue becomes empty.
- **Step 6:** When queue becomes Empty, then the enqueue or dequeue order gives the BFS traversal order.

Consider the following example graph to perform BFS traversal



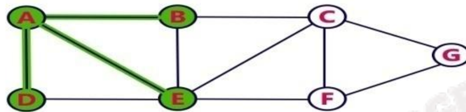
Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.



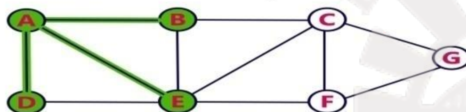
Step 2:

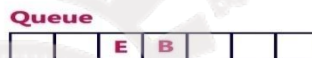
- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..



Step 3:

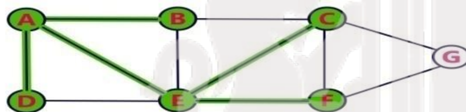
- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.





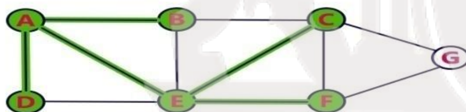
Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.



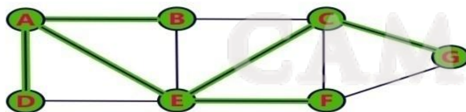
Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.



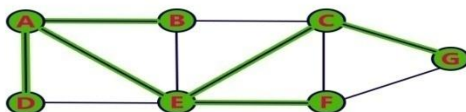
Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.



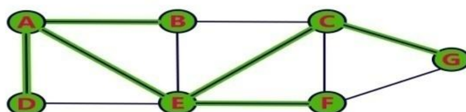
Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

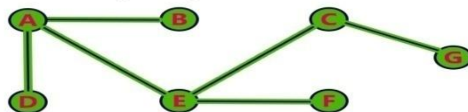


Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



BRIDGE TO THE FUTURE
CAMBRIDGE
INSTITUTE OF TECHNOLOGY

Depth-first Search Algorithm

Depth-first search begins at a starting node A which becomes the current node. Then, it examines each node N along a path P which begins at A. That is, we process a neighbour of A, then a neighbour of neighbour of A, and so on.

During the execution of the algorithm, if we reach a path that has a node N that has already been processed, then we backtrack to the current node. Otherwise, the unvisited (unprocessed) node becomes the current node. The algorithm proceeds like this until we reach a dead-end (end of path P). On reaching the deadend, we backtrack to find another path P. The algorithm terminates when backtracking leads back to the starting node A.

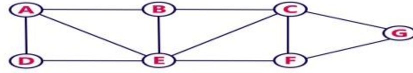
In this algorithm, edges that lead to a new vertex are called discovery edges and edges that lead to an already visited vertex are called back edges. Observe that this algorithm is similar to the in-order traversal of a binary tree. Its implementation is similar to that of the breadth-first search algorithm but here we use a stack instead of a queue.

We use the following steps to implement DFS traversal...

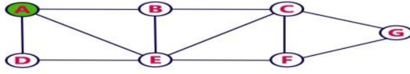
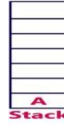
- **Step 1:** Define a Stack of size total number of vertices in the graph.
- **Step 2:** Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.
- **Step 3:** Visit any one of the **adjacent** vertex of the verex which is at top of the stack which is not visited and push it on to the stack.
- **Step 4:** Repeat step 3 until there are no new vertex to be visit from the vertex on top of the stack.
- **Step 5:** When there is no new vertex to be visit then use **back tracking** and pop one vertex from the stack.
- **Step 6:** Repeat steps 3, 4 and 5 until stack becomes Empty.
- **Step 7:** When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

CAMBRIDGE
INSTITUTE OF TECHNOLOGY

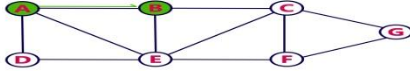
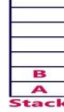
Consider the following example graph to perform DFS traversal



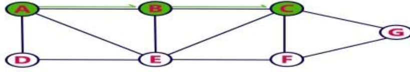
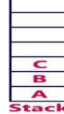
Step 1:
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.

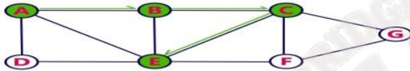
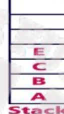
Step 2:
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.

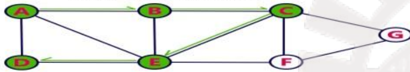
Step 3:
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.

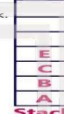
Step 4:
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack

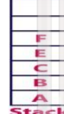
Step 5:
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack



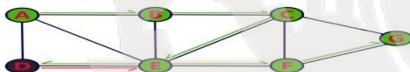
Step 6:
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.



Step 7:
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.

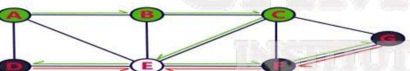


Step 8:
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.



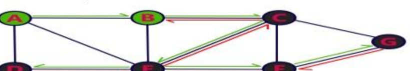
Step 9:
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.

Step 10:
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

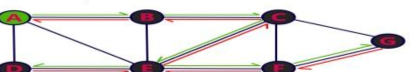


Step 11:
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.

Step 12:
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.



Step 13:
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.



Step 14:
- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.

- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.

CAMBRIDGE
INSTITUTE OF TECHNOLOGY

Applications OF graphs

- Graphs are constructed for various types of applications such as:
- In circuit networks where points of connection are drawn as vertices and component wires become the edges of the graph.
- In transport networks where stations are drawn as vertices and routes become the edges of the graph.
- In maps that draw cities/states/regions as vertices and adjacency relations as edges.
- In program flow analysis where procedures or modules are treated as vertices and calls to these procedures are drawn as edges of the graph.
- Once we have a graph of a particular concept, they can be easily used for finding shortest paths, project planning, etc.
- In flowcharts or control-flow graphs, the statements and conditions in a program are represented as nodes and the flow of control is represented by the edges.
- In state transition diagrams, the nodes are used to represent states and the edges represent legal moves from one state to the other.
- Graphs are also used to draw activity network diagrams. These diagrams are extensively used as a project management tool to represent the interdependent relationships between groups, steps, and tasks that have a significant impact on the project.

Introduction to sorting

Sorting means arranging the elements of an array so that they are placed in some relevant order which may be either ascending or descending. That is, if A is an array, then the elements of A are arranged in a sorted order (ascending order) in such a way that A[0] < A[1] < A[2] < ...... < A[N]. For example, if we have an array that is declared and initialized as int A[] = {21, 34, 11, 9, 1, 0, 22}; Then the sorted array (ascending order) can be given as: A[] = {0, 1, 9, 11, 21, 22, 34; A sorting algorithm is defined as an algorithm that puts the elements of a list in a certain order, which can be either numerical order, lexicographical order, or any user-defined order

Insertion Sort

Insertion sort is a very simple sorting algorithm in which the sorted array (or list) is built one element at a time. We all are familiar with this technique of sorting, as we usually use it for ordering a deck of cards while playing bridge. The main idea behind insertion sort is that it inserts each item into its proper place in the final list. To save memory, most implementations of the insertion sort algorithm work by moving the current data element past the already sorted values and repeatedly interchanging it with the preceding value until it is in its correct place. Insertion sort is less efficient as compared to other more advanced algorithms such as quick sort, heap sort, and merge sort.

Technique:

- The array of values to be sorted is divided into two sets. One that stores sorted values and another that contains unsorted values.
- The sorting algorithm will proceed until there are elements in the unsorted set.
- Suppose there are n elements in the array. Initially, the element with index 0 (assuming LB = 0) is in the sorted set. Rest of the elements are in the unsorted set.
- The first element of the unsorted partition has array index 1 (if LB = 0).
- During each iteration of the algorithm, the first element in the unsorted set is picked up and inserted into the correct position in the sorted set.

Consider the following unsorted list of elements...

15	20	10	30	50	18	5	45

Asume that sorted portion of the list empty and all elements in the list are in unsorted portion of the list as shown in the figure below...

Sorted | Unsorted

15	20	10	30	50	18	5	45

Move the first element 15 from unsorted portion to sorted portion of the list.

Sorted | Unsorted

15	20	10	30	50	18	5	45

To move element 20 from unsorted to sorted portion, Compare 20 with 15 and insert it at correct position

Sorted | Unsorted

15	20	10	30	50	18	5	45

To move element 10 from unsorted to sorted portion, Compare 10 with 20 and it is smaller so swap. Then compare 10 with 15 again smaller swap. And 10 is insert at its correct position in sorted portion of the list.

Sorted | Unsorted

10	15	20	30	50	18	5	45

To move element 30 from unsorted to sorted portion, Compare 30 with 20, 15 and 10. And it is larger than all these so 30 is directly inserted at last position in sorted portion of the list.

Sorted | Unsorted

10	15	20	30	50	18	5	45

To move element 50 from unsorted to sorted portion, Compare 50 with 30, 20, 15 and 10. And it is larger than all these so 50 is directly inserted at last position in sorted portion of the list.

Sorted | Unsorted

10	15	20	30	50	18	5	45

To move element 18 from unsorted to sorted portion, Compare 18 with 30, 20 and 15. Since 18 is larger than 15, move 20, 30 and 50 one position to the right in the list and insert 18 after 15 in the sorted portion.

Sorted | Unsorted

10	15	18	20	30	50	5	45

To move element 5 from unsorted to sorted portion, Compare 5 with 50, 30, 20, 18, 15 and 10. Since 5 is smaller than all these element, move 10, 15, 18, 20, 30 and 50 one position to the right in the list and insert 5 at first position in the sorted list.

Sorted | Unsorted

5	10	15	18	20	30	50	45

To move element 45 from unsorted to sorted portion, Compare 45 with 50 and 30. Since 45 is larger than 30, move 50 one position to the right in the list and insert 45 after 30 in the sorted list.

Sorted | Unsorted

5	10	15	18	20	30	45	50

Unsorted portion of the list has became empty. So we stop the process. And the final sorted list of elements is as follows...

5	10	15	18	20	30	45	50

ALGORITHM INSERTION-SORT (ARR, N)

Step 1: Repeat Steps 2 to 5 for K = 1 toN–1

Step 2: SET TEMP = ARR[K]

Step 3: SET J = K - 1

Step 4: Repeat while TEMP <= ARR[J]

SET ARR[J + 1] = ARR[J]

SETJ=J-1 [END OF INNER LOOP]

Step 5: SET ARR[J + 1] = TEMP [END OF LOOP]

Step 6: EXIT

To insert an element A[K] in a sorted list A[0], A[1], ..., A[K–1], we need to compare A[K] with A[K–1], then with A[K–2], A[K–3], and so on until we meet an element A[J] such that A[J] <= A[K]. In order to insert A[K] in its correct position, we need to move elements A[K–1], A[K–2], ..., A[J] by one position and then A[K] is inserted at the (J+1)th location..

Radix Sort

Radix sort is a linear sorting algorithm for integers and uses the concept of sorting names in alphabetical order. When we have a list of sorted names, the radix is 26 (or 26 buckets) because there are 26 letters in the English alphabet. So radix sort is also known as bucket sort. Observe that words are first sorted according to the first letter o/f the name. That is, 26 classes are used to arrange the names, where the first class stores the names that begin with A, the second class contains the names with B, and so on. During the second pass, names are grouped according to the second letter. After the second pass, names are sorted on the first two letters. This processis continued till the nth pass, where n is the length of the name with maximum number of letters. After every pass, all the names are collected in order of buckets. That is, first pick up the names in the first bucket that contains the names beginning with A. In the second pass, collect the name from the second bucket, and so on. When radix sort is used on integers, sorting is done on each of the digits in the number. The sorting procedure proceeds by sorting the least significant to the most significant digit. While sorting the numbers, we have ten buckets, each for one digit (0, 1, 2, …, 9) and the number of passes will depend on the length of the number having maximum number of digits.

Algorithm for RadixSort (ARR, N)

Step 1: Find the largest number in ARR as LARGE

Step 2: [INITIALIZE] SET NOP = Number of digits in LARGE

Step 3: SET PASS = 0

Step 4: Repeat Step 5 while PASS <= NOP-1

Step 5: SET= I=0 and INITIALIZE buckets

Step 6: Repeat Steps 7 to 9 while I<N-1

Step 7: SET DIGIT = digit at PASSth place in A[I]

Step 8: Add A[I] to the bucket numbered DIGIT

Step 9: INCREMENT bucket count for bucket numbered DIGIT [END OF LOOP]

Step 10: Collect the numbers in the bucket [END OF LOOP]

Step 11: END

Sort the numbers given below using radix sort. 345, 654, 924, 123, 567, 472, 555, 808, 911

In the first pass, the numbers are sorted according to the digit at ones place.

Bin 0	Bin 1	Bin2	Bin 3	Bin4	Bin 5	Bin 6	Bin7	Bin8	Bin 9
	911	472	123	654	345		67	808	
				924	555				

After this pass, the numbers are collected bucket by bucket. The new list thus formed is used as an input for the next pass. 911,472,123,654,924,345,555,67,808

In the second pass, the numbers are sorted according to the digit at the tens place

Bin 0	Bin 1	Bin2	Bin 3	Bin4	Bin 5	Bin 6	Bin7	Bin8	Bin 9
808	911	123		345	654	67	472		
		924			555				

After this pass, the numbers are collected bucket by bucket. The new list thus formed is used as an input for the next pass. 808,911,123,924,345,654,555,67,472

In the third pass, the numbers are sorted according to the digit at the hundreds place

Bin 0	Bin 1	Bin2	Bin 3	Bin4	Bin 5	Bin 6	Bin7	Bin8	Bin 9
67	123		345	472	555	654		808	911
									924

The sorted order is as above :67,123,345,472,555,654,808,911,924

<u>Address Calculation Sort</u>

- Is based on hashing.
- It is a distribution based sorting technique.
- A hash function is applied on each element of the unsorted list
- The address generated is taken as the position where the element is stored in a hash table organised as a array of pointers to elements.
- The elements that map to the same location are stored as a linked list of elements.

- If more than one element maps to the same location, they are inserted into the linked list in order using any sorting technique.
- After all elements have been hashed, the linked lists are concatenated to form the sorted list.
- The hash function should satisfy the property that if a key x is less than y,then f(x)<f(y).This is called order preserving property.

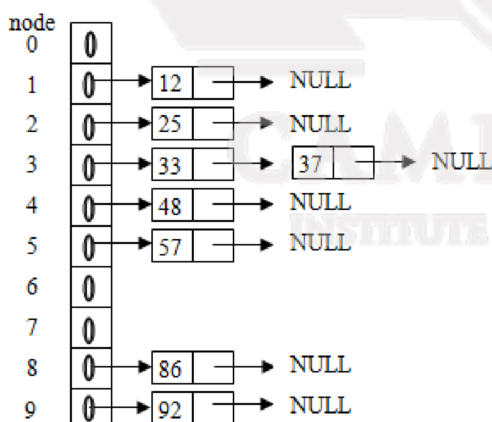
Example:

Sort 25 57 48 37 12 92 86 33 using address calculation sort

Let us create 10 sub lists. Initially each of these sublist 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 3rd sublist (4th) only, but must be checked with the existing elements for its proper position in this sublist.



Hashing

Why Hashing?

Internet has grown to millions of users generating terabytes of content every day. According to internet data tracking services, the amount of content on the internet doubles every six months. With this kind of growth, it is impossible to find anything in the internet, unless we develop new data structures and algorithms for storing and accessing data. So what is wrong with traditional data structures like Arrays and Linked Lists? Suppose we have a very large data set stored in an array. The amount of time required to look up an element in the array is

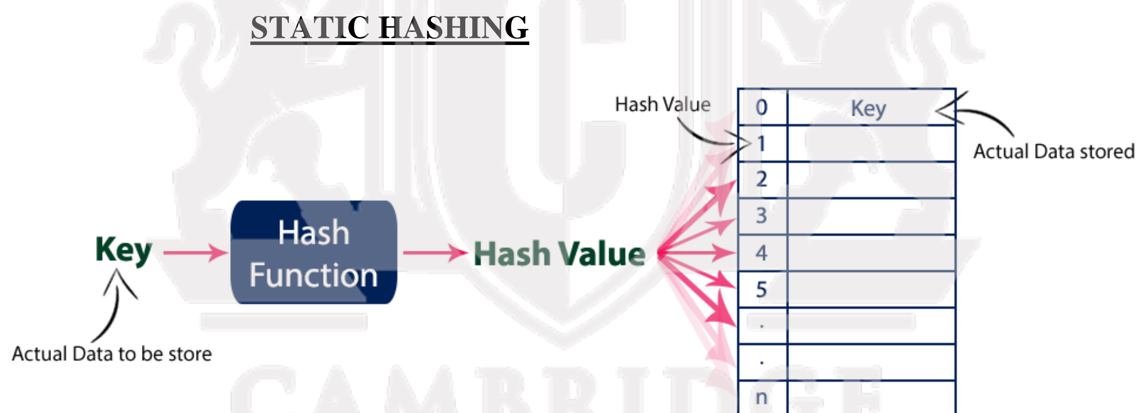
either O(log n) or O(n) based on whether the array is sorted or not. If the array is sorted then a technique such as binary search can be used to search the array. Otherwise, the array must be searched linearly. Either case may not be desirable if we need to process a very large data set. Therefore we discuss a new technique called hashing that allows us to update and retrieve any entry in constant time O(1). The constant time or O(1) performance means, the amount of time to perform the operation does not depend on data size n.

The Map Data Structure(Hash Map)(Hash function)

In a mathematical sense, a map is a relation between two sets. We can define Map M as a set of pairs, where each pair is of the form (key, value), where for given a key, we can find a value using some kind of a "function" that maps keys to values. The key for a given object can be calculated using a function called a **hash function**. In its simplest form, we can think of an array as a Map where key is the index and value is the value at that index. For example, given an array A, if i is the key, then we can find the value by simply looking up A[i]. The idea of a hash table is more generalized and can be described as follows.

The concept of a hash table is a generalized idea of an array where key does not have to be an integer. We can have a name as a key, or for that matter any object as the key. The trick is to find a hash function to compute an index so that an object can be stored at a specific location in a table such that it can easily be found.

STATIC HASHING



This kind of hashing is called **static hashing** since the size of the hash table is fixed.(an array)

Example:

Suppose we have a set of strings {"abc", "def", "ghi"} that we'd like to store in a table. Our objective here is to find or update them quickly from a table, actually in O(1). We are not concerned about ordering them or maintaining any order at all. Let us think of a simple schema to do this. Suppose we assign "a" = 1, "b"=2, … etc to all alphabetical characters. We can then simply compute a number for each of the strings by using the sum of the characters as follows.

"abc" = 1 + 2 + 3=6, "def" = 4 + 5 + 6=15 , "ghi" = 7 + 8 + 9=24

If we assume that we have a table of size 5 to store these strings, we can compute the location of the string by taking the sum mod 5. So we will then store

“abc” in 6 mod 5 = 1, “def” in 15 mod 5 = 0, and “ghi” in 24 mod 5 = 4 in locations 1, 0 and 4 as follows.

0	1	2	3	4
def	abc			ghi

Now the idea is that if we are given a string, we can immediately compute the location using a simple hash function, which is sum of the characters mod Table size. Using this hash value, we can search for the string.

Problem with Hashing -collision

The method discussed above seems too good to be true as we begin to think more about the hash function. First of all, the hash function we used, that is the sum of the letters, is a bad one. In case we have permutations of the same letters, “abc”, “bac” etc in the set, we will end up with the same value for the sum and hence the key. In this case, the strings would hash into the same location, creating what we call a “collision”. This is obviously not a good thing. Secondly, we need to find a good table size, preferably a prime number so that even if the sums are different, then collisions can be avoided, when we take mod of the sum to find the location. So we ask two questions.

Question 1: How do we pick a good hash function?

Question 2: How do we deal with collisions?

The problem of storing and retrieving data in O(1) time comes down to answering the above questions. Picking a “good” hash function is key to successfully implementing a hash table. What we mean by “good” is that the **function must be easy to compute and avoid collisions as much as possible.** If the function is hard to compute, then we lose the advantage gained for lookups in O(1). Even if we pick a very good hash function, we still will have to deal with “some” collisions.

The process where two records can hash into the same location is called collision. We can deal with collisions using many strategies, such as linear probing (looking for the next available location i+1, i+2, etc. from the hashed value i), quadratic probing (same as linear probing, except we look for available positions i+1 , i + 4, i + 9, etc from the hashed value i and separate chaining, the process of creating a linked list of values if they hashed into the same location.This is called collision resolution.

Popular hash functions

Hash functions that use numeric keys are very popular.. However, there can be cases in real-world applications where we can have alphanumeric keys rather than simple numeric keys. In such cases, the ASCII value of the character can be used to transform it into its equivalent numeric key. Once this transformation is done, any hash function can be applied to generate the hash value.

Division Method

It is the most simple method of hashing an integer x. This method divides x by M and then uses the remainder obtained. In this case, the hash function can be given as

h(x) = x mod M

The division method is quite good for just about any value of M and since it requires only a single division operation, the method works very fast. However, extra care should be taken to select a suitable value for M. Generally, it is best to choose M to be a prime number because making M a prime number increases the likelihood that the keys are mapped with a uniformity in the output range of values.

A potential drawback of the division method is that while using this method, consecutive keys map to consecutive hash values. On one hand, this is good as it ensures that consecutive keys do not collide, but on the other, it also means that consecutive array locations will be occupied. This may lead to degradation in performance.

Example :

Calculate the hash values of keys 1234 and 5462. Solution Setting M = 97, hash values can be calculated as:

h(1234) = 1234 % 97 = 70

h(5642) = 5642 % 97 = 16

Mid-Square Method

The mid-square method is a good hash function which works in two steps:

Step 1: Square the value of the key. That is, find k^2.

Step 2: Extract the middle r digits of the result obtained in Step 1.

The algorithm works well because most or all digits of the key value contribute to the result. This is because all the digits in the original key value contribute to produce the middle digits of the squared value. Therefore, the result is not dominated by the distribution of the bottom digit or the top digit of the original key value. In the mid-square method, the same r digits must be chosen from all the keys. Therefore, the hash function can be given as:

h(k) = s where s is obtained by selecting r digits from k^2.

Example Calculate the hash value for keys 1234 and 5642 using the mid-square method. The hash table has 100 memory locations. Solution Note that the hash table has 100 memory locations whose indices vary from 0 to 99.

This means that only two digits are needed to map the key to a location in the hash table, so r = 2.

When k = 1234, k^2 = 1522756, h (1234) = 27

When k = 5642, k^2 = 31832164, h (5642) = 21

Observe that the 3rd and 4th digits starting from the right are chosen.

Folding Method

The folding method works in the following two steps:

Step 1: Divide the key value into a number of parts. That is, divide k into parts k1, k2, ..., kn, where each part has the same number of digits except the last part which may have lesser digits than the other parts.

Step 2: Add the individual parts. That is, obtain the sum of k1 + k2 + ... + kn. The hash value is produced by ignoring the last carry, if any. Note that the number of digits in each part of the key will vary depending upon the size of the hash table. .

Example Given a hash table of 100 locations, calculate the hash value using folding method for keys 5678, 321, and 34567. Solution Since there are 100 memory locations to address, we will break the key into parts where each part (except the last) will contain two digits. The hash values can be obtained as shown below:

key	5678	321	34567
Parts	56 and 78	32 and 1	34, 56 and 7
Sum	134	33	97
Hash value	34 (ignore the last carry)	33	97

Collision Resolution Strategies

1. Open Addressing/Closed Hashing
2. Chaining

Once a collision takes place, open addressing or closed hashing computes new positions using a probe sequence and the next record is stored in that position

The process of examining memory locations in the hash table is called probing. Open addressing technique can be implemented using linear probing, quadratic probing, double hashing.

Linear Probing

When using a linear probe to resolve collision, the item will be stored in the next available slot in the table, assuming that the table is not already full.

This is implemented via a linear search for an empty slot, from the point of collision. If the physical end of table is reached during the linear search, the search will wrap around to the beginning of the table and continue from there.If an empty slot is not found before reaching the point of collision, the table is full.

If h is the point of collision, probe through h+1,h+2,h+3..................h+i. till an empty slot is found

Index	Key
[0]	72
[1]	
[2]	18
[3]	43
[4]	36
[5]	
[6]	6
[7]	

Add the keys 10, 5, and 15 to the previous table .

Hash key = key % table size

2 = 10 % 8

5 = 5 % 8

7 = 15 % 8

Index	Key
[0]	72
[1]	15
[2]	18
[3]	43
[4]	36
[5]	10
[6]	6
[7]	5

Searching a Value using Linear Probing

The procedure for searching a value in a hash table is same as for storing a value in a hash table. While searching for a value in a hash table, the array index is re-computed and the key of the element stored at that location is compared with the value that has to be searched. If a match is found, then the search operation is successful. The search time in this case is given as O(1). If the key does not match, then the search function begins a sequential search of the array that continues until:

- the value is found,or
- the search function encounters a vacant location in the array, indicating that the value is not present, or
- the search function terminates because it reaches the end of the table and the value is not present.

Probe Sequence ::(h+i)%Table size

Disadvantage:

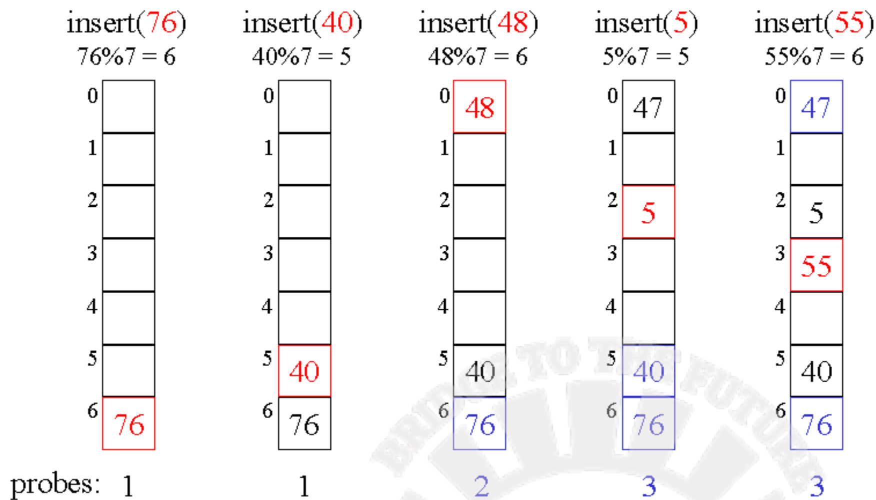
As the hash table fills, clusters of consecutive cells are formed and the time required for a search increases with the size of the cluster. It is possible for blocks of data to form when collisions are resolved. This means that any key that hashes into the cluster will require several attempts to resolve the collision. More the number of collisions, higher the probes that are required to find a free location and lesser is the performance. This phenomenon is called clustering. To avoid clustering, other techniques such as quadratic probing and double hashing are used.

Quadratic Probing

A variation of the linear probing idea is called **quadratic probing**. Instead of using a constant “skip” value, if the first hash value that has resulted in collision is h, the successive values which are probed are $h+1$, $h+4$, $h+9$, $h+16$, and so on. In other words, quadratic probing uses a skip consisting of successive perfect squares.

BRIDGE TO THE FUTURE CAMBRIDGE INSTITUTE OF TECHNOLOGY

Quadratic Probing Example ☺



Probe sequence :h,h+1^2,h=2^2,h=3^2..............................h+i^2

H(k)=(h+i$^{2)}$%Tablesize

<u>Double Hashing</u>

In double hashing, we use two hash functions rather than a single function. Double hashing uses the idea of applying a second hash function to the key when a collision occurs. The result of the second hash function will be the number of positions form the point of collision to insert.There are a couple of requirements for the second function:

- it must never evaluate to 0
- must make sure that all cells can be probed

A popular second hash function is: $Hash_2(key)$ = R - (key % R) where R is a prime number that is smaller than the size of the table.But any independent hash function may also be used.

Table Size = 10 elements
$Hash_1$(key) = key % 10
$Hash_2$(key) = 7 – (k % 7)

Insert keys : 89, 18, 49, 58, 69

Hash(89) = 89 % 10 = 9

Hash(18) = 18 % 10 = 8

Hash(49) = 49 % 10 = 9 a collision !
= 7 – (49 % 7)
= 7 positions from [9]

Hash(58) = 58 % 10 = 8
= 7 – (58 % 7)
= 5 positions from [8]

Hash(69) = 69 % 10 = 9
= 7 – (69 % 7)
= 1 position from [9]

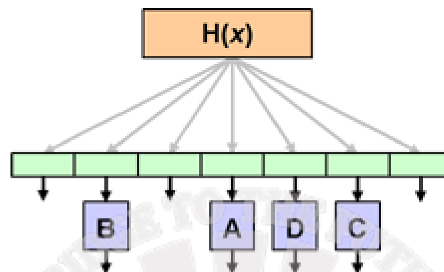
Index	Value
[0]	49
[1]	
[2]	
[3]	69
[4]	
[5]	
[6]	
[7]	58
[8]	18
[9]	89

Double hashing minimizes repeated collisions and the effects of clustering.

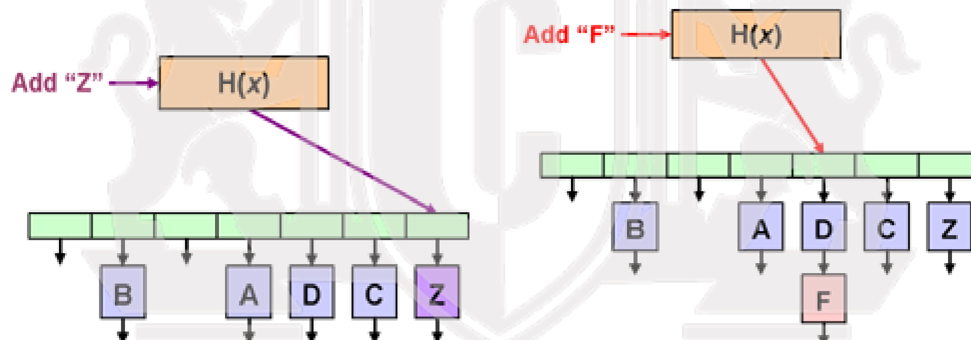
CAMBRIDGE INSTITUTE OF TECHNOLOGY

Chaining

Chaining is another approach to implementing a hash table; instead of storing the data directly inside the structure, have a linked list structure at each hash element. That way, all the collision, retrieval and deletion functions can be handled by the list, and the hash function's role is limited mainly to that of a guide to the algorithms, as to which hash element's list to operate on.



The linked list at each hash element is often called a chain. A *chaining* hash table gets its name because of the linked list at each element -- each list looks like a 'chain' of data strung together.Operations on the data structure are made far simpler, as all of the data storage issues are handled by the list at the hash element, and not the hash table structure itself.



Chaining overcomes collision but increases search time when the length of the overflow chain increases

Drawbacks of static hashing

1. Table size is fixed and hence cannot accommodate data growth.

2. Collsions increases as data size grows.

Avoid the above conditions by doubling the hash table size. This increase in hash table size is taken up, when the number of collisions increase beyond a certain threshold. The threshold limit is decided by the load factor.

Load factor

The load factor α of a hash table with n key elements is given by **α= n / hash table size**

The probability of a collision increases as the load factor increases. We cannot just double the size of the table and copy the elements from the original table to the new table ,since when

the table size is doubled from N to 2N+1, the hash function changes. It requires reinserting each element of the old table into the new table (using the modified hash function).This is called Rehashing. Rehashing in large databases is a tedious process and hence dynamic hashing.

Dynamic hashing schemes

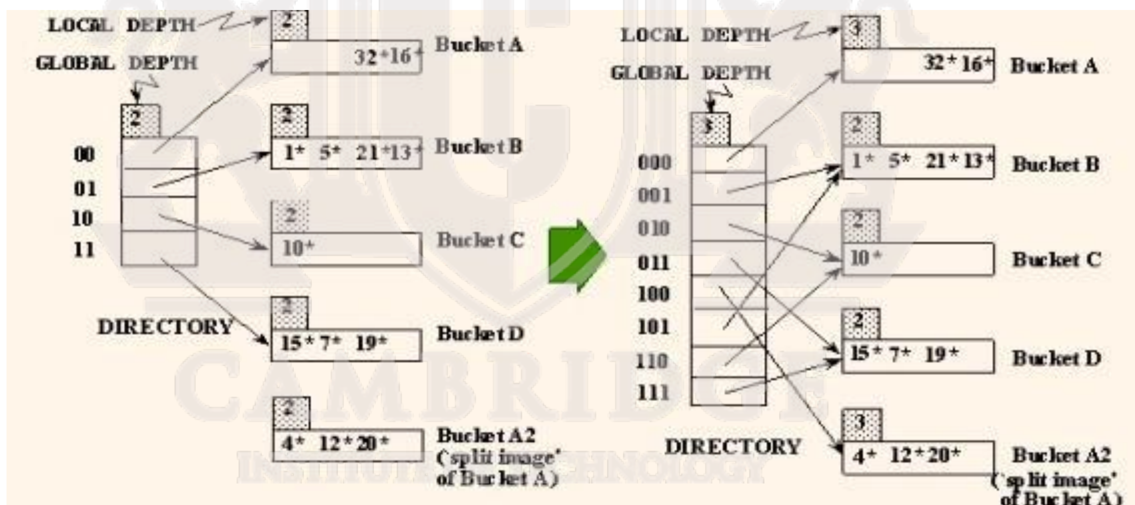
Dynamically increases the size of the hash table as collision occurs.There are two types:

Extendible hashing (directory): uses a directory that grows or shrinks depending on the data distribution. No overflow buckets

Linear hashing(directory less): No directory. Splits buckets in linear order, uses overflow buckets.

Extendible hashing :

- Uses a directory of pointers to buckets/bins which are collections of records
- The number of buckets are doubled by doubling the directory, and splitting just the bin that overflowed.
- Directory much smaller than file, so doubling it is much cheaper. Only one bin of data entries is split and rehashed.



Global Depth

– Max number of bits needed to tell which bucket an entry belongs to.

Local Depth

- The number of least significant digits that is common for all the numbers sharing the same bin.

On overflow:

If global depth =Local depth

1. Double the hash directory
2. SPlit the overflowing bin
3. Redistribute elements of the overflowing bin

4. Increment the global and local depth

If global depth >Local depth

1. SPlit the overflowing bin
2. Redistribute elements of the overflowing bin
3. Increment the local depth

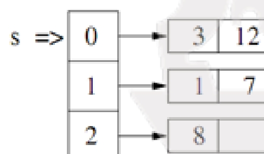
Linear Hashing

Basic Idea:

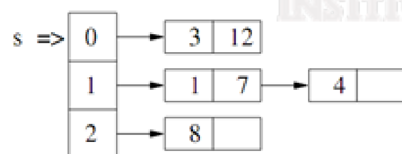
- Pages are split when overflows occur – but not necessarily the page with the overflow.
- Directory avoided in LH by using overflow pages. (chaining approach)
- Splitting occurs in turn, in a round robin fashion.one by one from the first bucket to the last bucket.
- Use a family of hash functions $\mathbf{h}_0$, $\mathbf{h}_1$, $\mathbf{h}_2$, ...
  - Each function's range is twice that of its predecessor.
- When all the pages at one level (the current hash function) have been split, a new level is applied.

- Insert in Order using linear hashing: 1,7,3,8,12,4,11,2,10,13.....

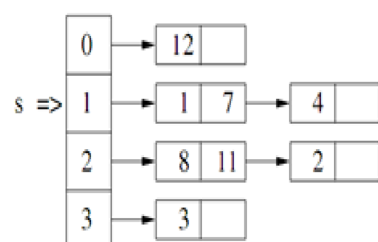
After insertion till 12:



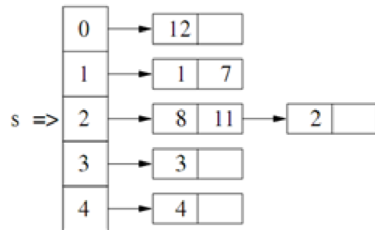
When 4 inserted overflow occurred. So we split the bucket (no matter it is full or partially empty). And increment pointer.



So we split bucket 0 and rehashed all keys in it. Placed 3 to new bucket as (3 mod 6 = 3) and (12 mod 6 = 0). Then 11 and 2 are inserted. And now overflow. *s* is pointing to bucket 1, hence split bucket 1 by re- hashing it.



After split:



Insertion of 10 and 13: as (10 mod 3 = 1) and bucket 1 < s, we need to hash 10 again using h1(10) = 10 mod 6 = 4th bucket.

Note :Files topics can be referred from the text book reference(reema thareja)

Prepared by

Geetha.P,Asst Professor

Pankaja K, Asso Professor

Dept of CSE,CiTech

HARD WORK BEATS TALENT WHEN TALENT DOES NOT WORK HARD

GOOD LUCK!!! ☺

BRIDGE TO THE FUTURE
CAMBRIDGE
INSTITUTE OF TECHNOLOGY