Unit I: Structures and Unions(co1)

Structure is a collection of data items of different data types under a common name.

Structure is a group of items in which every item is identified by its own name. These items are the members of that structure.

Syntax for the declaration of a structure

```
struct structure_name
{

        data_type   member1;
        date_type   member2;
                                .
                                .
        data_typememberN;
} ;
```

Example:

```
struct  abc
      {
        char first [10];
              char last[20];
}Sname, Ename;   (Structure variables)
```

is equivalent to

```
struct  abc
      {
        char first [10];
              char last[20];
      };
struct abcSname, Ename;
```

This declaration creates two structure variables. Sname and Ename each of which contains two members: first,last.

Using Typedef to declare structure variables

Using the keyword ***typedef*** we can define any type to the easiest name that we want to use.

Eg. : typedef struct abc node; // this defines ***struct abc*** as ***node***

At any place if we write node that is equivalent to struct node.

Another alternative usage of ***typedef*** is in the definition of the data type directly.

Eg. :

```
typedef struct student
        {
            char first[10];
            char last[20];
        } stu_type;
```

We can now use the typedef as in the following declaration:

stu_typeSname, Ename;

A structure member can be accessed by using the following format.

struct_variable_name.member_name

Eg :printf ("%s",sname.first); will display **first** (member) from **Sname**(structure variable).

Structures within a Structure

A structure can have the variable of another structure as a member. Such members can be more than one also.

eg: The following are the two structure definitions namely **adr, student** which are used in another structure **newaddr.**

```
struct adr
{
    char addr[40];
    char city[10];
    char state[3];
    char zip[6];
};

struct student
{
    char first[10];
    char last[20];
};
```

Now we can declare new structure **newaddr**

```
struct newaddr
  {
        struct student name;
```

```
        struct adr address;
};
```

Initialization of structure variables

The following example illustrates the initialization of structures.

```
struct book
{       char   title[20];
            char   Author[15];
            int    pages;
            float  price;
       };

struct book book1={“abcd”,”xyz”,100,25.50};
struct book  book2={“aaa”,”xxx”};  //   book2.pages, book2.price  will be initialized to zero.
struct book book3=book1;     // will copy the corresponding member of book1 to book3
```

Note : In above structure definition **book1.price** (etc.) can be treated like any other ordinary variable.

Assigning values to structure members

The following example illustrates the assignment of values in structures.

```
struct book
{       char   title[20];
            char   Author[15];
            int    pages;
            float  price;
}book1;

strcpy(book1.title, "CDS");
strcpy(book1.author,"K. Nagi Reddy");
book1.pages = 861;
book1.price= 215.00;
```

Reading values into the structure

Using scanf we can read the values into the structure.

```
scanf ("%s%s%d%f",book1.title,book1.author,&book1.pages,&book1.price);
```

Note: The structure variables cannot be used as the normal variables in all operations such as in relational statements, logical statements, etc.

Array Of Structures

The structure variables can be defined as an array if they required more in number.

Eg.
```
struct marks
{
    int sub1;
    int sub2;
    int sub3;
};
```

static struct marks student[3] = {(50,60,70),(40,50,60),(10,20,30)};
Here student is array of 3 elements 0,1&2 initializes members as

student[0].sub1 = 50;	student[1].sub1 = 40;	student[2].sub1 = 10;
student[0].sub2 = 60;	student[1].sub2 = 50;	student[2].sub2 = 20;
student[0].sub3 = 70;	student[1].sub3 = 60;	student[2].sub3 = 30;

Passing Structure Elements To Functions

Passing individual structure elements:The Individual elements of structures can be passed to functions as the normal variables by using **struct_var_name.member_name**

Example:
```
void main()
{
    void display(char[],char[],int);   // Function Prototype
    struct book
    {
        char name[25], author[25];
    int  pages;
  };
     static struct book b1={"DSTC","G. SORENSON",861};
     display(b1.name,b1.author,b1.pages);   // Function Call
}

void display(char *s, char *t, int n)    // Function definition
{
  printf("%s\n%s\n%d", s, t, n);
}
```

Passing Entire Structure To Functions:

```
struct book
{
  char name[25], author[25];
  int pages;
};

void main()
{
```

```
    void display(struct book);
    static struct book b1={"DSTC","G. SORENSON",861};
    display(b1);
}

void display(struct book b)
{
printf("s\n%s\n%d, b.name, b.author, b.pages);
}
```

Pointers to Structures

As for normal variables pointers can be declared to structures.
If **ptr** is a pointer to a structure, then the members of the structure can be accessed using **ptr->member_name** as in the following example.

Example:

```
void main()
{
    struct book
    {
        char name[25],author[25];
            int  pages;
      };
      struct book b1={"DSTC","TENEN",672};
      struct book *ptr;   /* (remember now ptr is not a variable, it is pointer) */
      ptr=&b1;
      printf("%s%s%d\n",b1.name,b1.author,b1.pages);
      printf("%s%s%d\n",ptr->name,ptr->author,ptr->pages);
}
```

An illustrated example on Structures

What data type are allowed to structure members? **Anything goes**: basic types, arrays, strings, pointers, even other structures. You can even make an **array of structures**.

Consider the program on the next few pages which uses an array of structures to make a deck of cards and deal out a poker hand.

```
#include <stdio.h>

struct playing_card
{
int pips;
char *suit;
} deck[52];

void make_deck(void);
void show_card(int n);
```

```
void main()
{
	make_deck();
	show_card(5);
	show_card(37);
	show_card(26);
	show_card(51);
	show_card(19);
}

void make_deck(void)
{
	int k;
	for(k=0; k<52; ++k)
	{
		if (k>=0 && k<13)
		{
			deck[k].suit="Hearts";
			deck[k].pips=k%13+2;
		}
		if (k>=13 && k<26)
		{
			deck[k].suit="Diamonds";
			deck[k].pips=k%13+2;
		}
		if (k>=26 && k<39)
		{
			deck[k].suit="Spades";
			deck[k].pips=k%13+2;
		}
		if (k>=39 && k<52)
		{
			deck[k].suit="Clubs";
			deck[k].pips=k%13+2;
		}
	}
}


void show_card(int n)
{
	switch(deck[n].pips)
	{
		case 11:
			printf("%c of %s\n",'J',deck[n].suit);
			break;
		case 12:
			printf("%c of %s\n",'Q',deck[n].suit);
			break;
		case 13:
```

```
                printf("%c of %s\n",'K',deck[n].suit);
                break;
            case 14:
                printf("%c of %s\n",'A',deck[n].suit);
                break;
            default:
                printf("%c of %s\n",deck[n].pips,deck[n].suit);
                break;
        }
}
```

Output

7 of Hearts
K of Spades
2 of Spades
A of Clubs
8 of Diamonds

Unions

Unions are C variables whose syntax looks similar to structures, but act in a completely different manner. A union is **a variable that can take on different data types** in different situations. The union syntax is:

union *tag_name*{
***type1 member1*;**
***type2 member2*;**
…
};

For example, the following code declares a union data type called **intfloat**and a union variable called **proteus**:

```
union intfloat {
float f;
int i;
};
union intfloat proteus;
```

Unions and Memory:

Once a union variable has been declared, the amount of memory reserved is just enough to be able to represent the **largest member**. (Unlike a structure where memory is reserved for **all** members).

In the previous example, 4 bytes are set aside for the variable **proteus** since a **float** will take up 4 bytes and an **int** only 2 (on some machines).

Data actually stored in a union's memory can be the data associated with **any** of its members. But **only one** member of a union can contain valid data at a given point in the

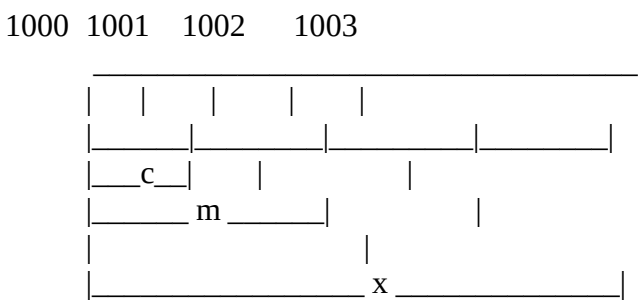
program. It is the user's responsibility to keep track of which type of data has most recently been stored in the union variable.

In the example:

```
union item
{
    int   m;
    float x;
    char  c;
}code;
```

**** The size of a union data type will be the size of the bigger member variable.**

In this example: **Memory reserved for variable item is: 4 bytes**



Unions Example

The following code illustrates the chameleon-like nature of the union variable **proteus** defined earlier.

```
#include <stdio.h>
void main() {
union intfloat {
float f;
int i;
} proteus;
proteus.i=4444 /* Statement 1 */
printf("i:%12d f:%16.10e\n",proteus.i,proteus.f);
proteus.f=4444.0; /* Statement 2 */
printf("i:%12d f:%16.10e\n",proteus.i,proteus.f);
}
```

Output

i: 4444 f:6.2273703755e-42
i: 1166792216 f:4.440000000e+03

• After **Statement 1**, data stored in **proteus** is an integer the the float member is full of junk.
• After **Statement 2**, the data stored in **proteus** is a float, and the integer value is meaningless.

Comparison between structures and unions

Structures	Unions
This is the combination of different types of variables	This is also the combination of different types of variables
The Keyword used is ‘struct’	The keyword used is ‘union’
The size of a structure is the sum of the sizes of its individual elements	The size of the union is the size of its biggest element.
At any point of time all the members are valid	At any point of time only one member is valid

UnitII: FILES(co2)

The concept of files is to store data on the disk. A file is a place on the disk where group of related data is stored. C language has number of functions to perform file operations like,

1. Creating a file
2. Opening a file
3. Reading data from a file
4. Writing data into a file
5. Closing a file

There are two methods to perform file operations in C Language.

1. Low-level I/O (Uses UNIX system calls)
2. High-level I/O (Uses C language's standard I/O library)

DEFINING AND OPENING A FILE

If want to store data in a file on the disk, we must use a pointer variable of data structure of a file which is defined as **FILE** in the standard I/O library.

General form and opening a file is as follows:

```
FILE *fp;
fp=fopen("filename","mode");
```

fopen() is a library function which opens a file given in *filename* argument in the given *mode*.

filename in function fopen() is a string of characters that make up a valid file for the operating system. It contains two parts, a primary name and an extension name. The extension name is optional.

Example: Abc.dat
Output
First.c
Transaction.txt

mode specifies the mode of opening and this can be

r open the file for reading only
w open the file for writing only
a open the file for appending

Recent compilers include additional modes of opening. They are

r+ opens in read mode, both reading and writing can be done
w+ opens in write mode, both reading and writing can be done
a+ opens in append mode, both reading and writing can be done

CLOSING A FILE

After all the operations are completed, a file must be closed. This can be as follows

```
flose(fp);
```

where fp is a file pointer.

INPUT/OUTPUT OPERATIONS ON FILES

Once a file is open, reading of or writing to it is accomplished using standard I/O routines. Standard I/O functions are listed bellow:

fopen()	Creates a new file or opens an existing file.
fclose()	Closes a file which has been opened.
getc()	Reads a character from a file.
putc()	Writes a character to a file.
fprintf()	Writes set of data values to a file.
fscanf()	Reads a set of data values from a file.
getw()	Reads an integer from a file.
putw()	Writes an integer to a file.
fseek()	Sets the position to a desired point in the file.
ftell()	Returns the current position in the file in terms of bytes from the starting position.
rewind()	Sets the position to the beginning of the file.

The getc and putcfunctions :

The simplest file I/O functions are getc() and puts().

putc() function writes a character to a specified file.

Syntax: putc(c,fp);

where **c** is a character variable and **fp** is a FILE pointer associated with a specific file on the disk.

Character contained in variable **c** will be written in to **fp**.

getc() reads a character from a specified file.

Syntax: c=getc(fp);

where **c** is a character variable and **fp** is a FILE pointer associated with a specific file on the disk.

Getc function returns a character from the file specified in **fp** to **c.**

The file pointer moves by one character position for every operations of getc() or putc(). The getc() will return an end-of-file marker EOF, when end of the file has been reached.

The fprintf() and fscanf() functions

The functions fprintf() and fscanf() perform I/O operations that are identical to printf() and scanf() functions, except of course that they work on files.

Syntax: fprintf(fp,"control string",list);

where *fp* is FILE pointer associated with a file that has been opened for writing. The *control string* contains output specifications for the items in the *list*.

Example: fprintf(fp,"%s %d %f",name,age,avg);

Syntax of fscanf()

fscanf(fp,"control string",list);

where *fp* is FILE pointer associated with a file that has been opened for reading. The *control string* contains output specifications for the items in the *list*.

Example: fscanf(fp,"%s %d %f",name,age,avg);

ERROR HANDLING DURING I/O OPERATIONS

It is possible that an error may occur during I/O operations on a file. Typical error situations include:

1. Trying to read beyond the end-of-file mark.
2. Device overflow.
3. Trying to use a file that has not been opened.
4. Trying to perform an operation on a file, when the file is opened for another type of operations.
5. Opening a file with an invalid file name.
6. Attempting to write to a write-protected file.

Examples:

/*The following program reads a text file and counts how many times each letter from 'A' to 'Z' occurs and displays the results.*/

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

int count[26];

void main(int argc, char *argv[])
{
  FILE *fp;
  char ch;
  int i;
  /* see if file name is specified */
  if (argc!=2) {
printf("File name missing");
exit(1);
  }

  if ((fp= fopen(agv[1], "r")) == NULL) {
printf("cannot open file");
exit(1);
  }

  while ((ch=fgetchar(fp)) !=EOF) {
ch = toupper(ch);
    if (ch>='A' &&ch<='Z') count[ch-'A']++;
  }

  for (i=0; i<26; i++)
printf("%c occurred %d times\n", i+'A', count[i]);

fclose(fp);
}
```

/*This program uses command line arguments to read and display the contents of a file supplied as an argument. */

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

```
#define CLEARS 12   /* constant */

main(int argc , char *argv[])
{
    FILE *fp , *fopen();
    int c;

putchar(CLEARS);

    while ( --argc> 0 )
       if ((fp=fopen(argv[1], "r"))==NULL)
         {
printf("I can't open %s\n", argv[1]);
           break;
         }
       else
         {
           while ((c= getc(fp)) !=EOF)
putc(c,stdout);  /* display to the screen */
fclose(fp);
         }
}
```

DATA STRUCTURES Unit-III (co4)

Data structures are to organize the related data to enable us to work efficiently with data, exploring the relationships within the data. For example: stacks, lists and queues are data structures.

Data structure is a study of different methods of organizing the data and possible operations on these structures.

Two such fundamental data structures are arrays *and* structures. *But these generally will be used in association with static memory allocation(will be discussed later). These are not convenient to use with Dynamic memory allocation.*

***MEMORY ALLOCATION**: This can be done either statically or dynamically.*

***STATIC MEMORY ALLOCATION**: If the memory is allocated at compile time , then it is called* Static Memory Allocation*. The problem with this method is, we should know the amount of memory required before the compile time. We can't allocate more memory in run time.*

***DYNAMIC MEMORY ALLOCATION**: If the memory is allocated at run time , then it is called* Dynamic Memory Allocation*. It is enough to know the amount of memory at the instance when it is required.*

STATIC MEMORY ALLOCATION	***DYNAMIC MEMORY ALLOCATION***
1. Memory will be allocated at compile time.	*Memory will be allocated at run time.*
2. We should know the amount of memory required before compile time	*It is enough to know the amount of memory when it is really required*
3. There will be the wastage of memory when more is allocated and used less.	*There will be no such wastage since the memory will be allocated only when it is needed.*

TYPES OF DATA STRUCTURES

Basically there are two types of data structures. They are:

1. LINEAR DATA STRUCTURES

a. Stacks
b. Queues
c. Linked Lists

2. NON LINEAR DATA STRUCTURES

a. Trees
b. Graphs

L I N K E D L I S T S

Linked list is a datastructure in which several nodes are linked together. Each node consists of an information field and an address field to store the address of next node. Generally there will be a pointer of same type of node which will consists the address of the first node and it is called as HEADER.

Linked lists are mainly of 3 types.

1. Singly Linked List (If nothing is mentioned, it is of this type)
2. Doubly Linked List (DLL)
3. Circular Linked List (CLL)

MEMORY ALLOCATION FUNCTIONS

malloc: Allocates required memory size in bytes and returns the address of the first byte of the allocated space.

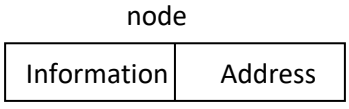
***calloc*:** Allocates contiguous space, initializes the elements to zero and then returns a pointer to the memory.

***free*:** Frees previously allocated space for the pointer passed as argument.

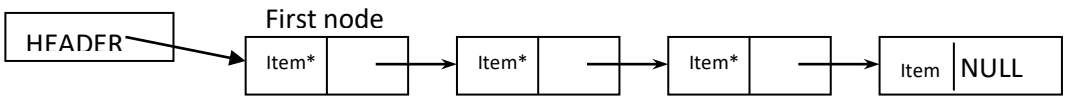
***realloc*:** Modifies the size of previously allocated space.

SINGLY LINKED LISTS (SLL)

Generally a node in SLL will look like



Generally a SLL node will look like



- Using dynamic memory allocation Nodes can be created or deleted at any point of time.
- Nodes can be inserted or deleted at any position of the list.
- HEADER will contain address of the first node.
- Address field of the last node contains NULL.

Declaration of a Node

A Node can be declared as a structure having two parts.

- Information Part (Can be an integer, or a combination of some variables of any type)
- Address Part (A pointer to the same type of structure)

Declaration:

```
struct  node
   {
      int item;
      struct node * link;
   };
```

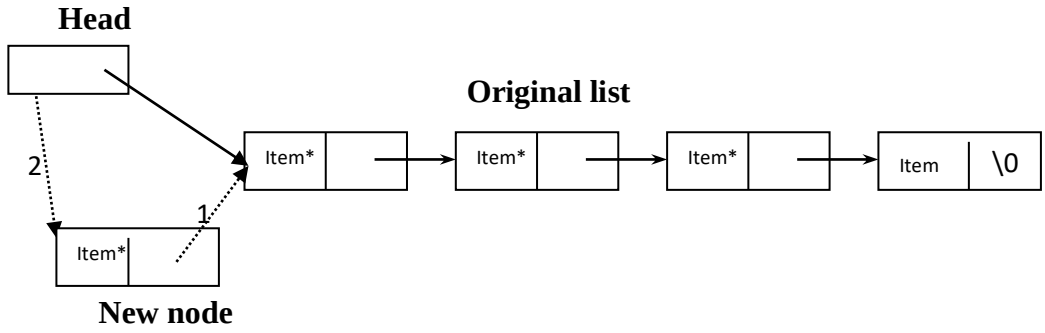
Operations on linked lists

There are several operations that we can perform on linked lists.
They are:

1. Initialization.
2. Insertion.
3. Deletion
4. Traversal or Display.
5. Find
6. Is empty. (Is full operation will not exist).

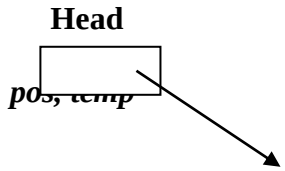
INSERTION: This can be done at the beginning or after position ***pos*** or at the end.

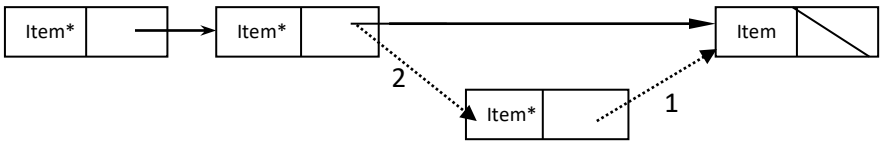
Insertion at begin:



1. Place the address of the first node of the original list(**Head**) in the **new node** link field.
2. Update the **Head** with **New node** address, by making it the starting node.

Insertion after position pos (This method is applicable for the 'Insertion at last' also):





new

- If you want to insert after node ***pos***, then traverse to ***pos*** node
- Let ***temp*** pointing to node ***pos***, ***new*** is the new node.

If ***pos*** = 0
Insert at begin
Else if (***pos***<=no of nodes)
new->link=***temp***->link(operation 1 shown in fig)
temp->link=***new***(operation 2 shown in fig)
Else
Not possible to insert after ***pos***

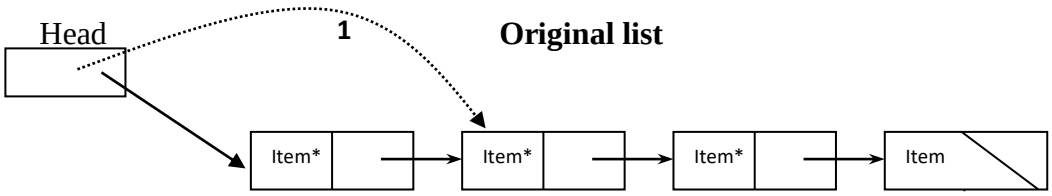
Insertion at last:

To insert a node at the end of the linked list, position ***temp*** at the last node and do the following operations.

new->link=**NULL**
temp->link=***new***

DELETION: This can be done at first, or last, or from position ***pos***.

Deletion at Begin:

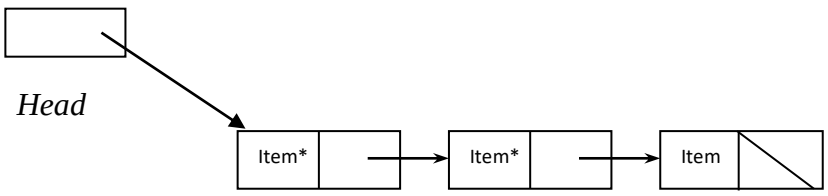


Node_tobe_deleted

To delete the first node, make the **Head** points to the second node by using
Head = Head->link (As indicated by **1** in the figure above)
If there is only one node then head will contain NULL after deletion.

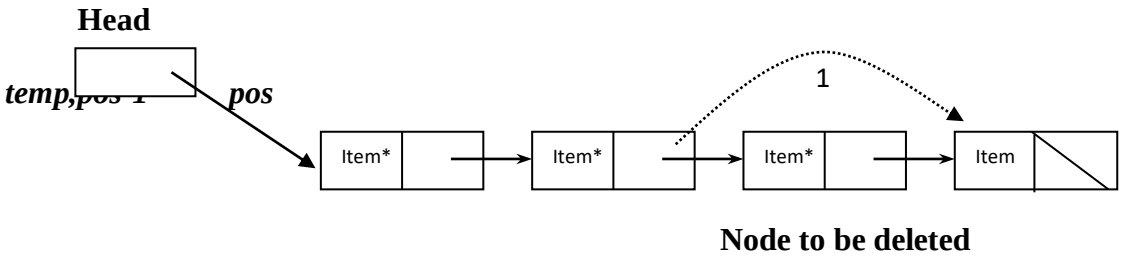
Note: Here, free is not implemented.

After deletion



Delete from a position *pos*:

This process is also applicable to delete the last node.



Process:

If pos=1
Delete at begin
Else if ***pos***< no of nodes
1.Traverse to the node ***pos***-1 and make **temp** pointing to position ***pos***-1.
2.temp->link = temp->link->link (As shown in operation 1 in fig.)
Else
Not possible – that node does not exist

Deletion at the end :

1.Traverse to the last but one node and point ***temp*** to that node.
2.Now make ***temp***->link = NULL.

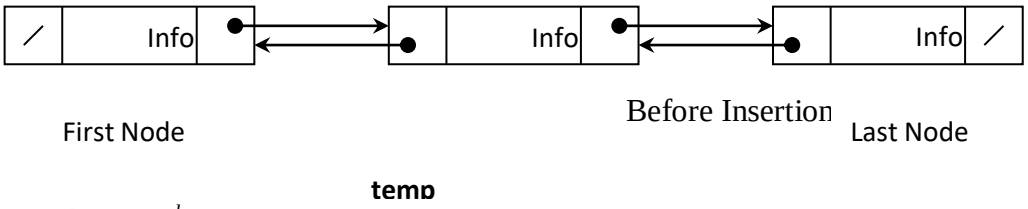
DOUBLY LINKED LISTS: In this every node will consist two address fields, one pointing to the next node and the other pointing to the previous. This allows us to traverse in both the directions.

Node Declaration:

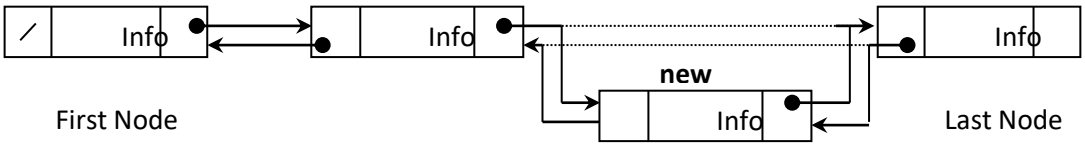
```
struct dnode
        {
                int info;
                struct dnode *pre, *next;
        };
```

If there is only one node, it's **pre** and **next** will be NULL.
If there is more than one node, then first node **pre** will be NULL and the last node **next** will be NULL.

INSERTION



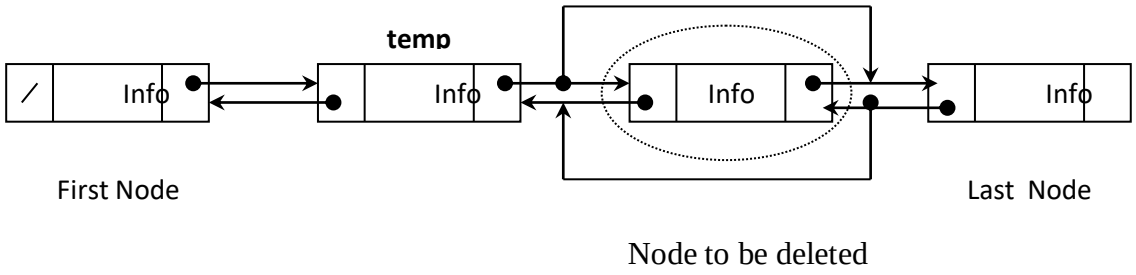
To Insert Nodee at 3rd Position:



Four changes will be made while inserting a node in the middle of the list.

```
new->next=temp->next
new->pre=temp
temp->next->pre=new
temp->next=new
```

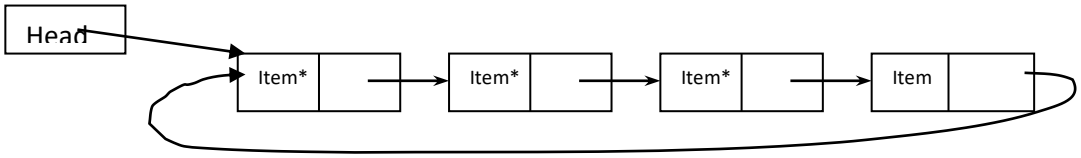
DELETION



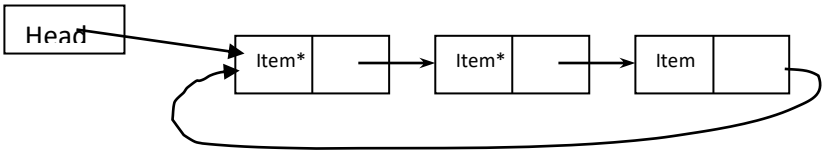
while deleting a middle node the following changes will be made
temp->next->pre=temp
temp->next=temp->next->next

CIRCULAR LISTS

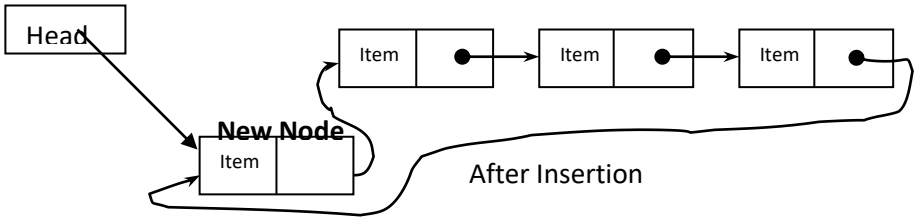
The circular list has no beginning and no end. The last node points back to the first node.



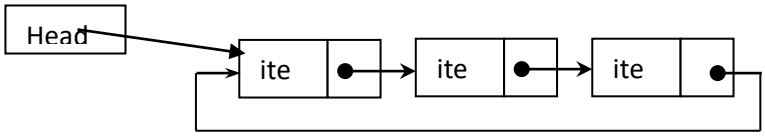
Insertion and deletion : While inserting at first position take care that you update your last node's address. After insertion, the list should be in the following manner



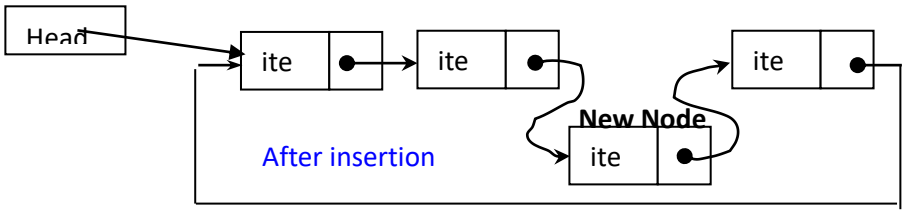
Before insertion



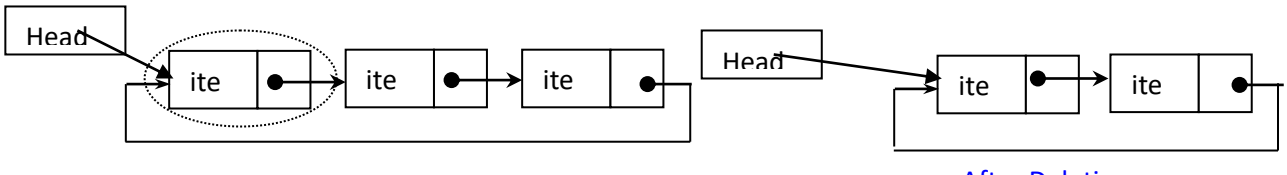
To insert in middle position, the same method used in singly linked lists is followed.



Before insertion



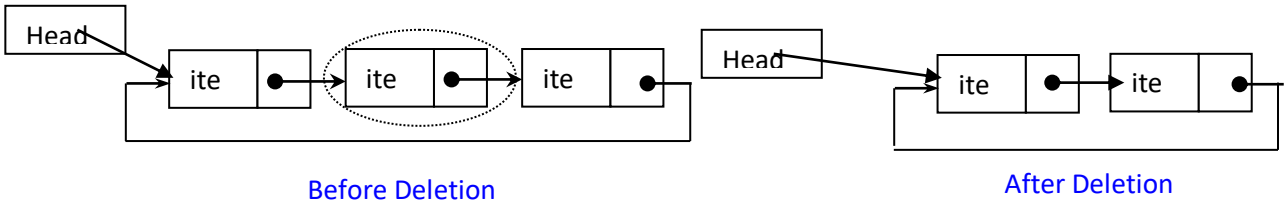
Delete front : Take care in updating the last node link field.



Before Deletion

After Deletion

Delete middle : No problem, same as in your singly linked lists



Before Deletion

After Deletion

DIFFERENCE BETWEEN STATIC STRUCTURES AND DYNAMIC STRUCTURES

Static structure (e.g. Array)	Dynamic Structure (e.g. Linked list)
1. Size of static structure is fixed i.e. it cannot be incremented or decremented during execution. At the time of creating the static structure user has to know its size	The dynamic structure size can be anything. The size of the dynamic structure can be increased or decreased at runtime. While creating the dynamic structure, user need not to specify its size.
2. Memory allocated will be contiguous	May not be contiguous.
3. There is no need to keep track of the next element	It needs to keep track of next element
4. Static Structure can be full at some point of time since its size is finite	Dynamic structure can never be full. Its size can be extended.

Advantages of Dynamic Structures:

- They can grow or shrink in size during execution, unlike arrays
- Does not waste memory space with the extra allocation
- Provide flexibility in allowing the items to be rearranged efficiently

Limitations of Dynamic Structures:

- Access to any arbitrary item is time consuming
- Some amount of memory is wasted for having the address fields in each node to link with other nodes

S T A C K S

One of the most important linear data structure is the stack.

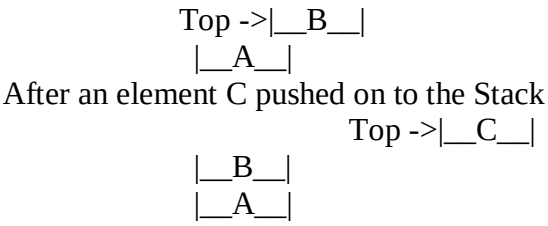
Definition: Stack is a Data Structure in which the insertions and deletions are made at the same end. This is also called as a 'LIFO' structure as the last element inserted is the first element to be deleted.

Generally in STACK

- The insertion operation is referred to as push
- The deletion operation is referred to as pop.
- The stacks are referred to as a LIFO (Last In First Out) list or FILO(First In Last Out) list.

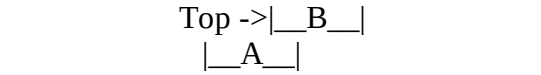
How does stack change?

One end of the stack is designated as the stack-top. New items may be pushed onto the top of the stack(in which case top of the stack moves upward to correspond to the new top element) OR an item may be popped from top of the stack(in which case top of the stack moves downward to correspond to the new top element).

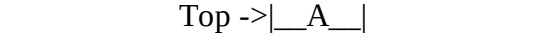


After an element C pushed on to the Stack

After an element popped from the Stack



After another element popped from the Stack



APPLICATIONS OF STACKS

1.To check the balancing of symbols like (,).
2.To implement recursive function calls.
3.Conversion of expressions
4.Evaluation of expressions.
5.In some of the Sorting Methods.
6.In memory management in operating system.
etc…….

EXPRESSIONS

There are three notations for writing expressions.

1. PREFIX Notation
2. INFIX Notation
3. POSTFIX Notation

If there is an expression ‘A+B’ , it is in INFIX notation as the operator is in middle. In PREFIX notation the same expression can be written as ‘+AB’ and in POSTFIX notation it is ‘AB+’.

'PRE','POST','IN' refer to the relative position of operator with respect to the two operands.

- In PREFIX notation the operator precedes the two operands.
- In POSTFIX notation the operator follows the two operands.
- In INFIX notation the operator is in between the two operands.

Converting an INFIX expression to POSTFIX expression:

A+(B*C) Parentheses emphasis
A+(BC*) Multiplication is Converted first
ABC*+ Addition is Converted Next
ABC*+ POSTFIX FORM

The rule during conversion process is that operations with highest precedence are converted first. To overcome the precedence, if () is used, then the part of expression inside () is to be converted first.

(A+B)*C INFIX FORM
(AB+)*C Addition is converted
AB+C* Multiplication is converted
AB+C* POSTFIX FORM

Examples:

INFIX	POSTFIX	PREFIX
A+B	AB+	+AB
A+B-CAB+C-		-+ABC
(A+B)*(C-D)	AB+CD-*	*+AB-CD
A$B*C-D+E/F/(G+H)	AB$C*D-EF/GH+/+	+-*$ABCD//EF+GH
((A+B)*C-(D-E))$(F+G)	AB+C*DE--FG+$	$-*+ABC-DE+FG
A-B/(C*D$E)	ABCDE$*/-	-A/B*C$DE

Step-by-Step Conversion from infix to Postfix and to Prefix

INFIX	POSTFIX	PREFIX
A+B	AB+	+AB
(A+B)*(C-D)	(AB+)*(CD-) AB+CD-*	(+AB)*(-CD) *+AB-CD
A-B/(C*D$E)	A-B/(C*DE$) A-B/(CDE$*) A-BCDE$*/	A-B/(C*$DE) A-B/(*C$DE) A-/B* C$DE

	ABCDE$*/-	-A/B* C$DE
A$B*C-D+E/F/(G+H)	A$B*C-D+E/F/(GH+) A$B*C-D+E/F/GH+ AB$C*-D+E/F/GH+ AB$C*-D+EF//GH+ AB$C*D-EF/GH+/+	A$B*C-D+E/F/(+GH) $AB*C-D+E/F/+GH *$ABC-D+//EF+GH -*$ABCD+//EF+GH +-*$ABC-D//EF+GH

Note: that the prefix form of a complex expression is not the mirror image of the post-fix form.

Algorithm: INFIX to POSTFIX

InfixToPostfix(Q,P) /*(Suppose Q is an arithmetic expression written in INFIX notation. This algorithm Converts it into equivalent POSTFIX Expression.) */

1. Push '(' onto STACK, and add ')' to the end of Q.
2. Scan Q form left to right and repeat steps 3 to 6 for each element of Q until the stack is empty.
3. If and operand is encountered , add it to P.
4. If a left paranthesis is encountered, push it onto STACH.
5. If an operator **X** is encountered , then
    i. Repeatedly pop from STACK and add to P each operator (on the top of STACK) which has the same precedence as or higher precedence than operator **X**.
    ii. Add operator **X** to STACK

   [End of If structure]
6. If a right parenthesis is encountered, then :
    i. Repeatedly pop from the STACK and add to P each operator (on the top of the STACK) until a left parenthesis is encountered.
    ii. Remove the left parenthesis. [Do not add the left parenthesis to P.]

   [End of If structure]

   [End of Step 2 loop.]
7. Exit.

<u>Evaluating A Postfix Expression</u>

In evaluating POSTFIX expression, while scanning from left to right if an *operand* comes PUSH it into stack, if an *operator* comes POP the two top operands 'op1' and 'op2' and perform the operation 'op2' *operator* 'op1'. PUSH the result into stack again.

This will be repeated until we reach the end of the POSTFIX expression. At the end the result will be in the stack.

Algorithm (to evaluating all single digit post fix expression eg: 623+-382/+*2$3+)

Stack - the empty stack.

/* Scan input string reading are element at a time into symbol */

```
while(not end of input)
{
    symb=next input character;
if(symb is an operand)
        push(stack,symb);
    else
```

```
{ /* Symbol is an operator */
            opnd2 = pop(stack);
            opnd1 = pop(stack);
            value = result of applying symb to opnd1 and opnd2;
            push(stack,value);
        } /* end else */
}    /* end while */
 return(pop(stack));
```

Example: ((A-(B+C))*D)$(E+F)

SYMBOL	POSTFIX STRING	OPSTACK
(		(
(		((
A	A	((
-	A	((-
(	A	((-(
B	AB	((-(
+	AB	((-(+
C	ABC	((-(+
)	ABC+	pop ((-
)	ABC+	pop (
*	ABC+-	(*
D	ABC+-D	(*
)	ABC+-D*	pop (
$	ABC+-D*	$
(	ABC+-D*	$(
E	ABC+-D*E	$(
+	ABC+-D*E	$(+
F	ABC+-D*EF	$(+
)	ABC+-D*EF+	$
	ABC+-D*EF+$	pop empty

Evaluation of a Postfix expression: 6 2 3 + - 3 8 2 / + * 2 $ 3 + result is 52

Symbol	Op1	Op2	Value	Opstack
6				6
2				2
3				3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
/	6	2	4	1,3,4
+	3	4	7	1,7
*	1	7	7	7
2	1	7	7	7,2
$	7	2	49	49
3	7	2	49	49,3
+	49	3	52	52

Unit IV: QUEUES(co5)

A Queue is a ordered collection of items in which insertions are performed at one end of the queue and deletions are made at other end of the queue. The first element inserted in queue is first element to be removed. For this reason a queue is some times called a FIFO(First In First Out) list,or LILO(Last In Last Out) list.

Types of queues

- Circular queue
- Priority queue
- Dequeue

Applications of Queues:

1. Check out line at a supermarket cash registries , queue at cinema halls
2. In CPU scheduling
3. In Printer Job sequencing.
4. Message queuing in computer network.

Operations on a Queue:

- Insertion in to a queue called enqueue.
- Deletions from a queue called dequeue.
- Traversal or Display of items.
- Find or search for an element.

PRIORITY QUEUE

The priority queue is a data structure in which the internal organization of the elements determine the result of its basic operations ENQUEUE and DEQUEUE.

There are two types of priority queues:

- Ascending Priority Queue(MIN HEAP)
- Descending Priority Queue(MAX HEAP)

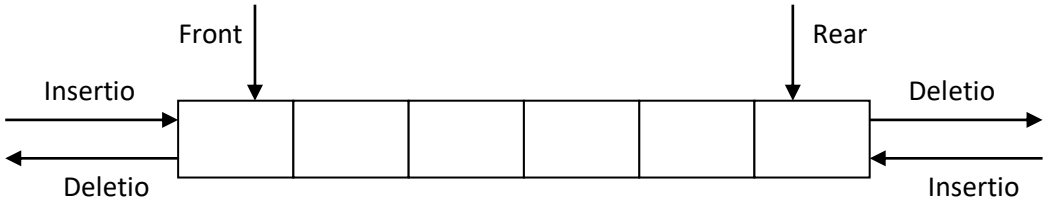
Ascending Priority Queue: This Queue is a collection of items into which items can be inserted arbitarily and from which the smallest item can be removed directly.

Descending Priority Queue: This is similar but the biggest item can be deleted directly.

DEQUEUE

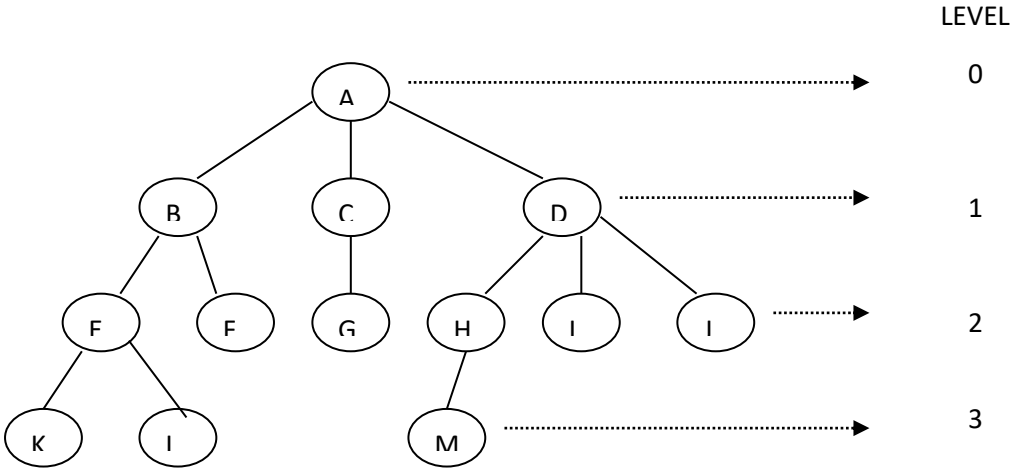
A deque is an ordered set of items from which items may be deleted at either end and into which items may be inserted at either end. There are insert begin, insert end, delete begin, delete end oprerations.

1. The input restricted dequeue allows insertions at only one end
2. The output restricted dequeue permits deletions from one end only



TREES

A ***tree*** is a finite set of one or more elements such that there is a specially designated node called the ***root.*** The remaining nodes are partitioned into n > = 0 disjoint sets $T_1 \ldots\ldots T_n$ where each of these sets is a tree $T_1 \ldots\ldots T_n$ are called the subtrees of the root.



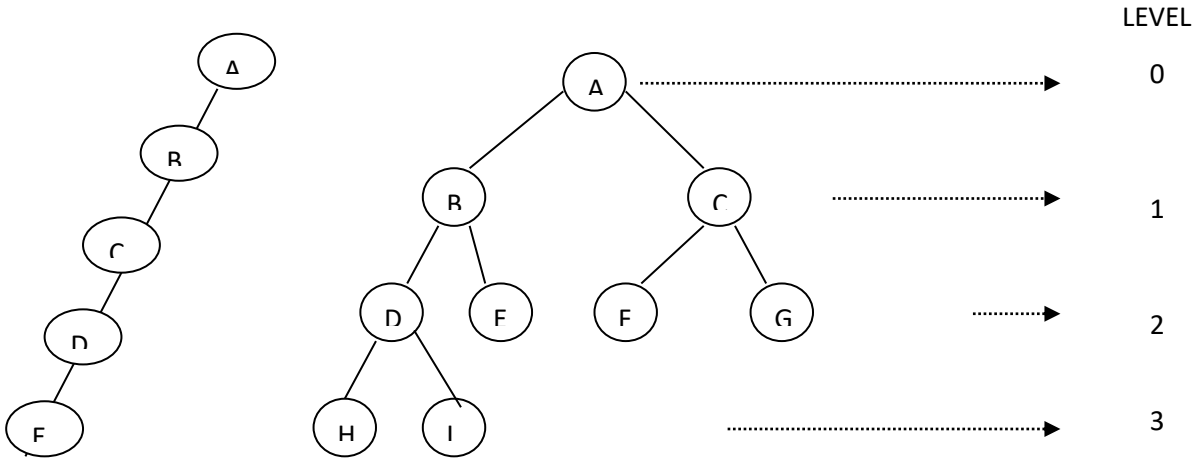
- The root contains A, and we normally draw trees with the root at the top.
- The number of *childs* of a node is called its ***degree.*** The degree of A is 3 and C is 1.
- Nodes that have degree zero are called terminal nodes (K, L, M, etc.) or ***leaf nodes***.
- A is a parent node of child nodes (B, C, D). ***childs*** of the same parent are said to be ***siblings***.
- The degree of the tree is the max degree of the nodes in the tree.
- The ***level*** of a node is defined by initially letting the root be at level Zero. If a node is at level P, then its children are at level at P+1.
- The ***height*** or ***depth*** of a tree is defined to be the max level of any node in the tree.
- is a A forest set of n>=0 disjoint trees.

BINARY TREES

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

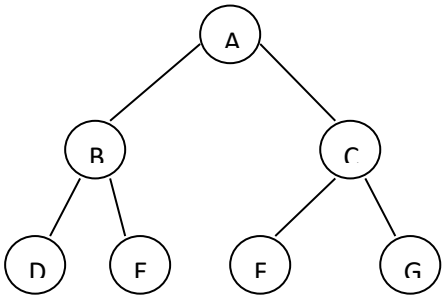
Fig (a) is left skewed tree.

(b) is a complete binary tree



(a). Skewed Tree (b). Complete Binary Tree

The binary tree of depth k having exactly $2^k - 1$ nodes is called a full binary tree of depth k.

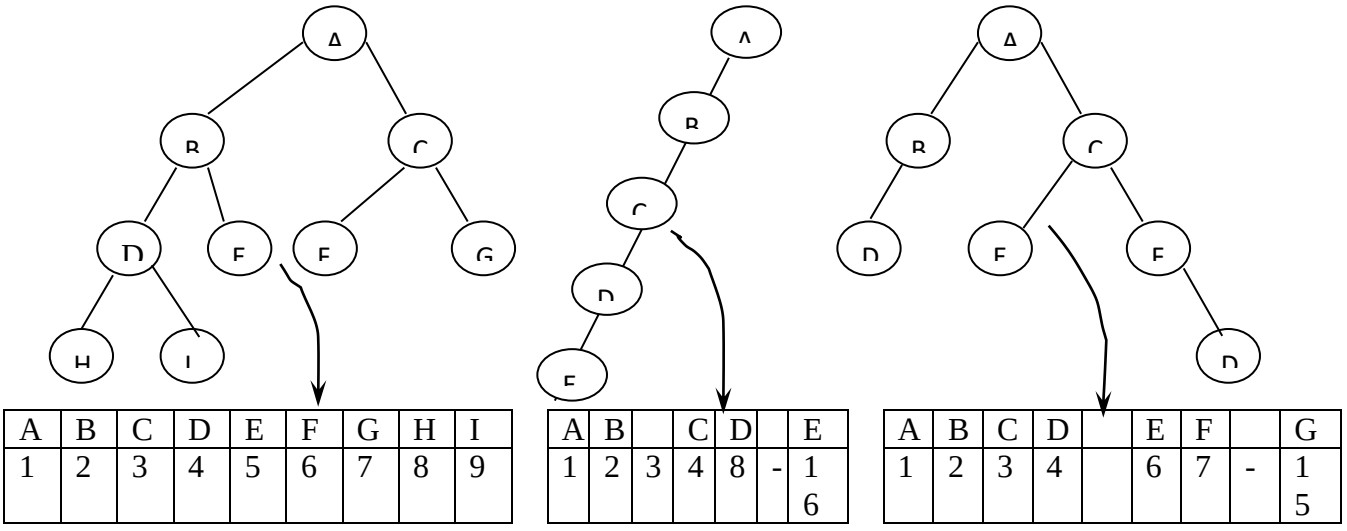


(b). Full Binary Tree of depth 3

- If a complete binary tree with n nodes is represented sequentially, then for any node with index i, 1<=i<=n, we have
  - parent (i) is at [i/2] if i#1 when i=1, i is the root and has no parent
  - left child (i) is at 2i if 2i<n if (2i >n), i has no left child
  - right child (i) is at 2i+1 if 2i + 1 <n if (2i+1 >n), i has no right child

Sequential representation of binary tree

- If left child -> 2i position
- If right child -> 2i+1 position

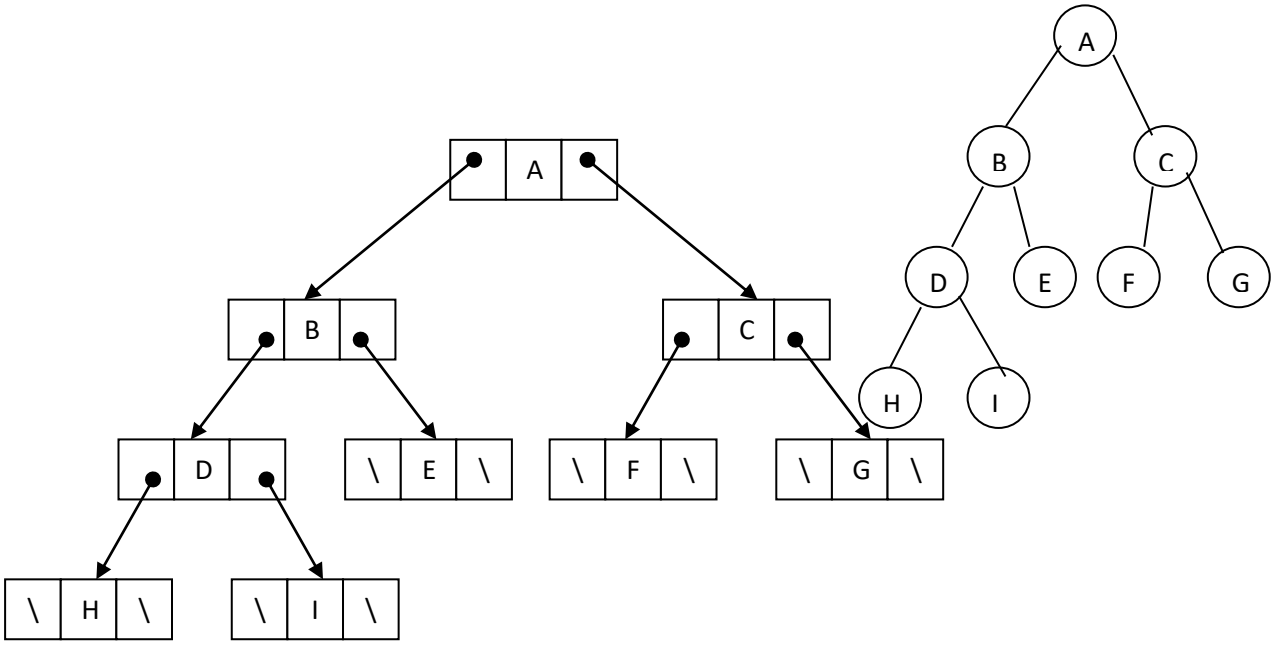


A	B	C	D	E	F	G	H	I
1	2	3	4	5	6	7	8	9

A	B		C	D		E
1	2	3	4	8	-	16

A	B	C	D		E	F		G
1	2	3	4		6	7	-	15

To represent the binary tree using linked lists, we use the concept of doubly linked list, consisting of two pointers, a left pointer pointing to left child and a right pointer pointing to the right child. If a node doesn't have left or right child the corresponding fields are ***NULL****.*

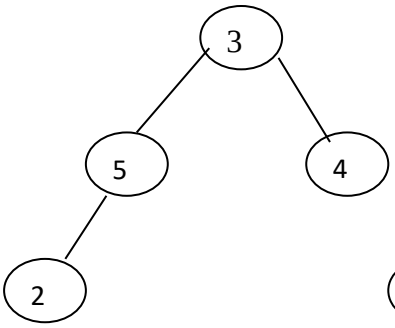


BINARY SEARCH TREE

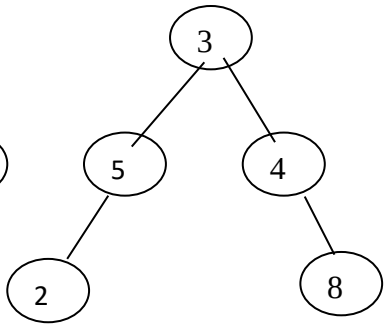
A binary search tree is a binary tree if it satisfies the following properties.

- An empty tree is a binary search tree.
- A tree with single element (root) is a binary search tree.
- Every element has a key and no two elements have the same key
- The keys in the ***left subtree*** are smaller than the key in the paren***t***
- The keys in the ***right subtree*** are larger than the key in the ***parent***
- The left and right subtrees are also binary search trees.

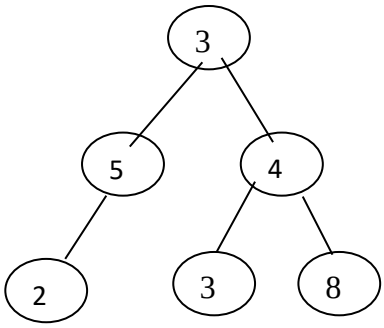
A binary search tree can support the operations search, insert and delete.
Consider a tree below with the values 30, 5, 40, 2



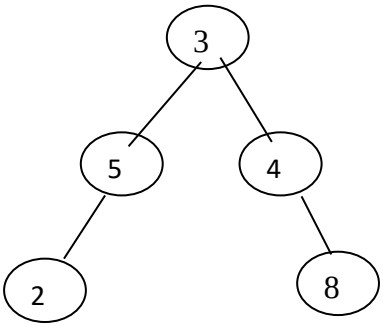
Now insert 80 into tree



Now insert 35 into tree

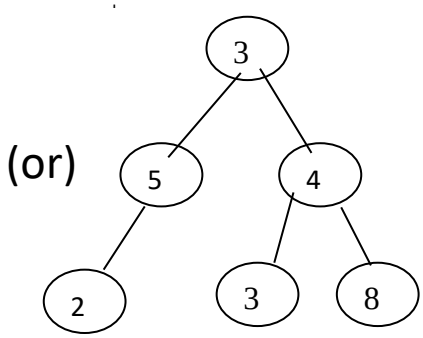


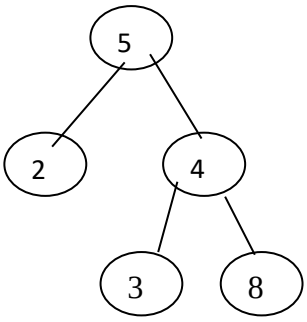
Now Delete 30 from tree



Now Delete 30 from tree

(or)

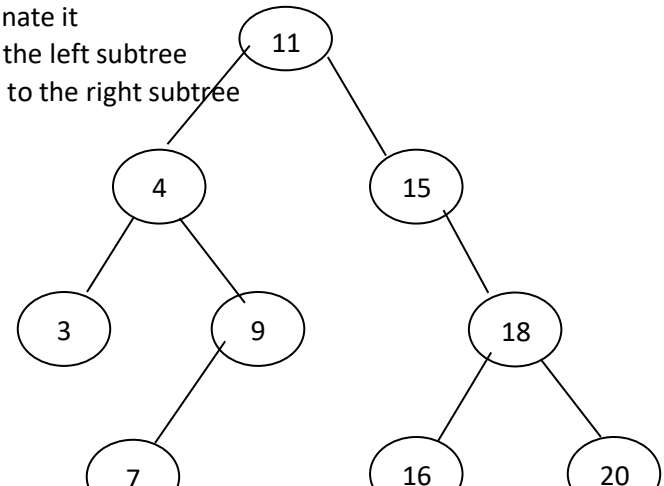




Here 5 is moved up to root and after that 5 is deleted

For Inputs : 14, 15, 4, 9, 7, 18, 3, 5, 16, 20, 17, 9, 4, 5

- If duplicates then eliminate it
- If less than root, go to the left subtree
- If greater than root, go to the right subtree



BINARY TREE TRAVERSALS

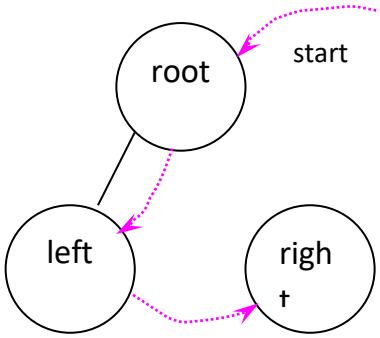
There are basically three types of traversals.

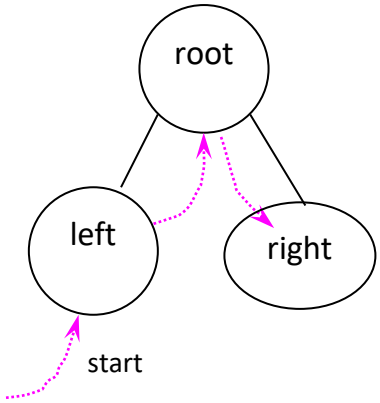
- Preorder Traversal (Root, Left, Right)
- Inorder Traversal (Left, Root, Right)
- Postorder Traversal (Left, Right, Root)

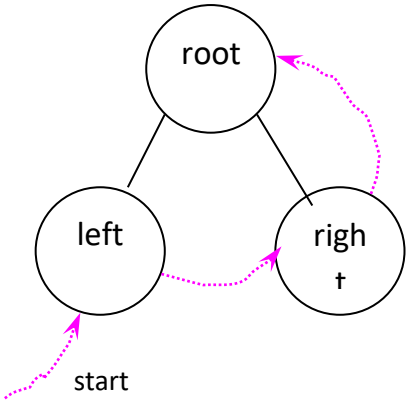
Preorder:
1. Visit the root
2. Traverse the left subtree in preorder
3. Traverse the right subtree in preorder

Inorder:
1. Traverse the left subtree in preorder
2. Visit the root
3. Traverse the right subtree in preorder

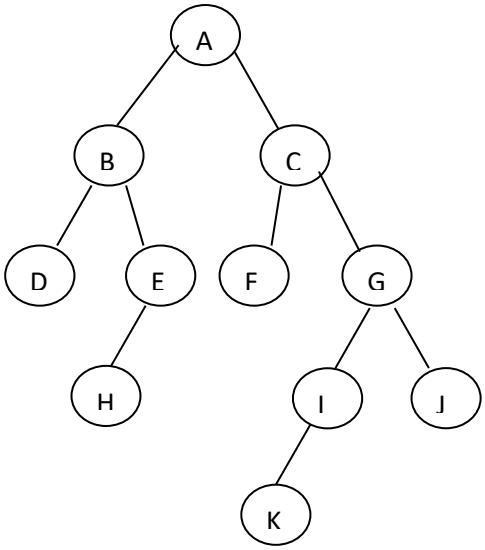
Postorder:
1. Traverse the left subtree in preorder
2. Traverse the right subtree in preorder
3. Visit the root



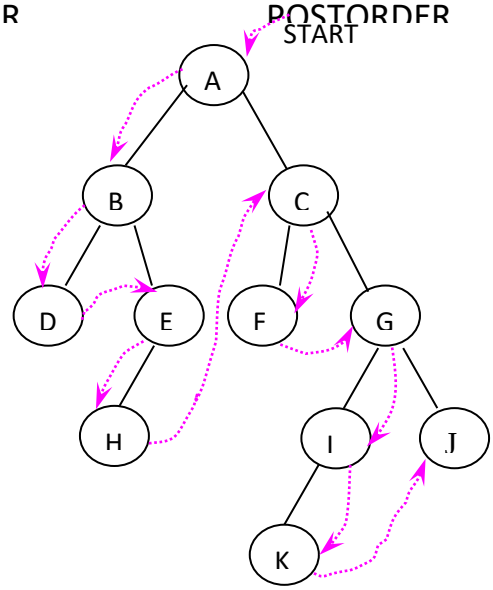




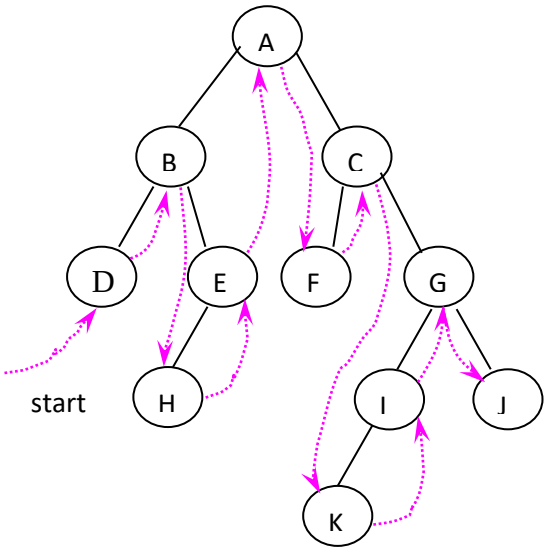
PREORDER INORDER POSTORDER



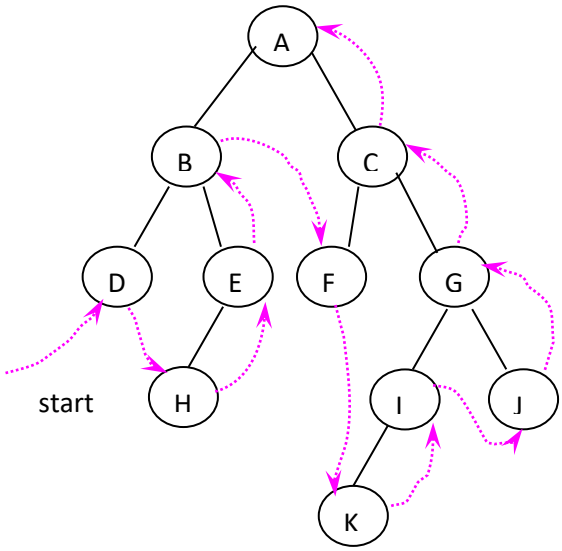
BINARY TREE



PREORDER : A B D E H C F G I K J



INORDER : D B H E A F C K I

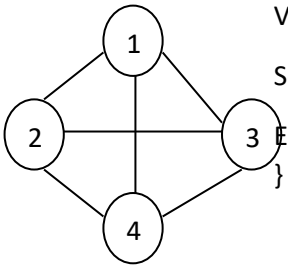


POSTORDER :D H E B F H I J G C A

16. GRAPHS

A graph G is defined as G={V,E} where

- The set V is a finite, non empty set of vertices
- The set E is a set of pairs of vertices, these pairs are called edges
  The notations V(G) and E(G) represents the sets of vertices and edges.
  G=(V,E) to represent a graph.
- In an undirected graph the pairs of vertices representing any edge is unordered. Thus the pairs (u,v) and (v, u) represent same edge.

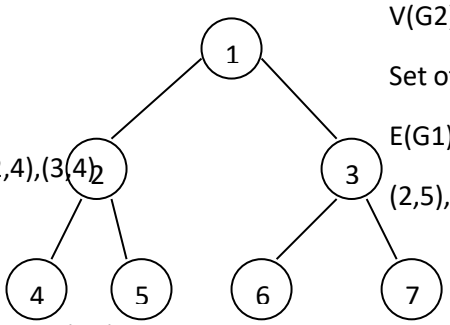


V(G1)={1,2,3,4}

Set of vertices

E(G1)={(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)}

(G1). Undirected Graph



V(G2)={1,2,3,4,5,6,7}

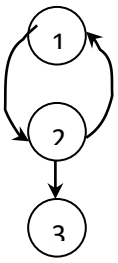
Set of vertices

E(G1)={(1,2),(1,3),(2,4),

(2,5),(3,6),(3,7)}

(G2). Undirected Graph

- In a directed graph each edge is represented by a directed pair (u,v) u is a tail and v the head of edge.
  Therefore (u, v) and (v, u) are different edges.



G3

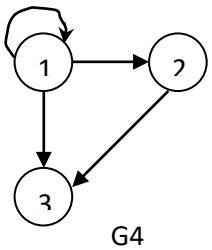
V(G3) = {1,2,3} E(G3)={(1,2),(2,1),(2,3)}

Here (1,2) & (2,1) are not same they are different edges

- We define edges and vertices of graph as sets,
  1. A graph may not have an edge from a vertex v back to itself. That is edges of from (v,v) or (u,u) are not legal. Such edges are known as self edges or self loops.

2. If we permit self edges, we obtain a data object referred to as a graph with self edges.

eg:



G4

edge on vertex 1 is a self edge

- In case of directed graph on n vertices, the max no of edges is n(n-1).
- If(u,v) is an edge in E(G), then we say vertices u and v are adjacent (u and v are adjacent to each other if it is a undirected graph, if it is a directed graph v is adjacent to u) and (u,v) is incident on vertices u and v . So vertices adjacent to vertex 2 in graph G2 are 4,5,1.
- The edges incident on vertex 3 in G2 are (1,3), (3,6) and (3,7)
- The edges incident to vertex 2 are (1,2), (2,1) and (2,3)
- The vertex adjacent to 2 in G4 is 3.

A path from vertex u to vertex v in graph G is a sequence of vertices u, i_1, i_2, ----i_k, v such that (u,i_1), (i_1,i_2) ---(I_k,v) are edges in E(G)

The length of a path is the number of edges on it

- A simple path is a path in which all vertices except possibly the first and last are distinct.

A path such as (1,2) (2,4) (4,3) is also written as 1,2,4,3.

Paths 1,2,4,3 and 1,2,4,2 of G1 are of length 3

Simple path not a simple path

A cycle is a simple path in which the first and last vertices are same

The path 1,2,3,1 is a cycle in G1

The path 1,2,1 is a cycle in G3

The degree of a vertex is the number of edges incident on to that vertex

The degree of vertex 1 in G1 is 3

The in degree of a vertex v is the number of incoming edges

The out degree of a vertex v is the number of outgoing edges

Degree of a vertex v = in degree of v + out degree of v

Vertex 2 of G3 has In degree 1, Out degree 2, Degree 3

GRAPH REPRESENTATIONS

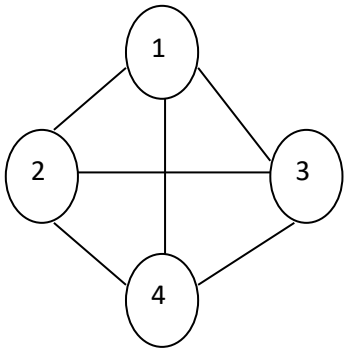
- Adjacency matrix
- Adjacency lists
- Adjacency multilists

Let G=(V,E) be a graph with n vertices , n>=1. The adjacency matrix of G is a two dimensional n*n array,

say a with the property a[i,j] =1 iff the (i,j) & (i,j) (directed graph) is in E(G)

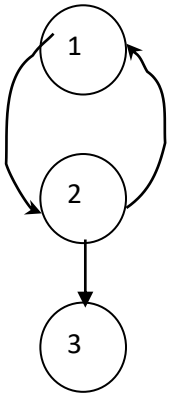
a[i,j] =0 if there is no such edge in G

from adjacency matrix, we can readily determine whether there is an edge connecting any two vertices i and j.



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

Adjacency matrix



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

Adjacency matrix

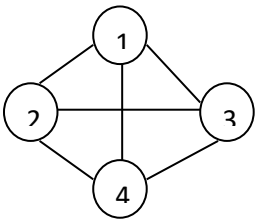
for a directed graph the row sum is the *out degree*, col. sum is the *in degree*

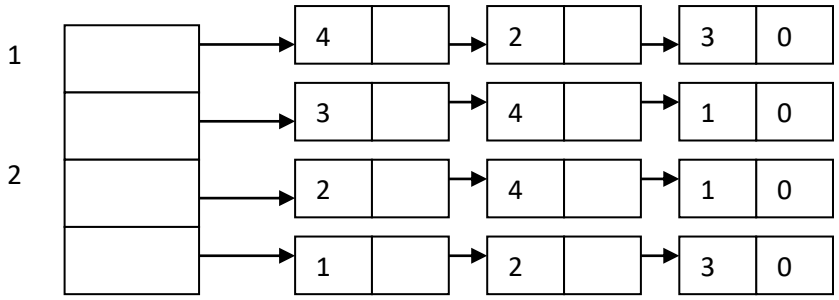
Adjacency lists:

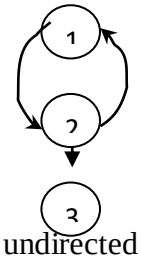
The n rows of adjacency matrix are represented as n linked lists

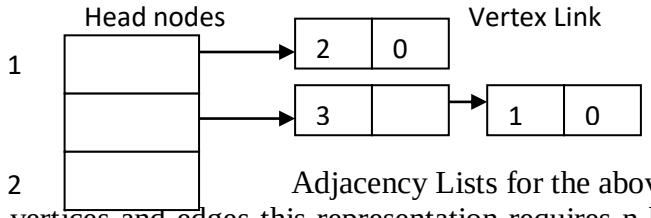
There is one list for each vertex in G the nodes in the list i represent the vertices that are adjacent from vertex i.

Each node has two fields' vertex and link.









Adjacency Lists for the above graphs

For an undirected graph with n vertices and edges this representation requires n head nodes and 2e list nodes.

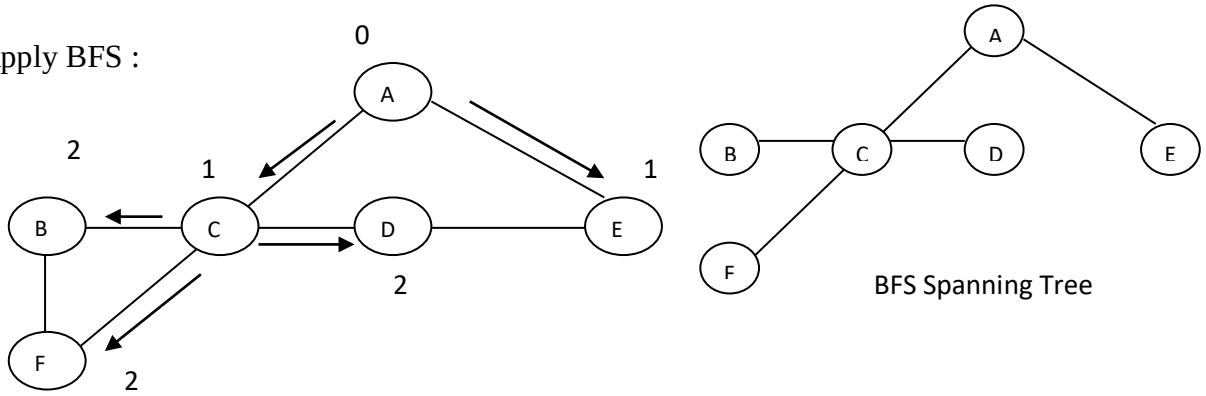
GRAPH TRAVERSALS

Breadth first search:

Breadth first search can be used to find the shortest distance from some starting node to the remaining nodes of the graph.

- This shortest distance is the minimum number of edges traversed in order to travel from the start node to the specific node
- Starting at a node v, this distance is calculated by examining all incident edges to node v and then moving on to an adjacent node w and repeating the process
- The traversal continuous till all nodes in the graph are examined or no new node is visited in the last iteration.

Apply BFS :

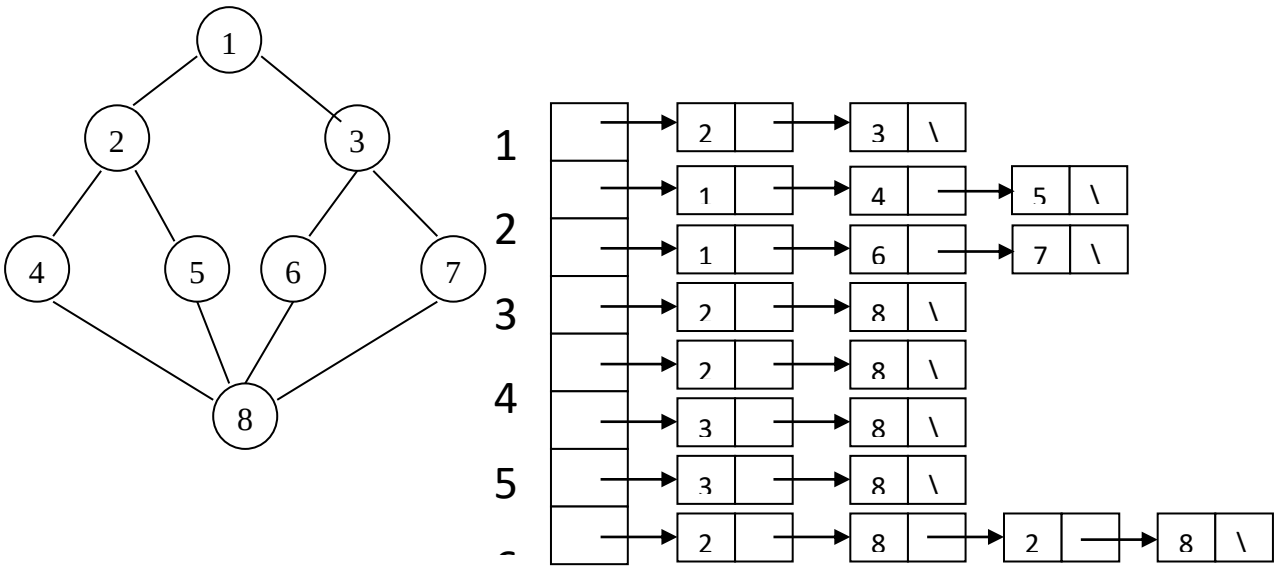


BFS Traversal : A C E B D F

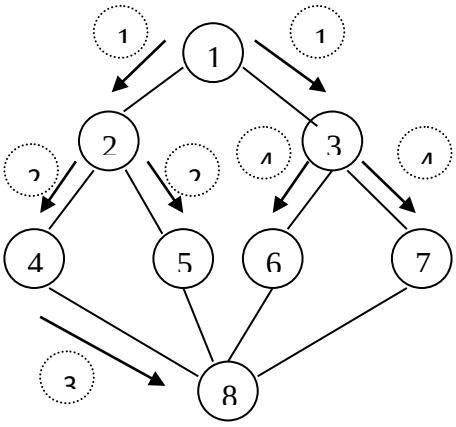
If BFS is used on a connected undirected graph G, then all vertices in G get visited and graph is traversed

If G is not connected, then at last, at least one vertex of G is not visited.

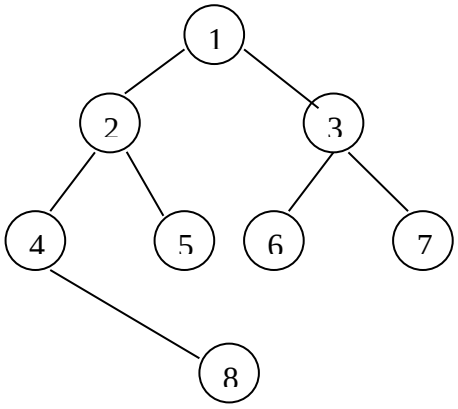
Adjacency list for graph



BFS traversing for above graph



BFS Spanning tree

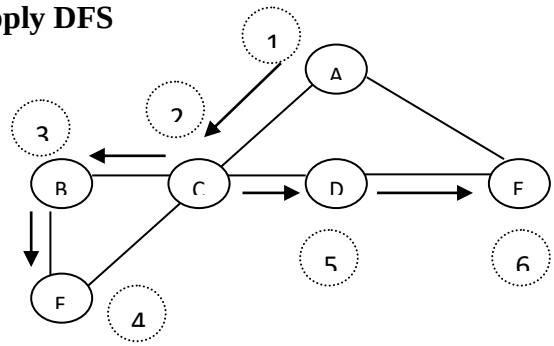


Depth first search:

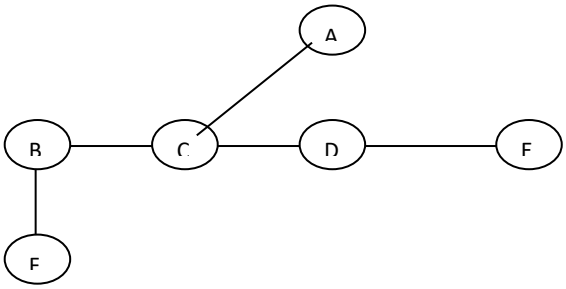
A node S is picked as a start node and marked. An unmarked adjacent node to S is selected and marked, becoming the new start node, leaving the original start node with unexplored edges for present.

- The search continues in the graph until the current path ends at a node with out degree zero or at a node with all adjacent nodes already marked.
- Then the search returns to the last node which still has unmarked adjacent nodes and continuous marking until all nodes are marked.

Apply DFS



DFS Spanning tree



Unit V: *Searching and Sorting(co3)*

Finding whether a required element is there or not in the given list of elements is known as ***searching.***

The efficiency of a search depends upon three things

- The size and organization of the list of elements we are searching
- The search method being used
- The efficiency of test condition used to determine if search is successful

Two Popular Search methods are:

- Linear search
- Binary search

LINEAR SEARCH

Search starts at the beginning of the list and search for the element until a match is found or the list is completed without a match.

- In the best case, the element will be found in the first comparison. Hence best case complexity = O(1).

- If there are n items in the list, then in the worst case (i.e when the element required is not there in the list) n comparisons are needed. Hence Worst case complexity = O(n).

- The avg case is also proportional to n and Avg case complexity = O(n).

Program – Linear search

```
#define size 10

int a[size],last=-1;
int l_search(int);

void main()
{
  int key,position,x;
  do
  {
printf("Enter A Value, (0) to Stop : “);
scanf("%d",&x);
     if(x!=0)
              a[++last]=x;
}while(x!=0&&last<size-1);
printf("enter the element to be searched");
scanf("%d",&key);
  position=l_search(key);
  if(position==-1)
printf("un successful search\n");
  else
printf("element found in %d location",position);
}

int l_search(int key)
/* Returns the position of the key, if the key not found returns –1 */
{
  int i;
  for(i=0;i<=last;i++)
  {
    if(a[i]==key)
                return(i+1);
  }
return(-1);
}
```

BINARY SEARCH

Binary search method proceeds by examining the middle element with the following operations.

If middle element = target element then success.
Else if middle element < target element then search the list after the middle.
Else search the list before middle element.

- Each time the comparison is made ,and if it is not the success then the list is divided into parts and the search continues in one of them.

- The search requires at most log n comparisons.Hence both average and worst cases of a binary search is proportional to $\log_2 N$ i.e O(log n)

- Best case of theBinary search comes when the target is found in the first comparison only. Hence best case complexity = O(1).

Program – Binary Search

```
#define size 10

int a[size],key,last;

int bsearch(int,int,int);

void main()
{
 int position,x;
 do
   {
printf("Enter A Value, (0) to Stop : “);
scanf("%d",&x);
     if(x!=0)
              a[++last]=x;
}while(x!=0&&last<size-1);
printf("enter the element to found");
scanf("%d",&key);
  position=bsearch(0,last,key);
  if(position==-1)
         printf("un successful search");
  else
         printf("element is found in %d position",position);
}


int bsearch(int low,inthigh,int key)
/* Returns the position of the key, if the key not found returns –1 */
{
```

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

Arranging the elements in a desired order (ascending or descending) is known as ***sorting***

The popular methods for Sorting are:

1. **Bubble sort (exchange sort).**
2. **Selection sort.**
3. **Insertion sort**
4. **Radix sort.**
5. **Bucket sort.**
6. **Merge sort.**
7. **Quick sort(Partition-Exchange Sort)**
8. **Heap sort**
9. **Tree sort**

Below we will go through some of the important sorting methods.

Bubble sort (exchange sort)

This is a simple but not very efficient method in which iterations goes through the list, swapping adjacent elements that are out of order until all the elements are in the order.

- The algorithm requires n-1 passes, if the list contain n elements since each pass will place one item in its correct place.
- Bubble sort must be avoided if the number of elements to be sorted are more.
- The Time Complexity of Bubble Sort = $O(n^2)$.

Selection sort

In this sort, in each iteration it will search the list to find the smallest element. Then it swaps this with the first element in the list and process continues with the remaining list , till the list is arranged in a sorted order.

The number of comparisons needed is n-1 in the first iteration, n-2 in the second iteration, n-3 in the third iteration, and so on and n-i in the i^{th}iteration. Hence there are at most (n-1)+(n-2)+-----+1=n(n-1)/2 comparisons, time complexity is $O(n^2)$.

In the i^{th} iteration the algorithm finds the lowest element A[I],A[I+1],…..A[n], and swaps it with A[I]. It means, for each pass I, selection sort ensures that the elements in positions 1 to I are in sorted order.

Quick sort (Partition exchange sort) :

Quick sort is the one of the fast sorting algorithm.

1. It is better algorithm when compared to merge sort as it requires no auxiliary memory.
2. It has $O(N\log_2 N)$ Time complexity for best and average case performance, and $O(N^2)$ for worst case performance.

3. The worst case will come when the elements list is already sorted.

In Quick sort, we divide the array of items to be sorted into two partitions and then call the quick sort procedure recursively to sort the two partitions.

Example :

Suppose A is the array of 12 elements:

Consider the first element in the list as PIVOT, L and R are the left and right indexes. Position R at the last element, i.e., Infinity which we will suffix to the given list as shown below.

Note: Underscore(_) represents L element and Bar represents R element.

pivot	L											R
(44)	33	11	55	77	90	40	60	99	22	88	66	∝

step 1: from the L search for the big element towards R, and from the R search for small element towards L.

			L						R		
(44)	33	11	55	77	90	40	60	99	22	88	66

if L<R then swap elements at L and R and continue step 1 until L>R. In this occurrence 55 and 22 will be swapped. And the process continues..

			L						R		
(44)	33	11	22	77	90	40	60	99	55	88	66
(44)	33	11	22	77	90	40	60	99	55	88	66
(44)	33	11	22	40	90	77	60	99	55	88	66
				R	L						
(44)	33	11	22	40	90	77	60	99	55	88	66

Here L>R and so stop searching, Now when L>R swap the PIVOT element with R element.

40	33	11	22	(44)	90	77	60	99	55	88	66
	First List				Second List						

Now the PIVOT element is in its correct position. Such that all elements in the left of PIVOT are less than that. And all elements in the right are greater than PIVOT. This will form two sub-lists, one to the left of the PIVOT and another to the right of PIVOT.Now Repeat the above process for the first list, then for the second list until a sub-list contains only one element which need not to be sorted.

Program – QUICK SORT:

```
#define size 20

int a[size];

void q_sort(int,int);
int partition(int,int);

void main()
{
```

```
 int i=0,x,j;
 do
 {
printf("Enter a value, (0) to Stop : ");
scanf("%d",&x);
  if(x!=0)  a[i++]=x;
}while(x!=0&&i<size);
q_sort(0,i-1);
 for(j=0;j<=i-1;j++)
          printf("%3d",a[j]);
 }

void q_sort(int low,int high)
{
 int p;
 if(low<high)
 {
          p=partition(low,high);
          q_sort(low,p-1);
          q_sort(p+1,high);
 }
}
```

```
int partition(int low,int high)
{
 int pivot,l,r,t;
 pivot=low;  l=low+1;  r=high;
 do
 {
  while(a[l]<a[pivot])
          l++;
  while(a[r]>a[pivot])
          r--;
  if(l<r)
  {
   t=a[l];
   a[l]=a[r];
   a[r]=t;
  }
}while(l<r);

 t=a[pivot];
 a[pivot]=a[r];
 a[r]=t;
 return r;
}
```

Merge sort

This is a sorting algorithm which will use divide and conquer strategy. In this method, in each step the list will be divided into two equal parts and individually sorted and then will be combined into the required list. For sorting the sub-lists we will use the same approach if the list contains more than one element.

1. The time complexity of merge sort is O(NlogN) for all cases.
2. This needs extra memory for merging purpose.

Program :

```
#include<stdio.h>
#define size 20

int a[size];
void m_sort(int,int);
void merge(int,int,int);

void main()
{
 int i=0,x,j;
 do
 {
printf("Enter a value, (0) to Stop : ");
scanf("%d",&x);
  if(x!=0)
  a[i++]=x;
 } while(x!=0&&i<size);
m_sort(0,i-1);
printf("Sorted Order is : ");
 for(j=0;j<=i-1;j++)
printf("%3d",a[j]);
}


void m_sort(int low,int high)
{
 int mid;
 if(low<high)
 {
  mid=(low+high)/2;
m_sort(low,mid);
m_sort(mid+1,high);
  merge(low,mid,high);
 }
}
```

```
void merge(int low,intmid,int high)
{
 int b[size],i,j,k,l;
i=low;
 j=mid+1;
 k=low;
 while(i<=mid&&j<=high)
 {
  if(a[i]<a[j])
          b[k++]=a[i++];
  else
          b[k++]=a[j++];
 }
 if(i>mid)
 {
  for(l=j;l<=high;l++)
          b[k++]=a[l];
 }
 else
 {
  for(l=i;l<=mid;l++)
          b[k++]=a[l];
 }
 for(l=low;l<=high;l++)
          a[l]=b[l];
}
```