USN | | | | | | | | | | |

Internal Assessment Test 3 – Nov. 2017

Sub:	Programming in C & Data Structure					Sub Code:	17PCD13	Branch:	Chemistry Cycle
Date:	17-11-2017	Duration:	90 min's	Max Marks:	50	Sem/Sec:	1st/ALL SECTIONS		OBE

Answer any FIVE FULL Questions

		MARKS	CO	RBT
1 (a)	What is a file? Explain fopen() with different modes and fclose() with syntax and example.	[1+5]	CO4	L2
(b)	Explain with syntax fprintf(), fscanf(), fgets(), fputs(). Give an example for each.	[4]	CO4	L2
2	Given two information files "ramayan.txt" and "mahabharat.txt" that contains details of character respectively. Write a C program to create a new file called "output.txt" and copy the content of files "ramayan.txt" and "mahabharat.txt" into output file.	[10]	CO4	L3
3 (a)	What is a pointer? Explain declaration and initialization of a pointer with example.	[4]	CO5	L2
(b)	Explain with examples pointer arithmetic and pointer to an array.	[6]	CO5	L3
4 (a)	Explain with syntax and example fseek(), rewind(), ftell().	[3+1+2]	CO4	L2
(b)	Explain Pointer to Pointer with example.	[4]	CO5	L3

USN | | | | | | | | | | |

Internal Assessment Test 3 – Nov. 2017

Sub:	Programming in C & Data Structure					Sub Code:	17PCD13	Branch:	Chemistry Cycle
Date:	17-11-2017	Duration:	90 min's	Max Marks:	50	Sem / Sec:	1st/ALL SECTIONS		OBE

Answer any FIVE FULL Questions

		MARKS	CO	RBT
1 (a)	What is a file? Explain fopen() with different modes and fclose() with syntax and example.	[1+5]	CO4	L2
(b)	Explain with syntax fprintf(), fscanf(), fgets(), fputs(). Give an example for each.	[4]	CO4	L2
2	Given two information files "ramayan.txt" and "mahabharat.txt" that contains details of character respectively. Write a C program to create a new file called "output.txt" and copy the content of files "ramayan.txt" and "mahabharat.txt" into output file.	[10]	CO4	L3
3 (a)	What is a pointer? Explain declaration and initialization of a pointer with example.	[4]	CO5	L2
(b)	Explain with examples pointer arithmetic and pointer to an array.	[6]	CO5	L3
4 (a)	Explain with syntax and example fseek(), rewind(), ftell().	[3+1+2]	CO4	L2
(b)	Explain Pointer to Pointer with example.	[4]	CO5	L3

5	What is dynamic memory allocation? Explain the different functions used for allocating and de-allocating memory with syntax and example.	[10]	CO5	L3
6	What is preprocessor? Explain any 5 pre-processer directives with an example for each.	[10]	CO6	L2
7(a)	Write a C program to compute sum, mean and standard deviation for an array of real numbers using pointers.	[5]	CO5	L3
(b)	Write a short note on Queues and Trees.	[5]	CO6	L2
8 (a)	What is data structure? List all primitive and non-primitive data structures.	[4]	CO6	L2
(b)	Write a short note on Stacks and Linked Lists.	[6]	CO6	L2

5	What is dynamic memory allocation? Explain the different functions used for allocating and de-allocating memory with syntax and example.	[10]	CO5	L3
6	What is preprocessor? Explain any 5 pre-processer directives with an example for each.	[10]	CO6	L2
7(a)	Write a C program to compute sum, mean and standard deviation for an array of real numbers using pointers.	[5]	CO5	L3
(b)	Write a short note on Queues and Trees.	[5]	CO6	L2
8 (a)	What is data structure? List all primitive and non-primitive data structures.	[4]	CO6	L2
(b)	Write a short note on Stacks and Linked Lists.	[6]	CO6	L2

SOLUTIONS

1 (a) What is a file? Explain fopen() with different modes and fclose() with syntax and example.

A file is sequence of bytes on the disk where a group of related data is stored.

→ Opening a file.

The general syntax for opening a file is:

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

fp → pointer to the data type FILE. contains all the information about the file.

filename → a string that holds the name of the file on the disk.

mode → a string representing how you want to open the file.

The different modes of opening a file are:

File Mode	Meaning of Mode	During inexistence of file
r	Open the file for reading	returns NULL.
w	Open the file for writing	If exists – contents are overwritten Not Exists – it creates a new file for writing.
a	Open the file for adding or appending data to the end of the file.	It will create a new file.
r+	Open the file for reading and writing	returns NULL

w+	Open the file for reading and writing	If exists – contents are overwritten Not exists – new file will be create
a+	Open the file for both reading and appending	Not exists – new file will be created.

→ closing a file

★ A file must be closed as soon as all the operations on it have been completed.

★ Ensures all information associated with the file is flushed out from the buffers and all links to the file are broken.

★ Prevents accidental misuse.

★ If there is a limit to the no^r of files that can be kept open simultaneously, closing of unwanted files might help open the required files.

★ When you want to reopen the same file in a different mode.

The general syntax for closing a file is

```
fclose(file-pointer);
```

Eg:-

```
FILE *fp,*fp1;
fp= fopen ("test.txt", "r");
fp1= fopen ("test.txt", "w");
...
...
fclose (fp);
fclose (fp1);
```

(b) Explain with syntax fprintf(), fscanf(), fgets(), fputs(). Give an example for each.

★ fprintf()

The general syntax is:

```
fprintf (file-pointer, "format specifiers", list);
```

list → variables, constants, strings.

```
char name[30];
int age;
```

Eg:- `fprintf (fl, "%s %d %f", name, age, 7.5);`

★ fscanf()

The general syntax is:

```
fscanf (file-pointer, "format specifiers", list);
```

- It returns EOF when end of file is reached.
  (The value is -1) → MACRO
  ↳ A single instruction that expands automatically into a set of instructions to perform a particular task.

Note: fprintf() and fscanf() returns the no^r of characters read/print

(1) `int fprintf(FILE *stream, const char *format, ...);`

(2) `int fscanf(FILE *stream, const char *format, ...);`

Eg:-
```
main()
{
    int num;
    FILE *fp;
    fp = fopen ("hello.txt", "w");
    if (fp == NULL)
    {
        printf ("Error");
        exit (1);
    }
    printf ("Enter num: ");
    scanf ("%d", &num);
    fprintf (fp, "%d", num);
    fclose (fp);
}
```

Note: exit is system call that terminates the process.

→ Eg:-
```
main()
{
    int num;
    FILE *fp;
    fp = fopen ("hello.txt", "r");
    if (fp == NULL)
    {
        printf ("Error");
        exit (1);
    }
    fscanf (fp, "%d", &num);
    printf ("value of num = %d", num);
    fclose (fp);
}
```

fgets()

→ The general syntax is: (or) string

```
char *fgets (char *s, int size, FILE *stream);
```

size → max no of characters to be read

→ Reads in at most one less than size characters from stream and stores them into the buffer pointed to by s

→ Reading stops after an EOF or \n is encountered.

→ Note: '\n' is also read and stored in the buffer.

→ A terminating null character ('\0') is stored after the last character in the buffer.

Example:

```
main()
{
    char s[50];
    fgets (s, 50, stdin);
    printf ("%s", s);
}
```

→ standard input device i.e., Keyboard.

Note:-

```
char s[50];
fgets (s, 3, stdin); //akhilaa
printf ("%s", s); //ak
```

→ This function returns s on success and NULL on error or when EOF occurs when no characters have been read.

* fputs()

→ The general syntax is : (or) string

int fputs (const char *s, FILE * stream);

const char *s → points to a set of characters in memory that is constant (which cannot be changed

FILE * stream → file pointer . where set of characters (string) will be written .

→ All the characters in strings except the '\0' are written to file stream .

→ This function returns .

- → on success - no^r of characters written
- → on error - EOF

Example :

```
main()
{
    FILE *fp;
    fp = fopen ("print.txt", W);
    fputs ("Welcome to PCD!!!", fp);
    fclose(fp);
}
```

2 Given two information files “ramayan.txt” and “mahabharat.txt” that contains details of character respectively. Write a C program to create a new file called “output.txt” and copy the content of files “ramayan.txt” and “mahabharat.txt” into output file.

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

int main()
{
        FILE *fp1, *fp2, *fp3;
        Char buff1[100], buff2[100];

        fp1 = fopen ( "ramayan.txt", "r" );
```

```
        if ( fp1 == NULL)
        {
                printf ( "Error in opening file 'ramayan.txt'\n");
                exit ( 0 );
        }

        fp2 = fopen ( "mahabharat.txt", "r" );
        if ( fp2 == NULL )
        {
                printf ( "Error in opening file 'mahabharat.txt'\n");
                exit ( 0 );
        }

        fp3 = fopen ( "out.txt", "w" );
        if ( fp3 == NULL )
        {
                printf ( "Error in opening file 'out.txt'\n");
                exit ( 0 );
        }

        while ( 1 )
        {
                fscanf ( fp1, "%s", buff1 );
                fscanf ( fp2, "%s", buff2 );

                if ( !feof ( fp1 ) && !feof ( fp2 ) )
                        // copy to out.txt file.
                        fprintf ( fp3, "\n%s\t%s", buff1, buff2 );
                else
                        break;
        }
        fclose ( fp1 );
        fclose ( fp2 );
        fclose ( fp3 );

        return 0;
}
```

3 (a) What is a pointer? Explain declaration and initialization of a pointer with example.

Pointers are variables that stores addresses as their values.

Declare a pointer variables

→ The general syntax ;

data-type * pointer_name ;

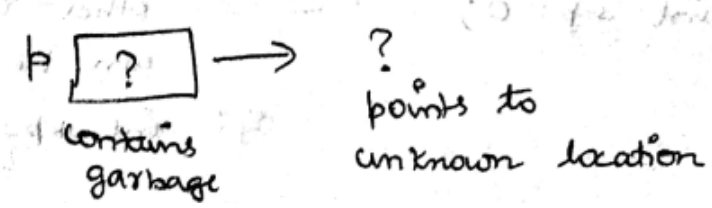
↑ tells that pointer_name is a pointer variable.

pointer_name points to a variable of type data_type.

→ eg:- int *p ; //Integer pointer

p is a pointer variable that points to a integer data type.

float *x ; // float pointer.



contains garbage

points to unknown location

Initialization of pointer variables

→ The process of assigning the address of a variable to pointer variable is called initialization.

→ Use assignment operator to initialise the variable.

Eg:-
```
int *p; // declaration
int a;
p = &a; //Initialization
```
(or)
```
int *p = &a; // note a must be declared before initializing.
```
Note, this is initializing p and not *p.

→ Always ensure that the pointer variables point to the corresponding type of data.
```
float a, b;
int x, *p;
p = &a;    // Wrong.
b = *p;
```

→ `int x, *p = &x; //valid`

declares x as an integer variable, p as a pointer variable and then initializes p to the address of x.

`int *p = &x, x; // Invalid.`

→ Also define a pointer variable with an initial value of NULL or 0 (zero).
```
int *p = NULL;
int *p = 0;
```
Note: Apart from NULL & 0 no other constant value can be assigned.

Eg:- `int *p = 4250; // Wrong`

(b) Explain with examples pointer arithmetic and pointer to an array.

→ A pointer is an address, which is a numeric value.
→ ∴ We can perform mathematical operations on a pointer.
→ 4 arithmetic operators that can be used on pointers
++, --, + and −.
→ Assume ptr contains address 1000. (i.e., ptr is an integer pointer. Assume integer takes 4 bytes).

ptr++ → will point to the next location 1004

ptr-- → will point to the previous location

→ Pointers may be compared by using relational operators such as ==, < and >.

Eg:- (1) main()
{
int n = 5; n → 5 (5000)
int *p;
p = &n; // 5000 p → 5000 (4320)
printf("Add of p = %p", p); // 4320
p = p+1; (or) p++;
printf("After Inc add of p = %p", p); // 5004
}

4 (a) Explain with syntax and example fseek(), rewind(), ftell().

ftell()

→ ftell takes a file pointer and returns a number of type long, that corresponds to the current position

→ This function is useful in saving the current position of a file, which can be used later in the program.

→ The general syntax is:

```
n = ftell (file_pointer);
```

n → relative offset (in bytes) of the current position. This means that n bytes have already been read (or written).

rewind()

→ rewind takes a file pointer and resets the position to the start of the file.

→ for example:

```
rewind (fp);
n = ftell (fp);
```

would assign 0 to n because the file position has been set to the start of the file by rewind.

→ This function helps us in reading a file more than once, without having to close and open the file

Note: ✱ The first byte in the file is named/numbered as 0, second as 1 and so on.
✱ whenever a file is opened for reading or writing, a rewind is done implicitly.

fseek()

→ fseek function is used to move the file position to a desired location within the file.

→ The general syntax is:

```
fseek (file-pointer, offset, position);
```

file-pointer → pointer to the file.
offset → a number or a variable of type long
position → an integer number.

→ The offset specifies the number of positions (bytes) to be moved from the location specified by position.

→ The position can take one of the following values:

Value	Meaning
0	Beginning of file
1	current position.
2	End of file.

→ The offset may be positive meaning move forwards, or negative meaning move backwards.

(b) Explain Pointer to Pointer with example.

Pointer - to - Pointer (Double Pointer / chain of pointers).

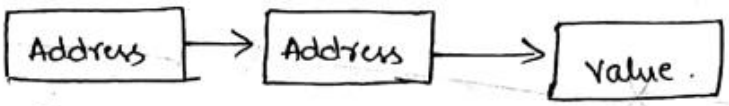
→ A pointer holds address of another variable of same type.

→ When a pointer stores/holds the address of another pointer then such type of pointer is known as pointer - to - pointer.

→ The syntax for declaring a pointer to pointer is:

```
data_type ** variable_name;
```

Eg:- `int **ptr;`



Eg:-

```
main()
{
    int var;
    int *ptr;
    int **pptr;

    var = 100;

    ptr = &var;

    pptr = &ptr;

    Print var    // 100

    Print *ptr   // 100

    Print **pptr // 100
}
```

5 What is dynamic memory allocation? Explain the different functions used for allocating and de-allocating memory with syntax and example.

The process of allocating memory at run-time.

malloc()

→ This function allocates size bytes and returns a pointer to the allocated memory.

→ The general syntax is:

```
void *malloc (size_t size);
```

(or)

```
ptr = (cast_type *) malloc (byte_size);
```

→ This reserves a block of memory of specified size and return a pointer of type void which can be casted into pointer of any type.

→ The memory is not initialized. If size is 0, then malloc() returns either NULL.

→ The general syntax can also be written as:

```
ptr = (data_type *) malloc (size);
```

where

ptr → pointer variable of type data_type.

data type → can be any basic type (int, char, float, double..) or user defined data type (structures).

size → no^r of bytes.

→ For example:

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

This allocates 400 bytes (Assume int takes 2 bytes) and ptr points to the address of first byte of memory.

→ Example:

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int n, i, *ptr;
```

```
printf("Enter the no^r of elements");
scanf("%d", &n);
ptr = (int*) malloc (sizeof(int) * n);

if (ptr == NULL)
{
    printf("Insufficient memory");
    return 1;
}

printf("Enter the elements");
for(i=0; i<n; i++)
    scanf("%d", ptr+i);

printf("The elements are");
for(i=0; i<n; i++)
    printf("%d", *(ptr+i));

return 0;
}
```

NOTE: malloc() doesn't initialize memory at execution/run time, so it has garbage value initially.

calloc()

→ This function allocates memory for an array of nmemb elements of size bytes each and returns a pointer to the allocated memory.

→ The general syntax is:

```
void * calloc(size_t nmemb, size_t size);
```

(or)

```
ptr = (cast_type *) calloc(n, element_size);
```

→ This function is used to allocate multiple blocks of memory.

→ calloc stands for contiguous allocation of multiple blocks and is mainly used to allocate memory for arrays.

→ The number of blocks is determined by the first parameter n.

→ The total no^r of bytes allocated is n*size and all bytes will be initialized to 0.

→ If n or size is 0, then calloc() returns NULL.

→ The general syntax can also be written as:

```
ptr = (data_type *) calloc (n, size);
```

ptr → pointer variable of type data_type.

data_type → can be any basic data type or user defined data type.

n → no^r of blocks to be allocated.

size → no^r of bytes in each block.

→ for example :

```
int *ptr;
ptr = (int *) calloc ( 5, sizeof (int));
```

This allocates 5 blocks of memory where each block is of 4 bytes (Assume int takes 4 bytes) each.

→ C program to find max of n numbers using dynamic arrays.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int n, i, j, *a;

    printf("Enter the no^r of elements");
    scanf("%d", &n);
```

```
a = (int *) calloc (n, sizeof(int));
if (a == NULL)
{
    printf ("Insufficient memory");
    return 1;
}
printf("Enter the elements");
for(i=0; i<n; i++)
    scanf("%d", &a[i]);

j=0;
for(i=1; i<n; i++)
{   if (a[i] > a[j])
        j = i;
}
printf("The biggest %d is found in pos %d",
        a[j], j+1);

free(a);
return 0;
}
```

realloc()

→ Before using this function, memory should have been allocated using malloc() or calloc().

→ Sometimes, the allocated memory may not be sufficient and we may require additional memory space.

→ Sometimes, the allocated memory may be much larger and we want to reduce the size of allocated memory.

→ In both the situations above, the size of the allocated memory

can be changed using realloc() and the process is called reallocation of memory. It is done as shown below.

★ realloc() changes the size of the block by extending or deleting the memory at the end of the block.

★ If the existing memory can be extended, ptr will not be changed.

★ If the memory cannot be extended, this function allocates a completely new block and copies the contents of existing memory block into new memory block and then deletes the old memory block.

→ The general syntax is:

```
void *realloc (void *ptr, size_t size);
```

(or)

```
ptr = (data_type *) realloc (ptr, size);
```

ptr → pointer to a block of previously allocated memory either using malloc() or calloc().

size → new size of the block.

→ The function returns the address of the first byte of allocated memory, if not NULL.

→ Example:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
    char *str;

    str = (char *) malloc (10);
    strcpy (str, "Information");
```

```
str = (char *) realloc (str, 30);
strcpy (str, "Information Science");
return 0;
}
```

free()

→ This function is used to de-allocate (or free) the allocated block of memory which is allocated using the functions calloc(), malloc() or realloc()

→ The general syntax is:

```
free (pointer);
```

→ It is important to deallocate the memory as it results in memory leak if not done.

→ Memory leak occurs when programmers create a memory in heap and forget to delete it.

6 What is preprocessor? Explain any 5 pre-processor directives with an example for each.

Before a C program is compiled in a compiler, source code is processed by a program called preprocessor. This process is called preprocessing.

→ #define statement is used for this.

→ The general syntax is (a min one blank space is must)

```
#define identifier string
```

If this statement is included in the program at the beginning, then the preprocessor replaces every occurence of the identifier in the source with string.

→ Used to define constants.

→ Examples:

```
#define   SIZE   100
#define   TRUE   1
#define   PI     3.
#define   CAPITAL  "Bangalore"
```

→ Note: All macros above are written in capitals. It is a general convention to identify them as symbolic constants.

→ A definition such as

```
#define X 5
```

will replace all occurances of X with 5. However, a macro inside a string does not get replaced.

→ Example:

```
#include <stdio.h>
#define M 5.
int main()
{
    int total;
    total = M * 100;
    printf("The value of M is %d", M);
    return (0);
}
```

File Inclusion

→ An external file containing functions or macro definitions can be included as a part of a program so that we need not rewrite those functions or macro definitions.

→ This is achieved using preprocessor directive #include

→ The general syntax is

#include "filename"

The preprocessor inserts / includes the entire contents of filename into the source code of the program.

→ When the filename is included within " " (double quotes) the search for the file is made first in the current directory and then in the standard directory.

Alternatively, this directive can take the form

#include <filename>

↳ file is searched only in standard directories

① #ifdef, #else, #endif.

→ This directive checks if/whether particular macro is defined or not. If it is defined, "if" clause statements are included in source file.

→ Otherwise, "else" clause statements are included in source file for compilation and execution.

Syntax :

```
#ifdef MACRONAME.
    Statement_block;
#endif
```

For example:

```
#include<stdio.h>
#define MAX 10

int main()
{
    #ifdef MAX.
        printf("Max is defined!!");
    #else
        printf("Max is not defined!!");
```

```
#endif
return (0);
}
```

Output: Max is defined !!

(2) #ifndef, #endif, #else.

→ This exactly acts as reverse as #idef directive. If particular macro is not defined, "if" clause statements are included in soura file.

→ Otherwise, "else" clause statements are included in soura file for compilation and execution.

Syntax:

```
#ifndef MACRONAME
    statement_block;
#endif
```

For Example:

```
#include <stdio.h>
#define MAX 100
int main()
{
    #ifndef MIN
    {   optional
        printf("Min is not defined!! Define it now");
        #define MIN 16.
    }
    #else
        printf("Min is already defined!!");
    #endif
    return(0);
}
```

Output: Min is not defined!! Define it now.

③ #if, #else, #endif

→ "if" clause statement is included in source file if given condition is true.

→ otherwise, "else" clause statement is included in source file for compilation and execution.

Syntax:

```
#if Expression
    Statement 1
    statement 2
    :
    statement n
#endif
```

For Example :

```
#include<stdio.h>
#define X 50
int main()
{
    #if (X==50)
        printf("This line will be included!!");
    #else
        printf("This line will not be included");
    #endif
    return(0);
}
```

Note : Expressions should be only a constant

Output : This line will be included!!

7(a) Write a C program to compute sum, mean and standard deviation for an array of real numbers using pointers.

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

int main()
{
        int n, i;
        double a [ 10 ], sum, mean, sd, total, var;

        // Read the number of integers.
        printf ( "\nEnter the value of n: " );
        scanf ( "%d", &n );

        // Read the integers.
        printf ( "\nEnter %d numbers: ", n );
        for ( i = 0; i < n ; i++ )
        {
                scanf ( "%lf", (a+i) );
        }
```

```
        //Compute sum.
        sum = 0;
        for ( i = 0 ; i < n ; i++ )
        {
                sum = sum + *(a+i);
        }
        printf ( "\nThe Sum is: %lf\n", sum );

        // Compute mean.
        mean = sum / n;
        printf ( "\nThe Mean is: %lf\n", mean );

        // Compute variance and standard deviation.
        total= 0;
        for ( i = 0; i < n ; i++)
        {
                total = total + pow ( (*(a+i)-mean), 2);
        }
        var = total / n;
        sd = sqrt ( var );
        printf ( "\nThe Standard Deviation is: %lf\n\n", sd );

        return 0;
}
```

(b) Write a short note on Queues and Trees.

Queues

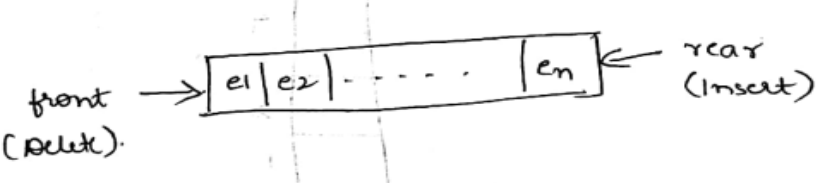
→ A queue is a data structure where elements are inserted from one end and elements are deleted from other end.

→ The end at which new elements are added / inserted is called rear end.

→ The end at which elements are deleted is called front end

→ It is also known as First In First Out (FIFO) data structure

→ The pictorial representation of a queue is



→ Types of queues
- Linear Queue (Normal / Ordinary Queue)
- Circular Queue
- Double ended queue (dequeue)
- Priority Queue.

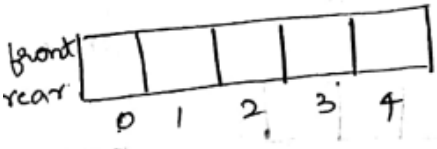
→ The various operations that can be performed on a queue are:

Insert → An element is insert from rear end
Delete → An element is deleted from front end.
Display → Display the contents of the queue.

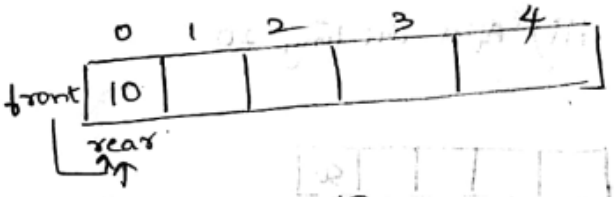
Insert Operation

→ Inserting an element into the queue from rear end.
→ Only one element is inserted into the queue from the rear end.
→ When the queue becomes full, it is not possible to insert any elements. Trying to insert an element even when queue is full results in overflow of Queue.

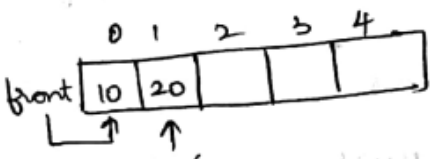
QUEUE_SIZE = 5.



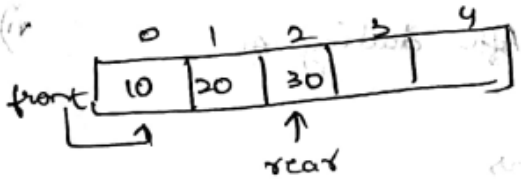
i) Empty Queue.



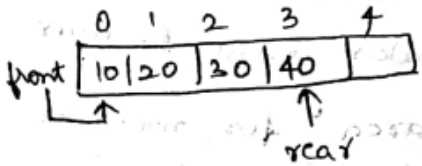
ii) Insert 10



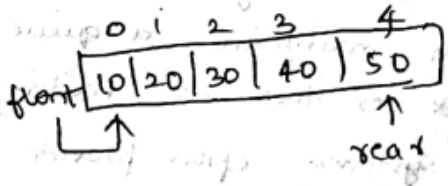
iii) Insert 20



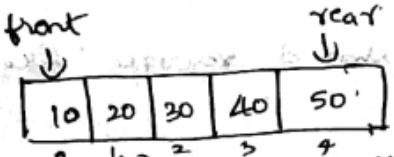
iv) Insert 30.



v) Insert 40



vi) Insert 50.

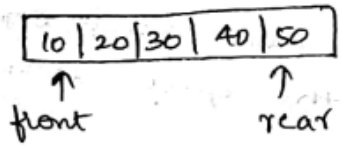


vii) Queue overflow while inserting 60

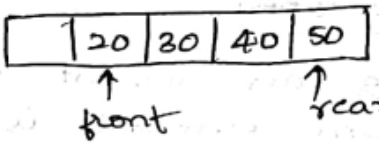
Delete Operation

→ Deleting an element from the queue from front end.

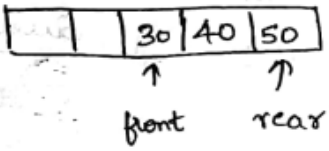
→ When queue becomes empty, it is not possible to delete elements. Trying to delete elements even when queue is empty, results in underflow of queue.



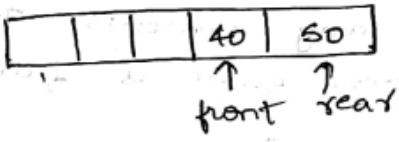
i) Queue Full



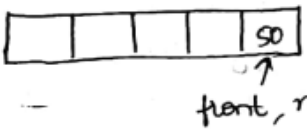
ii) After deleting 10



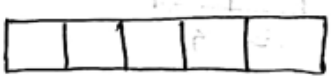
iii) After deleting 20



iv) After deleting 30



v) After deleting 40



vi) After deleting 50. Queue Empty.

Applications

→ When resource is shared among multiple users.
Eg:- CPU scheduling, Disk scheduling.

→ OS often maintains a queue of processes that are ready to execute or that are waiting for a particular event to occur.

→ Computer systems often provide a "holding area" for messages between two processes/programs/systems. (i.e., holding area is called buffer).

→ Serving requests on a single shared resource like printer, CPU task scheduling.

→ Handling of interrupts in real-time systems.

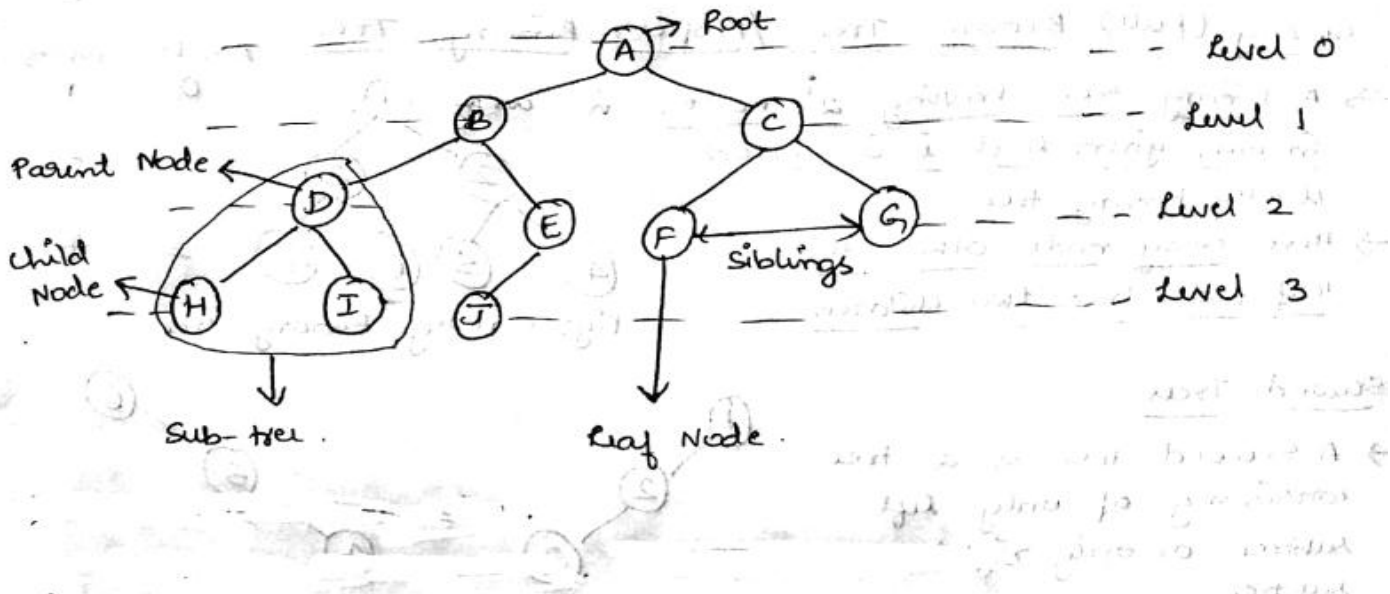
→ Queue of packets in data communication.

→ Queue of processes in OS.

Trees

→ A tree is a finite set of one or more nodes that show parent-child relation

→ Consider the following tree.



Binary Trees

→ A binary tree is a tree which has a finite set of nodes that is either empty or consist of a root and two subtrees called left subtree and right subtree.

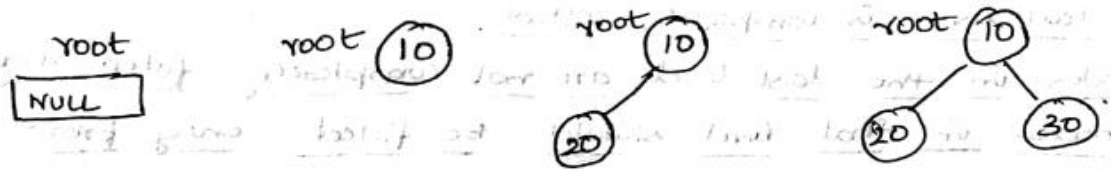


Fig: Binary Trees

Note: An empty tree is also a binary tree. Binary here means at most two i.e., zero, one or two subtrees are possible.

→ Types of binary trees.

1. Strictly Binary Tree (Full binary Tree)
2. Skewed tree.
3. Complete Binary Tree.
4. Binary search Tree.

Strictly (Full) Binary Tree / Proper Binary Tree

→ A binary tree having 2^i nodes in any given level i is called strictly binary tree.

→ Here every node, other than leaf node has two children

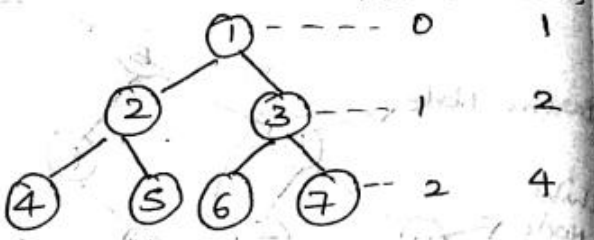
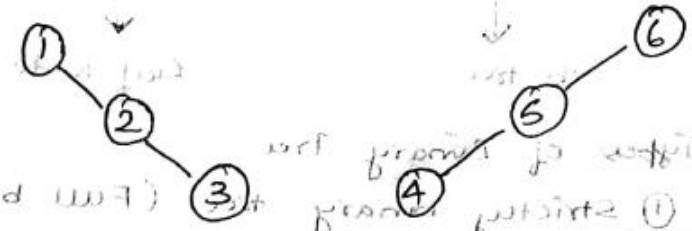


Fig: Strictly Binary Tree.

Skewed Tree

→ A skewed tree is a tree consisting of only left subtree or only right subtree.



Right skewed Tree. Left skewed Tree.

Complete Binary Tree.

→ A complete binary tree is a binary tree in which every level except the last level is completely filled.

→ If the nodes in the last level are not completely filled, then all the nodes in that level should be filled only from left to right

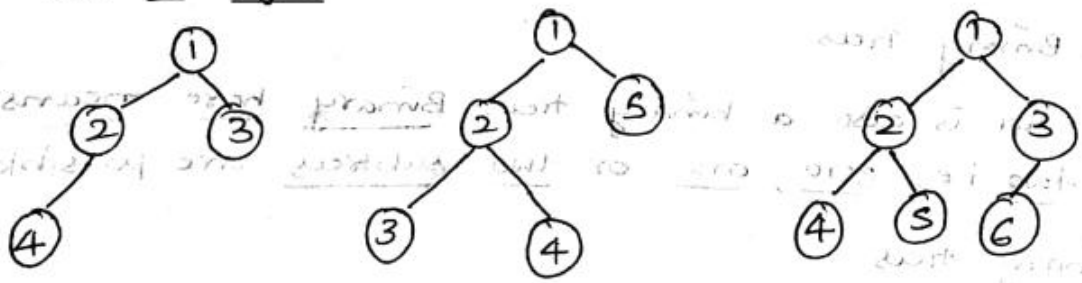


Fig: Complete Binary Tree.

Binary Search Tree.

→ A binary search tree is a binary tree in which for each node say X in the tree, elements in the left-subtree are less than X and elements in the right subtree are greater than X.

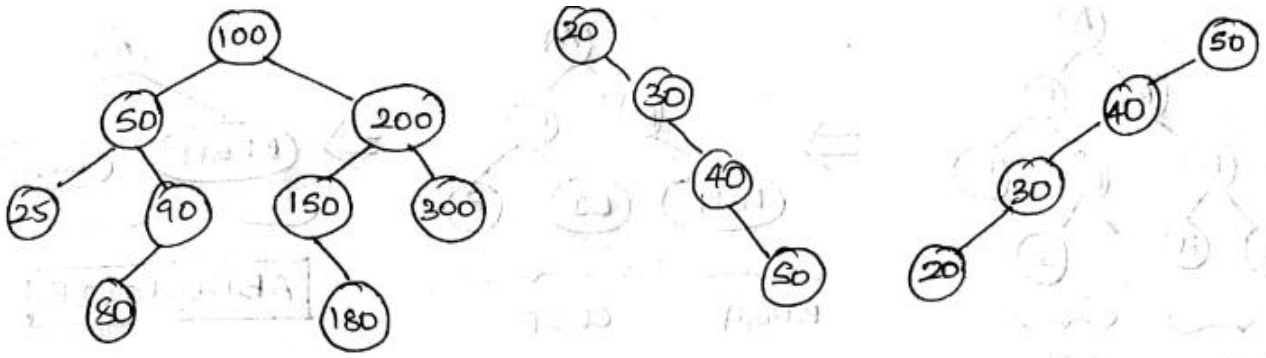


Fig: Binary search Trees.

→ The operations that can be performed on binary search trees are:

i) Insertion - insert an item into binary search tree.
ii) Searching - search for a specific item in the tree.
iii) Deletion - Deleting a node from a given tree.

Applications

→ Manipulate hierarchical data. (Eg: Files on a computer)
→ Make information easy to search (traversal)
→ Manipulate sorted list of data.
→ Routing algorithms (Eg: Spanning Tree Protocol).
→ Used as a workflow for composing digital images in visual effects.

8 (a) What is data structure? List all primitive and non-primitive data structures.

Data structure is a way of collecting and organising data in such a way that we can perform operations on these data in an effective way.

Primitive and non-primitive data types

Primitive data types

→ These are basic data types that are available in most of the programming languages

→ They directly operate upon the machine instructions.

→ Used to represent single values.

Eg:- Integer, Float, Double, Character.

Non-primitive data types

→ These are derived from primitive data types.

→ Used to store group of values.

Eg:- Arrays, Structures, Linked list, Stacks, Queues, Trees, Graph

Linear data structure

→ Elements are organised in some sequence or linearly.

Eg:- Arrays, Stacks, Queues, Linked list, structures

Non-linear data structure.

→ Elements are organised in some arbitrary function without any sequence.

Eg:- Trees and Graphs.

→ Uses memory efficiently.

→ Need not know the data items in prior.

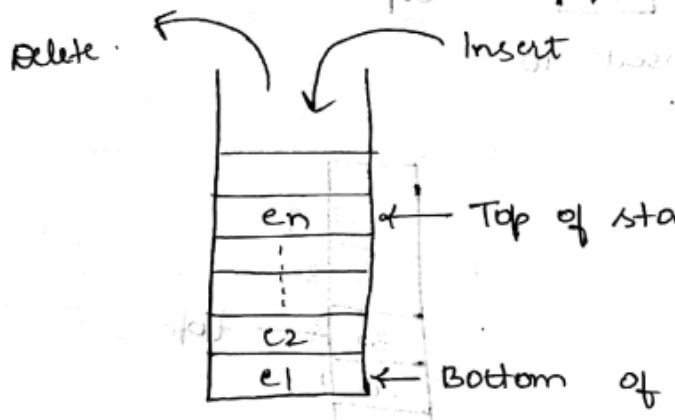
(b) Write a short note on Stacks and Linked Lists.

Stacks

→ A stack is a data structure in which elements are inserted at one end and deleted from the same end.

→ It is also known as Last In First Out (LIFO) data structure.

→ The pictorial representation of stack



→ The various operations that can be performed on a stack are :

Push → Inserting an element on the top of the stack.

Pop → Deleting an element from the top of the stack.

Display → Display contents of stack.

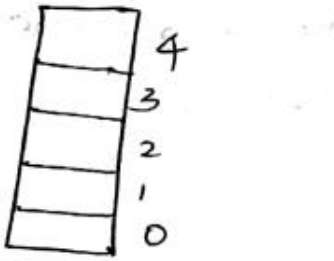
Push Operation

→ Inserting an element into the stack.

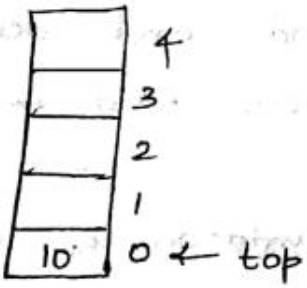
→ Only one item is inserted at a time and item has to be inserted only from top of the stack.

→ When the stack becomes full, it is not possible to insert any element. Trying to insert an element, even when stack is full results in overflow of stack.

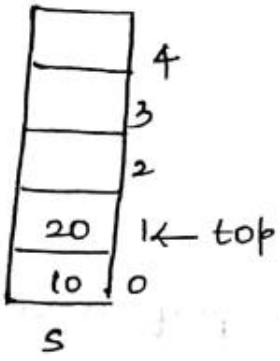
STACK_SIZE = 5



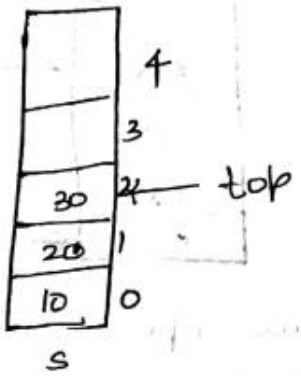
Empty stack.



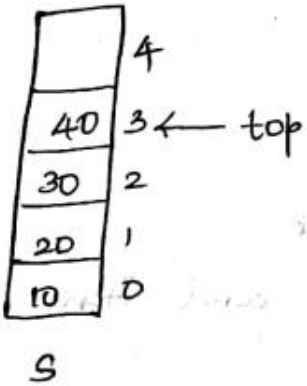
ii) Insert 10



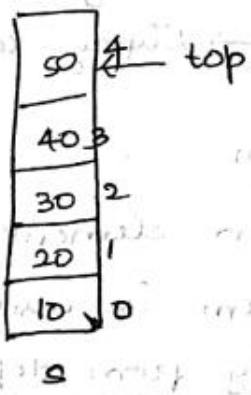
iii) Insert 20



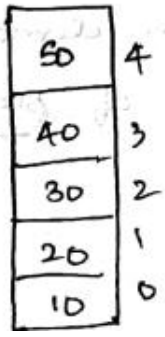
iV) Insert 30



V) Insert 40



vi) Insert 50



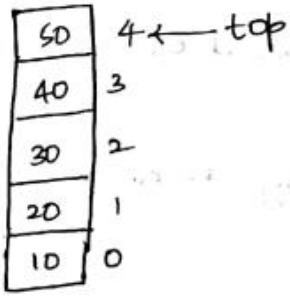
Insert 60, stack overflow.

Pop Operation

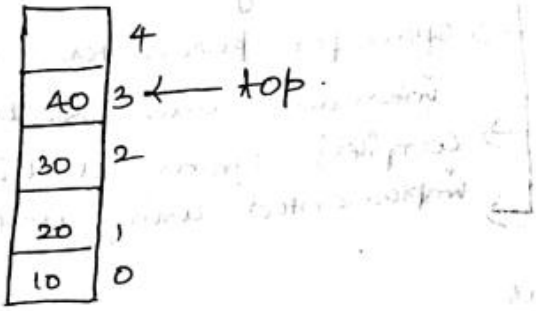
→ Deleting an element from the stack.

→ only one element can be deleted at a time and item has to be deleted only from top of the stack.

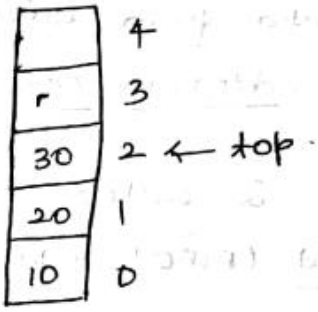
→ when elements are being deleted, there is a possibility of stack being empty. Trying to delete an element from an empty stack results in stack underflow.



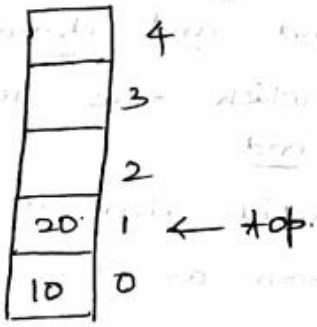
i) stack full



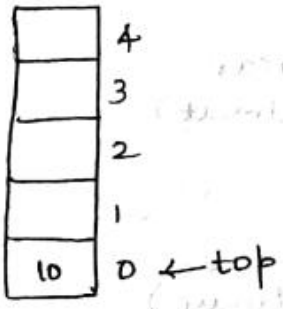
ii) After deleting 50



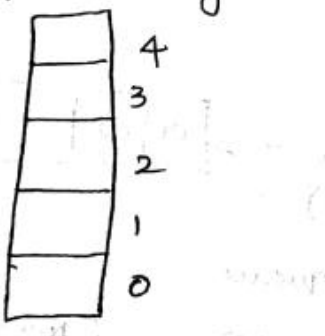
iii) After deleting 40



iv) After deleting 30



v) After deleting 20.



vi) After deleting 10
stack empty.

Applications

- → Conversion of expressions
- → Evaluation of expressions.
- → Recursion.
- → "undo" mechanism in text editors.
- → Backtracking – Maze problem.
- → Language processing
  - → Space for parameters and local variables is created internally using stacks.
  - → Compiler's syntax checking for matching braces is implemented using stack.

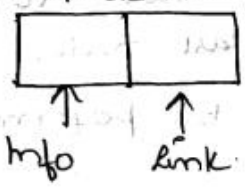
Linked Lists

→ A linked list is a data structure which is collection of zero or more nodes.

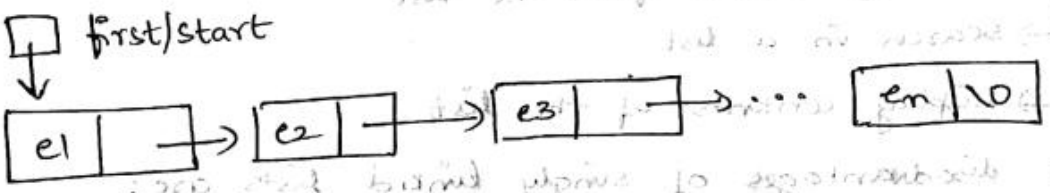
→ Each node in the list has two fields.

i) Info — used to store data or information.

ii) Link — contains address of the next node.



→ The pictorial representation of linked list is



→ Types of linked lists:

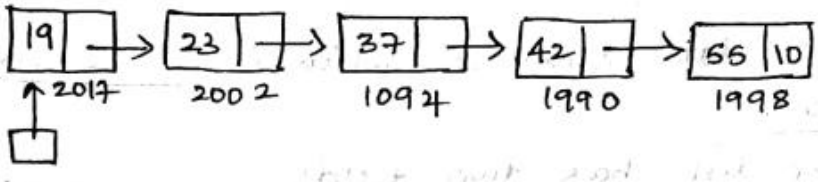
- → singly linked lists.
- → Doubly linked lists
- → circular singly linked lists
- → circular doubly linked lists.

Singly linked lists

→ A singly linked list is a collection of zero or more nodes where each node has two or more fields but only one link field which contains address of the next node. Each node

the list can be accessed using the link field which contains address of the next node.

→ For example : A singly linked list consisting of the items 19, 23, 37, 42, 55 is shown below :



Note : The variable first/start contains address of the first node. The link field of the last node contains 10 (NULL) indicating it is the last node.

→ The basic operations that can be performed on a linked list are :

- Inserting a node into the list
- Deleting a node from the list
- Search in a list
- Display contents of the list

→ The disadvantages of singly linked lists are :

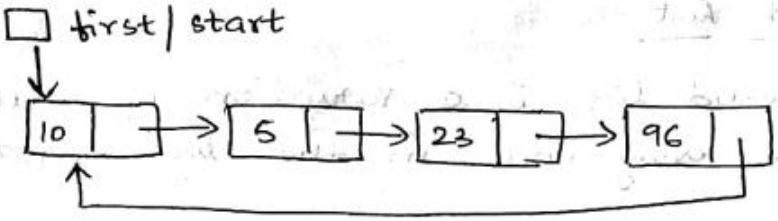
★ There is only one link field, hence traversing is done only in one direction.

★ To delete a particular node, address of the first node has to be provided.

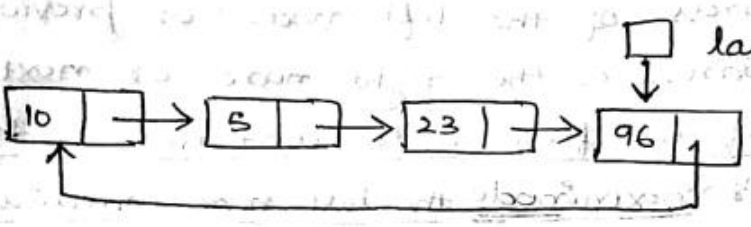
Circular Singly Linked List

→ A circular singly linked list is a singly linked list where the link field of the last node of the list contains address of the first node.

→ The pictorial representation of a circular singly linked list is as shown below :



(or)



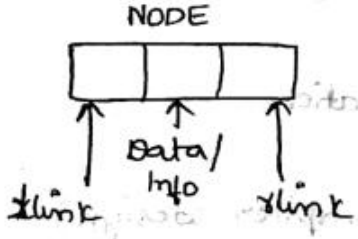
Doubly linked list

→ A doubly-linked list is a linear collection of nodes where each node is divided into three parts:

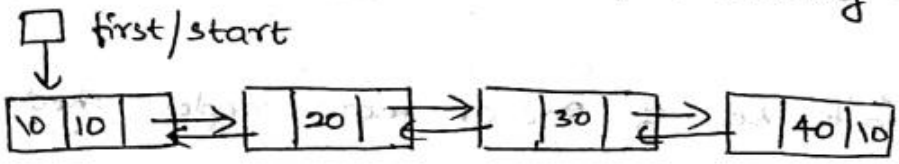
i) Info - used to store data or information.

ii) llink - contains address of the left node or previous node in the list.

iii) rlink - contains address of the right node or next node in the list.



→ The pictorial representation of a doubly linked list is:



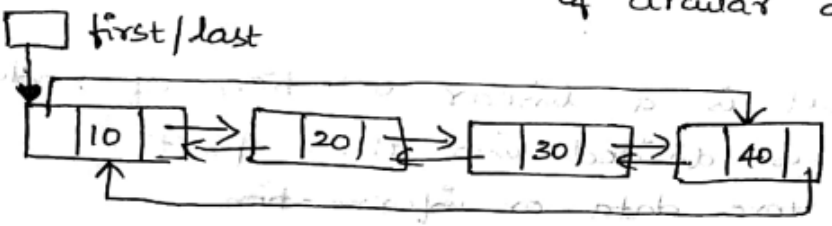
Circular Doubly Linked List

→ A circular doubly linked list is a variation of doubly linked list in which every node in the list has three fields:

i) Info – data/information is stored.
ii) llink – contains address of the left node or previous node
iii) rlink – contains address of the right node or next node

and the llink of the first node contains address of the last node whereas rlink of the last node contains address of the first node.

→ The pictorial representation of circular doubly linked list is



Applications

→ Used to implement stacks and queues.
→ Used to implement Graphs.
→ Implement Hash tables.
→ Evaluation of polynomials.
→ Useful for dynamic memory allocation.
→ Sparse matrices
→ In symbol table construction (compiler design).