Solution/ model answer to VTU question paper
3rd Sem-CSE/ISE - DSA – 18CS32
Dr. P. N. Singh, Professor(CSE), CMRIT, Bangalore

Module 1

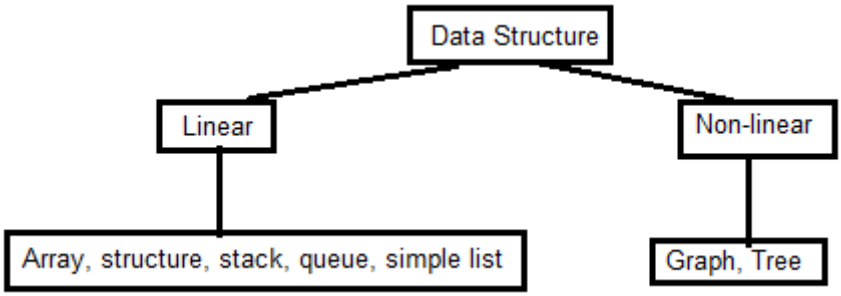
Q 1 a. What is data structure? What are various types of data structure? Explain. (5 marks)

Ans:

A structure represents name, type, size and format. A data structure is a represents to organize and store data in terms of name, type, size and & format in computers memory. Data may contain a single element or sometimes it may be a set of elements. Whether it is a single element or **multiple** elements but it must be organized in a particular way in the computer memory system.

General data structure types include the array, lists, vectors, the file, the record, the table, the tree, graphand so on. Any data structure is designed to organize data to suit a specific purpose so that it can be accessed and operated with appropriate ways.

Types of data structure:



Data Structure operations:

- Addition/Insertion: To add data at beginning or end or to insert data between two data
- Sorting: To sort data either in ascending or descending order to make the search operation faster
- Search: To search data within given collection by different ways
- Modification: To modify existing data with new data
- Deletion: To delete existing data

Q1b. What is a structure? How it is different from array? Explain different types of structure declarations with examples & give differences between Union & Structure. (10 Marks)

Ans:

Structure is a data structure whose individual elements can differ in type. It may contain integer elements, character elements, pointers, arrays and even other structures can also be included as elements within a structure.

Different from Array: Array is collection of similar data type but structure is composed of different data types. It represents a record with different fields. Structure of a student may have roll number of integer, name as string and marks as float.

struct is keyword to define a structure.

```
struct  tag  { type member1;
               type member2;
               type member3; ..... type member n;};
```

New structure type variables can be declared as follows:

struct tag var1, var2, var3, ... varn, var[10];

i) Array of structure:

Whole structure can be an element of the array. A student structure with members roll, name and marks:

```
struct student
{
    int roll;
    char name[30];
    int marks;
};
```

Now we can use this template to create array of 60 students for their individual roll, name and marks.

struct student s[60]; /* array of structure *

to access roll of student x : **s[x].roll**

ii) Array within structure:

A member of structure may be array. In above example name is also array of characters. We can create a structure of student with members roll, name & marks (array of integers) for 6 papers:

```
struct student
{
    int roll;
    char name[30];
    int marks[6]; /* array within structure */
}s[60];
```

To access the marks of student x in paper y: **s[x].marks[y]**

ii) Structure within Structure:

A structure may be defined within another structure or a structure variable may be declared as a member within another structure . Here date structure is defined within employee structure:

```
struct employee
{
    int eno;
    char name[30];
    long int salary;
    struct date   /*structure within structure */
    {
        int dd, mm, yy;
    }dob,doj;
}e[1000];
```

Or structure variable of date structure can be defined within employee if date structure is separately defined:

```
struct date
{
```

```
        int dd, mm, yy;
    };
struct employee
{
    int eno;
    char name[30];
    long int salary;
    struct date dob, doj; /* variable of date structure within employee*/
}e[1000];
```

To access day(dd) of date structure we will use member operator (.) dot operator in following way:

e[x].dob.dd

Difference between structure & union

Points	Structure	Union
1	Structure allocates storage space for all its members separately. Keyword: **struct**	Union allocates one common storage space for all its members. Union finds that which of its member needs high storage space over other members and allocates that much space. Keyword: **union**
2	Structure occupies larger memory space.	Union occupies lower memory space over structure.
3	We can access all members of structure at a time.	We can access only one member of union at a time.

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

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

Q1c. Define Pointers. How to declare and initialize pointers, explain with examples. (5 Marks)

Ans:

A pointer keeps address of data. A pointer is variable that represents the location (rather than the value) of a data item, such as a variable or an array element.

Pointers is an important feature of C and frequently used in C. They have a number of useful applications.

Pointer operator is asterisk (*).

To declare a pointer: *datatype *pointervariable;*

To initialize: pointervariable = &data;

& is address of operator.

Example1:

```
int n;
int *p;  /* here p is pointer variable & can be used to keep address */
p=&n;  /* here p keeps address of n */
```

Example2:

```
int a[5] = { 10,20.30,40,50};
int *p;
p = a;   /* It is equivalent to  p = &a[0];   Here it keeps address of first
element */
```

Or

Q2a. Explain dynamic memory allocation functions in detail. (6 Marks)

Ans:

Dynamic Memory allocation and de-allocation:
Memory can be allocated/ de-allocated at run-time as and when required. C has four in-built functions for the same with header file stdlib.h
malloc(), calloc() and realloc() to allocate and free() to de-allocate

malloc()

The name malloc stands for "memory allocation". The function malloc() reserves a block of memory of specified size in bytes and return a pointer of type void which can be casted into pointer of any form.
Syntax of malloc()
ptr=(cast-type*)malloc(byte-size);
example:
ptr=(int*)malloc(100*sizeof(int));

calloc()

The name calloc stands for "contiguous allocation". The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.
Syntax for calloc():
ptr=(cast-type*)calloc(n,element-size);
example:
ptr=(int*)calloc(25,sizeof(int));

realloc()

If the previously allocated memory is insufficient or more than sufficient. Then, you can change memory size previously allocated using realloc().
Syntax/example:
ptr=realloc(ptr, newsize);

free()

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

Q2b. Write the Knuth Morris Pratt pattern matching algorithm and apply the same to search the pattern 'abcdabcy' in the text 'abcxabcdabxabcdabcdabcy' (8 marks)

Ans:

Knuth-Morris-Pratt (KMP) string matching algorithm runs in O(m+n) time to find all occurrences of pattern P in S. **In KMP algorithm, a preprocessing is done in pattern string P and an array of length m is calculated.**

The basic idea behind KMP's algorithm is: whenever we detect a mismatch (after some matches), we already know some of the characters in the text of the next window. We take advantage of this information to avoid matching the characters that we know will anyway match.

Unlike Brute-Force/Naïve algorithm, where we slide the pattern by one and compare all characters at each shift, we use a value from lps[] to decide the next characters to be matched. The idea is to not match a character that we know will anyway match.

How to use lps[] to decide next positions (or to know a number of characters to be skipped)?

We start comparison of pat[j] with j = 0 with characters of current window of text.

We keep matching characters txt[i] and pat[j] and keep incrementing i and j while pat[j] and txt[i] keep **matching**.

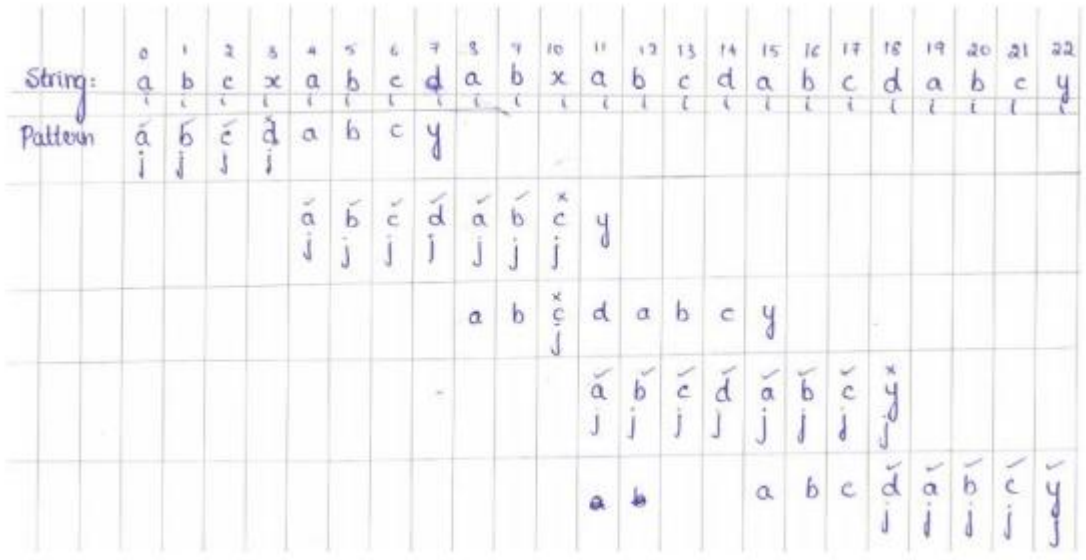
When we see a **mismatch**

We know that characters pat[0..j-1] match with txt[i-j…i-1] (Note that j starts with 0 and increment it only when there is a match).

We also know (from above definition) that lps[j-1] is count of characters of pat[0…j-1] that are both proper prefix and suffix.

From above two points, we can conclude that we do not need to match these lps[j-1] characters with txt[i-j…i-1] because we know that these characters will anyway match.

Demo for given pattern & string:



Q2c. Write a C program to (i) compare strings (ii) concatenate two strings. (6 marks)

Ans:

```
/* comparing & concatenating strings */
#include <stdio.h>
int compstr(char str1[], char str2[])
{
   int x,len1,len2;
   len1=0;
   while(str1[len1]!=NULL) len1++; /*length of string1 */
   len2=0;
```

```
   while(str2[len2]!=NULL) len2++; /*length of string2 */

   if(len1!=len2) return (0); /* if both have different length*/
   for(x=0;x<len1;x++)
     if(str1[x] != str2[x]) return (0); /* if any mismatch */

   return (1); /*if not returned 0 False yet */
}

void concat(char str1[], char str2[])
{
  int x,y;
  for(x=0;str1[x]!=NULL;x++)
        ; /* reaching to last character of string1 */
  for(y=0;str2[y]!=NULL;y++)
  {
     str1[x]=str2[y];
     x++;
  }
  str1[x]=NULL; /* last character after concatenation */
}

int main()
{
  char str1[50],str2[50];
  printf("Enter string 1:");
  gets(str1);
  printf("Enter string 2:");
  gets(str2);

  if(compstr(str1,str2))  printf("Both strings are identical\n");
  else printf("Both strings are not identical\n");

  concat(str1,str2);
  printf("concatenated string is %s\n",str1);
  return (0);
}
```

Module 2

Q3a. Define stack. Give the implementation of push, pop and display functions in detail.(7marks)

Ans:

A stack is a LIFO(Last in first out) data structure having one end open only to insert or delete items. The item inserted at last will be deleted at first.

A common model of a stack is a plate or coin stacker. Plates are "pushed" onto to the top and "popped" off the top.

A stack is generally implemented with only two basic principle operations

- Push : adds an item to a stack on “top”.
- Pop: extracts the most recently pushed item from the stack from “top”.

Procedure/Functions:

```
/*MAX is the size of stack & item is to be pushed incrementing
the top variable */

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

 /* function to pop the elements off the stack */
 int pop(int stack[])
 {
  int ret;
   if(top == -1)
   {
        printf("The stack is empty/underflow"  );
        return (NULL);
   }
   else
   {
       ret = stack[top];
       top = top-1;
       return (ret);
    }
  }
/* function to display stack elements */
 void display(int stack[])
 {
  int x;
   printf("The Stack:\n");
  for(x=top;x>=0;x--)
        printf(" %d\n", stack[x]);
 }
```

Q3b. Write the postfix from the following expression using stack:

(i) A $ B * C – D + E|F|(G+H) **(ii) A – B | (C * D $ E)** **(6marks)**

Ans:

These expression having $ symbol that can be treated as exponentiation having greater precedence.

Single vertical bar is understood as Bitwise OR .

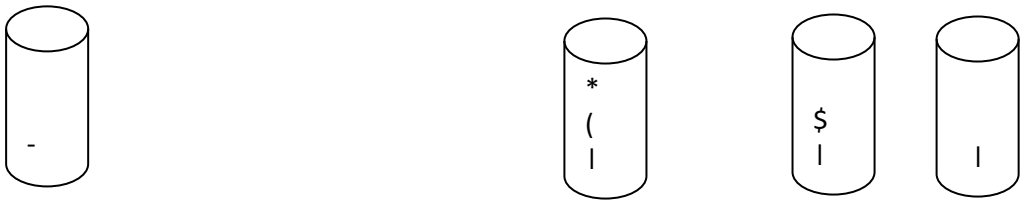
Converting to postfix using stack will require one operator stack to keep operator and one postfix string to add operands & operator (when another operator comes)
(i) A $ B * C – D + E|F|(G+H)

Postfix String	Operator Stack	Remarks
A	$	
A B $	*	Another operator comes * so, $ is added to postfix
AB $ C *	-	Another operator comes – so, * is appended to postfix
AB$C*D-		Another operator comes + so, – is appended to postfix expression
According to precedence E|F|(G+H) is converted first to E F | G H + | and added in postfix		
AB$C*D-EF|GH+|	+	Finally + is added at last
AB$C*D-EF|GH+|+		

Ans: AB$C*D-EF|GH+|+

(ii) **A – B | (C * D $ E)**

Operator Stack & postfix string is created. **Converting to postfix using stack will require one operator stack to keep operator and one postfix string to add operands & operator (when another operator comes)**



Postfix String

A	**B**	**-**	**C**	**D**	*****	**E**	**$**	**|**	

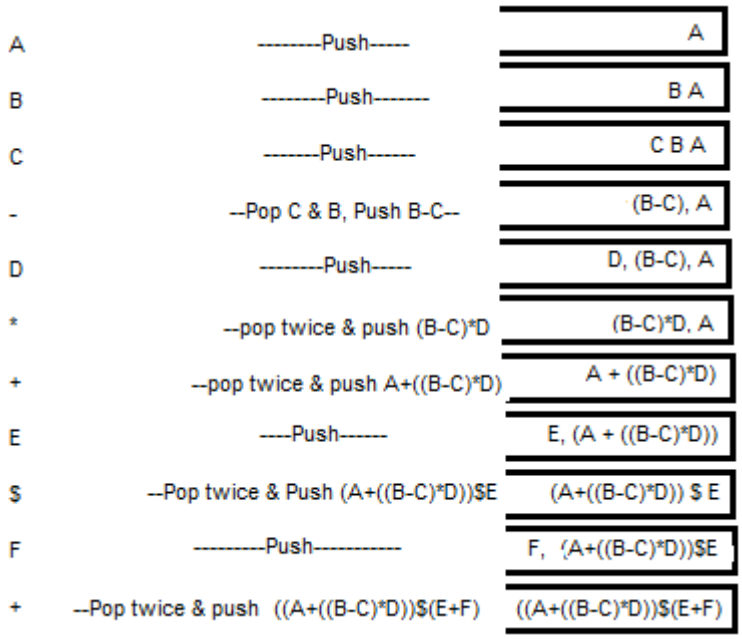
So, Postfix notation will be: A B – C D * E $ I

Q3c. Write an algorithm to evaluate a postfix expression and apply the same for the given postfix expression ABC-D*+E$F+ (7 Marks)

Ans:
Algorithm to evaluate postfix expression:

i. Read postfix expression from left to right (Taking care of precedence & Parentheses)
    a. If operand is encountered push it onto stack
    b. If operator is encountered, pop two operands (a= top element; b= next to top element)
    c. Evaluate b operator a and push onto stack
ii. Set final result to pop

Evaluating postfix expression **ABC-D*+E$F+**



Or

Q4a. Define recursion. Write a recursive function for the following:

(i) Factorial of a number (ii) Tower of Hanoi (5marks)

Ans:

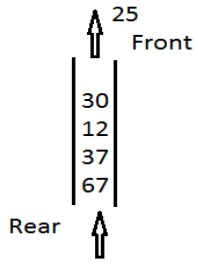
```
/i. *Factorial of a number */
long int fact(int n)
{
      if(n == 0 || n == 1)
            return (1);
      else
            return (n*fact(n-1));
}

/ii. *Tower of Hanoi */
#include<stdio.h>
void toh(int disks, charl[], char r[], char m[])
   /*l="Left", r="right" & m = "middle" */
{
       if(disks>0)
      {
            toh(disks-1, l, m, r);
            printf("Move disk %d from %s to %s\n",disks, l,r);
            toh(disks-1, m ,r , l);
      }
}

/* the function is called from main function as
            toh(disks,"left","right","middle");
*/
```

Q4b. What is the advantage of circular queue over ordinary queue? Write a C program to simulate the working of circular queue of integers using array. Provide the following operations:

(i) Insert (ii) Delete (iii) Display (8marks)

Ans:

Queue is FIFO (first in first out data structure). Items are processed and deleted according to their sequence of arrival in the queue. "Items are inserted from rear and deleted from front end.

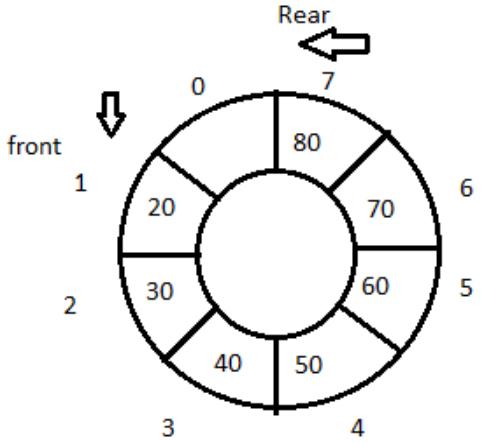


A linear queue of size 5

Disadvantages of linear queue:

- Items cannot be inserted from front end even the space is available
- Items cannot be deleted from rear or middle

This can be solved using a circular queue where front and rear are adjacent when queue is full. If any item is deleted then variable rear can be used to insert item to utilize it.



A circular Queue of size 8 (0 to 7)

One item is deleted from front end and queue[0] is vacant. Now front = 1 and rear =7 so by rear variable it can be occupied

```
#define MAX 10
int front=-1, rear=-1;

/*. Function to insert in circular queue */
void insertq ( int arr[], int item )
{
 if (( rear == MAX - 1 && front == 0)||( rear + 1 == front ))
 {
     puts("\nCircular Queue is full" );return ;
 }
 rear=(rear+1)%MAX;     arr[rear] = item ;
 if ( front == -1 )
       front = 0 ; /* when queue has been inserted an item */
}
```

```
/*ii. Function to delete in circular queue */
int deleteq(int arr[] )
{
 int data ;
 if ( front == -1 )
  {
       puts("\nQueue is empty") ;
       return(NULL);
  }
 data = arr[front] ; arr[front] = 0 ;
 if ( front == rear )
      front = rear = -1;    /* when all elements are deleted */
 else
      front=(front+1)%MAX;
 return (data) ;
}
```

```
/*iii. Function to display items from circular queue */
void display(int cq[])
{
    int x;
       printf("The items from circular queue:\n");
       for(x=0;x<MAX;x++)
            printf("%5d",cq[x]);
       printf("\n");
}
```

Q5b. Write a note of Dequeue & Priority Queue **(5marks)**

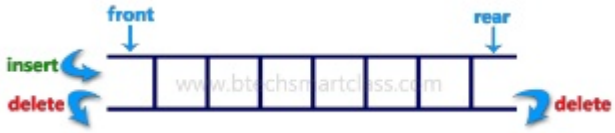
Ans:

A double-ended queue, or **deque**, supports insertion and deletion from the front and rear

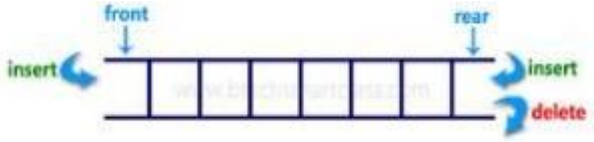
The possible operation performed on deque:

- Add an element at the rear end
- Add an element at the front end
- Delete an element from the front end
- Delete an element from the rear end

- An input-restricted deque is one where deletion can be made from both ends, but insertion can be made at one end only.



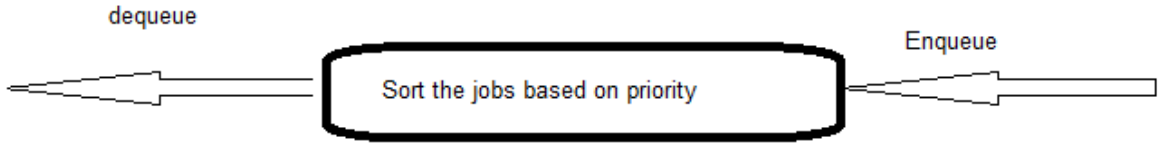
- An output-restricted deque is one where insertion can be made at both ends, but deletion can be made from one end only



Priority Queue:

A Priority Queue is a queue but different from a normal queue, because instead of being a "first-in-first-out", values come out in order by priority. It is an abstract data type that captures the idea of a container whose elements have "priorities" attached to them.

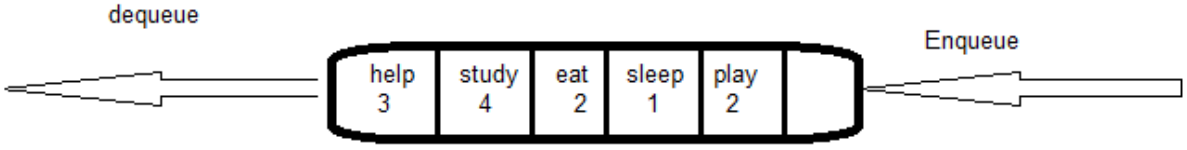
Jobs are rescheduled in the queue based on their priority number. In general a job least priority number will have highest priority. A job with higher priority is processed/de-queued before a job with lower priority.
Jobs/Items with the similar priority will be scheduled on FCFS (First Come First Serve) basis.



Priority Queue

Abstract of Priority queue:

```
struct pq
{
        char jname[20];
        int priority
}jobs[5];

jobs[0].jname="help";
jobs[0].priority=3;
jobs[1].jname="study";
jobs[1].priority=3;
jobs[2].jname="eat";
jobs[2].priority=2;
jobs[3].jname="sleep";
jobs[3].priority=1;
jobs[4].jname="play";
jobs[4].priority=2;
```



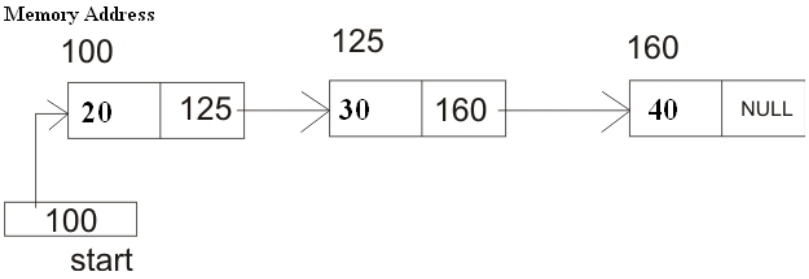
Before processing job/items are rescheduled based on priority:

Module 3

Q5a. What is a linked list? Explain different types of linked lists with neat diagram (7marks)

Ans:

A linked list is a dynamic data structure where a node keeps item and address of next node. Sometimes it is referred as self-referential structure. It can be considered as a list whose order is given by links from one item to the next. Each node has at least two members, one of which points to the next item or node in the list! These are defined as Single Linked Lists because they only point to the next item, and not the previous.

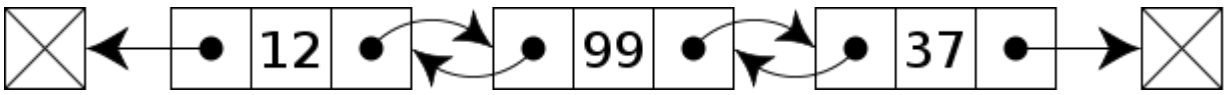


Linked list

Types of Linked List:

i. Singly linked list
ii. Doubly Linked list
iii. Circular linked list (singly and doubly)

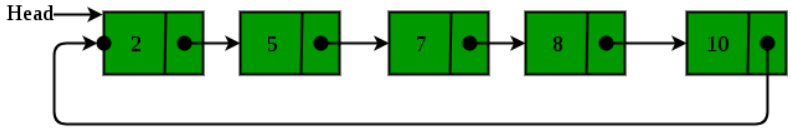
Doubly Linked List: a doubly-linked list is a linked list (linked data strcture) that consists of a set of set of data, each having two special links that contain address of the previous and to the next node in the sequence.



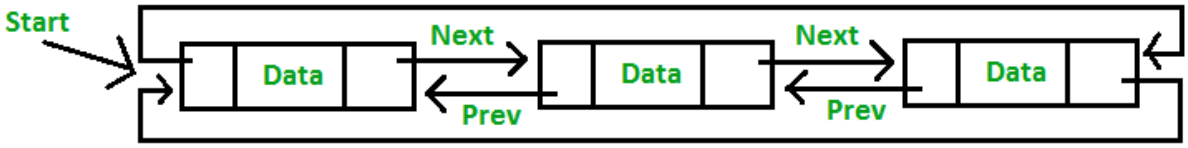
A doubly-linked list whose nodes contain three fields: an integer value, the link to the next node, and the link to the previous node.

The two links allow walking along the list in either direction with equal ease. Compared to a singly-linked list, modifying a doubly-linked list usually requires changing more pointers, but is sometimes simpler because there is no need to keep track of the address of the previous node

Circular Linked List: In Circular Linked Lists last node points to first. If it is Doubly circular linked list then prepvious pointer of the first node will point to the last node and next pointer of the last node will point to first node.



Circular (singly) Linked list



Circular Doubly Linked List

Q5b. Write a C function to insert a node at front and delete a node from rear end in a circular linked list. (8mars)

Ans:

/* C Function to insert a node at front in a circular linked list: */

```
struct Node *addfront(struct Node *last, int data)
{
  if (last == NULL)
     return addToEmpty(last, data);
  /* Creating a node dynamically. */
  struct Node *temp = (struct Node *)malloc(sizeof(struct Node));
  temp -> data = data;
  /* Now Adjusting the links */
  temp -> next = last -> next;
  last -> next = temp;
  return last;
}
```

/* C Function to delete a node from rear in a circular linked list: */

```
struct Node *addEnd(struct Node *last, int data)
{
    if (last == NULL)
        return addToEmpty(last, data);
    struct Node *temp = (struct Node *)malloc(sizeof(struct Node));
    temp -> data = data;
    temp -> next = last -> next;
    last -> next = temp;
    last = temp;
    return last;
}
```

Q5c. Write C function for the concatenation of two doubly linked lists. (5marks)

Ans:

/* function to concatenate two doubly linked lists even and odd

```
void concatenate()
{
   struct node *link;
   list = even;
   link = list;
   while(link->next!= NULL) {
      link = link->next; /* traversing to last of even DLL */
   }
   link->next = odd;
   odd->prev = link;
   /* assign link_last to last node of new list */
   while(link->next!= NULL) {
      link = link->next;
   }
   list_last = link;
}
```

Or

Q6a. Describe the doubly linked list with advantages and disadvantages. Write a C function to delete a node from a circular doubly linked list with the header node. (8marks)

Ans:

i. **Advantages of DLL over SLL**

A DLL can be traversed in both forward and backward direction.

ii. The delete operation in DLL is more efficient if pointer to the node to be deleted is given.

iii. In DLL, to delete a node, pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, we can get the previous node using previous pointer.

Disadvantages of DLL over SLL:

i. Every node of DLL Require extra space for an previous pointer.

ii. All operations require an extra pointer previous to be maintained. For example, in insertion, we need to modify previous pointers together with next pointers.

```
/* Function to delete a node from a circular doubly linked list */
   void deleteDLL(int position)
      {
       if(head==NULL)  return;
       if(position==1)
       {
           delete_head();
           return;
       }
       node *current = head;
       node *temp;
       int count = 0;
       do
       {
           count++;
           temp = current;
           current = current->next;
       }   while(current->next != head && count<position-1);
       if(count==position-1)
       {
           if(current==tail)
           {
               delete_tail();
               return;
           }
           temp->next = current->next;
           current->next->previous = temp;
           free(current);
```

```
            return;
        }
        printf("Position (%d) does not exist!\n", position);
    }
```

Q6b. For the given sparse matrix, give the diagramatic linked representation. (4marks)

$$A = \begin{pmatrix} 0 & 1 & 2 \\ 3 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Ans:

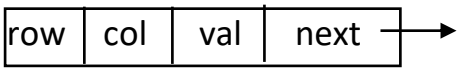
A sparse matrix has relative number of elements 0. Representing a sparse matrix using 2-D array takes substantial amount of space with no use.

A sparse matrix can be represented by triplets with maximum column of 3 and each row will have corresponding row number, column number of non-zero elements and non-zero element itself.

Those triples can also be represented by a singly linked list having members row, col, val (non-zero) and address of next node:

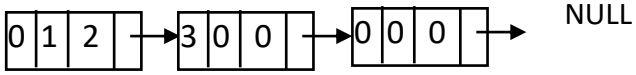
Structure definition:

```
struct spmlist
{
        int row,col,val;
        struct spmlist *next;
};
```



Triplets in given sparse matrix: 0 1 2, 3 0 0 & 0 0 0

Now linked representation



Module 4

Q6c. Write a C function to add two polynomials represented as circular list with header node. (8marks)

Ans:

```
NODE poly_add(NODE head1, NODE head2, NODE head3)
{
            NODE a,b;
            int coeff;
            a = head1->link;
            b = head2->link;
            while(a != head1 && b != head2)
            {
                        if(a->expon == b->expon)
                        {
                              coeff = a->coeff + b->coeff;
```

```
                        if(coeff != 0)
                                head3 = attach(coeff, a->expon, head3);
                                a = a->link;
                                b = b->link;
                        }
                        else if(a->expon > b->expon)
                        {
                                head3 = attach(a->coeff, a->expon, head3);
                                a = a->link;
                        }
                        else
                        {
                                head3 = attach(b->coeff, b->expon, head3);
                                b = b->link;
                        }
                }
                while(a != head1)
                {
                        head3 = attach(a->coeff, a->expon, head3);
                        a = a->link;
                }
                while(b != head2)
                {
                        head3 = attach(b->coeff, b->expon, head3);
                        b = b->link;
                }
                return head3;
}
```

Module 4

Q7a. What is a tree? With suitable example, define:

(i) Binary Tree (ii) Level of Binary tree (iii) Complete Binary Tree (iv) Degree of the tree (9marks)

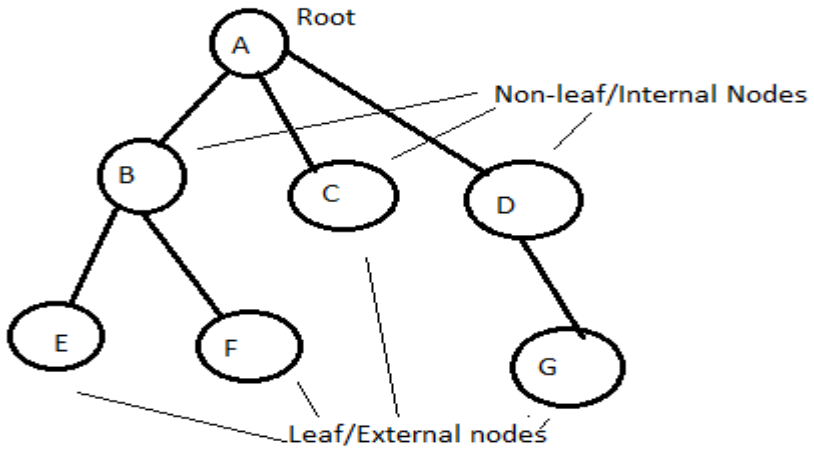
Ans:

A tree is a non-linear data structure that simulates a hierarchical tree structure stating with a root node and connected with subtree or children nodes.

Contrary to a physical tree, the root is usually depicted at the top of the tree structure and the leaves (leaf nodes) are depicted at the bottom.

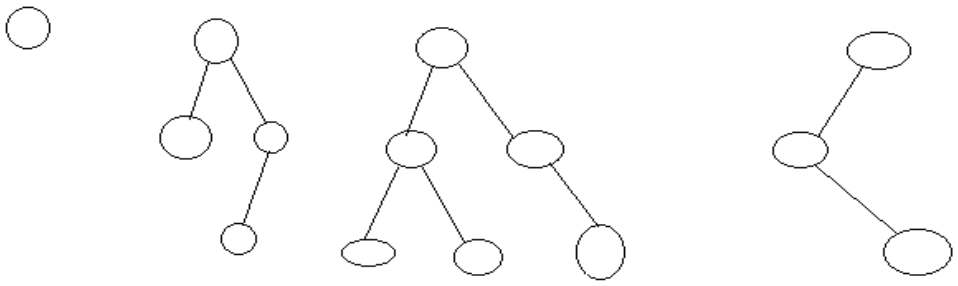
Leaf/External nodes will not have branch/child. Non-leaf/Internal nodes will have branch/child. Internal nodes are also known as parent node to their children node.

Children of same parent are known as siblings. In given diagram B,C & D are siblings. E and F are also siblings.



i. **Binary Tree:**

A Binary tree is a tree where a node may have 0, 1 or 2 (maximum) children (called left and right child). All sub-trees of Binary tree are also Binary tree.



Binary Trees

ii. **Level of a Binary Tree:**

Level (of a node): The number of parent nodes corresponding to a given a node of the tree. For example, the root, having no parent, is at level 0

Level of Binary Tree: Level of a binary tree start from root node with value 0 Everytime we move from top side of the tree towards the bottom side we increase the level by one.

Note- "Top down manner " means here traversal from the root node to one of its child node repeatedly

iii. **Complete Binary Tree:**

It is also termed as almost complete binary tree where nodes are added level by level & left to right. Only the last level few nodes may be added (if not full) from left to right.)

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

iv. **Degree of the tree:** Basically The degree of the tree is the total number of it's children i-e the total number nodes that originate from it. The leaf of the tree doesnot have any child so its degree is zero. Finally **Degree of a tree** is the highest degree of a node among all the nodes in the tree.

Q7b. Write the C routines to traverse the tree using (i) pre-order traversal (ii) post-order traversal (6marks)

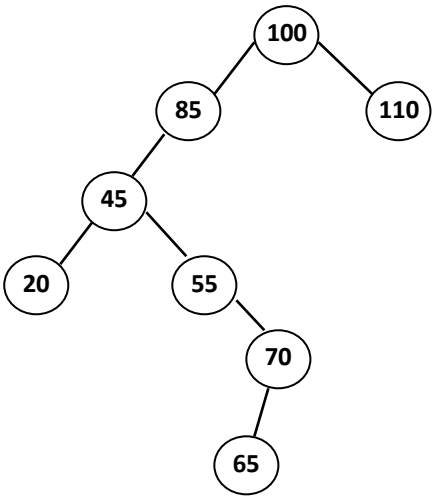
Ans:

```
/*Recursive function of Pre-order Traversal*/
void preorder(TREE *bintree)
  {
   if(tree)    {
               printf(" %s",bintree->str);
               preorder(bintree->left);
               preorder(bintree->right);
    }
  }
/* Recursive function of post-order traversal */
void postorder(TREE *bintree)
  {
   if(tree)    {
               postorder(bintree->left);
               postorder(bintree->right);
               printf(" %s",bintree->str);
    }
  }
```

Q7c. For the given data, draw a Binary search tree and show the array and linked representation of the same: 100, 85, 45, 55, 110, 20, 70, 65 (5marks)

Ans:

Binary Search Tree for given data:



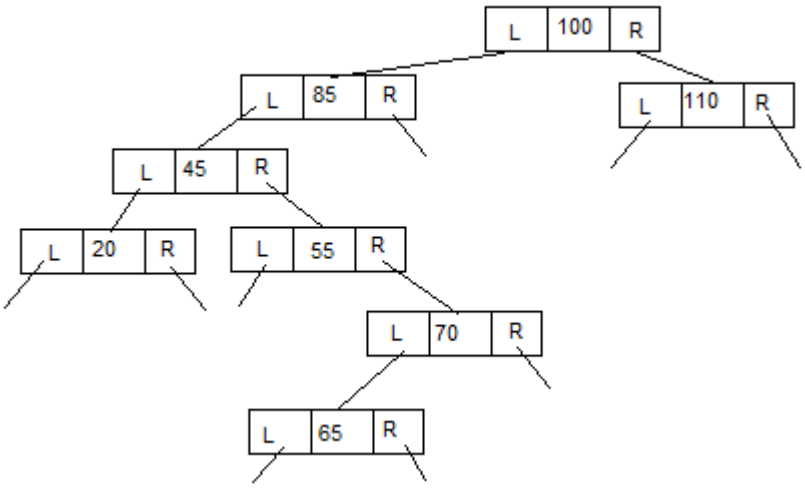
Array Representation:

Children of n^{th} node will be at $2n^{th}$ and $(2n+1)^{th}$ node

100 | 85 | 110 | 45 | -|-|-|20|55|- |- |- |-|- |- |- |- |- |70 |- |- |- | - |- | - | - | -| -| -| - | -|-|- |- |- |- |65|

1 2 3 4 5 6 7 8 9 10111213141516171819 202122 232425 26 27282930 3132333435363738

Linked Representation: All nodes structure L (left) pointer, data & R (right pointer)



Or

Q8a. What is the advantages of the threaded Binary tee over binary tree? Explain the construction of threaded binary tree for 10, 20, 30, 40 and 50 (7marks)

Ans:

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

Advantages:

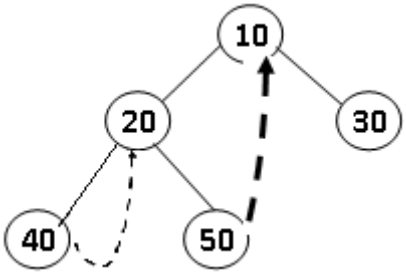
- Threaded trees make it possible to perform inorder-traversal without the use of stack or recursion.
- The threads make it possible to back-up to higher levels.

Rules for threaded Binary Tree

Level-order scheme visit the root first, then the root's left child, followed by the root's right child. All the node at a level are visited before moving down to another level.

Two methods/rules:

i. Use of parent field to each node.
ii. Use of two bits per node to represents binary trees as threaded binary trees.



Q8b. Define Expression tree for a given tree in Fig8(b) traverse the tree using in-order, pre-order & post-order traversal (7 marks)

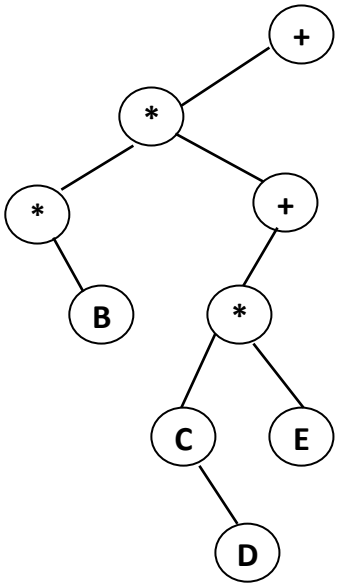


Fig 8(b)

Ans:

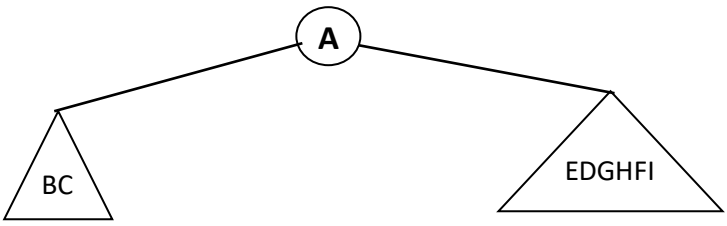
Expression tree is a binary tree in which each internal node corresponds to operator and each leaf node corresponds to operand.

Surprisingly in given tree operators are more than operands(B,C,D &E). I presume A is missing & A should be leaf node. However traversals are given here as per given tree.

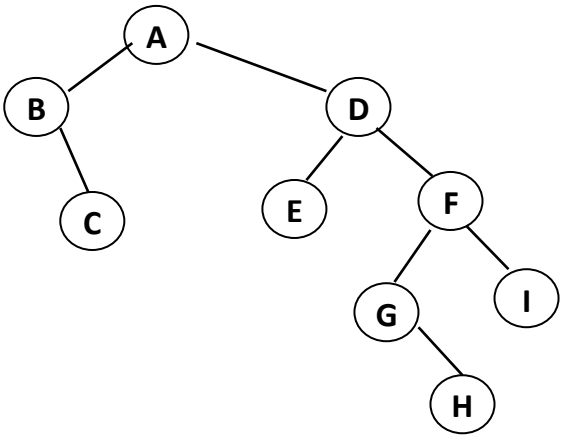
In-order Traversal: *** B * C D * E + +**
Pre-order Traversal: **+ * * B + * C D E**
Post-order Traversal: **B * D C E * + * +**

Q8c. Construct a Binary Search Tree by using the following in-order and pre-order traversal. **(6marks)**

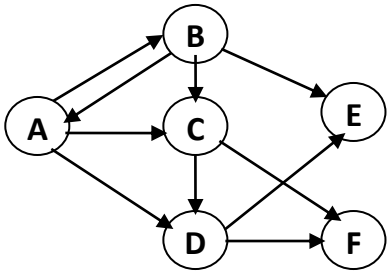
In-order: BCAEDGHFI
Pre-order: ABCDEFGHI
Ans:
Starting with A of pre-order (where root is visited first: Root-Left-Right) finding A in In-order which shows BC are left to A and EDGHFI are right to A (Left-Root-Right). So this way



This way repeating the same logic:
Ans:



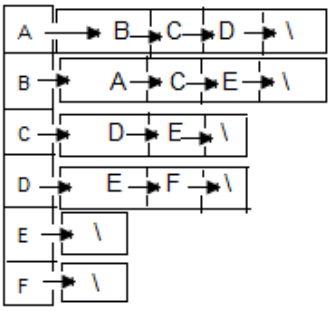
Q9a. Define Graph. For the given graph, show the adjacency matrix and adjacency list representation of the graph.[Ref. Fig Q9A] **(5marks)**



Ans:
Adjacency matrix of given directed (1 for path going to adjacent, 0 for no path going)

	A	B	C	D	E	F
A	0	1	1	1	0	0
B	1	0	1	0	1	0
C	0	0	0	1	0	1
D	0	0	0	0	1	1
E	0	0	0	0	0	0
F	0	0	0	0	0	0

Adjacency List Presentation:



Q9b. What are the methods used for traversing a graph? Explain any one with example and write C function for the same. (8marks)

Ans:

Graph traversal means visiting every vertex and edge exactly once in a well-defined order. The most common way of tracking vertices is to mark them. In general there are 2 ways to traverse graphs.

i. Breadth First Search (BFS)

BFS is the most commonly used approach.

We should start traversing from a selected node (source or starting node) and traverse the graph layer-wise thus exploring the neighbour nodes (nodes which are directly connected to source node). We must then move towards the next-level neighbour nodes. **For BFS Queue data structure is used.**

As the name BFS suggests, we are required to traverse the graph breadth-wise as follows:

A. First move horizontally and visit all the nodes of the current layer
B. Move to the next layer

ii. Depth First Search (DFS)

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

Backtrack means that when we are moving forward and there are no more nodes along the current path, we move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected. This recursive nature of DFS can be implemented using stacks. The basic idea is as follows:

- Pick a starting node and push all its adjacent nodes into a stack.
- Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.
- Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked.

This will prevent from visiting the same node more than once.

Here program of DFS is given using recursion

```
#include<stdio.h>
int G[10][10],visited[10]={0},n;
/*n is number of vertices and graph is stored in array G[10][10] */
void DFS(int i)
{
    int j;
    printf("\n%d",i);
    visited[i]=1;
    for(j=0;j<n;j++)
       if(!visited[j]&&G[i][j]==1)
            DFS(j);
}
```

```
void main()
{
    int i,j;
    printf("Enter number of vertices:");     scanf("%d",&n);
    //read the adjecency matrix
    printf("\nEnter adjecency matrix of the graph:\n");
    for(i=0;i<n;i++)
       for(j=0;j<n;j++)
            scanf("%d",&G[i][j]);
    DFS(0);
}
```

Q9c. Write a C function for insertion sort. Sort the following list using insertion sort. (7marks)
50, 30, 10, 70, 40, 20, 60

Ans:

/Function for insertion sort */

```
void inssort(int a[], int size)
{
  int x,y,picked;
  for(x=1;x<size;x++)/* loop starts from 2nd element */
  {
     picked = a[x];
     y=x;
     while(a[y-1] > picked && y > 0)
     {
       a[y]=a[y-1]; /* shifting */
       y--;
     }
     a[y]=picked; /* inserting */
   }
}
```

Sorting:

50, 30, 10, 70, 40, 20, 60
50

30 is "picked". 50 is greater than 30 so, 50 is shifted. There is no more to compare so, 30 is inserted at first

30 50 10 70 40 20 60
30 50

10 is picked, 50 is greater so shifted, 30 is greater so shited. There is no more to compare so, 10 is inserted at first.

10 30 50 70 40 20 60

70 is picked, 50 is less than 70 so inserted at same position

10 30 50 70 40 20 60
50 70

40 is picked, 70 is greater so shifted, 50 is greater so shited, 30 is less so, 40 is inserted

10 30 40 50 70 20 60
30 40 50 70

20 is picked. 70. 50. 40 & 30 have been shifted. 10 is less than 20 so, 20 is inserted.

10 20 30 40 50 70 60
70

60 is picked. 70 is greater therefore shifted. 50 is less than 60 so, 60 is inserted.

10 20 30 40 50 60 70

Now the array is sorted.

Or

Q10a. What is collision? What are the methods to resolves collision? Explain linear probing with an example. (7marks)

Ans:

Hashing is a technique which is designed to locate/calculate/map a key value to be stored/accessed. Hashing technique use a special function called the Hash function which is used to map a given value with a particular key for faster access of elements.

Popular hash function is key MOD size.

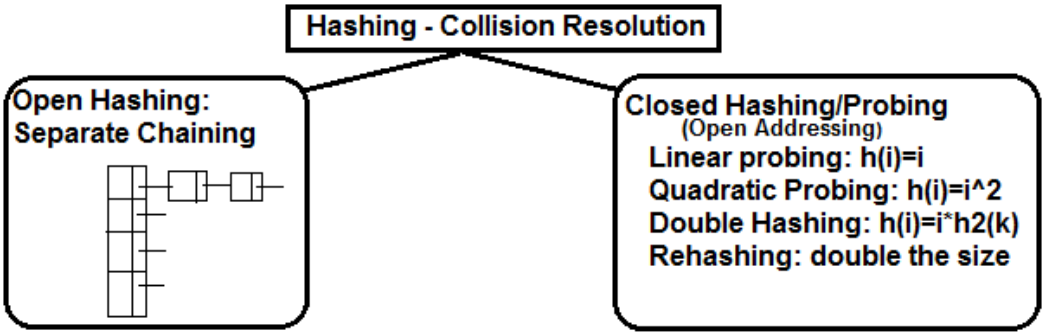
int hash(int key) { return key%size; }

Collision:

Hash function performs poor when multiple keys are mapped to the same location. This is collision. For collision resolution may techniques are suggested. The techniques also suffer problems of primary & secondary clustering.

It is suggested that size of the hash table should be a prime number to reduce collision & clustering.

Classification of Hashing collision resolution techniques.



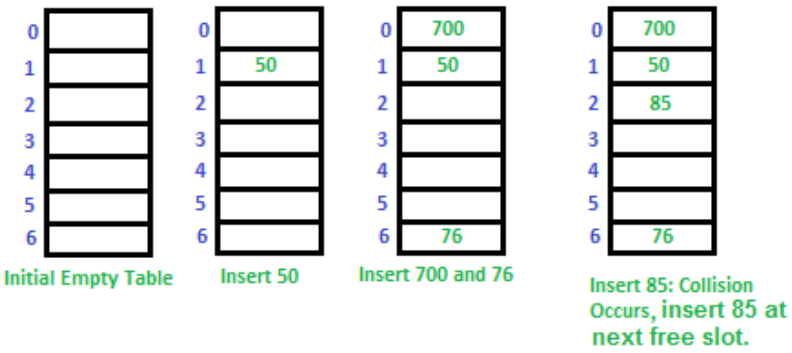
Linear Probing:

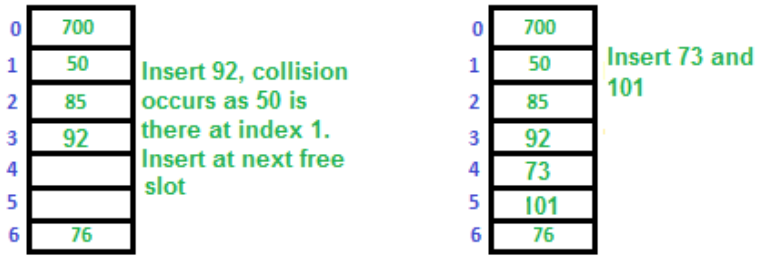
Linear Probing is basic & popular collision resolution technique in hashing when collision occurs. In linear probing, we linearly probe for next slot when collision occurs for the calculated key by the hash function.

If location loc is full, then we try loc+1. Location is coming to beginning if crossing size.

```
/*Hashing - Linear Probing */
#include <stdio.h>
int size;
long hash(int key) {  return (key%size); }
void insert(int hashtable[])
{
     int x,y,key;       long loc,temp;
      for( x=0;x<size;x++)
        {
             printf("\nEnter element %d : ",x+1);
             scanf("%d",&key);
             loc=hash(key);
             while(hashtable[loc]!='\0')
             {
                  loc++;
                  if(loc==size) loc=0;
              }
             hashtable[loc]=key;
        }
}
```

Example:





Q10b. Explain in detail about static & dynamic hashing (6marks)

Ans.

Hashing is an efficient technique to directly search the location of desired data on the disk without using index structure. Data is stored at the data blocks whose address is generated by using hash function. The memory location where these records are stored is called as data block or data bucket.

Static hashing:

- Single hash function h(k) on key k
- Desirable properties of a hash function
    - Uniform: Total domain of keys is distributed uniformly over the range
    - Random: Hash values should be distributed uniformly irrespective of distribution of keys O(1) search
- Example of hash functions: h(k) = k MOD m (size of hash table)
- Collision resolution
- Chaining
    - Load factor
    - Primary pages and overflow pages (or buckets)
    - Search time more for overflow buckets
- Open addressing
    - Linear probing
    - Quadratic probing
    - Double hashing

Problems of static hashing

- Fixed size of hash table due to fixed hash function
- **Primary/secondary Clustering (We should keep size of hash table a prime number to reduce clustering)**
- May require rehashing of all keys when chains or overflow buckets are full

Dynamic hashing:

In this hashing scheme the set of keys can be varied & the address space is allocated dynamically.
If a file F is collection of record a record R is key+data stored in pages (buckets) then space utilization:

Number of records/(Number of pages*Page capacity)

- Hash function modified dynamically as number of records grow
- Needs to maintain determinism
- Extendible hashing
- Linear hashing

Again: Dynamic hashing Hash function modified dynamically as number of records grow Needs to maintain determinism Extendible hashing and Linear hashing

- Organize overflow buckets as binary trees
- m binary trees for m primary pages
- $h_0(k)$ produces index of primary page
- Particular access structure for binary trees
- Family of functions : $g(k) = \{ h_1(k),....h_i(k),....\}$
- Each $h_i(k)$ produces a bit
- At level i, $h_i(k)$ is 0, take left branch, otherwise right branch Example: bit representation

Q10c. Briefly Explain basic operations that can be performed on a file. Explain Indexed sequential file organization. (7marks)

Ans:

A computer file is collection of data (text, program, image, animation or video) stored on permanent storage device. A computer file has a name. A file belongs to a type. In general type is recognized by its extension name.

In terms of operating system files can be classified as ordinary file, directory file, special file and FIFO file.

Basic operations can be performed on a file:

- **Read operation:** To read the content stored in a file
- **Write Operation:** To modify, overwrite the contents or to create a new file
- **Rename operation:** To change the name of a file
- **Copy operation:** To copy a file from one location to another
- **Move operation:** To move a file to another folder/directory or drive
- **Delete Operation:** To remove a file from storage
- **Run operation:** To open a file/ To execute a program file
- **Link operation:** To create links of files (symbolic link or hard link)

Indexed-sequential file organization:

ISAM is almost similar to sequential method only that, an index is used to enable the computer to locate individual records on the storage media. For example, on a **magnetic drum**, records are stored sequential on the tracks. However, each record is assigned an index that can be used to access it directly.

When there is need to access the records by indices and in indices to move sequentially to search the specific record, such organization of file is known as Indexed Sequential file Organization.

Example:

- Words are stored in a larger dictionary in sequential manner

- Indices are there for thumb tabs
- We do not search the word sequentially from the beginning
- We first go to index thumb tab then we search sequentially for the word
- Index part (level 1) and data part (level 2) are separately stored
- Index part stores the address of beginning of the data
- When new records are inserted in data file, the sequence is preserved and index is updated accordingly.
- Indices may be static and dynamic
- If data file changes due to insertions and deletions, static index may change but structure of the index will not change
- In case of dynamic index if data file changes due to insertion and deletion, structure of index may change.